

Step Skipping Acceleration Techniques on Recursive Logical Circuits. Practical implementations on FPGA.*

Géry J. A. Bioul, Martín Vázquez

Universidad Fasta, Facultad de Ingeniería
Gascón 3145 – B7600FNK – Mar del Plata. Argentina.
www.ufasta.edu.ar

Abstract

This paper presents step skipping acceleration techniques for a class of convergence algorithms computing arithmetic functions. In particular, an extension of the fast adder carry-skip procedure [1, 2] is carried out for special purpose cellular array circuits implementing iterative logical functions for which some propagating information may be fruitfully computed ahead of the current step output computation. This information is thus carried to the next stage, accelerating the overall calculation.

An application is given for the 2's complement sign changing circuit, then for the step-skipping acceleration circuits used in the implementation of the $\ln(x)$ convergence algorithm. FPGA implementations on Xilinx Virtex IV [6] have been achieved with comparative analysis of 32- to 512-bit computing algorithms.

Keywords: FPGA, $\ln(x)$, convergence method, step skipping, carry-skip adder, Xilinx Virtex 4.

* This project is supported by FASTA University, Faculty of Engineering, B7600FNK Mar del Plata, Argentina. Thanks are due to the INTIA-INCA labs. at UNCPBA – Tandil (Bs. As.) for technical support.

1. INTRODUCTION

The classical implementations of iterative functions such as ripple-carry adders, logarithmic, exponential or trigonometric functions are generally made up from sequential circuits (time iteration) or cascade circuits (space iteration). In both cases, the time complexity increases with the bit size of the operands. In the cascade case, some techniques have been presented to accelerate the processes. Most often the time saving has to be paid by some additional hardware cost. Such is the case for the carry look-ahead or carry-skip adders [1, 4].

In the carry-skip adder the cost of additional hardware is negligible in front of the notable time saving. The carry-skip technique is one of the key ideas in the step-skipping implementations reviewed hereafter.

A short survey of carry-skip adder is first presented to make this paper self consistent. A straight application to the binary 2's complement sign changing device is then considered with comparative time and computational complexities. Then applications to step acceleration and step skipping process for some arithmetic convergence algorithms are given with complexity figures and FPGA implementations and testing.

2. THE CARRY-SKIP ADDER REVISITED

2.1 Carry chain adder

The structure of an n -bit binary adder with separate carry calculation is shown in figure 1. The $G-P$ (*generate - propagate*) cell calculates

$$g(i) = x(i) \wedge y(i) , \text{ and } p(i) = x(i) \oplus y(i) ;$$

the $Cy.Ch.$ (*carry chain*) cell computes $q(i+1)$:

$$\mathbf{if } p(i) = 1 \mathbf{ then } q(i+1) := q(i); \mathbf{ else } q(i+1) := g(i); \mathbf{ end if};$$

and the *mod 2 sum* cell calculates $z(i) = (x(i) \oplus y(i) \oplus q(i)) = p(i) \oplus q(i)$.

Let C_{GP} and T_{GP} , $C_{Cy.Ch.}$ and $T_{Cy.Ch.}$, C_{sum} and T_{sum} , be the cost and the computation time of a $G-P$ cell, $Cy.Ch.$ cell and *mod 2 sum* cell, respectively. The cost and computation time of an n -bit carry chain adder are equal to

$$C_{carry-chain-adder}(n) = n.(C_{GP} + C_{Cy.Ch.} + C_{sum}), \tag{1}$$

$$T_{carry-chain-adder}(n) = T_{GP} + (n-1).T_{Cy.Ch.} + T_{sum}.$$

The critical path is shaded in figure 1. In the case of a VLSI implementation based on the *Manchester carry chain* configuration, the value of $T_{Cy.Ch.}$ is equal to the drain-source propagation time of one n -MOS transistor.

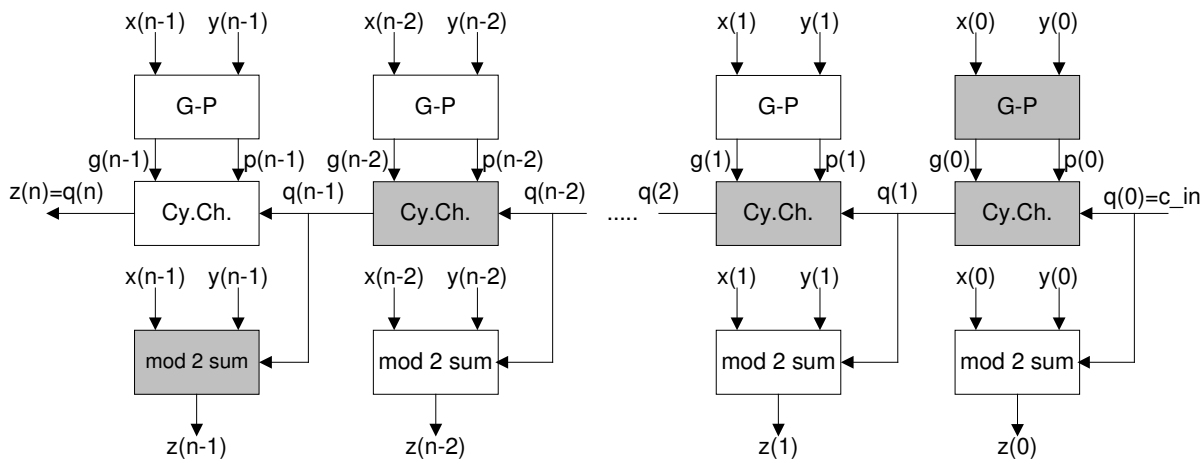


Figure 1: Carry chain adder

Comment 2.1

According to the carry computation algorithm the carry chain cell should compute the switching function $p(i).q(i) \vee \text{not}(p(i)).g(i)$. Nevertheless, as $p(i).g(i) = 0$, the preceding expression is equivalent to $p(i).q(i) \vee g(i)$. Furthermore, if the latest expression is used for defining the carry chain cell function, then $p(i)$ can be substituted by any function $p'(i)$ such that $p(i) \leq p'(i) \leq p(i) \vee g(i)$.

2.2 Carry-skip adder

Consider a group of s successive cells within a carry chain (figure 2). If all propagate functions $p(i.s), p(i.s+1), \dots, p(i.s+s-1)$ within the group are equal to 1 then $q(i.s+s) = q(i.s)$, and the carry $q(i.s)$ is said to be propagated through the group. In the contrary case, there is at least one cell, say number $i.s+j$, such that $p(i.s+j) = 0$ so that $q(i.s+j+1) = g(i.s+j)$. Assume that cell number $i.s+j$ is the last one such that $p(i.s+j) = 0$ and $p(i.s+j+1) = \dots = p(i.s+s-1) = 1$. Then $q(i.s+s) = g(i.s+j)$, and the carry $q(i.s+s)$ is said to be locally computed within the group.

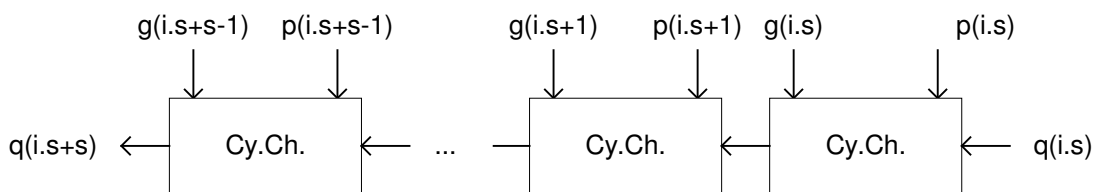


Figure 2: s -bit carry chain

An n -bit *carry-skip* carry chain is made up of n/s s -bit carry chains interconnected through 2-to-1 multiplexers (figures 3 and 4). Besides the generate and propagate functions, the generalized propagate functions $P(i.s+s-1:i.s) = p(i.s+s-1). \dots . p(i.s+1).p(i.s)$ must also be computed.

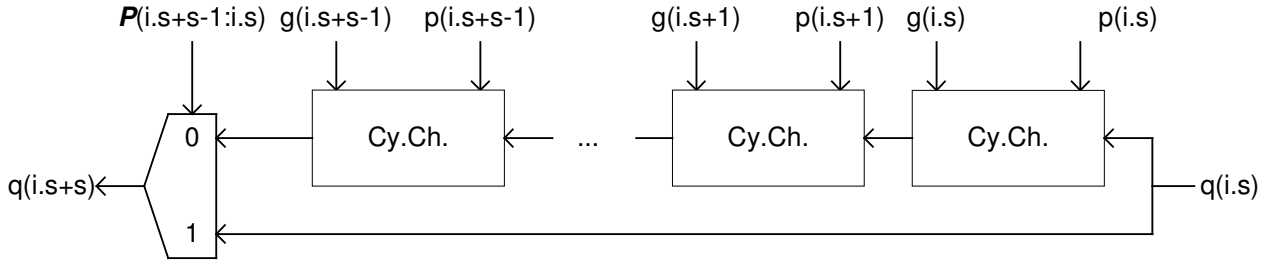


Figure 3: Carry chain: s -bit group

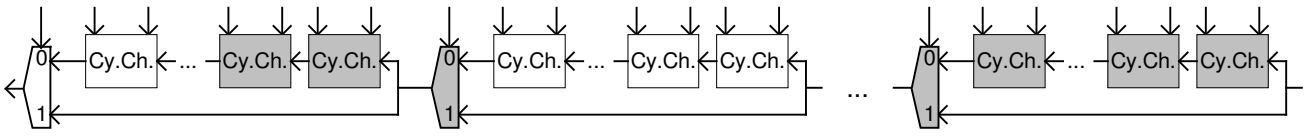


Figure 4: Carry-skip carry chain

The structure of a carry-skip adder is shown in figure 5. It is made up of n G - P cells, n $Cy.Ch.$ cells, $n \bmod B$ sum cells, n/s 2-to-1 multiplexers and n/s s -input AND gates (or any equivalent circuit) for computing $p(i.s+s-1:i.s)$. Its cost and computation time are equal to:

$$C_{carry-skip-adder}(n,s) = n.(C_{GP} + C_{Cy.Ch.} + C_{sum}) + (n/s).(C_{mux2-1} + C_{and(s)}), \quad (2)$$

$$T_{carry-skip-adder}(n,s) = T_{GP} + s.T_{Cy.Ch.} + (n/s - 1).T_{mux2-1} + (s - 1).T_{Cy.Ch.} + T_{sum}.$$

The critical path of the carry chain is shaded in figure 4. It has been assumed that $s.T_{Cy.Ch.} > T_{and(s)}$ and $T_{sum} > T_{Cy.Ch.} + T_{mux2-1}$, so that in the first group $p(s-1:0)$ is computed in parallel with the multiplexer inputs, and in the last group the critical path is from $q(n-s)$ to $z(n-1)$ through $s-1$ $Cy.Ch.$ cells and one $mod B$ sum cell.

Another interesting time is the delay $T_{carry}(n,s)$ from $q(0)$ to $q(n)$ assuming that all propagate, generate and generalized propagate functions have already been calculated:

$$T_{carry}(n,s) = s.T_{Cy.Ch.} + (n/s).T_{mux2-1}. \quad (3)$$

Comments 2.2

- 1 For great values of n and s the computation time (2) is roughly equal to $(n/s).T_{mux2-1} + 2.s.T_{Cy.Ch.}$. It must be compared with (1), that is to say (approximately) $n.T_{Cy.Ch.}$. The computation time reduction is due to the fact that the locally generated carries are calculated in parallel.
- 2 The s rightmost $Cy.Ch.$ cells of figure 4 belong to the critical path, so that the first multiplexer should be deleted, unless the corresponding adder is used as a building block for larger adders.

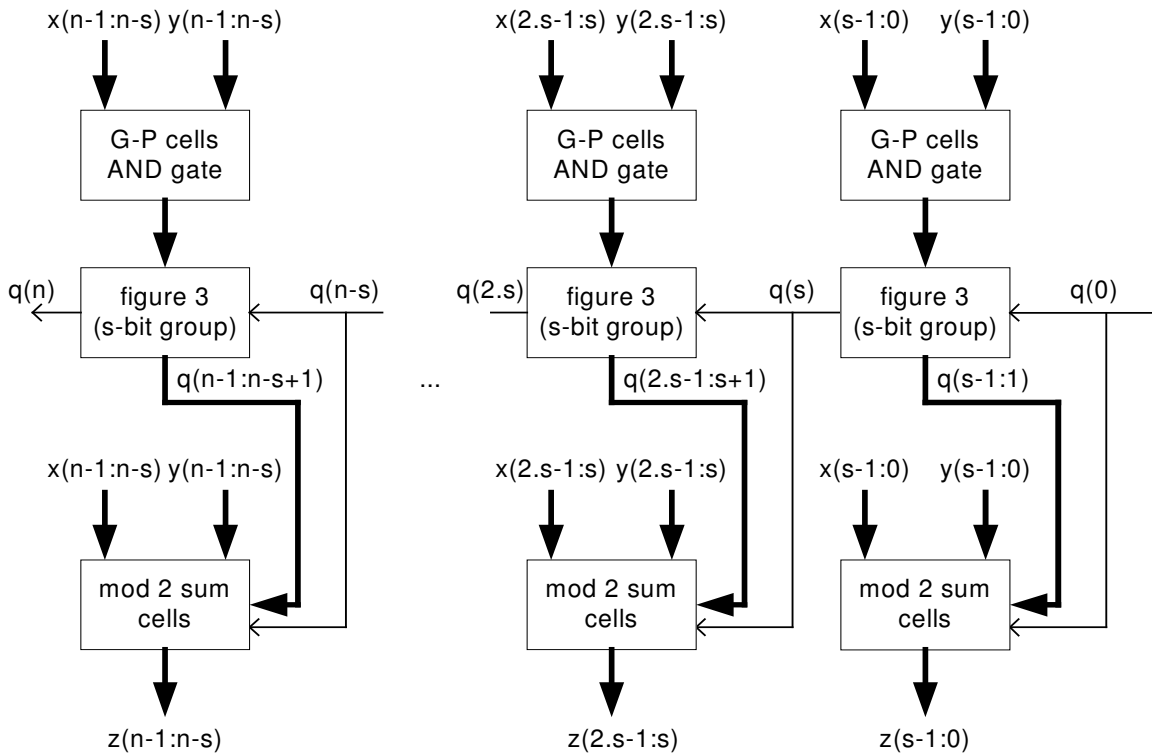


Figure 5: Carry-skip adder structure

Comments 2.2

- 3 For great values of n and s the computation time (2) is roughly equal to $(n/s) \cdot T_{mux2-1} + 2 \cdot s \cdot T_{Cy.Ch.}$. It must be compared with (1), that is to say (approximately) $n \cdot T_{Cy.Ch.}$. The computation time reduction is due to the fact that the locally generated carries are calculated in parallel.
- 4 The s rightmost $Cy.Ch.$ cells of figure 4 belong to the critical path, so that the first multiplexer should be deleted, unless the corresponding adder is used as a building block for larger adders.

3. TWO'S COMPLEMENT SIGN CHANGE CASCADE CIRCUIT

Changing sign in 1's complement expressed binary numbers is carried out through bitwise complementing. This operation is readily achieved by a circuit made up of complementing gates (e.g. 2-input XOR) acting in parallel. In 2's complement notation, changing sign can be basically performed by adding "1" to the bitwise complemented vector. An equivalent method comes from complementing all bits at the left of the rightmost "1". In both cases the circuit implementation does not appear to be parallel. The circuit presented in figure 6 is a possible sign change circuit implementation for changing the sign of a 2's complement binary vector.

In the same way as for the carry-skip technique for adders, the propagated signal can be accelerated by a careful partition of the circuit into n/s slices of s modules, as it appears in figure 7 where the enable sub circuit has not been represented for clarity. Figure 7 displays an s -module stage.

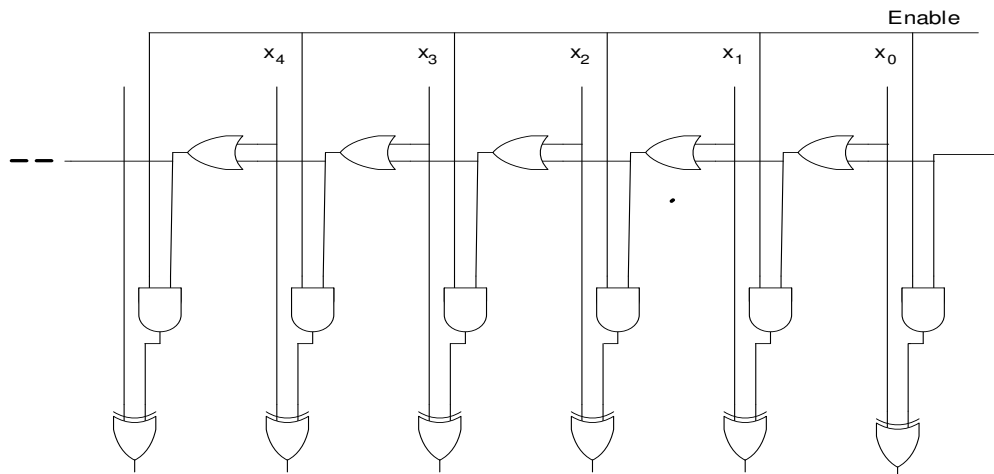


Figure 6: Sign change circuit in 2's complement representation

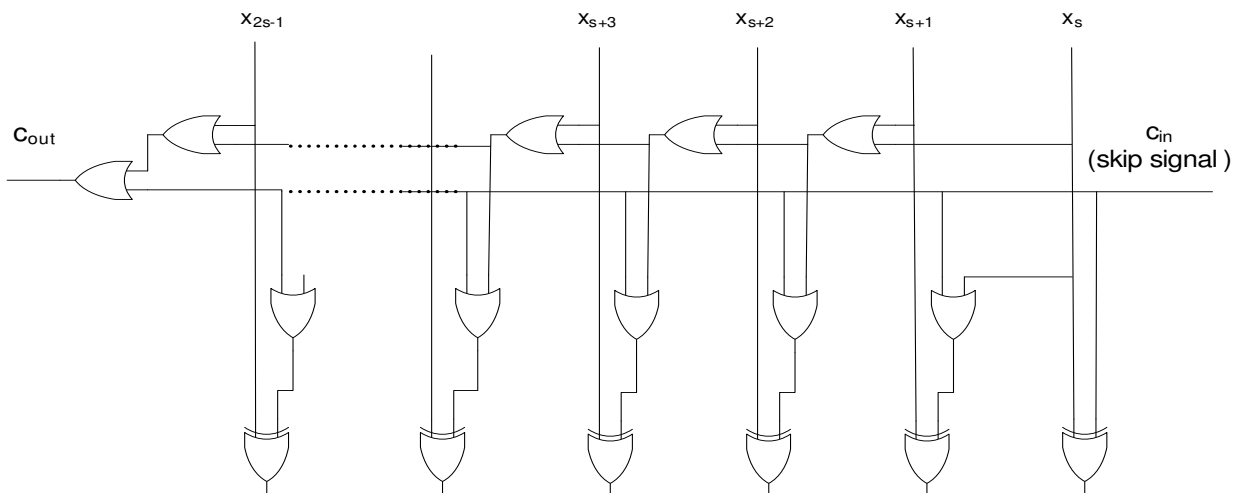


Figure 7: Accelerated sign change circuit in 2's complement representation

The skip signal input c_{in} , whenever valued 1, readily feeds the XOR output gates to complete the parallel bitwise complementation of all variables of the slice. Moreover this signal is readily conveyed to next stage through an OR gate. The function of this OR gate is somewhat comparable to that of the multiplexers of figure 4.

According to the technology at hand, the Boolean function of the circuit represented at figure 6 may be synthesized in a number of ways. Let us assume that optimized elementary modules with t -input binary variables are given. Each module performs the same function as that of figure 6. One deals with $(t+1)$ -input, $(t+1)$ -output components. The complete n -input circuit is thus a set of n/t modules, interconnected according to figure 8. In turn this set may be sliced into $(n/t)/s$ slices of s modules each. Within this scheme a carry-skip like accelerated circuit may be designed (figure 9).

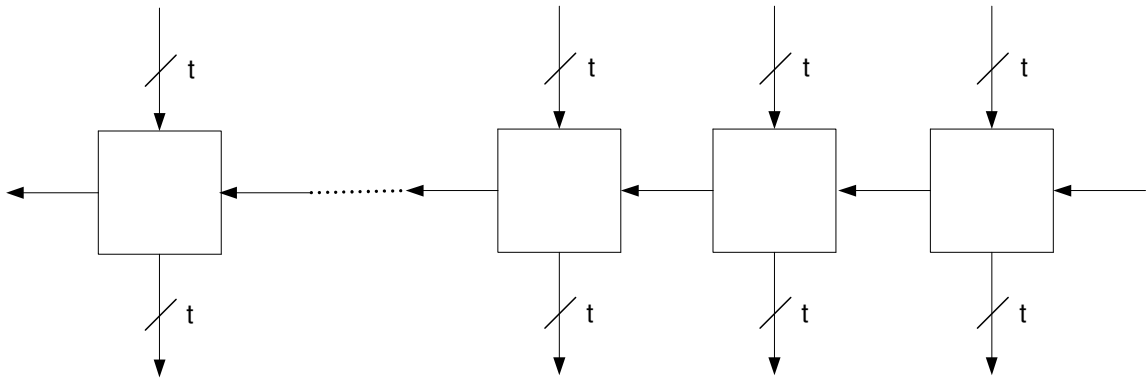


Figure 8: n/t modules circuit

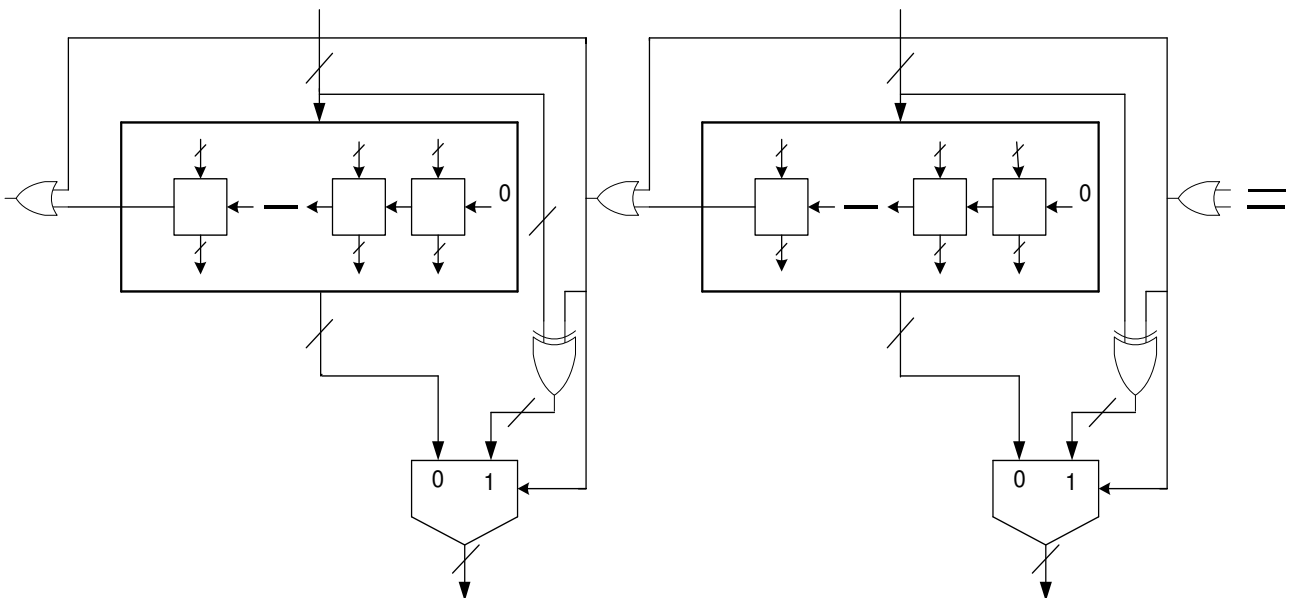


Figure 9: Two consecutive s -module slices out of the $(n/t)/s$ -slice circuit

4. $LN(X)$ CONVERGENCE ALGORITHM.

This section reviews the approximation algorithm to compute $\ln(x)$ using the multiplicative normalization method [1, 4, 5].

One defines

$$c(i) = 1 + a_i \cdot 2^{-i}, \quad a_i \in \{-1, 0, 1\} \quad (4)$$

as the *multiplicative normalizing function*, where a_i is selected in such a way that the sequence

$$x(i+1) = x(i) \cdot c(i) \text{ (auxiliary sequence)} \quad x(i) \in B(2^n) \quad (5)$$

converges towards 1. Then, the sequence

$$y(i+1) = y(i) - \ln c(i) \quad (6)$$

can be set to converge toward the result $\ln(x)$. If $y(0)$ and $x(0)$ are respectively set to 0 and to the argument x , and assuming $x(p) \cong 1$, one can demonstrate that

$$x(p) = x \cdot \prod_i c(i) \cong 1 \rightarrow 1/x \cong \prod_i c(i); \quad y(p) = y - \sum_i \ln c(i) = -\ln \prod_i c(i) = \ln(x). \quad (7)$$

To make the convergence of (5) possible, the argument x needs to be in a range

$$0.42 \leq x \leq 3.45. \quad (8)$$

This means that the argument x could need to be pre-scaled to fall in the range (8). An argument x in the range $[1, 2[$ (such as e.g. a floating-point mantissa) fits perfectly; otherwise use a straightforward pre-scaling operation that replaces x by x' such that $x = x' \cdot 2^s$ (x' in $[1, 2[$); the algorithm computes $\ln(x')$, then a final additive correction of $s \cdot \ln(2)$ is completed. Observe that the lower bound of (5) can be lowered to 0.21, as $(1+2^0)$ can be accepted as a first normalizing factor for computing $x(1)$.

In practical implementations of this algorithm, look-up tables are used to read out the successive values of $\ln(1 \pm 2^i)$, needed to compute $y(i+1)$ of (6). For x in $[\frac{1}{2}, 2[$, a_i can be selected according to the following rules:

$$a_0 = 0, \quad (9)$$

$$\text{if } x(i) > 1, \quad a_i = -x_i(i), \quad i \geq 1 \quad (10)$$

$$\text{if } x(i) < 1, \quad a_i = +x_i(i) \cdot \text{not}(x_{i-1}(i)), \quad i \geq 1 \quad (11)$$

Comments 4

- 1 The selection (9) is justified by the fact that a decision about multiplying by $a_i \cdot 2^i + 1$ (1) cannot be made before knowing the next bit. Actually, considering bit x_0 only (either 1 or 0) one cannot know whether the sequence $x(i)$ is already 1 (end of convergence process) or not.
- 2 When $x(i) > 1$, the strategy described by (10) consists in detecting the first non-zero bit of $x(i)$ then multiplying by $(-2^i + 1)$. When $x(i) > 1$, it can be shown [1] that, at step i , bits $x_{-k > -i}(i)$ are all zero's.
- 3 When $x(i) < 1$, the strategy described by (11) consists in detecting the last non-zero bit of $x(i)$ then multiplying by $(2^i + 1)$. When $x(i) \leq 1$, it can be shown [1] that, at step i , all bits $x_{-k > -i}(i)$ are one.

5. ALGORITHMS

5.1. Algorithm 1 - Logarithm computation by multiplicative normalization

The argument x is in $[\frac{1}{2}, 2[$: $x = x(0) \cdot x(1) \cdot x(2) \dots x(n)$. Let $xx(i,j)$ be the component j of $xx(i) = xx(i,0) \cdot xx(i,1) \cdot xx(i,2) \dots xx(i,n)$. Let $lut(i) = \ln(1 + a(i) \cdot 2^i)$ read from the table.

$a(0) := 0; c(0) := 1; xx(1) := x; yy(1) := 0;$

for i **in** $1 .. p-1$ **loop**

if $xx(i) = 1$ **then exit; end if;**


```

if  $xx(i) > 1$  then  $a(i) := -xx(i,i)$  else  $a(i) := xx(i,i) * \text{not}(xx(i,i+1))$ ; end if;
 $c(i) := 1 + a(i) * 2^{**}(-i)$ ;  $xx(i+1) := xx(i) * c(i)$ ;  $yy(i+1) := yy(i) - lut(i)$ ;
end loop;

```

5.2. Algorithm 2 - Logarithm computation by multiplicative, one-shift and add, normalization

The argument x is in $[\frac{1}{2}, 2[$: $x = x(0).x(1)x(2) \dots x(n)$. Let $xx(i,j)$ be the component j of $xx(i) = xx(i,0).xx(i,1)xx(i,2) \dots xx(i,n)$. Let $lut(i) = \ln(1+a(i).2^i)$ read from the table.

```

 $a(0) := 0$ ;  $xx(1) := x$ ;  $yy(1) := 0$ ;
for  $i$  in  $1 .. p-1$  loop
  if  $xx(i) = 1$  then exit; end if;
  if  $xx(i) > 1$  then  $a(i) := -xx(i,i)$  else  $a(i) := xx(i,i) * \text{not}(xx(i,i+1))$ ; end if;
   $xx(i+1) := xx(i) + a(i) * xx(i) * 2^{**}(-i)$ ;  $yy(i+1) := yy(i) - lut(i)$ ;
end loop;

```

Comments 5

1. Algorithm 5.1 is basically identical to algorithm 5.2. It just replaces the multiplication by $1+a(i) \cdot 2^{-i}$ by a shift and add procedure, that is, whenever $a(i)=1$, $xx(i+1)$ is computed as $xx(i) + xx(i)$ i -shifted on the right. This eventually allows a more economical hardware implementation: changing the multiplier by a shifter/adder device.
2. At the implementation level, acceleration procedures have been proposed [3] avoiding trivial steps whenever $a(i)=0$. For this sake a procedure has been implemented to detect sequences of zero's (resp. Sequences of one's). Fast detection circuits of this type are presented further.

6. FULL HARDWARE BIT-SEQUENCE DETECTION CIRCUITS

6.1 0-sequences detector

The circuit presented at figure 10 identifies the position of the first 1 appearing at the right of a 0-sequence. This search is only needed when the argument x is greater than 1, so $x_0=1$. Then the position of the first 1 is detected to set the step number i . Output k is set to 1 whenever x_k is the first 1 from left to right; all other output are at 0. The next step code i is then readily available through a coding circuit as shown at figure 11.

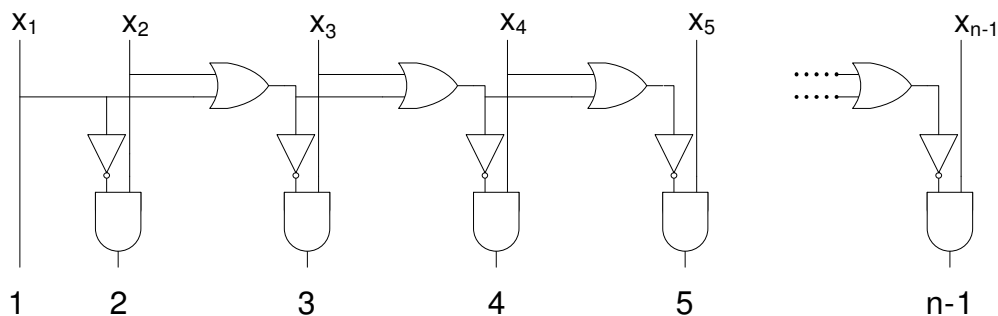


Figure 10: End of 0-sequence detector

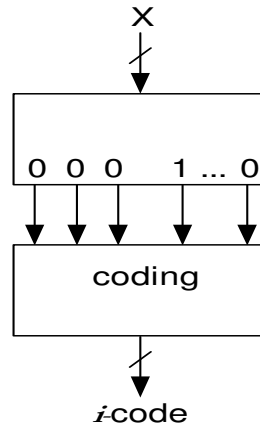


Figure 11: Next step coding

6.2 1-sequences detector

The circuit presented at figure 12 identifies the position of the first 0 appearing at the right of a 1-sequence. This search is only needed when the argument x is smaller than 1, i.e. $x_0=0$, moreover, by the way algorithm is carried out, one may assume that $x_1=1$. The position of the first 0 is then detected to set the step number i . Output i is set to 1 whenever x_{i+1} is the first 0 from left to right; all other output are at 0. The next step code i is then readily available through a coding circuit similar to that of figure 11.

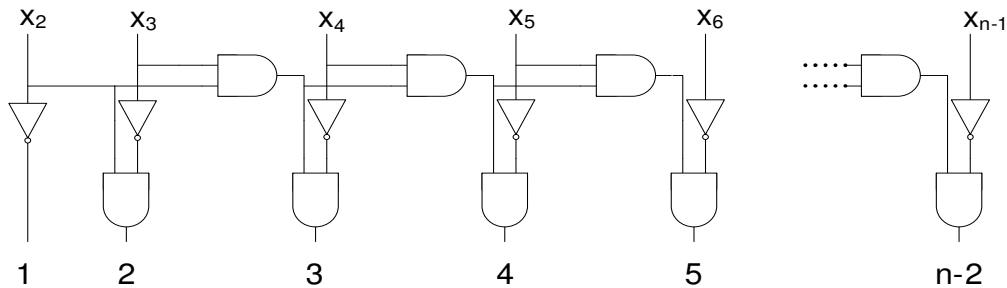


Figure 12: End of 1-sequence detector

6.3 Accelerating sequence detector circuits

Both circuits of figures 10 and 12 may be accelerated in the same way as the sign-change circuits of figure 9. Figure 13 shows how the circuits of figure 10 can be structured in module-chain sliced in such a way that as soon as a 1 is detected the following outputs are set to zero within a reduced delay. Respectively, figure 14 stands for a similar design for accelerating the propagation of the signal corresponding to the first detected 0.

The circuit of figure 13 (resp. figure 14) is made up of n/t modules, $(n/t)/s$ 2-input OR gates (resp. $(n/t)/s$ 2-input AND gates) and n 2-input multiplexers. Their costs and computation times are equal to:

$$C_{0-seq.detector}(n,t,s) = n/t \cdot (C_{mod.}) + (n/t)/s \cdot (C_{OR2-1}) + n \cdot C_{mux2-1}, \quad (12)$$

$$T_{0-seq.detector}(n,t,s) = s \cdot T_{mod} + (n/t)/s \cdot (T_{OR2-1}) + T_{mux2-1}$$

$$C_{1-seq.detector}(n,t,s) = n/t \cdot (C_{mod.}) + (n/t)/s \cdot (C_{AND2-1}) + n \cdot C_{mux2-1}, \quad (13)$$

$$T_{1-seq.detector}(n,t,s) = s \cdot T_{mod} + (n/t)/s \cdot (T_{AND2-1}) + T_{mux2-1}$$

The cost of the decoder (figure 11) has to be considered in both cases.

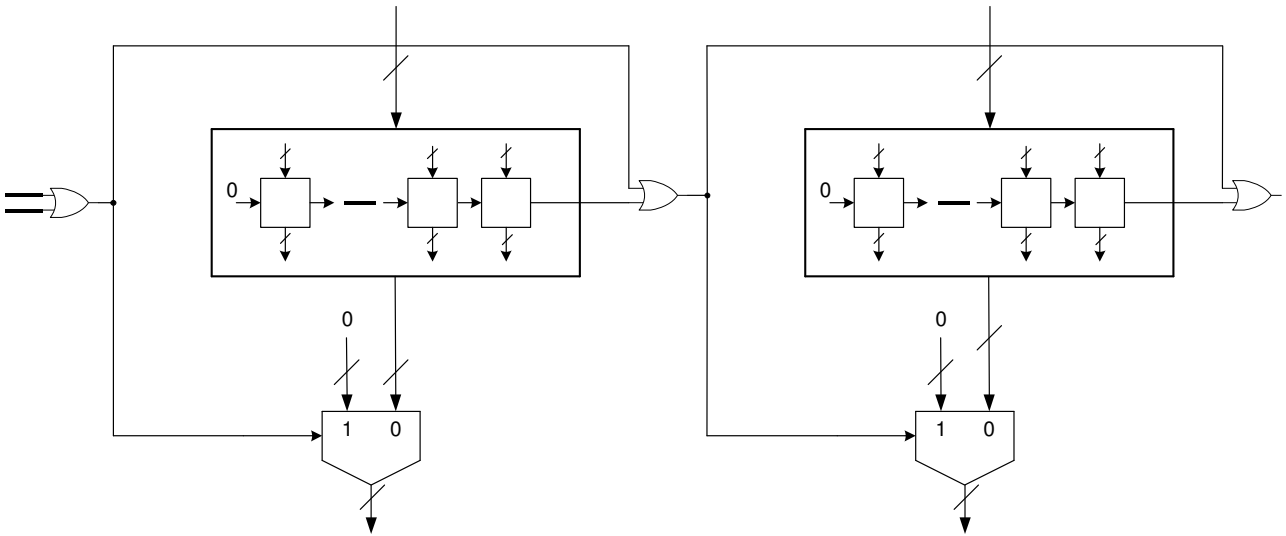


Figure 13: Accelerated end of 0-sequence detector

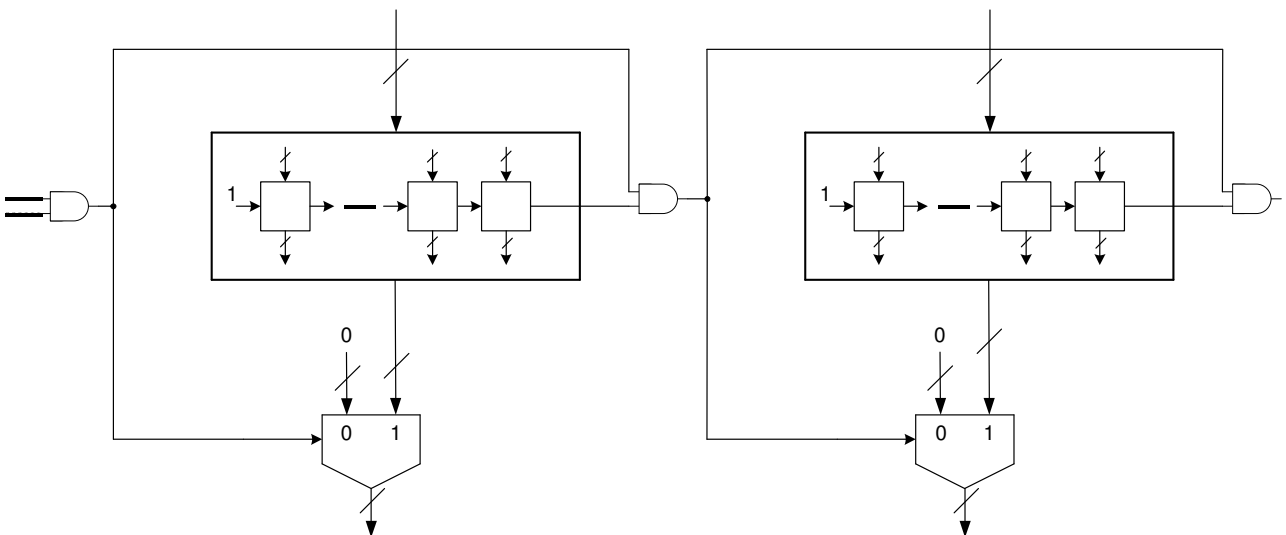


Figure 14: Accelerated end of 1-sequence detector

7. EXPERIMENTAL RESULTS

End of 0-sequence and end of 1-sequence circuits have been implemented on FPGA Virtex 4 Xilinx family (xc4vlx60-12ff1148, con 26624 *slices*) [6]. Comparative analysis has been carried on considering inputs sizes up to 512 bits. Synthesis has been performed within XST (Xilinx Synthesis Technology) [7] while physical implementation used Xilinx ISE (Integrated System Environment) versión 9.2.04i [8].

Table 1 displays the time and area performances for end of 0-sequence detection (figure 10) and end of 1-sequence detection (figure 12) for operands with up to 256-bit length (N). Table 2 displays the same features for the accelerated algorithms (circuits of figures 13 and 14 respectively) for operands with up to 256-bit length considering several block sizes (K). For each input size N , the best time performance appears shaded table 2.

Tabla 1. Computation time (ns) and Area (# of *slices*) for end of 0-sequence (resp. end of 1-sequence) implementations (figures 10 and 12).

| N | <i>End of 0-sequence</i> | | <i>End of 1-sequence</i> | |
|-----|--------------------------|-------------|--------------------------|-------------|
| | <i>Time</i> | <i>Area</i> | <i>Time</i> | <i>Area</i> |
| 16 | 13.78 | 12 | 14.91 | 11 |
| 32 | 13.74 | 27 | 16.65 | 25 |
| 64 | 16.71 | 56 | 28.80 | 50 |
| 128 | 24.43 | 116 | 38.68 | 103 |
| 256 | 38.08 | 237 | 50.31 | 224 |

Tabla 2. Computation time (ns) and Area (# of *slices*) for end of 0-sequence (resp. end of 1-sequence) accelerated algorithms implementations (figures 13 and 14).

| N | K | <i>End of 0-sequence</i> | | <i>End of 1-sequence</i> | |
|-----|-----|--------------------------|-------------|--------------------------|-------------|
| | | <i>Time</i> | <i>Area</i> | <i>Time</i> | <i>Area</i> |
| 16 | 4 | 13.78 | 12 | 13.41 | 11 |
| 32 | 4 | 12.88 | 27 | 15.59 | 25 |
| 32 | 8 | 15.59 | 24 | 16.22 | 24 |
| 64 | 8 | 17.02 | 59 | 18.98 | 48 |
| 64 | 16 | 15.69 | 53 | 17.39 | 54 |
| 128 | 8 | 25.79 | 121 | 30.06 | 111 |
| 128 | 16 | 17.91 | 110 | 18.96 | 110 |
| 128 | 32 | 16.27 | 127 | 20.31 | 112 |
| 256 | 8 | 39.70 | 239 | 40.68 | 234 |
| 256 | 16 | 24.94 | 226 | 22.38 | 229 |
| 256 | 32 | 21.07 | 264 | 27.04 | 223 |
| 256 | 64 | 24.03 | 252 | 30.54 | 219 |

Table 3 compares the computation times between algorithms, selecting, for the accelerated versions, the block size corresponding to the shortest time. One observes that the time saving increases with the operand size, reaching more than 50 % saving in some cases. Actually, using special purpose ASIC implementations, it is expected that the savings would be significantly better.

Table 3. Time comparisons between straight and accelerated algorithms.

| <i>N</i> | <i>End of 0-sequence</i> | | | <i>End of 1-sequence</i> | | |
|------------|--------------------------|--------------------|---------------|--------------------------|--------------------|---------------|
| | <i>Straight</i> | <i>Accelerated</i> | <i>Saving</i> | <i>Straight</i> | <i>Accelerated</i> | <i>Saving</i> |
| 16 | 13.78 | 13.78 | 0 % | 14.91 | 13.41 | 10 % |
| 32 | 13.74 | 12.78 | 7 % | 16.65 | 15.59 | 6 % |
| 64 | 16.71 | 15.69 | 6 % | 28.80 | 17.39 | 40% |
| 128 | 24.43 | 16.27 | 33 % | 38.68 | 18.96 | 51 % |
| 256 | 38.08 | 21.07 | 45 % | 50.31 | 22.38 | 55 % |

Table 4 deals with the areas requirements. For the accelerated algorithms the areas requirements correspond to the circuits with the minimum time consumption. One observes that the area requirements are not much different between algorithms. The greatest difference is 11 % for end of 0-sequence detection with 256-bit operands, or 8% for end of 1-sequence detection with 64-bit operands.

Table 4. Area requirements comparison.

| <i>N</i> | <i>End of 0-sequence</i> | | | <i>End of 1-sequence</i> | | |
|------------|--------------------------|--------------------|-------------------|--------------------------|--------------------|-------------------|
| | <i>Straight</i> | <i>Accelerated</i> | <i>Difference</i> | <i>Straight</i> | <i>Accelerated</i> | <i>Difference</i> |
| 16 | 12 | 12 | 0 % | 11 | 11 | 0 % |
| 32 | 27 | 27 | 0 % | 25 | 25 | 0 % |
| 64 | 56 | 53 | -5 % | 50 | 54 | 8 % |
| 128 | 116 | 127 | 9 % | 103 | 110 | 7 % |
| 256 | 237 | 264 | 11 % | 224 | 229 | 2 % |

A common technique to evaluate FPGA implementations performances consists of coping with input/output transfers, inserting I/O registers in the design. This allows evaluating delays in between input/output registers. Actually input/output delays in Xilinx family FPGA are not negligible at all. Tables 5 to 8 present the algorithms performances within the above mentioned measuring conditions. Table 5 displays results to be compared with those of table 1. On one hand one observes that computation times are significantly better. Actually Xilinx synthesizer optimizes the best when detecting registers leading to major clock frequencies. On the other hand, the area consumption increases, as the number of slices involves is drastically incremented with respect to the figures displayed in table 1. Although the implementation of registers partially justifies the slice consumption, the leading point comes from optimization techniques (such as duplicating logic resources) carried on by the synthesizer.

Table 5. Computation time (ns) and Area (# of *slices*) for end of 0-sequence (resp. end of 1-sequence) implementations (figures 10 and 12), with I/O registers.

| <i>N</i> | <i>End of 0-sequence</i> | | <i>End of 1-sequence</i> | |
|------------|--------------------------|-------------|--------------------------|-------------|
| | <i>Time</i> | <i>Area</i> | <i>Time</i> | <i>Area</i> |
| 16 | 1.81 | 23 | 2.02 | 23 |
| 32 | 2.48 | 53 | 2.52 | 48 |
| 64 | 2.91 | 118 | 3.30 | 111 |
| 128 | 5.02 | 268 | 5.27 | 273 |
| 256 | 6.19 | 531 | 6.73 | 504 |
| 512 | 8.11 | 1145 | 8.76 | 1088 |

The same fact appears in table 6, to be compared with table 2. As in table 2 the best time figures appear shaded.

Table 6. Computation time (ns) and Area (# of *slices*) for end of 0-sequence (resp. end of 1-sequence) accelerated algorithms implementations (figures 13 and 14), with I/O registers.

| <i>N</i> | <i>K</i> | <i>End of 0-sequence</i> | | <i>End of 1-sequence</i> | |
|----------|----------|--------------------------|-------------|--------------------------|-------------|
| | | <i>Time</i> | <i>Area</i> | <i>Time</i> | <i>Area</i> |
| 16 | 4 | 1.92 | 23 | 1.86 | 23 |
| 32 | 4 | 2.32 | 53 | 2.51 | 47 |
| 32 | 8 | 2.60 | 49 | 2.52 | 48 |
| 64 | 8 | 3.01 | 109 | 3.36 | 102 |
| 64 | 16 | 2.96 | 97 | 3.23 | 101 |
| 128 | 8 | 5.24 | 259 | 6.34 | 231 |
| 128 | 16 | 5.66 | 231 | 5.87 | 255 |
| 128 | 32 | 5.95 | 240 | 4.88 | 234 |
| 256 | 8 | 7.34 | 522 | 6.03 | 515 |
| 256 | 16 | 7.98 | 478 | 7.31 | 460 |
| 256 | 32 | 7.01 | 476 | 6.99 | 471 |
| 256 | 64 | 6.54 | 486 | 6.74 | 488 |
| 512 | 8 | 21.46 | 942 | 8.56 | 1054 |
| 512 | 16 | 10.19 | 974 | 9.62 | 937 |
| 512 | 32 | 8.68 | 936 | 8.94 | 937 |
| 512 | 64 | 8.59 | 966 | 8.87 | 970 |
| 512 | 128 | 8.63 | 1007 | 7.97 | 1047 |

As in table 3, table 7 compares the time figures of accelerated algorithms with respect to straight ones. The variations appear irrelevant. One assumes that the Xilinx synthesizer is leading to better or similar performances than the accelerated algorithms implementations. Full custom circuit will most probably take a better profit of the circuit reductions proposed in this paper

Table 7. Time comparisons between straight and accelerated algorithms with I/O registers.

| <i>N</i> | <i>End of 0-sequence</i> | | | <i>End of 1-sequence</i> | | |
|----------|--------------------------|--------------------|---------------|--------------------------|--------------------|---------------|
| | <i>Straight</i> | <i>Accelerated</i> | <i>Saving</i> | <i>Straight</i> | <i>Accelerated</i> | <i>Saving</i> |
| 16 | 1.81 | 1.92 | -6 % | 2.02 | 1.86 | 8 % |
| 32 | 2.48 | 2.32 | 6 % | 2.52 | 2.51 | 0 % |
| 64 | 2.91 | 2.96 | -2 % | 3.30 | 3.23 | 2 % |
| 128 | 5.02 | 5.24 | -4 % | 5.27 | 4.88 | 7,4 % |
| 256 | 6.19 | 6.54 | -6 % | 6.73 | 6.03 | 10 % |
| 512 | 8.11 | 8.59 | -6 % | 8.76 | 7.97 | 9 % |

Finally table 8 shows, as in table 4, the area consumption comparisons between straight and accelerated algorithms. It is noteworthy to observe that the accelerated algorithms lead to savings (up to 18 %) in slices consumption.

Tabla 8. Area requirements comparison.

| N | End of 0-sequence | | | End of 1-sequence | | |
|------------|--------------------------|--------------------|-------------------|--------------------------|--------------------|-------------------|
| | Straight | Accelerated | Difference | Straight | Accelerated | Difference |
| 16 | 23 | 23 | 0 % | 23 | 23 | 0 % |
| 32 | 53 | 53 | 0 % | 48 | 47 | -3 % |
| 64 | 118 | 97 | -18 % | 111 | 101 | -9 % |
| 128 | 268 | 259 | -3 % | 273 | 234 | -14 % |
| 256 | 531 | 486 | -8 % | 504 | 515 | 2 % |
| 512 | 1145 | 966 | -16 % | 1088 | 1047 | -4 % |

8. CONCLUSIONS

Some optimized procedures have been presented for recursive circuits for which skipping techniques provide significant time savings in propagation times. FPGA (Virtex 4 Xilinx) implementations have been carried out in order to evaluate performances in this particular popular technology. For implementations without I/O registers, the proposed acceleration lead to time savings up to 50 % with less than 11 % area requirements. With I/O registers the implementations do not produce time reduction although area consumption is somewhat reduced. This is due to a better optimization process whenever the synthesizer has to cope with I/O register design allowing a faster clock frequency. Actually the proposed circuits are mostly generic and not aimed at a particular FPGA technology. It is well assumed that full custom design would lead to a best time saving with minimum hardware consumption, as it appears in the performance formulas presented in this paper. Nevertheless FPGA technology is quite attractive as figures can be improved using relative location (*RLOC*) design techniques [9]. This remains as an open experimental project.

9. REFERENCES

- [1] J-P. Deschamps, G. Bioul, and G. Sutter, *Synthesis of Arithmetic Circuits, FPGA, ASIC, and Embedded Systems*, John Wiley Interscience, New York 2006.
- [2] G. Bioul, J.-P. Deschamps, G. Sutter, "Efficient FPGA Implementation of Carry-Skip Adders," III Jornadas de Computación Reconfigurable y Aplicaciones, Madrid, September 2003, pp. 81-90.
- [3] G. Bioul, M. Vazquez, N. Acosta, M. Oriol, "An improved convergence algorithm to compute $\ln(x)$ – FPGA implementations", XIII Congreso CACIC, Argentina, 2007.
- [4] PAR1999] B. Parhami, *Computer Arithmetic, Algorithms and Hardware Designs*, Oxford University Press, New York, 1999.
- [5] ERC2004] M.D. Ercegovac, and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, San Francisco, CA 2004.
- [6] Xilinx Inc. "Virtex 4 User Guide", Abril 2007, available in <http://www.xilinx.com>.
- [7] Xilinx Inc. "XST User Guide-9.2i", 2008, available in <http://www.xilinx.com>.
- [8] Xilinx Inc. "ISE 9.2 Documentación", 2008, available in <http://www.xilinx.com>.
- [9] Xilinx, Inc. "Constraints Guide – ISE9.2i", chapter 2 Relative Location (*RLOC*), 2008, available in <http://support.xilinx.com>