Automatic Allocation of Redundant Operators in Arithmetic Data path Optimization

Sophie Belloeil, Roselyne Chotin-Avot, Habib Mehrez and Alix Munier-Kordon
University Paris VI, LIP6/SOC Laboratory
4 place Jussieu,
75252 Paris Cedex 05, France
Sophie.Belloeil@lip6.fr

Abstract

Redundant operators such as adders and multipliers increase performance (timing and area) of high computational digital circuits. Mixing redundant operators with classical ones is nevertheless complex for circuit designers who might not have necessarily the required arithmetic knowledge. In this context, Computer Aided Design tools considering redundant arithmetic are of interest.

In this paper, several algorithms based on graph theory are described. They replace some classical operators of a design with redundant ones to minimize the overall timing. Several real life experiments are presented.

I. Introduction

Redundant operators such as adders and multipliers have very good performances considering time and area [7], [10], [14]. Using those operators in VLSI circuit design can thus appear advantageous, enabling architecture optimizations and consequently further improvements as for circuits performances. Several hand-made implementations have been done using those architectures, leading to good results [5], [6], [9], [16]: improvement up to 35% for the frequency with an area overhead bounded by 11% for a Discrete Cosine Transform Operator for example.

Mixing classical and redundant arithmetics in an explicit way can nevertheless appear quite tedious to non initiated designers, for whom, furthermore, the rapid pace of technological evolution puts a great "time to market" pressure. Such a pressure on design cycle combined with strict performance contraints make the automation of the introduction of redundant arithmetic in circuit design more and more useful, bringing it more accessible. There has therefore been an extensive research work on the introduction of redundant arithmetic in logical synthesis [11], [12], [15]. However, they focus mainly on using only Carry-Save adders, and choose to transform substractions and multiplications into additions. They do not address the possibility of using redundant multipliers as well as the Borrow-Save representation.

We have already proposed an approach based on pattern matching techniques in [2]. We also have demonstrated the interest of using the Borrow-Save representation in [3]. However, the pattern matching approach is not the most appropriate one for handling circuits which have operators with multiple fanouts. Algorithms have been developped for it to be handled, but increase a lot the time to perform the optimisation.

In this paper, we present a new approach based on graph theory. The two criteria considered are again the timing and the area minimization. Two heuristics solving this optimization problem are developped. The first one modifies classical operators into redundant ones as much as possible like the pattern matching approach. The second one uses a cost function in order to choose the best allocation possible for each operator. Furthermore, these two algorithms consider Borrow-Save architectures to handle substractions.

Several designs, such as a FIR filter, a Distance Computation Unit (DCU), a Fast Fourier Transform (FFT) butterfly and a Discrete Cosine Transform (DCT) are optimised, using our two algorithms.

The remainder of this paper is organized as follows: Section 2 contains a global description of the redundant arithmetic and the associated architectures. In Section 3, we describe our redundant optimization algorithms. Our experimental results are presented in Section 4. Section 5 is our conclusion.

II. Redundant arithmetic

A. Mixed arithmetic

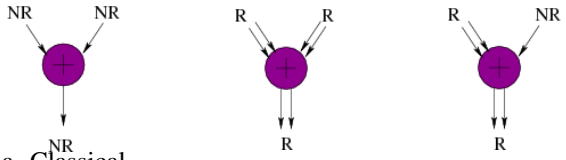
Redundant arithmetic involves two number representations [1]:

- **Carry-Save** representation: a digit is defined by $cs_i = cs_i^0 + cs_i^1$ with $cs_i \in \{0, 1, 2\}$ so that a number is considered as the sum of two terms: $CS = CS^0 + CS^1$.
- **Borrow-Save** representation: a digit is defined by $bs_i = bs_i^0 - bs_i^1$ with $bs_i \in \{-1, 0, 1\}$ so that a number is considered as the substraction of two terms: $BS = BS^0 - BS^1$.

The abbreviations CS and BS are commonly used for Carry-Save and Borrow-Save representations, as well as NR and R for respectively Non Redundant and Redundant representations.

The sole use of redundant arithmetic in data path description is not conceivable for several reasons. Firstly, we must preserve the NR representation of the inputs/outputs of the circuits. Secondly, we have to deal with non-arithmetic operators such as multiplexors, boolean operators, etc... Classical and redundant arithmetics have therefore to be compatible. A new arithmetic is presented, called **mixed arithmetic**, defined as the combination between classical and redundant representations. This involves:

1) having at disposal every arithmetic operator accepting both R and NR inputs/outputs: the three representations of classical, redundant and mixed adders are presented in Figure 1 for example.
2) being able to convert one representation to the other: for example, the conversion CS→NR is the addition between the two terms composing the CS number.



a. Classical adder b. Redundant adder c. Mixed adder

Fig. 1. Adders representation

B. Carry-Save architectures

1) Adders: The architecture of a **redundant adder**, adding two CS numbers, is presented in Figure 2. A similar architecture exists for a **mixed adder**, adding a CS number and a NR number, such as shown in Figure 3. Both adders provide a CS output made of the effective sum and the carries ($Output = S + C$).

These architectures show the main benefit of redundant arithmetic: it allows to suppress the carry propagation in the computation of an addition. The time to perform an addition of two numbers is thus constant, independent of the number of digits. Addition being an essential operator, the potential benefit of using redundant/mixed adders is significant.

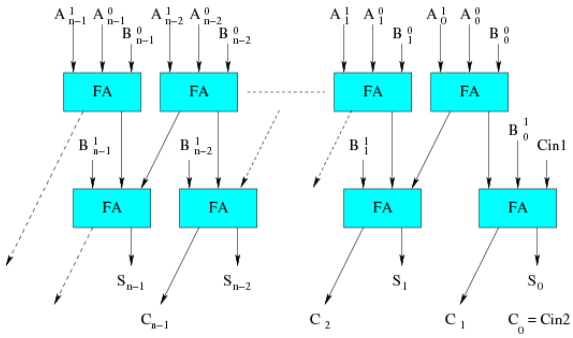


Fig. 2. Implementation of a redundant adder

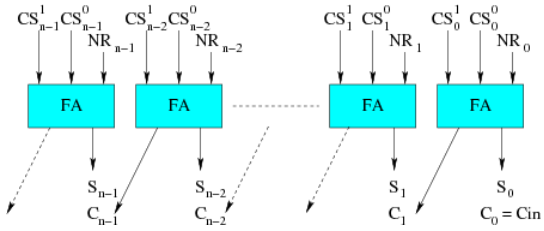


Fig. 3. Implementation of a mixed adder

2) Multipliers: The multiplier architecture frame (Figure 4) is divided into four parts [7]: (1) recoding (for inputs in R form), (2-3) partial products + sum, (4) conversion (to obtain the output in NR form). The second and third parts are mandatory; they represent the effective computation of the multiplication. The first and last parts are optionnal, depending on the representations of the inputs/output. The computations made are then $Output = (CS^0 + CS^1) * NR$ for mixed multipliers and $Output = (CS_0^0 + CS_0^1) * (CS_1^0 + CS_1^1)$ for redundant ones.

Since the output of the Wallace tree [14] (part 3) is in CS form, allowing the output of the multiplier to be in CS form generates the supression of the final conversion. CS multipliers are therefore bound to have a better critical time than classical ones, but a bigger area because of the input recoders (especially with two R inputs).

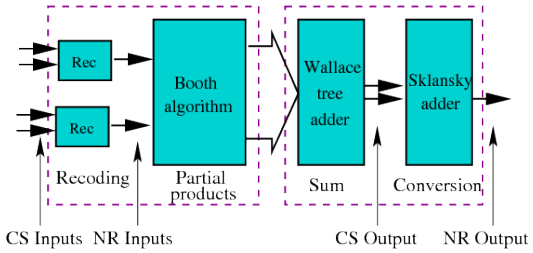


Fig. 4. Multiplier

C. Borrow-Save architectures

1) Adders: BS architectures allow to perform substractions. They have the same frame as the CS ones (cf. the mixed adder architecture in Figure 5). Using the BS representation, a mixed adder (adding a BS number and a NR number) consists in computing $Output = C + S = BS^+ - BS^- + NR$. A redundant adder (adding two BS numbers) consists in $Output = C + S = BS_0^+ - BS_0^- + BS_1^+ - BS_1^-$.

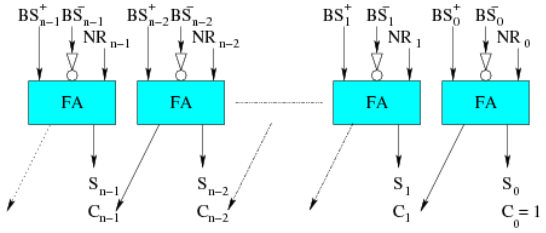


Fig. 5. Borrow Save mixed adder

The three different ways to implement the computation $(a-b)+(c-d)$ are shown in Figure 6: the NR architecture (Sklansky adders [14]: substractions transformed into additions using the two's complement) in Figure 6.a, the CS architecture (same treatment as for substractions) in Figure 6.b and the BS one in Figure 6.c. We have compared in [3] the use of those architectures, in terms of time and area [1]. The results presented show that the BS architectures performances are in the same order of magnitude as the CS ones.

2) Multipliers: BS architectures have the same frame as the CS ones [7]. The difference is in the recoding part. The computations made are then $Output = S + C = (BS^+ - BS^-) * NR$ for mixed multipliers and $Output = S + C = (BS_0^+ - BS_0^-) * (BS_1^+ - BS_1^-)$ for redundant ones.

We have compared in [3] the use of the different architectures to perform a computation involving substractions and multiplications: $(a - b) * (c - d)$. As before, three architectures habe been compared: the NR architecture (Sklansky adders and a classical multiplier), the CS one and the BS one. The results show that the BS architecture is faster and smaller than the CS one.

[1]Note that for all the tests presented, we used the place and route tools of the Cadence CAD System using the Alliance [8] CMOS Standard Cell Library in 0.35μm

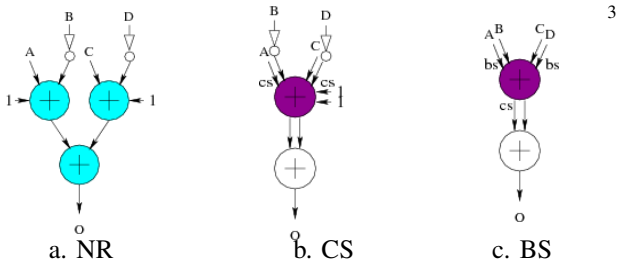


Fig. 6. Computation of $(a - b) + (c - d)$

D. Advantage of the Borrow-Save representation

It has been shown in [3] that the interest of using the BS representation is better emphasized when substractions are inside arithmetical chains. Let us consider the computation $(a + b) - (c + d)$. The three possible architectures to implement this computation are presented in Figure 7. We can see in Figure 7.b that, when using the CS architectures, the introduction of an inverter in the arithmetical chain prevents from using a redundant operator, leading to a non optimal design. Using the BS architecture avoids that issue, such as shown in Figure 7.c.

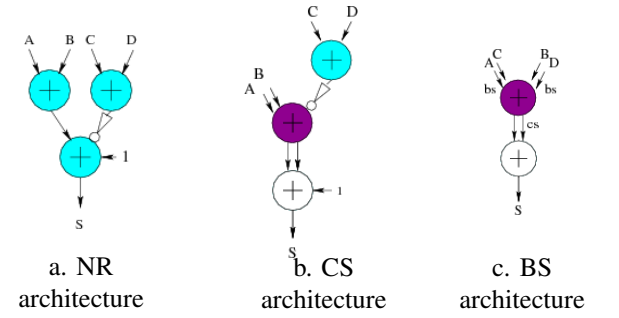


Fig. 7. Computation of $(a + b) - (c + d)$

The timing and area performances of these implementations are presented in [3]: using CS architectures results in 25.7% area improvement and 6.7% timing deterioration whereas using BS architectures result in 53.6% area improvement and 27.5% timing improvement, compared to the classical implementation (Figure 7.a). The BS implementation is therefore 94% smaller and 32% faster than the CS one.

III. Automatic optimization

Our aim is to take advantage of redundant arithmetic in order to improve circuits timing. Our CAD tools are part of a classical VLSI design flow and take place just before logical synthesis. We have developped two algorithms which, from a functional specification of a circuit described with the Stratus language [4], modify the specifications of the different operators and interconnections of the circuit, while preserving its behavior and inputs/outputs. After this process, the VLSI design flow remains unchanged.

This Section is organized as follows. First, some definitions are introduced. Secondly, our algorithms are described. Last but not least, the implementation of the final circuit is presented.

A. Definitions

Our optimization algorithms are based on graph theory. We therefore consider our circuits as graphs to use algorithmic tools to optimize them.

1) Arithmetic computation graph: An *arithmetic datapath* can be modelled using a directed acyclic graph $G = (V, \mathcal{A})$ such that:

1) the set of operators (arithmetic or not), inputs and outputs of the circuit represents the set of vertices V of the graph G,
2) all signals of the circuit represent the set of arc $(x, y) \in \mathcal{A}$.

For any vertex $x \in V$, $\Gamma^-(x)$ denotes the set of direct predecessors of x,

$$\Gamma^-(x) = \{y \in V, (y, x) \in \mathcal{A}\}.$$

2) Allocation function: An allocation function is defined in order to distinguish the signals in redundant representation from the others. The reasons a signal can not be put in redundant representation are if it is an input/output of the circuit, or if it is an input/output of a non arithmetical operator. Let $V_o \subset V$ be the set of vertices from V corresponding to binary arithmetic operators. Elements from V_o may be implemented using classical, mixed or redundant operators. Elements from $V - V_o$ correspond then to inputs, outputs or non arithmetic operators. The set of arcs that may be implemented using a redundant representation is then $\mathcal{A}_o = \{(x, y) \in \mathcal{A}, x \in V_o, y \in V_o\}$.

An *allocation* is a function $a : \mathcal{A} \rightarrow \{0, 1\}$ such that,

1) $\forall (x, y) \in \mathcal{A} - \mathcal{A}_o, a(x, y) = 0$;
2) $\forall (x, y) \in \mathcal{A}_o, a(x, y) = 1$ if (x, y) is in redundant representation, $a(x, y) = 0$ otherwise.

For any feasible allocation a, the set of arcs implemented using a redundant representation is $\mathcal{R}(a) = \{(x, y) \in \mathcal{A}_o, a(x, y) = 1\}$.

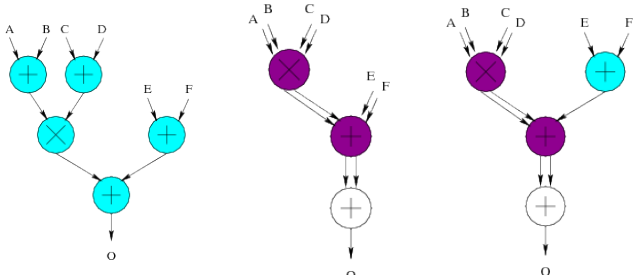
B. *All in redundant* algorithm

The *all in redundant* algorithm moves all arcs from $\mathcal{A}_o$ into redundant representation, *i.e.* for any arc $e = (x, y) \in \mathcal{A}_o$, $a(e) = 1$ so that $\mathcal{R}(a) = \mathcal{A}_o$.

In other words, this algorithm can be stated as follows: *Given a graph G of arithmetic computations, classical arithmetic operators are transformed, when possible, into their mixed or redundant form, with the preservation of the functionnality of the design.*

C. Optimal allocation algorithm

1) Motivations: Transforming systematically each arithmetical operator into its redundant form leads to good results but is not necessarily the optimal approach to obtain the best timing. The computation in Figure 8 emphasises this issue:



a. Cell b. All in redundant c. Optimal allocation

Fig. 8. Timing optimal allocation

- the critical path of the classical implementation contains two classical adders and one classical multiplier (cf. Figure 8.a),
- the *all in redundant* algorithm reduces the critical path to one redundant multiplier, one redundant adder and one classical adder (cf. Figure 8.b),
- since a classical adder is faster than a redundant multiplier, the addition between E and F can be performed in parallel with the multiplication without altering the critical path. Not transforming the adder results in changing the redundant adder in the critical path into a mixed adder. The critical path is therefore smaller (cf. Figure 8.c).

Table I presents the performances of the different architectures of Figure 8. They show an increase of the area and an improvement od the timing. The improvement of the timing being our main objective, those result confirm that, with an optimal allocation, the optimization of the timing is better than with the *all in redundant* allocation, with an area overhead.

width	Area (mm^2)			Time (ns)		
	a	b	c	a	b	c
8	0.24 ref	0.21 -9.8%	0.24 0%	60.5 ref	54.7 -9.6%	50.6 -16.4%
16	0.66 ref	0.61 -8%	0.68 +1.9%	81.64 ref	65.71 -19.5%	61.71 -24.4%

TABLE I. Computation of $(a+b)*(c+d)+(e+f)$

2) Cost of an allocation function: In order to find the best allocation choice for the operators, a cost function which evaluates each operator is defined, such as follows.

Let a be an allocation function such as defined in III-A.2. The cost of any arc $e = (x, y) \in \mathcal{A}$ represents the cost of its input vertex and therefore depends on $a(x, y)$ considering the 8 possibilities presented by Figure 9. This cost, noted $\mathcal{C}(a, (x, y))$, is computed as follows:

1) if $a(x, y) = 1$, then the arc (x, y) is implemented using a redundant representation. Its cost depends on the representation of the two inputs arcs of x following the cases (a), (b1), (b2) and (c) in Figure 9.
2) if $a(x, y) = 0$, then the arc (x, y) is implemented using a classical representation,
   a) if $x \in V - V_0$, the cost of (x, y) is a constant independent from a,
   b) if $x \in V_o$, the cost of (x, y) depends on the representation of the two input arcs of x following the cases (d), (e1), (e2) and (f) in Figure 9.

In case of arithmetical operators (cases 1 and 2.b), the cost is based on the complexity of the correponding architecture, for example: 1 for a mixed adder, 2 for a redundant adder, $log_2(n)$ for a slansky adder, ...

Let $\mathcal{P}(G)$ denotes the paths of G. The cost of any path $\nu \in \mathcal{P}(G)$ is $\sum_{e\in\nu} \mathcal{C}(a, e)$. The cost $\mathcal{C}(a)$ of an allocation is the maximum cost of a path from G, so

$$\mathcal{C}(a) = \max_{\nu\in\mathcal{P}(G)} \sum_{e\in\nu} \mathcal{C}(a, e).$$

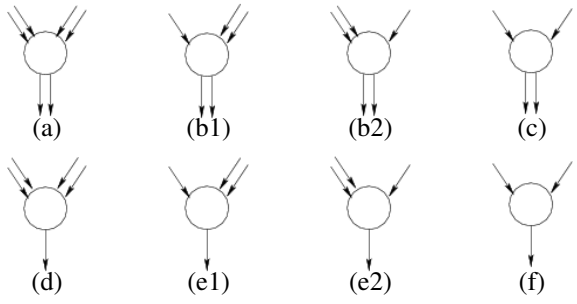


Fig. 9. All cases of connections

3) Allocation functions for in-trees: We suppose here that the arithmetic data-path graph is an in-tree denoted by τ. So, every vertex $x \in V$ has one successor in G.

4) Algorithm: The *optimal allocation* algorithm minimizes the cost function using dynamic programming. Let the arc $e = (x, y) \in \mathcal{A}$ and let $\tau(x)$ be the sub-tree of τ rooted in x. Let also $\mathcal{O}(e, 1)$ be a set of arcs from $\tau(x)$ in a redundant representation for an optimal solution for the graph $\tau(x)$ with the constraint that $a(e) = 1$. On the same way, let $\mathcal{O}(e, 0)$ be a set of arcs from $\tau(x)$ in a redundant representation for an optimal solution for the graph $\tau(x)$ with the constraint that $a(e) = 0$.

Sets $\mathcal{O}(e, 0)$ and $\mathcal{O}(e, 1)$ are built recursively as follows:

1) If $x \in V - V_o$, then every allocation function verifies $a(e) = 0$ and thus $\mathcal{O}(e, 1)$ is not defined. If x is an input vertex, we set $\mathcal{O}(e, 0) = \emptyset$. Otherwise, all the elements from $\Gamma^-(x)$ must be classical operators and $\mathcal{O}(e, 0) = \bigcup_{y\in\Gamma^-(x)} \mathcal{O}((y, x), 0)$.
2) Otherwise, $x \in V_o$ is a binary arithmetic operator and has exactly two inputs vertices denoted by y_1 and y_2.
   - For $\mathcal{O}(e, 0)$, we fix $a(e) = 0$ and the output of x is implemented using a classical representation. Then, $\mathcal{O}(e, 0)$ is the minimum cost solution between the four following alternatives:
     a) the arcs (y_1, x) and (y_2, x) are implemented using a redundant representation, so the first alternative is $\mathcal{O}((y_1, x), 1) \cup \mathcal{O}((y_2, x), 1) \cup \{(y_1, x), (y_2, x)\}$ (case (d) of Figure 9);
     b) the arc (y_1, x) is implemented using a classical representation, and the arc (y_2, x) is implemented using a redundant representation, so the second alternative is $\mathcal{O}((y_1, x), 0) \cup \mathcal{O}((y_2, x), 1) \cup \{(y_2, x)\}$ (case (e1) of Figure 9);
     c) the arc (y_1, x) is implemented using a redundant representation, and the arc (y_2, x) is implemented using a classical representation, so the third alternative is $\mathcal{O}((y_1, x), 1) \cup \mathcal{O}((y_2, x), 0) \cup \{(y_1, x)\}$ (case (e2) of Figure 9);
     d) lastly, arcs (y_1, x) and (y_2, x) are implemented using a classical representation and the fourth alternative is $\mathcal{O}((y_1, x), 0) \cup \mathcal{O}((y_2, x), 0)$ (case (f) of Figure 9).
   - $\mathcal{O}(e, 1)$ is evaluated on the same way by considering that the output of x is implemented using a redundant representation. Four alternatives are investigated following cases (a), (b1), (b2) and (c) of Figure 9.

The optimum solution is $\mathcal{O}((x, y), 0)$ with y, the root of τ (since $y \in V - V_o$).

D. Implementation

Once the used algorithm has determined the best representation of each arc, the corresponding optimized circuit has to be created. It can be feasible only if, for each operation (addition, substraction, multiplication), an architecture exists for every possible case of connection.

Table II presents all possible cases and the corresponding architectures. Three architectures exist for each operation (the classical one, the mixed one and the redundant one), which is sufficient to handle all the cases (a conversion $CS \rightarrow NR$ is done if the output of a R operator has to be NR). Comments can nevertheless be done upon the two cases marqued with a "*". In these cases, a null value has to be added in order to use redundant operators. The operation wanted is indeed $NR-(CS^0+CS^1)$ which is equal to $NR-CS^0-CS^1$. This operation can be implemented only by transforming it into $(NR-CS^0)+('0'-CS^1)$.

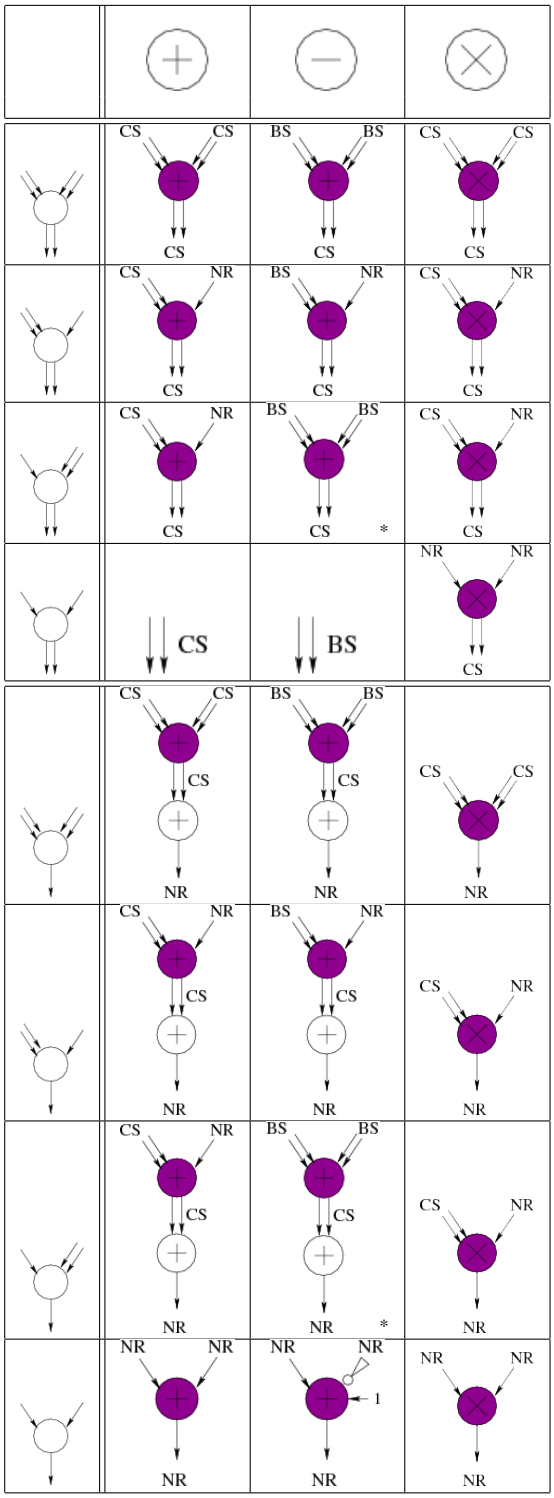


TABLE II. Architectures

IV. Experimental results

The tests performed are meant to show the usefulness of the redundant arithmetic. We have therefore made, for each benchmark, an implementation using classical arithmetic only, and one or several implementations using our algorithms. We present the performances of the circuits, in terms of timing and area, with and without optimizations.

The first two benchmarks presented (FIR and DCU) have been optimized with the two different optimization algorithms. They can be modelled using trees and are therefore supported by the *optimal allocation* algorithm. The other two benchmarks (FTT butterfly and DCT) can not be modelled using trees, the optimizations presented result then from the *all in redundant* algorithm.

1) FIR Filter operator: The filter architecture is shown in Figure 10. Three implementations are done: the classical one, the one resulting from the *all in redudant* algorithm, and the one resulting from the *optimal allocation* algorithm. Since this design contains no substraction, both algorithms use the CS representation.

The results (from a filter with 8 registers and 16 bits datas) are summarized in Table III. Thoses results show that both algorithms optimize timing and area. They also show that the *optimal allocation* algorithm optimizes better the timing, but less the area.

Architectures obtained with the different algorithms are shown in Figure 11 (exemple with 4 registers):

- Figure 11.a presents the result of the *all in redundant* algorithm, in which all arithmetical operators are transformed into their redundant form.

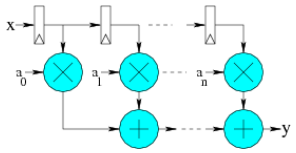


Fig. 10. FIR filter operator

	Classical	All redundant	Optimal
Area (mm^2)	3.97 ref	2.94 -25.92%	3.34 -15.83%
Time (ns)	134.39 ref	109.58 -18.47%	87.95 -34.56%

TABLE III. Results of the FIR filter

- Figure 11.b presents the result of the *optimal allocation* algorithm, in which the instanciation of several classical multipliers leads to mixed adders in the critical path instead of redundant adders. This choice can be made because redundant representations are not nessessary in this case to improve the timing: the classical multipliers remain faster than the computation made on the other input of the adders in the critical path.

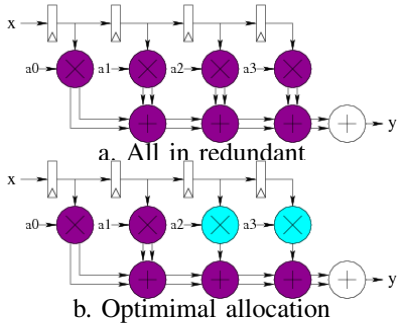


Fig. 11. FIR filter optimizations

2) DCU operator: The DCU architecture [5] is composed of two parts, such as shown in Figure 12: the first one computes the distances $(A_i - B_i)^2$, the second one performs the sum of these distances.

The results, summarized in Table IV, present the optimizations of both algorithms. This operator containing subtractions, each algorithm is used twice, once using the CS architectures, once using the BS ones. Those results show that the *optimal allocation* algorithm results in the same architecture as the *all in redundant* one: in this case the *optimal allocation* is indeed to transform all the arcs into redundant representation. They also show that the BS architectures produce a better timing and a better area than the CS ones.

3) FFT butterfly: The *butterfly* is the elementary cell composing the Fast Fourier Transform [16]. Its architecture

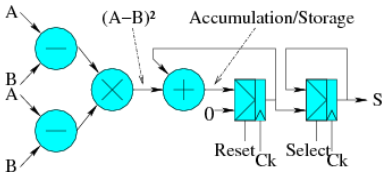


Fig. 12. DCU operator

	Classical	All Redundant		Optimal	
		CS	BS	CS	BS
Area (mm^2)	0.24 ref	0.29 +24.27%	0.24 0%	0.29 +24.27%	0.24 0%
Time (ns)	65.87 ref	56.72 -13.89%	54.44 -17.36%	56.72 -13.89%	54.44 -17.36%

TABLE IV. Results of the DCU operator

is shown in Figure 13. In this Figure, complex numbers are used, and we have:$X = A + w.B$ $Y = A - w.B$ where $w = Cos + i.Sin$.

The results, from the *all in redundant* algorithm with CS architectures and BS ones, are summarized in Table V. Once again, the BS architectures result in better performances than the CS ones.

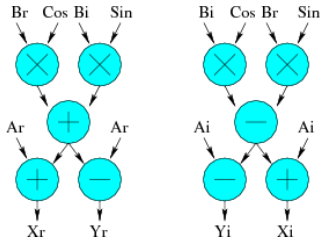


Fig. 13. FFT butterfly operator

	Classical	All Redundant	
		CS	BS
Area (mm^2)	0.77 ref	0.68 -10.91%	0.62 -19.54%
Time (ns)	63.68 ref	66.24 +4.02%	56.42 -11.39%

TABLE V. Results of the Butterfly operator

4) 1-D DCT operator: The architecture of the DCT is shown in Figure 14. It represents the the Loeffer Signal Flow Graph [13] which computes the 1-D DCT of 8 pixels in only one cycle.

The results are summarized in Table VI, resulting again in better performances for the BS architectures. Let us compare those implementations with the ones resulting from our previous approach in [2] and [3]: the optimal algorithm produces better area (up to -4.6%) and timing (up to -1.7%). In addition, the optimal algorithm is performed up to 84% faster than the pattern matching technique.

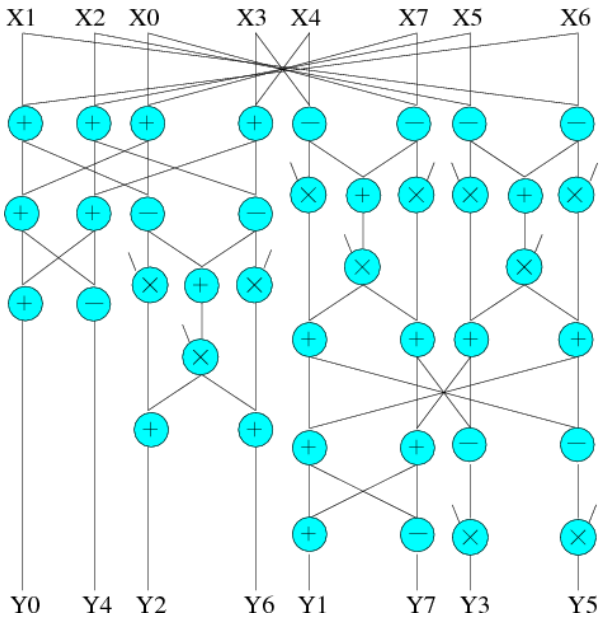


Fig. 14. 1-D DCT operator

	Classical	All redundant		Pattern Matching	
		CS	BS	CS in [2]	BS in [3]
Area (mm^2)	3.96 ref	3.91 -1.23%	3.73 -5.82%	4.1 +3.5%	3.85 -2.8%
Time (ns)	115.7 ref	100.25 -13.35%	95.99 -17.04%	100.53 -13.1%	97.65 -15.6%

TABLE VI. Results of the DCT operator

V. Conclusion

In this paper, two algorithms have been presented, which use automatically redundant arithmetic in order to optimize high computationnal digital circuits. The first algorithm does a systematic transformation of arithmetical operators into their redundant form. The second one does an optimal allocation for the timing. In addition, those algorithms use the Carry-Save architectures only or the Borrow-Save architectures also in order to optimize substractions. We aimed at introducing the different solutions and outlining their prons and cons.

Our experimental results can be summarized as follows. First of all, a systematic transformation of arithmetical operators into their redundant form is not the optimal approach to obtain the best timing. An algorithm finding the best allocation for each operator (classical, mixed or redundant) seems like the best alternative. Second of all, this new approach based on graph theory seems better adapted to DAGs than the previous one.

We aim at testing our algorithms on other benchmarks in order to strengthen our conclusion that the *optimal allocation* algorithm leads to better results. Since this algorithm is currently limited to trees, we also aim at extending its use to DAGs in order to be able to use it on more benchmarks.

References

[1] A. Avizienis. Signed-digit number representation for fast parallel arithmetic. *IRE Trans. Electronic Computers*, 10:389–400, 1962.

[2] S. Belloeil, R. Chotin-Avot, and H. Mehrez. Data path optimization using redundant arithmetic and pattern matching. In *Workshop on Design and Architectures*, 2007.

[3] S. Belloeil, R. Chotin-Avot, and H. Mehrez. Arithmetic data path optimization using borrow-save representation. In *IEEE annual symposium on VLSI*, 2008.

[4] S. Belloeil, D. Dupuis, C. Masson, J.-P. Chaput, and H. Mehrez. Stratus: A procedural circuit description language based upon python. In *International Conference on Microelectronics*, pages 275–278, 2007.

[5] Y. Dumonteix, Y. Bajot, and H. Mehrez. A fast and low-power distance computation unit dedicated to neural networks, based on redundant arithmetic. In *International Symposium on Circuits and Systems*, volume 4, pages 878–881, 2001.

[6] Y. Dumonteix, R. Chotin, and H. Mehrez. Use of redundant arithmetic on architecture and design of a high performance DCT macro-bloc generator. In *Conference on Design of Circuits and Integrated Systems*, pages 428–433, 2000.

[7] Y. Dumonteix and H. Mehrez. A family of redundant multipliers dedicated to fast computation for signal processing. In *International Symposium on Circuits and Systems*, volume 5, pages 325–328, 2000.

[8] A. Greiner and F. Pecheux. Alliance: A complete set of cad tools for teaching vlsi design. In *EuroChip Workshop*, 1992.

[9] A. Guyot. Ocapi: architecture of a vlsi coprocessor for the gcd and the extended gcd of large numbers. *Symposium on Computer Arithmetic*, pages 226–231, 1991.

[10] A. Guyot, Y. Herreros, and J.-M. Muller. Janus, an on-line multiplier/divider for manipulating large numbers. *Symposium on Computer Arithmetic*, pages 106–111, 1989.

[11] T. Kim, W. Jao, and S. Tjiang. Arithmetic optimization using carry-save-adders. In *Design Automation Conference*, pages 433–438, 1998.

[12] Y. Kim and T. Kim. Accurate exploration of timing and area trade-offs in arithmetic optimization using carry-save-adders. In *Conference on Asia South Pacific design automation*, pages 622–628, 2001.

[13] C. Loeffler, A. Lightenberg, and G. Moschytz. Practical fast 1d-dct algorithms with 11 multiplications. In *Intl. Conf. On Acoustics, Speech and Signal Processing*, pages 988–991, 1989.

[14] J. M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser Boston, 1997.

[15] J. Um, T. Kim, and C. Liu. Optimal allocation of carry-save-adders in arithmetic optimization. In *IEEE International Conference on Computer-Aider Design*, pages 410–413, 1999.

[16] A. Vacher, M. Benkhebbab, A. Guyot, T. Rousseau, and A. Skaf. A vlsi implementation of parallel fast fourier transform. *European Design and Test Conference*, pages 250–255, 1994.