Arithmetic Data path Optimization using Borrow-Save Representation

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

Abstract

Considering the performance increase provided by redundant operators such as adders and multipliers, it appears interesting to generalize the use of those operators in high computational digital circuit design. Using redundant arithmetic in conjunction with classical arithmetic is nevertheless a complex task. Optimization CAD tools which automate its use become therefore very helpful. However the existing approaches are restricted in using only the Carry-Save representation. In this paper we propose to overcome this limitation with an exploration of the possible optimizations of using the Borrow-Save representation also. To illustrate this, the optimizations of a Distance Computation Unit (DCU) and a Discrete Cosine Transform (DCT) operators are presented.

1. Introduction

Redundant operators such as adders and multipliers have very good performances in terms of time and area [13, 5, 8]. 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, such as circuits finding the greatest common divisor [7], or performing the fast Fourier transform [14]. Let us consider the implementation of a DCU [3] and a DCT [4] operators: it results in both cases in a significantly increase of the timing performance (up to 35%) with a small area overhead (17% in the worst case), compared to a classical implementation.

Using redundant arithmetic 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 more and more useful the automation of the introduction of redundant arithmetic in circuit design, bringing it more accessible. There has therefore been an extensive research work on the arithmetic optimizations in several areas such as high level synthesis [12] and logical synthesis [9, 10]. Those works however focus on using only the Carry-Save representation. This leads to good results, but it is not the best approach in order to handle substractions.

In this paper, we therefore present the use of the Borrow-Save representation for computations involving substractions. Our work, described in [2], consists in studying the chains of arithmetical operators, and proposing general rules for optimization. Our previous approach, which focused only on the Carry-Save representation, resulted, for the optimization of a DCT operator, in -13.1% for the timing with a area overhead of +3.3%. The performances presented in this paper result from the optimizations of the same operator, as well as a DCU operator, using the Borrow-Save representation. The remainder of this paper is organized as follow: Section 2 and 3 contain a global description of the redundant arithmetic and the associated architectures. In Section 4, we brieftly describe our redundant optimization tool. Section 5 presents the results of our experiments and then we conclude.

2. Redundant arithmetic

2.1. Number representation

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 classical and redundant representations.

2.2. Mixed arithmetic

The sole use of redundant arithmetic in data path description is not conceivable for several reasons. Firstly, we must preserve the NR representations 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. This involves: (1) having at disposal every arithmetic operator accepting both R and NR inputs/outputs, (2) being able to convert one representation to the other (e.g. the conversion CS→NR is the addition between the two terms composing the CS number). This new arithmetic is called **mixed arithmetic**, defined as the combination between classical and redundant representations.

3. Architectures

3.1. Carry-Save architectures

Adders The architecture of a **mixed adder** (adding a CS number and a NR number) is shown in Figure 1. A similar architecture exists for a **redundant adder** (adding two CS numbers). Both adders provide a CS output made of the effective sum and the carries ($Output = S + C$). This architecture shows 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 is thus constant, independent of the number of digits. Addition being an essential operator, the potential benefit of using redundant/mixed adders is significant.

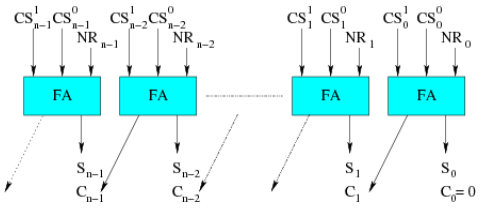


Figure 1. Implementation of a mixed adder

Multipliers The architecture frame (Figure 2) is divided into four parts [5]: (1) recoding, for inputs in R form, (2-3) partial products+sum (effective computation of the multiplication), (4) conversion, to obtain the output in NR form. The second and third parts are mandatory. The first and last parts are optionnal, depending on the representations of the

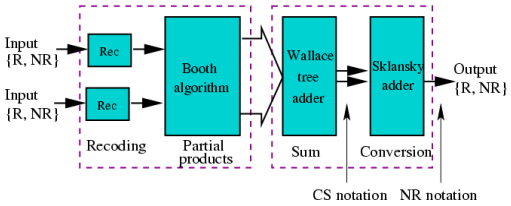
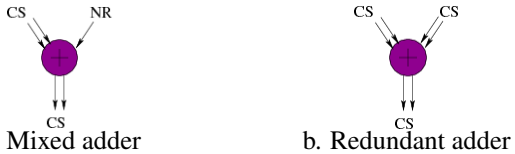


Figure 2. Implementation of a multiplier

inputs/output. The output of the Wallace tree [13] being in CS form, allowing the output of the multiplier to be in CS form produces the deletion 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).

Representations Schemantic representations used for mixed/redundant operators are shown in Figure 3. For all the tests presented, we used the place and route tools of the Cadence CAD System using the Alliance [6] CMOS Standard Cell Library in 0.35μm.



a. Mixed adder b. Redundant adder

Figure 3. Representations

3.2. Borrow-Save architectures

Adders BS architectures allow to perform substractions. Using the BS representation, a mixed adder (adding a BS number and a NR number) consists in computing: BS^+-BS^-+NR whereas a redundant adder (adding two BS numbers) consists in: BS_0^+-BS_0^-+BS_1^+-BS_1^-. Two kind of architectures exist in order to compute those operations: (1) with a CS output (Figure 4) and (2) with a BS output (Figure 5).

Table 1 compares the use of the different architectures, in terms of time and area, to perform an arithmetic computation involving substractions: a classical architecture (Sklansky adder [13]) (NR), the Carry-Save architecture (CS), the Borrow-Save architecture with a BS output (BS1) and the one with a CS output (BS2). Those results show that the BS architectures are, at least, as good as the CS one. The BS1 architecture seems to have a better timing and the BS2 a better area. They indicate also that the larger the numbers are, the better the improvements are.

As it is going to be explained in the following section, the interest of using BS architectures depends on the position

width	Area (mm^2)				Time (ns)			
	NR	CS	BS1	BS2	NR	CS	BS1	BS2
8	0.06 ref	0.03 -50%	0.04 -33.3%	0.03 -50%	23.9 ref	22.4 -6.3%	20.5 -14.2%	22.4 -6.3%
16	0.14 ref	0.07 -50%	0.09 -35.7%	0.05 -64.3%	30 ref	24.4 -18.7%	23.8 -20.7%	25.7 -14.3%
32	0.31 ref	0.15 -51.6%	0.19 -38.7%	0.14 -54.8%	37.7 ref	28.7 -23.9%	28.1 -25.5%	30 -20.4%
64	0.66 ref	0.3 -54.5%	0.39 -40.9%	0.3 -54.5%	48 ref	34.8 -27.5%	34.1 -29%	36.1 -24.8%
128	1.47 ref	0.65 -55.8%	0.82 -44.2%	0.65 -55.8%	63.3 ref	44 -30.5%	43.4 -31.4%	45.4 -28.3%

Table 1. (a - b) + (c - d)

of the substraction in the arithmetical chain: it is more profitable when the substraction is in the middle of the chain.

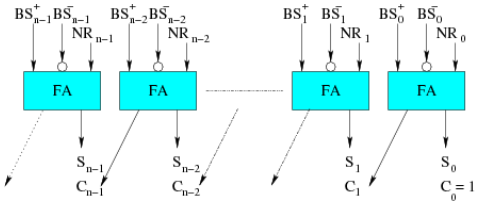


Figure 4. BS adder with a CS output

(a) Adder's architecture

(p) plus-plus-minus cell [8]

Figure 5. BS adder with a BS output

Multipliers BS architectures have the same frame than the CS ones [5]. The difference is in the recoding part. The computations made are then S+C=(BS^+-BS^-)*NR for mixed multipliers and S+C=(BS_0^+-BS_0^-)*(BS_1^+-BS_1^-) for redundant ones.

Table 2 compares the use of the different architectures to perform a computation involving substractions and multiplications: a classical architecture (Sklansky adders+classical multiplier), the CS architecture and the BS one. The results show that the BS architecture has a faster timing than the CS one and that the area overhead is smaller. They indicate also that the larger the numbers are, the better the improvements are.

width	Area (mm^2)			Time (ns)		
	NR	CS	BS	NR	CS	BS
8	0.18 ref	0.25 +38.9%	0.21 +16.7%	50.1 ref	48.4 -3.4%	45.9 -8.4%
16	0.5 ref	0.68 +36%	0.57 +14%	69.3 ref	60.6 -12.5%	57.1 -17.6%

Table 2. (a - b) * (c - d)

4. Automatic optimization

4.1. Pattern matching

Our aim is to take advantage of redundant arithmetic by introducing automatically redundant operators in the process of circuit design. Our CAD tool is part of a classical VLSI design flow and takes place just before logical synthesis: from a functional specification of a circuit and a knowledge in arithmetic, we obtain an optimized *v*irtual description (i.e. before structural mapping) of the circuit. Our tool modifies the specification of the different operators and interconnections, while ensuring the feasibility of the mapping toward a target technology. After this process, the VLSI design flow remains unchanged. It is out of the scope of this paper to detail our tool, interested readers will find informations in [2]. Here we will summarize its main features. The principle is to search for **patterns** (collection of arithmetical operators and their interconnections) which are bound to be replaced by new ones, composed of redundant operators, such as shown in Figure 6. Pattern couples are called **rules**. The fundamental assumption for a rule is that a pattern and its substitute have the same behavior and the same inputs/outputs. The second assumption is that the substitute pattern is more optimized than the first one.

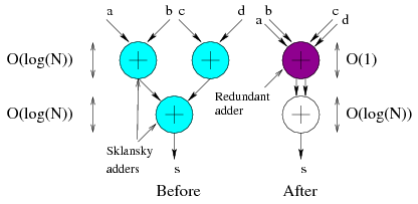


Figure 6. A pattern and its substitute

Three sets of rules have been established: a Carry-Save set and two Borrow-Save sets (one using the BS→BS adders architectures and the other the BS→CS ones). In each set, every pattern has been evaluated in terms of timing and area in order to check that the substitute pattern of each rule has better performances than the one it is substituted to. Each set has been created in order to be sufficient to handle most cases of connection between arithmetic operators: with smaller sets, results might not be optimal, in opposition, bigger sets would not lead to better results, and would make tough the choice of an optimal rule if several patterns matched.

4.2. Advantage of the Borrow-Save representation

We aim at showing the interest of using the BS representation to optimize substractions. It is based on two assumptions. First of all, the BS architectures are as optimized as the CS ones. Second of all, the use of the CS architectures leads to the introduction of invertors in arithmetical chains (because, in order to handle substractions, it is done after having changed the substractions into additions using two's complement), which can block possible optimizations, such as shown in Figure 7. Figure 7.b shows that the use of the BS representation allows to avoid that issue.

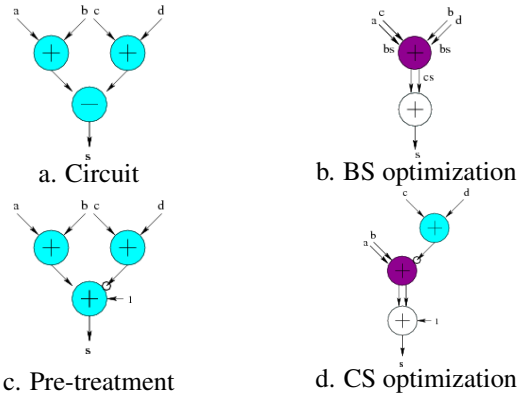


Figure 7. Problem due to a substraction

Table 3 compares the use of the different architectures to perform three arithmetic computations with substractions

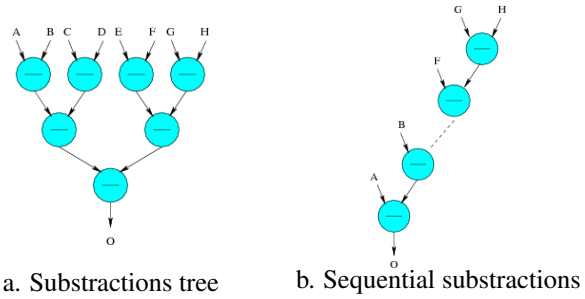


Figure 8. Substraction computations

emphasizing this issue: (1) the computation of Figure 7.a, (2) an eight operands substraction tree (Figure 8.a), (3) eight operands sequential substractions (Figure 8.b). Those results show that both BS architectures result in a better timing and area than the CS one. They highlight better the point of using the BS architectures than the ones of Table 1 as the computations are chosen intentionally in order to have several substractions in a row, i.e. several invertors in the middle of the arithmetical chains.

4.3. Multiple operation trees

One important issue to handle is the management of multiple operation trees. Our tool provides three different ways to handle this.

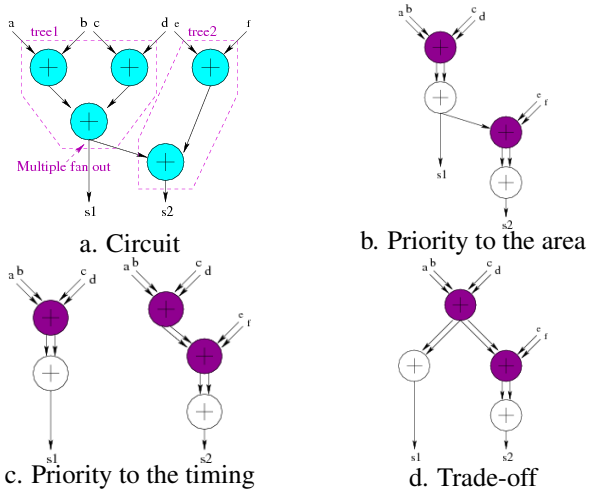


Figure 9. Multiple operation trees

Let us consider the operation of Figure 9.a:

- Each tree can be optimized separately: this produces the minimum area, but a non optimal optimization of the timing (because of an extra cost of CS→NR conversions). This behavior is called **Priority to the area** (Figure 9.b).

Computation	width	Area (mm^2)				Time (ns)			
		NR	CS	BS1	BS2	NR	CS	BS1	BS2
(a+b)-(c+d)	8	0.05	0.04	0.04	0.03	23.1	27.4	20.4	22.4
		ref	-20%	-20%	-40%	ref	+18.6%	-11.7%	-3%
	16	0.13	0.1	0.09	0.07	29.3	33.5	23.8	25.7
		ref	-23.1%	-30.8%	-46.1%	ref	+14.3%	-18.8%	-12.3%
	32	0.3	0.23	0.19	0.14	36.9	41.1	28.1	30
		ref	-23.3%	-36.7%	-53.3%	ref	+11.4%	-23.8%	-18.7%
	64	0.63	0.48	0.39	0.3	47.3	51.5	34.1	36.1
		ref	-24.7%	-38.7%	-51.8%	ref	+8.9%	-27.9%	-23.7%
	128	1.41	1.05	0.82	0.65	62.6	66.8	43.4	45.4
		ref	-25.7%	-41.6%	-53.6%	ref	+6.7%	-30.7%	-27.5%
tree	8	0.13	0.1	0.12	0.11	36	44.5	33.3	34.5
		ref	-23.1%	-7.7%	-15.4%	ref	+23.6%	-7.5%	-4.2%
	16	0.32	0.22	0.26	0.24	44.9	53.3	39.4	40.7
		ref	-31.2%	-18.7%	-25%	ref	+18.7%	-12.2%	-9.3%
	32	0.72	0.49	0.59	0.55	55.3	63.7	47.1	48.3
		ref	-31.9%	-18.1%	-23.6%	ref	+15.2%	-14.8%	-12.7%
	64	1.6	1.1	1.3	1.2	69	77.4	57.5	58.7
		ref	-31.2%	-18.8%	-25%	ref	+12.2%	-16.7%	-14.9%
	128	3.54	2.38	2.84	2.67	88.6	97	73	74
		ref	-32.8%	-19.8%	-24.6%	ref	+9.5%	-17.6%	-16.5%
sequential	8	0.14	0.14	0.11	0.09	76.1	76.1	58.3	53.4
		ref	0%	-21.4%	-35.7%	ref	0%	-23.4%	-29.8%
	16	0.32	0.32	0.24	0.2	93.6	93.6	70.4	64.7
		ref	0%	-25%	-37.5%	ref	0%	-24.8%	-30.9%
	32	0.72	0.72	0.53	0.44	114.7	114.7	83.6	78
		ref	0%	-26.4%	-38.9%	ref	0%	-27.1%	-32%
	64	1.6	1.6	1.15	0.98	140.1	140.1	100.7	95
		ref	0%	-28.1%	-38.7%	ref	0%	-28.1%	-32.2%
	128	3.54	3.54	2.49	2.16	174.1	174.1	124.6	118.2
		ref	0%	-29.7%	-39%	ref	0%	-28.4%	-32.1%

Table 3. Computations involving substractions

- In order to avoid that problem, another behavior, called **Priority to the timing**, consists in treating every expression with a separate tree and without any ressource sharing: this generates a minimal timing, but an excessive overload as for the area (Figure 9.c).

- The most optimal solution is to make **trade-offs** between area and timing as discussed in [10]: it allows to optimize the timing while minimizing the area penalty (Figure 9.d).

5. Benchmarks

DCU operator The architecture [3] is composed of two parts (Figure 10): the first one computes the distances (A_i - B_i)^2, the second one performs the sum of these distances.

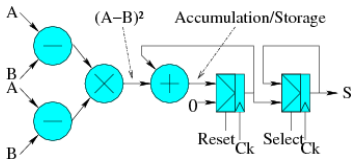


Figure 10. DCU operator

1-D DCT operator Several algorithms have been proposed in order to compute the 1-D DCT. We have chosen the Loeffer Signal Flow Graph [11] which has been widely used. This implementation (Figure 11) computes the 1-D DCT of 8 pixels in only one cycle.

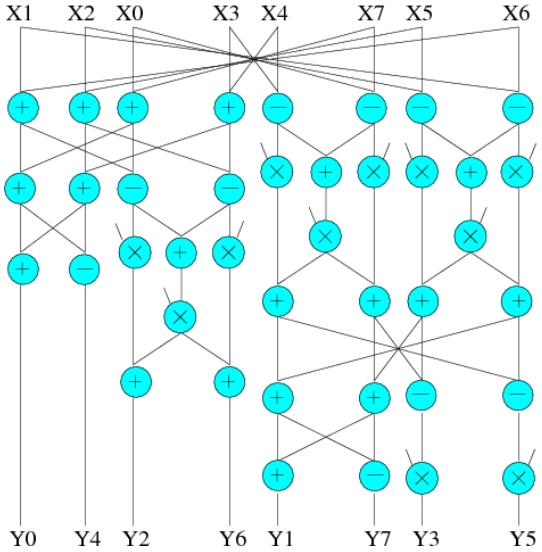


Figure 11. 1-D DCT operator

Architecture	Mode	Area (mm^2)				Time (ns)			
		No	CS	BS 1	BS 2	No	CS	BS 1	BS 2
DCU Operator	-	0.24 ref	0.29 +20.8%	0.24 0%	0.24 0%	65.9 ref	56.7 -14%	54.4 -17.4%	54.4 -17.4%
1-D DCT operator	Priority to the area	3.96 ref	3.9 -1.5%	3.9 -1.5%	3.95 -0.25%	115.7 ref	114.84 -0.7%	114.84 -0.7%	117.04 +1.2%
	Priority to the timing	3.96 ref	7.41 +87.1%	7.43 +87.6%	8.11 +104.8%	115.7 ref	106.27 -8.1%	98.84 -14.6%	96.25 -16.8%
	Trade -off	3.96 ref	4.09 +3.3%	4.91 +24%	3.85 -2.8%	115.7 ref	100.53 -13.1%	96.57 -16.5%	97.65 -15.6%

Table 4. Experimental results

Experimental results The tests performed are meant to show the usefulness of the Borrow-Save representation. We have therefore made, from the classical arihmetic representation, the three different kinds of optimizations: using the CS architectures, and the two kind of BS architectures.

The DCT architecture contains multiple operation trees, and therefore is a good example in order to test the performances of the three kinds of options of the tool also: **Priority to timing**, **Priority to area** and **Trade-off**. Table 4 shows the performances obtained with the different behaviors as for timing and area: it results in a better timing improvement for both BS architectures compared with the CS one, with, in the worst case, a limited area overhead. Note that those BS results offer better performances in area than the manual implementation presented in [3, 4] with the performances in timing in the same order of magnitude.

6. Conclusion

In this paper, we have presented the use of the Borrow-Save representation during automatic optimization of high computational digital circuits using redundant arithmetic. We aimed at showing that using this representation in conjonction with the Carry-Save representation leads to better results than using only the Carry-Save representation, when optimizing computations involving substractions. Experimental results indicate that the use of Borrow-Save architectures leads to a better improvement of the critical time than the use of the Carry-Save ones, with areas in the same order of magnitude. This shows that our work can be used effectively on several designs with mixture of additions, substractions and multiplications. We therefore aim at testing our tool on other benchmarks with substractions in order to strenghten our conclusion.

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] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] A. Mignotte, J.-M. Muller, and O. Peyran. Mixed arithmetic: Introduction and design structure. In *International conference on Massively Parallel Computing Systems*, 1996.

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

[14] 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.