A High-Level Synthesis Approach to Optimum Design of Self-Checking Circuits

Anna Antola, Vincenzo Piuri, Mariagiovanna Sami
Dipartimento di Elettronica e Informazione, Politecnico di Milano
piazza L. da Vinci 32, 20133 Milano, Italy
email {antola,piuri,sami}@elet.polimi.it

Abstract

We present an innovative solution to design of self-checking systems implementing arithmetic algorithms. Rather than substituting self-checking units in system synthesized independently of self-checking requirements, we introduce self-checking in high-level synthesis as a requirement already for scheduling the DFG. Rules granting error detection allow optimum partitioning of the DFG; minimum-latency, resource-constrained scheduling is performed with the support of such partitioning so as to optimize the number of checkers as well as that of other resources.

1. Introduction

High-level synthesis has been a subject of research and development since several years now; starting from an algorithmic description of a specific device's computation, techniques have been defined to derive an intermediate graph description and to perform a number of operations and optimizations leading ultimately to *scheduling, allocation* and *binding* [1,2]. Main figures of merit are *latency, throughput* and *resources*; even though the final product of high-level synthesis consists of an RT-level datapath and of a transition table for the control FSM, such figures of merit can be evaluated, at least roughly, on the basis of parameters characterizing the library cells used for the subsequent architectural synthesis.

A particular problem, not usually taken into account, concerns design of *self-checking systems*; this feature is usually accounted for at a lower level, e.g., by introducing suitable modifications in the state table and coding in the state assignment phase of an FSM [3,4] or by adopting coded data and related self-checking operating units in the datapath [5]. Such solutions are adopted *a posteriori*, after high-level synthesis has been completed, and thus are not optimized with respect to the specific computation performed by the ASIC. Let us focus on data path synthesis; the conventional approach leads to excessive area increase, since it tends to protect the individual operation rather than to exploit the characteristics of the coding approach with respect to the computation to be performed. For the same reason, there is also a deterioration in both latency and throughput since the control step must now accommodate checking as well as nominal operation. It appears worthwhile to introduce self-checking requirements already in the initial steps of high-level synthesis, modifying the synthesis approach so as to optimize area and performances even while granting autonomous error detection.

We consider, in particular, data path synthesis for systems described by *arithmetic operations* only. This restriction while not forcing basic limitations of our approach, allows us to consider simpler structures adopting a single and cost-effective coding solution throughout the whole circuit and thus avoiding insertion of transcoding operations in the algorithmic flow. We achieve self-checking by use of *arithmetic codes*, e.g., AN codes or residue codes [5]; for such codes it has been already proved that - for some specific computations such as convolution - it is possible to drastically reduce the number of checkers without impairing the detection capability [6,7].

In the present paper, we consider general arithmetic data flow graphs, with unconstrained topology; our scope is to optimize the *number of checkers* to be inserted, so that:

- detectability of errors is maintained (error assumptions such as, e.g., single error will be related to error confinement within suitable subgraphs of the DFG);
- area overhead with respect to the non-self-checking circuit is kept as low as possible, while keeping minimum computation latency;
- length of the control step is kept as limited as possible.

In the following sections, we analyze first of all propagation of detection capacity through a DFG, identifying conditions that lead to *aliasing* (reference will be made to a sample code, although the approach is totally general). Based on these conditions, the concepts of *Detectable Subgraphs* and of *Maximum Detectable Subgraphs* is introduced and techniques for extracting these subgraphs from the DFG are presented; properties

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00 ©1996 IEEE

of such subgraphs are discussed, allowing to consider creation of an *optimal partition* of a given DFG minimizing the number of checkpoints. We extend also a minimum-latency, time-constrained scheduling algorithm to account for optimum scheduling of checkers.

2. A code-related analysis of DFGs

In the present paper, we refer as a running example to single-error detecting arithmetic codes, such as 3N code and residue code in base 3, all capable of detecting single errors in additions, subtractions and multiplications; only these types of operations will be allowable in the DFGs here examined. While for additions and subtractions detection capacity extends to errors present in one of the input operands as well as in the operator, for multiplication it holds with the assumption of correct input operands (i.e., errors are restricted to the operator) [5, 8].

The final architecture supporting the computation consists of arithmetic circuits, of registers storing operands and results and of a switched interconnection network (either multiplexer-based or bus-based); we insert checking operations in the clock cycle following loading of a result in a register, so that an error in a register will be considered as equivalent to an error in the arithmetic circuit generating the result. The interconnection network will be considered fault-free, as in most of the current literature (further considerations on this point, relaxing such restriction, will be discussed in section 5).

Data flow graphs here examined are all DAGs (Directed Acyclic Graphs): it has been proved that it is always possible to represent a computation by such a graph, transferring conditions that determine presence of cycles to the Control Flow Graph. A DFG consists of *nodes* representing *operations* (each node o_j contains a mark identifying the operation type) connected by *edges* representing *data dependencies*: there is an edge e_{ij} from o_i to o_j if and only if results produced by operation o_i constitute an input for o_j. Particular edges (provided with a source node and no sink node) denote *primary outputs*; results produced by a node can be simultaneously primary outputs *and* inputs to other nodes. Whenever the results produced by a node are used by more than one successor nodes, rather than introducing multiple edges we will use a fan-out type of graphical representation.

Our first goal is to identify the conditions - related both to topology of the DFG and to operations performed by the nodes - that lead to *aliasing*, with respect to the adopted error model (in our case, a single error). In other words, considering a single error located in any node of the DFG, and as a consequence of propagation along the paths departing from this node, we wish to determine if at any given point along such path it becomes impossible to detect the error, even though the result produced is not correct. *Aliasing*, as a consequence of which an error-affected result is still a codeword and as such cannot be detected by a checker, should not be confused with *masking*, by which the result produced by a fault-affected unit is still correct.

Conditions for aliasing derive from the specific error syndrome detected by the code; in the examples here chosen, errors on the output of a faulty arithmetic device have the form $\pm 2^i$. Aliasing in a network of arithmetic circuits occurs whenever - in the presence of a single fault - the error syndrome for a result affected by the error is divisible by 3; this may occur either if one input to a multiplier is faulty (it is sufficient for the other input to be divisible by 3) or if reconvergent paths in the DFG lead to a final error syndrome divisible by 3.

The presence of reconvergent paths in the DFG is not a *sufficient* condition for aliasing; for example, if two paths transferring the same error converge on the inputs of one adder, the result produced by this adder will be affected by an error $\pm 2^{2i}$, which is still detectable. Conversely, aliasing occurs if the two paths to the adder's inputs are affected by the syndromes $\pm 2^i$ and $\pm 2^{i+1}$.

To prevent aliasing, and thus grant the self-checking property to a system implementing the DFG, checking has to be performed at suitable points within the computation and corresponding restrictions have to be adopted in scheduling and allocation.

We identify first the *necessary checking points* in the DFG, i.e., the points where it is necessary to verify correctness of the computation in order to guarantee the detectability of all errors in the whole DFG for the adopted error model and code. We define the output of an operation o_j to be *explicitly checked* if it is verified by a checker. It is *implicitly checked* with respect to a explicitly-checked operation o_i if any error (considered by the adopted coding technique) at the output of o_j itself is propagated without aliasing to the output of o_i. Assuming the primary inputs to be error free, necessary checking points must be placed:

- at each primary output of the DFG, since its correctness must be certified as soon as the output is generated in order to guarantee the use of correct data in the activities following the DFG computation;
- at the output of an operation if and only if aliasing occurs at the output of at least one immediate successors of such an operation.

Due to the chosen coding techniques, the second condition requires that the inputs of all multiplications (if they are not primary inputs) must be checked, as well as the output of any addition/subtraction which induces aliasing in at least one addition/subtraction in its immediate successors.

To identify the minimum number of checking points and their position, we formalise the following concepts.

The *Detectable Subgraph* $DS(o_j)$ associated with node o_j of the *DFG* is a subgraph having k inputs and one *checked output* given by the output of operation o_j, such that, assuming the inputs to be checked, any single error within the subgraph is detected by checking the output of o_j. A *DS* may have any number of outputs, but the detection property is asserted with respect to the output of o_j only.

A *Maximum Detectable Subgraph MDS* is a *DS* not completely contained in any other *DS*. Extending at least one input-output path in an *MDS*, i.e., including one additional operation, would lead to aliasing. As a consequence, to grant correctness of the output of an *MDS*, it is mandatory to insert a checking point on such output, being the *MDS* inputs checked.

The single-error assumption within a *DS* or an *MDS* implies that any operation (operand) appearing in one *DS* (*MDS*) is mapped onto a separate operator (register). This does not exclude reuse of operators (registers) in independent *DS's*, provided each instance of use is separately checked.

Creation of all *MDS'*s in a DFG can be obtained by:

Algorithm 1: MDS's creation.

a) create a *levelized DFG*, where
   - all primary inputs have level 0;
   - for any node o_j, if l_j^1 and l_j^2 are the levels of its immediate predecessors, level l_j is computed as $\max(l_j^1, l_j^2)+1$.

b) for each level l (starting with $l=1$) and for each node o_j^l in level l, build the subgraph $DS_M(o_j^l)$ which completely includes all the possible $DS(o_j^l)$'s. Since primary outputs and multiplier inputs *compel* checking, when building $DS_M(o_j^l)$ these points are considered *as if* they were primary inputs: i.e., backward propagation of a path from o_j^l stops when reaching such points. Otherwise, propagation stops as soon as addition of a further node o_k^m creates aliasing.

c) for each $DS_M(o_j^l)$ such that o_j^l does not generate a primary output, check if it is completely included in at least one other $DS_M(o_i^h)$: in such a case, $DS_M(o_j^l)$ cannot be an *MDS* and is marked in consequence.

d) unmarked $DS_M(o_j^l)$'s constitute the set of all possible *MDS'*s for the given DFG. □

Note that, due to the definition of step b, the primary outputs are outputs of *MDS'*s.

*MDS'*s are not necessarily disjoint. As a consequence, checking the outputs of all *MDS'*s may be redundant to detect the presence of errors. Therefore, the number of *MDS'*s constitutes an *upper bound* to the number of necessary checking points.

Theorem 1. Let $DS(o_1)$ and $DS(o_2)$ be two *DS'*s such that $DS(o_1) \cap DS(o_2) \neq \emptyset$. Consider $DS^* = DS(o_1) - DS(o_1) \cap DS(o_2)$. The following can be proved:

1. the inputs to DS^* coming from $DS(o_1) \cap DS(o_2)$ are *implicitly checked*,
2. DS^* is a detectable subgraph provided $DS(o_2)$ (or a *DS* completely including it) is checked as a whole.

Proof. Let e' be an edge connecting DS^* with $DS(o_2)$; by the rules defining both the DFG and the *DS'*s, e' comes from a node o_j in $DS(o_1) \cap DS(o_2)$ belonging simultaneously to at least one path ending on o_1 in $DS(o_1)$ and one path ending on o_2 in $DS(o_2)$. If $DS(o_2)$ is checked as a whole, an error on the output of o_j is detected in the checked output of $DS(o_2)$ itself. Conversely, if the checker associated with o_2 does not signal any error, the data flowing on e' are correct. Therefore, by definition, the output of the operation o_j (i.e., input e') is implicitly checked. Since all inputs of DS^* are either explicitly or implicitly checked, by the definition of detectable subgraph, DS^* is a *DS*. □

The construction rules for *DS'*s grant that data flowing on e' will not re-enter $DS(o_2)$ from DS^*. Theorem 1 can be applied to deal with any number of non-mutually disjoint *DS'*s: it is sufficient to apply it iteratively to pairs of *DS'*s, until disjoint subgraphs are created.

3. Optimum partitioning of the DFG with respect to detection

To guarantee error detection in the DFG, the output of each operation must be checked explicitly or implicitly, i.e., each operation must belong to at least one *DS*.

Lemma 1. A *cover C* of the DFG consisting of a set of *DS'*s (not necessarily disjoint) covering the whole DFG (i.e., such that all nodes of the DFG belong to at least one *DS* in *C*) allows detection with respect to *any single error within each DS in the cover*. It thus constitutes a *detectable cover*.

Proof: Consider any $DS^* \in C$: its inputs are either primary inputs, multiplier inputs, fan-outs of primary outputs - and as such explicitly checked - or else they are fan-outs of nodes belonging to another *DS'*, and as such implicitly checked by checking the output of *DS'*. The checked output of DS^* is *explicitly checked* with respect to any error occurring inside DS^*; as for all other outputs of DS^*, by construction, they derive from fan-out on nodes in DS^* belonging to a path checked by the output of DS^*, and are *implicitly checked*. The DFG being a DAG, no checking ambiguity can occur and each node is checked with respect to single errors within (at least) one *DS*. □

The number of checking points associated with a detectable cover is the cardinality of the cover itself.

A *minimum detectable cover* (*mdc*) is a detectable cover having minimum cardinality among all possible covers. The minimum number of checking points required by a DFG is thus the cardinality of an *mdc* of the DFG itself. Minimum detectable covers are in general not unique, i.e., there may exist different covers characterized by the same cardinality N_C but differing in at least one *DS*.

Identification of an *mdc* should imply the exploration of all possible combinations of *DS'*s covering the DFG. To reduce the search complexity, we prove:

Theorem 2. There is always a minimum detectable cover *mdc* composed exclusively by *MDS'*s.

Proof: Consider an *mdc** in which at least one *DS* is not an *MDS*. Substitute this *DS* with an *MDS* completely including it, i.e., such that each operation in the *DS* belongs to the *MDS*. By construction, such an *MDS* always exists. Substitution does not modify the detecting capacity of the cover since all errors detected at the checked output of the *DS* are still propagated and detected at the output of the considered *MDS*, by its definition. The substitution does not increase the cardinality of the detectable cover: as a consequence, the new detectable cover is still minimum. □

From this theorem, we can derive several consequences. First of all, the covering algorithm identifying the *mdc'*s of the DFG needs to explore the covers composed by *MDS'*s only. Besides, it is possible to specify more clearly and relax the requirement on checking the multiplication inputs: in fact, inputs to a multiplication need to be either implicitly checked by a *DS* (in the chosen cover *C*), which is different from the *DS** to which the multiplication belongs, or else they must be explicitly checked. Anyway, the *DS** including the multiplication cannot include any of its predecessors.

The positions of the necessary checking points for a given *mdc C* are the outputs of all *MDS* belonging to *C*.

Analysis of the *mdc'*s is relevant also with respect to identification of the minimum number of operators of each type strictly required to guarantee detectability, assuming that minimum-latency scheduling is implemented. Within every *DS* of the *mdc*, each resource (operator or register) must be used only once to avoid aliasing; referring in particular to operators, if a *DS* contains *k* instances of an operation of type α, the operations will have to be mapped onto *k* different operators of type α. However, a single operator can be used for mapping two operations belonging to two disjoint *DS'*s without affecting the detection capacity, since a possible error due to a fault in each instance is viewed (and detected independently) as an error in each of the two *DS'*s and does not constitute a multiple not identifiable by the adopted code. This holds for multiple-error detecting codes, by scaling the requirements. Similar reasoning applies to variables an registers.

As a consequence, in an *mdc* composed by disjoint *DS'*s, the minimum number of instances of a given type of resource is equal to the maximum number of instances of such a resource required in any *DS* of the *mdc*.

If the *MDS'*s constituting the cover are all disjoint, scheduling may begin without further processing. If some *MDS'*s are not disjoint, an *mdc* composed only by disjoint *DS'*s is derived by applying the Theorem 1. This disjoint partitioning is necessary since - when scheduling the DFG - each operation will be scheduled in one definite control step and thus it will be associated with one *DS* only. The *mdc* modified so as to consist of disjoint *DS'*s will be used for scheduling the DFG. Guidelines to perform partitioning of non-disjoint *MDS'*s into disjoint *Ds'*s need to take into account the minimization of the number of resources for each type of operation required in the given *mdc*.

As an example, consider the DFG shown in Fig. 1 (see [1,2]). We characterize each node *i* of the DFG with a subscript denoting the operation's nature (addition, subtraction, multiplication) and a superscript denoting the *MDS* to which the operation belongs. The *MDS'*s are:

A: $\left\{1_*^A\right\}$ ***B***: $\left\{2_*^B\right\}$ ***C***: $\left\{6_*^C\right\}$

D: $\left\{3_*^D,4_-^D,5_-^D,7_*^D\right\}$ ***E***: $\left\{8_+^E,9_-^E\right\}$ ***F***: $\left\{10_*^F,11_-^F\right\}$

The covering table of the DFG has a column for each node and a row for each *MDS*. In our example, it is:

- all *MDS'*s act as "essential implicants" in a cover; the *mdc* is thus unique and it requires six checking points;
- all *MDS'*s are mutually disjoint; thus, each node is attributed to one and only one *MDS* and the *mdc* consists of all and only the *MDS'*s.

As a consequence of these characteristics and of the fact that within each distinct *DS* used in the final partitioning of the DFG each operation must be instantiated by a distinct physical operator (operators may be reused between *different DS'*s), the minimum number of resources is 2 multipliers, 2 subtractors and 1 adder.

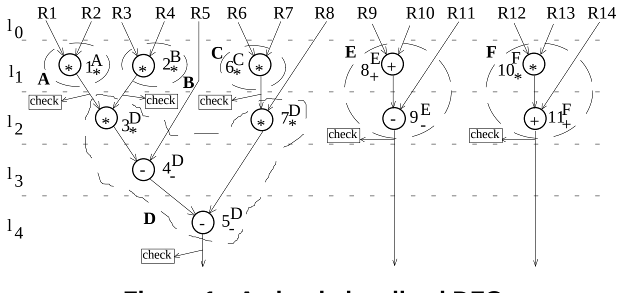

Figure 1 - A simple levelized DFG.

4. Scheduling the DFG with the detectable capability

The scheduling approach we suggest is an extension of a time-constrained scheduling algorithm that takes into account the further constraints introduced by the *mdc* and the associated *DS*'s. To this purpose, we need to introduce a basic assumption on the control step in which the checking operation associated with the output of an operation o_j is scheduled. If the output of o_j is a primary output, the checking operation must be scheduled in the same control step as o_j; otherwise, it is scheduled in the control step immediately following that in which o_j itself is scheduled. In other words, there is no freedom on the relative positioning of o_j and of its checking.

The number N_C of checking points in the *mdc* is the *upper bound* for the number of checkers. To identify a *lower bound* n_C for this number, we explore of the original DFG; there need to be at least as many checkers as the maximum number of checking points located at the same level on the critical paths (i.e., input-output paths of maximum length).

A problem of obvious interest concerns the relation between the number N_C of checking points and the number n_C of checkers after the scheduling has been completed. To this effect, we can prove:

Theorem 3. Given two *mdc*'s mdc_1 and mdc_2, being N_C their cardinality, denoting by S the number of control steps for minimum-latency scheduling and by ν the number of checkers required by mdc_1, if $N_C = \nu S$ then mdc_2 cannot generate a scheduling with a lower number of checkers.

Proof: The proof is by contradiction. Assumption $N_C = \nu S$ means that all checkers are used in each control step. To reduce the number of checkers, mdc_2 should use at most ν-1 checkers in each control step. Since the number of checking operations is still N_C, we have $N_C \leq (\nu - 1)S$, which contradicts the assumption. □

For the example in Fig. 1, it is $N_C=6$; there are two critical paths (from 1 to 5 and from 2 to 5) with two checking points at level 2 (on the inputs of node 3), thus $n_C=2$. The length of the critical paths is 4.

To proceed with scheduling, we perform first both ASAP and ALAP schedulings, by taking into account the checking points together with the other operations. The symbol denoting an operation is in bold if a checking point is associated with that operation. S_i denotes the i-th control step; each line gives the operation scheduled on the same control step. The ASAP scheduling is:

S_1: $\mathbf{1}_*^{\mathbf{A}}$ $\mathbf{2}_*^{\mathbf{B}}$ $\mathbf{6}_*^{\mathbf{C}}$ 8_+^{E} 10_*^{F}

S_2: 3_*^{D} 7_*^{D} $\mathbf{9}_-^{\mathbf{E}}$ $\mathbf{11}_+^{\mathbf{F}}$

S_3: 4_-^{D}

S_4: $\mathbf{5}_-^{\mathbf{D}}$

The cost is 4 multipliers, 2 subtractors, 1 adder, and 3 checkers. The ALAP scheduling costs 2 multipliers, 2 subtractors, 1 adder, 3 checkers, being given by:

S_1: $\mathbf{1}_*^{\mathbf{A}}$ $\mathbf{2}_*^{\mathbf{B}}$

S_2: 3_*^{D} $\mathbf{6}_*^{\mathbf{C}}$

S_3: 4_-^{D} 7_*^{D} 8_+^{E} 10_*^{F}

S_4: $\mathbf{5}_-^{\mathbf{D}}$ $\mathbf{9}_-^{\mathbf{E}}$ $\mathbf{11}_+^{\mathbf{F}}$

Mobility is evaluated for all operations, *including checking*. At this point, a force-directed scheduling algorithm is adopted, by taking into account checking as well as all other operations and by adopting the lower bounds derived from the *MDS*'s and from the DFG as limit goals for the number of resources. This leads to:

S_1: $\mathbf{1}_*^{\mathbf{A}}$ $\mathbf{2}_*^{\mathbf{B}}$

S_2: 3_*^{D} $\mathbf{6}_*^{\mathbf{C}}$ $10_*^{\boldsymbol{F}}$

S_3: $4_-^{\boldsymbol{D}}$ $7_*^{\boldsymbol{D}}$ $8_+^{\boldsymbol{E}}$ $\mathbf{11}_+^{\mathbf{F}}$

S_4: $\mathbf{5}_-^{\mathbf{D}}$ $\mathbf{9}_-^{\mathbf{E}}$

The final cost amounts to 2 multipliers, 2 subtractors, 1 adder and 2 checkers - i.e., the minimum theoretical cost possible, as seen previously. It is worthwhile noting that the minimum cost *without error detection* would have amounted to 2 multipliers, 1 adder and 1 subtractor [2]; inserting checkers on the structure defined in a conventional way wouldhave required one checker for each operator, thus a total of four checkers. Given circuit complexity of the operators involved, addition of one subtractor by use of our technique is well offset by reduction by two of the number of checkers.

Consider the more complex example of Fig.2, in which all problems discussed in the present paper are examined. The *MDS*'s derived from this DFG are:

A: $\{2_*, 6_-, 9_+\}$ **B**: $\{2_*, 3_*, 6_-, 10_-\}$ **C**: $\{4_-, 7_+, 11_-\}$

D: $\{1_*, 2_*, 5_+, 8_+, 12_-, 14_+\}$ **E**: $\{4_-, 7_+, 13_*, 15_+\}$

Again, all *MDS*'s are essential and the minimum cover is unique: $Nc=5$. Since there are two primary outputs at the

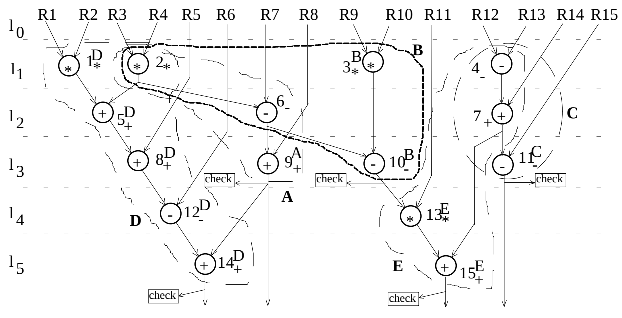

Figure 2 - A levelized DFG with a non-disjoint mdc.

end of critical paths (no other checking points at the same level on critical paths are found), it is $n_c=2$.

We have here a complex case of intersecting *MDS*'s: several disjoint covers can be considered. The first one adopts *MDS*'s **A** and **C**, and the following *DS*'s:

B': $\{3_*, 10_-\}$ **D'**: $\{1_*, 5_+, 8_+, 12_-, 14_+\}$ **E'**: $\{13_*, 15_+\}$

The minimum-latency scheduling requires five control steps; the minimum operator cost, derived exclusively from the chosen *DS*'s, amounts to 1 multiplier, 2 subtractors, and 3 adders. Examining the DFG allows to increase the minimum number of multipliers to 2, since there are two multipliers at level 1 on critical paths. The results of our scheduling is:

S_1: $1_*^{\boldsymbol{D}}$ $2_*^{\boldsymbol{A}}$ $4_-^{\boldsymbol{C}}$

S_2: $3_*^{\boldsymbol{B}}$ $5_+^{\boldsymbol{D}}$ $6_-^{\boldsymbol{A}}$ $7_+^{\boldsymbol{C}}$

S_3: $8_+^{\boldsymbol{D}}$ $\mathbf{9}_+^{\mathbf{A}}$ $\mathbf{10}_-^{\mathbf{B}}$

S_4: $12_-^{\boldsymbol{D}}$ $13_*^{\boldsymbol{E}}$ $\mathbf{11}_-^{\mathbf{C}}$

S_5: $\mathbf{14}_+^{\mathbf{D}}$ $\mathbf{15}_+^{\mathbf{E}}$

The total cost amounts to 2 multipliers, 2 subtractors, 3 adders, and 2 checkers, namely, the theoretical minimum for the chosen self-checking solution.

Consider now a second cover, consisting of **A, B', D',** as before, while **E** is taken in full and **C''**=$\{11_-\}$ is adopted. A scheduling requiring only 1 adder but 3 checkers can be identified: the cost can be considered equivalent to that of the previous one.

Note that an optimum scheduling unconstrained by detectability requires 2 multipliers, 2 adders and 1 subtractor; adding individual checkers would have introduced *5* checkers. Again, the *MDS*-based solution lowers the total cost.

5. Conclusions

The approach for high-level synthesis of arithmetic DFG's has been discussed here with reference to adoption of single-error detecting codes; in this frame, the single-error assumption is restricted to appearance within the individual *DS* (and, when allocation is performed, within the unit implementing it). Extension to multiple-error detecting solutions is straightforward, following the same guidelines (in particular, the error model will again be related to the individual *DS*).

Initially, we introduced a fault assumption by which only errors located in arithmetic devices and in registers were taken into account. Actually, most errors affecting the interconnection network can be considered as well, by a proper fault collapsing. In fact:

1. faults affecting the wiring between registers and adders/subtractors are implicitly checked as if they affected the register or the arithmetic device;
2. faults affecting the wiring from the output of a multiplier to a register may be associated either with the register or with the multiplier; errors on the wiring from the checked inputs to a multiplier are not checked - thus, they constitute part of the system's *hard-core*;
3. if a multiplexer-based approach is adopted for the interconnection network, faults can be collapsed as in the previous points; faults affecting a multiplexer on a multiplier's input can be collapsed with faults in the source register provided checking is performed at the outputs of the multiplexer instead than at the outputs of the register.

All the above, obviously, involves extending the single-error assumption to a suitably defined subsystem, including, together with operators and registers associated with each *DS*, segments of wiring as well.

Choice of coding influences area required by arithmetic devices and registers as well as clock cycle. In general, the choice is between separate and non-separate codes. Usually, separate codes require larger area but preserve the cycle length requested by non-checking operations; conversely, non-separate codes are often supported by lower-complexity circuits but - by increasing the width of the word upon which the arithmetic device operates - lead to increase of the clock cycle length.

Further problems to be studied concern first of all considering different operation latencies (in particular, for adders/subtractors with respect to multipliers); extension of our approach to arithmetic/logical DGF's, involving choice of different codes and ensuing conditions for detectability, is also being examined. Finally, the impact of the self-checking solution on *allocation* must be examined, in view of optimal allocation. Work is at present going on along these lines.

References

[1] D.Gajski, N.Dutt, A.Wu and S.Lin: *High-Level Synthesis*, Kluwer Academic Publishers, Boston, MA, 1992

[2] G. de Micheli: *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, 1994

[3] N.K.Jha, S-J.Wang: "Design and Synthesis of Self-Checking VLSI circuits", *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 12, No.6, pp.879-887, June 1993

[4] C.Bolchini, R.Montandono, F.Salice, D.Sciuto: "Self-checking FSM's based on a constant-distance state encoding", *proc. IEEE DFT 95*, Lafayette, USA, Nov. 1995

[5] T.R.N. Rao: *Error coding for arithmetic processors*, Academic Press, NY, 1974

[6] V. Piuri: "Fault-tolerant systolic arrays: an approach based upon residue arithmetic", *Proc. ARITH-8*, Como, Italy, 1987

[7] V. Piuri: "Fault-tolerant array processors: an approach based upon A*N codes", *Proc. ISCAS'88*, Helsinki, Finland, 1988

[8] M.Annaratone, R. Stefanelli: "A multiplier with multiple error correction capability", in *Proc. IEEE Arith 6*, 1983