

High-level Synthesis of Data Paths with Concurrent Error Detection

Anna Antola, Vincenzo Piuri, Mariagiovanna Sami
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza L. Da Vinci 32, 20133 Milano, Italy
e-mail: {antola,piuri,sami}@elet.polimi.it

Abstract

High-level synthesis of data paths with concurrent self-checking abilities is discussed to balance redundancy, latency, and checking effectiveness. The nominal and the checking computations are scheduled and allocated contemporaneously by using a force-directed approach to limit the number of redundant units required to achieve detection within the latency of the nominal computation only. Resource sharing between the nominal and the checking computation is used to minimise the redundancy, while keeping error aliasing as reduced as possible.

1. Introduction

Introduction of fault-tolerance capacities in ASICs or in application-specific cores has received much interest recently; in particular, high-level synthesis techniques that lead to fault-tolerant designs have been proposed. Solutions granting self-checking capacities with respect to different types of faults [1-4] or even tolerance to faults [5,6] have been presented in the literature. A common aim – considering applications in embedded systems where safety requirements are accompanied by cost limitations – is to achieve self-checking or fault-tolerance with lower area increase than that required by straightforward redundancy. Architectures taken into account belong to the “FSM with Data Path” class, attention being focused on design of the Data Path (different techniques being best suited to FSM design, see e.g. [7]).

In the present paper, we propose an approach to design of data paths capable of concurrent self-checking by duplicating and comparing the nominal computation at a limited cost, i.e., without duplicating all units in the circuit. Results produced by the data path are checked in such a way as to grant error detection with no aliasing possibility; if we consider the data path as the implementation of a specified *process*, it might be said that the level of concurrently corresponds to the complete process rather than to the individual operation. Code-based solutions have been previously suggested for such a problem ([3,10]); in such case, redundancy is present through extension of the codewords and introduction of a number of checkers, aliasing being referred to the code’s capacity of detection. Use of codes introduces a further problem concerning the choice of best-suited coding approach; in fact, the mentioned proposal deals with *arithmetic* operations only. An alternative approach, presented in [4,11], avoids use of coding and adopts a *semi-concurrent* self-checking scheme based on repeated

execution of operations by different units (as done also in [8]); scheduling of the “checking DFG” over a larger number of control steps than that required for the “nominal DFG” and resource sharing allow to reduce the total number of functional units with respect to straightforward duplication. An initial – optimum – design of the nominal data path is taken as the starting point for subsequent synthesis.

The concurrent self-checking synthesis approach we present here, on the contrary, affords simultaneous synthesis of both nominal and checking DFG, aiming at reducing resource requirements while avoiding any risk of aliasing. To this end, a modified *force-directed* technique is used [9], such that at each step operations are not only scheduled but also allocated, and that register and checker allocation is performed as well. Hardware reduction with respect to the complete duplication with comparison is achieved by exploiting the possibilities of resource sharing in the scheduling algorithm; checking points are introduced only when sharing may induce aliasing. The solution is presented with reference to linear-code DFGs; actually, extension to the case of branches and loops is straightforward, since concurrency does not lead to the type of problems envisioned for semi-concurrent self-checking. Reference architecture and fault model are presented in section 2; the scheduling-and-allocation technique is discussed in section 3.

2. Fault model and reference architecture

The *fault model* adopted in the present paper considers presence of a *single fault* in the whole data path; faults may appear in functional units and in registers (initially at least, the interconnection network is assumed fault-free). To provide a *reference architecture* against which the one obtained by means of our synthesis technique will be evaluated, the *nominal architecture* is initially designed with a standard approach granting time and area requirements. Latency adopted for this nominal design will be the same used for the self-checking circuit. The *reference self-checking architecture* is obtained by straightforward duplication of the nominal one; checking is performed on each primary output. The number of primary outputs is the initial upper bound to the number of checkers required (no sharing of checkers between checking operations is adopted). Probability of *aliasing*, i.e., that presence of an error may go undetected, is null since no sharing is envisioned between nominal and checking data paths; not only do replicated executions of any operation – or replicated storage of any variable – always involve different units, but also replicas of any execution path within the two data paths always involve separate sets of units. Cost of the reference architecture for the self-checking data path is immediately obtained; as for the controlling FSM, it may be noticed that – given totally concurrent self-checking – the number of states is not modified with respect to the non-self-checking instance.

The design approach we now propose *allows resource sharing* between nominal and checking data paths, *provided no aliasing may result*. This last requirement may lead to necessity of extending the set of resources, either by replicating functional units or registers (this is inevitable, e.g., when one instance only of a particular unit is present in the nominal data path) or by introducing *intermediate checking operations* during the execution flow. As a starting point for high-level synthesis, a self-checking DFG is first created by duplicating the nominal one and by attaching checking operations at the primary outputs of the nominal DFG instance (see Fig. 1). This is then treated as a single DFG that needs to be scheduled and allocated, being the checking operation viewed as regular operations. Obviously, the final architecture will have replicated sets of inputs (“nominal” and “checking”) and of primary outputs.

To avoid aliasing, some conditions must be satisfied. The immediate one – that no operation be performed by the same unit nor variable stored in the same register in the two replicated computations – is *necessary* to avoid aliasing, but not sufficient; a complete statement requires that at any given point in the execution flow, the set of operations and variables along the path going from *checked variables* to a given edge (i.e., to the related variable) in nominal and checking data paths do not involve shared functional units or registers. Variables directly verified by checking operations and primary inputs are checked variables. To grant such condition, either redundant units may be added or checking must be performed *at the last point in the execution flow* before sharing occurs.

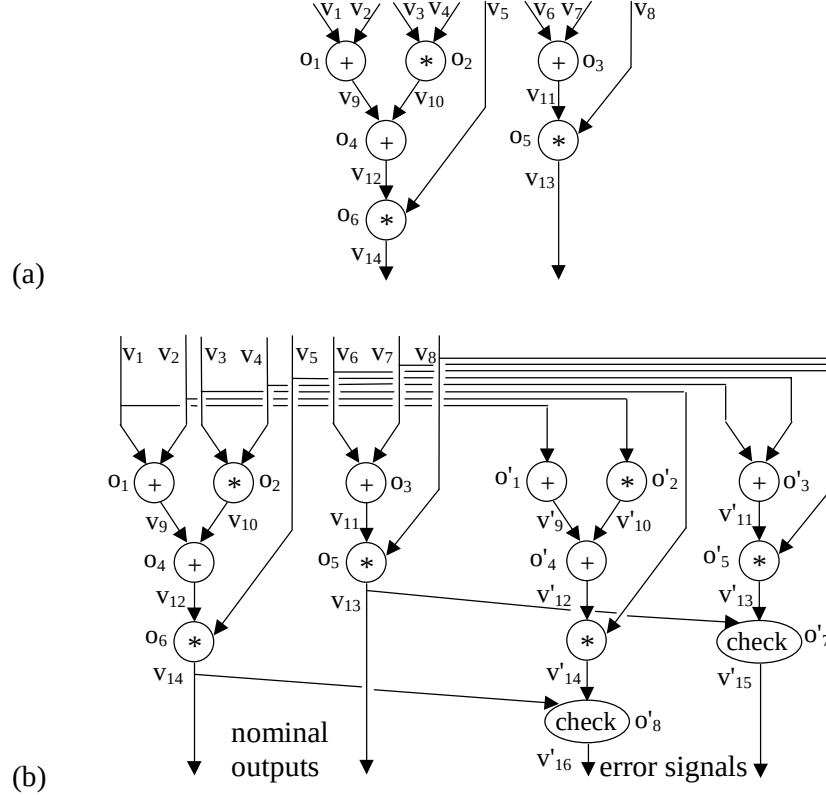


Figure 1: The nominal (a) and the self-checking (b) DFGs.

3. The scheduling and allocation approach

To perform scheduling and allocation of the operations both in the nominal and the checking DFGs as well as the allocation of variables, we propose a novel algorithm based on a force-directed scheme.

A time-constrained approach is assumed, maximum latency L being defined so as to include latency of checking operations. For each operation o_i (including the checking ones on the DFG outputs), mobility m_i is first evaluated. For checking operations, since checker's outputs must be available for external observation and DFG validation at the end of the L -th step, the mobility of the checker is taken as the distance from the ASAP label of the operation generating such output to the L -th step.

The basic force-directed scheduling approach aims at balancing resource costs through the control steps by distributing operations as equally as possible along them. The technique is based on a probabilistic analysis of the operation schedule and the related probabilistic cost. Probability p_{ji} of scheduling operation o_i in step s_j is computed as $p_{ji} = 1/(m_i+1)$. The initial probability diagram for the example of Fig. 1 is shown in Fig. 2.

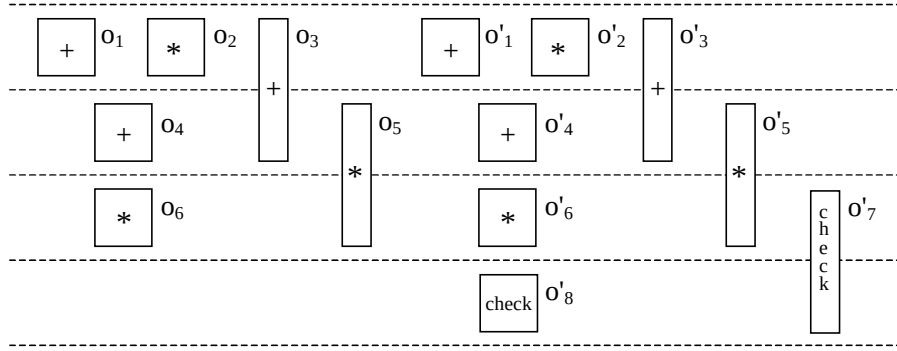


Figure 2: The operations' scheduling probability diagram (for latency $L = 4$).

Operation allocation is interleaved with the scheduling steps: whenever an operation can be allocated since the aliasing condition can be verified, also the variables holding its inputs are allocated. To select the resources (functional units and registers) for allocation while taking into account the aliasing issue, the set a_i of preceding resources (*predecessors*) is associated to an operation o_i . An o_i predecessor is a functional unit or a register (except the ones holding the immediate inputs of the operation itself) used in the data path from the nearest checked variables to the operation o_i itself. The functional unit f_i and the register r_i on which the operator o_i and the variable v_i are allocated must not belong to a_i .

Since we aim at using the technique for allocation as well as for scheduling, and for variables as well as for operations, we create from the start a diagram explicitly showing lifetimes of all variables present in the self-checking DFG, based on *mobility of source and destination operations*. We introduce to this end the concept of *possible life* of a variable v_i as the time elapsed from the earliest step s_m in which the operation generating v_i can be scheduled in its current mobility to the farthest step s_M in which the last operation using v_i can be scheduled in its current mobility. An example of the diagram of the possible variable life is shown in Fig. 3.

All operations having probability equal to one must be scheduled in the given step in order to complete the computation within the desired latency L . Such operations could be allocated if no detection issue was considered. However, since the anti-aliasing condition includes analysis of all *resource predecessors* of an operation (both functional units and registers), allocation of scheduled operations must be performed progressively from the primary inputs to the outputs as soon as it is possible.

One at a time, we take into account for possible allocation all the scheduled operations as yet not allocated, starting from the ones nearest to the primary inputs. We repeat the analysis and, possibly, the allocation as long as there are operations that satisfy the above conditions. For each of these operations, one at a time, we identify a feasible allocation onto the available functional units; unless the operation is directly connected to primary inputs or to *checked variables*, the anti-aliasing criterion must be considered when performing the allocation.

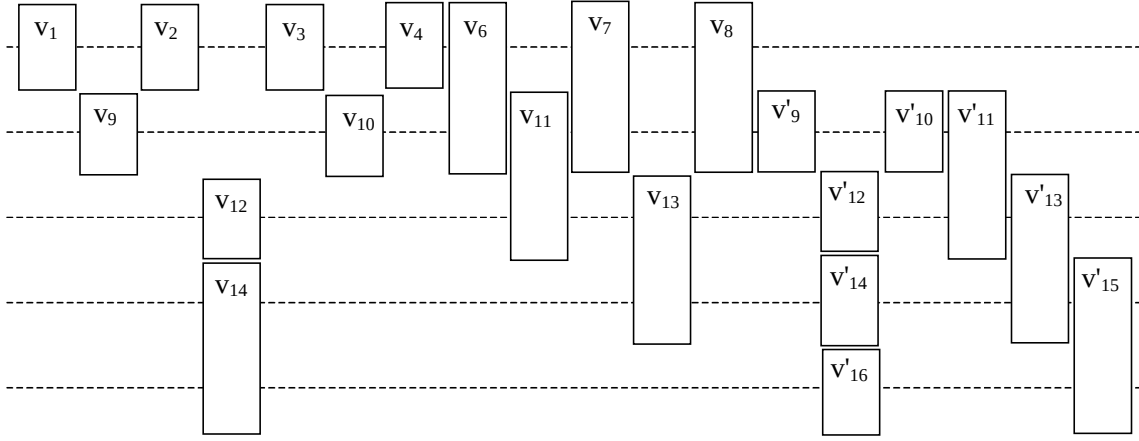


Figure 3: The variables' possible life diagram (for latency $L = 4$).

If no functional unit is available, a new one is introduced in the self-checking architecture and is used for the considered operation o_i .

If there is at least one available unit but it does not satisfy the anti-aliasing condition, we have two options. The simplest one is to introduce a new functional unit on which we allocate the operation: since such unit has never been used before, aliasing is avoided. In practice, this may be just a temporary solution (aliasing occurrence may be only delayed to a subsequent step), and anyway introduction of additional functional units becomes rapidly too expensive. The alternative solution consists in inserting a checking operation on that input of operation o_i placed on the path that induces aliasing. Checking allows to verify correctness of the intermediate result that can thus be delivered to the subsequent operations independently from the predecessors which generated it. In practice, from now on a *checked input* can be treated as a primary input; as a consequence, the set of predecessors associated with the checked input to o_i is reset to the empty set.

Checking operations thus introduced are equivalent to the ones on the DFG outputs, i.e., they need to be scheduled and allocated by using the same criteria of the other operations (except the anti-aliasing criterion since the checkers are assumed to be self-checking). This clearly motivates the necessity and the effectiveness of the interleaved execution of scheduling and allocation. Mobility of these new checking operations is evaluated as for the checking operations on DFG outputs: it starts from the step in which the intermediate output to be checked is generated down to the step L . The scheduling probability of these operations is computed exactly as the other operations. (Note that the number of checkers required may increase with respect to the reference architecture).

At this point, the variables holding the inputs of a scheduled operation o_i have a *certain* life, not only a possible life: we can therefore allocate their variables on the free registers according to the anti-aliasing condition, if any; otherwise, we can introduce a new register or a checking operation as for the operations.

After each variable allocation, it is necessary to update the diagram of the possible variable life in order to support the analysis for the subsequent variable allocations, while operation mobilities are not changed since we are considering now scheduling and allocation of operations having zero mobility.

Allocation of a zero-mobility operation with allocated predecessors may, in turn, allow allocation of other zero-mobility operations whenever all predecessors of such operations have been allocated. The previous operations concerning allocation of zero-mobility operations and

the related input variables are therefore repeated until none of such operations remain unallocated.

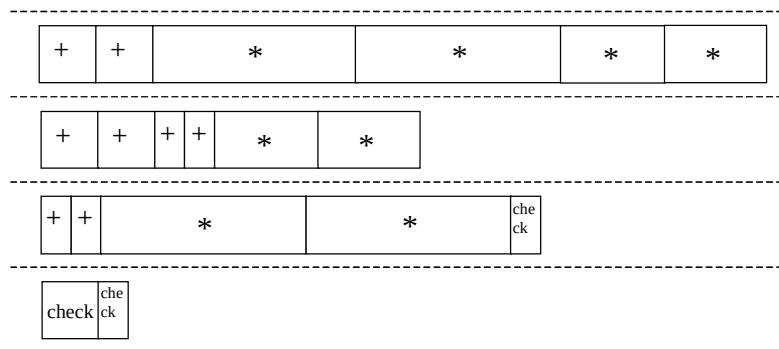


Figure 4: The allocation cost diagram.

The proposed approach can therefore be summarised as follows:

Force-directed scheduling and allocation

1. evaluate operation mobilities;
2. evaluate the probability of scheduling the operations in each control step;
3. evaluate the maximum possible life of the variables from the current diagram of the mobilities;
4. schedule all “ready” operations with zero mobility;
5. as long as there is a scheduled but not yet allocated operation having all predecessors already allocated, repeat:
 - 5.1. allocate one of such operations;
 - 5.2. allocate its input variables;
 - 5.3. if the allocation of the operation or one of its input variables leads to aliasing, choose one of the following solutions:
 - 5.3.1. introduce a new corresponding resource and allocate the operation/variable on it;
 - 5.3.2. introduce a checking operation on the suitable input of the edge of the DFG that determines aliasing;
 - 5.4. update the life diagram of the variables;
6. update operation mobilities and probability of operation scheduling;
7. if there is an unscheduled operation, then:
 - 7.1. select an unscheduled operation by using the force-directed strategy;
 - 7.2. schedule this operation;
 - 7.3. update operation mobilities;
 - 7.4. go to step 5.

4. Conclusions

In this paper, an innovative approach to high-level synthesis of data paths with concurrent error detection ability is presented by taking into account optimisation of all resources (functional units, checkers, and registers) within the given maximum acceptable latency by resource sharing. The nominal computation is duplicated and compared to detect the presence of errors. Scheduling and allocation of all resources are interleaved to identify a minimum-cost solution since allocation introduces checking operations on intermediate results to be scheduled in order to avoid aliasing. Complete duplication of the resources is avoided by exploiting the ones which are idle in each scheduling step.

In Fig. 5, the difficultly-optimizable case of the fifth-order elliptical filter is shown by assuming that the application imposes a maximum acceptable latency equal to 20 clock cycles;

3 adders, 1 checker, and 28 registers (including the 13 input registers) are required by the proposed algorithm, while pure duplication with comparison needs 4 adders, 1 checker, and 27 registers (being registers often smaller than adders, especially for carry-look-ahead and conditional-sum adders).

References

- [1] A. Orailoglu, R. Karri: "Automatic Synthesis of Self-Recovering VLSI Systems", *IEEE Trans. Comp.*, vol. 45, n. 2, Feb. 1996, pp. 131-142
- [2] S.N. Hamilton, A. Orailoglu: "Concurrent error recovery with near-zero latency in synthesized ASICs", *Proc. DATE'98*, Paris, France, Feb. 1998, pp. 604-609
- [3] A. Antola, V. Piuri, M.G. Sami: "Optimising High-Level Synthesis for Self-Checking Arithmetic Circuits", *Proc. IEEE DFT'96*, Boston, MA, USA, November 1996
- [4] A. Antola, V. Piuri, M.G. Sami: "A Low-Redundancy Approach to Semi-Concurrent Error Detection in Data Paths", *Proc. DATE'98*, Paris, France, Feb. 1998
- [5] G. Buonanno, M. Pugassi, M.G. Sami: "A High-Level Synthesis Approach to Design of Fault-Tolerant Systems", *Proc. 1997 IEEE VLSI Test Symposium*, Monterey, CA, USA, Apr. 1997
- [6] A. Orailoglu, "On-line fault resilience through gracefully degradable ASICs", *Journal of Electronic Testing: Theory and Applications*, vol. 12, 1998, pp. 145-151
- [7] C. Bolchini, R. Montandon, F. Salice, D. Sciuto: "Self-checking FSM's based on a constant-distance state encoding", *IEEE Proc. DFT 95*, Lafayette, USA, Nov. 1995
- [8] B. Iyer, R. Karri: "Introspection: A Low Overhead Binding Technique During Self-Diagnosing Microarchitecture Synthesis", *Proc. Design Automation Conference 96*, Las Vegas, June 1996, pp. 137-142
- [9] P.G. Paulin, J.P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs", *IEEE Trans. on CAD-ICS*, vol. 8, n. 6, June 1989, pp. 661-679
- [10] A. Antola, V. Piuri, M. Sami, "A High-Level Synthesis Approach to Optimum Design of Self-Checking Circuits", *Proc. European Design Automation Conference EURO-DAC'96*, Ginevra, Svizzera, settembre 1996, pp. 382-387
- [11] A. Antola, V. Piuri, M. Sami, "Semi-concurrent error detection in data paths", *Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems DFT'97*, Parigi, Francia, ottobre 1997, pp. 298-306

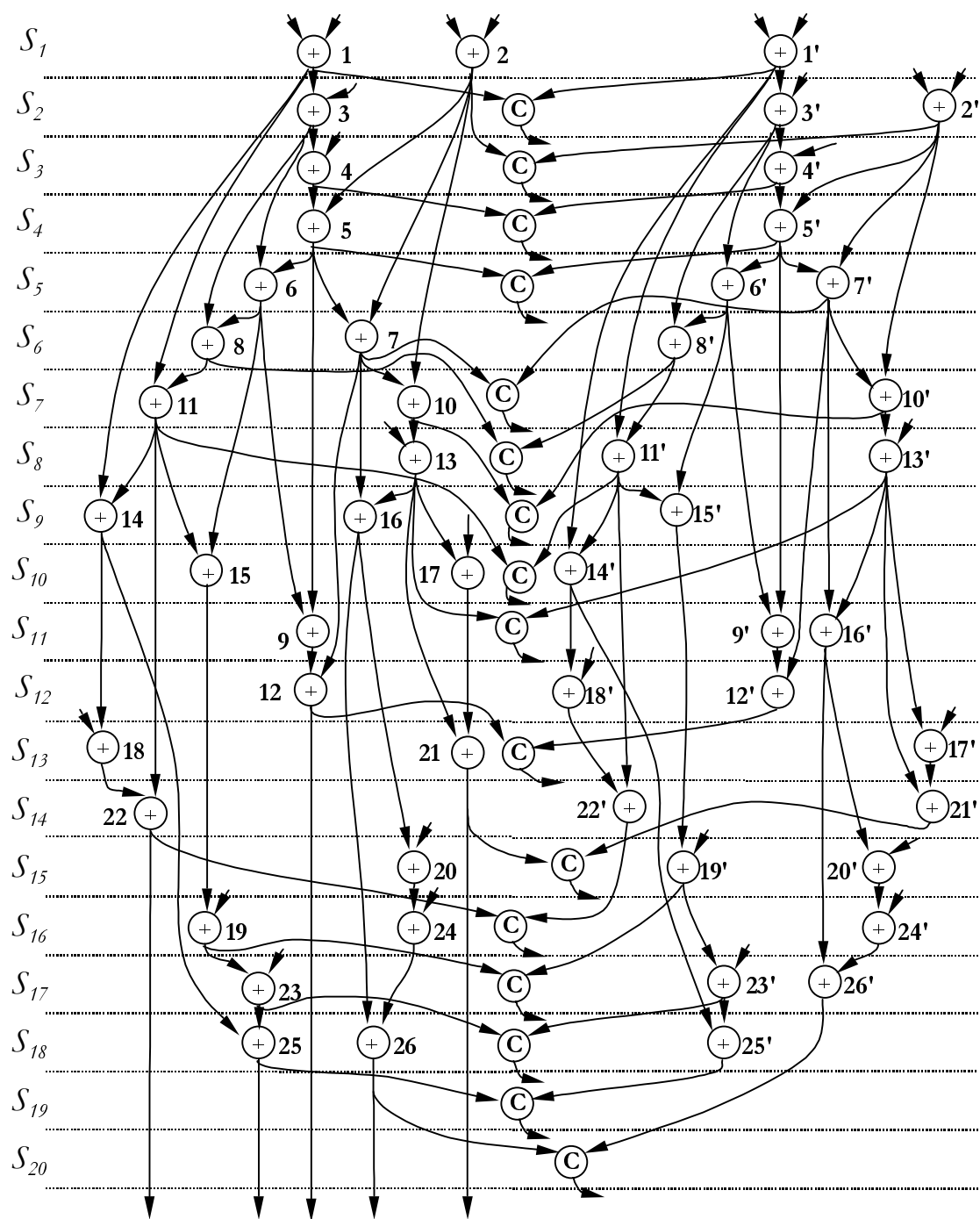


Figure 5: The fifth-order elliptical filter.