# Assignment of Storage Values to Sequential Read-Write Memories

Sabih H. Gerez* and Erwin G. Woutersen†
Department of Electrical Engineering, University of Twente, The Netherlands
email: s.h.gerez@el.utwente.nl, wouter@natlab.research.philips.com

## Abstract

*Sequential read-write memories (SRWMs) are RAMs without an address decoder. A shift register is used instead to point at subsequent memory locations. SRWMs consume less power than RAMs of the same size. Algorithms are presented to check whether a set of storage values fits in a single SRWM and to automatically map storage values in as few SRWMs as possible. Benchmark results show that good assignments can be obtained in spite of the limited addressing capabilities.*

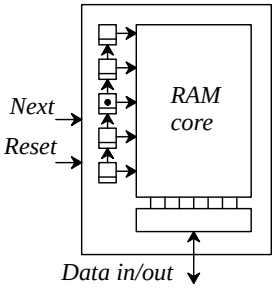


**Figure 1. A sequential read-write memory.**

## 1. Introduction

High-level synthesis, the automatic mapping of an algorithmic description of some computation to a description at the register-transfer level, is normally tackled by dividing the problem into a number of subproblems and solving them sequentially (see e.g. [3, 4]). These subproblems include *scheduling* and *assignment*. Scheduling maps an operation in the algorithmic description to a time instant and assignment deals with the mapping to computational units, storage units and interconnection.

This paper addresses the memory assignment problem for the case that a schedule has already been fixed. In general, it will be necessary to store intermediate results in some way in order to satisfy the result of scheduling. These intermediate results are called *storage values* [16] and it is part of the assignment task to map these values on memory elements available within an architectural model.

Normally *registers* and *register files* are used for the purpose of storage in a data path (see e.g. [6]). Registers have a single memory location while register files have multiple locations identified by an address (similar to a RAM). Recently, *queues*, *stacks*, and *bidirectional queues* have been proposed as alternative memory elements by Aloqeely and Chen [1]. They consist of structures of shift register chains in which the writing and reading of data occurs in first-in-first-out order (queue: write in first register and read from last register in chain), last-in-first-out order (stack: write to and read from first register only) or a combination of both (bidirectional queue: write to or read from either the first or the last register in the chain). Another variation of these structures is the *circular FIFO* proposed by Bennour and Aboulhamid [2] (this structure is similar to a queue with the additional property that the data from the last register can be fed back to the first one).

This paper considers yet another type of memory element: the *sequential read-write memory* (SRWM). It can be seen as "RAM without address decoding". Instead of an address decoder there is a shift register. Only one of the bits in this register is high and it is used as a pointer to a memory location to which data can be written or from which data can be read. A control signal (`next`) can make this bit point to the next higher memory location in the next control step and another control signal (`reset`) can reset the shift register such that the high bit points at the lowest memory location. This has been visualized in Figure 1. These memories have been designed and used by Heubi et al. [8, 9]. Obviously, the read-only version of such a memory is well suited for the storage of multiplication coefficients in the re-

*The work reported in this document was initiated when the author was an *academic guest* at the Institute of Microtechnology, Neuchâtel, Switzerland in the months June and July of 1994. The author is grateful to the Ecole Polytechnique Fédérale de Lausanne, Switzerland for providing the financial support for the visit.

†Author's current affiliation: Philips Research Laboratories, ED&T Synthesis, Eindhoven, The Netherlands.

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

alization of DSP algorithms (they are read repetitively in a fixed order). The read-only version can also be adapted for the realization of controllers [10].

Compared to *random access memories*, sequential memories are far more constrained for the storage of intermediate values produced and consumed in the data path. They might still be considered to be used as a *read-write memory*, because they are simpler: as there is no address decoding, only few control signals are necessary (instead of an address) which also leads to a simplification of the controller.

SRWMs have a similar behavior as the structures proposed by Aloqeely and Chen, but are more interesting as they are generally smaller and use less power (the data is, for example, not moved around in every control step). A study comparing SRWMs and shift-register structures for area and power consumption has been reported by Heubi [8]; it turns out that SRWMs are better in both aspects once that the memories are sufficiently large (in word length and number of memory locations). When a common RAM is compared with an SRWM for the same number of memory locations, the SRWM is smaller and uses less power provided that the number of memory locations is sufficiently large. The gain comes from the shift register whose size grows linearly with the number of memory locations, while an address decoder grows more than linearly.

Storage values that do not have read or write conflicts can be stored in a single RAM without bothering about the address at which they will reside: all addresses are equally accessible. This is not the case for an SRWM: the read and write times of storage values restrict the address to which a specific value can be mapped. The first problem discussed in this paper deals with the problem of determining whether a specific set of storage values fits in a single SRWM and providing the address for each storage value when there is a fit. The second problem is to partition an arbitrary set of storage values in a minimal number of subsets such that each subset can be mapped on a single SRWM. A secondary goal is to minimize the total of number memory locations used. The first problem is solved exactly by a backtracking algorithm. This algorithm is later used as a subroutine of a heuristic that solves the second problem.

The memory elements discussed in this paper should be seen as an alternative to memory elements that are traditionally used. There may be circumstances in which they provide the best design solution. Therefore, a tool as the one described here, should be seen as part of a toolbox that also contains synthesis tools for other memory types.

The structure of this paper is as follows. First two combinatorial optimization problems related to the use of SRWMs in high-level synthesis are formulated. Then algorithms are proposed to solve these problems. The final part of the paper consists of the presentation of experimental results obtained by these algorithms and some conclusions.

## 2. Formulation of the problems

The application domain addressed in this paper is digital signal processing, characterized by the (almost) infinite iteration of some algorithm. Such an algorithm is characterized by a parameter $T_0$ called the *iteration period* [7]. All computations are repeated every $T_0$ control steps, where a control step corresponds to a single period of a system clock. This does not mean that all operations belonging to a single execution of an algorithm need to be performed within $T_0$ control steps. Assuming an *overlapped scheduling* [15] model, the *schedule span* can be larger than $T_0$.

One iteration period covers the control steps $T = \{0, 1, \ldots, T_0 - 1\}$. The result of a scheduling algorithm, containing the mapping in time of a single iteration, will, however, have references to control steps in $Z$ (the set of all integers). The storage values are part of the set $S = \{s_0, s_1, s_2, \ldots\}$. For each value $s \in S$, $\omega(s)$ is the write time, the time when the value should be stored in memory: $\omega : S \rightarrow Z$. A value can be read more than once after having been written. The time instances at which this happens are indicated by the function $\rho : S \rightarrow 2^Z$, where $2^Z$ is the *power set* of $Z$ (the set of all its subsets).

In the rest of the analysis, two different clocking schemes are considered. In one, reading and writing takes place in the same clock phase. In the other, a control step has distinct clock phases for reading and writing. The two schemes will be referred to as the *single-phase* and *multiple-phase* clocking models respectively.

The *latest read time* for a storage value is given by the function $\epsilon : S \rightarrow Z$: $\epsilon(s) = \max_{t \in \rho(s)}$. The *lifetime interval* [16] for each storage value is given by the function $\pi : S \rightarrow 2^Z$. If a single-phase clock scheme is used this function is defined as: $\pi(s) = \{z \in Z | \omega(s) \leq z \leq \epsilon(s)\}$. If a multiple-phase clock scheme is used this function is defined as: $\pi(s) = \{z \in Z | \omega(s) < z \leq \epsilon(s)\}$. The *lifetime* of a storage value $s$ is defined as: $\epsilon(s) - \omega(s)$.

Even when the values of $\omega(s)$ and the members of $\rho(s)$ may not always be in the set $T$, it is useful for the rest of the analysis to concentrate on time values in $T$. Therefore, the auxiliary functions $\omega_m : S \rightarrow T$ and $\rho_m : S \rightarrow 2^T$ are introduced: $\omega_m(s) = \omega(s) \bmod T_0$ and $\rho_m(s) = \{t \bmod T_0 | t \in \rho(s)\}$. In a similar way, $\epsilon_m : S \rightarrow T$, $\epsilon_m(s) = \epsilon(s) \bmod T_0$, $\pi_m : S \rightarrow 2^T$, and $\pi_m(s) = \{t \bmod T_0 | t \in \pi(s)\}$.

Figure 2 shows an example of a set of storage values after taking all time values modulo $T_0$ ($T_0 = 10$). An arrow pointing downward indicates the write time of a storage value and an arrow pointing upward a read time. Note that the repetitive nature of the problem is expressed by repeating control step 0.

For the time being, it is assumed that all storage values $s \in S$, have a lifetime of at most $T_0$. In case of storage

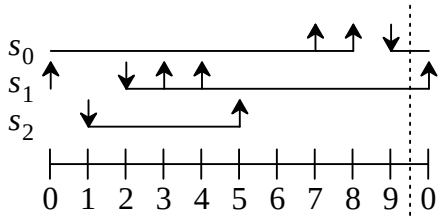


**Figure 2. A set of storage values.**

values with an actual lifetime longer than $T_0$, they are supposed to have been split into values with a lifetime of at most $T_0$ in a preprocessing step.

The following combinatorial optimization problems with respect to synthesis with sequential read-write memories are considered here:

- REALIZABILITY. Given an iteration period $T_0$, a set of storage values $S$ and the functions $\omega$ and $\rho$, find out whether all storage values in $S$ can be stored in a single SRWM. If so, indicate the address to which each storage value should be assigned.
- MINIMAL GROUPING. Given an iteration period $T_0$, a set of storage values $S$ and the functions $\omega$ and $\rho$, find a partition of $S$ in disjoint subsets $S_1, S_2, ....S_k$, such that each subset can be realized on a single SRWM and $k$ is minimal.

## 3. The realizability problem

Because the realizability problem is a central problem in SRWM synthesis, it has been chosen to solve it exactly. As the problem is NP-complete [18], it is solved by a backtracking algorithm as explained below.

A *valid address assignment* is denoted by the function $\alpha : S \rightarrow A$. Here $A$ is the set of address locations $\{0, 1, \ldots\}$. To store $|S|$ storage values, never more than $|S|$ memory locations are necessary (in practice often less, because storage values can share memory locations if they do not have to be stored simultaneously).

A *read/write conflict* between storage values is defined slightly differently for the two clocking schemes. It is denoted by the Boolean function $\gamma : S \times S \rightarrow \{0, 1\}$. For a single-phase clock: $\gamma(s_i, s_j) \equiv \omega_m(s_i) = \omega_m(s_j) \vee \omega_m(s_i) \in \rho_m(s_j) \vee \omega_m(s_j) \in \rho_m(s_i) \vee \rho_m(s_i) \cap \rho_m(s_j) \neq \emptyset$. For a multiple-phase clock: $\gamma(s_i, s_j) \equiv \omega_m(s_i) = \omega_m(s_j) \vee \rho_m(s_i) \cap \rho_m(s_j) \neq \emptyset$.

Two storage values are said to overlap if their lifetime intervals overlap. This is given by the Boolean function $\Omega : S \times S \rightarrow \{0, 1\}$:

$$\Omega(s_i, s_j) = \begin{cases} 0 & \pi_m(s_i) \cap \pi_m(s_j) = \emptyset \\ 1 & \text{otherwise} \end{cases}$$

The *access time* for a destination address ($d$), given the current position of the address pointer ($p$) is given by the function $\Delta : A \times A \rightarrow N^+$:

$$\Delta(p, d) = \begin{cases} d - p & d \geq p \\ d + 1 & d < p \end{cases}$$

If $d < p$, a `reset` and $d$ `next` signals are necessary. Otherwise, just $d - p$ `next` signals are sufficient.

The decision to assign a storage value to a specific address can have consequences for the assignments feasible for the other storage values. This is determined by the number of control steps which is available to move the pointer from the address of one storage value to the address of another. Dealing with this aspect requires the definitions of an *action*, a *successor* and the *available time*.

An *action* is either a read or a write action. A storage value $s_j$ is called a *successor* of $s_i$ if and only if an action on $s_i$ is immediately followed by an action on $s_j$. Here "immediate" means that no other storage value has an action in the mean time. Besides, a storage value is not its own successor by definition. The *available time* is given by the function $\tau : S \times S \rightarrow N^+$. If $s_j$ is a successor of $s_i$, $\tau(s_i, s_j)$ is equal to the smallest number of control steps between an action on $s_i$ and an action on $s_j$, where these actions follow each other immediately. If $s_j$ is not a successor of $s_i$, $\tau(s_i, s_j)$ has an infinite value.

Now the requirements for the realizability problem can be formulated:

1. *No conflicts:* $\forall s_i, s_j \in S, i \neq j : \gamma(s_i, s_j) = 0$.
2. *No overlap:* a function $\alpha$ should be found such that $\forall s_i, s_j \in S, i \neq j : \Omega(s_i, s_j) = 1 \Rightarrow \alpha(s_i) \neq \alpha(s_j)$.
3. *Sufficient time:* $\alpha$ should also obey $\forall s_i, s_j \in S, i \neq j : \Delta(\alpha(s_i), \alpha(s_j)) \leq \tau(s_i, s_j)$.

Checking for the "no-conflicts requirement" is trivial. The remaining requirements are well-suited for a backtracking search procedure [11]. For each storage value for which more than one address assignment is possible, a tentative choice for one of the addresses is made. The other choices will possibly be considered after backtracking. Before continuing the recursive search process, the consequences of the "no overlap" and "sufficient time" requirements are computed, thus restricting the choices for the address assignment of the remaining storage values. A solution has been found when all storage values have a single possible address assignment that obeys the two requirements. If a situation occurs where some storage value cannot be assigned to any location, no solution can be found by further searching and backtracking can start directly. Besides, one has the choice to stop after the *first* solution found or to continue searching in order to find the *best* solution (with the least number of memory locations).

Specific combinations of read-write patterns can directly put initial constraints on address locations before starting the search. For example, the existence of three storage values with overlapping life times $s_i, s_j, s_k \in S$, $\tau(s_i, s_k) = 1$ and $\tau(s_j, s_k) = 1$ implies $\alpha(s_k) = 0$ (only the `reset` signal can change the address pointer from two different locations to the same location in one control step). Generalizations of this rule and other similar patterns can be found in [17].

The reader should check that the problem of Figure 2 is realizable with the unique solution that maps $s_0$ to address 2, $s_1$ to address 0 and $s_2$ to address 1.

## 4. The minimal grouping problem

Given the fact that the exact solution of the realizability problem is already quite complex, it can be stated that an exact solution of the minimal grouping problem is not feasible. Actually, the minimal grouping problem is NP-hard [18]. Obviously, any heuristic that attempts to solve the minimal grouping problem, will need to use an algorithm for the realizability (at least to check whether a given grouping is correct; this could either be done by an exact or by a heuristic method). As the realizability algorithm proposed has an exponential worst-case time complexity, an effective heuristic requiring a "limited" use of the realizability check should be found (search methods that visit many potential solutions, like simulated annealing [12], cannot be considered).

The algorithm proposed for the minimal grouping problem consists of two stages both of which use the routine for realizability test. The first is an *initial assignment* phase in which a solution with some number of memories is constructed such that the assignments to all memories is realizable. The pseudo-code for a simple algorithm that solves this task is given in Figure 3. The storage values are first ordered in decreasing order of "difficulty". The difficulty expresses the overlap that the storage value has with other storage values and the number of read/write conflicts it has. Randomization is used to break ties. The algorithm takes each storage value and tests whether the value can be mapped on one of the memories already allocated. If not, a new memory is allocated to hold this storage value. The set of storage values assigned to a memory with index $i$ is denoted by $R_i$.

The second stage is an *iterative improvement* stage. Its pseudo-code description is given in Figure 4. The main idea is to take the set $P$ of all storage values in the first memory and try to assign them to any of the remaining memories. If this does not succeed for some storage value, this value ends up in the newly created "overflow" memory. As the set of storage values that is being redistributed is known to be realizable, the worst that can happen is that all its elements end up in the overflow memory. So, the number of memories never increases, but may decrease.

```
maxmem := 0;
for all s ∈ S "in order of decreasing difficulty" do
  flag := true;
  for i := 1 to maxmem do
    if flag and realizable({s} ∪ R_i)
    then R_i := {s} ∪ R_i;
         flag := false;
    fi
  od;
  if flag
  then maxmem := maxmem + 1;
       Rmaxmem := {s}
  fi
od;
```

**Figure 3. Minimal grouping: initial assignment.**

The process just described is repeated while being careful not to enter an infinite loop. If the set $P$ at a certain iteration equals any of the sets $P$ encountered in an earlier iteration, the iterative improvement process is terminated. Experiments show that this happens after some 20 to 30 iterations. During this search process the solution with the minimal number of total memory locations (for the optimal number of memories) is recorded as the best solution.

## 5. Experimental results

The results to be presented in this section have been obtained by an implementation of the algorithms mentioned above in C++ on an HP 735 machine (results in earlier publications [5, 19] were obtained by a prototype implemented in CMU Common Lisp; the Lisp prototype also contained a postprocessing routine that was not implemented this time). The two versions of realizability algorithm were implemented (see Section 3) and it turned out that using the *best-solution* version did not improve the quality of the solution compared to the *first-solution* version. All results reported here were obtained with the first-solution version.

Sets of storage values with "regular" write and read patterns, as used in the publications of Aloqeely and Chen [1], have not been investigated. It is obvious that sequential memories are especially suited for this type of patterns. The goal of the experiments performed was to see whether SRWMs could as well be considered for "irregular" problems. Although many benchmarks for high-level synthesis are known, generally accepted benchmarks for the isolated problem of memory synthesis do not exist. Therefore, for

```
repeat
  P := R_1;
  for i := 1 to maxmem – 1 do
    R_i := R_{i+1}
  od;
  maxmem := maxmem – 1;
  for all s ∈ P do
    flag := true;
    for i := 1 to maxmem do
      if flag and realizable({s} ∪ R_i)
      then R_i := {s} ∪ R_i;
           flag := false;
      fi
    od;
    if flag
    then maxmem := maxmem + 1;
         R_maxmem := {s}
    fi
  od
until "loop condition detected";
```

**Figure 4. Iterative improvement.**

the first set of experiments, a family of benchmark examples proposed by Parhi [14] has been selected. They consist of the repetitive transpositions of arbitrary size matrices. Consider e.g. the following $3 \times 4$ matrix:

$$\begin{bmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \end{bmatrix}$$

of which the elements arrive in a row-by-row order (one element per control step): $s_0$, $s_1$, $s_2$, $s_3$, $s_4$, etc. Transposition requires that the elements are output in a column-by-column order: $s_0$, $s_4$, $s_8$, $s_1$, $s_5$, etc. This requirement directly defines the $nm - 1$ storage values (the matrix element at the lower left has a lifetime of zero). Once that all elements of one matrix have been processed, a new matrix to be transposed arrives. Obviously, $T_0 = nm$ for an $n \times m$ matrix. Parhi shows that for these problems $(n-1)(m-1)$ is a lower bound for the number of memory locations (and that this lower bound can be met using tailored shift register structures interconnected in a specific way).

These benchmarks have the property that they contain storage values with lifetimes longer than $T_0$. They have been split in the way mentioned in Section 2. An implementation with RAMs would also require splitting, unless special addressing hardware is used [13]. Table 1 shows some results obtained for different matrix dimensions both for the single-phase (indicated by "sp") and multiple-phase ("mp") clocking models. For each problem 10 runs have been performed to be less dependent on the influence of randomization. The given CPU time is the average per run. For each benchmark, the number of memories and the total number of memory locations found are given. In the cases that different results were found in different runs, the minimum and maximum values are given in the form "min..max". The values between parentheses are lower bounds. For the number of memories, this is a "heuristically computed" lower bound for the case that RAMs instead of SRWMs would be used for the given splitting. For the total number of memory locations, the lower bound in the hardware model of Parhi is given (see above); it is not likely that the lower bound can be met in a hardware model of RAMs or SRWMs.

| problem | # memories | | # locations | CPU time [sec] |
|---|---|---|---|---|
| $5 \times 5$ sp | 4..5 | (4) | 22..24 (16) | 0.21 |
| $5 \times 5$ mp | 3..4 | (3) | 18..19 (16) | 0.50 |
| $6 \times 6$ sp | 5..6 | (4) | 33..35 (25) | 1.7 |
| $6 \times 6$ mp | 4 | (3) | 28..30 (25) | 1.6 |
| $7 \times 7$ sp | 6 | (4) | 45..49 (36) | 5.8 |
| $7 \times 7$ mp | 5 | (3) | 39..44 (36) | 14 |
| $8 \times 8$ sp | 7 | (4) | 61..64 (49) | 110 |
| $8 \times 8$ mp | 5..6 | (3) | 53..58 (49) | 130 |

**Table 1. Results for matrix transposition.**

With respect to the necessary number of memories, it can be seen that this number is generally close to the lower bound (given by the possible RAM assignment). Also the total number of memory locations is often close to the lower bound. It should be noted that the shift-register structure of Parhi [14] is capable to read from and write to any register within a single control step. It turns out that only few extra locations are necessary when using a multiple-phase clock. The difference with the lower bound is, obviously, larger for the more constrained single-phase clock scheme.

The second set of experiments intended to evaluate the minimal grouping heuristic. For this purpose, random benchmarks were used that were known to fit in a single memory (by first constructing a pattern of `next` and `reset` signals and deriving write and read times from them). Merging $n$ such benchmarks with the same value for $T_0$ leads to a problem that is guaranteed to have a solution with $n$ SRWMs. The results found are given in Table 2. It can be seen that the minimal grouping heuristic performs quite well for these benchmarks and often finds a solution with at most $n$ memories, although there is a tendency in the results to deteriorate as the benchmarks become more complex. Just to give an impression, the most complex benchmark ($T_0 = 80$ and $n = 5$) consisted of 72 storage values (performing 177 actions) and the ten runs of the program produced solutions with 54 to 56 locations and using an average run time of 47 seconds.

| $T_0$ | $n=2$ | $n=3$ | $n=4$ | $n=5$ |
|---|---|---|---|---|
| 20 | 2 | 3 | 4 | 4 |
| 30 | 2 | 3 | 4..5 | 5 |
| 40 | 3 | 3 | 4 | 5 |
| 50 | 2 | 3 | 4 | 5 |
| 60 | 2 | 3..4 | 5 | 5..7 |
| 70 | 2 | 3..4 | 4..5 | 5..6 |
| 80 | 2 | 3..4 | 5 | 6..7 |

**Table 2. Random benchmark results.**

## 6. Conclusions and further research

This paper has shown that SRWMs may be an attractive alternative to RAMs for the storage of values in data-path synthesis. Even in the case of "irregular" read-write patterns, SRWMs could be applied in spite of their limited addressing capabilities.

Two algorithms were presented to automatically map storage values on SRWMs. The first one solves the realizability problem exactly using an algorithm that has an exponential worst-case time complexity. The second one uses the first as a subroutine to solve the minimal-grouping problem in a heuristic way. It, therefore, also has an exponential worst-case time complexity.

An important issue for future research is to speed up these algorithms, possibly by relaxing the requirement of solving the realizability problem exactly. The ultimate issue is to integrate an SRWM-synthesis tool within a larger framework of tools for data-path synthesis, especially a synthesis system for low-power design.

## References


[1] M. Aloqeely and C.Y.R. Chen. A new technique for exploiting regularity in data path synthesis. In *European Design Automation Conference, EURO-DAC*, pages 394–399, 1994.

[2] I.E. Bennour and E.M. Aboulhamid. Register allocation using circular FIFOs. In *International Symposium on Circuits and Systems*, pages 560–563, 1996.

[3] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[4] D.D. Gajski, N.D. Dutt, A.C.H. Wu, and S.Y.L. Lin. *High-Level Synthesis, Introduction to Chip and System Design.* Kluwer Academic Publishers, Boston, 1992.

[5] S.H. Gerez and E.G. Woutersen. A high-level synthesis tool for the assignment of storage values to sequential read-write memories. In *IFIP TC 10 WG 10.5 International Workshop on Logic and Architecture Synthesis*, pages 220–230, Grenoble, December 1995.

[6] B.S. Haroun and M.I. Elmasry. Architechtural synthesis for DSP silicon compilers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(4):431–447, April 1989.

[7] S.M. Heemstra de Groot, S.H. Gerez, and O.E. Herrmann. Range-chart-guided iterative data-flow-graph scheduling. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 39:351–364, May 1992.

[8] A. Heubi, M. Ansorge, and F. Pellandini. Architecture VLSI faible consommation pour le traitement numérique du signal. In *Proceedings GRETSI '93*, pages 1083–1086, Juanles-Pins, France, September 1993.

[9] A. Heubi, S. Grassi, M. Ansorge, and F. Pellandini. A low power VLSI architecture with an application to adaptive algorithms for digital hearing aids. In M.J.J. Holt, C.F.N. Cowan, P.M. Grant, and W.A. Sandham, editors, *Signal Processing VII: Theories and Applications (Proceedings of the EUSIPCO-94 Seventh European Signal Processing Conference)*, pages 1875–1878, 1994.

[10] H.A. Hilderink and J.A.G. Jess. Rom-based multi thread controller. In *IFIP Workshop on Logic and Architecture Synthesis*, pages 231–241, Grenoble, December 1993.

[11] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, Maryland, 1978.

[12] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–690, May 1983.

[13] J.L. van Meerbergen, P.E.R. Lippens, W.F.J. Verhaegh, and A. van der Werf. Relative location assignment for repetitive schedules. In *European Conference on Design Automation with the European Event on ASIC Design, EDAC/EUROASIC*, pages 403–407, 1993.

[14] K.K. Parhi. Systematic synthesis of DSP data format converters using life-time analysis and forward-backward register allocation. *IEEE Transactions on Circuit and Systems - II: Analog and Digital Signal Processing*, 39(7):423–440, July 1992.

[15] K.K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers*, 40(2):178–195, February 1991.

[16] L. Stok and J.A.G. Jess. Foreground memory management in data path synthesis. *International Journal of Circuit Theory and Applications*, 20:235–255, 1992.

[17] E.G. Woutersen. The application of sequential read-write memories in high-level synthesis. Master's thesis, University of Twente, Department of Electrical Engineering, Laboratory of Network Theory, December 1995. EL-BSC-097N95.

[18] E.G. Woutersen and S.H. Gerez. Some complexity results in memory mapping. In *Third HCM BELSIGN Workshop*, Corsica, April 1996.

[19] E.G. Woutersen and S.H. Gerez. The application of sequential read-write memories in high-level synthesis. In *GRONICS '96: Groningen Information Technology Conference for Students*, pages 93–100, Groningen, The Netherlands, February 1996.