To be published in: GRONICS '96, Groningen Information Technology Conference for Students,
University of Groningen, The Netherlands, February 23, 1996.

# The Application of Sequential Read-Write Memories in High-Level Synthesis

**Erwin G. Woutersen and Sabih H. Gerez***

Department of Electrical Engineering, Laboratory of Network Theory (EL-BSC)
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail: s.h.gerez@el.utwente.nl

## Abstract

In this paper, a new memory element that can be used as an alternative to register files in data-path synthesis, is proposed. This is the sequential read-write memory (SRWM). Its organization is similar to that of a standard random access memory (RAM), with the difference that the address decoder has been replaced by a shift-register-like structure. It is used to sequentially point at the memory locations for read or write. Once sufficiently large, this new memory element needs less chip area and consumes less power than a standard RAM (of the same size). However, the problem of assigning a set of storage values to a set of memory locations becomes much more difficult.

In this paper algorithms are presented to check whether a set of storage values fit in a single SRWM (exactly, by means of branch-and-bound) and to automatically map storage values in as few SRWMs as possible (by means of heuristics). By using these algorithms, the memory assignment problem for SRWMs can be solved. Several examples show that solutions found using SRWMs are close to solutions feasible by RAMs.

**Keywords:** High-level synthesis, Memory assignment, Sequential memories.

## 1 Introduction

A research area in the field of design automation of digital systems that is receiving increasing attention in the last years is *high-level synthesis*: the translation from a behavioral description at the algorithmic level into a structural description at the register transfer level which implements the given behavior. Generally, in high-level synthesis three types of problems are distinguished. These are *partitioning*, *scheduling* and *assignment* (see e.g. [5]).

The partitioning problem deals with decomposing large behavioral descriptions into several smaller ones. That way, the synthesis problem becomes more tractable. Furthermore, by dividing large problems, packaging constraints can be met. The latter two problems, scheduling and assignment are tightly interrelated. Scheduling deals with mapping operations in the behavioral description to time instances, while assignment deals with mapping operations in the behavioral description to hardware resources. Generally, the scheduling problem is solved prior to the assignment problem.

Therefore, in this paper it is assumed that a schedule has already been fixed. In general, once a schedule has been fixed, it will be necessary to store some intermediate results in the data-path in order to satisfy the given schedule. These intermediate results are called *storage values* [17]. A part of the assignment problem deals with the task of mapping these storage values onto memory elements available within the target architectural model.

This paper addresses the memory assignment problem for an alternative memory element, the *sequential read write memory* (SRWM) [8, 9, 10, 11, 12, 19]. In an integrated circuit designed by Heubi et al. [12], SRWMs have been applied successfully. In this design the memory assignment problem was solved manually. To make SRWMs more easily applicable, in this paper two algorithms, related to the memory assignment problem for SRWMs are considered.

This paper presents a summary of [19] that the reader can consult for a more thorough description of the subject. The structure of this paper is as follows. First the SRWM is introduced. Then two combinatorial optimization problems related to the use of SRWMs in high-level synthesis are formulated and analyzed. Algorithms to solve these problems are proposed next. The paper finishes with the presentation of several experimental results obtained by these algorithms and some conclusions.

*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.

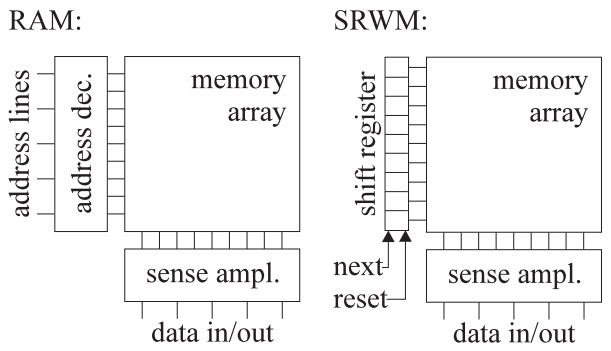


**Figure 1:** *Block diagrams of RAM and SRWM.*

## 2 Memory Elements

As a result of the scheduling, a set of storage values is defined. Each storage value is written once and read one or more times. In the mean time the value has to be stored in some kind of memory. Normally *registers* and *register files* [5, 17] are used for this purpose. Here a register is a single memory location, while a register file consists of several memory locations with one or more read/write ports. The different locations are identified by an address. Register files are most frequently implemented by means of single port *random access memories* (RAMs) [18]. Recently, some alternative memory elements like *queues*, *stacks* and *bi-directional queues* have been proposed by Aloqeely and Chen [1, 2, 3]. These structures consist of chains of shift registers. Another alternative, the *circular queue* has been studied by Bennour and Aboulhamid [4]. This paper considers yet another alternative type of memory element, the SRWM.

A RAM basically consists of a *memory array* (where the actual data is stored) surrounded by several supporting circuits (see Figure 1). The data in the memory array is stored in columns and rows (words). Each word can be accessed by asserting its associated word line. The selection of the word lines is done by the *address decoder*, which asserts one of the word lines dependent on its inputs. For small RAMs the address decoder typically consists of an inverter/buffer circuit followed by an array of $n$-input AND arrays (where $n$ is the number of address lines). Here for each memory word one $n$-input AND gate is required.

The hardware structure of the SRWM is almost identical to the hardware structure of a RAM (see Figure 1). Only the address decoder has been replaced by a shift-register like structure. In this shift register a single high bit is used to sequentially point at the active memory location (which is used to read from or write to). The location of this pointer is controlled by two control signals. One (`next`) triggers a shifting operation and makes the pointer point to the next higher memory location in the next control step. Another (`reset`) is used to reset the shift register, such that the high bit points at the lowest memory location. For this purpose a new shift register has been designed by Heubi et al. [10]. This structure needs 13 transistors to drive each memory word line and assures that at any time only one memory word is active. An alternative model, which allows the usage of a third control signal (`previous`) to decrease the position of the pointer, has also been studied in [19]. However, in this paper only the former model will be considered.

In synchronous circuits, all read and write actions are triggered by the clock. If a *single clock* is used, one of the clock phases can be used to either read or write the data, while the other clock phase is used to change the pointer. In circuit design, also *multiple-phase clock schemes* are commonly used [18]. In that case, one of the clock phases can be used to read from the memory, while another clock phase is used to write data to the memory. A third clock phase is still available to change to pointer. Such a multiple-phase clock scheme allows the reuse of a memory location within the same clock cycle.

As opposed to RAMs, which allow random access of any memory location, SRWMs are far more constrained. However, in RAMs, the size of the address decoder grows more than linearly with the number of memory words, while in SRWMs the size of the shift register grows linearly with the number of memory words. Therefore, once they are sufficiently large (with respect to the number of memory locations), SRWMs are expected to be smaller and consume less power than RAMs. Furthermore, fewer control signals are necessary and therefore the controller can be simplified as well. Therefore, despite their limitations, these alternative structures are still considered to be used as an alternative for RAMs.

Although SRWMs have a similar behavior as the shift-register structures suggested recently [1, 4], they are more interesting as they are generally smaller and consume 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 et al. [10, 11]. It turns out that SRWMs are better in both aspects once that the memories are sufficiently large (in word length and number of memory positions).

The memory assignment problem for RAMs is relatively easy. As long as storage values do not have *read/write conflicts*, they can be stored in a single RAM without bothering about their addresses [17]. Because of their limited addressing capabilities, the assignment of a set of storage values to a SRWM is far less obvious. Not only read/write conflicts should be taken into account, but also the fact that the decision to assign a storage value to a specific address location can have consequences for the assignments possible for the other storage values.

# 3 Problem Formulation

## 3.1 Some Definitions

The application domain addressed in this paper is digital signal processing, characterized by the (almost) infinite repetition of some algorithm. Such an algorithm is characterized by a parameter $T_0$ called the *iteration period* or *periodicity* [16]. All computations are repeated every $T_0$ control steps, where a control step corresponds to a complete cycle of the 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* or *loop folding* model, the schedule span can be larger than $T_0$. The problem is defined as follows:

**Definition 1.** One *iteration period* in a schedule consists of the time instants $T = \{0, 1, ..., T_0 - 1\}$.

**Definition 2.** The intermediate values, which have to be stored in the datapath will be called *storage values* [17] and will be part of the finite set $S = \{s_0, s_1, s_2, ...\}$.

**Definition 3.** The *write time* of each storage value is given by the function $\omega : S \rightarrow Z$. Here $Z$ is the set of all integers.

**Definition 4.** The *set of read times* of each storage value is given by the function $\rho : S \rightarrow 2^Z$. Where $2^Z$ is the *power set* of $Z$ (the set of all its subsets). Note that a storage value can not be read before it is written. Therefore $\forall s \in S, \forall z \in \rho(s) : z \geq \omega(s)$.

**Definition 5.** The *last read time* of each storage value is given by the function $\epsilon : S \rightarrow Z : \epsilon(s) = \max_{t \in \rho(s)}$.

**Definition 6.** The *lifetime interval* [17] 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: $\pi(s) = \{z \in Z | \omega(s) < z \leq \epsilon(s)\}$. If a multiple-phase clock scheme is used this function is defined as: $\pi(s) = \{z \in Z | \omega(s) \leq z \leq \epsilon(s)\}$.

**Definition 7.** The *lifetime* of each storage value is given by the function $\lambda : S \rightarrow N^+ : \lambda(s) = \epsilon(s) - \rho(s)$. Here $N^+$ is the set of strictly positive integers.

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 this 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: $\omega_m(s) = \omega(s)$ mod $T_0$ and $\rho_m(s) = \{t \bmod T_0 | t \in \rho(s)\}$. In a similar way, $\epsilon_m : S \rightarrow T : \epsilon_m(s) = \epsilon(s) \bmod T_0$ and $\pi_m : S \rightarrow 2^T : \pi_m(s) = \{t \bmod T_0 | t \in \pi(s)\}$.

Figure 2 shows an example of a set of storage values. To express the infinite repetition of the same read and write actions, the time axis is often represented as a circle. In this time circle the lifetime intervals of the variables are shown. After taking all

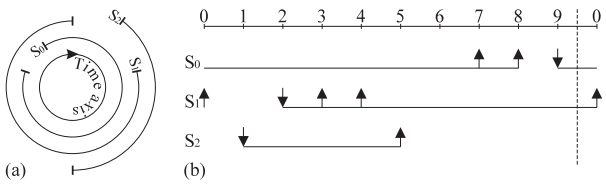


**Figure 2:** *A set of storage values.*

time values modulo $T_0$, the same figure can also be shown more clearly on a linear time scale. An arrow pointing downward indicates the write time of a variable and an arrow pointing upward a read time. In the figure, $T_0 = 10$.

For the time being, it is assumed that all storage values $s \in S$, have a lifetime in the range $[1, T_0]$ if a multiple-phase clock scheme is used and $[1, T_0 - 1]$ if a single-phase clock scheme is used. Storage values with a lifetime 0 do not have to be stored and are therefore not considered. Storage values with an actual lifetime longer than (or equal to) $T_0$ can not be assigned to a single memory location, but always need a register transfer [17] unless special hardware is used [13, 4]. Therefore these storage values are supposed to have been split into several values with a lifetime of at most $T_0$ (or $T_0 - 1$) in a preprocessing step (*value splitting* [19]).

## 3.2 The Combinatorial Optimization Problems

Solving a memory assignment problem generally consists of several mutually related phases. These phases include *value grouping* and *register assignment* [17]. The value grouping phase minimizes the number of interconnections in the datapath by grouping the values such that the values in each group can share a register file. During the register assignment phase, registers (memory locations) are assigned for each group separately. The main task of that part of the synthesis process is to use as few registers as possible by storing values with nonoverlapping lifetimes at the same location.

To solve the value grouping problem, there needs to be some way to decide whether a set of storage values can be stored in a single memory. If RAMs are used, this problem is almost trivial. Any set of variables can be stored in a single RAM as long as no read or write conflicts exist. If SRWMs are used, this problem is much more difficult. Therefore some new combinatorial optimization problems have been formulated [8]. In this paper two of them will be discussed:

- 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.

The *minimal grouping problem* is strongly related to the value grouping problem. However because of the special nature of SRWMs, it is not identical. The *realizability problem* does not only deal with the decision whether a set of values can be stored in a single SRWM, but it deals with the register assignment problem as well. In this paper, algorithms for both problems are described.

# 4 Realizability

## 4.1 Preliminary Analysis

The objective of the realizability problem is to map each storage value in a given set $S$ to an address in a SRWM. This should be done in such a way that the resulting mapping is able to meet all requirements given by the problem description ($T_0$, $\omega$ and $\rho$). Such a mapping is called a *valid address assignment*. While solving the realizability problem, a valid address assignment which uses the fewest possible number of registers has to be found.

**Definition 8.** A *valid address assignment* is denoted by the function $\alpha : S \rightarrow A$. Here $A$ is the set of address locations $\{0, 1, ..., N_w\}$.

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).

An obvious limitation of a register file with only a single read/write port is that at any time only one data transfer can take place. If two or more concurrent transfers are required, a read/write conflict occurs. It should be noted that if a multiple-phase clock scheme is used, read and write actions take place at different parts of the clock cycle. Therefore, in a multiple-phase clock scheme different actions can not interfere. This leads to a different definition of a read/write conflict for different clock schemes.

**Definition 9.** A *read/write conflict* 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: $\gamma(s_i, s_j) \equiv \omega_m(s_i) = \omega_m(s_j) \vee \rho_m(s_i) \cap \rho_m(s_j) \neq \emptyset$.

To minimize the necessary number of memory locations, it is useful to store as many storage values as possible at the same memory location without that two values are stored at the same memory location at the same time. To formalize this observation, a new function $\Omega$ is defined:

**Definition 10.** Two storage values are said to overlap if their lifetime intervals overlap. This is given by the boolean function $\Omega : S \times S \rightarrow \{0, 1\}$:

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

Note that two variables which overlap if a single-phase clock scheme is used, may not overlap if a multiple-phase clock scheme is used.

A special property, which distinguishes SRWMs from RAMs is the access time. For RAMs the access time is constant for every address. For SRWMs, the access time depends on the current position of the address pointer. Therefore a new function is defined:

**Definition 11.** For SRWMs, the *access time* (in control steps) for a destination address ($d$), given the current position of the 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` followed by $d$ `next` signals is necessary. Otherwise, just $d - p$ `next` signals are sufficient.

From this it can be seen that 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. Therefore, some new definitions are introduced.

**Definition 12.** 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. Further, a storage value can never be its own successor.

**Definition 13.** 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. This means that no timing constraints are introduced.

Considering all these aspects, it is possible to formulate the following theorem:

**Theorem 1.**
Given an iteration period $T_0$, a set of storage values $S$ and the functions $\omega$ and $\rho$. These storage values can be stored in a single SRWM if and only if:

1. $\gamma(s_i, s_j) = 0 \quad \forall s_i, s_j \in S, i \neq j$
2. There exist a valid mapping $\alpha$, such that:
   (a) $\alpha(s_i) \neq \alpha(s_j) \quad \forall s_i, s_j \in S : \Omega(s_i, s_j) = 1$
   (b) $\Delta(\alpha(s_i), \alpha(s_j)) \leq \tau(s_i, s_j) \quad \forall s_i, s_j \in S$

This theorem is a direct consequence of the definitions in this section. Therefore the proof (which can be found in [19]) is omitted in this paper.

### 4.2 Complexity Analysis

An important characteristic of an algorithm is its *computational complexity*. In complexity theory only decision problems are considered. Therefore, the considered problem first has to be converted to a decision version. This can be done by introducing an additional parameter $K$ and asking whether the problem can be solved using $K$ or less memory locations.

**Theorem 2.**
The decision version of the realizability problem is *NP-complete*.

*Proof:*
For this proof, we consider the subset of instances where $\tau(s_i, s_j) = 0 \vee \tau(s_i, s_j) \geq |S| \quad \forall s_i, s_j \in S$. Because the number of memory locations is always lower than or equal to $|S|$, also $\Delta(p, d) \leq |S| \quad \forall p, d$. Therefore, for those instances, $\Delta(\alpha(s_i), \alpha(s_j)) \leq \tau(s_i, s_j) \quad \forall s_i, s_j \in S, i \neq j : \gamma(s_i, s_j) \neq 0$. Therefore, condition 2b of Theorem 1 loses its significance for the considered subset.

Then, it is sufficient to show that this problem is isomorphic to *graph k-colorability* problem [6]. Therefore, the overlap function $\Omega$ is used for the construction of the so-called *overlap graph* $O(S, E)$. The vertex set of this graph consists of the storage values in $S$. There exist an edge $(s_i, s_j) \in E$ if and only if $\Omega(s_i, s_j) = 1$. Using this graph representation, the problem of finding a register assignment using a minimal number of registers becomes identical to the graph k-colorability problem. Even though this graph is derived from a set of "circular" intervals (see Figure 2) and therefore belongs to the special class of *circular-arc graphs*, the graph k-colorability problem remains NP-complete [7]. Therefore, also the realizability problem is NP-complete. □

### 4.3 The Algorithm

Because the realizability problem is a central problem in SRWM synthesis, it has been chosen to solve this problem exactly, despite its NP-completeness. This is done by means of a *branch-and-bound* algorithm in which the second condition of Theorem 1 plays a central role. This algorithm has of course an exponential time complexity.

First the algorithm enters a preparatory phase in which the algorithm verifies whether the first condition of Theorem 1 (no read/write conflicts) holds and checks for several special situations which can lead to direct conclusions that constrain the search space [8, 19]. This part of the algorithm operates in polynomial time.

Then, the actual branch-and-bound algorithm starts by making a tentative choice to assign one of the storage values to a specific address. The other choices will be possibly considered after backtracking. Because of the second condition of Theorem 1, such a choice often has consequences for the possible assignments of the other storage values. These consequences are evaluated before continuing the recursive search process. If a situation occurs where some storage value can not be assigned to any location, no solution can be found by further searching and backtracking can start. A solution has been found when all storage values have been assigned such that the second condition of Theorem 1 is met.

*Example:*
For the problem presented in Figure 1, the algorithm operates as follows. Because there are no read/write conflicts, the first condition of Theorem 1 is met. Furthermore, all three storage values have to be stored simultaneously. Therefore, no two storage values can be stored at the same address. A possible choice is to assign $s_0$ to address 0. Because $\tau(s_0, s_1) = 1$, there is one clock cycle to move the pointer from $\alpha(s_0) = 0$ to $\alpha(s_1)$. In one clock cycle, the pointer can either be increased by one or reset to 0. Since address 0 is already in use, there is only one option: increasing the pointer. Hence, $s_1$ has to be stored at address 1. Now, because $\tau(s_1, s_2) = 1$, one can also conclude that $s_2$ has to be stored at address 2. And then, because $\tau(s_2, s_1) = 1$ one can conclude that $s_1$ can only be stored at address 3. Since $s_1$ already had to be stored at address 1, a contradiction occurs. Therefore the choice to store $s_0$ at address 0 turned out to be incorrect and another choice has to be made.

The reader should check that this example is indeed realizable, with a unique solution that maps $s_0$ to address 2, $s_1$ to address 0 and $s_2$ to address 1.

## 5 Minimal Grouping

### 5.1 Complexity Analysis

The objective of the minimal grouping problem is to partition a set $S$ of storage values in disjoint subsets $S_1, S_2, ..., S_k$, such that each subset can be stored in a single SRWM and $k$ is minimal. This problem can be converted to a decision problem by introducing a parameter $K$ and asking whether a partitioning into $K$ or less subsets is feasible.

**Theorem 3.**
The decision version of the minimal grouping problem is *NP-hard*.

*Proof: (sketch)*
For this proof, we also start considering the subset of instances where $\tau(s_i, s_j) = 0 \vee \tau(s_i, s_j) \geq |S| \quad \forall i, j$. Furthermore, the memories are considered to be

large enough to store the $|S|$ storage values on unique addresses. Therefore, the second condition of Theorem 1 loses its significance.

Then, the conflict function, $\gamma$ can be used to construct the *conflict graph*, $C(S, E)$. The vertex set of this graph consists of the storage values in $S$. There exists an edge $(s_i, s_j) \in E$ if and only if $\gamma(s_i, s_j) = 1$. Using this graph representation, the minimal grouping problem becomes a *graph coloring* problem, which is known to be NP-complete [6]. As it is not difficult to show that any instance of graph coloring can be polynomially reduced to an instance of minimal grouping [19], this problem is at least NP-complete. However, since it is necessary to check that each element of the partition is realizable, this problem is NP-hard. □

### 5.2 The Algorithm

Although an exact solution is often more desirable, an exact solution for this problem is not even feasible for very small problems. Therefore, a heuristic has been developed. Due to the exponential time complexity of the realizability algorithm, it is necessary to find a heuristic, which requires a "limited" usage of the realizability check. Therefore, many general purpose methods for combinatorial optimization, which visit many potential solutions (like simulated annealing or genetic algorithms) could not be considered.

Instead a variation on the *First Fit* algorithm [6] has been used. This algorithm has originally been developed for solving the *bin packing problem*. This problem is: Given a finite set $U = \{u_1, u_2, ..., u_n\}$ of "items" and a rational size $s(u) \in [0, 1]$ for each item $u \in U$, find a partition of $U$ into disjoint subsets $U_1, U_2, ..., U_k$ such that the sum of sizes in each $U_i$ is no more than 1 and such that $k$ is minimal.

The First Fit algorithm starts with an infinite sequence $B_1, B_2, ...$ of unity-capacity bins, all of which are empty. Then the algorithm places the items into the bins, one at a time, in order of increasing index. It does this according to the following simple rule: always place the next item $u_i$ into the lowest-indexed bin in which it will fit without exceeding the bin capacity.

This same algorithm can be adapted for SRWMs by replacing the sequence of empty bins by a sequence of empty memories $M_1, M_2, ....$ Instead of items, now storage values are placed into the memories. The placing rule becomes: always place the next storage value $s_i$ into the lowest-indexed memory in which it is possible to realize an assignment together with the storage values which already have been placed into that memory.

The found result can be optimized iteratively by removing the first memory, $M_1$ from the solution and using the same first fit algorithm to distribute the storage values from this memory among the other memories. In this way, each iteration one memory is removed, while at most one new memory is introduced. This means that the total number of memories will never increase, but might decrease. This iterative optimization procedure is continued until a set of storage values has to be redistributed for the second time. In practice, this situation typically occurs after 20 to 30 iterations.

## 6 Experimental Results

### 6.1 Setup of the Experiments

All presented algorithms have been implemented in CMU Common Lisp running on an HP 735 machine. Because of the used minimal grouping heuristic, the initial order of the storage values influences both the result and the necessary CPU time. Therefore each test has been performed several times (100 runs for the smaller problems, ranging down to 10 runs for the largest problems). The presented CPU times are the average times.

Because this research only focuses on the memory assignment problem, many standard benchmark examples for high-level synthesis could not be considered. These benchmarks have been designed for complete data-path synthesis tools, which include a scheduling and assignment. Although, a conversion program that converts the results of an in-house scheduling and assignment tool [14] has been developed, the choice for a specific scheduler results in nonstandard benchmarks. Therefore, a scheduler-independent family of benchmarks has been used.

### 6.2 Benchmark Results

One family of benchmarks, which does not need a scheduler is the family of *matrix transposition* benchmarks. These problems consider the repetitive transposition of arbitrarily sized 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): $s_0$ followed by $s_1$, $s_2$, $s_3$, $s_4$, etc. Transposition requires that the storage values are output in a column-by-column order: $s_0$, followed by $s_4$, $s_8$, $s_1$, $s_5$, etc. Once that all elements of one matrix have been processed, a new matrix to be transposed arrives. Therefore, $T_0 = nm$ for an $n \times m$ matrix. These problems have been studied in detail by Parhi [15]. He shows that for the transposition of an $n \times m$ matrix, a latency between the input and the output of at least $(n-1)(m-1)$ is necessary. Subsequently, the lower bound of the necessary number of memory locations is also equal to $(n-1)(m-1)$. Parhi presents a method to reach

| problem | # mems. found | # locs. found | CPU time [sec] |
|---|---|---|---|
| $3 \times 3$ sp | 2 (2) | 7..8 (4) | 0.15 |
| $3 \times 3$ mp | 2 (2) | 5..6 (4) | 0.09 |
| $4 \times 4$ sp | 4 (4) | 13..14 (9) | 2.1 |
| $4 \times 4$ mp | 3 (3) | 10..12 (9) | 1.2 |
| $5 \times 5$ sp | 4..5 (4) | 22..25 (16) | 8.4 |
| $5 \times 5$ mp | 4 (3) | 17..19 (16) | 10 |
| $6 \times 6$ sp | 5..6 (4) | 33..37 (25) | 74 |
| $6 \times 6$ mp | 4..5 (3) | 26..30 (25) | 46 |
| $7 \times 7$ sp | 6 (4) | 46..48 (36) | 320 |
| $7 \times 7$ mp | 5 (3) | 37..41 (36) | 400 |
| $8 \times 8$ sp | 7 (4) | 61..64 (49) | 5700 |
| $8 \times 8$ mp | 5..6 (3) | 50..54 (49) | 1700 |
| $4 \times 6$ sp | 4..5 (4) | 19..24 (15) | 15 |
| $4 \times 6$ mp | 4 (3) | 18..19 (15) | 9.0 |
| $4 \times 8$ sp | 5 (4) | 26..31 (21) | 36 |
| $4 \times 8$ mp | 4..5 (3) | 24..27 (21) | 52 |

**Table 1:** *Some experimental results.*

this lower bound by using a specially interconnected shift-register structure.

Table 1 shows the results for several different matrix dimensions. These problems have been solved for both clock models (single-phase and multiple-phase). Except for the larger problems, every problem has been solved 100 times. This led to a certain variation in the results. A lower bound is given as well (between brackets). The lower bound on the number of memory locations refers to the model of Parhi, although there is no guarantee that this lower bound can be reached using other hardware models than the shift-register structures suggested by Parhi.

The lower bound on the number of memories is based on a "hand computed" RAM assignment. Because of concurrent read/write actions, the storage values have to be divided over several RAMs. Only by using multi-port memories or special addressing hardware [13], it is always possible to store all values in a single RAM.

From the results presented in Table 1 it can be observed that the necessary execution time grows exponentially with the number of storage values. This is the expected behavior. Note that the results differ from those presented in [9] due to an improved implementation of the algorithms.

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). Only for the larger problems, where many storage values have to be stored in a single memory, the necessary number of SRWMs is larger than this lower bound. This might be due to sub-optimal results obtained by the minimal grouping heuristic.

The necessary number of memory locations are often close to the lower bound. It should be noted that the shift-register structure of Parhi [15] is capable to read from and write to a 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.

To demonstrate that the algorithms can also deal with more irregular problems (which are not considered by Aloqeely and Chen [1, 2, 3]), a set of storage values for the well-known "fifth-order wave-digital filter" benchmark has been generated using an in-house scheduling and assignment tool [14]. The example considered contains 20 storage values of which some are read several times. The lifetimes vary in the range $[1, 17]$. The iteration period, $T_0$ is 18. Because of existing read/write conflicts, at least 4 RAMs are necessary to store al storage values. The example can also be solved using 4 SRWMs.

# 7 Conclusions

In this paper an alternative memory element, the sequential read-write memory (SRWM) has been proposed. It uses less chip area and consumes less power than common RAM.

The new memory elements can be used as an alternative to register files in data-path synthesis. However, because of their limited addressing possibilities, the memory assignment problem becomes much more difficult. Therefore, two related combinatorial optimization problems have been proposed, the realizability problem (detect whether a set of storage values can be stored in a single SRWM) and the minimal grouping problem (automatically map a set of storage values into as few SRWMs as possible). It has been proven that the problems are NP-complete and NP-hard respectively.

Because of its importance, an algorithm which solves the realizability problem exactly (by means of branch and bound) has been developed. For the minimal grouping problem a new heuristic has been developed.

A set of matrix transposition benchmarks indicate that, despite their limited addressing capabilities, the application of these new memory elements does not lead to serious hardware inefficiencies. Therefore the algorithms presented here, could be incorporated in a toolbox for high-level synthesis. Preferably, this toolbox should contain distinct tools for the synthesis of different memory elements, allowing the designer to choose the best solution for specific parts of the design.

Further research should mainly focus on increasing the speed of the realizability function. Constrained logic programming, as reported in [4], looks promising in this respect.

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. *Sequencers: A New Alternative for Data Path Synthesis.* PhD thesis, Department of Electrical and Computer Engineering, Syracuse University, June 1995.

[2] 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.

[3] M. Aloqeely and C.Y.R. Chen. Sequencer-based data path synthesis of regular iterative algorithms. In *31st ACM/IEEE Design Automation Conference*, pages 155–160, 1994.

[4] I.E. Bennour and E.M. Aboulhamid. Register allocation using circular FIFOs. In *IFIP TC 10 WG 10.5 International Workshop on Logic and Architecture Synthesis*, pages 243–257, Grenoble, December 1995.

[5] 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.

[6] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, San Francisco, 1979.

[7] M.R. Garey, D.S. Johnson, G.L. Miller, and C.H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic and Discrete Methods*, 1(2):216–227, June 1980.

[8] S.H. Gerez. Combinatorial optimization problems related to the use of sequential read-write memories in high-level synthesis. Technical Report EL-BSC-079N94, University of Twente, Department of Electrical Engineering, December 1994.

[9] 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.

[10] A. Heubi. Architecture faible consomation pour le traitement numérique du signal. Technical Report 330 PE 10/92, Institute of Microtechnology, University of Neuchâtel, October 1992.

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

[12] 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.

[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] A. Oláh, S.H. Gerez, and S.M. Heemstra de Groot. Scheduling and allocation for the high-level synthesis of DSP algorithms by exploitation of data transfer mobility. In *International Conference on Computer Systems and Software Engineering, CompEuro 92*, pages 145–150, May 1992.

[15] 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.

[16] K.K. Parhi. Calculation of minimum number of registers in arbitrary life time chart. *IEEE Transactions on Circuits and Systems - II: Analog and Digital Signal Processing*, 41(6):434–436, June 1994.

[17] 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.

[18] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, A Systems Perspective.* Addison-Wesley, Reading, Massachusetts, second edition, 1993.

[19] 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.