A New Technique for Exploiting Regularity in Data Path Synthesis

Mohammed Aloqeely and C.Y. Roger Chen

Department of Electrical and Computer Engineering
Syracuse University, Syracuse, NY 13244-1240

Abstract

Exploiting Regularity has been the key to the success of many techniques for digital systems design. This paper presents a novel approach for exploiting the regularity in memory access that exists in many DSP and matrix computations, in order to reduce the access delay of memory and to cut down hardware cost. In this approach, data (variables) that have regular access patterns are not stored in a random access memory element; instead they are kept floating in special storage structures called sequencers, thus avoiding the bottleneck of accessing random access memories and register files and saving the overhead of memory address generation and decoding. A theoretical foundation for modeling the allocation of two types of sequencers, namely queues and stacks, is established. In addition, algorithms are developed to map variables to queues and stacks and to integrate them into conventional high-level synthesis procedures. Experimental results show an encouraging improvement in the performance of designs as well as a significant reduction in hardware cost.

1: Introduction

The high-level synthesis of digital systems from behavioral descriptions has gained a lot of attention from researchers in the CAD community during the last few years [1-4]. The two main steps in the synthesis process are the scheduling of operations in the graphical representation (e.g., DFG or SFG) to control steps (c-steps) and the allocation of hardware that implements the schedule [1].

The allocation task is usually divided into three subtasks, namely, functional units (FUs) allocation, interconnection allocation and storage allocation [1,2]. The problem of allocation has been addressed by many researchers including [3,4]. Most of these researchers have assumed a hardware model that includes a set of FUs, a set of interconnection buses, and a set of registers or register files. Based on this model storage allocation maps variables to registers or register files [2-4]. The assignment of variables to memory locations in register files offers the designer the random access advantage that allows the sharing of a memory location by more than one variable if their lifetimes do not overlap. Unfortunately, to enjoy the flexibility of random access capability, one has to pay the price of its undesirable attributes. If the behavior incudes a repetitive and regular processing of large streams of data then the size of register files can become relatively large, which not only adds more address generation and decoding hardware, but also leads to longer access delay due to decoding circuitry and long data driving lines. These problems motivate the search for new alternative methods for storage allocation.

In this paper we propose the use of what we refer to as *sequencers* as an alternative to register files. Sequencers [10,11], which are best exemplified by queues and stacks, depend on the sequence in which variables are written and read to guarantee correct data retrieval operations. Furthermore, sequencers do not include decoders and hence do not suffer from the disadvantages of random access memories. What motivates the use of sequencers is that after scheduling and FUs allocation, all operands to all FUs and all of the sequencing information about these operands (e.g., the control step at which an operand becomes available and the control step at which it is needed by an FU, etc.) are already specified. Therefore it is possible to arrange that the operands stay floating in the data path and move in a pipeline-like fashion between functional units and sequencers without the need for storing them in a random access memory element. Moreover, we have found that most algorithms of ASIC applications in signal processing and matrix computations contain a very high degree of regularity in the way variables are created and needed. Thus when, how and in what sequence the variables will be needed are usually very much predictable.

Most previous approaches for storage allocation completely ignore regularity and always store variables in memory or register files. Furthermore, very few techniques have been proposed to design special storage structures in order to avoid accessing memory, but nevertheless such attempts were mainly problem specific [8]. Unlike the work presented herein, none of the previous approaches, gives a formal description of such a class of special storage structures and provides a theoretical foundation and effective procedures for the mapping of variables to these structures.

In this paper we will restrict our discussion of sequencers to queues and stacks. We present formal definitions and solutions to the problem of allocating variables to queues and the problem of allocating variables to stacks. We also give a general strategy for applying the proposed allocation procedures to integrate with any conventional storage allocation scheme. The rest of the paper is organized as follows. In the next section the basic hardware concepts and model are presented. The allocation procedure is given in Section 3. Examples and experimental results are given in Section 4 and conclusions are presented in the last section.

2: Target Architecture

We start by giving definitions of some of the terms that we used in this paper.

Definition 1: A *sequencer* is a collection of registers grouped together in a near neighbor connection manner that enables data to move through the registers in a pipeline fashion. In addition, individual registers are not randomly accessible, yet instead, data can

enter and exit the sequencer through one or more of the “end” registers.

By virtue of their simple and regular structure, sequencers have the following interesting properties:

a. The data transfer time (i.e., access delay) of the sequencer is almost the same as that of its individual registers and is independent of the size (i.e., the number of registers) of the sequencer.
b. The number of control lines of the sequencer (e.g., push and pop signals in Fig. 1 (b)) is relatively small and is independent of the size of the sequencer.

Several examples of sequencers are shown in Fig. 1. Figures 1 (a) and 1 (b) show a queue and a stack respectively. Fig. 1 (c) shows a bidirectional queue which can enter and output data from both ends, and thus can function as a queue or as a stack. Fig. 1 (d) shows a bidirectional ring that enters and outputs data through one of the registers. The ring can be designed to perform multiple shifts in both directions to facilitate accessing data at different times. The designer can modify the designs of these sequencers or even come up with other types of sequencers that are tailored towards the nature of the application(s) to be implemented.

Basically if a variable *v* is allocated to a queue *Q* then *v* is stored in *Q* (i.e., is added at the head of *Q*) at *v*'s write step and is dequeued (i.e., removed form the tail of *Q*) at the read step of *v.* If a variable *v* is allocated to a stack *STK* then it is pushed on top of *STK* at *v*'s write step and is popped up from the top of *STK* at *v*'s read step.

The proposed hardware model is organized as follows: A set of FUs execute the operations. Inputs and outputs to functional units can be stored in sequencers or in register files. The FUs and memory elements communicate through buses. If more than one bus needs to be connected to the input of an FU then they are connected through multiplexers. Outputs of FUs are connected to buses through tri-state drivers. The system uses a two phase clock. Our objective is to eliminate register files or at least make them as small in size as possible, by attempting to assign as many variables as possible to sequencers so that the whole data path will have an architecture that resembles a large pipeline consisting of the FUs and the sequencers. It should be emphasized that our approach can be applied to more sophisticated hardware models (e.g., models that support pipelined functional units, or functional pipelining, etc.). This simple model has been chosen for the purpose of illustrating the proposed approach in a clear and abstract manner.

3: Problem Formulation

Given a set of variables *S*, where $S=\{v_1,.., v_n\}$, the objective is to find a mapping from *S* to a number of sequencers (i.e., queues or stacks) that optimizes a cost function (e.g., number of sequencers or total number of registers). We assume that scheduling and FU allocation have already been done. Therefore every variable *v* that belongs to *S* is described by the following information: (Label, Write Step, Read Step, Source, Destination). The *label* is basically an integer identifier, the *write step* (denoted as *WS*(*v*)) is the c-step at which *v* is defined (i.e., written by a source), the *read step* (denoted as (*RS*(*v*)) is the c-step at which *v* is read by a destination. The *source* and *destination* are integer values to represent the FU or I/O port that produces and uses variable *v*, respectively. If a variable *v* is written/read more than once then variable splitting [4] is used to convert it to a single write, single read variable.

Definition 2: A *control sequence* of a sequencer is the sequence of values assigned to its control lines during all the c-steps of the schedule. A control sequence *CS* of a sequencer *satisfies* the write/read requirements of a variable *v* if applying *CS* on the sequencer guarantees that *v* is written in the sequencer at *v*'s write step and is fetched from the sequencer at *v*'s read step.

In general, two types of constraints have to be dealt with in allocating variables to sequencers. First, we have to make sure that there is no access conflict between variables allocated to the same sequencer. Second, we have to make sure that there is no control sequence conflict between variables allocated to the same sequencer (i.e., there is a control sequence that satisfies the write/read requirements of all variables allocated to a sequencer).

In what follows, we will address the problem of allocating variables to queues and the problem of allocating variables to stacks. The primary objective in both cases is to minimize the number of queues and stacks, which also helps to reduce the total number of registers and interconnect required. Detailed descriptions of algorithms are omitted due to space limitations.

3.1: Allocating variables to queues

In allocating variables to queues, a variable *v* is added to the head of a queue *Q* by shifting it into *Q*, and is dequeued from *Q* by shifting it out of *Q*. Based on this scheme, the allocation procedure is somewhat complicated because the shift-register-like structure imposes some constraints on the mapping procedure. If a variable is mapped to a queue of size *n* then the number of c-steps between its write step and read step (i.e., its *lifetime interval length* = (read step - write step)) must be greater than or equal to *n* because at least *n* c-steps are needed to shift the variable through the registers of the queue. It should be stressed that the lengths of the lifetime intervals of variables allocated to a queue need not be equal. For example v_1 and v_2 in Fig. 2 can be mapped to the same queue of

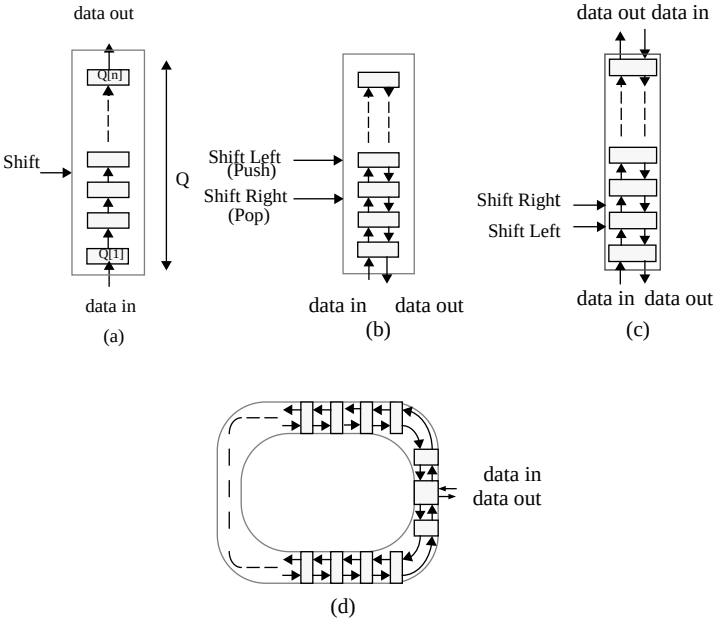

Fig. 1: Examples of sequencers. (a) A queue implemented as a unidirectional shift register (b) A stack implemented as a bidirectional shift register. (c) A bidirectional queue. (d) A bidirectional ring.

size 3 because the control sequence shown in the figure satisfies the write and read timing requirements of both of them.

To get a clear view of how variables can be grouped in the same queue, we start by giving the following definitions.

Definition 3: A *valid queue allocation* $P_{i,j,l} = [l,CS_j]$ (where l is an integer value that is less than or equal to the length of v_i and CS_j denotes a control sequence) is defined for each variable v_i such that if the control sequence CS_j is applied on a size l queue, it will satisfy the read and write requirements of v_i. In other words, it represents one valid control sequence for allocating v_i to a size l queue. The *set of valid queue allocations* $\Psi(v_i)$ is defined as the set that includes all $P_{i,j,l}$'s of v_i (i.e., the set of all possible control sequences for allocating v_i to queues of any size).

Using the above definitions, grouping a set of variables into a queue, such that no access or control sequence conflict exists between variables is formalized by the following theorem.

Theorem 1: A set of variables $V = \{v_1, v_2, ... v_k\}$ can be allocated to the same queue iff:

1. $RS(v_i) \neq RS(v_j)$ and $WS(v_i) \neq WS(v_j) \quad \forall\ v_i, v_j \in V$; and
2. $\Psi(v_1) \cap \Psi(v_2) \cap ... \cap \Psi(v_k) \neq \{\}$.

Proof:

Sufficiency: Obviously, if condition 1 holds then no read or write conflicts exists. If condition 2 holds then there must exist at least one common valid queue allocation for all variables in V (i.e., a queue of size l and a control sequence for this queue that satisfies the read and write requirements of all the variables in V). Therefore, if conditions 1 and 2 hold then all variables in V can be allocated to the same queue.

Necessity: If condition 1 does not hold then there must be a read or write conflict between two or more variables and hence they cannot be allocated to the same queue. If condition 2 does not hold then there must be at least one variable of V whose read or write timing requirements are not satisfied and thus the set of variables cannot be allocated to the same queue. ❑

Consider v_1 and v_2 in Fig. 2 as a simple example.

$P_{1,1,3}$ = [3, (shift shift shift shift no shift shift)] $\in \Psi(v_1)$.

$P_{2,1,3}$ = [3, (shift shift shift shift no shift shift)] $\in \Psi(v_2)$.

Since [3, (shift shift shift shift no shift shift)] is in both $\Psi(v_1)$ and $\Psi(v_2)$ then $\Psi(v_1) \cap \Psi(v_2) \neq \{\}$ and therefore v_1 and v_2 can be allocated to the same queue of size 3 as shown in the figure. We chose not to list all elements of $\Psi(v_1)$ and $\Psi(v_2)$ because of space limitations.

The above theoretical discussion illustrates that the grouping of variables into queues involves searching a huge search space especially if the number of variables is large. Therefore, it is crucial to find a way to reduce the size of the search space in order to find a practical allocation method. The remedy of this problem comes by taking into consideration that we are dealing mostly with applications that exhibit a certain degree of computational regularity. In such applications variables are clustered into classes of variables or "clusters" where the variables of each cluster have the same lifetime interval length and have distinct yet consecutive write and read steps. In our approach, we take advantage of this observation to reduce the search space and simplify the allocation process by dealing with clusters of variables instead of dealing with individual variables. The allocation procedure is divided into two phases. In the first phase, variables are grouped into clusters according to their lifetime interval lengths, such that all variables in a cluster

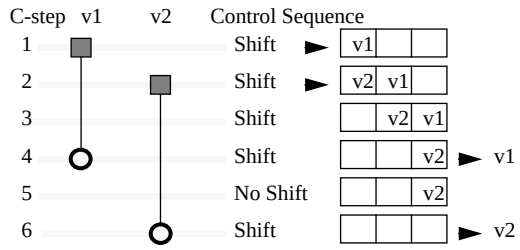

Fig. 2: $v1$ and $v2$ allocated to a queue of size 3.

have the same lifetime interval length, but have distinct write steps. In this way, all variables in a cluster need to stay floating (or to be "delayed") for an equal number of clock cycles. Moreover, since they have distinct read and write steps, they can share a queue that delays them the required number of clock cycles. Then, in the second phase, clusters of variables that do not have conflicting control sequences are merged and allocated to queues. We have adopted a simple sufficient condition for merging a group of clusters into a queue of size k, which is as follows:

1. No two variables at two different clusters have lifetime overlaps.
2. k is greater than or equal to the maximum *lifetime density* (i.e., the maximum number of live variables at any c-step) of all clusters and is less than or equal to the minimum lifetime interval of all variables in all clusters.

It can be seen easily that the first condition guarantees that there will be no conflict in control sequences. The second condition guarantees that the size of the queue is large enough to hold all the variables assigned to the queue and small enough to flush all variables out at the required time. In this way a queue can be viewed as a k-cycle delay element (or more precisely, at least k-cycle delay element).

Finally, the size of a queue can often be further reduced (*post optimized*) after performing the initial allocation. The idea is to detect if there is a number of c-steps during which, no variable is shifted in or out of a queue. Then, an equivalent number of registers are removed from the queue. The control sequence compensates for the missing registers (delays) by freezing (i.e., not shifting) the new queue an equivalent number of c-steps.

3.2: Allocating variables to stacks

The stack allocation approach is based on the notion of stack compatibility. A compatibility relation is established between variables that can share the same stack. In the following paragraphs we will introduce this concept and show how it is used in the allocation approach.

Definition 5: The lifetime of a variable v_1 *includes* the lifetime of another variable v_2 iff:

(a) $WS(v_1) < WS(v_2)$; and

(b) $RS(v_1) > RS(v_2)$.

Theorem 2: A set of variables V can be allocated to the same stack iff for any two variables v_i and v_j in V, one of the following conditions holds:

1. The lifetime intervals of v_i and v_j do not intersect.

2. The lifetime of v_i includes the lifetime of v_j or vice versa.

The proof is omitted due to space limitations. ❑

Definition 6: Two variables v_1 and v_2 are *stack compatible* iff their lifetime intervals do not intersect or the lifetime of one of them includes the other's.

Therefore, the problem can be represented graphically as illustrated in Fig. 3 (d). Every vertex in ***G*** represents a variable v_i in *S*. There is an edge between v_i and v_j in ***G*** iff either condition 1 or 2 holds (i.e., v_i and v_j are stack compatible). Hence the stack allocation problem reduces to the graph clique partitioning problem, which is NP-complete for general graphs [5]. Similar to the well known *interval* graphs for which the coloring problem is tractable (i.e., computable in polynomial time) [19], the resulting compatibility graph originates from relations between intervals, which makes it deceivingly appealing that the clique partitioning for such a graph or equivalently, the coloring of its complement (i.e., the conflict graph) could be computed in polynomial time. Nonetheless, the truth of the matter is that the clique partitioning of stack-compatibility graphs is NP-complete. To verify this claim we will introduce some of the basic relationships between intervals on a line and their induced graphs.

Given a collection of intervals on a line, each pair of intervals will satisfy exactly one of the following properties.

- *Overlap*: The two intervals intersect but neither properly contains the other.
- *Containment*: One of the two intervals properly contains the other.
- *Disjointness*: The two intervals have empty intersection.

A graph ***G*** is called an *overlap* graph if its vertices may be put into one-to-one correspondence with the collections of intervals on a line such that two vertices are adjacent in ***G*** iff their corresponding intervals overlap (not just intersect) [19].

To be more formal, let $\Gamma = \{I_x \mid x \in \boldsymbol{V}\}$ be a collection of intervals on a line. The pairs of distinct indices (which represent edges) are partitioned into three mutually disjoint sets ***A, B*** and ***C*** as follows: For distinct $x, y \in V$**,**

$(x, y) \in \boldsymbol{A}$ iff $\varnothing \neq I_x \cap I_y \neq I_x, I_y$
(i.e., the intervals overlap[1]);

$(x, y) \in \boldsymbol{B}$ iff either $I_x \subset I_y$ or $I_y \subset I_x$
(i.e., one interval properly contains the other;

$(x, y) \in \boldsymbol{C}$ iff $I_x \cap I_y = \varnothing$
(i.e,. the intervals are disjoint).

Thus we have that (***V, A***) is the overlap graph represented by Γ, (***V, A+B***) is the interval graph represented by Γ and (***V, B+C***) is the stack compatibility graph represented by Γ as depicted in Fig. 3. Unlike the case in interval graphs, the coloring problem for overlap graphs is NP-Complete although the problem of finding the maximal clique and the problem of finding the maximal *stable set* (i.e., maximal size subset of vertices no two of which are adjacent) for them are tractable [12].

Now, our claim about the time complexity of the clique partitioning of stack-compatibility graphs is verified by the following theorem.

Theorem 3: The problem of mapping a set of variables to a minimum number of stacks is NP-Complete.

1. In the special case when I_x coincides with I_y it is assumed that (x, y) belongs to *A* rather than *B*.

Proof:

It is sufficient to show that this problem is isomorphic to the coloring problem in *overlap* graphs which has been shown to be NP-complete [12]. Obviously the stack mapping problem herein which is modeled as a clique partitioning of the stack compatibility graph, can be modeled as a coloring problem for the conflict graph (which is the complement of the compatibility graph). It turns out that the resulting conflict graph is identical to the classical overlap graph with the exception that the latter deals with continuous intervals while the former deals with discrete intervals. Therefore, the stack allocation problem is NP-Complete. ❑

In a previous paper, we have proposed solutions for this problem using integer linear programming and interconnection guided heuristic search [11]. In this paper, we present a new greedy heuristic that takes advantage of the special properties of stack compatibility graphs.

Heuristic for allocating variables to stacks: Once the problem has been identified as a clique partitioning problem, any of the effective techniques (e.g., Tseng and Siewiorek [3]) can be applied to solve the problem. However, we have used a new method that exploits the special properties of stack compatibility graphs. As mentioned earlier, the stable set problem and the clique problem are tractable for *overlap* graphs. Gavril [12] designed an algorithm that computes the maximum stable set of an overlap graph (which corresponds to a maximum clique in our stack-compatibility graph) in $O(n^3)$ time, where n is the number of vertices. His technique is adopted in our allocation heuristic. The heuristic finds the maximum clique in the stack compatibility graph, assigns its variables to a stack, removes the variables form the graph and the process is repeated until the graph is empty. While this method of course does not guarantee the minimality of the total number of stacks, it however attempts to create at least a number of reasonably large stacks that can capture a large portion of the variables. The variables that belong to the remaining small size stacks (as we will see later) can be grouped in register files instead in the final design. The rational behind this gready policy is to try to limit the

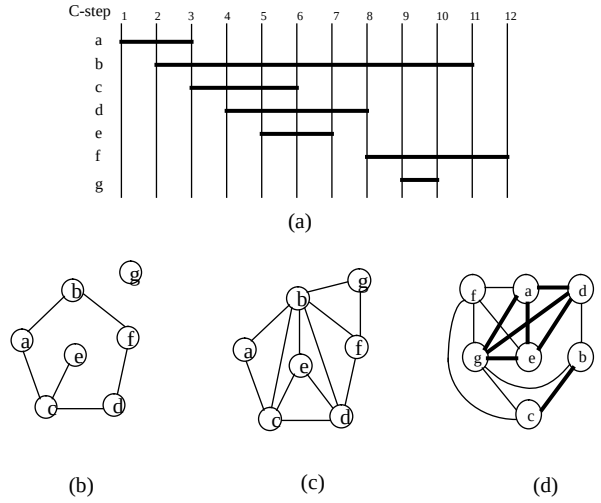

Figure 3: (a) Lifetime intervals. (b) The overlap graph = (*V, A*).

(c) The interval graph =(*V, A+B*). (d) The stack comp. graph ***G*** =(*V, B+C*).

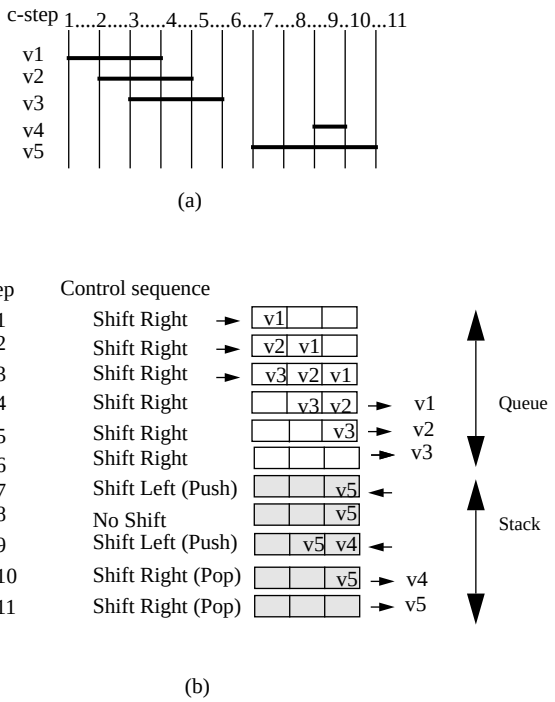

Figure 4: The merge of a stack and a queue into a bidirectional queue.
(a) Lifetimes of variables. (b) Status of the bidirectional queue in all c-steps.

total number of memory elements (i.e., sequencers and register files) from becoming too large; a situation which not only would increase the interconnection hardware, but also would introduce more delay due to multiplexers, increase of capacitive loads on buses [7] and so fourth. The reader may refer to [10] for the detailed description of the algorithm.

3.3: Putting it all together

Since the adequacy of sequencers allocation procedures is problem dependent because they actually try to detect and utilize access patterns regularity, we have suggested a general interactive procedure that can be used in order to assist the designer in applying the most suitable allocation scheme. Our global allocation strategy has three stages. In the first stage we try to allocate all variables to queues since they are the least costly choice. Then based on a rejection criterion (e.g., register utilization or size of queue [10]), we reject some of the queues that do not meet the minimum allowed utilization. Those variables which have been allocated to rejected queues will be used as input to the next stage in which we try to allocate variables to stacks. Similarly stacks that do not meet the minimum allowed utilization will be rejected. Next, compatible stacks and queues are merged into bidirectional queues as illustrated in Fig. 4 [10]. The remaining variables can then be used as input to any conventional allocator to be allocated to register files. The main steps of the procedure are shown in Fig. 5.

4: Experimental results

The proposed algorithms have been implemented on a SUN SPARCstation I running SUN OS. To demonstrate the advantages

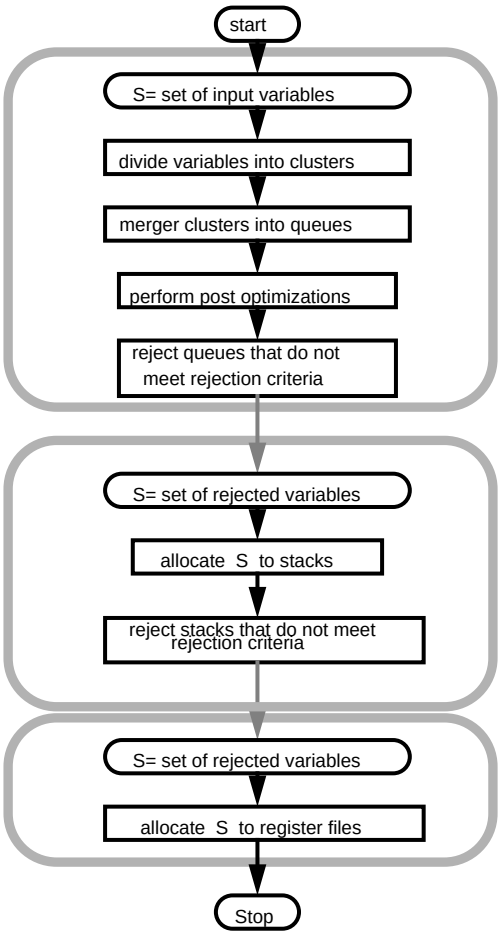

Fig. 5. The main steps in the general allocation procedure

of the proposed approach, various examples were used in the experiment. Benchmark examples of the 1988 workshop on high-level synthesis [9] were not included because they are mostly irregular or in an irregular form, and thus do not suit our approach that exploits regularity. Nevertheless we found a large number of applications that give excellent results under our approach. Two of which are presented in this paper. The CPU execution time for all of them is less than one second.

The first example is an FIR filter used in [6] and [7]. The description of the FIR filter is given by:

$$y_n = \sum_{j=0}^{N} a_j \cdot x_{n-j}$$

A special case of the FIR filter (the 16-point FIR) is a well known example in the high-level synthesis literature [2]. To show how well our approach scales, we applied our method on the FIR example with several values of N and we used the more regular FIR representation given in [6]. The graphical representation of the FIR filter with N=4 is shown in Fig. 6 (a). Fig. 6 (b) shows the lifetime intervals of variables when scheduled with a multiplier and an adder. The results of this example for the case when N=4 and for

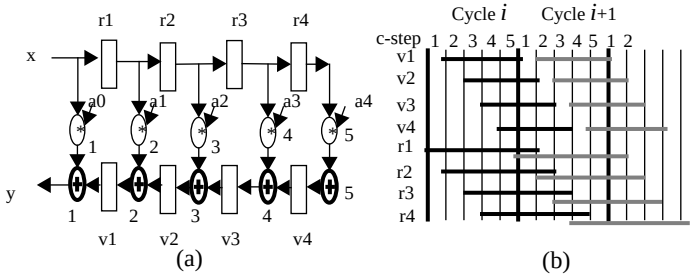

Fig. 6 : Example 2: FIR filter given by : $y_n = \sum_{j=0}^{N} a_j \cdot x_{n-j}$

(a) The graphical representation. (b) Lifetime intervals of variables.

the general case when N=k are shown in Table 1 and are compared with those obtained by another allocation scheme that allocates to register files. In our comparison, it is assumed that the access delay of a register file equals the decoding delay plus the delay associated with a register transfer operation which is denoted as c, whereas the access delay of a queue equals c. A decoder with n outputs can be realized by (n-1) 1-to-2 decoders. The delay of a 1-to-2 decoder is denoted as d. The first two entries in the table represent address generation and decoding cost and the last entry represents access delay of memory elements. It is clear that our approach eliminates address generation and decoding cost and reduces access delay. Fig. 7 shows the improvement in access delay obtained by using sequencers over using register files, versus N, assuming that the ratio c/d=4. The figure shows that our approach gives a significant speedup that increases as N (and hence the total number of registers) gets larger.

The second example is a matrix vector multiplication algorithm that computes AX=Y where A is an $M \times N$ matrix and X and Y are vectors of length N and M respectively. The results of this example for the case when M=400 and N=160, implemented using different numbers of FUs are shown in Table 2.

5: Conclusions

A novel approach for exploiting regularity in data path synthesis is presented. The concept of using a more sophisticated hardware model that contains what we term as *sequencers* has been applied to the synthesis process, and is aimed to achieve two main objectives. First, to improve the data transfer delay of storage elements since sequencers, which are best exemplified by queues and stacks, are mainly implemented as unidirectional or bidirectional shift registers, and hence they do not suffer from decoding delays that grow proportionally with the size of a register file. Second, to eliminate the cost of memory address generation and decoding. Furthermore, algorithms and procedures have been developed for allocating variables to stacks and queues and to integrate the proposed techniques into conventional high-level synthesis procedures. Experimental results for a number of DSP applications show very encouraging improvement in performance as well as significant reduction in hardware cost. We believe that this approach opens up a new unexplored frontier for the synthesis of high performance Isaacs that have a high degree of regularity and require short clock cycles.

References

[1] M. C. McFarland, A.C. Parker and R. Composano, "The High Level Synthesis of Digital Systems," *Proc. of IEEE*, vol. 78, no. 2, pp. 301-318, Feb. 1990.

[2] D. Gajski, N. Dutt, A. Wu, and S. Lin, *HIGH-LEVEL SYNTHESIS: Introduction to Chip and System Design.* Kluwer Academic Pub., 1992.

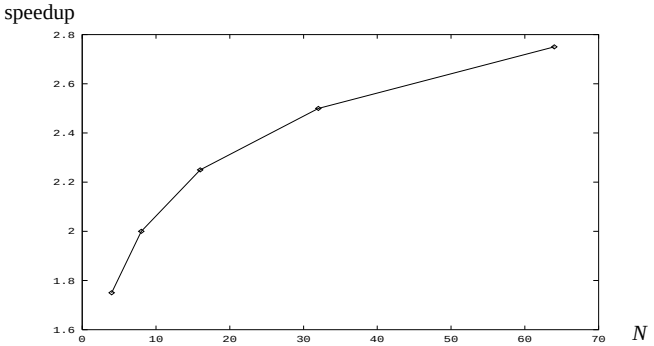

Fig. 7: Speedup of memory elements access delay versus N.

[3] C. Tseng and D. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. on Computer-Aided Design*, July 1986, pp. 379-395.

[4] F. S. Tsai and Y. C. Hsu, "STAR: An Automatic Data Path Allocator," *IEEE Trans. on Computer-Aided Design*, Sept. 1992, pp. 1053-1064.

[5] M. Garey and D. Johnson, *Computers and Intractability.* Freeman, San Francisco, California, 1979.

[6] C.Y. Roger Chen, and Michael Moricz, "A Delay Distribution Methodology for the Optimal Systolic Synthesis of Linear Recurrence Algorithms," *IEEE Trans. on Computer Aided Design*, vol. 10, No. 6, June 1991, pp. 685-697.

[7] C. Mead and L. Conway, *Introduction to VLSI systems.* Addison Wesley, 1980.

[8] K. Parhi, "Systematic Synthesis of DSP Data Format Converters Using Life-Time Analysis and Forward-Backward Register Allocation," *IEEE Trans. on Circuits and Systems-II: Analog and Digital Signal Processing*," vol. 39 No. 7, July 1992.

[9] G. Borriello and E. Detjens, "High-Level Synthesis: Current Status and Future Directions," *in Proc. of the 25th DAC*, New York, NY, June 1988, pp. 477-482.

[10] M. Aloqeely, "Sequencers: a New Alternative for Data Path Synthesis," Ph.D. Dissertation in progress at Syracuse University.

[11] M. Aloqeely and C.Y. Roger Chen, "Sequencer-Based Data Path Synthesis of Regular Iterative Algorithms," *In Proc. of the 31*st *DAC*, June, 1994, pp. 156-61.

[12] M. C. Columbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

TABLE 1 : RESULTS OF THE FIR FILTER EXAMPLE

Approach	Proposed design		Design using register files	
	N=4	N=k	N=4	N=k
# address lines to memory	0	0	5	$\approx 2.\ [\log k]$
# 1-to-2 decoders	0	0	7	$\approx 2\ .\ k$ -1
# registers	9	$\approx 2.k$ +1	9	$\approx 2\ .k$ +1
# register files	0	0	2	2
# sequencers	2 queues	2 queues	0	0
max. memory access delay	$\approx c$	$\approx c$	$\approx c + 3.d$	$\approx c + d.[\log k]$

c: Delay associated with accessing an individual register.
d: Propagation delay of a single 1-to-2 decoder.

TABLE 2 : RESULTS OF THE MATRIX VECTOR MULTIPLICATION EXAMPLE

Approach	Proposed design		Design using register files	
	4 + 4 *	10 + 10 *	4 + 4 *	10 + 10 *
# address lines to memory	0	0	24	50
# 1-to-2 decoders	0	0	153	141
# registers	157	151	157	151
# register files	0	0	4	10
# sequencers	4 queues	10 queues	0	0
max. memory access delay	$\approx c$	$\approx c$	$\approx c + 6.d$	$\approx c + 5.d$