VLSI Design
1994, Vol. 1, No. 3, pp. 217–232
Reprints available directly from the publisher
Photocopying permitted by license only

Using PDM on Multiport Memory Allocation in Data Path

CHIEN-IN HENRY CHEN
Dept. of Electrical Engineering, Wright State University, Dayton, OH

(Received November 13, 1989, Revised March 10, 1990)

A data path consists of memory elements (i.e. registers), data operators (i.e. ALUs) and interconnection units (i.e. buses) to control the data transfers in the digital system. Many approaches to memory synthesis have been proposed in the literature. However, only single port memory is considered for register allocation and no efficient synthesis approach for multiport memory synthesis. In this paper, an efficient method, *Partitioned Dependence Matrix (PDM)*, is presented for memory synthesis which deals not only with single port memory synthesis but also multiport memory synthesis according to the design constraints. With suitable modifications, the proposed technique can also be applied to multiport memory synthesis in which the maximum number of read ports is different from the maximum number of write ports. Therefore, the entire design space is explored and has the capability to handle early architectural design exploration so that the quality of designs produced by an automatic synthesis tool is more adequate for production use in comparison to manual design. Illustrations of applying this method to different synthesis examples are presented. Results and improvements over previous techniques are demonstrated. A key element in our approach is the successful adoption of techniques originally developed for problems in test generation to the field of memory synthesis.

Key Words: *Multiport memory; Clique partitioning; Resource allocation; Synthesis*

INTRODUCTION

From the input specification, the synthesis system produces a description of register-transfer (RT) level structure that realizes a specified behavior. The structure may be divided into two parts, a data path and a control unit. The data path consists of memory elements (i.e. registers), data operators (i.e. ALUs) and interconnection units (i.e. buses) to control the data transfers in the digital system. The control unit is a finite state machine capable of generating the control signals to evoke a data transfer in the data path so as to produce the specified behavior. High-level data path synthesis is concerned with the automatic generation and allocation of registers, ALUs and buses. Many approaches to automated data path synthesis have been proposed in the literature [1–11, 17–22]. However, current high-level synthesis tools lack the capability to handle early architectural design exploration so that the quality of designs produced by an automatic synthesis tool is not completely adequate for production use in comparison to manual design.

An interesting and powerful approach to all phases of data path synthesis was proposed by Tseng and Siewiorek [1, 3]. In their approach, the problems of register, operator and bus synthesis are cast into the "Generalized Clique Partioning" (GCP) problem in which it is necessary to partition the nodes of a compatibility graph into a set of disjoint clusters. Their objective is to partition the nodes of this graph in such a way that the minimal number of disjoint clusters is obtained. In terms of the corresponding data path element, this implies that the minimal hardware cost for a given degree of system concurrency is thereby achieved.

One of the drawbacks of their approach is that the time complexity of the GCP algorithm is large and involves following multiple paths with backtracking; this occurs especially in bus system synthesis. To overcome these problems, a more efficient approach, Weighted Cluster Partitioning (WCP), was presented in [12], which eliminates the need for backtracking. WCP, and a variation called WCP II, were shown to produce designs requiring the fewest number of buses in large synthesis examples. These algorithms

have polynomial time complexity and also yield excellent results when applied to the test generation problem of Built-In Self-Test design in [13].

With suitable modification, WCP and WCP II can be applied to the other two phases of resource allocation. However, WCP or WCP II like GCP can only be applied to allocate registers having the property of disjoint access requirements to single port memory modules. Obviously, there are several advantages to merge registers to form multiport memories, such as saving interconnections, reducing the number of multiplexers, minimizing the chip area, and being more easily testable due to a smaller number of design modules. Moreover, multiport memory synthesis can be applied to many applications according to the architectural design. In [11], it was shown how the problem of finding the maximal set of registers which can be grouped into a multiport memory module can be treated as the 0-1 integer programming problem. A branch-and-bound strategy [14] was then used to obtain the solution. As stated in [11], this technique uses a sequential approach (i.e., generating the memory modules in sequence) which generates a locally optimal solution and may not generate the globally optimal solution of producing the minimum number of multiport memory modules.

In this paper, we exploit and modify the algorithm that has been proposed to solve the test generation problem in Ref. [15, 16] to generate a more globally optimal solution for multiport memory synthesis than does the previous techniques [11]. It will be shown how these algorithms can be modified and adapted to the multiport memory synthesis. The limits of the problems we aim to solve and the definitions and algorithms of the proposed PDM techniques will be discussed. The efficiency of this approach is illustrated through some explicit synthesis examples. Section 5 describes the PDM techniques can be adapted and modified for the synthesis of multiport memory with ports of different type. Simulation results and comparisons with other techniques are presented. Following this the complexity analysis of PDM techniques are discussed. Finally, the results of this work are summarized.

DEPENDENCE MATRIX

Let R_i and R_j be two registers. For single port memory synthesis, R_i and R_j can be allocated to the same memory module if and only if R_i and R_j have disjoint access times. However, for multiport memory synthesis, R_i and R_j can be allocated to the same memory module with K ports, where $K \geq 2$, even if R_i and R_j are accessed simultaneously.

In the following, *Lemma 1* and *Lemma 3* are the design constraints of multiport memory synthesis with ports of **same** and **different** type respectively, which are adapted from Ref. [11] for the convenient discussion of the algorithm proposed in this paper.

Lemma 1: Let $R_1, R_2, \ldots, R_m$ be registers. $R_1, R_2, \ldots$, and R_m can be allocated to the same memory module with K ports if and only if no more than K of these registers are accessed simultaneously.

The ports in the multiport memory module may not all be of the same type, i.e., some may be read only while others may be write only or read/write. First, we will consider only multiport memory modules with the same type of ports (i.e. read/write). Second, with suitable modification, the technique can be applied to synthesis of memory modules with ports of different type, and this part of discussion will be presented in the later section.

Definition 1: A **dependence matrix DM**(C) for a code sequence C has s rows and n columns. Each row represents one of the control steps in the code sequence C and each column represents one of the registers. An entry is 1 if and only if the corresponding register is accessed in the specified control step. All other entries are 0.

The dependence matrix $DM(C)$ is easily calculated from a given code sequence C. In the following discussion, we use the code sequence of Ref. [1], shown in Table I, as an example. The $DM(C)$ for the code sequence C in Table I is shown in Table II.

TABLE I
Example Code Sequence of Ref. [1]

```
S1: R3 = R1 + R2, R12=R1;
S2: R5 = R3 -R4, R7 = R3 * R6, R13=R3;
S3: R8 = R3 + R5, R9 = R1 + R7, R11 = R10 / R5;
S4: R14 = R11 AND R8, R15 = R12 OR R9;
S5: R1 = R14, R2 = R15;
```

TABLE II
DM(C) for Table I

	Registers														
	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
S1	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0
S2	0	0	1	1	1	1	1	0	0	0	0	0	1	0	0
S3	1	0	1	0	1	0	1	1	1	1	1	0	0	0	0
S4	0	0	0	0	0	0	0	1	1	0	1	1	0	1	1
S5	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1

PARTITIONED DEPENDENCE MATRIX

The first step in finding the minimum number of memory modules is to determine the set of registers which can be grouped into the same memory module. This can be done by partitioning the dependence matrix *DM(C)* to form a partitioned dependence matrix.

Definition 2: Given a code sequence *C* and a memory module with *K* ports, a **partitioned dependence matrix PDM(C,K)** corresponding to *DM(C)* is formed by partitioning the columns of the *DM(C)* into sets so that:

(1) each row of a set has at most *K* 1-entries.

(2) the number of sets *p* is a minimum.

The partitioned dependence matrices *PDM(C,K)*, *K* = 1, 2, 3, 4 corresponding to Table II are shown in Table III (a)–(d). For each partitioned dependence matrix, there is a corresponding partition *p* of the registers in which all registers belonging to the same set of columns of *PDM(C,K)* are allocated to the same memory module with *K* ports. For example,

TABLE III(a)
Partitioned dependence matrix for Table II. *PDM(C,1)* for Table II

	Registers														
	R1	R4	R2	R5	R3	R14	R6	R8	R7	R12	R9	R13	R10	R15	R11
S1	1	0	1	0	1	0	0	0	0	1	0	0	0	0	0
S2	0	1	0	1	1	0	1	0	1	0	0	1	0	0	0
S3	1	0	0	1	1	0	0	1	1	0	1	0	1	0	1
S4	0	0	0	0	0	1	0	1	0	1	1	0	0	1	1
S5	1	0	1	0	0	1	0	0	0	0	0	0	0	1	0

TABLE III(b)
PDM(C,2) for Table II

	Registers														
	R1	R2	R4	R5	R3	R6	R8	R14	R7	R9	R12	R13	R10	R11	R15
S1	1	1	0	0	1	0	0	0	0	0	1	0	0	0	0
S2	0	0	1	1	1	1	0	0	1	0	0	1	0	0	0
S3	1	0	0	1	1	0	1	0	1	1	0	0	1	1	0
S4	0	0	0	0	0	0	1	1	0	1	1	0	0	1	1
S5	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1

TABLE III(c)
PDM(*C*,3) for Table II

	Registers														
	R1	R2	R3	R5	R6	R14	R4	R7	R8	R9	R12	R13	R10	R11	R15
S1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0
S2	0	0	1	1	1	0	1	1	0	0	0	1	0	0	0
S3	1	0	1	1	0	0	0	1	1	1	0	0	1	1	0
S4	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1
S5	1	1	0	0	0	1	0	0	0	0	0	0	0	0	1

TABLE III(d)
PDM(*C*,4) for Table II

	Registers														
	R1	R2	R3	R4	R5	R6	R8	R14	R15	R7	R9	R10	R11	R12	R13
S1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0
S2	0	0	1	1	1	1	0	0	0	1	0	0	0	0	1
S3	1	0	1	0	1	0	1	0	0	1	1	1	1	0	0
S4	0	0	0	0	0	0	1	1	1	0	1	0	1	1	0
S5	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0

as seen in Table III (a)–(d), to form register files for the code sequence shown in Table I, we need eight single-port memory modules or four 2-port memory modules or three 3-port memory modules or two 4-port memory modules.

PDM: ALGORITHM, IMPLEMENTATION AND RESULTS

In this section, a detailed discussion of the algorithm used to form *PDM*(*C,K*) from *DM*(*C*) is presented. We will make use of the technique proposed to solve the graph coloring problem posed in Ref. [16] and show that, with a suitable modification of the technique, it can be applied to solve the memory synthesis problem.

The basic components of the algorithm are as follows. The columns of the *DM*(*C*) are ordered from 1 to n where n is the number of registers. An integer number label is assigned to each column. The first column is labeled as 1. The other columns are sequentially labeled with as small a label number as possible with some constraints (for details, see *Definition* 4 below.) After that, we check if the current maximum label equals the lower bound (for details, see *Lemma* 2 below). If it does, then that maximum label is the minimum number of sets p in *PDM*(*C,K*). Those columns having the same label are allocated into the same set. In other words, the registers corresponding to columns having the same label are allocated to the same memory module. Otherwise, there is an attempt to decrease the label of that column which has the maximum label number and which is the lowest in the ordering. It can be shown that this is an efficient approach to change the labels of the columns which are lower than the maximum label column in the ordering so that the maximum label number can be decreased (i.e., the number of partitioned sets, p, is decreased). We continue this process until a minimal labeling scheme is found. Before we discuss the full details of the algorithm, we need to define some basic operations:

Definition 3: ColumnParents(CR_i): Let CR_i represent the column of *DM*(*C*) corresponding to the register R_i. This operation lists those columns (i.e. CR_j) of *DM(C)* which have a 1-entry in a row where CR_i has 1-entry in the same row and whose order (the ordinal of the column associated with a register) are less than that of CR_i in the ordering (i.e. $j < i$).

Example: In Table II, CR_1, CR_3, and CR_4 have '1' entry while CR_5 has also '1' entry in the same row.

Therefore, CR_1, CR_3 and CR_4 are called the ColumnParents of CR_5. Similarly, CR_1, CR_2, CR_3, CR_8, CR_9 and CR_{11} are the ColumnParents of CR_{12}.

Definition 4: Label(CR_i): CR_i is a column to be labeled. Assume that the columns CR_1 through CR_{i-1} have been labeled. The column CR_i is labeled with as small a positive integer as possible with the constraint that no more than K columns in *ColumnParents*(CR_i) have the same label and the number is greater than zero.

Definition 5: ColumnAncestors(CR_i): CR_i is a column whose ancestors are to be determined. Every column in the *ColumnParents*(CR_i) is defined as a member of *ColumnAncestors*(CR_i). Every column which is a *ColumnParent* of a *ColumnAncestor* of CR_i is also a member of *ColumnAncestors*(CR_i).

Example: In Table II, CR_2 is not a ColumnParent of CR_5, but it is a ColumnAncestor of CR_5. According to the definition of ColumnAncestor, ColumnParent of a ColumnAncestor is considered to be a ColumnAncestor. CR_3 is ColumnParent of CR_5, so CR_3 is a ColumnAncestor of CR_5. CR_2 is a ColumnParent of CR_3, therefore CR_2 is a ColumnAncestor of CR_5. Similarly, CR_1, CR_2, CR_3, CR_4, CR_5, CR_6 and CR_7 are the ColumnAncestors of CR_{13}.

Definition 6: Relabel(s,maxorder,maxlabel): Let s be the order of the column where the relabel operation begins. Let CR_{max} be the column which has the maximum label and lowest in order (we may have more than one column having the *maxlabel*) after the relabeling procedure and let *maxlabel* be the maximum label before the relabeling process. The order of CR_{max} is returned in *maxorder*. Starting from i = s + 1 to i = n, we label CR_i by *Label(CR_i)*. If a column is labeled as *maxlabel*, or if the last column is labeled, then the procedure is completed. If *Relabel* is called with $s = 1$, then the label of CR_1 is set to 1 and the regular procedure is executed.

Definition 7: Backtrack(s1,s2,flag,maxlabel): CR_{s1} is the column whose label is being decreased. If the backtrack procedure cannot make any improvement in the labeling, then it returns *false* in the *flag*. Otherwise, it returns *true* in the *flag*, and that indicates that there exist a column CR_{s2} whose label may be increased so that the maximum label can be decreased. Let *maxlabel* be the maximum label before the backtracking process (i.e. the label of CR_{s1}). Let V be the set of the columns whose order is lower than CR_{s1}, and S be the set of *ColumnAncestors* of CR_{s1}. The detailed procedure of *backtrack* includes two steps:

Step 1: If S is empty, then set *flag* is false and exit. Otherwise, set *flag* is true and find the largest ordered column CR_{s2}, which belongs to both S and V. Let $S = S - \{CR_{s2}\}$.

Step 2: The label of CR_{s2} should be increased as little as possible with the constraints that no more than K columns in *ColumnParents*(CR_{s2}) have the same label and the label should be less than *maxlabel*. Then, call the procedure *Relabel(s2,maxorder,maxlabel)*. After the relabeling process, if the maximum label is decreased, than *flag* is set to true and exit the *backtrack* procedure. Otherwise, go to step 1.

Example: In the code sequence of example 1: single port memory synthesis shown in Table IV, after the application of *Label* procedure the column registers CR_1 and CR_2 are labeled '1', CR_3 is labeled '2', CR_4 is labeled '3' and CR_5 is labeled '4'. Therefore, the maxorder is '5' and the maxlabel is '4'. The *Backtrack* procedure is then applied to decrease the maxlabel number among all the column registers. In the Table IV (*c*), *Backtrack*(5,3,*true*,4) indicates that CR_5 is the column register whose label is being decreased and CR_3 is the column register whose label is being increased (the *true* flag is returned, which indicates there exists the column register CR_3 whose label can be increased so that the maxlabel '4' may be decreased). The procedure *Relabel*(3,5,4) indicates that '3' is the order of the column register CR_3 where the *label* operation begins. CR_5 is the column register which has the maximum label '4' and lowest in the order. So, starting from i = 4 to 5, we label CR_i by the operation *Label(CR_i)*.

Lemma 2: The lower bound on the number of multiport memory modules can be derived from the

TABLE IV(a)
Example 1: single port memory synthesis. A code sequence

```
S1: R3 = R1 + R4;
S2: R4 = R3 * R5;
S3: R2 = R5 - R3;
```

TABLE IV(b)
DM(*C*) for Table IV(a)

	Registers				
	R1	R2	R3	R4	R5
S1	1	0	1	1	0
S2	0	0	1	1	1
S3	0	1	1	0	1

TABLE IV(c)
Example of the algorithm to form $PDM(C,1)$ for Table IV(b)

Algorithm	labels				
	CR1	CR2	CR3	CR4	CR5
Relabeled(1,5,4)	1	1	2	3	4
Backtrack(5,3,true,4)	1	1	3	3	4
Relabeled(3,5,4)	1	1	3	2	4
Backtrack(5,2,true,4)	1	2	3	2	4
Relabeled(2,3,3)	1	2	3	2	1
Backtrack(3,-,false,3)	1	2	3	2	1

TABLE IV(d)
$PDM(C,1)$ for Table IV(b)

	Registers				
	R1	R5	R2	R4	R3
S1	1	0	0	1	1
S2	0	1	0	1	1
S3	0	1	1	0	1

following equation.

$$LowerBound = \left\lceil \frac{max.\ no.\ of\ registers\ accessed\ simultaneously}{no.\ of\ ports\ in\ memory\ module} \right\rceil$$

The complete algorithm of forming $PDM(C,K)$ from $DM(C)$ can be stated as follows.

Step 1: *Order the columns CR_i of DM(C) from 1 to n, where n is the number of registers.*

Step 2: *Call Relabel(1,maxorder,maxlabel).*

Step 3: *If label of the column whose order is maxorder is equal to the LowerBound, then go to step 4. Otherwise, call Backtrack(maxorder,s2,flag,maxlabel). If flag is false, then go to step 4. Otherwise, call Relabel(s2,maxorder,maxlabel). If the label of the column whose order is maxorder is less than maxlabel, then maxlabel is set to the lower label. Return to step 3.*

Step 4: *The value of maxlabel is the minimum number of sets p in PDM(C,K). The columns of DM(C) that have the same labels belong to the same partition.*

Example 1: Consider the single port memory synthesis of the registers in the code sequence as shown in Table IV(a). The dependence matrix $DM(C)$ is shown in Table 4(b). The step-by-step operation of the algorithm is shown in Table IV(c). Using the final results from Table IV(c), the partitioned dependence matrix can be derived (see Table IV(d)). Note that three memory modules ($LowerBound = [\frac{3}{1}] = 3$) are required for the data path synthesis, i.e., they are $\{R_1, R_5\}$, $\{R_2, R_4\}$ and $\{R_3\}$.

However, using the 0-1 integer programming technique proposed in [11], the problem reduces to

$Max\ (x_1 + x_2 + x_3 + x_4 + x_5)$

Subject to

$x_1 + x_3 + x_4 \leq 1$

$x_3 + x_4 + x_5 \leq 1$

$x_2 + x_3 + x_5 \leq 1$

The formulation of the algorithm in Ref. [11] attempts to include the largest number of registers into the current memory module, a desirable result. However, our overall objective is to find the minimum number of memory modules that will cover all the registers which do not have any conflict in use during memory access. Thus, one may expect that the local optimization produced by the 0-1 integer programming technique will act to limit the degree of global optimization that can be achieved. Besides, the formulation leads to the arbitrary selection of one of several equally valid choices at each step of the algorithm. Thus, many possible solutions may be obtained. In the above problem, the algorithm may produce the first grouping of registers to be $\{R_1, R_2\}$ (that is $x_1 = x_2 = 1$). Then, the same approach is applied to the remaining registers. The final solution of example 1 by using the 0-1 integer programming technique may generate four memory modules, specifically $\{R_1, R_2\}$, $\{R_3\}$, $\{R_4\}$, and $\{R_5\}$. Note that this result is the same as the result of the first step of our algorithm, as shown in Table IV(c). This contrasts with the final result of three memory modules obtained at the final step of our procedure. Thus, the lack of any backtracking step in the 0-1 integer programming approach leads to the selection of a locally optimal but globally sub-optimal solution to the problem.

Example 2: Consider the 2-port memory synthesis of the registers in the code sequence as shown in Table V(a). The dependence matrix $DM(C)$ is shown in Table V(b). The operation of our algorithm is shown in Table V(c) and, from its final results, the partitioned dependence matrix can be derived in Table V(d) in which two 2-port memory modules ($LowerBound = \lceil \frac{4}{2} \rceil = 2$) are required, i.e., they are $\{R_1, R_3, R_5\}$ and $\{R_2, R_4, R_6\}$. However, using the 0-1 integer programming technique, it may produce three 2-port memory modules, specifically $\{R_1, R_2\}$, $\{R_3, R_4, R_6\}$, $(R_5\}$. as the final solution.

For completeness, we also used the proposed technique for multiport memory synthesis for the example code sequence in Ref. [1]. After applying the technique to the dependence matrix $DM(C)$ in Table II, we find the partitioned dependence matrices $PDM(C,K)$ for $K = 1,2,3,4$ as shown in Tables 3(a)–(d).

TABLE V(a)
Example 2: multiport memory synthesis. A code sequence

```
S1: R1 = R2 + R5, R4 = R2 * R5;
S2: R2 = R3 - R4, R5 = R3 / R4;
S3: R1 = R2 + R6, R3 = R2 * R6;
```

TABLE V(b)
$DM(C)$ for Table V(a)

	Registers					
	R1	R2	R3	R4	R5	R6
S1	1	1	0	1	1	0
S2	0	1	1	1	1	0
S3	1	1	1	0	0	1

MULTIPORT MEMORY SYNTHESIS WITH PORTS OF DIFFERENT TYPES

Consider a multiport memory module M with m ports where r ports are read only, w ports are write only, and the rest $m - r - w$ are read/write ports.

Lemma 3: Let $R_1, R_2, \ldots, R_m$ be registers. $R_1, R_2, \ldots$, and R_m can be allocated to the same memory module M if and only if the following three conditions are satisfied:

(1) no more than $m - w$ of these registers are accessed by read instructions simultaneously.

(2) no more than $m - r$ of these registers are accessed by write instructions simultaneously.

(3) no more than m of these registers are accessed by read/write instructions simultaneously.

Definition 8: A **dependence matrix DM(C,R,W)** for a code sequence C has s rows and n columns. Each row represents one of the control steps in the code sequence C and each column represents one of the registers. An entry is R (or W) if and only if the corresponding register is accessed by read (or write) instruction in the specified control step. All other entries are 0.

The dependence matrix $DM(C,R,W)$ is easily calculated from a given code sequence C. In the following discussion, we use the code sequence in Table I, as an example. The $DM(C,R,W)$ for the code sequence C in Table I is shown in Table VI.

The problem of finding the minimum number of memory modules with ports of different type for registers allocation still can be solved by partitioning the dependence matrix $DM(C,R,W)$ to form a partitioned dependence matrix $PDM(C,m,r,w)$.

Definition 9: Given a code sequence C and a memory module M with m ports where r ports are read only, w ports are write only, and the rest $m - r - w$ are read/write ports, a **partitioned dependence matrix PDM(C,m,r,w)** corresponding to $DM(C,R,W)$

TABLE V(c)
Example of the algorithm to form $PDM(C,2)$. for Table V(b)

	labels					
Algorithm	CR1	CR2	CR3	CR4	CR5	CR6
Relabeled(1,5,3)	1	1	2	2	3	2
Backtrack(5,2,true,3)	1	2	2	2	3	2
Relabeled(2,2,2)	1	2	1	1	2	2
Backtrack(2,-,false,2)	1	2	1	1	2	2

TABLE V(d)
PDM(C,2) for Table V(b)

	Registers					
	R1	R3	R5	R2	R4	R6
S1	1	0	1	1	1	0
S2	0	1	1	1	1	0
S3	1	1	0	1	0	1

is formed by partitioning the columns of the $DM(C)$ into sets so that:

(1) each row of a set has at most $m - w$ R-entries.

(2) each row of a set has at most $m - r$ W-entries.

(3) each row of a set has at most m R/W-entries.

(4) the number of sets p is a minimum.

In the following a detailed discussion of the algorithm used to form $PDM(C,m,r,w)$ from $DM(C,R,W)$ is presented. We will make use of the technique for memory synthesis with ports of same type which was proposed in the last section and show that, with a suitable modification, it can be applied to solve the multiport memory synthesis problem with ports of different type.

Definiton 10 (a): ColumnParents(CR_i) for multiport memory with READ and WRITE ports, but without READ/WRITE ports: Let $(CR_i)_r$ be the entry of the i-th column, r-th row of the dependence matrix DM. CR_j is considered to be a parent of $(CR_i)_r$ if and only if

(i) CR_j has R-entry in the r-th row where CR_i has R-entry in the r-th row; or

(ii) CR_j has W-entry in the r-th row where CR_i has W-entry in the r-th row.

ColumnParents of CR_i are then defined as a union of all parents of $(CR_i)_r$ for $1 \le r \le s$, i.e. $\cup_{r=1}^{s}$ *parents of* $(CR_i)_r$.

Example: In Table VI, $(CR_7)_1$, $(CR_7)_4$ and $(CR_7)_5$ have no parents. CR_5 is a parent of $(CR_7)_2$. CR_1, CR_3 and CR_5 are parents of $(CR_7)_3$. Therefore, ColumnParents of CR_7 are the union of all parents of $(CR_7)_r$ for $1 \le r \le 5$, i.e. $\cup_{r=1}^{5}$ *parents of* $(CR_7)_r = \{CR_1, CR_3, CR_5\}$.

Definiton 10 (b): ColumnParents(CR_i) for multiport memory with READ, WRITE and READ/WRITE ports: Let $(CR_i)_r$ be the entry of the i-th column, r-th row of the dependence matrix DM. CR_j is considered to be a parent of $(CR_i)_r$ if and only if CR_j has R-entry or W-entry in the r-th row where CR_i has R-entry or W-entry in the r-th row.

ColumnParents of CR_i are then defined as a union of all parents of $(CR_i)_r$ for $1 \le r \le s$, i.e. $\cup_{r=1}^{s}$ *parents of* $(CR_i)_r$.

Example: In Table VI, $(CR_7)_1$, $(CR_7)_4$ and $(CR_7)_5$ have no parents. CR_3, CR_4, CR_5 and CR_6 are parents of $(CR_7)_2$. CR_1, CR_3, CR_5 are parents of $(CR_7)_3$. Therefore, ColumnParents of CR_7 are the union of all parents of $(CR_7)_r$ for $1 \le r \le 5$, i.e. $\cup_{r=1}^{5}$ *parents of* $(CR_7)_r = \{CR_1, CR_3, CR_4, CR_5, CR_6\}$.

Definition 11: Label(CR_i): CR_i is a column to be labeled. Assume that the columns CR_1 through CR_{i-1} have been labeled. The column CR_i is labeled with as small a positive integer as possible with the following constraints:

(1) no more than $m - w$ columns in *ColumnParents*(CR_i) have the same label and have a R-entry in a row where CR_i has a R-entry in the same row.

(2) no more than $m - r$ columns in *ColumnParents*(CR_i) have the same label and have a W-entry in a row where CR_i has a W-entry in the same row.

(3) no more than m columns in *ColumnParents*(CR_i) have the same label and the number is greater than zero.

Definition 12: Relabel$(s,maxorder,maxlabel)$: same as *Definition 6.*

Definition 13: Backtrack$(s1,s2,flag,maxlabel)$; CR_{s1} is the column whose label is being decreased. If the backtrack procedure cannot make any improvement in the labeling, then it returns *false* in the *flag*. Oth-

TABLE VI
DM(C,R,W) for Table I

	Registers														
	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
S1	R	R	W	0	0	0	0	0	0	0	0	W	0	0	0
S2	0	0	R	R	W	R	W	0	0	0	0	0	W	0	0
S3	R	0	R	0	R	0	R	W	W	R	W	0	0	0	0
S4	0	0	0	0	0	0	0	R	R	0	R	R	0	W	W
S5	W	W	0	0	0	0	0	0	0	0	0	0	0	R	R

erwise, it returns *true* in the *flag,* and that indicates that there exist a column CR_{s2} whose label may be increased so that the maximum label can be decreased. Let *maxlabel* be the maximum label before the backtracking process (i.e. the label of CR_{s1}). Let V be the set of the columns whose order is lower than CR_{s1}, and S be the set of *ColumnAncestors* of CR_{s1}. The detailed procedure of *backtrack* includes two steps:

Step 1: If S is empty, then set *flag* is false and exit. Otherwise, set *flag* is true and find the largest ordered column CR_{s2} which belongs to both S and V. Let $S = S - \{CR_{s2}\}$.

Step 2: The label of CR_{s2} should be increased as little as possible with the following constraints:

(1) no more than $m - w$ columns in *Column-Parents*(CR_i) have the same label and have a R-entry in a row where CR_i has a R-entry in the same row.

(2) no more than $m - r$ columns in *Column-Parents*(CR_i) have the same label and have a W-entry in a row where CR_i has a W-entry in the same row.

(3) no more than m columns in *Column-Parents*(CR_i) have the same label.

Moreover, the increased label should be less than *maxlabel.* Then, call the procedure *Relabel*(*s2,maxorder,maxlabel*). After the relabeling process, if the maximum label is decreased, then *flag* is set to true and exit the *backtrack* procedure. Otherwise, go to step 1.

Lemma 4: The lower bound on the number of multiport memory modules M can be derived from the following equation.

$$\text{LowerBound} = \text{Max}\left\{\text{Max}\left\{\left\lceil\frac{r(i)}{m-w}\right\rceil, \left\lceil\frac{w(i)}{m-r}\right\rceil\right\}\right\} \quad \text{for } i = 1 \text{ to s.}$$

where r(i) = no. of registers accessed by read instruction at control step i
w(i) = no. of registers accessed by write instruction at control step i

The complete algorithm of forming *PDM*(C,m,r,w) from *DM*(C,R,W) can be stated as follows.

Step 1: Order the columns CR_i of *DM*(C,R,W) from 1 to n, where n is the number of registers.

Step 2: Call *Relabel*(1,*maxorder,maxlabel*).

Step 3: If label of the column whose order is *maxorder* is equal to the *LowerBound,* then go to step 4. Otherwise, call *Backtrack*(*maxorder,s2,flag,maxlabel*). If *flag* is false, then go to step 4. Otherwise, call *Relabel*(*s2,maxorder,maxlabel*). If the label of the column whose order is *maxorder* is less than *maxlabel,* then *maxlabel* is set to the lower label. Return to step 3.

Step 4: The value of *maxlabel* is the minimum number of sets p in *PDM*(C,m,r,w). The columns of *DM*(C,R,W) that have the same labels belong to the same partition.

Example 3: Consider the 3-port memory synthesis (2 ports are read only, 1 port is write only) of the registers in the code sequence as shown in Table I. The dependence matrix *DM*(C,R,W) is shown in Table VI. After application of the PDM algorithm to the dependence matrix, three multiport memory modules (*LowerBound* = 3) are required for the data path synthesis, i.e., they are $\{R_1, R_3, R_4, R_8, R_{13}, R_{14}\}$, $\{R_2, R_5, R_6, R_9, R_{10}, R_{12}, R_{15}\}$ and $\{R_7, R_{11}\}$. Note that the multiport memory module M has 3 memory ports where 2 are read only and 1 is write only. Using the previous techniques for memory modules with ports of same type, we still need the same number "3" of memory modules, but, the memory modules have all three ports of same type i.e. read/write.

Using the 0-1 integer programming technique proposed in [11] to the example 3, the problem reduces to

$$\text{Max } (x_1 + x_2 + x_3 + \ldots\ldots\ldots + x_{15})$$

Subject to

$$x_1 + x_2 \le 2$$
$$x_3 + x_{12} \le 1$$
$$x_1 + x_2 + x_3 + x_{12} \le 3$$
$$x_3 + x_4 + x_6 \le 2$$
$$x_5 + x_7 + x_{13} \le 1$$
$$x_3 + x_4 + x_5 + x_6 + x_7 + x_{13} \le 3$$
$$x_1 + x_3 + x_5 + x_7 + x_{10} \le 2$$
$$x_8 + x_9 + x_{11} \le 1$$
$$x_1 + x_3 + x_5 + x_7 + x_8 + x_9 + x_{10} + x_{11} \le 3$$
$$x_8 + x_9 + x_{11} + x_{12} \le 2$$
$$x_{14} + x_{15} \le 1$$
$$x_8 + x_9 + x_{11} + x_{12} + x_{14} + x_{15} \le 3$$
$$x_{14} + x_{15} \le 2$$
$$x_1 + x_2 \le 1$$
$$x_1 + x_2 + x_{14} + x_{15} \le 3$$

The formulation of the algorithm in Ref. [11] attempts to include the largest number of registers into the current memory module, a desirable result. Therefore, the algorithm will produce the first grouping of registers to be $\{R_2, R_6, R_7, R_8, R_{10}, R_{12}, R_{15}\}$ (that is $x_2 = x_6 = x_7 = x_8 = x_{10} = x_{12} = x_{15} = 1$). Then, the same approach is applied to the remaining registers. The final solution of example 3 by using the 0-1 integer programming technique generates four memory modules, specifically $\{R_2, R_6, R_7, R_8, R_{10}, R_{12}, R_{15}\}$, $\{R_1, R_3, R_4, R_5, R_{14}\}$, $\{R_9, R_{13}\}$ and $\{R_{11}\}$.

TABLE VII
Example of the algorithm to form $PDM(C,3,2,0)$ for Table VI

Algorithm	Labels														
	CR1	CR2	CR3	CR4	CR5	CR6	CR7	CR8	CR9	CR10	CR11	CR12	CR13	CR14	CR15
relabeled(1,11,4)	1	2	1	1	1	2	2	2	3	2	4	2	3	1	2
backtrack(11,10,T,4)	1	2	1	1	1	2	2	2	3	3	4	2	3	1	2
relabeled(10,11,4)	1	2	1	1	1	2	2	2	3	3	4	2	3	1	2
backtrack(11,8,T,4)	1	2	1	1	1	2	2	3	3	3	4	2	3	1	2
relabeled(8,11,4)	1	2	1	1	1	2	2	3	2	2	4	2	3	1	2
backtrack(11,10,T,4)	1	2	1	1	1	2	2	3	2	3	4	2	3	1	2
relabeled(10,11,4)	1	2	1	1	1	2	2	3	2	3	4	2	3	1	2
backtrack(11,7,T,4)	1	2	1	1	1	2	3	3	2	3	4	2	3	1	2
relabeled(7,11,4)	1	2	1	1	1	2	3	2	3	2	4	2	3	1	2
backtrack(11,10,T,4)	1	2	1	1	1	2	3	2	3	3	4	2	3	1	2
relabeled(10,11,4)	1	2	1	1	1	2	3	2	3	3	4	2	3	1	2
backtrack(11,8,T,4)	1	2	1	1	1	2	3	3	3	3	4	2	3	1	2
relabeled(8,11,4)	1	2	1	1	1	2	3	3	2	2	4	2	3	1	2
backtrack(11,10,T,4)	1	2	1	1	1	2	3	3	2	3	4	2	3	1	2
relabeled(10,11,4)	1	2	1	1	1	2	3	3	2	3	4	2	3	1	2
backtrack(11,6,T,4)	1	2	1	1	1	3	3	3	2	3	4	2	3	1	2
relabeled(6,11,4)	1	2	1	1	1	3	2	2	3	2	4	2	3	1	2
backtrack(11,10,T,4)	1	2	1	1	1	3	2	2	3	3	4	2	3	1	2
relabeled(10,11,4)	1	2	1	1	1	3	2	2	3	3	4	2	3	1	2
backtrack(11,8,T,4)	1	2	1	1	1	3	2	3	3	3	4	2	3	1	2
relabeled(8,11,4)	1	2	1	1	1	3	2	3	2	2	4	2	3	1	2
backtrack(11,10,T,4)	1	2	1	1	1	3	2	3	2	3	4	2	3	1	2
relabeled(10,11,4)	1	2	1	1	1	3	2	3	2	3	4	2	3	1	2
backtrack(11,7,T,4)	1	2	1	1	1	3	3	3	2	3	4	2	3	1	2
relabeled(7,11,4)	1	2	1	1	1	3	3	2	3	2	4	2	3	1	2
backtrack(11,10,T,4)	1	2	1	1	1	3	3	2	3	3	4	2	3	1	2
relabeled(10,11,4)	1	2	1	1	1	3	3	2	3	3	4	2	3	1	2
backtrack(11,8,T,4)	1	2	1	1	1	3	3	3	3	3	4	2	3	1	2
relabeled(8,11,4)	1	2	1	1	1	3	3	3	2	2	4	2	3	1	2
backtrack(11,10,T,4)	1	2	1	1	1	3	3	3	2	3	4	2	3	1	2
relabeled(10,11,4)	1	2	1	1	1	3	3	3	2	3	4	2	3	1	2
backtrack(11,5,T,4)	1	2	1	1	2	3	3	3	2	3	4	2	3	1	2
relabeled(5,13,4)	1	2	1	1	2	1	3	1	2	2	3	2	4	1	2
backtrack(13,6,T,4)	1	2	1	1	2	2	3	1	2	2	3	2	4	1	2
relabeled(6,11,4)	1	2	1	1	2	2	1	2	3	2	4	2	4	1	2
backtrack(11,10,T,4)	1	2	1	1	2	2	1	2	3	3	4	2	4	1	2
relabeled(10,11,4)	1	2	1	1	2	2	1	2	3	3	4	2	4	1	2
backtrack(11,8,T,4)	1	2	1	1	2	2	1	3	3	3	4	2	4	1	2
relabeled(8,11,4)	1	2	1	1	2	2	1	3	2	2	4	2	4	1	2
backtrack(11,10,T,4)	1	2	1	1	2	2	1	3	2	3	4	2	4	1	2
relabeled(10,11,4)	1	2	1	1	2	2	1	3	2	3	4	2	4	1	2
backtrack(11,7,T,4)	1	2	1	1	2	2	3	3	2	3	4	2	4	1	2
relabeled(7,7,3)	1	2	1	1	2	2	3	1	2	2	3	2	1	1	2

Example 4: Consider the 3-port memory synthesis (2 ports are read only, 1 port is read/write) of the registers in the code sequence as shown in Table I. Three memory modules are derived from the partitioning results. The step-by-step operation of the algorithm is shown in Table VII. Using the final results from the algorithm, the partitioned dependence matrix can be derived in Table VIII.

SIMULATION RESULTS

The fifth order elliptic wave filter was chosen as a benchmark example for which the force-direct schedule [17] is shown in Figure 1. The available hardware consists of two adders and one pipelines multiplier where the multiplication requires an execution time that is twice as long as that for additions. One input

TABLE VIII
PDM(*C*,3,2,0) for Table VI

	Registers														
	R1	R3	R4	R8	R13	R14	R2	R5	R6	R9	R10	R12	R15	R7	R11
S1	R	W	0	0	0	0	R	0	0	0	0	W	0	0	0
S2	0	R	R	0	W	0	0	W	R	0	0	0	0	W	0
S3	R	R	0	W	0	0	0	R	0	W	R	0	0	R	W
S4	0	0	0	R	0	W	0	0	0	R	0	R	W	0	R
S5	W	0	0	0	0	R	W	0	0	0	0	0	R	0	0

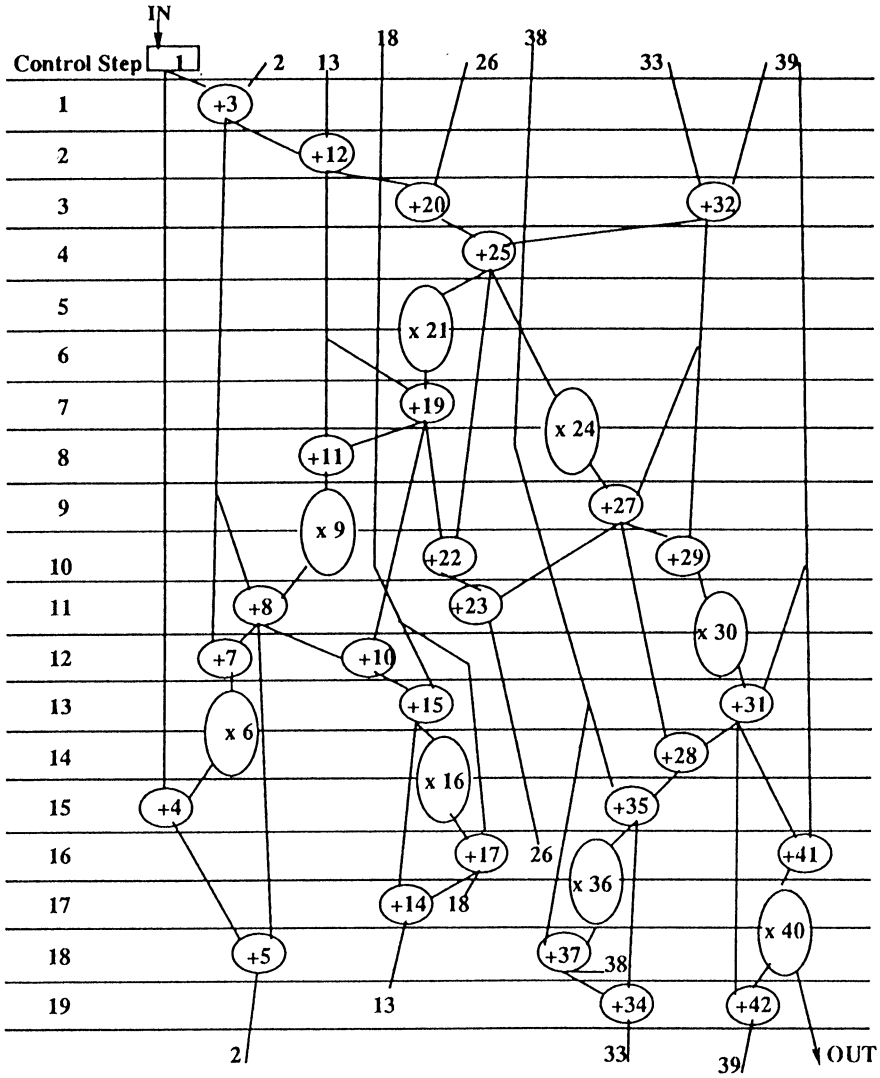

FIGURE 1 Force-directed schedule of wave filter.

to the multiplier is always a constant, and has been omitted in the Figure 1.

Using the PDM with 2-port memory synthesis for the wave filter example, 5 memory modules are partioned and they are: (We use integer representing the corresponding register number.)

*M*1: {1,2,4,5,7,9,10,11,12,13,14,17,20,21,23}
*M*2: {3,6,8,15,24,25,26,32,34,36}
*M*3: {16,18,19,22,28,30,33,37,40}
*M*4: {27,29,31,35,38}
*M*5: {39,41,42}

The register to memory port mapping for different control steps, generated by PDM, is listed in Table IX. The data path design is shown in Figure 2. For completeness, we also give the results produced by six previously proposed methods ("HAL" [17], "SPLICER" [18], "CATREE" [19], "SPAID" [20], "EMUCS" [21], and "Grant and Denyer" [22]) for the wave filter example. Table X compares the cost using this approach "PDM" with that obtained using six previously proposed methods. As can be seen from Table X, the PDM requires fewer MUX inputs, register files, and buses.

Using the PDM with 2-port memory synthesis for the code sequence shown in Table I, 4 memory modules are partioned. The register to memory port mapping is listed in Table XI. The data path design is shown in Figure 3. The ALU1 and ALU3 do no operations but pass data to registers in control step 5.

COMPLEXITY ANALYSIS

The PDM algorithm described in the previous sections has been implemented in *C* code where we use the two-dimensional linked list data structure to store

TABLE IX
Register to memory port mapping of Fig. 2

Control Step	M1		M2		M3		M4		M5	
	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2
1	R1	R2	R3							
2	R12	R13	R3							
3	R20	R12	R26	R32	R33				R39	
4		R20	R32	R25						
5			R25							
6		R21								
7	R12	R21		R25	R19					
8	R12	R11		R24		R19				
9	R11		R24	R32			R27			
10		R9	R32	R25	R19	R22	R29	R27		
11	R9	R23	R8	R3	R22		R29	R27		
12	R7	R10	R3	R8	R19	R30				
13	R7	R10	R15		R18	R30	R31		R39	
14			R15	R6	R28		R27	R31		
15	R1	R4		R6	R28	R16	R35	R38		
16	R23	R17	R26		R16	R18	R35	R31	R41	R39
17	R17	R14	R15	R36	R18				R41	
18	R4	R5	R36	R8	R37	R40	R38	R38		
19	R14	R13		R34	R37	R40	R31	R35	R42	
	R5	R2		R34		R33			R42	R39

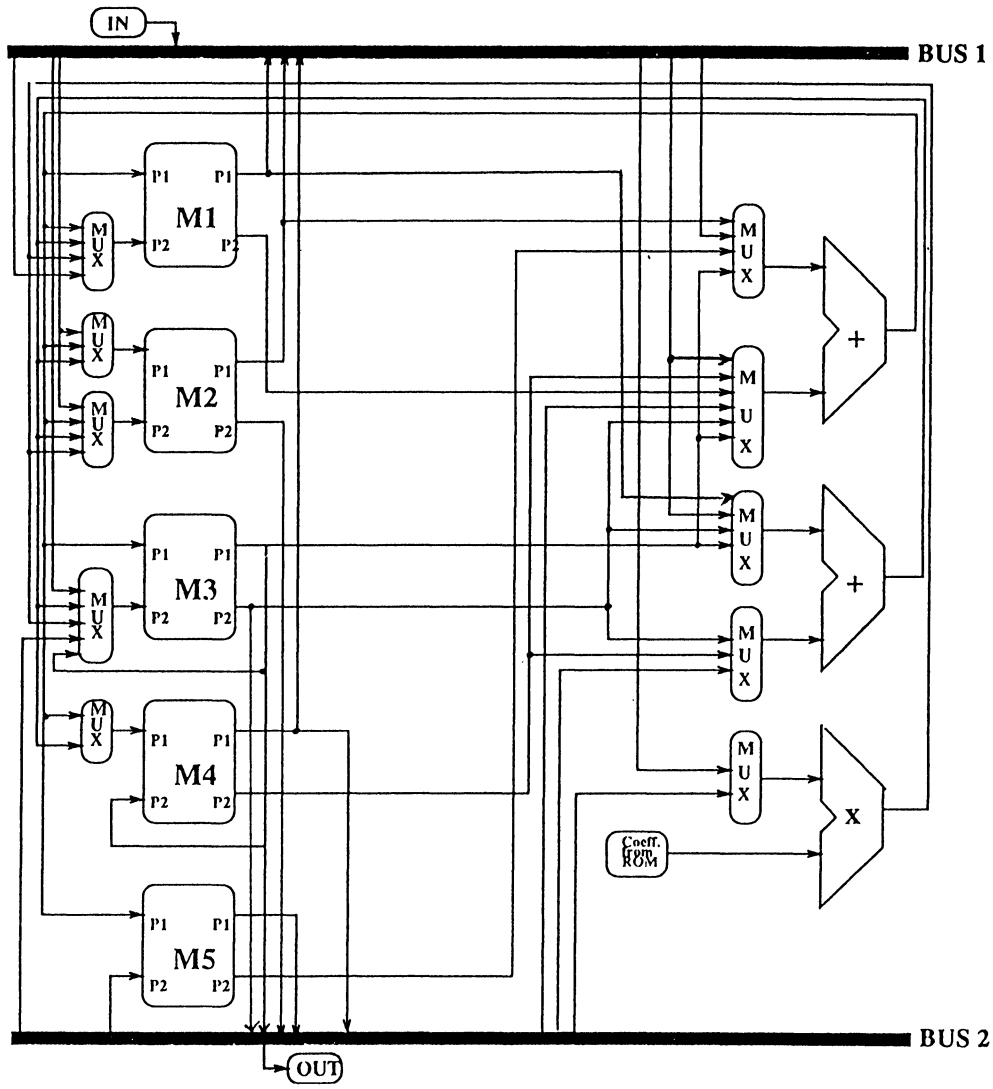

FIGURE 2 Data path for wave filter using 2-port memory.

the values of R or W-entry in the dependence matrix. The major operations in the PDM algorithm include: *columnparent, relabel,* and *backtrack.* The complexity of each operation is summarized as follows: for each column register, the complexity for columnparent is $O(MN)$ and the complexity for relabel and backtrack is $O(MN^2)$. If there are N registers, each register has at most $N - 1$ columnancestors in the worst case. Then, relabel and backtrack operations will be executed at most $N \times (N - 1)$ times in the worst case. The *complexity* of the algorithm can then be calculated as $O(MN^4)$ where N is number of columns (i.e. registers) and M is number of rows (i.e. control steps) in dependence matrix DM(C). We may refine the algorithm to have the *complexity* of $O(MN^3)$ by binding the *Column – Ancestor* to the one with the lowest order. Using the refined algorithm, most of the results (no. of memory modules) obtained are the same as the results of the PDM without refinement, and the run time is significantly reduced.

SUMMARY

This paper presents an efficient algorithm to explore the design space for memory allocation in data path synthesis. The technique can be applied not only to single-port memory synthesis but also to multiport memory synthesis. It avoids certain locally optimal solutions to achieve more globally optimal solutions

TABLE X
Cost summary for wave filter example

System	HAL	SPLICER	CATREE	SPAID	EMUCS	Grant & Denyer	PDM
Cycles	19	21	17	19	19	19	19
Multipliers	1	1	2	1	2	1	1
Adders	2	2	4	2	2	2	2
Mux Inputs	26	35	38	17	15	18	12
Registers	12	N/A	12	19	12	17	5*
Buses	6	N/A	7	5	7	N/A	2

* PDM uses 5 two-port Registers

than were obtained in the previously proposed 0-1 integer programming technique. With suitable modifications, the proposed technique can also be applied to multiport memory synthesis in which the maximum number of read ports is different from the maximum number of write ports. Thus, an alternative data path design which requires less hardware in multiport memory synthesis can be achieved.

The proposed techniques of memory synthesis is applied to the code sequence generated from compiler. As we know, the results of memory synthesis can be affected by manipulating the code sequence. Therefore, investigating the interface between the compiler and memory synthesis to achieve a better design becomes interesting and important in this area. Finally, with suitable modifications, the partitioning techniques should also be applicable to the other phases of data path synthesis, namely the allocation of data operators and interconnection units. These issues are currently under investigation.

TABLE XI
Register to memory port mapping of Fig. 3

Control Steps	M1		M2		M3		M4	
	P1	P2	P1	P2	P1	P2	P1	P2
S1	R1	R2	R3			R12		
S2	R5	R4	R3	R6	R13	R7		
S3	R1	R5	R3	R8	R7	R9	R10	R11
S4			R8	R14	R12	R9	R15	R11
S5	R1	R2	R14				R15	

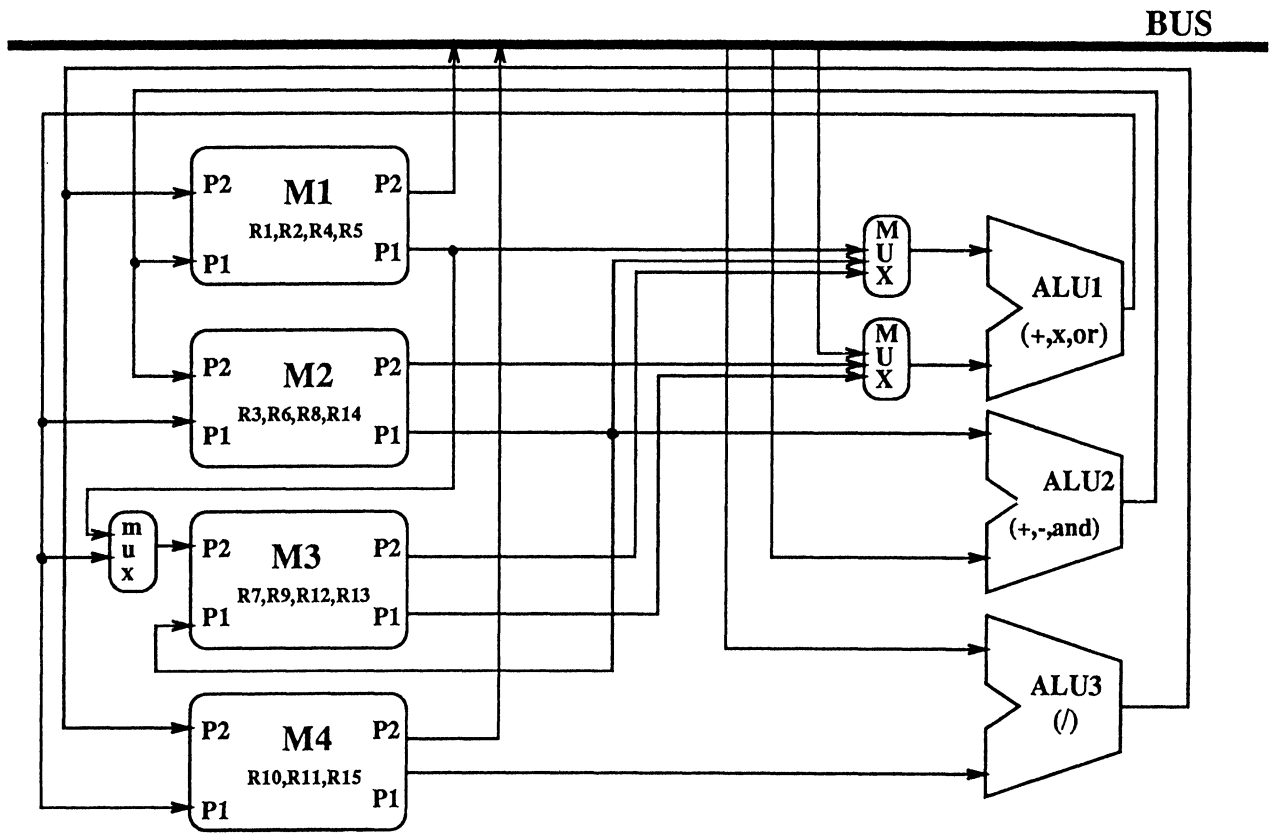

FIGURE 3 Data path for code sequence of Table I using 2-port.

Acknowledgment

The author wishes to thank Prof. Gerald Sobelman for his valuable discussion of initial idea of PDM and the anonymous referees for their constructive comments on the first draft paper. This work was supported in part by the Ohio State Research Challenge Award 660808 and the Graduate School of the Wright State University.

References

[1] C.J. Tseng and D.P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. on Computer-Aided Design*, CAD-5, pp. 379–395, 1986.

[2] H.C. Torng and N.C. Wilhelm, "The Optimal Interconnection of Circuit Modules in Microprocessor and Digital System Design," *IEEE Trans. on Computers*, C-26, pp. 450–457, 1977.

[3] C.J. Tseng and D.P. Siewiorek, "The Modeling and Synthesis of Bus System," *Proc. 18th ACM/IEEE Design Automation Conference*, pp. 471–478, 1981.

[4] C.Y. Hitchcock III and D.E. Thomas, "A Method of Automatic Data Path Synthesis," *Proc. 20th ACM/IEEE Design Automation Conference*, pp. 484–489, 1983.

[5] L.J. Hafer and A.C. Parker, "Automated Synthesis of Digital Hardware," *IEEE Trans. on Computers*, C-31, pp. 93–109, 1982.

[6] L.J. Hafer and A.C. Parker, "A Formal Method for the Specification, Analysis, and Design of Register-Transfer level Digital Logic," *IEEE Trans. on Computer-Aided Design*, CAD-2, pp. 4–18, 1983.

[7] A.C. Parker, F. Kurdahi, and M. Milnar, "A General Methodology for Synthesis and Verification of Register-Transfer Design," *Proc. 21st ACM/IEEE Design Automation Conference*, pp. 329–335, 1984.

[8] S.W. Director, A.C. Parker, D.P. Siewiorek, and Donald E. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems," *IEEE Trans. on Circuits and Systems*, CAS-28, pp. 634–645, 1981.

[9] B.M. Pangrle and D.D. Gajski, "Design Tools for Intelligent Silicon Compilation," *IEEE Trans. on Computer-Aided Design*, CAD-6, pp. 1098–1112, 1987.

[10] C.J. Tseng and D.P. Siewiorek, "Facet: A Procedure for the Automated Synthesis of Digital Systems," *Proc. 20th ACM/IEEE Design Automation Conference*, pp. 490–496, 1983.

[11] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, J.G. Linders and J.C. Majithia, "Allocation of Multiport Memories in Data Path Synthesis," *IEEE Trans. on Computer-Aided Design*, CAD-7, pp. 536–540, 1988.

[12] C.-I.H. Chen and G.E. Sobelman, "Cluster Partitioning Techniques for Data Path Synthesis," to appear in *VLSI Design: An International Journal of Computer-Chip Design, Simulation, and Testing.*

[13] C.-I.H. Chen and G. E. Sobelman, "An Efficient Approach to Pseudo-Exhaustive Test Generation for BIST Design," *IEEE International Conference on Computer Design*, pp. 576–579, 1989.

[14] J.L. Byrne and L.G. Proll, "Solution of Linear Programs in 0-1 Variables by Implicit Enumeration Algorithm 341," *Commun. Ass. Comput.*, pp. 782, 1968.

[15] E.J. McCluskey, "Verification Testing—A Pseudoexhaustive Test Technique," *IEEE Trans. on Computers*, C-33, pp. 541–546, 1984.

[16] F. Hirose and V. Singh, "McDDP, A Program for Partitioning Verification Testing Matrices," Center for Reliable

Computing, Stanford Univ., Stanford, CA. Tech. Rep. 82-7, 1982.
[17] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling for the Behabioral Synthesis of ASICs," Data Path Synthesis," *IEEE Trans. on Computer-Aided Design*, CAD-6, pp. 661–679, 1989.
[18] P.G. Pangrie, "Splicer: A Heuristic Approach to Connectivity Binding," *Proc. of 25th Design Automation Conf.*, pp. 536–541, 1988.
[19] C.H. Gebotys and M.I. Elmasry, "VLSI Design Synthesis with Testability," *Proc. of 25th Design Automation Conf.*, pp. 16–21, 1988.
[20] B.S. Haroun and M.I. Elmasry, "Automatic Synthesis of a Multi-Bus Architecture for DSP," *Proc. of 25th Design Automation Conf.*, pp. 44–47, 1988.
[21] D.E. Thomas etc., "The System architect's Workbench," *Proc. of 25th Design Automation Conf.*, pp. 337–343, 1988.
[22] D.M. Grant and P.B. Denyer, "Memory, Control and Communication Synthesis for Scheduled Algorithms," *Proc. of 27th Design Automation Conf.*, pp. 162–167, 1990.

Biography

CHIEN-IN HENRY CHEN is an Assistant Professor of Electrical Engineering at Wright State University. He received the B.S. degree from the National Taiwan University, Taiwan, in 1981, the M.S. degree from the University of Iowa, Iowa City, in 1986, and the Ph.D. degree from the University of Minnesota, Minneapolis, in 1989, all in the Electrical Engineering. During the summer 1992, he worked as a Faculty Research Associate at Wright Laboratory, Wright-Patterson Air Force Base, Dayton, Ohio. His current research interests are in areas of computer-aided design, simulation and testing of VLSI circuits, design for testability, and fault-tolerant computing.