VLSI Design
1994, Vol. 2, No. 1, pp. 17–32
Reprints available directly from the publisher
Photocopying permitted by license only

Register-Transfer Synthesis of Pipelined Data Paths*

NOHBYUNG PARK† and FADI J. KURDAHI‡
Department of Electrical & Computer Engineering, University of California, Irvine, CA 92717

(Received March 21, 1990, Revised August 11, 1992)

We present a new approach to the problem of register-transfer level design optimization of pipelined data paths. The output of high level synthesis procedures, such as Sehwa, consists of a schedule of operations into time steps, and a fixed set of hardware operators. In order to obtain a register-transfer level design, we must assign operations to specific operators, values to registers, and finish the interconnections. We first perform module assignment with the goal of minimizing the interconnect requirements between RT-level components as a preprocessing procedure to the RT-level design. This will result in a smaller netlist which makes the design more compact and the design process more efficient. In addition to reducing the total number of interconnects, this approach will also reduce the total number of multiplexors in the design by eliminating unnecessary multiplexing at the inputs of shared modules. The interconnect sharing task is modeled as a constrained clique partitioning problem. We developed a fast and efficient polynomial time heuristic procedure to solve this problem. This procedure is 30–50 times faster than other existing heuristics while still producing better results for our purposes. Using this procedure, we can produce near optimal interconnect sharing schemes in a few seconds for most practical size pipelined designs. This efficient approach will enable designers to explore a larger portion of the design space and trade off various design parameters effectively.

Key Words: *Module assignment; Interconnect sharing; High level synthesis; Register-transfer design; Pipelined data paths; Multiplexor allocation; Register allocation*

1. INTRODUCTION

During the past decade, much of the research in design automation has focused on the physical level design of VLSI circuits and printed circuit boards. Although there are many inherently complex and difficult tasks left unsolved, the efforts at this level have produced many useful theories and practical results which have been actually put into use in industry. High-level and register-transfer synthesis work also has started producing useful results. At this stage of development of electronic design automation, we believe that there is increased demand for a better interface between these somewhat independently studied levels of design automation. Especially from the high-level synthesis point of view, high-level design tradeoffs and optimizations must be made based on their effects on the final cost and performance of the physical implementations in order to guarantee optimality of the final design. An interesting synthesis experiment which demonstrated this aspect of high-level synthesis was reported by McFarland [1].

Synthesis of digital systems from behavioral specifications is one of the most challenging problems in design automation nowadays. The main goal is to automatically generate a register-transfer level design starting from a behavioral specification, and a set of constraints on overall design parameters such as cost (area) and speed. Behavioral synthesis is a computationally intractable problem when optimal solutions are desired. In fact, most subproblems related to synthesis are known to be NP-complete or NP-hard. In order to reduce the complexity of the global problem to manageable levels, the synthesis process is divided into subtasks as depicted in Figure 1. The specific subtasks addressed in this work are shaded.

The work reported in this paper aims at completing the behavioral pipelined data path synthesis package

*This work was supported by NSF Grants #CCR-8708894 and #MIP-8909677.

†Dr. Park is currently with Samsung Electronics, SEOUL, KOREA.

‡Please send all correspondence to the corresponding author.

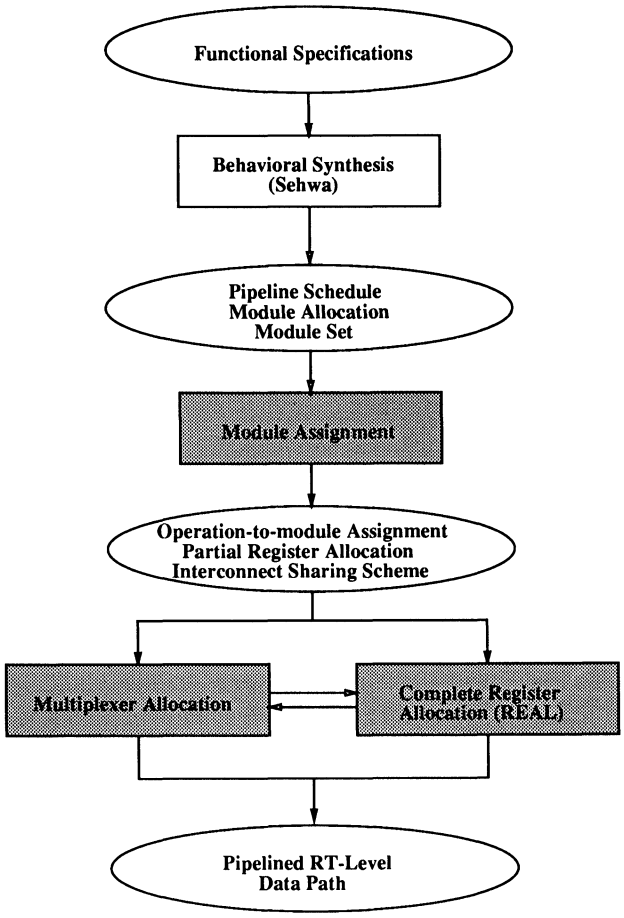


FIGURE 1 A behavioral synthesis system.

Sehwa [2] and to integrate it with other synthesis tools developed by the authors, CSSP [3] and REAL [4]. In this paper we focus on the register-transfer synthesis problem, specifically on the module assignment and interconnect sharing subtasks of RT-level synthesis of pipelined data paths. Once a good module assignment and interconnect sharing schemes are obtained, the pipelined RT-level design can then be completed by using a multiplexor and bus allocation program which can accept partial pre-existing bindings, such as MABAL [5].

While the bus allocation, module assignment, and interconnect minimization problems have been discussed in the literature before [6] [7], little work has been published on the module assignment problem for pipelined data path designs. In [6], module assignment is either forced by schedule or carried out in an ad hoc fashion. Furthermore, the techniques described work only on non-pipelined data path designs. BUD [7] attempts to cluster the design prior to scheduling, thus module assignment and interconnect sharing are forced as a result of this clustering. In our approach, scheduling and resource allocation are not forced by any *a priori* clustering in order to achieve maximum resource utilization for pipelining. Finally, BUD also applies only to non-pipelined designs. Other important works addressing various aspects of synthesis of pipelined data paths are reported in [8], [5], [9], [10], and [11].

2. SYNTHESIS OF PIPELINED DATA PATHS

The task of pipeline synthesis is to produce a register-transfer implementation of a pipelined digital computer from the data flow (function) description of the tasks to be executed. Constraints set by the designer are: the total available cost budget and the minimum required performance. The design tasks include:

1. *scheduling:* determine execution order and timing of operations,
2. *resource allocation:* determine how many of which types of resources to allocate before or after scheduling, and
3. *register-transfer level synthesis:* detailed assignment of operations to operators, and placement and interconnection of hardware modules.

When an optimal design (with respect to design constraints) is desired, these tasks cannot be performed separately. For example, the shortest possible schedule does not always guarantee the fastest performance since it might force the resource allocation in such a way that the fastest possible initiation rate of the tasks is not feasible due to resource conflicts between consecutive tasks. Also, an expensive pipeline does not always guarantee better performance since scheduling may not be able to utilize all the available resources. Finally, the feasibility of resource allocation within cost constraint can be checked only after the register-transfer level synthesis is completed.

Currently, the pipeline synthesis software package Sehwa [2] synthesizes pipelined data paths at the behavioral level. The inputs to Sehwa are a data flow graph with either a cost or a performance constraint and a module library. Sehwa produces a pipeline schedule in time steps, a list of modules (the number of each selected type of modules), and a resource allocation scheme to time steps in the schedule.

Figure 2 shows an example input dataflow graph to the pipeline synthesis program Sehwa [12]. The graph shows operations on values and their ordering. In this graph, there are two types of operations,

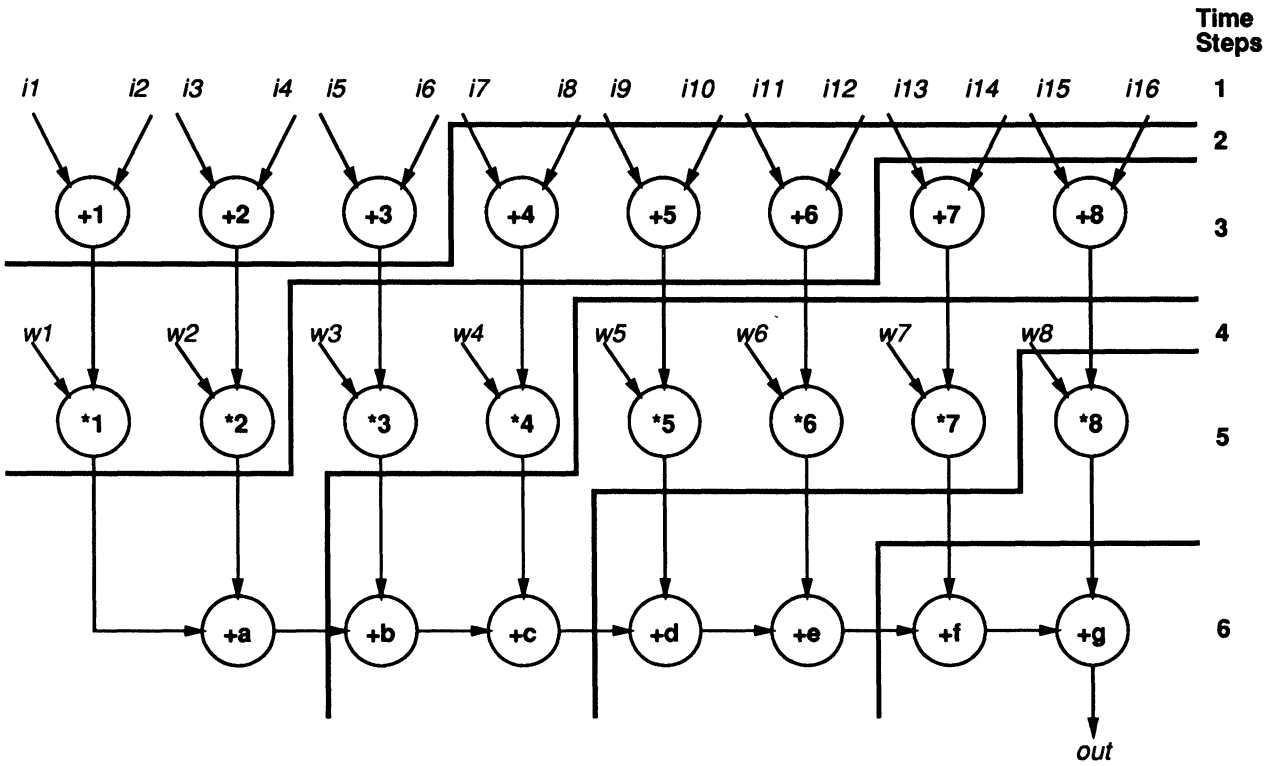


FIGURE 2 An optimally scheduled data flow graph of a 16-point FIR filter with three multipliers and five adders for the latency of three

add(+) and *multiply*(*). The figure also shows the output from the pipeline synthesis program Sehwa. The partitioning of the data flow graph into time steps is shown with solid lines. Scheduling determines how many of each type of operators to allocate to each time step. The constraint is not to allocate to each group of overlapping time steps more than the total number of available modules. For this example, the optimization goal is to minimize the length of the schedule within the resource constraint.

The allocation of operators to time steps is depicted in the *allocation table* (Table I) which shows time steps divided into three groups. The time steps in each group overlap in time due to pipelined execution. Three multipliers and five adders are chosen and shared over all six time steps. The number of rows in the allocation table corresponds to the number of available modules. The allocation table is an extension of a conventional reservation table [13] [14].

The results of scheduling and resource allocation together with the chosen module set are the inputs to the module assignment task, which performs specific operation-to-operator mapping. In a pipelined data path design, if the pipeline input initiation latency, L, is one, all the operators perform useful

TABLE I
The Resource Allocation Table for the Schedule of Figure 2

Operator	Time steps					
	1	4	2	5	3	6
adder	+1		+4		+7	
	+2		+5		+8	
	+3		+6		+a	
		+b		+d		+f
		+c		+e		+g
multiplier		*5	*1		*3	
		*6	*2		*4	
		*7		*8		

operations at all times, except when there are hazards or conditional branches. In such cases, no operators or registers can be reused during the execution of each cycle, which makes register-transfer synthesis straightforward and simple. However, if the latency L is greater than one, then each operator can be reused at most L times during the execution of each cycle. In such a case, sharing interconnects is very important since the excess amount of wiring space due to multiplexing at the inputs of the operator being shared might weigh against the advantage of operator sharing for lower cost designs.

3. REGISTER-TRANSFER SYNTHESIS OF PIPELINED DATA PATHS

The high-level pipeline synthesis procedure, Sehwa, produces a schedule of operations into time steps, the types and numbers of modules needed, and a resource allocation scheme to time steps. This scheduling is optimal with respect to the given and chosen constraints (e.g. a cost constraint given by the user and an optimal latency chosen by Sehwa within the cost constraint). In order to complete the register-transfer pipeline data path, we need to perform the following tasks:

- specific operation-to-module assignment within each time step,
- multiplexor assignment for shared inputs to modules,
- storage (stage latches) assignment for pipelined values, and
- interconnection of all these components.

These design tasks interact with each other. For example, the numbers of multiplexors and registers necessary depend on the specific operation-to-module assignment. The amount of interconnects depends partly on how the above components are to be connected. When an optimal design is desired, we need to enumerate all possible RT-level implementations of the given pipeline data path and compare the final layouts. However, exhaustive enumeration is computationally intractable. Even the operation-to-module assignment alone can have exponential time complexity ($O(n^m)$ in the worst case where n is the number of operations and m is the number of modules).

In order to produce a near optimal design in a reasonable amount of time, we need to prioritize the importance of the effect of the design decisions on the final implementation and perform the design tasks in some sequential fashion, although not completely sequential. In this paper, we propose such an approach.

4. MODULE ASSIGNMENT FOR PIPELINED DATA PATHS

One of the most important optimization tasks in VLSI design is to minimize the total chip area, which also tends to minimize circuit delays. Previous synthesis work has attempted to minimize the active area in layouts by minimizing the total number of active components in the design, such as operators, registers and multiplexors. However, as VLSI designs get more and more complex, wiring area becomes the dominating factor in layout area. The current approach to area optimization addresses wiring area minimization as a side effect of optimizing the active area and not as a first order effect. The main premise of the work presented in this paper is that *the wiring effect must be treated as a first order effect during the register-transfer level synthesis in order to optimize the total chip layout area.* In order to minimize the total wiring area, we need to

1. minimize the number of interconnection nets to layout and
2. perform optimal placement and routing.

The second problem has been extensively studied before and will not be addressed in this paper. Rather, we focus on the first problem, minimizing the number of nets to layout. By attempting to minimize the number of interconnections, we are automatically simplifying the placement and routing tasks.

During RT-level synthesis of pipelined data paths, the number of interconnections depends partially on the allocation scheme of operations to modules and values to registers. For a scheduled data flow graph, if the initiation latency $L > 1$, each module can be used L times as long as it is not assigned more than once in any overlapping time steps during pipelining. Naturally, a wire which is connected to an output of such a shared module can also be shared together with the module. If the destination operations share the same module, we can share the interconnect without any multiplexors or output control circuitry, as shown in Figure 3. In other words, interconnect sharing also minimizes multiplexors. This fact distinguishes the proposed interconnect sharing scheme from the conventional bus sharing which requires

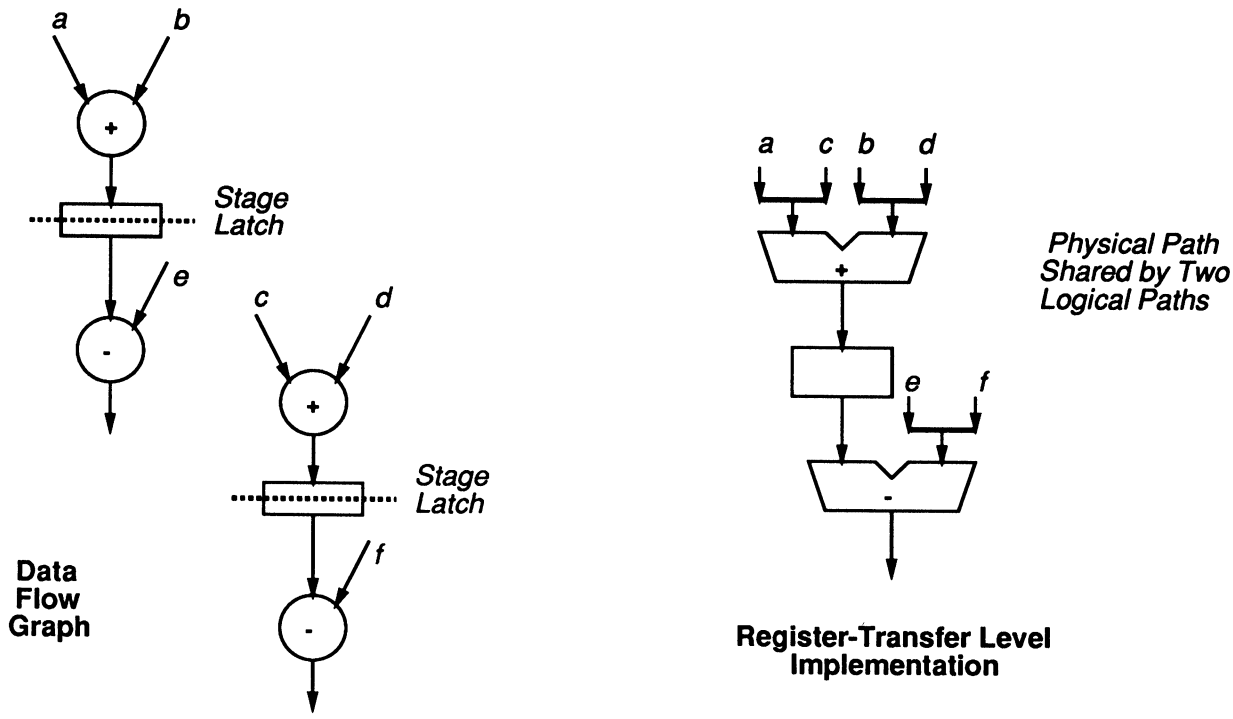


FIGURE 3 Sharing of physical paths

sharing which requires either input multiplexors or output control circuitry for the modules sharing a common bus. If desired, bus sharing can still be implemented even after the interconnect sharing scheme is fixed. Likewise, we can share registers needed to store the value on the shared interconnects. In other words, *optimal interconnect sharing tends, in general, to optimize the multiplexors and, in some cases, registers at the same time.*

The interconnect sharing pattern is determined by the specific operation-to-module assignment. For this reason, our approach to register-transfer level implementation of pipelined data path must start with a good module assignment.

Following behavioral synthesis, the number of modules needed for each type of operation is known. In an ad hoc approach, the operations are then assigned to modules individually, the only criterion being that the module be capable of performing the particular operation and that it is not being used to perform other concurrent operations. This ad hoc approach may work fine when the hardware cost function does not include interconnect costs. Assigning an operation to any module capable and free to perform it does not increase or decrease the design cost function. However, when interconnect cost is factored into the cost function, then the assignment of operations to modules *does* indeed affect the overall system cost.

Figure 4 (a) shows a portion of a scheduled dataflow graph. In (b), a "greedy" module assignment resulted in operations 1 and 3 sharing adder A, 2 and 4 sharing adder B, and 5 and 6 sharing subtractor C. Now if we attempt to share *patterns* instead of individual operators, then patterns (1,5) and (2,6) are similar and can hence share an adder connected to a subtractor, patterns (3) and (4) are also similar and can therefore share an adder. This will result in the assignment shown in (c) which is less complex and will most probably result in a more compact layout. In addition to having a simpler interconnect pattern, the assignment in (c) has one less multiplexor than the assignment in (b).

4.1 Problem Definition and Procedure Outline

The inputs (received from Sehwa) to the module assignment task are:

- the scheduled data flow graph which shows operator-to-time step assignment, interconnects between operations, and stage latch requirements, and
- the corresponding allocation table showing the number and types of modules assigned to each time step.

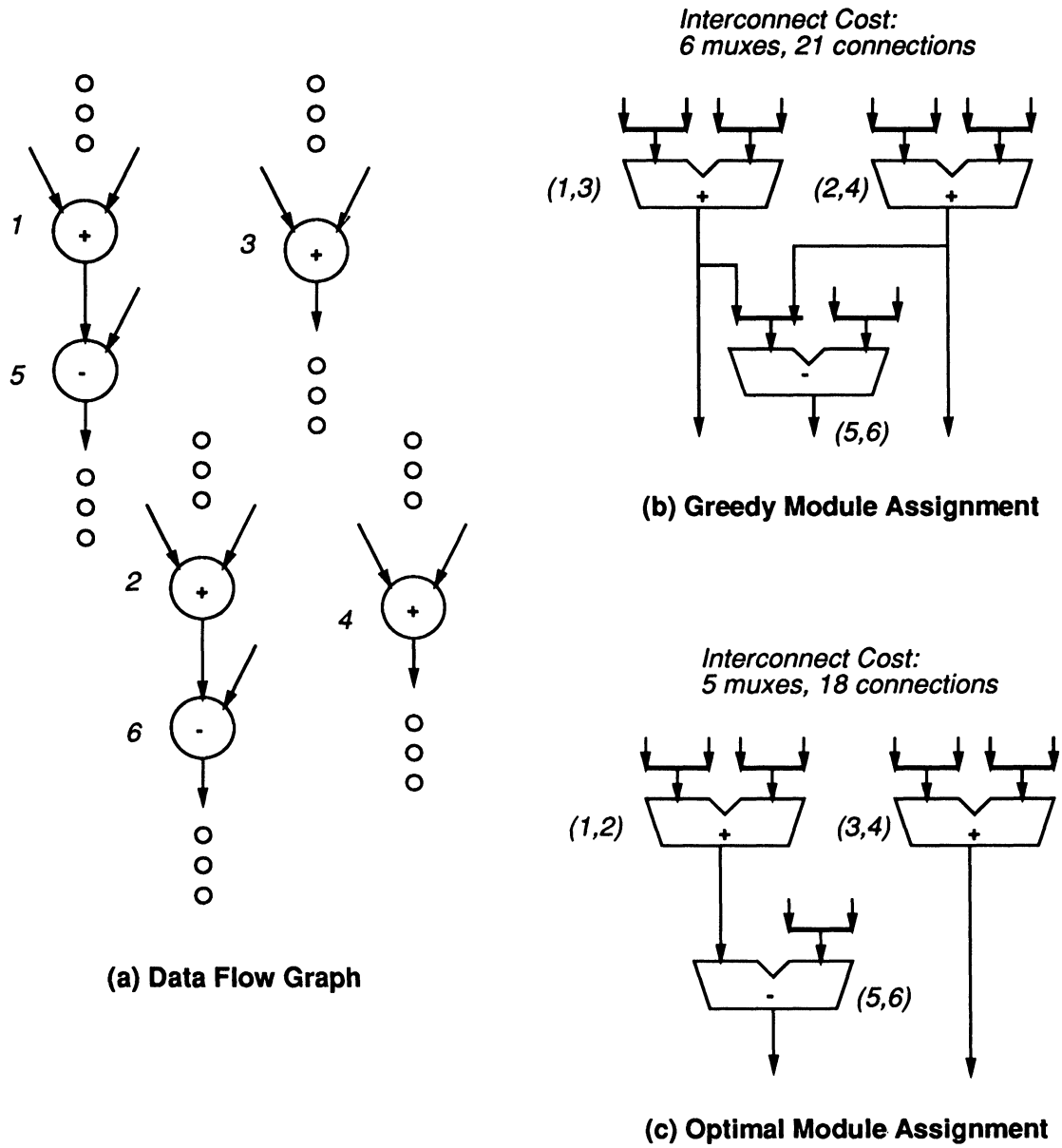


FIGURE 4 Module assignment example

The outputs are:

- specific operation-to-module assignments,
- interconnect sharing schemes, and
- stage latch placement on the shared interconnects.

The optimization goal here is to maximize interconnect sharing in a way that will also maximize stage latch and register sharing and minimize multiplexors.

We now define the conditions of module assignment for interconnect sharing, and then discuss the general procedure of module assignment for maximum interconnect sharing. We first define several terms for clarity of discussion.

Definition 1: A **path vector** is defined as an ordered list of the operations and stage latches along a directed path in the scheduled data flow graph for pipelining. A path vector always starts from an operation and terminates at an operation. The length of a path vector is defined as the number of operations (nodes) in that path.

Definition 2: A pair of path vectors, P_i and P_j are called **compatible** if and only if all the following conditions are true.

1. P_i and P_j have identical resource-usage patterns, i.e., contain identical numbers of operations and stage latches in the same order.
2. Corresponding elements in P_i and P_j can be assigned to the same module (the same operator or stage latch).
3. Every element in P_i does not overlap in time with the corresponding element in P_j during pipelining, i.e., they can share the same module in different non-overlapping time steps.

The pairs of matching operations or stage latches in two compatible path vectors can be assigned to the same module (operator or register). In other words, the two compatible path vectors use the same set of modules and interconnects in the same order in nonoverlapping time steps, as we have seen in the example of Figure 3. Examples of compatible path vectors in the scheduled data flow graph of Figure 2 are <(+1 r *1 r+a) (+4 r *4 r+c)>, <(+6 r r *6) (+8 r r *8)>, and <(+b +c) (+d +e) (+f +g)>. In these expressions, 'r' denotes a stage latch.

For a set of compatible path vectors, interconnect sharing is possible when every pair of path vectors in the set are compatible with each other. For example, for the compatible set <(+b +c) (+d +e) (+f +g)>, the additions +b, +d and +f can be assigned to an adder, say adder 1, and +c, +e and +g can be asigned to another, say adder 2. This way, the three compatible path vectors can share the same physical interconnect from adder 1 to adder 2 without any input multiplexors.

4.2 Synthesis Procedure

We first outline the procedure for register-transfer level implementation of pipelined data paths. Then we will present the detailed descriptions of the major steps of the procedure.

PROCEDURE MOD-ASSIGN;

Step 1: Enumerate all the compatible path vectors of *certain lengths* in the scheduled dataflow graph.
{*Approach:* Use depth-first graph traversal algorithm starting from each operation in the graph except those in the last time step of the schedule.}

Step 2: Determine a good interconnect sharing pattern by grouping path vectors that can share the same interconnect.
{*Approach:* Find cliques of size L (pipe initiation latency) or smaller using a polynomial-time clique partitioning heuristic.)}
/* A module can be reused at most L times in a pipeline with initiation latency L */

Step 3: Resolve conflicts between groups (cliques) obtained in Step 2 (since path vectors are not necessarily disjoint, implementing interconnect sharing for a group may make some part of the sharing in other groups not feasible).

Step 4: Perform operator and register assignment according to the resulting operation/value-to-operator/register assignment and interconnect sharing schemes.

Step 5: Complete register-transfer design by adding interconnects, multiplexors, and registers for the rest of the pipelined data path.

In Step 1 of the procedure, path vectors of chosen lengths are generated. Ideally, in order to get the optimal interconnect sharing scheme, we need to enumerate all the possible paths of all possible lengths. Then we need to test interconnect sharing for all possible combinations of the paths. Unfortunately, the path enumeration problem alone has exponential time complexity. Finding an optimal sharing scheme for one such combination is also of exponential time complexity. For this reason, to be realistic, we only consider path vectors of length two (nodes) in our discussion. The path vectors of longer lengths can still be covered as chains of these shorter path vectors. The procedure PATH_ENUM, described in Appendix A.1, uses a recursive, depth-first graph traversal algorithm, DF_Path, to list all paths of a certain length (or number of nodes). The time complexity of this procedure is $O(n^c)$ where n is the number of nodes and c is the length of the path vectors.

From the path vector list, we can build graphs which show compatibility relationships among the path vectors. The conditions for compatibility were given by Definition 2. The completed compatibility graphs for the path vectors of Table II are shown in Figure 5. The edge connecting nodes 1 and 2 shows that the paths 1 and 2 can share the same modules (operators and registers) and thus the interconnects. The COM_Path algorithm, described in Appendix A.2, builds compatibility graphs from the list of path vectors.

As mentioned before, in pipelining with latency L, each module can be reused L times. Naturally, the next task will be finding cliques of size L from each connected subgraph. Unfortunately, the clique partitioning problem is NP-Complete. An efficient clique partitioning heuristic procedure is needed.

TABLE II
The path list

Path Number	Path	Time Slot	Path Number	Path	Time Slot
1	(+b +c)	(4 4)	2	(+d +e)	(5 5)
3	(+f +g)	(6 6)	4	(+1 r *1)	(1 2)
5	(+2 r *2)	(1 2)	6	(+3 r r *3)	(1 3)
7	(*1 r +a)	(2 3)	8	(*2 r +a)	(2 3)
9	(+4 r *4)	(2 3)	10	(+5 r r *5)	(2 4)
11	(+6 r r *6)	(2 4)	12	(+a r +b)	(3 4)
13	(*3 r +b)	(3 4)	14	(*4 r +c)	(3 4)
15	(+7 r *7)	(3 4)	16	(+8 r r *8)	(3 5)
17	(+c r +d)	(4 5)	18	(*5 r +d)	(4 5)
19	(*6 r +e)	(4 5)	20	(*7 r r +f)	(4 5)
21	(+e r +f)	(5 6)	22	(*8 r +g)	(5 6)

Note that we need to get only the cliques of size L or smaller. Also, maximum interconnect sharing is achieved by *minimizing the total number of cliques and not by maintaining the size of each clique.* To take advantage of these facts, we developed a new clique partitioning procedure instead of using existing general procedures such as Tseng's [6]. The time complexity of this procedure is $O(L \times E \times N^2)$ in the worst case, where L is the pipe latency, E is the number of edges (pairs of compatible paths), and N is the number of nodes (paths) in the compatibility graph. However, as we find cliques, these cliques are removed from the graph and therefore the number of remaining paths is reduced rapidly and the procedure converges fast. The new heuristic procedure, L_CLIQ, described in Appendix A.3, finds near optimal solutions for most graphs when the clique size is smaller than 10. The procedure is programmed in *C* on a SUN3 workstation and runs in a few seconds for graphs with 100 nodes and 1000 edges.

Next, we need to carry out specific module assignment according to the results of module assignment and interconnect sharing schemes. Steps 3 and 4 of the overall procedure (MOD-ASSIGN) are carried out in parallel. The procedure, CLIQ_ASSIGN, described in Appendix A.4, uses a resource allocation table to keep track of current status of module assignment and availability of modules, as shown in Table III. For the precise usage of the resource allocation table, refer to [2].

At this stage of the synthesis, there might be operations not assigned to any hardware modules yet. There are two approaches to the completion of RT-level designs. First, we can repeat the overall procedure, MOD-ASSIGN, with shorter path vectors which contain the unassigned operations. A Second approach is to complete the module assignment using a greedy algorithm (since the possibility of finding more interconnect sharing schemes at this stage will be very low).

Once module assignment is complete and an interconnect sharing scheme is obtained, we need to complete the RT-level design by adding more registers and multiplexors. In our approach, we perform

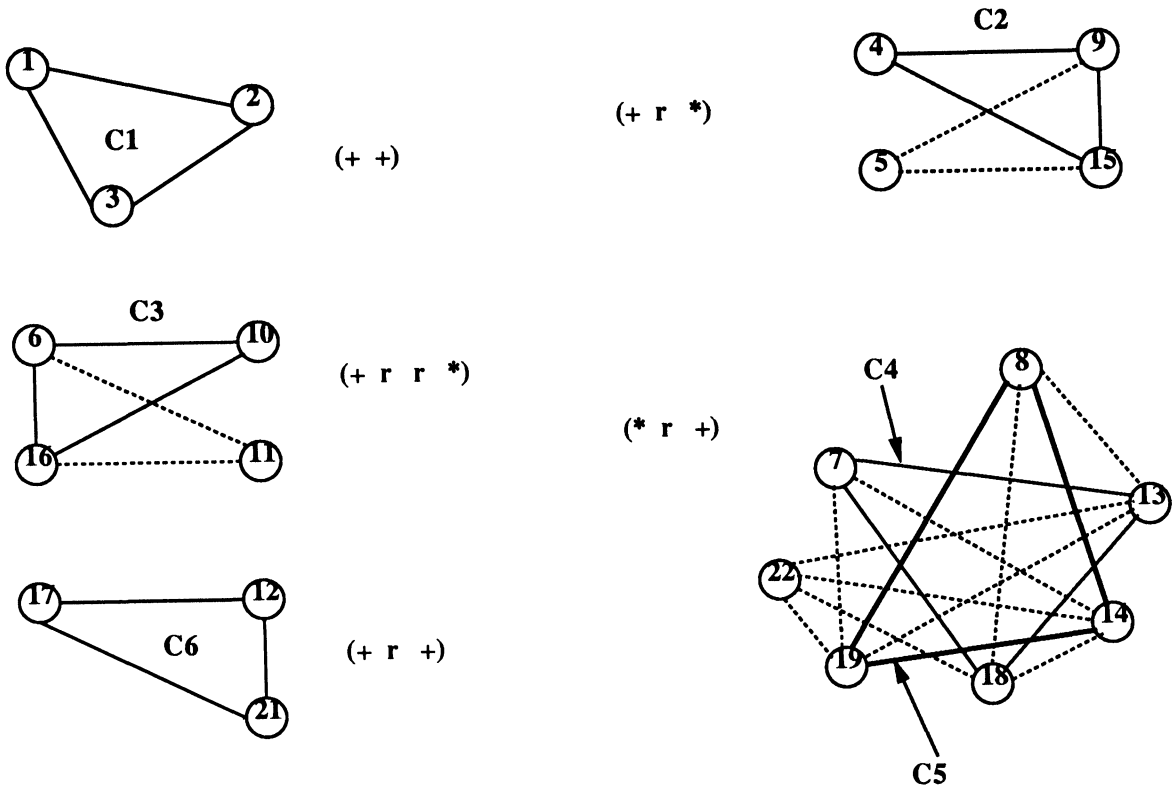


FIGURE 5 Compatible path graphs

TABLE III
Module assignments for greedy and otpimized designs

Hardware Module	Operations	
	Optimized	Greedy
A1	+1, +4, +7	+1, +4, +7
A2	+3, +5, +8	+2, +5, +8
A3	+b, +d, +f	+3, +6, +a
A4	+c, +e, +g	+b, +d, +f
A5	+2, +6, +a	+c, +e, +g
M1	*1, *4, *7	*5, *1, *3
M2	*3, *5, *8	*6, *2, *4
M3	*2, *6	*7, *8

optimal register assignment first using the register allocation procedure REAL [4]. Then the multiplexor placement is forced by the module assignment and the register assignment.

4.3 Example

Since there has been no reported work on module assignment and interconnect sharing for pipelined designs, we cannot provide any comparative data on the performance of our approach. However, in this section, we apply our technique to the 16-point FIR filter in Figure 2. From this example we generate two different RT-level designs, one using a greedy module assignment, the other using our technique. Results of the design comparison are presented at the end of this section.

The data flow of a 16-point FIR filter of Figure 2 receives inputs from an external 16-bit shift register, which is not shown here but might actually be built into the same chip with the filter. The data flow graph is scheduled with three multipliers and five adders, which is the minimum set of modules in order to achieve an initiation latency L of three (clock cycles). The allocation table (Table I) shows resource allo-

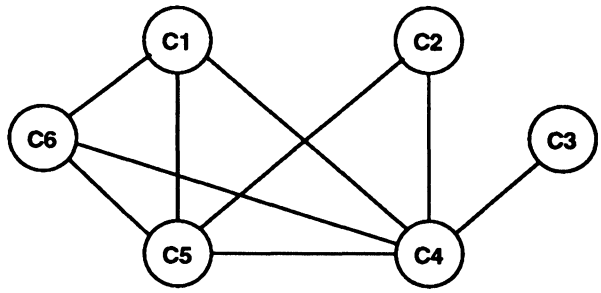


FIGURE 6 Clique interaction graph

cation to each group of overlapping time steps during pipelining. As mentioned before, once we get the scheduled data flow graph with resources allocated to each group of overlapping time steps, we need to perform the specific module assignment to each operation. The objective function is to minimize the number of interconnection nets, which might also minimize the number of multiplexors and stage latches necessary to complete the register-transfer pipeline design.

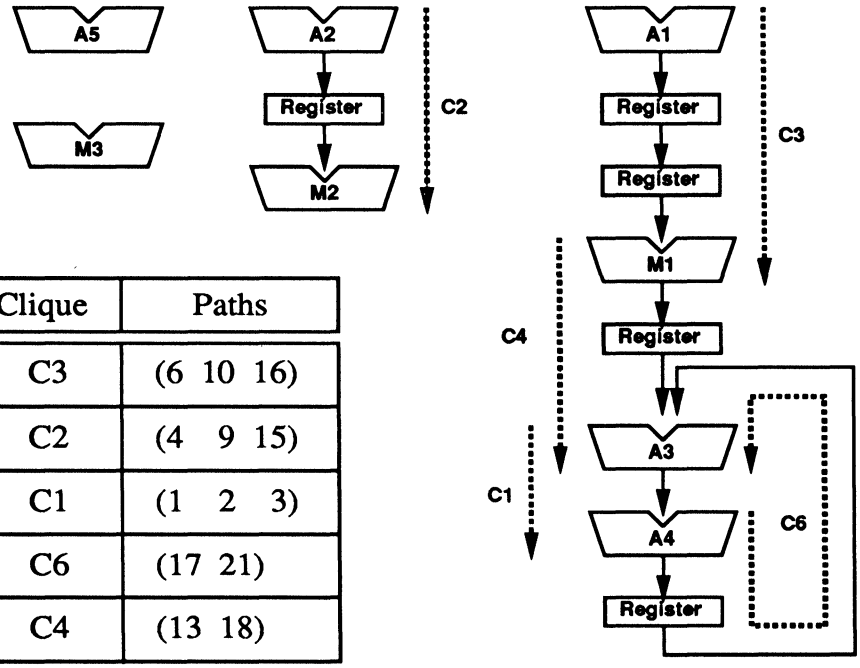
First, we construct a list of path vectors. In this example, we use path vectors of length two only for simplicity of illustration as shown in Table II.

The second step is to group the path vectors into compatible path vectors. The result of this grouping is shown in Figure 5. Next, we need to partition each of the compatible path graphs into cliques of size L (pipe initiation latency) or smaller. The solid edges in the graphs of Figure 5 show the result of the clique partitioning.

Once we get the cliques of compatible paths, we need to check whether there are any conflicts in implementing all the interconnect sharing schemes. For example, clique C1 contains a path from +b to +c and C4 contains a path from *3 to +b. Sharing an interconnect for C1 might make sharing a single interconnect for all the paths in C4 not feasible. For this reason, we construct a graph which shows possible interaction between the cliques as shown in Figure 6.

In our current implementation, we start module assignment with the cliques with minimum number of interaction edges with other cliques. Intuitively, if we start implementing the clique with largest number of interaction edges, we are likely to affect and destroy the interconnect sharing pattern of many other cliques. For this example, the first clique to implement will be either C2 or C3. Actually, we have chosen C2 first and C3 next, using the lowest-index-first tie-breaking rule, followed by C1, C5, and then C4.

Figure 7 shows the final result of the interconnect sharing and the corresponding module assignment. Cliques C2, C3, and C1 are implemented without any conflict with each other. However, interconnect sharing for all three path vectors in C5 (12, 17, 21) is not feasible. The reason is as follows. As shown in Figure 7, interconnect sharing for C1 requires two



Clique	Paths
C3	(6 10 16)
C2	(4 9 15)
C1	(1 2 3)
C6	(17 21)
C4	(13 18)

FIGURE 7 Interconnect sharing results and partial module assignment

adders, Adder 3 and Adder 4. These adders needed to be used together in three consecutive time steps (4, 5 and 6) and thus cannot be reused in any other time steps since the pipeline initiation latency is three. Interconnect sharing for C5 needs +a, +c and +e to be assigned to the same adder. Adder 4 has already been assigned to +c, +e and +g and therefore Adder 4 cannot be assigned to +a. Therefore, path vector 17 (+a r +b) cannot share the same module nor the same interconnect with the

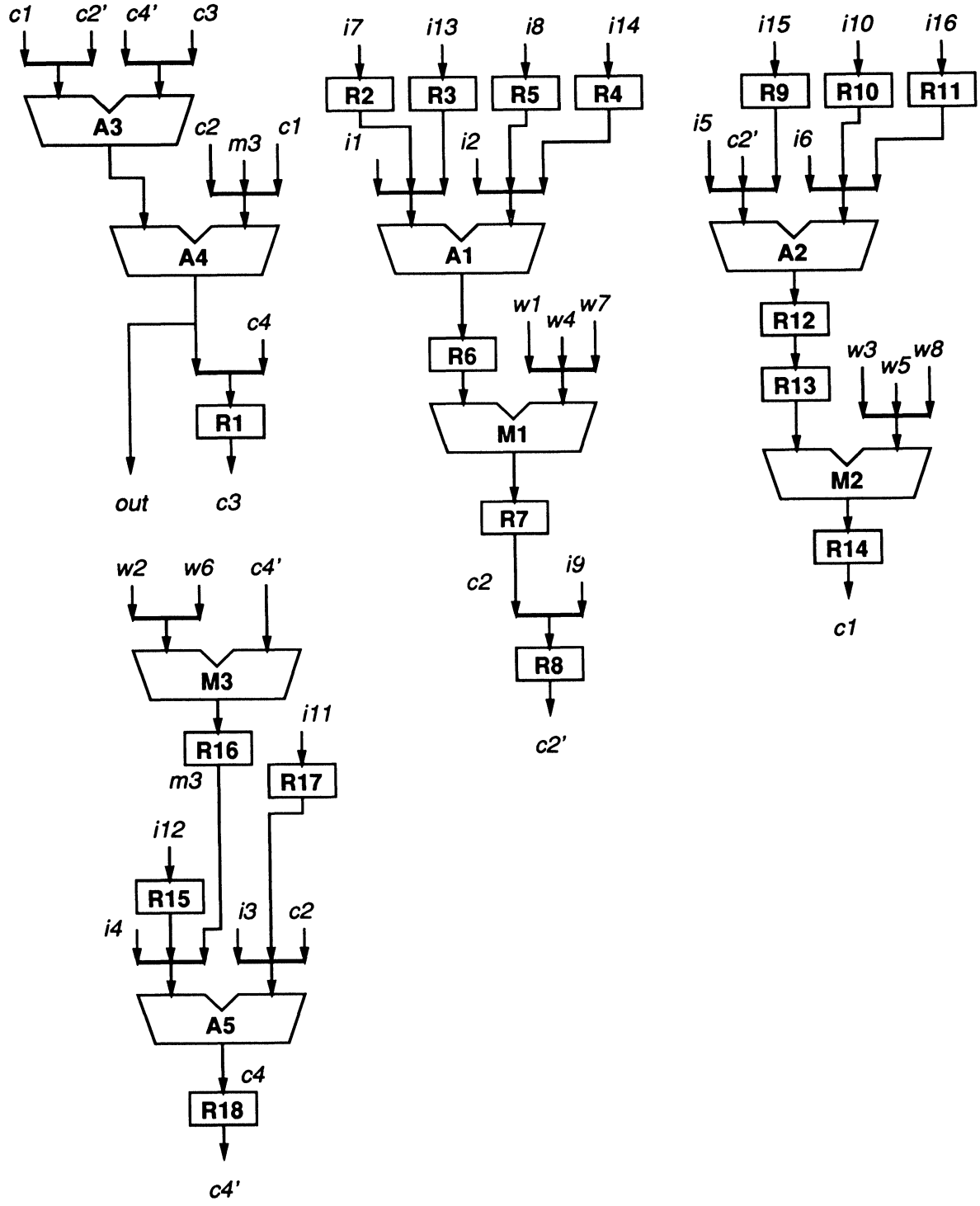


FIGURE 8 Complete RT-level design with optimized module assignment

other path vectors in C5, 17 and 21. By the same reason, for C4, only path vectors 13 and 18 can share the same operators and interconnect. This phase produces a partial module assignment with the goal of minimizing the number of interconnection net, as shown in Figure 7. The rest of the register-transfer synthesis will be the completion of the module assignment, and placement of multiplexors and registers. The final design was completed by hand and register sharing was optimized using a procedure described in [4]. Figure 8 shows the completed RT-level design.

We also generated another RT-level design using a greedy approach to module assignment, as shown in Figure 9. Table III shows the module assignments for both approaches. Table IV shows comparative

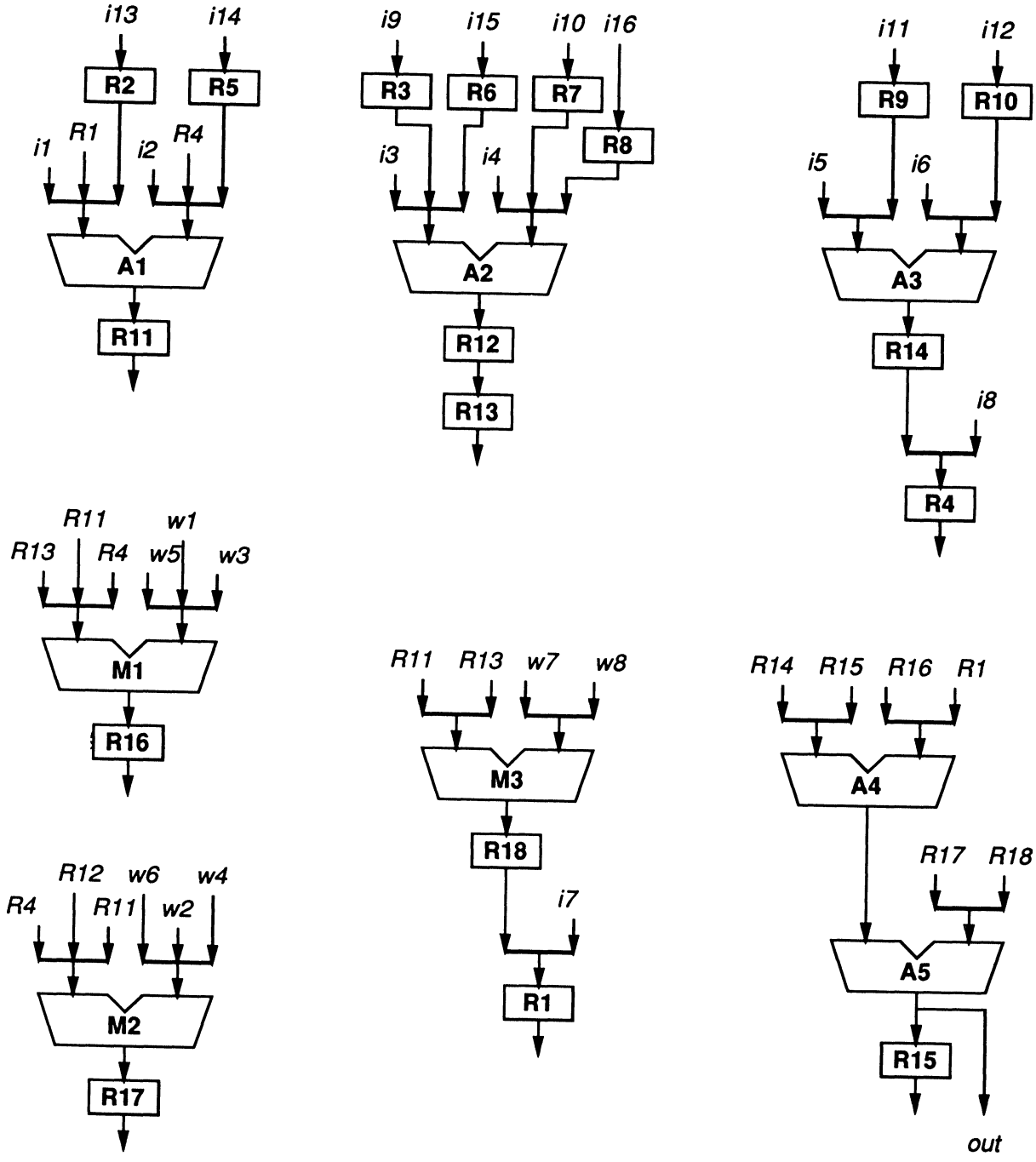


FIGURE 9 An RT-level design with greedy module assignment

TABLE IV
Comparison of greedy versus optimized designs

Design	Adders	Multipliers	Registers	Multiplexors	Interconnect
Greedy	5	3	18	27	42
Optimized	5	3	18	23	34

statistics on various design components. While both designs have the same number of adders, multipliers, and registers, the optimized design represents a significant improvement over the non optimized one, using four less multiplexors and eight, 8-bit interconnects (or 64 wires). This represents about 20 percent savings in the size of the steering logic.

5. CONCLUSION

In this paper, we presented an approach to solving the problem of optimal synthesis of register-transfer pipelined data paths. Synthesis tasks are partitioned and prioritized according to their importance in area optimization of the final register-transfer pipelined data path designs. In the proposed approach, the highest priority is given to the module assignment task. We presented an efficient solution to the module assignment problem as part of register-transfer synthesis of pipelines. The main objective is to maximize sharing of interconnections among the modules. A model for sharing using compatibility graphs was proposed and an efficient heuristic was implemented in Franz Lisp running on a SUN3 workstation. This procedure produces near optimal interconnection sharing schemes in seconds for most practical size examples. As a result of this assignment, the total number of interconnections is expected to be minimal, and the multiplexing to be simpler. This produces area efficient layouts with improved performance. Once module assignment is done, the subsequent tasks of register and multiplexer allocations, need to be performed before we can obtain a complete RT-level design implementation. The register allocation problem was addressed in [4]. The remaining task of multiplexer allocation is currently being researched, and efficient solutions must be proposed before a complete synthesis system can be realized which can produce complete RT-level designs from behavioral specifications. Our system produced near optimal register-transfer pipelined data paths for most examples which ran quickly as mentioned before. We believe that this type of synthesis work is necessary for the high-level synthesis tools to be fully automated and to be practical.

APPENDIX

A. MODULE ASSIGNMENT PROCEDURES

A.1 Path Enumeration Procedure

```
PATH_ENUM(Graph, max_ path_ len) {
/* compute all paths with max_ path_ len (> 2) nodes; */
var path_ list[t] : set of path vectors starting with nodes
                    in time step t of the schedule;
  max_ t: the number of time steps in the schedule;
  /* initialize output path vector slits */
  FOR t := 1 to max_ t - 1 DO path_ list[t] := { };
  FOR t = 1 to max_ t - 1 DO
    FOR every node u in time step t of Graph DO {
      current_ path := (u);
```

```
        FOR every child v of u DO {
          /* depth-first graph traversal */
          DF_ Path(current_ path, 1, t, v);
      }
    }
  }

  PROC DF_ Path(current_ path, path_ len, Tcurrent, v) {
  /* recursive depth-first graph traversal; */
    Tprev := Tcurrent;
    Tcurrent := time_ step_ of(v);
    current_ path := append (Tcurrent - Tprev) r's to current_ path;
    current_ path := append v to current_ path;
    path_ len := path_ len + 1;
    IF path_ len = max_ path_ len
      THEN path_ list[t] := path_ list[t] + { current_ path};
      ELSE {
        For every child w of v Do {
          DF_ Path(current_ path, path_ len, Tcurrent, w);
        }
      }
    return;
  }
```

A.2 Compatibility Graph Generation Procedure

```
PROC COM_ Path(path_ list) {
/* compare path vectors and setup compatibility graph; */
var compatible[i,j] : boolean;
  L : pipeline initiation latency;
  compatible[p,q] := 0 for all i and j;
  FOR i = 1 to (max_ t - 1) DO
    FOR every path vector P in path_ list [i] DO
      FOR j=i+1 to max_ t - 1, j < > i+cL for any integer c, DO
        FOR every path vector Q in path_ list[j] DO
  IF P and Q are compatible
          THEN compatible[p,q] = 1;
}
```

A.3 L-Clique Partitioning Procedure

```
  PROC L_ CLIQ (c_ graph, L) {
  /*
  partition the compatibility graph into minimum number of cliques;
  c_ graph : compatibility graph;
  L : pipeline initiation interval;
  */
    node_ list := set of all vertices in the compatibility graph;
    clique_ list := { };
    WHILE not empty(node_ list) DO {
      u := a vertex with least degree;
      node_ list := node_ list - { u};
      clique := { u};
      size := 1;
```

```
    WHILE (size < L) DO {
      next_ nodes := neighbors of u with least degree in node_ list;
      IF empty(next_ nodes)
        THEN break;
        ELSE IF there is one vertex, v, in next_ nodes
               {append v to clique;
                delete v from node_ list;
                break;}
        ELSE {v := any vertex in next_ nodes that has
                   maximum number of common neighbors with u;
              append v to clique;
              node_ list := node_ list { v};
              size := size + 1}
    }
    clique_ list := clique_ list + clique;
  }
}
```

A.4 Clique Assignment Procedure

```
PROC CLIQ_ ASSIGN(cliques) {
/*
Each clique is a node in the graph.
*/
  /* build clique interaction graph */
    FOR every operation o DO
      connect all the nodes that share the operation o ;
  /* implement module assignment while resolving conflicts */
    WHILE there are nodes in the interaction graph DO {
      sort the nodes in non-decreasing order of degree;
      u := the node with the smallest degree;
      delete u from the interaction graph;
      size := the number of elements in path vectors in u;
      FOR i = 1 to size DO {
        CASE (i-th elements are)
          operations:
            check resource allocation table if any i-th
              operations of the path vectors in u are already
              assigned to modules (by previous cliques);
              IF any i-th elements are already assigned
                THEN pick a module, m, which covers most
                  number of the i-th elements;
    assign to module m the i-th
                  operations not assigned yet;
                ELSE pick a new module and assign all
                  the i-th operations to it;
          registers:
                assign a register module;
      }
    }
}
```

References

[1] M. McFarland, "Reevaluating the design space for register-transfer hardware synthesis," in *Proceedings of ICCAD-87, pp. 262–265, IEEE Computer Society, November 1987.*

[2] N. Park and A.C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Transactions on CAD,* Vol. 7, pp. 356–370, March 1988.

[3] N. Park and A. Parker, "Theory of clocking for maximum execution overlap of high-speed digital systems," *IEEE Transactions on Computers,* Vol. 37, pp. 678–690, June 1988.

[4] F.J. Kurdahi and A.C. Parker, "REAL: A program for register allocation," in *Proceedings of the 24th Design Automation Conference,* pp. 80–85, IEEE/ACM, July 1987.

[5] A.C. Parker et. al., "Experience with the ADAM design system," in *proc. 26th Design Automation Conf.,* pp. 56–61, 1989.

[6] C. Tseng, *Automated Synthesis of Data Paths in Digital Systems.* PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April 1984.

[7] M. McFarland, "Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral specifications," in *Proc. 23rd Design Automation Conf.,* pp. 474–480, IEEE/ACM, July 1986.

[8] K.S. Hwang et al., "Scheduling and hardware sharing in pipelined data paths," in *Proc. ICCAD-89,* pp. 24–27, 1989.

[9] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. CAD,* Vol. CAD-7, Mar. 1989.

[10] R. Potasman, D. Gajski, and A. Nocolau, "Percolation based synthesis," in *Proc. 27th DAC,* 1990.

[11] S. Note et al., "Cathedral-III: Architecture-driven high-level synthesis for high throughput dps applications," in *Proc. 28th DAC,* 1991.

[12] N. Park, *Synthesis of High-Speed Digital Systems.* PhD thesis, Dept. of Electrical Engineering, University of Southern California, September 1985.

[13] E. Davidson, "The design and control of pipelined function generators," in *Proceedings of 1971 International IEEE Conference on Systems, Networks, and Computers,* pp. 19–21, 1971.

[14] P.M. Kogge, *The Architecture of Pipelined Computers.* New York, N.Y.: McGraw-Hill, 1981.

Biographies

NOHBYUNG PARK received the B.S. degree in Electronics Engineering from Seoul National University, Seoul, Korea, in 1976, the M.S. degree in Electrical Engineering from the Korea Advanced Institute of Science and Technology, Seoul, Korea, in 1978, and the Ph.D degree in Electrical Engineering from the University of Southern California, Los Angeles, CA, in 1985. From 1976 to 1981 he was a design engineer and project manager in the Division of Industrial Equipment at Samsung Electronics Company, Seoul, Korea, where he designed computer interfaces, high-speed peripherals, and small business computer systems. From 1986 to 1990 he was an Assistant Professor in the Department of Electrical & Computer Engineering at the University of California, Irvine. He is currently director of research and development at Samsung Electronics Company in Seoul, Korea. His current research interests are high-speed VLSI and digital system design, pipeline synthesis, and design automation.

FADI J. KURDAHI received the Bachelor of Engineering degree in Electrical Engineering from the American University of Beirut, Beirut, Lebanon in 1981. He received the M.S. degree in Electrical Engineering and the Ph.D. degree in Computer Engineering from the University of Southern California, Los Angeles, CA, in 1982 and 1987, respectively. Since 1987, he has been an Assistant Professor in the Department of Electrical & Computer Engineering at the University of California, Irvine. His areas of interest are high-level synthesis of digital circuits, VLSI systems design and layout, and design automation. Dr. Kurdahi received an NSF Research Initiation Award in 1989, and two ACM/SIGDA fellowships in 1991 and 1992. He can be reached by email at *kurdahi@uci.edu* on the Internet, or *KURDAHI@UCI on the Bitnet. Fadi Kurdahi is a member of IEEE and ACM.*