Automated Synthesis of Data Paths in Digital Systems

CHIA-JENG TSENG, MEMBER, IEEE, AND DANIEL P. SIEWIOREK, FELLOW, IEEE

Abstract—**This paper presents a unifying procedure, called *Facet*, for the automated synthesis of data paths at the register-transfer level. The procedure minimizes the number of storage elements, data operators, and interconnection units. A design generator named *Emerald*, based on *Facet*, was developed and implemented to facilitate extensive experiments with the methodology. The input to the design generator is a behavioral description which is viewed as a code sequence. *Emerald* provides mechanisms for interactively manipulating the code sequence. Different forms of the code sequence are mapped into data paths of different cost and speed. Data paths for the behavioral descriptions of the AM2910, the AM2901, and the IBM System/370 were produced and analyzed. Designs for the AM2910 and the AM2901 are compared with commercial designs. Overall, the total number of gates required for *Emerald's* designs is about 15 percent more than the commercial designs. The design space spanned by the behavioral specification of the AM2901 is extensively explored.**

I. Introduction

THE research presented in this paper is a portion of the Carnegie-Mellon University Design Automation (CMU-DA) system [8]. Using the ISPS description [5] as input, the CMU-DA system proceeds through global optimization, design style selection, data-memory allocation, physical module binding, control allocation, chip partitioning, and mask generation phases. This paper describes the result of some research in the data-memory allocation phase.

The data paths of a digital system generally contain three types of primitive elements: storage elements, data operators, and interconnection units. Given a behavioral description, the problem of data-memory allocation includes five subproblems. They are the specification of data flow and control flow, the allocation of storage elements, the allocation of data operators, the allocation of interconnection units, and the exploration of the design space.

Prior research in data-memory allocation is outlined in Section II. The ISPS description specifies the data flow and control flow. The input to the data-memory allocator is an intermediate form, named the Value Trace (VT) [13], [20] of the ISPS description. *Facet* views the VT as a code sequence. Section III contains a code sequence which is used as a running example to illustrate the synthesis procedure. Given a list of operation sequences (in some sense, this means the performance is specified), the problem of design improvement is concerned with the minimization of the number of storage elements, data operators, and interconnection units. *Facet* transforms these three minimization problems into the clique-partitioning problem. The notion of clique-partitioning is introduced in Section IV. Sections V–VII describe the formulations and present algorithms for generating solutions for each of these three problems. Sections VIII and IX discuss the implementation of an automated data path synthesizer based on the algorithms. Evaluation of designs is discussed in Section X and the evaluation criteria are used to compare automatically synthesized designs to commercial designs in Section XI. Section XII contains conclusions and suggestions for future work. Finally, details of the clique-partitioning algorithms are included in the Appendix.

Manuscript received July 18, 1984. This work was done at the Carnegie-Mellon University, Pittsburgh, PA, and was supported by the National Science Foundation under Grant ECS-8207709.

C-J. Tseng is with AT&T Bell Laboratories, Murray Hill, NJ 07974.

D. P. Siewiorek is with the Department of Electrical and Computer Engineering and the Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 8608705.

II. Related Work

Over the last two decades, significant effort has been devoted to the development of methodologies for the synthesis of data paths in digital systems. This section contains a selective survey of the related research which can be found in the literature.

The EXPL system [4] uses a behavioral description as the design specification and a module set for implementation of the design. The behavior is described in ISP which is a high level hardware description language. The module set used is the Digital Equipment Corporation PDP-16 Register Transfer Modules (RTM). The behavioral description of the system to be designed is compiled into an internal representation (a graph model). Based on the graph model, a number of serial-to-parallel and parallel-to-serial transformations are developed for exploring the design space. Heuristic techniques are used to limit the search process. Examples (an RTM multiplier and a controller for a conveyor-bin system) are given to illustrate the design procedure. Applying these transformations to the initial designs of these examples, improvements up to a factor of two in cost and speed are observed.

Rege [17] addresses the problem of designing a small set of high level (higher than the register transfer level and lower than the computer level) Data modules (D-modules) useful for constructing digital systems. He divides the design activity into a number of well-defined subactivities. The activities are defined by means of three spaces: the specification space, the solution space, and the space of primitive components. He first proposes a basic model for the specification space. Then he develops a set of design and analysis techniques to survey the space

0278-0070/86/0700-0379$01.00 © 1986 IEEE

of possible data part implementations for a given computational task. From this a number of parametrizable D-components suitable for designing flexible high level D-modules are developed. Finally, the analysis techniques and the parametrizable D-components are used to build high level D-modules. The use of these high level D-modules to build different systems to a given user specification is also demonstrated.

A distributed style data-memory allocator is described in [9]. The tasks of allocating the data-memory of distributed systems are identified. A small number of improvement techniques are discussed. However, the possible variations of an implementation for a behavioral specification are not considered.

Based on behavioral specifications of digital systems, Hafer [10] develops a mathematical model for the data part to describe the conditions and relationships which must be satisfied. The model is then extended to facilitate automated synthesis of the data part. The use of the mixed-integer linear programming technique results in optimal solutions in terms of the given constraints. The major limitation is that, even for very small specifications, the runtime of generating a design explodes rapidly with complexity.

Two data-memory allocators are described in [21]. One, named DAA, is a rule-based data-memory allocator. DAA investigates the feasibility of applying techniques in artificial intelligence to the problem of synthesizing digital hardware. The other one, named EMUCS, adopts an algorithmic approach. Given a VT description, EMUCS maps abstract data flow elements onto hardware elements in a step by step fashion. In each step tables of costs which reflect the feasibility of binding each abstract element onto each hardware element are generated. The allocator then searches these tables to find the binding that would potentially be the least costly. The algorithm tends to perform local optimization. The lack of ability to find a globally optimal solution is the major limitation of this algorithm.

A data path generator which consists of a library of procedurally defined cells, advanced graphic tools, and a composition program is described in [19]. The input to the system is a description of what registers and operators are desired in the data path. The output is the mask-level geometric layout for an actual circuit. The graphics editor is used to design primitive cells. The composition program places and interconnects the cells into a bit-sliced array. The design tool, which is tailored to a limited domain, is intended for a highly specialized form of microprocessor, one which directly embeds an algorithm (such as the Lisp interpreter of Scheme-81) on the chip. A great deal of knowledge about the particular features of the domain is embedded in the system.

A system which is concerned with the design of central processor data paths is described in [12]. Among other things, the system contains an algorithm library and a hardware library. With the system architecture, performance requirements, and maximum cost limit specified, the system generates a set of data paths and the microprogram to control the data paths as output. By searching the algorithm library and hardware library, a solution is generated by an enumeration process. Heuristics are used to limit search space.

An interactive computer aided design system developed at the IBM Thomas J. Watson Research Center is described in [7]. The system first produces a naive implementation automatically from a functional specification. It then interacts with the designer, allowing evaluation with respect to some factors, such as timing constraints, fan-in and fan-out requirements, etc. The designer can improve the design incrementally by applying local transformations until the design is acceptable for manufacturing. To find a set of transformations and an application sequence to a wide range of initial behavioral descriptions is also a part of the research goal.

The MIMOLA design system [11], [27] is intended to be an interactive design aid. The input to the system is the behavioral specification in MIMOLA, a high level procedural hardware descriptive language. The design process is divided into several steps including syntax analysis, compilation, and allocation. A behavioral specification is first transformed for maximal parallelism, and an implementation is generated to support the sequence of operations. Then, the design process is iterated to reduce the cost of the implementation by restricting the hardware available to the system. A statistical analyzer is used to provide information such as the utilization measure and the expected frequency of use of each hardware module to aid the designer in deciding if (and how) the data part must be constrained in order to meet cost objectives. Estimates of operation delays and microprogram size are given to evaluate the resultant design.

In summary, each of the above approaches has at least one of the following limitations.

1) Local optimization rather than global optimization is considered.
2) The requirements in run time and memory space suffer from combinatorial explosion.
3) The methodology is either ad hoc or tailored to some special architecture.

This paper presents a formal procedure for the synthesis of data paths. The procedure unifies the tasks of data-memory allocation. Both global optimality and execution efficiency (in time and space) are considered in the development of the methodology. Extensive experiments have verified that it produces high quality designs.

III. A Code Sequence

As indicated in Section 1, the input to the data-memory allocator is a VT description. A VT specifies all the data flow and control flow information in the original ISPS description.[1] *Facet* views the VT as a list of operations,

[1]The ISPS description contains *variables* from which *values* are derived. The value trace is an ordered sequence of *value* usage. The data-memory allocator assigns several values to an individual *storage element* such that the computation is still correctly carried out. The mapping between ISPS variables and storage elements is not one-to-one or onto. Indeed, storage elements may contain several ISPS variables or an ISPS variable may appear in several storage elements.

called a code sequence. A basic block is a linear sequence of operation codes having one entry point (the first operation executed) and one exit point (the last operation executed) [1]. A code sequence generally contains a number of basic blocks linked by conditional or concurrent control constructs. Table I is a code sequence which consists of a single basic block. Code statements appearing on the same line are executed in parallel by a single control step. This code sequence will be used as a running example to illustrate the synthesis procedure.

A code sequence can appear in many different forms. Different code sequences are mapped into data paths of different cost and speed. The design space can be explored by manipulating the code sequence.

The next section will introduce the notion of clique-partitioning which unifies the tasks of data-memory allocation.

IV. Clique Partitioning

The resources for data paths are storage elements, data operators, and interconnection units. Given a code sequence, the synthesis of data paths involves three tasks: assigning the variables to a suitable number of storage elements, data operators, and interconnection units. Two variables can be assigned to the same physical resource if and only if there is no conflict in the use of the two resources. Let N be an integer which is greater than two. Then, N variables can be assigned to the same physical resource if and only if each pair of these N variables does not have usage conflict. Let each variable be represented as a node in an undirected graph. The relationship of sharability can be represented by the connectivity of the two nodes. Mapping of storage elements, data operators, and interconnection units can all be formulated into the clique-partitioning problem whose mathematical definition is presented in the next paragraph.

Let G be a graph consisting of a finite number of nodes and a set of undirected edges connecting pairs of nodes. A nonempty collection C of nodes of G forms a complete graph if each node in C is connected to every other node of C. A complete graph C is said to be a **clique** [18] with respect to G if C is not contained in any other complete graph contained in G. The **clique-partitioning problem** is to partition the nodes in G into a number of disjoint clusters such that each node appears in one and only one cluster. Furthermore, each of these clusters itself forms a complete graph (clique).

Many applications require the partitioning of a graph into the minimum number of disjoint cliques. Minimization is consistent with finding the cliques in the graph one by one. However, the search for cliques in a graph has been proved to be *NP-complete*. Related research can be found in [3], [6], [14], [15], [26]. A procedure which partitions a graph into a near minimum number of cliques is given in the appendix (Algorithm 1). The procedure uses the neighborhood property (as described in the appendix) among nodes to partition a graph into a set of disjoint cliques. The time complexity of the procedure is a polynomial function of the numbers of nodes and edges in the graph [24], [25]. The procedure has been applied to a significant number of graphs and found to generate optimal partitionings. However, the neighborhood property is not a sufficient condition for finding the clique in a graph and sometimes a suboptimal solution is possible.

The goal of data-path synthesis is to minimize the number of storage elements, data operators, and interconnection units. Therefore, the problems of data-path synthesis can be formulated into the clique-partitioning problem. However, direct application of the basic clique-partitioning procedure to the data-memory allocation problem does not generate good solutions. Two other notions are necessary. One is the notion of divide-and-conquer and the other is the notion of transitive property. The interpretation of these two notions on data-memory allocation will be detailed in Sections V–VII. The mathematical interpretation of these two notions is given in the following paragraphs.

Given a graph G, each edge of G represents some kind of relationship between the two nodes. The profit of grouping some set of nodes may override the profit of grouping other sets of nodes. Assume that the edges of the graph can be classified into several categories according to the profit measure of grouping each pair of connected nodes. Then a subgraph can be constructed from those edges which belong to the same category. The generalized clique-partitioning algorithm uses these subgraphs to direct the task of clique-partitioning and avoid grouping pairs of nodes randomly. This corresponds to the notion of divide-and-conquer. The notion of transitive property evolves from the clique-partitioning procedure itself. The algorithms we developed combine two nodes at a time. Each time a pair of nodes is merged into a cluster, the profit measure of some edges may need to be updated. This is defined as the transitive property. The transitive property can be classified into two types: the loose form transitive property and the generalized transitive property. Details are given in the Appendix.

The generalized clique-partitioning algorithms proceeds in the following way. The subgraph in which pairs of nodes have the best profit measure is reduced first. Then the pairs of nodes having the next level of profit measure are collected and reduced. Repeatedly applying the procedure to the other subgraphs, the process is stopped when a subgraph of a specified category or the original graph G becomes empty. Again, details of the generalized clique-partitioning algorithms are included in the Appendix (Algorithm 3).

Having introduced the notion of clique-partitioning, the following sections will describe the *Facet* formalization for the synthesis of data paths. Examples will be used to illustrate the formulations and solutions for the synthesis of data paths.

V. Allocation of Storage Elements

As indicated in [24], it is generally beneficial to assign more than one variable to the same physical location. This section discusses the issues of minimizing the number of storage elements.

5.1. Sufficient Conditions for Combining Two Variables

A code sequence generally contains many variables. Given a set of variables, the problem of storage allocation is to combine those variables which can share a storage element. What are the sufficient conditions for combining two variables? A variable is *live* between the time of its definition and last use. A variable is *dead* between the time of its last use and the next definition. If the live periods of two variables are not overlapped, they have disjoint lifetimes. Obviously, two variables can be combined if they have disjoint lifetimes. In reality this constraint can be relaxed. Two variables A and B can be combined if their lifetimes are overlapped in such a way that one of them is used as a source and the other is used as the destination or vice versa in the same statement. In addition, the variable which is used as the source is *dead* in the next time interval, i.e., the use is a "last use." Pure data transfers are special cases.

5.2. A Procedure for Compacting Variables

If there are n variables and each pair of variables are proved to be combinable (there are $n(n-1)/2$ different pairs), then these n variables can be assigned to the same physical location. Let the nodes of a graph be the variables and each pair of nodes which can be combined be joined by an edge. Then a graph which contains the lifetime relationships among all the variables can be constructed. Since the goal is to assign these variables to the minimum number of physical locations, this is translated into the clique-partitioning problem.

The combination of each pair of variables which are related by pure data transfers would cause these operations to be eliminated. This improvement reduces the number of control functions. If a horizontal list in the code sequence is occupied by pure data transfers, the control step can be deleted resulting in a faster implementation. To take this property into account, the reduction of the original graph is separated into two phases. In the first phase the edges which are associated with pure data transfers in some time intervals are collected to form a subgraph. Let the original graph and the subgraph be represented by G and $G1$, respectively. The edges in G and $G1$ satisfy the loose form transitive property.[2] Algorithm 3 in the appendix can be applied. Having completed the partitioning of the subgraph, Algorithm 1 is then applied to the remaining edges of the original graph.

Once the variables have been compacted, the code sequence is updated. The names which are grouped together are assigned to the same name. Operations of moving the content of a variable to itself are deleted.

[2]Let A, B, and C be three variables whose live periods do not overlap. Furthermore, pure data transfers appear between A and B as well as A and C. But, there is no pure data transfer between B and C. If the variables A and B are assigned to the same storage element, pure data transfers will appear between C and the composite variable (A, B). The same effect can be observed if the variables A and C are assigned to the same storage element. This is referred to the transitive property for storage allocation.

TABLE I
A CODE SEQUENCE

```
V3  = V1 + V2 ;      V12 = V1
V5  = V3 - V4 ;      V7  = V3 * V6 ;     V13 = V3
V8  = V3 + V5 ;      V9  = V1 + V7 ;     V11 = V10 / V5
V14 = V11 and V8 ;   V15 = V12 or V9
V1  = V14 ;          V2  = V15
```

TABLE II
RESULTS OF LIFETIME ANALYSIS

L and D represent live and dead respectively.

Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15
Entry	L	L	D	L	D	L	D	D	D	L	D	D	D	D	D
1	L	L	L	L	D	L	D	D	D	L	D	L	D	D	D
2	L	D	L	L	L	L	L	D	D	L	D	L	D	D	D
3	L	D	L	L	L	L	L	L	L	L	L	L	D	D	D
4	D	D	D	L	D	L	D	L	L	L	L	L	D	L	L
5	L	L	D	L	D	L	D	D	D	L	D	D	D	L	L
Exit	L	L	D	L	D	L	D	D	D	L	D	D	D	D	D

5.3. Lifetime Analysis

According to the previous discussion, the lifetime analysis is an essential process for constructing the graph which identifies the constraints of storage sharing. The problem of lifetime analysis is well understood in the area of compiler design. Details are referred to [1].

5.4. Construction of the Lifetime Compatible Graph

Let the live/dead status of all the variables be represented by a lifetime list. The compatible graph is the graph consisting of all the edges which join combinable pairs of nodes. To construct a compatible graph, a complete graph which consists of all the nodes is first created. The lifetime list and the code sequence are then traced and inspected. Unless the conditions given in Section 5.1 are satisfied, the edges which join those variables which are live in the same time interval are deleted from the graph. If an edge has already been deleted, this step is ignored. Those edges which associate with pure data transfers in some time intervals are marked.

5.5. Grouping Registers into Scratch Pad Memories

Having assigned all the variables to suitable physical locations, the next step is to investigate the possibility of grouping several registers into sets of scratch pad memories. Those variables which have disjoint access time can be grouped together. This problem can also be formulated into the clique-partitioning problem.

5.6. An Example

Let the code sequence in Table I be given. Assume that the program is itself a loop. Having executed the statements in the last line, the control flow is passed back to the statements in the first line. Applying the lifetime analysis algorithm to the example, the status of these variables in each time interval is indicated in Table II.

Having derived the live/dead history of each variable, the compatible graph can be constructed. As mentioned

TABLE III
THE COMPATIBLE VARIABLE-PAIRS

(1,9)	(1,13)	(1,14)*	(2,3)	(2,5)	(2,7)
(2,8)	(2,9)	(2,11)	(2,13)	(2,15)*	(3,8)
(3,13)*	(3,14)	(3,15)	(4,13)	(5,8)	(5,11)
(5,13)	(5,14)	(5,15)	(6,13)	(7,9)	(7,13)
(7,14)	(7,15)	(8,13)	(8,14)	(9,13)	(9,15)
(10,13)	(11,13)	(11,14)	(12,13)	(12,15)	(13,14)
(13,15)					

before, a complete graph is first constructed. Using Table II, the conflict graph can be constructed in the following way. Considering the time interval defined as "1", the variables V1, V2, V3, V4, V6, V10, and V12 are live. Each edge formed by these live variables must be deleted from the graph. The nodes V2 and V3 are used as a source and destination in the same statement. In addition, the source variable is dead in the next time interval. Therefore, the edge (2, 3) is not deleted. Repeatedly applying the procedure to the entire lifetime, the resulting compatible graph is given in Table III.

Among these combinable variable-pairs, those edges which are accompanied by "*" are associated with pure data transfers in some time intervals. They are used to construct the second graph. Algorithm 3 is used to reduce these two graphs. It results in the following composite nodes: {1, 14], {2, 15}, and {3, 13}.

Applying Algorithm 1 to the reduced graph, the variables are finally partitioned into eight clusters. They are {1, 14}, {2, 7, 9, 15}, {3, 8, 13}, {4}, {5, 11}, {6}, {10} and {12}. The variables in each of these clusters can be assigned to the same physical location. The code sequence in Table I can be refined into the form in Table IV.

There are fifteen variables in the original code sequence. They have been compacted into eight. Furthermore, the last step has been eliminated. The program can now be realized in four steps.

VI. ALLOCATION OF DATA OPERATORS

The allocation of data operators consists of two tasks. One is the combination of the same kind of operators. The other is the grouping of various kinds of operators into arithmetic and logic units. The goal is to assign these data operations to the minimum number of clusters. The problem is again formulated into the clique-partitioning problem.

6.1. The Formulation

A data operator is called an isolated operator if it is exclusively assigned to a triple statement (a statement which consists of one operation, one or two sources, and one destination). What is the effect of grouping two isolated data operators into one composite unit? If these two operations are the same, the number of operators is reduced by one. In addition, depending on the corresponding source operands and the destination variables are the same or not, the number of multiplexers and wired-broadcast trees may be increased or decreased. What is the effect of merging an isolated operator into an Arithmetic Logic Unit (ALU) or combining two ALU's? First, the buses and the gating elements connected to the input and output ports of the ALU can be shared. If the operations have common sources or destination, the original gating elements for the input or output ports of these modules can also be shared. Therefore, it is generally beneficial to merge an isolated data operator into an ALU or to combine two ALU's [24]. An important issue for the allocation of data operators is choosing an appropriate set of operators to group together.

Let two isolated operations be given. Inspecting the relationship between these two operations, there are sixteen cases [24]. These sixteen cases can be classified into eight categories. They are listed below.

- *G8:* The operations and the three pairs of variables are all the same.
- *G7:* The operations are different but the three pairs of variables are the same.
- *G6:* The operations and two pairs of variables are the same. The third pair of variables is different.
- *G5:* Two pairs of the variables are the same. The operations and one pair of variables are different.
- *G4:* The operations and one pair of variables are the same. The other two pairs of variables are different.
- *G3:* One pair of the variables is the same. The operations and the other two pairs of variables are different.
- *G2:* The operations are the same. All three pairs of variables are different.
- *G1:* The operations and all three pairs of variables are different.

All of these subgraphs satisfy the generalized transitive property.[3] For simplicity, the loose form transitive property is assumed for them. The algorithm for allocating data operators can be described as follows:

1) Create a complete graph whose nodes are indices of all the data operators. Trace through the code sequence and delete those edges connecting nodes which are used simultaneously. Identify the category of each edge.

2) Collect edges of Category 8. Use Algorithm 3 to reduce ***G*** and ***G8***.

3) Having reduced the subgraph of Category 8, the graph ***G*** together with the subgraphs of Categories 7, 6, 5, 4, 3, 2, and 1 are reduced one by one.

6.2. An Example

Let each operation in Table IV be assigned to a specific name. An assignment is given in Table V. Using the code sequence and operator assignment, the compatible graph

[3]Let three data operators which do not have usage conflicts be given. The edge formed by each pair of these three operators belongs to one of the eight subgraphs. If two of these three nodes are merged, the edge formed by the remaining node and the composite node may belong to a new subgraph. This is referred to the transitive property for the allocation of data operators.

TABLE IV
IMPROVED CODE SEQUENCE

V3 = V1 + V2 ;	V12 = V1	
V5 = V3 - V4 ;	V2 = V3 * V6	
V3 = V3 + V5 ;	V2 = V1 + V2 ;	V5 = V10 / V5
V1 = V5 and V3 ;	V2 = V12 or V2	

TABLE V
ASSIGNING OPERATOR IDENTIFIERS

V3 = V1 $+_1$ V2 ;	V12 = V1	
V5 = V3 $-_1$ V4 ;	V2 = V3 $*_1$ V6	
V3 = V3 $+_2$ V5 ;	V2 = V1 $+_3$ V2 ;	V5 = V10 $/_1$ V5
V1 = V5 and_1 V3 ;	V2 = V12 or_1 V2	

Operator Identifiers:
$+_1$ $-_1$ $*_1$ $+_2$ $+_3$ $/_1$ and_1 or_1

Indices:	1	2	3	4	5	6	7	8

TABLE VI
G: THE EDGES OF THE ORIGINAL COMPATIBLE GRAPH

$(1,2)^1$	$(1,3)^1$	$(1,4)^4$	$(1,5)^6$	$(1,6)^1$	$(1,7)^1$	$(1,8)^3$
		$(2,4)^3$	$(2,5)^1$	$(2,6)^1$	$(2,7)^3$	$(2,8)^1$
		$(3,4)^3$	$(3,5)^3$	$(3,6)^1$	$(3,7)^3$	$(3,8)^3$
					$(4,7)^3$	$(4,8)^1$
					$(5,7)^1$	$(5,8)^5$
					$(6,7)^3$	$(6,8)^1$

G is depicted in Table VI. The superscript integer at the right side of each edge is the category identifier of the edge.

The edge of Category 6 in ***G*** is retrieved to form the subgraph ***G6***. ***G6*** only consists of one edge; it is (1, 5). The original graph ***G*** and the subgraph ***G6*** are first reduced. The nodes 1 and 5 are combined. In the reduced graph the categories of the edges (1, 3) and (1, 8) are updated to 3 and 5 respectively. The reduction procedure is continued until the list of edges becomes empty. The data operators are finally grouped into three clusters. They are {1, 3, 5, 8}, {2, 4, 7}, and {6}.

VII. ALLOCATION OF INTERCONNECTION UNITS

This section discusses the issues related to the allocation of interconnection units. The procedure involves two steps: alignment of operands and allocation of the interconnection units.

7.1. Alignment of Operands

An operation may be either commutative or noncommutative. For those commutative operations, the synthesizer has the freedom of flipping the position of the two operands. If the operands of all the operations are suitably aligned, the number of interconnection units can be decreased. Let the operands of unary and noncommutative operations be collected in two sets according to their positions with respect to the operations. The operands of the commutative operations can be suitably aligned by comparing the operands with variables in these two sets.

7.2. The Formulation

Interconnection variables which are never used simultaneously can be grouped together to form buses. The goal is to group the interconnection variables into the minimum number of clusters. The problem is again formulated into the clique-partitioning problem.

To obtain a good bus style design, it is essential to minimize both the number of buses and the total number of drivers and receivers [23]. There is no profit in combining two interconnection variables which originate from different sources and connect to different sinks. On the other hand, it is generally beneficial to group those interconnection variables which originate from the same source or connect to the same sink to share a common bus. The details of the formulation are given below.

A complete graph in which nodes consist of all the interconnection variables is first constructed. Then the code sequence is traced through. In each time interval, if two interconnection variables are used simultaneously, the edge formed by these two nodes is deleted. Those interconnection variables which are associated with the same source, even when they are used concurrently, can still share a common interconnection. Therefore, when we construct the compatible graph, these entries are not deleted.

Let the compatible graph be represented by ***G***. When the compatible graph is constructed, if two interconnection variables originate from the same source or connect to the same sink, the edge which joins these two variables is marked. All the marked edges are collected to form the second graph (named ***G1***). The loose form transitive property is applicable to ***G*** and ***G1***.[4] Algorithm 3 can be applied.

7.3. Refining the Initial Allocation

An initial design might have a "join-node" in which more than one bus is connected to a single input port. It is necessary to insert a multiplexer in front of the input port. The "join-node" can easily be found by checking the data paths connected to an input port. If an input port is connected to more than one bus, then the node needs to be refined.

7.4. An Example

The example is based on the code sequence in Table V and the ALU's allocated in Section 6. Inspecting the operands of the operations associated with ALU2, it is found that the positions of the two operands of the statement "$V1 = V5\ \text{and}_1\ V3$" need to be flipped. It is changed into the form of "$V1 = V3\ \text{and}_1\ V5$". Using the indices

[4]Let three interconnection variables i, j, and k be given. Assume there are no usage conflicts among these three variables. Furthermore, the variables i and j as well as i and k have common source or common destination. But, the variables j and k do not originate from the same source or connect to the same sink. If the variables i and j are assigned to the same bus, the variable k and the composite variable (i, j) will have common source or common destination. The same effect can be observed if the variables i and k are assigned to the same bus. This is referred to the transitive property for bus allocation.

TABLE VII
INDICES OF INTERCONNECTION VARIABLES

Source Name	Destination Name	Indexing Integer
V1	V12	1
V1	ALU1.In1	2
V2	ALU1.In2	3
V3	ALU1.In1	4
V3	ALU2.In1	5
V4	ALU2.In2	6
V5	ALU2.In2	7
V5	ALU3.In2	8
V6	ALU1.In2	9
V10	ALU3.In1	10
V12	ALU1.In1	11
ALU1.Out	V2	12
ALU1.Out	V3	13
ALU2.Out	V1	14
ALU2.Out	V3	15
ALU2.Out	V5	16
ALU3.Out	V10	17

TABLE VIII
G: LIST OF EDGES WHICH JOIN COMBINABLE INTERCONNECTION VARIABLES

[1,2]	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(1,9)	(1,10)	(1,11)	(1,12)	(1,14)	(1,15)
(1,16)	(1,17)	[2,4]	(2,6)	(2,9)	[2,11]
(2,14)	(2,16)	(3,4)	(3,6)	[3,9]	(3,16)
[4,5]	(4,7)	(4,8)	(4,10)	[4,11]	(4,13)
(4,14)	(4,15)	(4,17)	(5,13)	[6,7]	(6,8)
(6,10)	(6,11)	(6,13)	(6,14)	(6,15)	(6,17)
[7,8]	(7,9)	(7,13)	(7,16)	(8,9)	(8,11)
(8,13)	(8,14)	(8,16)	(9,10)	(9,11)	(9,13)
(9,14)	(9,15)	(9,17)	(10,11)	(10,13)	(10,14)
(10,16)	(11,13)	(11,15)	(11,16)	(11,17)	[12,13]
(13,14)	[13,15]	(13,16)	(13,17)	[14,15]	[14,16]
(14,17)	[15,16]	(16,17)			

in Table VII the compatible graph in Table VIII (*G*) is constructed. In Table VIII those edges which join interconnection variables with the same source or the same sink are enclosed by square brackets. The graph formed by these edges is called *G1*.

In *G1* nodes 14 and 16 have the maximum number of common neighbors. They are combined. *G* and *G1* are reduced. The next node to be selected should be node 15. Inspecting the nodes which are connected to node 15 and the composite node {14, 16}, it is found that both (13, 14) and (13, 15) are contained in *G*. However, among them, only (13, 15) belongs to *G1*. Since the edge (13, 15) is being deleted, the edge (13, 14) should be added into *G1*. The nodes 13 and 14 are then combined to form the composite node {13, 14, 15, 16}. This composite node forms a cluster.

Repeatedly applying the above algorithm to *G* and *G1*, the interconnection variables are finally partitioned into eight groups. They are {13, 14, 15, 16}, {1, 2, 4, 11}, {6, 7, 8}, {3, 9}, {5}, {10}, {12} and {17}. Fig. 1 depicts the completed allocation of the data-memory part.

VIII. IMPLEMENTATION

As indicated in Section I, the input to the *Emerald* design generator is the value trace. When a value trace is read in, it is represented as a hierarchical linked list. At the top level, it contains a list of VT-bodies which assume the role of procedures in other high-level languages. Each VT-body points to a list of basic blocks. Each basic block

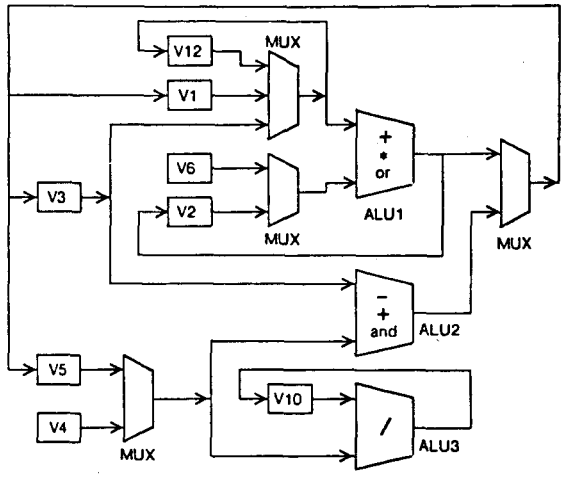


Fig. 1. Data paths of the example.

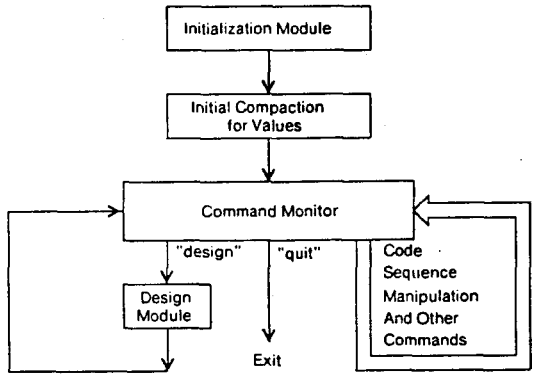


Fig. 2. Organization of the design generator.

contains a two-dimensional list of operations. All the operations which are executed in the same time interval are linked in a horizontal list. A horizontal list is named a *CodeList*. The *CodeLists* are vertically chained, in order of execution, to form a basic block.

The output is an Allocated Data Path (*ADP*) which consists of five linear lists: storage elements, constants, zeros and ones for padding operations, operators, and interconnection units.

Each operation node in the code sequence contains all the information associated with the operation which includes pointers to the storage elements, data operators, and interconnection units in the *ADP*. Each operator in the *ADP* also contains a pointer back to the operation.

The code sequence representation resembles an assembly language program. The last operation in a basic block generally specifies a transfer of control. Possible transfers of control include jumping directly or indirectly to the beginning of another basic block or the beginning of a VT-body.

The structure of the *Emerald* design generator is depicted in Fig. 2. The initialization module reads in the VT file and creates internal data structures for the specification. The design generator then performs some initial

compaction for values.[5] The command monitor processes commands entered by the designer. Commands are provided for manipulating the code sequence, printing the code sequence, activating the design process, etc. Once the designer specifies the code sequence, the design module can be invoked to complete a design for the code sequence. Functions of the design module include further compaction of values, packing data operations into a suitable number of ALU's or Aggregate Comparison Units (ACU's), and the allocation of interconnection units among operators and storage elements. Using the commands provided by the command monitor, the designer can generate as many designs as he/she likes. These commands can be categorized in two groups. Details will be described in the next section.

IX. Mechanisms for Design Space Exploration

The two categories of mechanisms for exploring design alternatives are manipulation of the code sequence and selection of design criteria.

9.1. Manipulation of the Code Sequence

Given a code sequence, *Emerald* creates a design for the code sequence. Alternative designs can be generated by manipulating the code sequence. As long as a modification of the code sequence does not violate data dependency among different operations, the designer can manipulate the code sequence in whatever way he/she likes.

For simplicity, the modification of a code sequence is limited to one operation at a time. There are two possible manipulations:

1) Move an operation from its current *CodeList* to another *CodeList*.
2) Serialize the code sequence by inserting a new *CodeList* between two *CodeLists* and move the operation to the new *CodeList*.

When the designer inputs a modification request, the system needs to check whether the change is allowed. Let the manipulation be limited to a basic block. Generally speaking, three conditions need to be checked:

1) Is the operation allowed to move away from its current *CodeList*?
2) Is the operation allowed to move to the new *CodeList*?
3) Is the operation allowed to move across all the *CodeLists* located between the current *CodeList* and the destination *CodeList*?

The relocation of an operation may either be in the forward or the backward direction. A forward relocation moves the operation to a *CodeList* before the current *CodeList*. A backward relocation moves the operation to a *CodeList* after the current *CodeList*. The above three conditions for moving an operation are different for forward and backward relocations.

9.2. Selection of Design Criteria

The allocation of data paths for digital systems is a complicated process; it involves making many decisions. This subsection presents some issues relevant to the selection of design criteria.

When an ISPS description is translated into a value trace, the association of a value with its ISPS counterpart is generally available. Some variables in the ISPS description are defined for input/output operations. To preserve their original roles, these variables should not be combined with other variables. *Emerald* provides the designer with the capability of specifying *reserved variables*. Those values introduced by a *reserved variable* are packed and assigned to the original variable before the compaction of other values is done. These values are not combined with other values. *Emerald* allocates a unique storage element for each reserved variable. In the resulting design, the storage element serves the same role as that defined in the behavioral specification.

Emerald treats all the array memories defined in the behavioral specification as reserved variables. The access of an array memory requires memory address registers and memory data registers to hold the address and data. Conventional von Neumann machines contain a single memory address register and a single memory data register. Processors designed recently often used multiport memory. A multiport memory has multiple address and data registers. *Emerald* provides the designers with both options. The first option is to allocate a dedicated memory address register and memory data register for an array memory. The second option is to let the system compact those values associated with memory accesses into a suitable number of memory address registers and memory data registers.

Having allocated the storage elements and constants, the process proceeds to the compaction of data operators. Assigning zero to a variable can be realized either by moving a zero to the variable or by a "clear" operation. Shifting functions can be implemented in either a barrel shifter or a shift register. Adding by one can be implemented as a counter, a local incrementer, or as part of an ALU. Similarly, subtracting by one can also be realized as a counter, a local decrementer, or as a part of an ALU. Commercial systems often use dedicated hardwares to realize special functions. Options for implementing some special functions are also provided in *Emerald*. *Emerald* asks the designer to make a decision if there is more than one choice. Experiments can be performed to investigate the effects of using different hardware.

X. Evaluation Criteria

The evaluation of designs is generally technology dependent. Even with the technology specified, it is still very difficult to estimate the exact cost and performance of a

[5]To relate the data flow among different basic blocks and VT bodies, the value trace contains interface values. Direct data transfers are introduced to link pairs of these values. These pairs of values, unless they are generated from formal parameters of procedure calls, can generally be combined. Therefore, they are compacted first.

design. To the best of our knowledge, evaluation models which are universally acceptable are still not available. The development of such a universal evaluation model was not a goal of this research. Instead, component counts are used to compare the cost of different designs. Several cost measures are provided. For data operators, the number of individual operators and the number of aggregate modules (the number of ALU's and ACU's) are presented. For storage elements, the number of registers and the number of bits for these registers are calculated. Since memory arrays are identical in all designs, their costs are not considered. For interconnection units, the number of multiplexers is given. The number of inputs for each multiplexing function is also specified. A multiplexer with n inputs can be realized by $n - 1$ two-input multiplexers. The total number of two-input multiplexers is calculated to compare the cost of interconnection units for each design.

Some typical TTL chips [22] are used to estimate the total number of gates required for a design. The SN74157 quadruple two-input multiplexers contains 15 gates. Therefore, each two-input multiplexer is assumed to contain 3.75 gates. The SN7476 JK flip-flop contains 32 gates for four bits. Each bit of storage element is assumed to require 8 gates. These two units will be used to calculate the number of gates required for storage elements and multiplexers. The number of gates required for aggregate data operators is separately estimated. For those aggregate data operators which contain simple operations such as AND, OR, NOT, EQL, NEQ, the number of gates is assumed to be the sum of gates required to realize these functions. For composite data operators which contain several operations, the SN74181 ALU chip is used. Delays of some TTL chips are used to evaluate the delays of some interconnection paths. Some experimental results are presented in the next section.

IX. Experimentation

Emerald provides a fairly large number of mechanisms for exploring design alternatives. Given a VT, the number of alternative designs is almost unlimited. To validate the usefulness of the allocation procedure, the choice of an appropriate set of experiments is an important task. At this time, our major concern focuses on inspecting the quality of designs created by the allocation procedure, checking versatility of the design domain, and examining the design space. To achieve these goals, results of experiments on the ISPS descriptions for the AM2910, the AM2901, and the IBM System/370 were chosen for demonstration.[6] These three behavioral descriptions differ significantly in functions. Designs generated for these specifications are carefully investigated. Data paths produced for the AM2910 and the AM2901 are compared with the commercial designs. In cases where differences are observed, reasons for the differences are discussed. The comparisons provide an alternative way for evaluating the quality of the allocated designs. The design space for the AM2901 ISPS description is extensively explored.

[6]The ISPS descriptions used in this paper were excerpted from the library of ISPS descriptions on a computer system at Carnegie-Mellon University, Pittsburgh, PA.

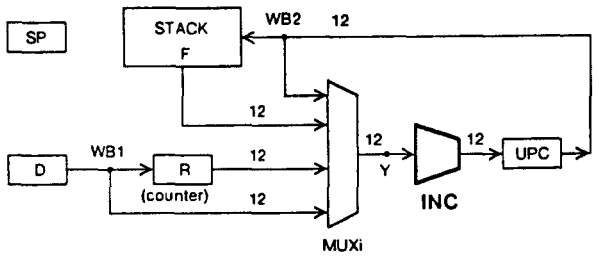


Fig. 3. Data paths of the commercial AM2910.

TABLE IX
A Segment of the AM2910 ISPS Description

```
Y()<11:0> :=
   begin

   DECODE i =>
     begin
     "0 := JZ    := (Y = SP = 0; FULL = 1),

     "4 := PUSH  := (Y = uPC; push.(); IF pass => R = D),

     "C := LDCT  := (Y = uPC; R = D),

     "E := CONT  := (Y = uPC),
     end next

   end
```

11.1. Experiment on the AM2910

The AM2910 is a bipolar microprogram controller intended for use in high-speed microprocessor applications. Its data paths are depicted in Fig. 3. The controller contains a four-input multiplexer that is used to select either the register/counter (R), direct input (D), microprogram counter (uPC), or stack as the source of the next microinstruction address. The microprogram counter is composed of a 12-bit incrementer followed by a 12-bit register. Further details can be found in [2].

The VT file for the AM2910 contains 19438 bytes. Based on the VT, *Emerald* created a code sequence and allocated the data-memory. The CPU time for this run was 280 s on a VAX-11/780.

There are small differences between the code sequence and the original ISPS description. A segment of the original ISPS description is given in Table IX to illustrate the differences. As depicted in Table IX, the procedures *PUSH*, *LDCT*, and *CONT* contain the statement $Y = uPC$. *Emerald* combines the values for variables uPC and Y in the statement $Y = uPC$ of procedures *PUSH*, *LDCT*, and *CONT*. These operations are deleted. However, these data transfers are not eliminated. They appear as the data transfers for linking interface values between the calling and called VT-bodies. Therefore, they are moved into the context of the calling procedure Y.

Fig. 4 is the fastest implementation allocated by *Emerald*. Comparing Figs. 3 and 4, small differences can be observed. In Fig. 3 a wired-broadcast tree (*WB1*) is used to link D, R, and the multiplexer *MUXi*. Another wired-

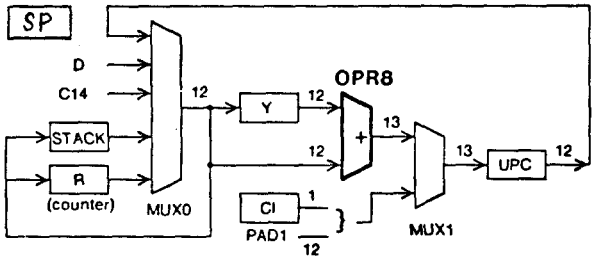


Fig. 4. Emerald's design for the AM2910.

TABLE X
QUANTITATIVE COMPARISONS FOR THE AM2910

Design Id	Emerald's Design	Commercial Design
# of Individual data operators	1	1
# of Gates Required for Data Operators	108	108
# of 9-bit Registers	1	1
# of 12-bit Registers	2	4
# of 13-bit Registers	2	0
Total No. of Bits	59	57
# of Gates Required for Storage Elements	472	456
Total No. of 2-input Multiplexers	61	36
# of Gates Required for Multiplexers	229	135
Total No. of Gates Required for the Design	809	699

broadcast tree (*WB2*) is used to link the output port of *uPC*, input ports of *STACK*, and *MUXi*. To minimize the number of buses, *Emerald* merges these two wire-broadcast trees and *MUXi* into a larger multiplexer.

Emerald recognized that *R* is associated with the operation of incrementing by one and asked the designer to make a choice among counter, local incrementer, and ALU. In the commercial design *R* is implemented as a counter. For consistency, it was requested to be implemented as a counter.

In the commercial design an incrementer is used to perform the operation $uPC = Y + CI$. The "*CI*" is a single-bit register. To maintain consistency in size for operands of the operation, a padding operation is included in the value trace. A padding operation introduces a suitable number of zeros or ones to the most significant bit-positions of the original value. *Emerald* does not recognize that the operation can be implemented as a counter or incrementer. Instead, an ALU with a single add operation is allocated. The statement $uPC = Y + CI$ is translated into two statements: moving the output of the padding operation into *uPC* followed by $uPC = Y + uPC$.

C14 is a constant introduced in the ISPS description. This constant is not shown in the commercial design.

Table X compares the number of registers, data operators, and multiplexers used by the commercial and allocated designs. Since detailed information of the status bits is not available for the commercial design, comparisons are done only for the 12-bit data paths.

As indicated in the table, the allocated design uses two more bits than the commercial design. These two bits are the *carry* bits of arithmetic operations. The number of data operators is the same for these two designs. As for interconnection units, the allocated design needs 61 while the commercial design uses only 36 two-input multiplexers. The value trace introduces one constant and one variable, which require additional interconnection units to direct the data flow. One constant (*C14*) increases the number of inputs of a multiplexer (*MUX0*) by one. The variable *CI* (*carry-in*) and the padding bits create an extra multiplexer (*MUX1*). The use of *C14* and *CI* accounts for 25 two-input multiplexers.

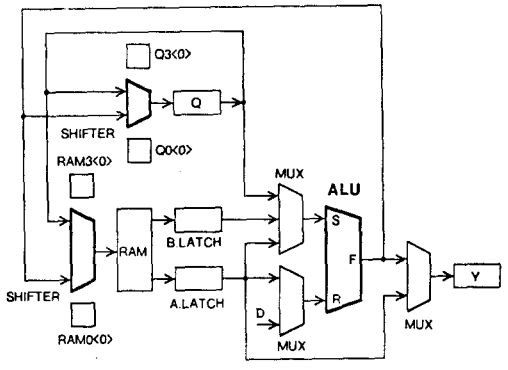


Fig. 5. Data paths of the commercial AM2901.

If the SN7483 is used as the adder or incrementer, the number of gates required for data operator is 108. Gate counts for other entities can be found in Table X. The total number of gates used by *Emerald's* design is 15.7 percent more than the commercial design.

The longest interconnection path for the commercial design consists of a four-input multiplexer (*MUXi*) while the longest interconnection path for Emerald's design consists of a five-input multiplexers. If the SN74153 and the SN74351 chips are used, these two data paths have the same delay of 17 ns.

11.2. *Experiment on the AM2901*

The AM2901 is a bit-sliced microprocessor chip which consists of a 16-word by 4-bit two-port RAM, a high-speed ALU, and the associated shifting, decoding and multiplexing circuitry. Its data paths are depicted in Fig. 5. Additional details can be found in [2].

The original ISPS description for the AM2901 contains eight procedures. Among them three procedures are used to compute the status bits. In practice, these status bits are not computed separately. They are the side effects of the arithmetic or logic operations and are, therefore, generated by a small number of AND, OR, and NOT gates. The three procedures for computing status bits introduce many data operations to the main data flow. To reduce the difficulty in analyzing the data, the algorithms for computing the status bits were deleted. Based on the remaining behavioral description, a VT file which contains 8421 bytes was generated. Using this VT file as input, the fastest implementation generated by *Emerald* is compared with the

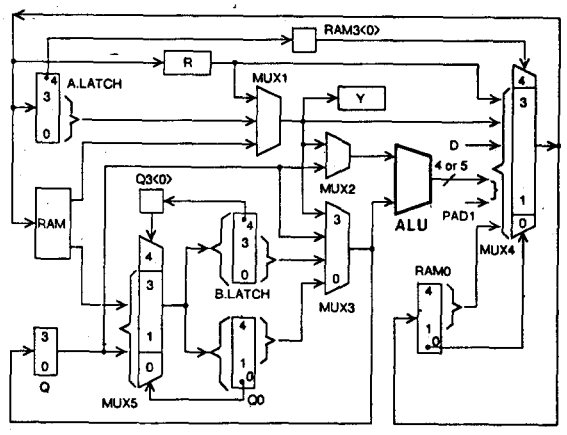


Fig. 6. Data paths allocated from the ISPS for the AM2901.

TABLE XI
A SEGMENT OF THE AM2901 ISPS DESCRIPTION

```
**Access Computation**{us}

source :=          ! Source calculation
  begin
  A.LATCH = RAM[A]; B.LATCH = RAM[B] next
  DECODE src =>
    begin
    #0  := (R = A.LATCH; S = Q
    #1  := (R = A.LATCH; S = B.LATCH),
    #2  := (R = 0      ; S = Q
    #3  := (R = 0      ; S = B.LATCH),
    #4  := (R = 0      ; S = A.LATCH),
    #5  := (R = D      ; S = A.LATCH),
    #6  := (R = D      ; S = Q
    #7  := (R = D      ; S = 0
    end
  end;

destination :=
  begin
  DECODE dest =>
    begin
    #0  := (Q = F; Y = F),
    #1  := (Y = F),
    #2  := (Y = RAM[A]; RAM[B] = F),
    #3  := (Y = F; RAM[B] = F),
    #4  := (Y = F; RAM[B] @ RAM0 = RAM3 @ F;
            Q @ Q0 = Q3 @ Q),
    #5  := (Y = F; RAM[B] @ RAM0 = RAM3 @ F),
    #6  := (Y = F; RAM3 @ RAM[B] = F @ RAM0;
            Q3 @ Q = Q @ Q0),
    #7  := (Y = F; RAM3 @ RAM[B] = F @ RAM0)
    end
  end.
```

commercial implementation. The CPU time for this run was 25 s on a VAX-11/780.

A segment of the original ISPS description for the AM2901 is given in Table XI. During the lifetime analysis, values generated by the variable Y are sometimes concluded as *dead values* (values which are defined and never used). Y was assumed to be a reserved variable. D represents an external input. It was intentionally put in the list of reserved variables in the beginning of the design process.

Comparing the resultant code sequence and the original ISPS description, some differences were observed. First, the variables S and Q are merged. The procedure *SOURCE* contains several statements that move the conent of Q into S ($S = Q$). These statements are eliminated. Another difference is that the values associated with the variables R and F are assigned to the same register (represented as R in the allocated data paths). Finally, the statement Q @ $Q0 = Q3$ @ Q are translated into two statements. The case becomes $Q0\ \langle 4:0\rangle = Q3\ \langle 0:0\rangle$ @ $Q\ \langle 3:0\rangle$ followed by $Q\ \langle 3:0\rangle = Q0\ \langle 4:1\rangle$. A similar result was found for the shifting function associated with the input of the array memory.

TABLE XII
QUANTITATIVE COMPARISONS FOR THE AM2901

Design Id	Emerald's Design	Commercial Design
# of ALUs	1	1
# of Individual data operators	7	7
# of Gates Required for Data Operators	70	70
# of 1-bit Registers	2	4
# of 4-bit Registers	5	10
# of 5-bit Registers	5	0
# of 9-bit Registers	1	1
Total No. of Bits	56	53
# of Gates Required for Storage Elements	448	424
Total No. of 2-input Multiplexers	48	32
# of Gates Required for Multiplexers	180	120
Total No. of Gates Required for the Design	698	614

Fig. 5 depicts the data paths of the commercial AM2901 while Fig. 6 illustrates the design produced by *Emerald*. Comparing the allocated design with commercial design leads to some observations. Since the ISPS description for the AM2901 contains simultaneous accesses to the *RAM*, a two-port memory was allocated. The memory data registers *A.LATCH*, *B.LATCH*, and *Y* are either connected to the first output port of the array memory through the multiplexer *MUX*1 or connected to the second output port through the multiplexer *MUX*5. Another interesting observation is on the implementation of the shifting function. As indicated in Table XI, the ISPS description specifies the shifting function as a concatenation operation. The statement Q @ $Q0 = Q3$ @ Q is an example. *Emerald* does not recognize this statement as a shifting operation. Instead, three registers $Q3$, Q, and $Q0$ which contain one, four, and five bits were allocated. These three registers and two multiplexers *MUX*5 and *MUX*3 are used to realize the shifting function. A similar structure was allocated for the shifting operation in front of the input port of the array memory. The associated data paths are *RAM*0, *RAM*3, *R*, *MUX*1, *MUX*2, and *MUX*4. Unlike Q, $Q0$, and $Q3$, R is not exclusively used for the shifting function. It is shared by other operations. If multiplexers for the shifting function are isolated from the rest of the data paths, the allocated data paths are quite similar to the commercial implementation.

Table XII compares the number of registers, data operators, and multiplexers used by the commercial and allocated designs. The following assumptions were made:

1) Y, which is a tri-state output buffer in the commerical design, is considered as a register.

2) Each of the input and output ports of the ALU (R, S, and F) is counted as one register.
3) The external input (D) is considered as a register.

Including the two 4-bit memory address registers for the RAM, which are not shown in Fig. 5, the commercial design contains four 1-bit registers, ten 4-bit registers, and one 9-bit register. As indicated in the table, the allocated design uses 3 more bits than the commercial design. These bits are essentially allocated for the concatenation operations. The number of data operators is the same for these two designs. As for interconnection units, the commercial design uses twelve 3-input multiplexers[7] and eight 2-input multiplexers. This is equivalent to thirty-two 2-input multiplexers. Emerald's design needs 48 2-input multiplexers. Constants and registers for concatenation operations account for the necessity of the extra interconnection units.

If the SN74181 is used as the arithmetic and logic unit, the number of gates required for the data operator is 70. Gate counts for other entities can be found in Table XII. The total number of gates used by *Emerald's* design is 13.7 percent more than the commercial design.

The longest interconnection path for Emerald's design is from *A. LATCH* through *MUX*1 and *MUX*3, to the input of the ALU. This data path can be realized with two cascaded 4-input multiplexers. The longest interconnection path of the commercial design can be realized by a 4-input multiplexer. If the SN74153 chip is used, the typical delays of the commercial design and *Emerald's* design are 17 and 34 ns, respectively.

11.3. Exploring the Design Space for the AM2901 ISPS Description

A VT file which contains 16664 bytes was generated from the ISPS description of the AM2901. This ISPS description uses three procedures to compute condition codes and status bits. In these three procedures several operations can be simultaneously executed. *Emerald* creates designs by compaction. Therefore, the fastest implementation created by *Emerald* contains several ALU's. The significant difference in cost and speed between designs with different number of ALU's forms an interesting design space.

Looking at the initial code sequence, the maximum number of arithmetic and logic operations appearing in a *CodeList* is five. The design space is explored in such a way that the maximum number of arithmetic and logic operations in a *CodeList* is reduced by one at a time. Having serialized all the arithmetic and logic statements, simultaneous memory accesses were serialized. The last design resulted from serializing all the statements. In total, seven designs were generated. Data associated with these designs are summarized in Tables XIII–XVII[8]. The calculation of the average number of steps required is based on three assumptions:

1) Equal probability is assumed for each branch associated with a branching operation.
2) The time for transfers of control is not considered.
3) Memory access is assumed to require one step.

Some observations on the data included in these tables are given below:

1) The structure of storage elements remains un-

[7]Each of the shifting functions contains four 3-input multiplexers.

[8]These data are for the AM2901 ISPS description which includes the three procedures for computing status bits. Table XII, on the other hand, contains data for the simplified ISPS description.

TABLE XIII
DATA ON STORAGE ELEMENTS

Design Id	#1	#2	#3	#4	#5	#6	#7
# of 1-bit Reg	13	13	13	13	13	13	13
# of 4-bit Reg	14	14	14	14	14	14	13
# of 5-bit Reg	5	5	5	5	5	5	5
# of 9-bit Reg	1	1	1	1	1	1	1
Total No. of Bits	103	103	103	103	103	103	99
# of Gates for Storage Elements	824	824	824	824	824	824	792

TABLE XIV
DATA ON OPERATORS

Design Id	#1	#2	#3	#4	#5	#6	#7
# of 1-bit ALUs	7	7	7	7	7	7	7
# of 1-bit Individual Data Operators	11	11	11	11	11	11	11
# of 4-bit ALUs	5	4	4	3	1	1	1
# of 4-bit Individual Data Operators	16	16	13	13	7	7	7
# of 3-bit ACUs	2	2	2	2	1	1	1
# of 3-bit Individual Comparators	3	3	3	3	2	2	2
# of 4-bit ACUs	1	1	1	1	1	1	1
# of 4-bit Individual Comparators	2	2	2	2	2	2	2
# of Gates for Data Operators	160	160	148	148	112	112	112

TABLE XV
DATA ON INTERCONNECTION UNITS

Design Id	#1	#2	#3	#4	#5	#6	#7
# of 2-input MUX	19	31	12	19	4	4	4
# of 3-input MUX	18	18	13	10	4	3	2
# of 4-input MUX	1	2	2	7	0	1	1
# of 5-input MUX	9	8	5	5	8	8	0
# of 6-input MUX	1	4	14	7	3	3	0
# of 7-input MUX	4	1	5	4	3	7	4
# of 8-input MUX	1	6	1	3	4	0	3
# of 9-input MUX	7	2	0	0	2	2	5
# of 10-input MUX	0	0	0	1	4	4	1
# of 11-input MUX	0	0	0	0	0	0	1
# of 13-input MUX	0	0	0	0	0	0	1
# of 14-input MUX	0	0	0	0	0	0	1
# of 16-input MUX	0	0	0	0	0	0	1
Equivalent Total No. of 2-input MUX	186	189	171	169	157	154	155
# of Gates for Multiplexers	698	709	642	634	589	578	582

TABLE XVI
COST AND PERFORMANCE FOR EACH DESIGN

Design Id	#1	#2	#3	#4	#5	#6	#7
Total No. of Gates Required	1682	1693	1614	1606	1525	1514	1486
Average No. of Steps	17 3/4	18	19 1/8	20 7/8	24	25	31 5/8

TABLE XVII
RUN TIME FOR EACH DESIGN

Unit: seconds

Design Id	#1	#2	#3	#4	#5	#6	#7
User Time	122.0	102.0	83.2	88.5	73.7	72.4	69.5
System Time	10.3	11.3	7.6	9.9	9.0	7.0	5.1
Total Run Time	132.3	113.3	90.8	98.4	82.7	79.4	74.6

changed for the first six designs. The code sequence for the last design was carefully arranged so that the live periods of some registers are separated. Therefore, the resulting allocation contains fewer bits than other designs. The association between values and allocated storage elements for these seven designs does show slight differences. Generally speaking, the tradeoffs on storage elements are not significant.

2) As expected, the number of aggregate and individual data operators strongly depends on the amount of parallelism. Serialization of arithmetic and logic operations generally reduces the number of aggregate and individual data operators.
3) Besides Design #2, the number of two-input multiplexers and the total number of gates required for a design decrease as the code sequence is serialized.
4) The effect of data paths on the performance is obvious. The ratio of speeds between the fastest and the slowest implementations is approximately 1.78.
5) The program was run on a *Unix*[9] operating system for the VAX-11/780. The run time for the initialization phase is approximately 76 seconds, which includes 62 seconds of user time and 14 seconds of system time.[10] The average run time for generating one design is slightly less than 96 s. The entire run for generating seven designs spends 748 seconds of CPU time (including run time for preparing the initial code sequence).

11.4. Experiment on the IBM System/370

The IBM System/370 is a mainframe computer. An ISPS description for its behavior is available on a computer system at Carnegie-Mellon University. The VT file translated from this ISPS description contains more than 1.5 Mbytes. The total number of input and output values for all the VT-bodies is more than 15 000. In order to handle large designs, *Emerald* was reimplemented with a checkpointing facility to allow partial processing to survive system crashes. Dubbed *Emerald II*, the new implementation successfully completed a design for the IBM System/370. It took 68 h to prepare the code sequence. Using the backup file for the code sequence as input, it took five and one-half hours to generate one design (30 min were spent in restoring the internal data structures). A design which took all the design options by default is presented in this section.

The ISPS description for the IBM System/370 contains specifications in register sets, constants, memory mapping, input/output operations, and instruction sets (including floating point instructions). Six array memories are declared: the main memory (*MB*), the storage keys (*STKEYS*), the general purpose registers (*R*), the floating point registers (FPREGISTER), and the floating error registers (*FVU*). The arithmetic and logic instructions essentially operate on 32-bit operands. However, 64-bit operations are included to perform multiplication, division, modulus, and dynamic address translation. In addition, floating point instructions introduce some 68-bit operations while memory mappings require 8-bit and 24-bit operations. Indeed, the IBM System/370 contains a large number of data types. The architectural details can be found in [16].

Comparing the code sequence generated by *Emerald II* and the original ISPS description, small deviations were observed. In ISPS the accesses to main memory may be of 8-bit, 16-bit, 32-bit, or 64-bit data transfers. In the VT these memory accesses are converted into a suitable number of 8-bit accesses. Another peculiarity is on the operations of reading and writing a subfield of a variable. In general, a constant is used to identify the offset of the subfield to be transferred. In the VT for the IBM System/370, some of these operations use a variable to specify the offset.[11] In this case the subfield to be transferred may change with time. To appropriately direct the data transfer, multiplexing and demultiplexing *operators* are introduced in the data paths. Besides these two points, the resultant code sequence stay fairly close to the original ISPS description.

The registers allocated by *Emerald II* can be classified into five categories. According to their function, they can be for global declarations, local declarations, procedure names with size attribute, formal parameters, and working storage such as temporary registers or status bits, etc.

As mentioned above, the IBM System/370 contains a large number of data types. The size of operations for these data types vary in a wide range. When the compatible graph for the data operators is constructed, *Emerald* checks the size of each pair of operators. If the difference in size is larger than a given constant, the pair is deleted from the graph. *Emerald* asks the designer to input the constant before allocating the aggregate data operators. The default value for the constant is one for ALU's and zero for ACU's. In other words, if the size of two arithmetic or logic operations differs in less than two bits, then they can be combined. Otherwise, they are separated. Two

[9] *Unix* is a trademark of AT&T Bell Laboratories.

[10] The system time is the time spent in serving operating system calls while the user time is the time used by the user program.

[11] For example, a variable is used to select the interrupt channel.

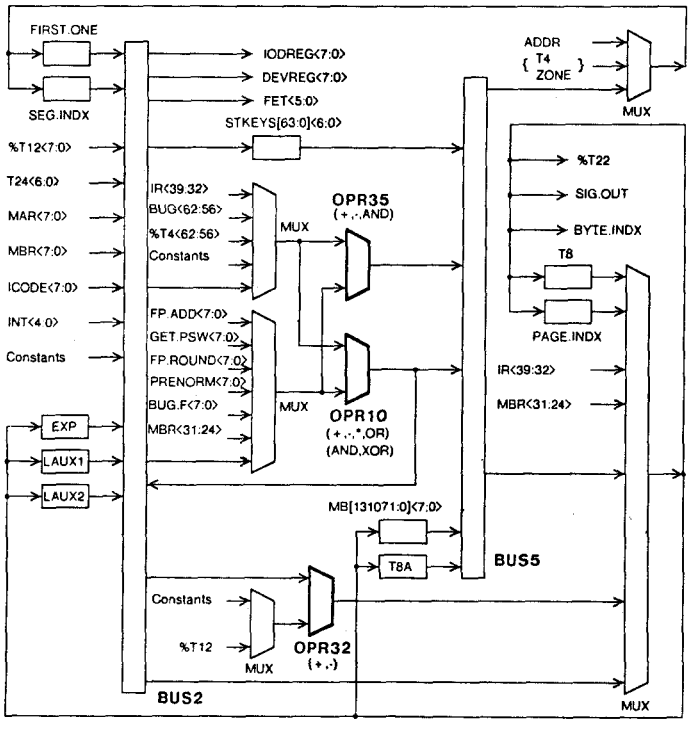


Fig. 7. 8-bit data paths of the IBM System/370.

TABLE XVIII
COMPONENT COUNTS FOR THE IBM SYSTEM/370

# of ALUs	26
# of Individual Data Operators in ALUs	64
# of ACUs	19
# of Individual Data Operators in ACUs	37
# of Multiplexing and Demultiplexing Operators	4
# of Gates Required for Data Operators	13950
# of Bits for Registers	2207
# of Bits for Shift Registers	522
Total No. of Bits	2729
# of Gates Required for Storage Elements	22224
Total No. of 2-input Multiplexers	7599
# of Gates Required for Multiplexers	28496
Total No. of Gates Required for the Design	64670

comparators can be combined only if they are of the same size. Therefore, many ALU's and ACU's were allocated and their sizes range from 1–112 bits.

Several interconnections result from the combination of global data flow and are, therefore, good candidates for a bus structure. Two of these structured interconnections which contain large number of input and output ports are represented as general buses. Data transfers between pairs of storage elements and data operators flow through these global buses.

The 8-bit data paths are depicted in Fig. 7. Component counts for various entities are given in Table XVIII. The total number of gates required for the design is estimated to be 64670. Further details of the design can be found in [25].

The longest interconnection path consists of a general bus followed by a 137-input multiplexer. If the SN74150 sixteen-input multiplexers are cascaded to form the 137-input multiplexer, typical delay for the longest data path is 46 ns.

XII. CONCLUSIONS

This paper presents a unifying procedure for the automated synthesis of data paths in digital systems. Based on the methodology, a design generator was implemented. Extensive experiments have been done with the design generator. Designs for the ISPS descriptions of the AM2910, the AM2901, and the IBM System/370 were selected to show the quality and limitations of automated synthesis. Generally speaking, if the design expert is restricted to the ISPS description, designs generated by *Emerald* are expected to be nearly identical to the design created by human experts. The design generator was able to generate data paths for a mainframe computer, the IBM System/370. Its practicality is verified.

The system can be enhanced in many ways such as augmenting the capability to recognize equivalent functions, providing the designer with the possibility of local modification, etc. Various scheduling algorithms for adjusting the code sequence can also be incorporated to facilitate different approaches of exploring a design space. Since the same procedure is applied to the allocation of storage elements, data operators, and interconnection units, investigating the effects of changing the allocation order for these three resources is also an interesting research. If the system is integrated into an automated design environment, the effort of digital design can be significantly reduced.

APPENDIX
THE CLIQUE-PARTITIONING ALGORITHMS

Let G be an undirected graph and its nodes be indexed by integers. Assume nodes i and j are connected and i is smaller than j. The edge which joins nodes i and j is represented by the integer-pair (i, j). Each of these nodes is the neighbor of the other node. If a third node is connected to the other two nodes, it is said to be a common neighbor of the pair. Two data structures are used to represent the graph. *NodeList* is used to store the nodes in the graph. It is a two-dimensional data structure. Each horizontal list contains a number of nodes which form a clique. Initially, each node of the graph occupies a horizontal list. When several nodes are grouped into a clique, they are coalesced into the same horizontal list. *EdgeList* is used to store the edges in the graph. All the edges which have the same "left node" (the node with the smaller index) are linked in a horizontal list. The indices in a horizontal list are sorted in an increasing order. The "left nodes" of all the horizontal lists are vertically linked together. Again, they are sorted in an increasing order.

Given a sorted list of edges of a graph, the following algorithm partitions the nodes of a graph into disjoint clusters. Each cluster forms a clique.

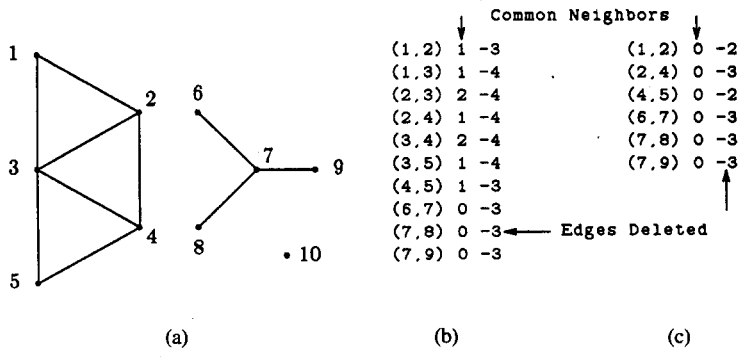


Fig. 8. Graph used by the example.

Algorithm 1: The Basic Clique-Partitioning Algorithm

1) Scan through the list of edges (*EdgeList*). For each (i, j), $i < j$, compute its number of common neighbors.
2) Pick the edge (p, q) which has the maximum number of common neighbors. Combine the lists of nodes headed by p and q.The smaller one of p and q is used as the head of the resulting clique.[12] Update the list of edges of G (as described in Algorithm 2). If the list of edges is empty, the graph partitioning is completed.
3) Assume p is the head of the resulting clique. Pick an edge which joins node p and other nodes and has the maximum number of common neighbors. Let the edge be (p, r), or (r, p) if r is smaller than p. Save r, or p if r is smaller than p. Update the list of edges of G (as described in Algorithm 2). If node p (or r if r is smaller than p) no longer appears in the *EdgeList*, go to Step 2 and start to collect the next cluster. Otherwise, repeat Step 3.

In Steps 2 and 3, if there is more than one pair having the maximum number of common neighbors, the numbers of edges which would be excluded are computed. The pair which excludes the least number of edges is selected. If more than one pair excludes the same number of edges, we choose one arbitrarily.

The number of common neighbors for each pair of connected nodes can be calculated by inspecting the list of edges. How is the number of edges to be excluded computed if a pair of nodes is grouped together? A node k which is connected to only one of i and j is no longer connected to the composite node (i, j). Thus the edge (i, k) or (j, k) must be deleted. A node k which is connected to both i and j is still connected to the composite node (i, j). Only one of edges (i, k) and (j, k) needs to be deleted. For consistency, each time one of these two edges needs to be deleted, the edge (j, k) is deleted. Therefore, the numbers of edges to be excluded can also be computed by inspecting the list of edges.

Once a pair of nodes is picked, the edge list needs to be updated in the following ways.

[12]Using the smaller node as the head of the resulting clique is just a matter of convenience. It does not influence the final result.

Algorithm 2: The Graph Updating Algorithm

1) Delete those edges which need to be deleted.
2) Recompute the numbers of common neighbors and the numbers of edges to be deleted for those pairs of connected nodes which remain in the list of edges.

An Illustrative Example

Let the graph depicted in Fig. 8(a) be given. The list of edges, the number of common neighbors and the edges to be deleted for each pair of connected nodes are depicted in Fig. 8(b). For example, the number of common neighbors and the number of edges to be excluded for (1, 2) can be computed in the following way. Node 3 is the only node which is connected to both nodes 1 and 2. The number of common neighbors for (1, 2) is thus one. If nodes 1 and 2 are grouped together, the edges (1, 2), (2, 3), and (2, 4) need to be deleted. Therefore, the number of edges to be excluded is three.

As indicated in Fig. 8(b), the pairs of nodes (2, 3) and (3, 4) have the maximum number of common neighbors and exclude the same number of edges if either pair is combined. Let nodes 2 and 3 be the first pair to be grouped together. Nodes 1 and 4 are connected to both nodes 2 and 3. They are connected to the composite node in the reduced graph. Node 5 is only connected to one of these two nodes, therefore, it is not connected to the composite node in the reduced graph. The list of edges of the reduced graph is depicted in Fig. 8(c).

To reduce the graph in Fig. 8(c), the numbers of common neighbors of the edges which consist of the composite node are compared. Both the edges (1, 2) and (2, 4) have the same number of common neighbors. Choosing the edge (1, 2), the number of edges to be excluded from the graph is less than choosing the edge (2, 4). Therefore, the edge (1, 2) is selected. The composite node which contains the nodes 1, 2, and 3 is no longer connected to other nodes. They belong to a cluster.

Repeatedly applying the procedure to the reduced graph, the nodes in the original graph are partitioned into six clusters. They are {1, 2, 3}, {4, 5}, {6, 7}, {8}, {9}, and {10}.

As mentioned in Section IV, direct application of the basic clique-partitioning algorithm to the data-memory allocation problem does not generate good solutions. And,

the notion of divide-and-conquer as well as the notion of transitive property are introduced to overcome the problem of randomly selecting nodes to merge. The following paragraphs present the generalized clique-partitioning algorithm.

Let (p, q), (p, r) [or (r, p) if r is smaller than p], and (q, r) [or (r, q) if r is smaller than q] be three edges in G where p is smaller than q. As indicated in Algorithm 1, if p and q are combined, then the composite node is represented by the smaller node p. Furthermore, the edge (q, r) is deleted from G.

Let (p, r), (q, r), and (p, q) belong to three different categories, which are represented by i, j, and k. Assume the profits of grouping a pair of nodes in categories i, j, and k are ordered in an increasing manner. In addition, these three edges have the following form of transitive property (named the generalized transitive property). If the nodes p and q are grouped together, then nodes p and r can be included in a new category l, where l is the lower case of L. The edges in category l have a profit measure better than edges in category i. The algorithm is given below.

Algorithm 3: A Generalized Clique-Partitioning Algorithm

1) Scan through the list of edges of category k($\boldsymbol{Gk}$). For each (i, j), compute its number of common neighbors.
2) Pick the edge (p, q) which has the maximum number of common neighbors from $\boldsymbol{Gk}$.
3) Instead of directly applying Algorithm 2 to $\boldsymbol{G}$ and $\boldsymbol{Gk}$, update $\boldsymbol{G}$ and $\boldsymbol{Gk}$ in the following way. For each node r which is only connected to one of the nodes p and q in the graph $\boldsymbol{G}$, the edge is deleted from $\boldsymbol{G}$. If the edge is also an edge of category k, it is deleted from $\boldsymbol{Gk}$. For each node r which is connected to both p and q in $\boldsymbol{G}$, the edge (q, r) is deleted from $\boldsymbol{G}$. If this edge is contained in $\boldsymbol{Gk}$, it is also deleted. Assume that (p, r) is an edge of category i and (q, r) is in the list of edges for category j. Due to the combination or grouping of p and q, the edge (p, r) becomes an edge of category l. The category indentifier of (p, r) is changed to l. If the profit of combining pairs of nodes in category l is the same as or better than that of combining pairs of nodes in category k, the edge is included in $\boldsymbol{Gk}$. Meanwhile, the number of common neighbors and the number of edges to be excluded for each of the edges remaining in $\boldsymbol{Gk}$ are updated.

The subgraph in which pairs of nodes have the best profit measure is reduced first. Then the pairs of nodes having the next level of profit measure are collected and reduced. Repeatedly applying the procedure to the other subgraphs, the process is stopped when a subgraph of a specified category or the original graph $\boldsymbol{G}$ becomes empty.

The transitive properties defined in Sections V–VII assume that the category identifiers j, k, and l refer to the same category. This is actually a special case of the generalized transitive property. It is named the loose form transitive property.

Acknowledgment

Dr. Stephen W. Director, Dr. Dario Giuse, and Dr. Donald E. Thomas have provided many constructive remarks. Special thanks are due to Bill Birmingham for reading and polishing early drafts of this paper. Finally, the authors are grateful to the referees for many suggestions which have improved the clarity of the presentation.

References

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Reading, MA: Addison-Wesley, 1977.
[2] *The AM2900 Family Data Book*. Advanced Micro Devices, Inc. 1976.
[3] J. G. Augustson and J. Minker, "An analysis of some graph theoretical cluster techniques," *J. ACM* 17(4):571–588, Oct. 1970.
[4] M. R. Barbacci, "Automated Exploration of the Design Space for Register Transfer (RT) systems," Ph.D. dissertation, Carnegie-Mellon Univ., Nov. 1973.
[5] M. R. Barbacci, G. E. Barnes, R. G. Cattell, and D. P. Siewiorek, "The symbolic manipulation of computer descriptions: ISPS computer description language," Carnegie-Mellon Univ., 1979.
[6] C. Bron and J. Kerbosch. "Finding all cliques of an undirected graph—Algorithm 457," *Commun. ACM*, 16(9):575–577, September, 1973.
[7] J. Darringer and W. Joyner. "A new look at logic synthesis," in *17th IEEE Design Automation Conf. Proc.* p. 543–549, June 1980.
[8] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. E. Thomas, "A design methodology and computer aids for digital VLSI systems," *IEEE Trans. Circuits Syst.*, vol. CAS-28, p. 634–645, July 1981.
[9] L. J. Hafer and A. C. Parker, "Automated synthesis of digital hardware," *IEEE Trans. Computers*, vol. C-31, pp. 93–109, Feb. 1982.
[10] L. J. Hafer and A. C. Parker, "A formal method for the specification, analysis, and design of register-transfer level digital logic," *IEEE Trans. Computer-Aided Design*, vol. CAD-2, pp. 4–18, Jan. 1983.
[11] P. Marwedel, "The MIMOLA design system: Detailed description of the software system," in *16th Design Automation Conf. Proc.*, pp. 59–63, 1979.
[12] G. McClain, "Optimal design of central processor data paths," Ph.D. dissertation, Dep. Electrical and Computer Eng., University of Michigan, Apr. 1972.
[13] M. C. McFarland, "The VT: A database for automated digital design," Technical Report DRC-01-4-80, Design Research Center, Carnegie-Mellon University, December, 1978.
[14] G. D. Mulligan and D. G. Corneil, "Corrections to Bierstone's Algorithm for Generating Cliques," *J. ACM* 19(2):244–247, Apr. 1972.
[15] M. C. Paull and S. H. Unger, "Minimizing the number of states in incompletely specified sequential functions," *IRE Trans. Electronic Computers*, vol. EC-8, pp. 356–367, Sept. 1959.
[16] N. S. Prasad, "Architecture and Implementation of Large Scale IBM Computer Systems," Q.E.D. Information Sciences, Inc., Wellesley, MA, 1981.
[17] S. L. Rege, "Designing variable data format modules with cost-performance tradeoffs," Ph.D. dissertation, Carnegie-Mellon Univ., Aug. 1974.
[18] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice*. Englewood Cliffs, NJ: Prentice-Hall 1977.
[19] H. E. Shrobe, "The data path generator," in *Proc. Conf. Advanced Research in VLSI*, MIT, Cambridge, MA, Jan. 1982, pp. 175–181.
[20] E. A. Snow, "Automation of module set independent register-transfer level design," Ph.D. dissertation, Carnegie-Mellon University, Apr. 1978.
[21] *The TTL Data Book for Design Engineers*. Texas Instruments, Inc., 1981.
[22] D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan, and R. A. Walker, "Automatic data path synthesis," *IEEE Computer*, vol. 16, pp. 59–73, Dec. 1983.
[23] C. J. Tseng and D. P. Siewiorek, "The modeling and synthesis of bus systems," in *Proc. Eighteenth Design Automation Conf.*, pp. 471–478. June 1981.

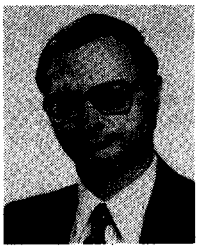
[24] C. J. Tseng and D. P. Siewiorek, "A note on the automated synthesis of bus style systems," Tech. Rep. Dep. of Electrical Eng., Carnegie-Mellon University, October, 1982.
[25] C. J. Tseng, "Automated synthesis of data paths in digital systems," Ph.D. dissertation, Dep. of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh, PA, Apr. 1984.
[26] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, "A new algorithm for generating all the maximal independent sets," *SIAM J. Computing*, vol. 6, no. 3, pp. 505–517, Sept. 1977.
[27] G. Zimmerman, "The MIMOLA design system: A computer aided digital processor design method," in *16th Design Automation Conf. Proc.*, pp. 53–58, 1979.

*

Chia-Jeng Tseng (S'82–M'85) received the B.S. degree in engineering science from National Cheng Kung University, Taiwan, Republic of China, and the Ph.D. degree in electrical engineering from Carnegie-Mellon University, Pittsburgh, PA.

Currently, he is a member of technical stall at AT&T Bell Laboratories in Murray Hill, NJ. His research interests include design automation, computer architecture, and analysis of algorithms.

Dr. Tseng is a member of the Association for Computing Machinery and Sigma Xi.



Daniel P. Siewiorek (S'67–M'72–SM'79–F'81) was born in Cleveland, OH, on June 2, 1946. He received the B.S. degree in electrical engineering (summa cum laude) from the University of Michigan, Ann Arbor, in 1968, and the M.S. and Ph.D. degrees in electrical engineering (minor in computer science) from Stanford University, Stanford, CA, in 1969 and 1972, respectively.

While completing his doctoral studies at Stanford University, he held a Research Assistantship in the Digital Systems Laboratory, working on fault tolerant computing. Dr. Siewiorek joined the Departments of Computer Science and Electrical Engineering at Carnegie-Mellon University, Pittsburgh, Pennsylvania in 1972. He has served as a consultant to several commercial and government organizations, including Digital Equipment Corporation, the Jet Propulsion Laboratory, the Naval Research Laboratory, Research Triangle Institute, United Technologies Corporation, AccuRay Corporation, ITT, NASA, Siemens Corporation, and Encore Computer Corporation.

While at Carnegie-Mellon University, he helped to initiate and guide the Cm* project that culminated in an operational 50 processor multi-microprocessor system. He also participated in the Army/Navy Military Computer Family (MCF) project to select a standard instruction set. The ISP (Instruction Set Processor) hardware description language was extensively developed to support the MCF program. He also directed the construction of C.vmp, a triply redundant fault tolerant computer and formulated a hardware design automation project. His current research interests include computer architecture, reliability modeling, fault-tolerant computing, modular design, and design automation. He has served as Associate Editor of the Computer System Department of the *Communications of the Association for Computing Machinery* and as Chairman of the IEEE Technical Committee on Fault Tolerant Computing.

Dr. Siewiorek was recognized with an Honorable Mention Award as Outstanding Young Electrical Engineer for 1977, given by the Eta Kappa Nu (National Electrical Engineering Honor Society). He was awarded the Frederick Emmons Terman Award by the American Society for Engineering Education in 1983 for outstanding young electrical engineering educator. He is a member of ACM, Tau Beta Pi, Eta Kappa Nu, and Sigma Xi.