Facet: A Procedure for the Automated Synthesis of Digital Systems[1]

Chia-Jeng Tseng and Daniel P. Siewiorek
Departments of Electrical Engineering and Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

In the past decade significant effort has been devoted to the development of methodologies for design at the register-transfer level. However, effective and versatile procedures are still not available. This paper presents an efficient procedure for the automated synthesis of data paths at the register-transfer level. The procedure minimizes the numbers of storage elements, data operators, and interconnection units. In addition, the procedure has the capability of exploring alternatives in the design space. In some preliminary experiments the procedure produced designs nearly identical to commercially produced designs.

1 Introduction

The Carnegie-Mellon University Design Automation (CMU-DA) system has been developed over the past six years [6]. Using the ISPS description [3] 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 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. The issues of specifying the initial operation sequences are described in Section 2. 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 numbers of storage elements, data operators, and interconnection units. These three minimization problems can be formulated into the clique-partitioning problem. The clique-partitioning problem will be detailed in Section 3. Section 4, 5, and 6 describe the formulations and present algorithms for generating solutions for each of these three problems. Exploring the design space is the topic of Section 7. Section 8 contains conclusions and suggestions for future work.

[1]This research has been supported by the National Science Foundation under Grant ENG 78-25755. The procedure is mainly concerned with the search of cliques in graphs. The similiarity in shape for a facet and a clique stimulated us to use "Facet" as the name of the procedure.

2 Specification of Initial Code Sequences

The input to the data-memory allocator is the value trace (VT) [7, 12]. A VT preserves all the data flow and control flow information in the original ISPS description. This section presents a procedure for specifying the initial code sequences.

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 VT basic block is first converted into a two-dimensional list of operation sequences. To preserve the maximum parallelism in a VT, a specific name is assigned to each value in the VT. Taking advantage of the data dependency relationships among different operations, the operation sequences are compacted in an as early as possible (AEAP) manner. The AEAP strategy (or the First-come First-serve strategy) has been applied in microcode compaction and proved to generate near-optimal solutions [5].

Using the AEAP strategy, an operation is moved forward to the horizontal list just behind the horizontal list where (at least) one of its operands is defined. Starting from the entry point of a basic block, the statements are processed one by one. Each time a triple statement (a statement which consists of one operation, one or two sources, and one destination) is processed, its operands are compared with the names defined in the previous line. If one of its operands is defined in the previous line, the operation is located there. Once the location of a statement is specified, its destination name is compared with the destination names of the other statements in the horizontal list. If some statement has the same destination name, then the original one is a redundant statement. The redundant statement can be eliminated. Table 1 is a VT-like data flow specification which will be used as a running example throughout the paper. Table 2 is the code sequence obtained by using the AEAP strategy. The statement enclosed by parentheses is a redundant statement and should be eliminated.

```
V3  = V1 + V2
V5  = V3 - V4
V7  = V3 * V6
V8  = V3 + V5
V9  = V1 + V7
V11 = V10 / V5
V12 = 100
V13 = V3
V12 = V1
V14 = V11 and V8
V15 = V12 or V9
V1  = V14
V2  = V15
```

Table 1: A VT-like Data Flow Specification

```
V3  = V1 + V2 ;      (V12 = 100) ;     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 2: Compacted Code Sequences

3 Some Procedures for Partitioning a Graph into Disjoint Cliques

Let G be a graph consisting of a finite number of nodes and a set of undirected edges connecting pairs of nodes. A non-empty 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** [11] 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 [2, 4, 8, 9, 10, 15]. A procedure which partitions a graph into a near minimum number of cliques is given in this section. The procedure uses the neighborhood property (as described in the next subsection) 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 [14]. The procedure has been applied to several 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 generated.

3.1 The Algorithm

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.

Algorithm 1:

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.[2] 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 Step 2 and Step 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.

Algorithm 2:

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.

3.2 An Illustrative Example

Let the graph depicted in Figure 1 (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 Figure 1 (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 wich 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.

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

As indicated in Figure 1 (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, it is not connected to the composite node in the reduced graph. The list of edges of the reduced graph is depicted in Figure 1 (c).

To reduce the graph in Figure 1 (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 }.

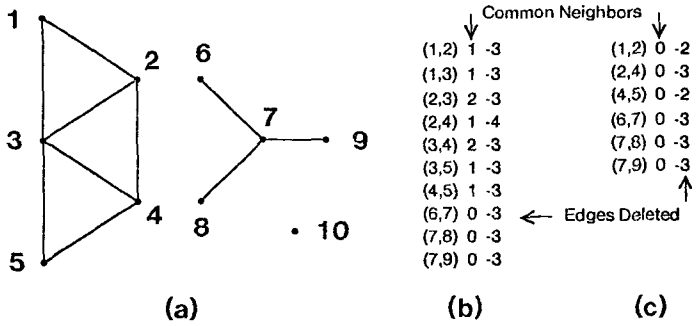

Figure 1: Graph Used by the Example

3.3 Modifying Algorithm 1 to Meet Demands of the Real World

Naive application of the clique-partitioning algorithm to the data-memory allocation problem does not generate good solutions. In this subsection two other notions are introduced to direct the application of the clique-partitioning procedure. One is divide and conquer. The other is the transitive property. The interpretation of these two notions on data-memory allocation will be detailed in Sections 4, 5, and 6.

Given a graph G, each edge of G represents some kind of relationship between the two nodes. When Algorithm 1 is applied, it is quite possible that several pairs of nodes have the same number of common neighbors and exclude the same number of edges if any pair is combined. It is also possible that the profit of grouping some set of nodes overrides 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 modifying clique-partitioning algorithm uses these subgraphs to direct the task of clique-partitioning and avoid grouping pairs of nodes randomly.

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 for category *k* (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 Gk.
3. Instead of directly applying Algorithm 2 to G and Gk, update G and Gk in the following way. For each node *r* which is only connected to one of the nodes *p* and *q* in the graph G, the edge is deleted from G. If the edge is also an edge of category *k*, it is deleted from Gk. For each node *r* which is connected to both *p* and *q* in G, the edge *(q,r)* is deleted from G. If this edge is contained in 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 identifier 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 Gk. Meanwhile, the number of common neighbors and the number of edges to be excluded for each of the edges remaining in 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 G becomes empty.

The transitive properties defined in Sections 4, 5, and 6 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.

Illustrative examples for the modified clique-partitioning algorithm are provided in Sections 4, 5, and 6.

4 Allocation of Storage Elements

As indicated in [14], 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.

4.1 Sufficient Conditions for Combining Two Variables

Given a set of variables, the problem 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.

4.2 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 actually 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, it further results 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. Algorithm 3 in Subsection 3.3 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 list of operation sequences 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. It is then possible that the code sequences can be further compacted. Therefore, the AEAP compaction is repeated once to make the final refinement for the operation sequences.

4.3 Lifetime Analysis

According to the previous discussion, it is concluded that the lifetime analysis is an essential process for the minimization of the number of storage elements. The problem of lifetime analysis is well understood in the area of compiler design. Details can be referred to [1].

4.4 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 list of code sequences are then traced and inspected. Unless the conditions given in Subsection 4.1 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.

4.5 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.

4.6 An Example

Let the compacted code sequences in Table 2 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 3.

Having derived the live/dead history of each variable, the compatible graph can be constructed. As mentioned before, a complete graph is first constructed. Using Table 3, 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 4.

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

Table 3: Result of Lifetime Analysis

(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)					

Table 4: The Compatible Variable-pairs

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 sequences in Table 2 can be refined into the form in Table 5.

```
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 5: Improved Code Sequences

5 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.

5.1 The Formulation

A data operator is called an isolated operator if it is exclusively assigned to a triple statement. What is the effect of grouping two isolated data operators into one 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 numbers of multiplexers and wired-broadcast trees may be increased or decreased. What is the effect of merging an isolated operator into an 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 [14]. 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 [14]. These sixteen cases can be classified into eight categories. They are listed below.

1. G8: The operations and the three pairs of variables are all the same.
2. G7: The operations are different but the three pairs of variables are the same.
3. G6: The operations and two pairs of variables are the same. The third pair of variables is different.
4. G5: Two pairs of the variables are the same. The operations and one pair of variables are different.
5. G4: The operations and one pair of variables are the same. The other two pairs of variables are different.
6. G3: One pair of the variables is the same. The operations and the other two pairs of variables are different.
7. G2: The operations are the same. All three pairs of variables are different.
8. G1: The operations and all three pairs of variables are different.

All of these subgraphs satisfy the generalized transitive property. 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 sequences 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.

5.2 An Example

Let each operation in Table 5 be assigned to a specific name. An assignment is given in Table 6. Using the code sequence and operator assignment, the compatible graph G is depicted in Table 7. The superscript integer at the right side of each edge is the category identifier of the edge.

```
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 and1 V3 ;   V2  = V12 or1 V2

Operator Identifiers:
          +1  -1  *1  +2  +3  /1  and1  or1

Indices:  1   2   3   4   5   6   7    8
```

Table 6: Assigning Operator Identifiers

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

Table 7: G: The Edges of the Original Compatible Graph

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 }.

6 Allocation of Interconnection Units

This section discusses the issues in the allocation of interconnection units.

6.1 Alignment of Operands

An operation may be either commutative or noncommutative. For those commutative operations, the designer 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. The operands of the commutative operations can be suitably aligned by comparing the operands with variables in these two sets.

6.2 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 [13]. There is no profit in combining two interconnection variables which originate from different sources and destine to different sinks. On the other hand, it is generally beneficial to group those interconnection variables which originate from the same source or destine 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 destine to the same sink, the edge which joins these two vaiables 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. Algorithm 3 in Subsection 3.3 can be applied.

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.

6.3 An Example

The example is based on the code sequence in Table 6 and the ALU's allocated in Section 5. Inspecting the operands of the operations associated with ALU2, it is found that the positions of the two operands of the statement "*V1 = V5 and$_1$ V3*" need to be flipped. It is changed into the form of "*V1 = V3 and$_1$ V5*". Using the indices in Table 8, the compatible graph in Table 9 (G) is constructed. In Table 9, 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.

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 8: Indices of 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)			

Table 9: G: List of edges which join combinable interconnection variables

In G1 nodes 14 and 16 have the maximum number of common neighbors. The 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 }. Figure 2 depicts the completed allocation of the data-memory part.

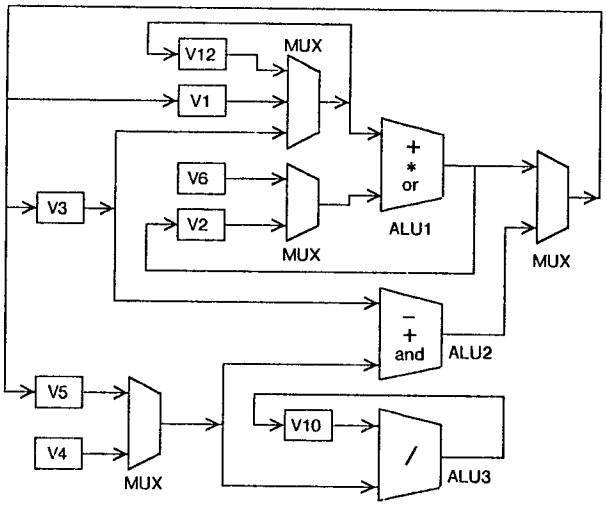

Figure 2: Data Paths of the Example

7 Exploring the Design Space

Design tradeoffs are fundamental to exploring a design space. Cost and speed are generally used to define the tradeoffs. If the serialization of some statements make the live periods of two variables satisfy the conditions given in Subsection 4.1, then these two variables can be assigned to the same register. Similiarly, two operators of the same kind, two ALU's, or two buses can be grouped together if all the parallelism associated with them has been eliminated. These tradeoffs are named the basic tradeoffs.

Let a base design which is directly translated from a VT and improved by the algorithms presented in Sections 4, 5, and 6 be given. Assume that there are *N* basic tradeoffs. The effect of any one, any two, any three, etc. of these tradeoffs can be considered. The ultimate case is to consider the effect of all *N* tradeoffs. To limit the search space, only a limited number of composite tradeoffs are considered.

By inspecting the improved code sequence and the allocated data paths in the previous sections, it is not difficult to find all the basic and composite tradeoffs for the example. We leave this as an exercise for the readers.

8 Conclusions and Future Work

This paper presents a procedure for data-memory allocation. The procedure has been programmed and in some preliminary experiments has produced designs nearly identical to commercially produced designs. Further research will focus on more extensive experimentation.

Acknowledgements

Comments and suggestions by Drs. Mario R. Barbacci, Stephen W. Director, and Donald E. Thomas are gratefully acknowledged.

References

[1] A. V. Aho and J. D. Ullman, "Principles of Compiler Design," Addison-Wesley, Reading, MA, 1977.

[2] J. G. Augustson and J. Minker, "An Analysis of Some Graph Theoretical Cluster Techniques," Journal of the ACM 17(4):571-588, October 1970.

[3] M. R. Barbacci, G. E. Barnes, R. G. Cattell, and D. P. Siewiorek, "The Symbolic Manipulation of Computer Descriptions: The ISPS Computer Description Language," Technical Report, Department of Computer Science, Carnegie-Mellon University, March 1978.

[4] C. Bron and J. Kerbosch, "Finding All Cliques of an Undirected Graph -- Algorithm 457", Communications of the ACM 16(9):575-577, September 1973.

[5] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett, "Some Experiments in Local Microcode Compaction for Horizontal Machines," IEEE Transactions on Computers, C-30(7), July 1981.

[6] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. E. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems," IEEE Transactions on Circuits and Systems, CAS-28(7), July 1981.

[7] M. McFarland, "The Value Trace: A Data Base for Automated Digital Design," Master Thesis, Department of Electrical Engineering, Carnegie-Mellon University, December 1978.

[8] J. W. Moon and L. Moser, "On Cliques in Graphs," Israel Journal of Mathematics (3):23-28, March 1965.

[9] G. D. Mulligan and D. G. Corneil, "Corrections to Bierstone's Algorithm for Generating Cliques," Journal of the ACM 19(2):244-247, April 1972.

[10] M. C. Paull and S. H. Unger, "Minimizing the Number of States in Incompletely Specified Sequential Switching Functions," IRE Transactions on Electronic Computers (EC-8):356-367, September 1959.

[11] E. M. Reingold, J. Nievergelt, and N. Deo, "Combinatorial Algorithms: Theory and Practice," Prentice-Hall, 1977.

[12] E. A. Snow, "Automation of Module Set Independent Register-Transfer Design," Ph.D. Thesis, Department of Electrical Engineering, Carnegie-Mellon University, April 1978.

[13] C. J. Tseng and D. P. Siewiorek, "The Modeling and Synthesis of Bus Systems," Proceedings of the Eighteenth Design Automation Conference, pages 471-478, ACM SIGDA and IEEE Computer Society DATC, June 1981.

[14] C. J. Tseng and D. P. Siewiorek, "A Note on the Automated Synthesis of Bus Style Systems," Technical Report, Department of Electrical Engineering, October 1982.

[15] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, "A New Algorithm for Generating All the Maximal Independent Sets," SIAM Journal of Computing 6(3):505-517, September 1977.