GLASS: a Graph-Theoretical Approach for Global Binding

Rosa M. Badia and Jordi Cortadella *

Polythecnic University of Catalonia, Dept. of Computer Architecture,
Campus Nord, Mòdul D6, Gran Capità s/n, Barcelona-E08071

Module binding is the behavioral synthesis step that derives a structural representation of the datapath. It involves three tasks: functional unit, storage unit, and interconnection unit binding. This paper presents *GLASS*, a graph-theoretical formulation of the binding problem that performs all binding tasks simultaneously. Operations, variables, and data transfers are represented as vertices of the same compatibility graph and binding decisions are interlaced among different types of hardware modules (ALUs, registers, buses, ...) in order to minimize the estimated datapath area. The solutions generated by *GLASS* are superior to those obtained by existing graph-theoretical P-time techniques.

1. INTRODUCTION

Behavioral synthesis aims at the generation of a structural description of a circuit from a behavioral description [1] by mapping high-level objects (operations, variables) into register transfer components (functional, storage and interconnection units).

Scheduling and *allocation* are two of the most important tasks in the synthesis of circuits from behavioral specifications. Scheduling distributes operation execution along time by assigning each operation to a control-step in synchronous systems, or by defining a partial execution order in asynchronous systems [2]. Allocation determines the hardware units that must be used in the circuit. Scheduling and allocation are interdependent tasks. In some synthesis systems, such as Facet [3], scheduling is performed before allocation, others perform allocation before scheduling, such as Caddy/DSL [4], while others combine the execution of both tasks simultaneously, such as Slicer/Splicer [5] or [8].

Allocation can be furthermore divided into two main tasks [6]: unit *selection* and unit *binding*. The former determines the number and type of modules of the circuit. The latter maps the operations and variables into instances of functional, storage and interconnection units. Again, binding can be decomposed into three interdependent tasks that must be executed in some order. Most systems perform functional-unit binding before storage binding, while interconnections are defined last. In any case, the solution found by each task depends on the outcome of the previous ones.

This papers presents *GLASS*, a novel approach to solve the three binding tasks (functional-unit, storage and interconnection binding) *simultaneously*. The proposed algorithm is based on the problem of graph clique-partitioning and can be executed in polynomial time. The major contribution of this approach is the representation of operations, variables and data transfers as vertices of the same compatibility graph. Binding decisions are interlaced among different types of units, thus attempting to reach a final configuration close to the global minimum.

The solutions obtained by *GLASS* improve the quality of the best known solutions obtained by the existing polynomial-time techniques, and they are comparable, although slightly worse, to those obtained by CPU-intensive techniques based on iterative improvements or heuristic-search approaches.

Section 2 presents previous work in the area of binding algorithms. Section 3 describes how the compatiblitity graph is built and edge weights calculated. Section 4 reviews the weight-directed clique partitioning algorithm. Section 5 shows some results obtained with the algorithm. Conclusions and future work are presented in section 6.

*Work funded by CYCIT TIC 91-1036 and ACiD-WG (Esprit 7225)

Table 1
Results for the Elliptic Filter [19]

	#mult	#add	#registers	#mux 2:1	#bus	#bus inputs
Solution 1	1	2	11	20	7	28
Solution 2	1	3	12	9	4	17

2. PREVIOUS WORK

The existing algorithms for unit binding can be classified according to three main tendencies:

control-step-based algorithms: All binding tasks are executed for each control step before processing the next one.

global algorithms: All binding tasks are executed separately, but globally with respect to the schedule.

iterative improvement algorithms: Given an already synthesized datapath, its quality is improved by successive reallocation of modules.

The first two groups include all the constructive algorithms (datapath is built from scratch). Greedy constructive approaches [7], linear programming approaches [8, 9] , and branch and bound [5, 10] belong to the first group. The second group covers all the graph-theoretical formulations such as clique partitioning [3], left-edge algorithm [11], and weighted bipartite-matching algorithm [12]. Iterative improvement algorithms can achieve high quality solutions at the expense of using CPU-intensive techniques, such as "hill-climbing" [13] or simulated evolution [14].

From all the existing techniques, only greedy constructive methods and graph-theoretical formulations can afford solving the unit binding problem in a moderate CPU time for larger examples. Moreover, control-step based approaches are only focused to synchronous schedules, while one of our main interests is the synthesis of asynchronous circuits [15].

2.1. GLASS overview

Graph-theoretical formulations of the binding problem often use the same formulation for the three tasks (functional unit, storage and interconnect), although they are solved separately. *GLASS* uses the graph clique partitioning algorithm, but solves the three binding tasks simultaneously. The main contribution of *GLASS* is the representation of all the functional units, registers and interconnect units as vertices of the same compatibility graph. Unlike previous approaches, binding decisions such as merging two operations into one functional unit or two variables into one register can be interleaved and taken at any time, independently of the type of hardware module being shared. In doing so, decisions that e.g. would give good results for functional unit binding but would highly increase interconnectivity would be avoided.

To illustrate the validity of this approach, we present a simple example (see table 1). The first solution was obtained by restricting *GLASS* to bind first all the operations to functional units, then variables to registers, and finally data transfers to buses. The second solution was obtained by allowing *GLASS* to interleave all the binding decisions. Although the first solution is superior in terms of functional units and registers, it gives an inferior result in terms of interconnection units. Considering all the units simultaneously (solution 2) a better result is obtained.

3. SCHEDULED CDFG AND COMPATIBILITY GRAPH (CG)

3.1. Definitions

Unit binding starts from a scheduled Control Data Flow Graph (CDFG), $G(V, E)$, in which scheduling and unit selection have already been performed. Vertices represent operations, variables or data-transfers and edges represent dependencies. For each vertex $v \in V$ we will use the following terminology:

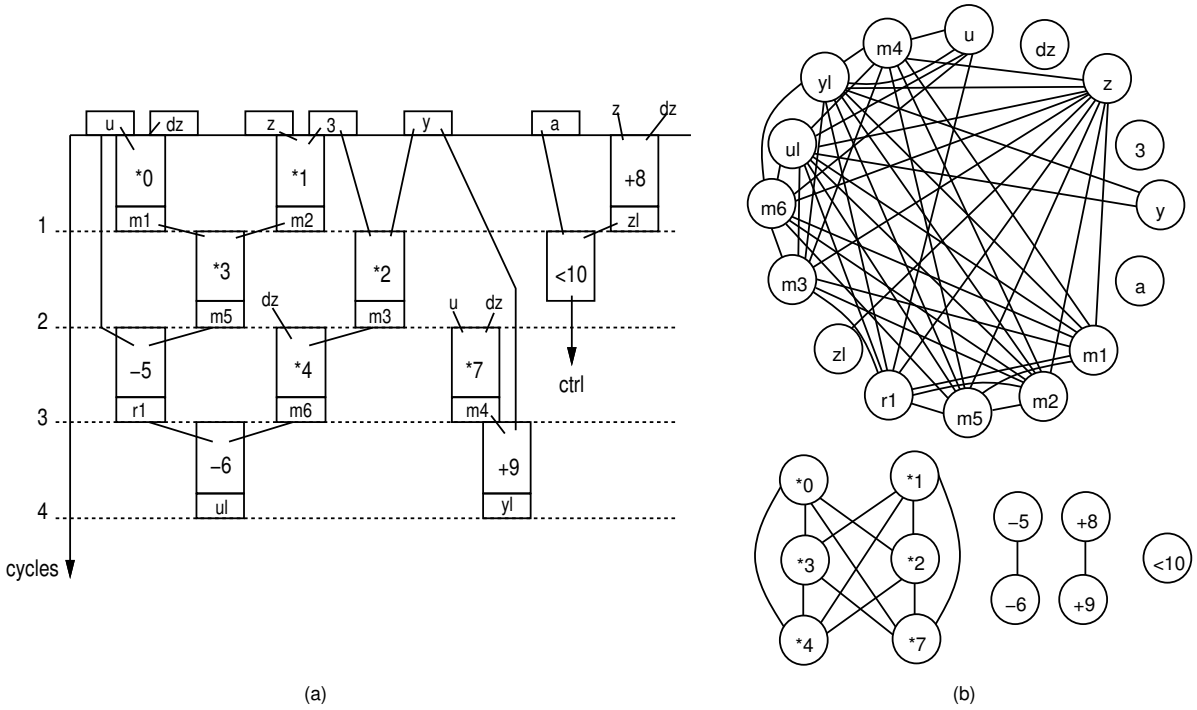

Figure 1. (a) Scheduled CDFG for the differential equation; (b) Compatibility graph for the same example

v_t	:	type of vertex: operation (`OP`), variable (`VAR`), or data transfer (`DT`)
v_f	:	type of functional unit allocated to v
$pred(v)$	=	$(pred_1(v), \cdots, pred_n(v))$ is the list of predecessors of node v ($\mid pred(v) \mid = n$)
$succ(v)$	=	list of successors of node v

We will assume that the elements of $pred(v)$ (input operands of v) are ordered and we will denote by $pred_i(v)$ the i-th predecessor of v ($i = 1..|pred(v)|$). If an operation is commutative the order of the predecessors can be exchanged. In the case of variables and data transfers, $|pred(v)| = 1$.

The compatibility graph (CG) used for clique partitioning, $\hat{G}(\hat{V}, \hat{E})$, initially has the same number of vertices as the scheduled CDFG. The edges in $\hat{E}$ represent the resource compatibility between pairs of vertices, e.g. two operations are compatible if they have been scheduled for different control steps and they have been allocated to the same type of functional unit.

A lifetime analysis must be performed to evaluate the compatibility of the variables. Two variables are compatible (can be assigned to the same register) if their lifetimes do not overlap. Similarly to operations, two data transfers are compatible if they are executed at different control steps.

Hereafter u, v, and e will be used to denote vertices and edges of the scheduled CDFG ($u, v \in V$ and $e \in E$) and $\hat{u}, \hat{v}$, and $\hat{e}$ to denote vertices and edges of the CG ($\hat{u}, \hat{v} \in \hat{V}$ and $\hat{e} \in \hat{E}$). Figure 1 shows a compatibility graph.

A *weight function*, $W : \hat{E} \rightarrow \mathbb{R}$, is also defined for the edges of the CG. For each edge $\hat{e} = (\hat{u}, \hat{v})$, $W(\hat{e})$ is defined according to the interconnection affinity of $\hat{u}$ and $\hat{v}$ [2].

Given two vertices $u, v \in V$, $comp(u, v)$ is de-

[2] The interconnection affinity of two nodes denotes the gain in interconnection cost that results if both nodes are assigned to the same hardware unit

fined as:

$$comp(u,v) = \begin{cases} true & \text{if } u \text{ and } v \text{ are compatible} \\ false & \text{otherwise} \end{cases}$$

Throughout the execution of the clique partioning algorithm, pairs of connected vertices of the CG are merged into one vertex. Therefore, the CG is dynamically modified and each node of the CG represents several nodes from the scheduled CDFG. The *merged set* of vertex $\hat{v}$, $Mset(\hat{v})$, is the set of vertices $\{u_1, \ldots, u_n\}$, $u_i \in V$, that have been merged into vertex $\hat{v}$. Initially $Mset(\hat{u}) = \{u\}$. The function *merged* is thus defined as follows:

$$merged(u,v) = \begin{cases} true & \exists \hat{w} \in \hat{V} \text{ such that} \\ & u, v \in Mset(\hat{w}) \\ false & \text{otherwise} \end{cases}$$

Note that if $u = v$ then $merged(u,v) = true$. If two connected vertices $\hat{u}, \hat{v}$ are merged into node $\hat{w}$, then

$$Mset(\hat{w}) = Mset(\hat{u}) \cup Mset(\hat{v}) \tag{1}$$

3.2. CG operations

3.2.1. Vertex Merging

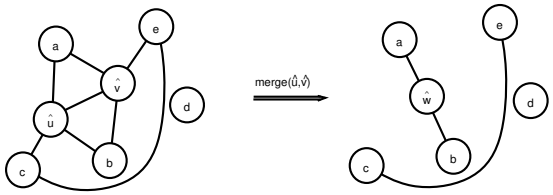

Figure 2. Example of function $merge(\hat{u}, \hat{v})$

As mentioned before, pairs of connected vertices $(\hat{u}, \hat{v})$ of the CG are merged into a new vertex $\hat{w}$. The function $merge(\hat{u}, \hat{v})$ is next described:

merge($\hat{u}$,$\hat{v}$){
  add vertex $\hat{w}$ to $\hat{V}$;
  foreach $\hat{z} \in \hat{V}$ s.t. $(\hat{u}, \hat{z}), (\hat{v}, \hat{z}) \in \hat{E}$ **do**
    add edge $(\hat{w}, \hat{z})$;
  foreach $(\hat{u}, \hat{z}) \in \hat{E}$ **do** *remove edge $(\hat{u}, \hat{z})$;*
  { This also removes edge $(\hat{u}, \hat{v})$ }
  foreach $(\hat{v}, \hat{z}) \in \hat{E}$ **do** *remove edge $(\hat{v}, \hat{z})$;*
  remove vertex $\hat{u}$;
  remove vertex $\hat{v}$;
}

New edges are added between vertex $\hat{w}$ and those vertices that are connected with both $\hat{u}$ and $\hat{v}$. Finally, vertices $\hat{u}$ and $\hat{v}$ and all edges connecting them to other vertices are removed. Figure 2 depicts an example of vertex merging.

3.2.2. Edge Weight Calculation

Next, the calculation of edge weights is described. Given an edge $\hat{e} = (\hat{u}, \hat{v}) \in \hat{E}$, $W(\hat{e})$ is calculated as follows:

$$W(\hat{e}) = \alpha_t(W_{in}(\hat{e}) + W_{out}(\hat{e})) \tag{2}$$

where $W_{in}(\hat{e})$ and $W_{out}(\hat{e})$ are the partial weights of $\hat{e}$ corresponding to the interconnetivity affinity of $\hat{u}$ and $\hat{v}$ inputs and outputs respectively. α_t is an input factor that can take different values according to the type of vertices represented by $\hat{u}$ and $\hat{v}$ (`OP`, `VAR`, or `DT`). For example α_{OP} if the factor applied to those edges connecting vertices that represent operations. This factor represents the gain in area obtained when two operations share a single functional unit. Giving high values to α_{OP}, the edges connecting vertices that represent operations will be given higher weights, and therefore will be merged before than others. That allows us to simulate the execution of the binding tasks separately.

$W_{in}(\hat{e})$ and $W_{out}(\hat{e})$ are calculated as follows. Let $\hat{e} = (\hat{u}, \hat{v})$. For each pair of nodes $u \in Mset(\hat{u}), v \in Mset(\hat{u})$, and for each i-th predecessor of u and v the relationship between $pred_i(u)$ and $pred_i(v)$ it is evaluated. If $pred_i(u)$,$pred_i(v)$ are merged, a weight factor W_{m_in} is added to the total weight. If $pred_i(u)$,$pred_i(v)$ are compatible, W_{c_in} is added to the total weight.

$W_{in}(\hat{e})$ {
  { Let $\hat{e} = (\hat{u}, \hat{v})$ }
  $W = 0$;
  foreach $u \in Mset(\hat{u})$ **do**
    foreach $v \in Mset(\hat{v})$ **do**
      for $i = 1$ **to** $|pred(u)|$ **do**
        if $merged(pred_i(u), pred_i(v))$ **then**
          $W := W + \frac{W_{m_in}}{|pred(u)|}$;
        else if $comp(pred_i(u), pred_i(v))$ **then**
          $W := W + \frac{W_{c_in}}{|pred(u)|}$;
      enddo
    enddo
  enddo
  $W := \frac{W}{|Mset(\hat{u})| \cdot |Mset(\hat{v})|}$;
  return(W);
}

$W_{out}(\hat{e})$ {
  { Let $\hat{e} = (\hat{u}, \hat{v})$ }
  $W = 0$;
  foreach $u \in Mset(\hat{u})$ **do**
    foreach $v \in Mset(\hat{v})$ **do**
      $temp := 0$;
      foreach $u' \in succ(u)$ **do**
        foreach $v' \in succ(v)$ **do**
          if $merged(u', v')$ **then**
            $temp := temp + W_{m_out}$;
          else if $comp(u', v')$ **then**
            $temp := temp + W_{c_out}$;
        enddo
      enddo
      $W := W + \frac{temp}{|succ(u)| \cdot |succ(v)|}$;
    enddo
  enddo
  $W := \frac{W}{|Mset(\hat{u})| \cdot |Mset(\hat{v})|}$;
  return(W);
}

W_{m_in} and W_{c_in} are the weight factors used when a predecessor (input) of a vertex $u \in Mset(\hat{u})$ and a predecessor of a vertex $v \in Mset(\hat{v})$ are *merged* or *compatible*. When the predecessors of u and v are *merged* no extra interconnection will be required for this input if u and v are also merged. When the predecessors of u and v are *compatible* it is possible (in case the predecessors are sometime merged) that no extra interconnection will be needed for this input if u and v are merged. W_{m_in} must be greater than W_{c_in} in order to give priority to merged predecessors. We have given to these factors the values $W_{m_in} = 3$ and $W_{c_in} = 1$. Giving higher values to W_{m_in} the algorithm becomes greedier. On the other hand, if $W_{m_in} = W_{c_in}$ the information of the CG is not completely exploited.

W_{m_out} and W_{c_out} are the same weight factors as W_{m_in} and W_{c_in} applied to the outputs. These factors have been set to the values $W_{m_out} = 2$ and $W_{c_out} = 1$.

3.3. Commutativity

Some operations, like addition or multiplication are commutative. The algorithm exploits the commutativity in order to minimize interconnection. The following process is applied to evaluate the weight of edges connecting vertices with commutativity: First it is calculated the edge weight without swapping the operands. Next it is evaluated the weigth swapping the operands of one vertex. It is selected the maximum of both weights and information about which configuration has the maximum weight is stored.

4. ALGORITHM

Solving the binding problem is equivalent to finding a *clique partitioning* of the global compatibility graph. A *clique* is a set of vertices that form a complete subgraph, i.e., a set of vertices where every pair of distinct vertices is adjacent. Minimum clique partitioning has been proven to be NP-complete [16]. In [3] a polynomial-time heuristic that constructs a near-minimum clique partitioning was proposed. This heuristic was modified in [17] by assigning weights to the edges of the compatibility graph in order to optimize interconnect area.

The algorithm selects two vertices to be merged according to the weight of the edge that connect them and to the number of common neighbors ν and the number of edges to be deleted η , which

have been defined in [6] as:

- For all edges $\hat{e} = (\hat{v}, \hat{w})$, $e \in \hat{V}$ the number of common neighbors ν is defined as:

  $$\nu(\hat{e}) = \mid \{\hat{v}' \in \hat{V} : (\hat{v}', \hat{v}) \in \hat{E} \wedge (\hat{v}', \hat{w}) \in \hat{E}\} \mid$$

- and the number of edges η to be deleted if the vertices $\hat{v}$ and $\hat{w}$ are merged into vertex $\hat{v}$:

$$\eta(\hat{e}) = \mid \{(\hat{v}, \hat{v}') \in \hat{E} : (\hat{w}, \hat{v}') \notin \hat{E}\} \cup \{(\hat{w}, \hat{v}') \in \hat{E}\}$$

The weight-directed clique partitioning algorithm is next described:

```
weight-directed clique partitioning(CG){
  for all ê ∈ Ê do calculate W(ê);
  repeat
     τ ← max_{ê∈Ê}(W(ê))
    for all ê ∈ Ê_red do calculate ν(ê) and η(ê);
    repeat
      ê_best ← select_vertex(τ);
      merge(v̂, ŵ);
      for all ê ∈ Ê do calculate W(ê);
      for all ê ∈ {ê ∈ Ê : W(ê) ≥ τ} do
        calculate ν(ê) and η(ê);
      while {ê = (v̂, v̂'), ê ∈ Ê : W(ê) ≥ τ} ≠ ∅ do
        ê_best ← select_adjacent_vertex(τ, v);
        merge(v̂, ŵ);
        for all ê ∈ Ê do calculate W(ê);
        for all ê ∈ {ê ∈ Ê : W(ê) ≥ τ} do
            calculate ν(ê) and η(ê);
      enddo
    until {ê ∈ Ê : W(ê) ≥ τ} = ∅;
  until E = ∅;
}
```

The outer loop selects the edge $\hat{e}_{best} = (\hat{v}, \hat{w})$ which has maximum weight (τ) and maximum $\nu(\hat{e})$ (if several vertices have the same $\nu(\hat{e})$ the one with minimum $\eta(\hat{e})$ is selected). The vertices $\hat{v}, \hat{w}$ are merged into vertex $\hat{v}$ and the edges that are no longer valid are deleted. The values $W(\hat{e})$, $\nu(\hat{e})$ and $\eta(\hat{e})$ are updated for the affected edges. The inner loop selects in each iteration one vertex adjacent to $\hat{v}$ with maximum weight and merges it into $\hat{v}$. The process is repeated until all edges have been removed from the compatibility graph.

5. Results

This section presents some results obtained with known benchmarks. Results obtained with simple examples like the *Differential equation* [18] or the FACET example [3] are comparable or better than the obtained by other systems. These results are not reported due to the small size of the problems. Results obtained for the *Fifth-order Wave Elliptic Filter* [19] are compared with results obtained from HAL [18], SPLICER [20], MABAL [7], ARYL/LYRA [12], SALSA [13] , ELF [14], SPAID [21] and SCHALLOC [22].

Table 2 reports the results obtained using 1 multiplier (two cycles of latency) and 2 adders in a 21-cycle schedule. *GLASS* improves the results obtained by previous P-time approaches and are comparable (although worse) to those obtained by CPU-intensive techniques (SALSA, ELF and SPLICER). The first solution uses 10 registers and 21 2:1 multiplexors. The second uses 11 registers and 19 2:1 multiplexors. We must emphasize the short CPU-time required by *GLASS*.

Table 3 depicts the results obtained using 2 adders and 1 pipelined multiplier with a 19-cycle scchedule. Tables 4 and 5 present the results obtained when a bus-based architecture is used. For both tables , *GLASS* improves the results obtained by other approaches.

6. CONCLUSIONS

This paper has presented a novel graph-theoretical approach to the binding problem where all tasks are simultaneously performed. Operations, variables and data-transfers are represented in a single compatibility graph where pairs of vertices can be merged at any moment, independently of which type of operation represent.

Although some results obtained are inferior to those obtained with CPU-intensive techniques, it improves all previous P-time methods.

The compact representation of the unique compatibility graph looks like a promising starting point for CPU-intensive techniques. The evaluation of these techniques with this representation is left as future work.

Table 2
Results for the Elliptic Filter (2 adders, 1 multiplier, 21 cycles). The table represents: number of registers, no. of equivalent 2:1 multiplexors, no. of mux inputs, total no. of multiplexors, CPU time, and technique used. † It does not use coefficient ROM

	#registers	#mux 2:1	#mux in	#mux	CPU time	Technique
HAL	12	–	31	–	2-8 min	Clique Partitioning
SPLICER†	–	34	43	9	55 sec	Branch & Bound
MABAL	10	30	42	12	10.52 sec	Heuristic Algorithm
ARYL	10	22	30	8	0.65 sec	Bipartite Weighted Matching
LYRA	10	23	31	8	3.38 sec	Bipartite Weighted Matching
SALSA	10	15	–	–	8-10 min	Iterative Improvement
SALSA	11	16	–	–	8-10 min	Iterative Improvement
ELF	11	16	24	8	1 hour	Simulated Evolution
GLASS	10	21	27	6	0.53 sec	Clique Partitioning
GLASS	11	19	25	6	0.48 sec	Clique Partitioning

Table 3
Results for the Elliptic Filter (2 adders, 1 pipelined multiplier, 19 cycles).

	#registers	#mux 2:1	#mux in	#mux	CPU time	Technique
MABAL	16	32	45	13	7.48 sec	Heuristic Algorithm
SALSA	10	19	–	–	8-10 min	Iterative Improvement
SALSA	11	16	–	–	8-10 min	Iterative Improvement
SALSA	12	18	–	–	8-10 min	Iterative Improvement
ELF	11	19	30	11	1 hour	Simulated Evolution
GLASS	11	23	29	6	0.43 sec	Clique Partitioning
GLASS	12	22	31	9	0.45 sec	Clique Partitioning

REFERENCES

1. M.C. McFarland, A.C. Parker, and R. Camposano. The High-Level Synthesis of Digital Systems. In *Proc. of the IEEE*, vol. 78, pages 201-318, Feb. 1990
2. R.M. Badia and J. Cortadella. High-Level Synthesis of Asynchronous Systems: Scheduling and Process Synchronization. In *Proc. of EDAC-93*, pages 70-74, 1993.
3. C-J. Tseng and D.P. Siewiorek. Automated synthesis of data-paths in digital systems. *IEEE Transactions on Computer-Aided Design*, vol. CAD-5, no. 3, pages 379-395, 1986.
4. R. Camposano and W. Rosenstiel. Synthesizing Circuits from Behavioral Descriptions. *IEEE Transactions on Computer-Aided Design*, vol. 8, paages 171-180, Feb.. 1989.
5. B.M. Pangrle and D.D. Gajski. Design tools for intelligent silicon compilation. *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, pages 1098-1112, 1987.
6. D.D. Gajski, N.D. Dutt, A.C-H Wu, and S.Y-L Lin. *High-Level Synthesis. Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.
7. MABAL: A Software Package for Module and Bus Allocation. *International Journal of Computer Aided VLSI Design*, pages 419-436, 1990.
8. M. Balakrishnan and P. Marwedel. Integrated Scheduling and Binding: a synthesis approach for design space exploration. In *Proc. of the 26th ACM/IEEE Design Automation Conference*, pages 68-74, 1989.
9. M. Rim, R. Jain, and R. De Leone. Optimal Allocation and Binding in High-Level Synthesis. In *Proc. 29th ACM/IEEE Design*

Table 4
Results for the Elliptic Filter (2 adders, 1 multiplier, 21 cycles) using a bus-based architecture. The table represents: number of registers, no. of buses, no. of bus inputs, no. of equivalent 2:1 multiplexors, no. of mux inputs, CPU time, and technique used.

	#registers	#bus	#bus in	#mux 2:1	#mux in	CPU time	Technique
MABAL	10	2	11	22	32	14.46 sec	Heuristic Algorithm
SCHALLOC	13	9	–	40	53	179 sec	Branch & Bound
GLASS	12	5	19	8	11	4.43 sec	Clique Partitioning
GLASS	12	3	10	13	19	4.45 sec	Clique Partitioning

Table 5
Results for the Elliptic Filter (2 adders, 2 multipliers, 18 cycles) using a bus-based architecture.

	#registers	#bus	#bus in	#mux 2:1	#mux in	CPU time	Technique
MABAL	11	2	14	30	43	16.70 sec	Heuristic Algorithm
SPAID	19	6	18	–	26		Best First Search
GLASS	12	3	11	16	24	4.87 sec	Clique Partitioning

Automation Conference, pages 120-123,June 1992.

10. J. Septién, D. Mozos, F. Tirado, R. Hermida, and M. Fernández. Heuristics for Branch-and-Bound Global Allocation. In *Proc. EURO-DAC 92*, pages 334-340, Sept. 1992.
11. F.J. Kurdahi and A.C. Parker. REAL: A Program for Register Allocation. In Proc. of the 24th ACM/IEEE Design Automation Conference, pages 210-215, 1987.
12. C-Y Huang, Y-S Chen, Y-L Lin, and Y-C Hsu. Data Path Allocation Based on Bipartite Weighted Matching. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 499-504, 1990.
13. G. Krishnamoorthy and J.A. Nestor. Data Path Allocation Using an Extended Binding Model. In *Proc. 29th ACM/IEEE Design Automation Conference*, pages 279-284,1992.
14. T.A. Ly, W.L. Elwood, and E.F. Girczyc. A Generalized Interconnect Model for Data Path Synthesis. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 168-173, 1990.
15. J. Cortadella and R.M. Badia. An Asynchronous Architeccture Model for Behavioral Synthesis. In *Proc. EDAC-92*, pages 307-311, March 1992.
16. M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman and Company, New York, 1979.
17. P.G. Paulin and J.P. Knight. Scheduling and Binding Algorithms for High-Level Synthesis. In *Proc. 26th ACM/IEEE Design Automation Conference*, pages 1-6, 1989.
18. P.G. Paulin and J.P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Trans. on CAD*, pages 661-679, Jun 1989.
19. P. Dewilde and E. Deprettere and R. Nouta. Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms. In *VLSI and Modern Signal Processing*, pages 258-264, ed. T. Kailath, 1985.
20. B.M. Pangrle. SPLICER: A Heuristic Approach to Connectivity Binding. In *Proc. 25th ACM/IEEE Design Automation Conference*, pages 536-541,1988.
21. B.S. Haroun and M.I. Elmasry. Automatic Synthesis of a Multi-Bus Architecture for DSP. In *Proc. of ICCAD-88*, pages 44-47, 1988.
22. N. Berry and B.M. Pangrle. SCHALLOC: an Algorithm for Simultaneous Scheduling and Connectivity Binding in a Data Path Synthesis System. In *Proc. of EDAC-90*, pages 78-82, 1990.