87-59

Module Selection for Pipelined Synthesis

Rajiv Jain, Alice Parker
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-0781

Nohbyung Park
Department of Electrical Engineering
University of California
Irvine, CA 92717

Abstract

Module selection is one of the many functions which have to be performed during behavioral synthesis of pipelined designs. Module selection is the process of choosing the types of modules (e.g. carry-look-ahead adder) to implement each operation (e.g. addition). In this paper, we give a limited solution to the module selection problem for pipelined designs. A model for estimating area-time tradeoffs [3] for pipelined designs is used to formulate the module selection problem, and an overview of the solution technique is given. Complexities introduced by non-optimal designs and user constraints are also addressed. The results have been validated using designs generated by an automated pipeline synthesis program.

1 Introduction

There are a number of tasks to be performed to carry out data path synthesis, including scheduling, operator allocation and module selection. Ideally, simultaneous solution to all these tasks is required to guarantee that optimal designs will be found. However, since each of the tasks in isolation is probably NP-complete, a simultaneous solution to all the tasks to achieve near-optimal designs seems to be computationally infeasible. In practice, the tasks must be ordered to reduce the complexity of the problem.

In the past, selection of module styles or types had been viewed as a function to be performed after data path scheduling and allocation had been completed. The synthesis programs determined how many of each operator were required, but the specific operator implementation (e.g. carry-look-ahead vs. ripple-carry) was not decided until after scheduling and operator allocation were completed. This assumption did not restrict the design space severely, since most synthesis programs assumed that each operation took one time step, regardless of the function being performed or implementation of that function. Thus, scheduling of operations could proceed independently of module selection.

This research was supported by the Semiconductor Research Corporation Contract 86-01075, and by the Department of Navy, the Department of Army and the Department of Air Force Contract No. N00039-87-C-0194.

In the last several years, however, data path synthesis programs have removed the simplifying assumption about operations taking a single time step (see [1,9,10,11]). Today, state-of-the-art programs schedule multiple operations chained together into single time steps, and achieve designs which are faster and more area efficient. However, in order to perform this more complex scheduling, some information about actual module delays is now required. In addition, many synthesis programs use more sophisticated cost measures during data path synthesis, such as chip area, which requires actual cell area and interconnect costs.

In order to provide the information described above to the scheduler, module selection should be performed prior to scheduling. BUD [9] was one of the earliest synthesis programs to accomplish this, and uses a straightforward heuristic which selects the module with minimum $area - time^n$ product, where n is a variable set by the user.

This paper presents a rigorous technique for module style selection for pipelined designs. This technique is based on the ability to predict the location in the design space of the area-time tradeoff curve for a given design and given module set. This predictive ability, in turn, is based on the straightforward optimization criteria for digital design that all modules are utilized as many cycles as possible.

Many future systems will use module generation to create modules which meet certain area or performance constraints. The techniques presented here can also be applied not only to *select* existing modules but also to *specify* the required characteristics of modules to be generated.

1.1 Motivation

The savings in processing time using module selection prior to synthesis is a major motivation for this work. For example, Sehwa [10], a pipeline data path synthesis program, is a part of the USC ADAM (Advanced Design AutoMation) system. The input to Sehwa is a data flow graph (in a data flow graph, a node is an operation and arcs are values) and a module set. Sehwa determines the quantity of each type of operator required and the scheduling of the data flow graph. These tasks are performed using a single module set. If the user wishes to explore a different portion of the design space, then the design process is repeated using a new module set. With automated module selection, this

repetitive processing can be eliminated.

An example data flow graph is shown in Figure 1. The design space for this data flow graph using three different module sets is shown in Figure 2. There are three design curves for three module sets. Every design point on a curve corresponds to an actual design produced by Sehwa with different values of latency[1]. The design space covered by Sehwa for design exploration for a good design is 834400 mil^2 to 8300 mil^2 in area and 375 nS to 117920 nS in time. On the other hand if only one module set (say module set 1) were used for searching the design space, Sehwa would cover a much smaller design space. The module selection program explores the larger design space and selects an optimal module set meeting the user constraints.

1.2 Assumptions

Our solution to the module selection problem is a limited one in that we have made the following assumptions. The third assumption is what differentiates this work from previous module selection research.

1. It is assumed that resynchronization (flushing the pipeline) does not occur. The two main reasons for resynchronization are exception conditions and conditional branches. Signal processing applications seldom have conditional branches, and hence resynchronization due to conditional branching is rare. Furthermore, as the data is normalized to meet the pipeline's arithmetic precision capabilities, exception conditions due to arithmetic errors are reduced. (See [6] for examples.)
2. An operation must be completed within one clock cycle. An operation cannot be scheduled into two or more stages. Of course, any such partitioning can be achieved by a priori division into two or more sub-operations.
3. Module selection is performed prior to scheduling and operator allocation.
4. If there are two modules which implement the same operation, then the faster module is bigger than the slower one.
5. It is assumed that every operation in the data flow graph can be implemented by at least one module or combination of modules in the library. If it does not exist, then an intelligent interface (like Fred [12]) can supply module parameters based on the existing modules. For example, if the data flow graph has an 8-bit addition operation then an intelligent interface can extrapolate 4-bit adder parameters (area and delay) and provide the module selection program with parameters for a 8-bit adder. A module generator could be invoked to implement one.

[1]Latency is the number of clock cycles between initiations of two successive data inputs, as used in [5].

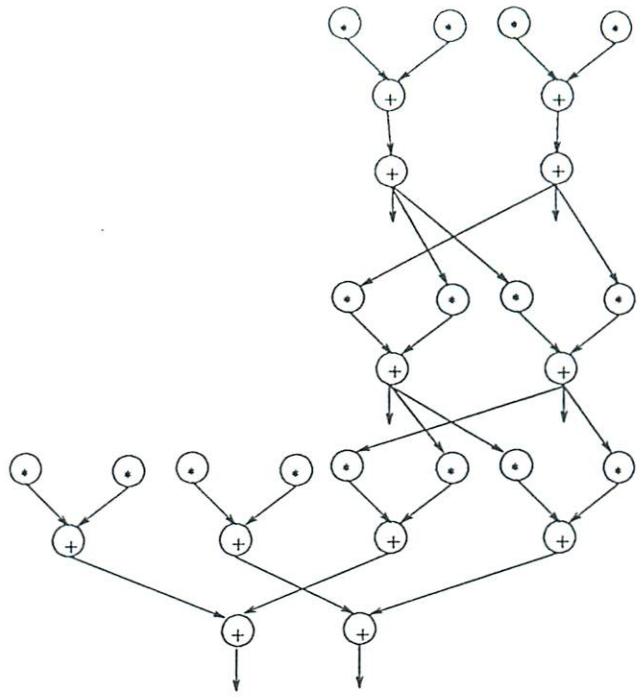


Figure 1: AR Filter Data Flow Graph

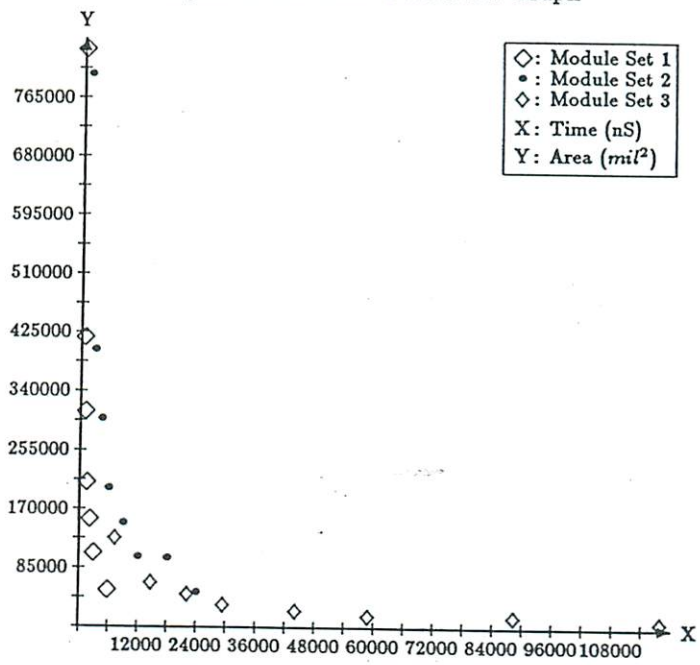


Figure 2: Designs Produced by Sehwa Using Several Module Sets

1.3 Related Research

Of the many research papers on data path synthesis of digital systems, there are very few which address module selection (e.g. [2,8]). In most synthesis research, the problem of module selection has been simplified in that a fixed mod-

ule set is predetermined for implementation [1]. We believe this to be the first effort which formalizes and proposes a solution to the module selection problem for pipelined designs.

Leive [8] proposed a solution to the general module selection problem for non-pipelined designs where module binding is done after scheduling and operator allocation. Each module is evaluated by an individual optimization function, from which the best module is selected for implementation. It is not known whether optimizing every operation type leads to the globally optimal solution for the module selection problem or not. As we shall see in Section 3, optimizing the module type for every operation type individually does not necessarily lead to a globally optimal result for pipelined designs.

In [2], module binding and module selection problems are solved concurrently for non-pipelined designs using a mixed integer-linear programming (MILP) technique. The basic drawback of this technique is that the solution entails enormous computer runtime and is not practical for realistic examples. Also, the module selection problem by itself is not independently solved, thus forcing the user to solve the whole problem of data path synthesis. However, the results obtained for module binding and module selection are optimal.

2 Solution Approach

2.1 Overview of the Module Selection System

The input to our module selection system is (i) a behavioral description of the target design in terms of a data flow graph, (ii) a library of modules which can perform the operations of the data flow graph, and (iii) a cost or performance constraint (for example, the area of the design should not exceed 200 mil^2). The module selection system chooses appropriate modules from the library for implementation.

The module selection system chooses a module set after three processing stages. The first stage generates a set of module sets, and rank orders the module sets according to an objective function to be described below. If there are no constraints, the module set which minimizes the objective function is chosen, and the program terminates. Otherwise processing stage two is begun. If there is an area constraint on the design, then the program finds the module set which meets the constraint and has the minimum objective function of all sets which meet the constraint. If there is a performance constraint, then the program finds the module set which meets the constraint and has minimum area.

Many times, the resources in a digital system are not fully utilized every time step. In such cases, the design space is not as well behaved, and local searching must be performed in stage 3 to find the best module set meeting constraints.

2.2 Choice of Objective Function

The objective function used to rank order the module sets is the area-performance product of the target design, using that module set. This objective function, taken from [3], estimates a lower bound area-time curve for a given data flow graph and a module set to be

$$A \times T = constant \qquad (2.2.1)$$

A is the functional area of the design. $A = \sum_{i=0}^{m-1}(a_i \times o_i)$ where a_i is the area of module which implements operation i, o_i is the quantity of these modules and m is the number of different types of operations in the data flow graph. T is the delay between two successive initiations of new data. $T = c \times l$ where c is the clock cycle of the design and l is the latency.

2.3 Candidate Module Set Selection

Stage 1 consists of module generation and rank assignment. Algorithm 1 given in Figure 3 generates several module sets for every possible value of clock cycle which minimizes Equation 2.2.1. We know from [3] that the best possible clock cycle for a design is $c = maximum(d_i)$, where d_i is the delay of all the modules selected for the design. The quantity of different values which clock cycle can take is bounded by $\sum_{i=0}^{m-1} p_i$, where p_i is the number of modules which implement operation type i. Thus, the number of module sets meeting the requirements of Equation 2.2.1 is $\sum_{i=0}^{m-1} p_i$ in the worst case. This is a subset of the total number of possible module sets ($\prod_{i=0}^{m-1} p_i$). The set of different delays which exist for all module types for all operations forms the set of possible values of clock cycles. This set is used in Step 1 of the algorithm. Each candidate clock cycle is then selected for consideration. For the selected clock cycle there will be several AT curves. The curve for the selected clock cycle with minimum value of AT is the one which minimizes Equation 2.2.1. This is easily achieved by ensuring that the delay of every module selected does not exceed the selected clock cycle and has minimum area. The pruning of the search space for a selected clock cycle is done in Step 3 of Algorithm 1.

Once the module sets are generated, they are sorted in increasing order of their AT value. If there are no constraints, the module set with smallest AT value is selected.

```
1. For every possible value of clock cycle {
        /* i.e. for every unique value of dl_ij */

2.      For each operation type {

3.          Choose the module with minimum area and
                delay less than or equal to the clock cycle.
                If one cannot be found, then this clock cycle
                is rejected.
        }

4.      Compute estimated AT of the design from
        Expression 2.2.1 using this module set.
    }
```

Figure 3: Algorithm 1

2.4 Location of Module Sets Meeting Constraints

Every module set generates a number of design points depending on the values of latency. However, the user may specify a *constraint* which is not met by the design points generated using the best module set. In this situation the user may have to settle for an inferior solution which meets the constraint. Thus we have the problem of finding the best possible module set which does meet the user constraint.

Figure 4 shows an example of selecting the best possible module set assuming that each design point is operator-optimal. Different curves correspond to different module sets and the design points on each curve are with different latency. Suppose the user specifies an area constraint of 200 mil^2 (shown by a dotted horizontal line). Then, module sets 1 and 2 cannot satisfy this constraint and are rejected. Only module sets 3 and 4 can satisfy the constraint. Of these two, module set 3 minimizes Equation 2.2.1 and is selected as the best choice. The search procedure processes the sorted module set list in order, and selects the first module set whose cheapest design (longest latency) meets the constraint. A similar approach is used when the user specifies a performance constraint.

2.5 Local Searching with Non-Optimal Designs

In reality there will be non-optimal design points since not all operators will be utilized fully every cycle. To handle this situation, a smaller region in the design space is explored for the best possible solution. The procedure determines whether a module set meets the constraint by determining whether there is a *best case* design which meets the constraint. If the constraint is on area then the end of the curve which represents longest latency is checked.

Let the jth module set be selected in the procedure described in Section 2.4.

1. The design points for the jth module set are generated[2]. Let x be the design point which satisfies the constraint and which has the minimum AT among all the design points generated using this module set.
2. If any design point of either module set $j-1$ or $j+1$ has a smaller AT than the AT of point x and meets the constraint, then this module set is conditionally selected as module set j and Steps 1 and 2 repeated.

3 Experiments and Results

Several experiments were conducted to ensure that Algorithm 1 did indeed generate the best module sets and satisfy the following two requirements:

[2]This can be easily done by using the estimation procedure of [3].

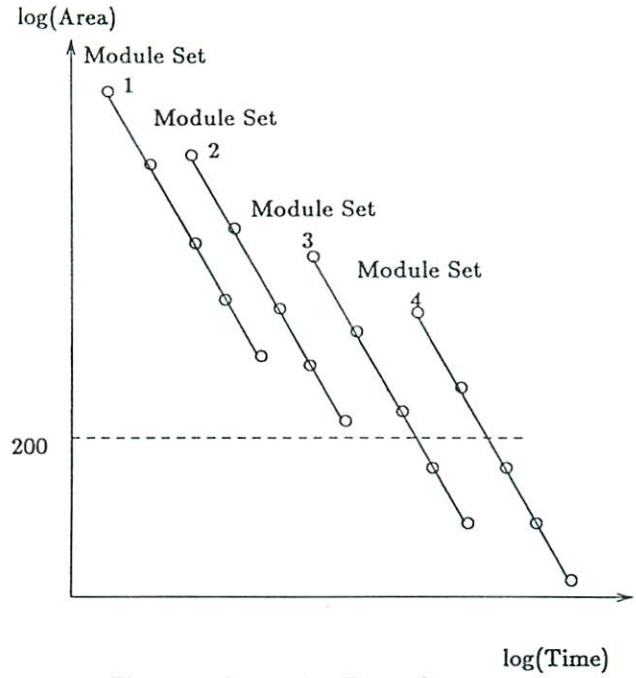


Figure 4: Constraint Example

1. the design space explored by considering all the $\prod_{i=0}^{m-1} p_i$ module sets contains the same optimal surfaces as that explored by the selected $\sum_{i=0}^{m-1} p_i$ (or fewer) module sets, and
2. if a design point meeting some constraint can be generated by the unselected module sets, then the selected module sets can also generate a design point which can not only satisfy the same constraint but better the cost-performance. This is because the module sets which generate design curves closer to the origin are selected over of the ones further from the origin.

The results were verified using Sehwa. The library of modules, generated by an area estimation program PLEST [7], consisted of three add modules, three subtract modules, and three multiplication modules (Table 1). Three data flow graphs taken from [3] were pipelined using Sehwa. The AR filter data flow graph (Figure 1) consists of addition and multiplication operation types. As the library has three add modules and three multiplication modules, a total of nine various combinations of module sets were formed for this example. Sehwa was executed using these module sets for the AR filter. The results shown in Figure 5 are plotted on a normalized log-log scale for better readability.

Algorithm 1 was executed using the AR filter data flow graph and the library. Algorithm 1 generated five module sets which are listed in Table 2 (the entries are sorted in increasing AT). Comparing the results in Figure 5 produced by Sehwa with those produced by Algorithm 1 in Table 2, it is seen that the five module sets produced by Algorithm 1 do encompass as much optimal surface in the design space as all nine module sets. The selected five module sets *cover* the unselected four module sets. By *covering* we mean that

a design meeting a constraint which can be produced by the unselected module set can also be generated by the selected five module sets; not only that, but the designs on the optimal surface generated using the selected module set have a better cost-performance than the designs generated by the unselected module set. Algorithm 1 generates five module sets for the AR filter data flow graph. The selected module set (m_1, a_3) is a redundant module set which can be covered by the other four selected module sets.

Module Name	Operation	Area mil^2	Delay nS
a1	Addition	4200	340
a2		2880	530
a3		1200	1510
s1	Subtraction	4200	340
s2		2880	530
s3		1200	1510
m1	Multiplication	49000	375
m2		9800	2950
m3		7100	7370

Table 1: Module Parameters

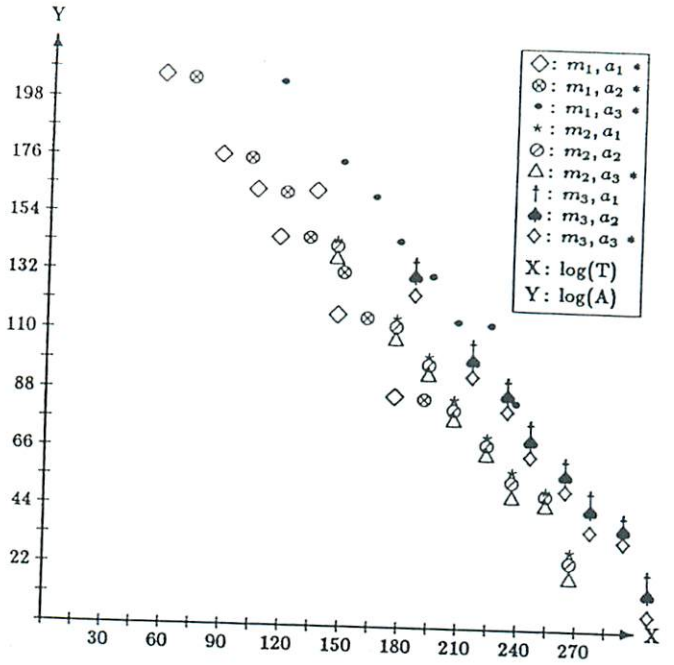
AR Lattice Filter	Conditional Data Flow Graph	Random Data Flow Graph
$(m_1, a_1)^*$		$(m_1, a_1, s_1)^*$
(m_1, a_2)	$(a_1, s_1)^*$	(m_1, a_2, s_2)
(m_2, a_3)	(a_2, s_2)	(m_2, a_3, s_3)
(m_3, a_3)	(a_3, s_3)	(m_1, a_3, s_3)
(m_1, a_3)		(m_3, a_3, s_3)

* Module set with minimum AT.

Table 2: Module Sets Generated Using Algorithm 1

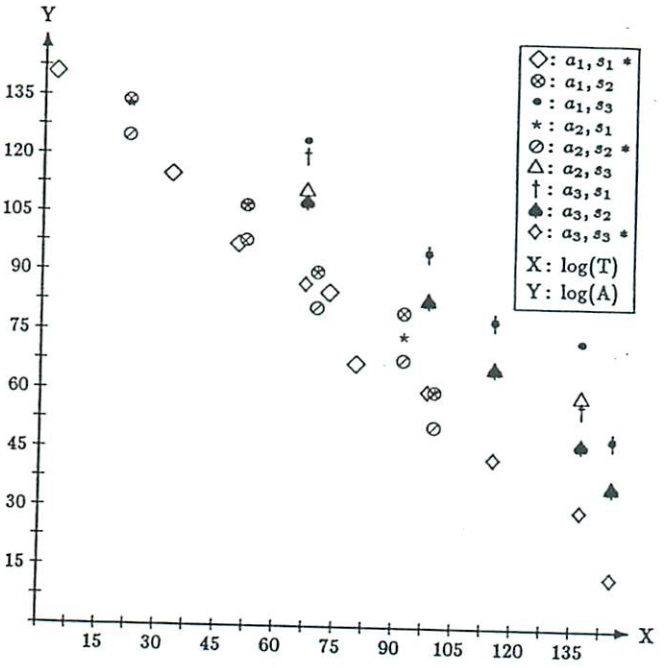
Similar results for other data flow graphs were obtained. Figures 6 and 7 show results when non-optimal designs were encountered and local search took place. In Figure 6, module set (a_2, s_2) was selected over (a_1, m_1) when the area constraint of 84 was specified. In Figure 7 module set (m_1, a_2, s_2) was selected over (m_1, a_1, s_1) when an area constraint of 98 was specified.

We shall now give an example where selection of modules whose individual AT products are minimum does not result in global optimization. Let us assume that module m_1 does not exist. By selecting modules on the basis of optimization of individual modules, module set (m_2, a_1) would be chosen for the AR lattice filter example. Seeing the results in Figure 5, we observe that this module set is not selected. Instead, a better choice which would cover the same optimal design space with better AT is the module set (m_2, a_3). Thus, optimizing individual nodes in the data flow graph does not necessarily lead to a globally optimal solution for pipelined design.



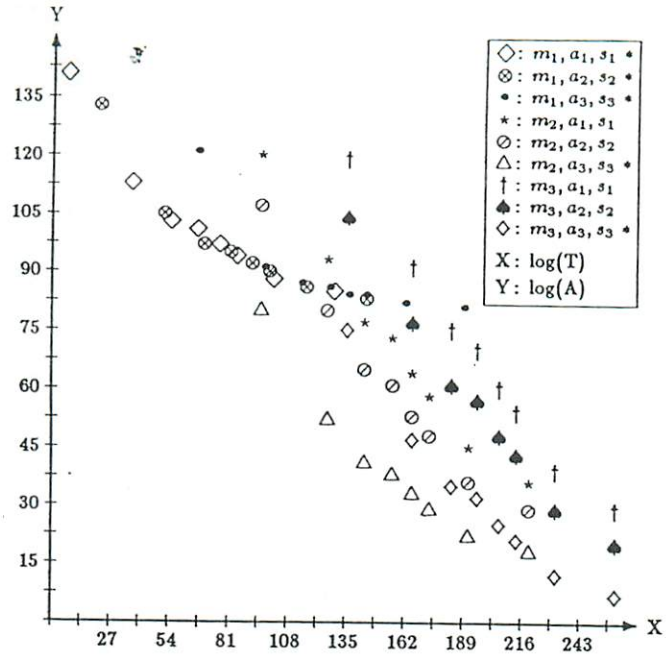
* indicates the module set selected by Algorithm 1

Figure 5: Area Time Curves For AR-Lattice Filter



* indicates the module set selected by Algorithm 1

Figure 6: Area Time Curves For Conditional Dataflow Graph



* indicates the module set selected by Algorithm 1

Figure 7: Area Time Curves For Random Dataflow Graph

4 Conclusions and Future Research

The algorithms presented in this paper select an optimal module set from a library for implementing a pipelined design with a design constraint. The model presented here is general and is applicable to any pipelined synthesis tool with the assumptions mentioned above. The algorithms presented here are polynomial in time [4]. The algorithms have been coded in C and have runtimes of seconds on a SUN 3 workstation.

In the above analysis, only the operator cost was considered. The optimization can be further refined to include the cost of multiplexers and registers. This can be achieved by using an equation which includes the estimated register and multiplexer cost in place of Equation 2.2.1 to start with. The search procedure for the best module set given the user constraint may have to be modified when this additional cost becomes important as the latency increases. Another possible method of including the effect of register, multiplexer and wiring may be to consider a lumped model. This would imply adding an estimate of global wiring, register and multiplexer cost and delay to the module cost and delay. The effect of resynchronization on module selection has to be studied also.

References

[1] E. Girczyc and J. Knight, **An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling**, *Proc. 1984 Int. Conf. on Computer Design - ICCD*, IEEE, Oct. 1984.

[2] L. J. Hafer and A. C. Parker, **A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic**, *IEEE Trans. CAD.*, Vol. 2, No.1, January 1983.

[3] R. Jain, A. C. Parker and N. Park, **Predicting Area-Time Tradeoffs for Pipelined Designs**, *Proc. 24th Design Automation Conference*, DAC, June 1987.

[4] R. Jain, A. C. Parker and N. Park. **Module Selection for Pipelined Designs**, *Technical Report CRI 87-59*, Department of Electrical Engineering - Systems, University of Southern California, 1987.

[5] P. M. Kogge, **The Architecture of Pipelined Computers**, *McGraw-Hill Book Company*, 1981.

[6] S. Y. Kung, **On Supercomputing With Systolic/Wavefront Array Processors**, *Proceedings of the IEEE*, Vol. 72, No.7, July 1984.

[7] F. J. Kurdahi and A. C. Parker, **PLEST: A Program for Area Estimation of VLSI Integrated Circuits**, *Proc. 23rd Design Automation Conference*, DAC, June 1986.

[8] G. M. Leive, **The Design, Implementation, and Analysis of an Automated Logic Synthesis and Module Selection System**, *Ph. D. Thesis*, Carnegie-Mellon University, Pittsburgh, 1981.

[9] M. C. McFarland, S. J., **Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions**, *Proc. 23rd Design Automation Conference*, DAC, June 1986.

[10] N. Park and A. C. Parker, **Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications**, *IEEE Trans. CAD*, Vol. 7 No. 3, March 1988.

[11] A. C. Parker, J. Pizarro and M. C. Mlinar, **MAHA: A Program for Datapath Synthesis**, *Proc. 23rd Design Automation Conference*, DAC, June 1986.

[12] W. Wolf, **Fred: A Procedural Database for VLSI Design**, *Proc. 23rd Design Automation Conference*, DAC, June 1986.