SMART: Tools and Methods for Synthesis of VLSI Chips with Processor Architecture

T. Bergstraesser, J. Gessner, K. Hafner, S. Wallstab

Siemens AG
Corporate Laboratories for Information Technology
P.O.Box 830953, D-8000 Munich 83, FRG

Abstract

A design environment supporting processor synthesis in data path style is presented in this paper. The programming model of a processor described in Common Lisp is transformed into a hardware structure by means of tools integrated into this environment. The generation of alternative designs is supported by the interactive graphical manipulation of behavior and hardware structure representations and their correspondences. The synthesis procedure is explained using an example.

1. Introduction

The automatic generation of production and test documents by design tools (silicon compilers) has become established practice in the design of application-specific integrated circuits (ASICs). In this process the circuit to be designed is described at the structural level, for example by a logic circuit diagram.

Our research team is concerned with the implementation of algorithms by VLSI circuits and the development of chip architectures adapted to specific problems. The lack of tools for evaluation and synthesis becomes apparent especially in the early stages of design.

In the SMART project (Synthesis of Modular Architectures with Test Support), therefore, we are developing tools for the synthesis of VLSI architectures from a behavioral description.

Our analyses of existing research tools in high level synthesis show that only a significant restriction on the algorithms to be implemented, on the target architectures, and on the design steps to be automated allow us to develop tools which can be used in practice within a short time.

The drawback of this solution is that in many cases interactive optimization of the synthesized hardware structure is necessary. To support this the synthesis procedure is divided into single steps, which are performed individually by different tools. This concept helps the user to understand the behavior of the synthesis tools. For this reason the expensive trial and error steps required to produce a satisfying solution are reduced.

Our main goal was to implement tools for those subproblems in circuit synthesis which are most error prone in the manual design procedure and cost most time and effort. An example of such a bottleneck in chip architecture design is the coding of control section functions.

2. Target architecture

The operative part of our target architecture is composed of function blocks, which are connected via a bus system. The use of architectures with dual-bus systems and their extension with local buses as an implementation scheme allows cost-effective implementation of operations with one or two operands mainly involved here [4].

The actual restriction of our tools to such a regular target architecture is a crucial point in reducing the complexity of the automatic synthesis, i.e. the process of mapping the behavioral description onto the hardware. In this process we currently accept the trade-off that only a limited class of algorithms can be transformed by the system and that the user must select a particular (processor-like) kind of behavioral description.

3. The SMART environment

As stated in the introduction, our design environment consists of tools for synthesis subtasks. The problems with "tool boxes" are the data structures required within the programs and the interfaces required between the subtools. In an uncoordinated configuration, every partial solution must generate and manage its own data structures.

If the data structures are integrated in a development environment and made available via a well-defined interface to all subtools, then the problems mentioned above can be avoided. In addition, the overhead in generating the individual programs is reduced, since existing structures can be used as a basis. Fig.3.1 shows the configuration of the SMART environment.

As the first design step in this environment the user has to enter the programming model, i.e. the operations and data types of the processor under design (EDITOR in Fig. 3.1). To support the architecture design by computer aided tools, a suitable format for the description is required.

Our investigations of behavior-oriented input languages of existing synthesis systems showed them to permit descriptions almost exclusively on the bit and bit-pattern level. This type of representation requires the user to have a detailed knowledge of the system to be designed, which he cannot expect to have or may even not want to have at this time. Examples are the in-

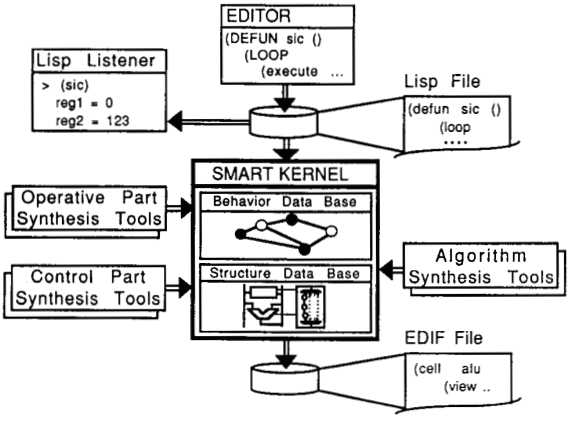

Fig. 3.1 The SMART environment

struction format and the instruction coding in a processor. A symbolic description in terms of "load", "store", "add", etc. appears to be more suitable here.

We use the programming language Common Lisp [5] for a behavior-oriented symbolic description of systems, but only to represent the algorithm and not to specify constraints such as time or area. Thus, no changes of the syntax and semantics of the programming language are necessary.

The advantage of this approach is that existing programming environments can be used. Further, the specification may be executed for validation on a computer when it is regarded as a program. The interpretative processing permitted by Common Lisp allows an interactive simulation to be carried out on the algorithmic level (Lisp Listener in Fig. 3.1).

The internal data structures which represent the behavior are generated from the behavioral specification by means of an intermediate code compiler (SMART KERNEL in Fig 3.1). Both the internal behavior and hardware structure representations can be manipulated by the user through graphical display windows in mouse and menu mode. Every object has an attached menu with permitted commands. Commands are provided for manipulating objects, listing informations about objects and starting synthesis subtasks.

The processor synthesis in the SMART environment is composed of subtasks communicating only via internal data structures. The user is therefore able to generate alternative designs by controlling the sequence of subtasks and by manipulating the data structures before, after, and in some cases even during the synthesis procedures.

The remainder of this paper is dedicated to the main subtasks during the semi-automatic synthesis in the SMART environment. As an example we make use of the rudimentary processor SIC (Simple Instruction Computer). Its programming model has only four types of instructions (*ADD, SUB, LOAD, STORE*), but it includes the essential addressing modes (*register-direct, register-indirect, immediate*).

4. Algorithm synthesis

The input for this synthesis step is the program flow graph [1]. This graph not only contains the functions, operations and variables which are present in the input description, but also represents relationships between them. The call hierarchy of the functions is modeled by the call graph, the control flow of the functions by the connection of the basic blocks. The basic blocks are sequences of operations in the behavioral description which can be processed sequentially. The data flow contained in each basic block is described by a data flow graph. Fig. 4.1 shows the call graph of the SIC, the control flow graph of the FETCH-OPERAND function and the data flow graph for the sequence of operations required to load an immediate operand.

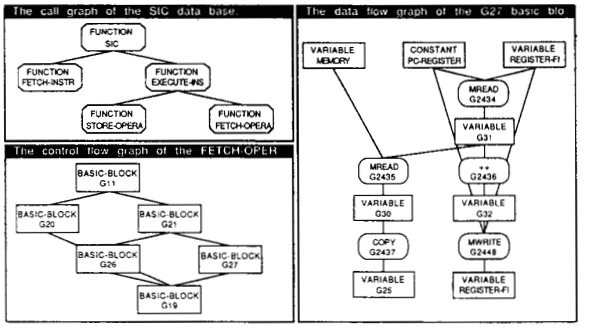

Fig. 4.1 The call graph, a control flow graph and a data flow graph of the SIC.

The most important transformations for optimizing the specified processor behavior are:

- **Function integration**

In this step, function call statements can be replaced by the function body. In the SMART environment the user is able to select the call statement or the function which should be processed. In our current small example, all function calls are expanded, i.e. the function structure entered by the user is totally lost. Fig. 4.2 shows the control flow graph of the SIC after the expansion of all functions.

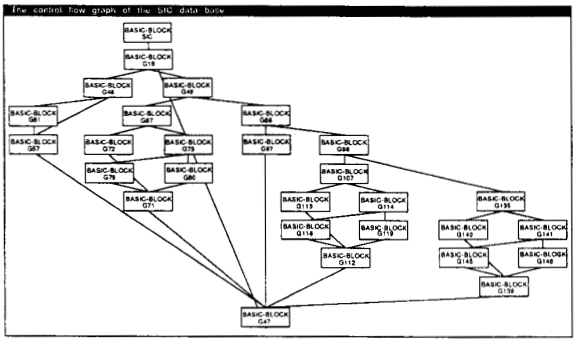

Fig. 4.2 The control flow graph of the SIC after function integration.

This step permits the control flow graph to be enlarged and thus enhances the optimization options. This transformation also considerably reduces the outlay for the

subsequent analysis and optimization steps, because the boundaries of the functions and the context of the function calls do not need to be considered by the optimization algorithms.

- **Combination of branches**

The aim of this transformation is to combine the branches in the control flow as much as possible at the beginning of the graph. In the class of processor-like descriptions considered in our synthesis procedure, this always results in a single basic block for every instruction. This is possible, because after instruction decoding (second basic block in Fig. 4.2) the value of every decision variable in the remaining control flow graph is known. By obeying the data dependences, the transformation of the control flow graph in Fig. 4.2 results in the graph shown in Fig. 4.3. The initialization, instruction fetch and instruction decoding are executed in the first basic block. The selection of one of the permitted instruction combinations then takes place in the following chain of basic blocks. Afterwards the selected instruction is executed and control returns to the second basic block.

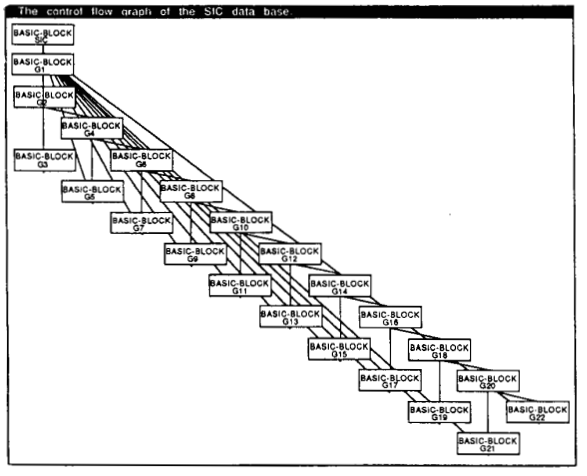

Fig. 4.3 The control graph of the SIC after combination of branches.

Two objectives are pursued with this transformation. Firstly, it represents a "standardization" which eliminates influences resulting from the specification structure selected by the user, and secondly the combination of basic blocks permits the data flow graphs to be enlarged and thus enhances the optimization options. Its drawback is that a considerable increase in the number of nodes must be accepted as a trade-off due to the redundant operations in the paths. When the operations are assigned to hardware components, this redundancy is eliminated by multiple utilization of functional units, if this is permitted by the constraints.

5. Operative part synthesis

In this step, the data flow graphs obtained from the algorithm synthesis are mapped onto the target architecture.

The following steps are performed here:

- **Bus and register allocation**

To solve the conflicts introduced by the usage of a restricted number of buses and registers, we use parts of an algorithm well known in state minimization of FSMs (Finite State Machines). This modified version of the Paull-Unger [6] recursive procedure for determining compatible state-pairs computes the maximum sets of concurrent bus and register usages. If no such sets can be found, then intermediate variables must be inserted and the process must be restarted. The intermediate variables are composed to a scratch pad memory. To avoid any data conflicts arising due to overwriting values this scratch pad is not explicitly addressable on the programmer's level.

The output of this sub-task is a dependence graph [3] showing the predecessor and successor relations between the operations. Fig. 5.1 shows the data flow graph of the *ADD immediate* instruction (data flow from top to bottom) and the dependence graph after bus and register allocation.

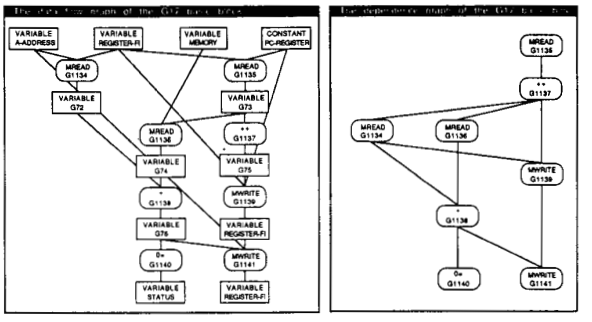

Fig. 5.1 Data flow graph and dependence graph of the *ADD immediate* instruction.

- **Scheduling**

We use a modified form of the Critical Path Method (CPM), commonly used in project planning, to define the exact temporal sequence of the operations.

In this method, all the nodes of a data flow graph are supplied with the earliest starting and finishing times. The delay times of the operations are determined from the modules in the module library. This calculation takes place in the forward direction, i.e. in line with the data flow in the data flow graph.

When the graph has been processed, the maximum delay is known and the latest starting and finishing times for the nodes are calculated in the reverse direction. At the same time the buffer times, which specify the amount by which the nodes can be shifted, are also determined. These values are used during hardware allocation to select the realization for each node.

- **Hardware allocation**

In this step, the variables, constants and operations are allocated to hardware elements. While selecting the design representation for our design environment, we laid special emphasis on the possibility of a non-isomorphic decomposition [7] between the behavioral and structural descriptions. Relationships between the elements of these two layers are set up only via references

(cf. Fig. 5.2) generated during hardware allocation or by the user.

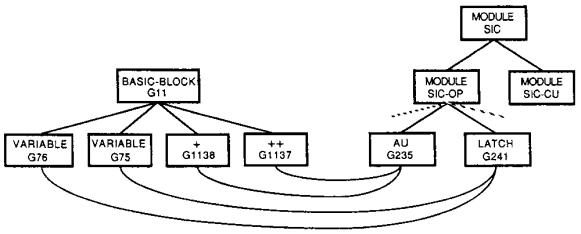

Fig. 5.2 Behavior and hardware structure decomposition and domain mapping

The algorithm must decide whether a node of the data flow graph should be implemented by a new instance of a hardware module from the library or by an existing module in the operative part.

The bases for this choice are firstly the buffer plus the delay time specified for the node, within which the delay time of the components must fall, and secondly the effects on the area of the data path, the wiring channel and the control section.

The difficulty in finding a cost function for these design decisions which is suitable for practical use made us choose the regular target architecture mentioned above.

The node implementation option offering the best area solution and obeying the specified maximum delay time is then selected. Fig 5.3 shows the generated data path of the SIC.

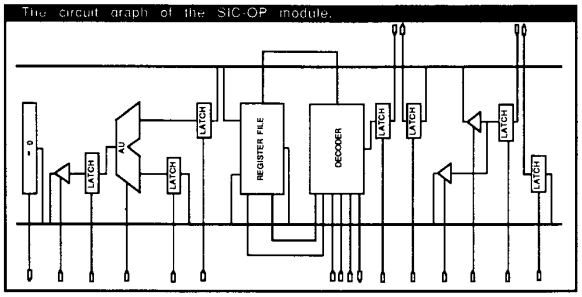

Fig. 5.3 Operative part of the SIC

6. Control part synthesis

To support the optimization and design of the control section we implemented a special representation scheme. In this scheme, every control unit of the control section can contain a control graph, whose branches are controlled by the incoming signals and which in turn defines the activation of the outgoing signals.

During operative part synthesis the necessary control nodes are integrated in the control graph contained in a control unit module. The control section synthesis is performed on this control graph, e.g., state optimization and selection of the form of implementation.

Fig. 6.1 shows the part of the SIC control graph for the *ADD* instruction (*register-direct, register-indirect, immediate*).

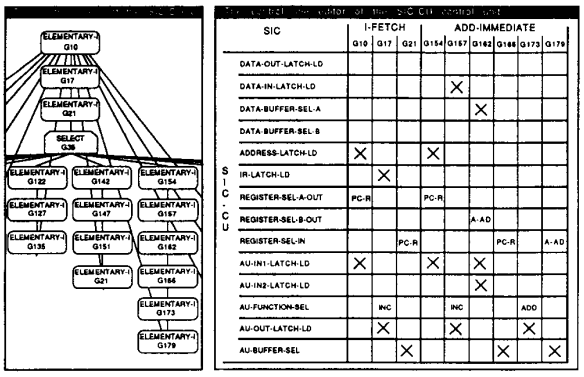

Fig. 6.1 Control graph and control line editor

The semi-automatic optimization of the control graph is carried out by a control line editor which permits manipulation of the allocation of control signals to nodes in the control graph. Fig. 6.1 shows the window of this editor for the *ADD immediate* instruction.

7. Conclusion and future work

After the synthesis of the hardware structure, the realization is performed by existing tools such as structural silicon compilers.

To extend our environment we are integrating a floor planning tool [2] and developing algorithms for the synthesis of pipelined processor architectures. Further areas of research interest are extended synthesis of the control section and evaluation functions for more general chip architectures, especially for the control sections. We are currently using the SMART tools in parallel with the manual design of an application-specific processor.

8. Acknowledgments

The work on which this report is based was partly supported by funds from the Federal Department of Research and Technology of the Federal Republic of Germany (Research code No. NT2828/9). However, the authors alone are responsible for the contents.

References

[1] Aho, A.V.; Hopcroft, J.E; Sethi, R.; Ullman, J.D.: Compilers – Principles, Techniques, and Tools. Addison-Wesley, 1986.

[2] Endriss, H.; Klein-Hessling, G.; Lawitzky, G.; Schallenberger, B.: Fast Evaluation of Design ALternatives - A New Approach for Exploratory Chip Design. Microprocessing and Microprogramming, North Holland, Vol. 21, 1988.

[3] Kuck, D.J.; Kuhn, D.; Padua, A.; Leasure, B.; Wolfe, M.: Dependence Graphs and Compiler Optimizations, Comm. ACM, 1981, pp. 207-218.

[4] Moeller, W.D.; Sandweg, G.: The Peripheral Processor PP4 – A Highly Regular VLSI Processor. 11th Int. Symp. on Computer Architecture, 1984, pp. 312-317.

[5] Steele, G.L.: Common Lisp, Digital Press, 1984.

[6] Unger, S.H.; Paull, M.C.: Minimizing the Number of States in Incompletely Specified Sequential Switching Functions. IEE Trans. Electronic Computers, EC-8, 1959, pp. 346-356.

[7] Walker, R.A.; Thomas, D.E.: A Model of Design Representation and Synthesis. 22nd Design Automation Conf., 1985, pp.453-459