24

Module generators and their integration in an architectural systhesis system

J.F.M. Theeuwen
Eindhoven Universisty of Technology
P.O. Box 513, 5600MB Eindhoven, The Netherlands.
Telephone: 31-40-473344. Fax: 31-40-464527.
email: J.F.M.Theeuwen@es.ele.tue.nl

Abstract

*Currently a lot of research is going on in the field of architectural synthesis. Many problems are addressed in this field, like scheduling, allocation, module binding and register allocation. To be able to judge the value of an architectural synthesis system in real world and to make the system appropriate for real world applications an extended library of basic operations is required. Not only the number of basic operations has to be large enough, it should also be possible to make a trade off in terms of area, speed and required number of clock cycles for an operation. In this paper we show how flexible module generators can be connected to an architectural synthesis system. Moreover we will describe two module generators, a divider generator and a multiplier generator.

Keywords

Architectural synthesis, module generators.

1 INTRODUCTION

Currently a lot of research is going on in the field of architectural synthesis. Many problems are addressed, like scheduling, allocation, module binding and register allocation. There are a number of well known benchmark examples like FDCT (Fast Discrete Cosine Transform) [Mall90] and WDF (Fifth order elliptical filter)[dewi85]. For benchmarking purposes the library of basic operations that is used is rather simple and does not have much relation with the real world. Especially the notion about area and delay is too simple. To be able to judge the value of an architectural synthesis system in real world and to make the system appropriate for real world applications an extended library of basic operations is required. Not only the number of basic operations has to be large enough, it should also be possible to make a trade off in terms of area, speed and required number

*This research has been made possible by the support of the ESPRIT III BRA project 6855, called Link

G. Saucier et al. (eds.), *Logic and Architecture Synthesis*

of clock cycles for an operation. To make the above described requirements possible two things should be available:

(1) module generators that are able to generate basic functions in many different versions (speed, area, number of clock cycles, bit width).
(2) communication from the module generators to the architectural synthesis system, to inform the architectural synthesis system the ability of the module generators.

In the next chapter an overview of the developed architectural synthesis system will be given, with special emphasis on the description and selection of the library elements. After that the generation of a customized library for a certain design will be explained. At the two examples of module generators will be shown.

2 THE ARCHITECTURAL SYNTHESIS SYSTEM

The task of an Architectural Synthesis System is to translate a behavioral description of a chip into a digital network description. This network has to be composed of functional modules (like ALU's, multipliers, adders), storage modules (register files, ROM, RAM), and controllers. One of the requirements on these items is that there is an (automatic) way to generate a layout for the required module). The task of translating the behavioral description into a network is rather complicated, so architectural synthesis is partitioned into a number of sub steps:

- Selection: What kind of modules are required
- Allocation: How many resources are necessary
- Binding: Which operations have to be performed on which specific resource
- Scheduling: When should specific operations be activated

In general all these problems are NP complete and a lot of research is going on in these fields. To allow a large freedom for research in the field of architectural synthesis, an interface, the New Eindhoven Architectural synthesis Toolbox (called NEAT) has been developed, which is independent of the synthesis tools and the design trajectory chosen. A central item in this system is the way the design data are represented. In an architectural synthesis system three domains of data can be distinguished:

- Behavior:
  The behavior of a chip is often specified by an algorithm, written in a special language like VHDL, Silage, Ella or Hardware C. To resolve the different natures of the languages a central format is needed. This format should have a close relation with the nature of hardware, but should not impose a particular hardware structure. A Data Flow Graph (DFG) is chosen as the central format [eijn92]. A DFG is a directed graph consisting of nodes representing operations, and edges representing transfer of values (tokens). The behavior of a data flow graph is defined by a so called token flow mechanism. Tokens are objects which can bear a particular value. Tokens are transported from origin nodes (producing a value) to destination nodes (consuming a value) by the edge which connects the two nodes. A node can start its execution when tokens are available

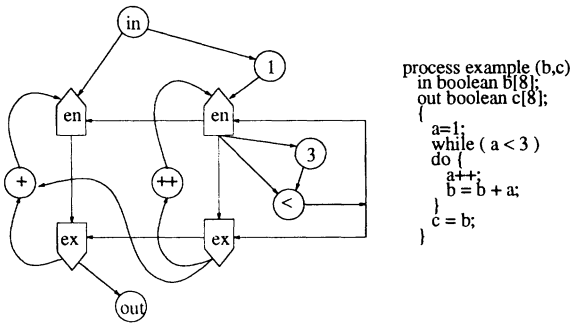

Figure 1 A hardware C program with the corresponding DFG

on all the incoming edges of the node. After execution the node produces tokens on all the outgoing edges of the node. To support special language constructs like loops and branches, nodes with a different execution model are introduced. In figure 1 an example of a Hardware C algorithm and its corresponding data flow graph are shown.

- Time:
  A control graph is used to specify the behavior of the controlling finite state machine. The nodes in the control graph correspond to states, and the edges denote possible state transitions. Control graphs are extended with some special constructs to explicitly represent conditionals, loops, multiple active states and hierarchy.
- Structure:
  A network graph is used to describe the resulting digital network. The nodes of the network graph correspond to the physical modules. Edges represent the interconnections between these modules.

2.1 Intra-domain relations

If a graph is used somewhere as an operation, it will appear as a single node. The node is called an instantiation of the graph. The type of the node is used to refer to the graph that is instantiated, and hence defines the behavior of the node. Standard operations or modules, like additions, multiplications, ALU's etc. can be created by instantiation of the corresponding graph. These operations or modules can be supplied to the system by libraries. In chapter "libraries in the NEAT system" we will discuss this item more extensively.

2.2 Inter-domain relations

In passing through some architectural synthesis processes, relations arise among objects of different domains. A scheduler for instance relates data flow nodes to particular states; a binder relates data flow nodes to functional modules. There are two kinds of such inter-domain relations:

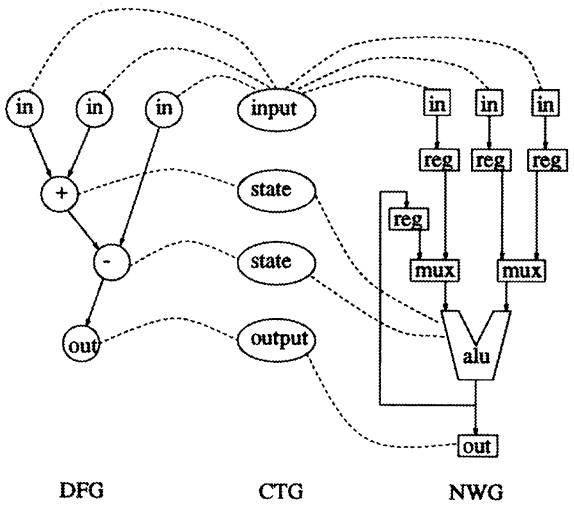

Figure 2 Representation of a complete design

- Graph level:
  relates behavior, time and structure, i.e. which data flow graph can be implemented upon which network graph, and which control graph specifies the time behavior. The objects used to denote these relations are called graph links.
- Node level:
  relates operations, the states in which they are executed, and the modules upon which they are executed. The objects to denote these links are called node links. Node links are the fine grain relations among graphs; they denote relations between data flow nodes, control nodes and network nodes.

An example can be found in figure 2

3 LIBRARIES IN THE NEAT SYSTEM

As explained in a previous chapter the basic operations like addition multiplication etc. are stored in libraries. In these libraries all the properties of the available modules and operators have to be stored. There are three basic items that should be described in the libraries:

(1) The basic operators that are available in the system.
Examples of operators are:+ , - , *, /, etc.

Together with the operation type the name and the type of the input and output terminals have to be annotated. These data are described in the data flow domain.

(2) The basic modules that are available in the system.
Examples are : adder, subtracter, ALU etc.
From these modules the input and output pins have to be described, (type, name, bit width). Also properties like, speed and power consumption of the module can be described on this place. These data are described in the network domain.

(3) The relation between the operators and the basic modules.
The library must tell the system that a + operation can be performed on either an adder or on an ALU. This can be described by graph links between the data flow domain and the network domain. Also the relation between the terminals in the data flow domain and the pins in the network domain have to be described. This can be done by the node links.

3.1 Generation of customized libraries

As you can imagine the power of an architectural synthesis system is heavily influenced by the size and flexibility of the library. The more different kinds of modules for a given operator exist, the more freedom the synthesis tools have to select the appropriate module, and the more possibilities there will exist to share the functionality of that module with other operators. If for a given module different versions in the area-speed domain are available the architectural synthesis system can try to use the smallest version of a module still providing the required speed. It is also evident that for each module there have to exist versions for a lot of different bit widths. Taking into account all the above described properties that have to be present in the library it is clear that the library will be rather large. This has lead to the decision to generate a specialized library for each design. This specialized library will contain only those library elements that will be useful for the given design, i.e. only the appropriate modules with the appropriate bit widths will be described.

3.2 The library generator

The basic structure of the customized library is described in the so called template library. In this library the basic dependencies between operators and modules are described. For instance on this place it is stored that a + operator can be implemented by either an adder, an adder/subtracter or an ALU. By examining the data flow graph to be synthesized, the library generator obtains information about the required operations with the accompanying bit widths. It is evident that it is nearly impossible to design for all the modules many different versions for all the required bit widths and for the different points in the area time domain by hand. This would take a lot of work and it would be very tedious to keep the library up to date. Because of this, module generators for the required operators are written. These module generators not only produce the VHDL description that can be synthesized into layout but also generate data about the performance of the resulting modules in terms of area, delay and power consumption. These data are also used by the library generator to generate the customized library. The overall design and information flow is shown in figure 3.

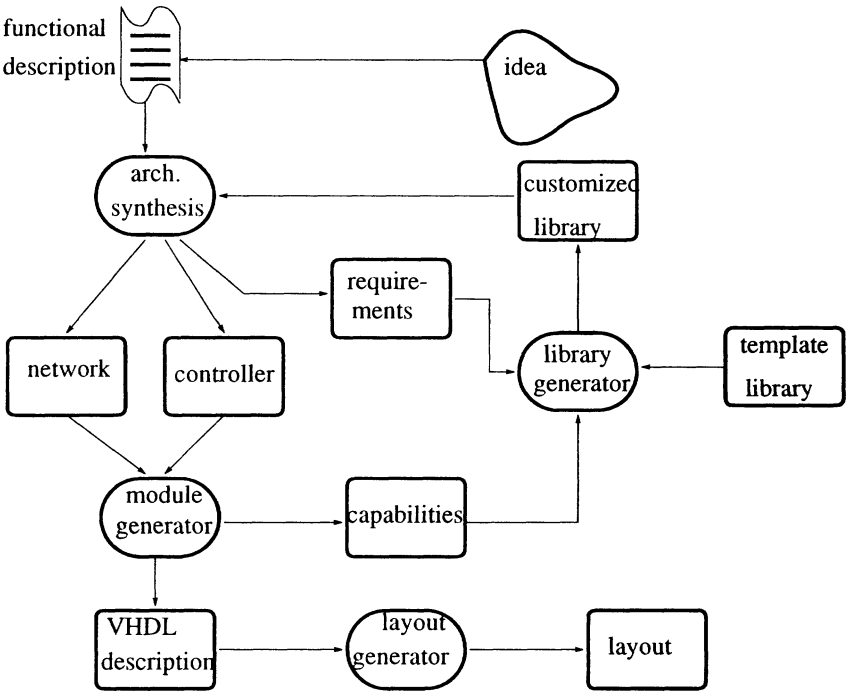

Figure 3 The design flow

4 THE MODULE GENERATORS

In this chapter we will show two examples of module generators that are developed for use in our architectural synthesis system. To be able to use the module generators in as many design systems and technologies as possible the module generators produce a structural VHDL description that can be synthesized into layout by available systems like for instance the "asic synthesizer" from Compass Design automation.

4.1 The Block multiplier generator

The architecture used in the block multiplier generator is described in [arts91], [the93]. The basic idea consists of dividing both operands X and Y of Nx and Ny bits into kx and ky blocks of nx and ny bits (Nx = kx * nx, Ny = ky * ny). The blocks of nx and ny bits will be multiplied with each other in a carry save multiplier and the intermediate results are added in an accumulator. By changing Nx and Ny many different versions of a multiplier can be generated.

4.2 The block divider

The algorithm chosen resembles very much the tail division algorithm used by people performing a division by hand. First the divider and the dividend are aligned (roughly: most significant bits at the same position) Then during each clock cycle n bits of the result are going to be computed. Each cycle it is checked how many times the divider can be subtracted from the dividend. This is done by trying to subtract the multiples 1 to 2n -1 of the divider from the dividend. The largest multiple that results in a non negative result is subtracted and n bits of the result are known. After this the divider is shifted n places to the right and the process is repeated. Studying this algorithm it is clear that for larger values of n this algorithm takes a lot of hardware because all the 2n - 1 multiples of the divider are to be computed and stored. Moreover there need to be 2n - 1 subtracters to check which multiple of the divider can be subtracted from the dividend. Hardware could be saved here by doing the subtractions sequentially instead of parallel. Currently however this option is not exploited.

5 CONCLUSIONS

In this paper we showed that it is possible to make a tight but flexible connection between module generators and an architectural synthesis system. This connection gives the architectural synthesis system the possibility to make full advantage of the capabilities of the module generators. Currently the modules are generated in a standard cell layout style. In the future we will incorporate the use of data paths for some parts of the modules.

6 REFERENCES

[arts91]
Arts, H.M.A.M., Stok, L, Eijndhoven J.T.J. van, "Flexible block multiplier generation", Readers Digest of Technical Papers of the international conference on computer aided design, Santa Clara, November 1991, pp 106-110.
[eijn92]
J.T.J. van Eijndhoven and L. Stok, "A data flow graph exchange Standard", Proceedings of the European Conference on design automation, pp. 193-199, Brussels, March 1992
[dewi85]
Dewilde, P, Deprettere, E, Nouta, R,"parallel and pipelined VLSI implementations of signal processing algorithms, pp 258-264, Prentice Hall, Englewood Cliffs, 1985
[mall90]
Mallon. D.J., Denyer, P.B.,"A new approach to pipeline optimization", proceedings of the European conference on design automation, pp. 83-88, Glasgow, March 1990, IEEE Computer Society Press
[the93]
Theeuwen, J.F.M., Arts, H.M.A.M. Arts, Eindhoven van , J.T.J., Sleuters, H.J.H., Wijdeven, J.H.P.,"Module generation in an architectural synthesis environment" pp. 359-371, , IFIP Transactions: Synthesis for control dominated circuits, North Holland, 1993