Design of an Address Generator

Reiner W. Hartenstein, Helmut Reinig, Markus Weber

Department of Computer Science, University of Kaiserslautern,
P.O. Box 3049, W-6750 Kaiserslautern, Germany,
Phone: ++631 205 2606, Fax: ++631 205 2640,
e-mail: abakus@informatik.uni-kl.de

Abstract

This paper describes our experiences with the hardware description language Verilog during the development of the Xputer prototype. At first it introduces the novel non-von Neumann architecture of the Xputer, its need for efficient address generation and the basic structure of the Generic Address Generator. After a short introduction to Verilog, we discuss the problems with this hardware description language and show how to get around using some design restrictions. At the end a outlook on testing and simulation possibilities is given.

I - Introduction

Many applications require the same data manipulations to be performed on a large amount of data. These so called generic algorithms (e.g. systolic algorithms) are normally described as nested loops. A conventional von Neumann computer is not able to optimize such loops. Repetitive address patterns and data manipulations are interpreted on a per instruction basis. Thus address and control overhead causes the inefficiency of the von Neumann architecture. We are developping a new non-von Neumann computer, which we call Xputer. The Xputer is specially designed to reduce this overhead and a much better performance has been achieved for this class of algorithms [3]. The novelty of the Xputer concept requires the evaluation of several design alternatives and possible implementations quite early in the design process. This lead us to describe all the hardware in the high-level hardware description language Verilog at first, to be able to try several design alternatives at low cost and short turn-around time. That way we could improve the Xputer architecture before building the first prototype.

II - The Xputer principle

The clue to speed up generic algorithms is that the Xputer configures the data manipulations within the inner loop into one complex operator. The required operands are addressed under hardware control, which is possible because the regular addressing patterns allow a parametrized computation. This is very efficient for generic algorithms but inefficient for other applications on the other hand.

That way an algorithm is divided into two parts. All data manipulations are translated into one

complex operator, which is mapped onto the reconfigurable structure of the rALU (reconfigurable ALU). Thus we achieve a fine grain parallelism which speeds up data manipulations compared to sequential execution on a von Neumann processor. Both the control flow to establish nested loops and the address computations are transformed into parameter sets for Generic Address Generators (GAGs). This helps us to make use of the regularities in the address patterns to omit addressing overhead during run time. The rALU configuration and GAG parameters are computed at compile time and loaded once into rALU and GAGs. Then the algorithm runs as fast as the data can be provided.

II.1 - Flexible and Fast Address Computation

Crucial to the Xputer principle is the hardwired generation of long and complex address sequences from one parameter set, because every reconfiguration of the GAGs requires valuable memory cycles to select and provide the parameters. Secondly, the addresses have to be computed as fast as the data can be accessed in memory to prevent a slow-down induced by the GAGs. With the existing technology it is possible to design such a special address generation unit. We call this unit Generic Address Generator (GAG). It is able to compute about 4 billions of different address patterns from a relatively small parameter set and so covers most generic algorithms. The GAG also allows data-dependent address generation to provide universality and very fast computation to the Xputer.

II.2 - A short overview of the MoM 3 system

The GAG shall be included in the new Xputer prototype, the Map Oriented Machine 3 (MoM 3). The MoM consists of three main parts. The micro Instruction Sequencer (μIS), up to eight GAGs and the rALU Operators (see figure 1.1).

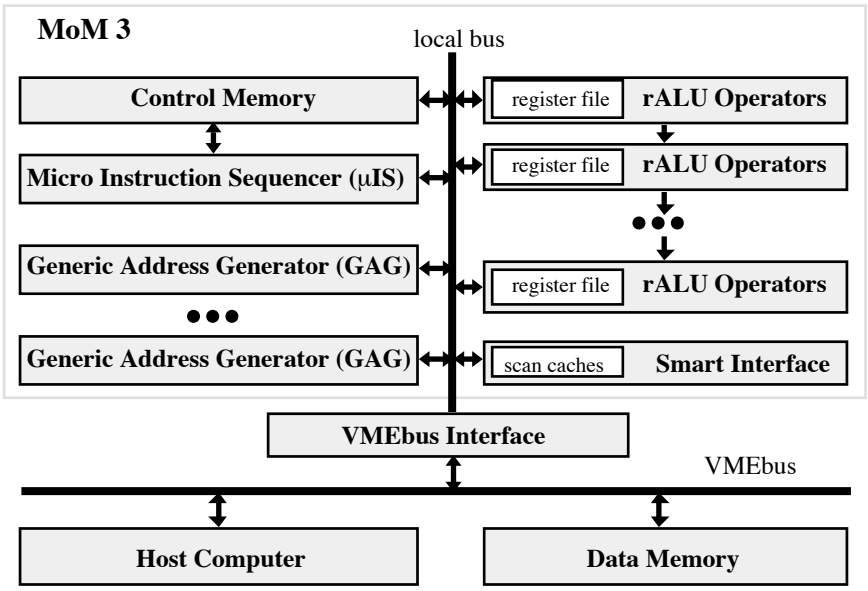

Figure 1.1: The MoM 3 architecture

All components are connected by a local bus and can access the data memory of a host computer via a VMEbus Interface. The μIS is a normal von Neumann type sequencer, which supplies the GAGs with parameters and configures the rALU. After the GAGs and the rALU are configured, the μIS passes control to the GAGs. These compute the addresses for the rALU in parallel. When the GAGs have finished their patterns they return control to the μIS. So with a sequence of different address patterns and with complex operations even a large program may be executed.

II.3 - A short overview of the GAG

To compute the complex address patterns for the movement of the hardware window (cache) over the memory, each GAG is divided in three major parts (see figure 1.2). The Reference Address Generator (RAG) and the Memory Address Generator (MAG) are two stages of a pipeline, while the Control Logic operates in parallel to both of them. That way we achieve a high performance address generation. Since all three parts work in parallel, the speed of the address sequence is indeed mainly limited by the performance of the VMEbus and the GAG's interface to the local bus.

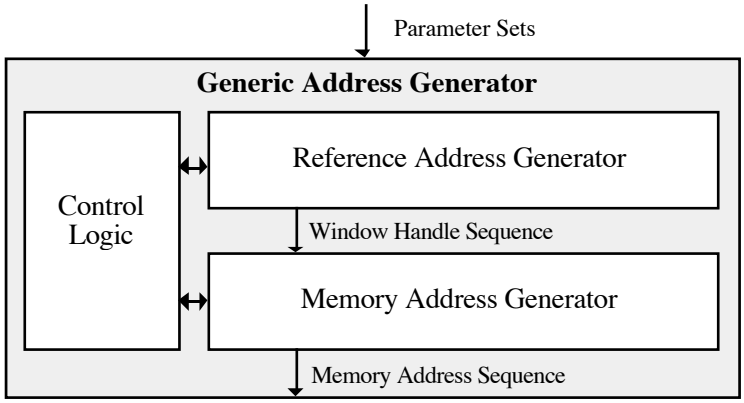

Figure 1.2: The Generic Address Generator

The Reference Address Generator (RAG) computes the address of lower-left corner, the window handle, of the cache. This two-dimensional address is then passed to the Memory Address Generator (MAG). The MAG generates the actually needed memory addresses for the update of the cache contents. It is also programmable to avoid all unnecessary data transfers. The Control Logic watches for segment violations and termination conditions. If either of these happen, it stops all GAGs and returns control to the μIS.

III - Demands on a Hardware Description Language

As shown above the GAG is a very complex circuit. Especially the high parallelism and the synchronisation problems resulting thereof ask for a high-level hardware description language. The language has to provide the facility to describe and simulate the basic concepts of synchronisation and communication between modules in detail. It must enable the designer to simulate the functionality of the GAG while developping it. That means the HDL must allow a mixed description of the future hardware. While the GAG itself is descriped at comparably low level, the additional modules of the Xputer system, which are needed for simulation, like main memory, μIS, rALU ..., are described at a high behavioural level. Verilog puts all these abilities to the designers disposal [1,2].

III.1 - The Language Verilog

The hardware description language Verilog is offered as a part of the Cadence Design Framework. In its syntax and partly in its semantic it is similar to the programming language C. Therefore it is easy and quickly to learn. Verilog provides not only modules, procedures and other high level constructs, such as the case- or if-then-else-statement, but also complex mechanisms for synchronization, like functions to create and wait for user-defined events. On such high level we can specify and test the functionality of the modules quickly, without bothering about the future hardware implementation in detail. Simulation is supported by textual as well as by very comfortable graphical tools. Since Verilog is event-driven, the simulation is very fast and also capable to manage large designs in a short time. Our experience shows, that Verilog provides rapid prototyping, such that even a novice can describe the behaviour of a module and test basic

interaction concepts, like busprotocols and synchronization, very quickly. Mixed descriptions are possible, too. It is possible to describe a peripheral module for example on the functional level, whereas the major circuit is described on gate level to obtain timing simulations.

III.2 - Problems with Verilog

Verilog especially supports behavioural modelling. Low level description is possible, but causes some problems. A disadvantage of Verilog is the fact, that it is intended to describe only synchronous circuits. If you want to design an asynchronous subcircuit without using the event mechanisms, Verilog doesn't force any signal delays. This is a common source of errors. Furthermore Verilog doesn't include a library with simulation data of real standard cells. It is the job of the designer to choose realistic delays for gates, wires etc. Therefore only simulation results on functionality are reliable. All other results like timing checks have to be a coarse assumption. Such a simulation additionally would require a low level description using only language elements, which are available as standard cells. But such low level descriptions are not advisable, because the large design effort is lost, due to the lack of a tool to generate netlists for the Cadence Design Framework thereof. At least such tools are not part of the Eurochip Design Action software and sold at unaffordable prices otherwise. Furthermore the Cadence Design Framework includes simulation tools as well. These tools are based on a standard cell library and allow more detailed and realistic testing. So using Verilog beneath the behavioural level (for functional verification) is not advisable.

IV - Describing the Generic Address Generator

Because there is no tool to port Verilog data to the Cadence Design Framework, we had to do this manually. To simplify this task, we obeyed some hard restrictions which abstraction level and what language elements to use in the description of the GAG (Note that this does not apply to modules, which are described at high behavioural level to complete the simulation environment). These restrictions had the major intention to describe only realistic hardware components on behavioural level. So we combined two contradicting goals. At one hand we were still able to do comfortable simulation and verification of basic concepts and even change and improve them. On the other hand the Verilog description remained easily portable to the Cadence Design Framework by replacing language constructs by the appropriate sets of standard cells. Some of the restrictions are listed below.

IV.1 - Design Restrictions for the GAG

We didn't use any control statements (e.g. For, While), because the translation to Final State Machines (FSM) and signal wires in the hardware would be too error-prone. Initial statements, which are executed only once, were only used to describe the stimuli. This is equivalent to the future hardware where there will be no automatic initialisation, too. The Case-Statement was only used to describe a multiplexer, which means that only assignments to one register (no control flow) could be described in one Case-Statement. These restrictions lead to modules that are divided into two parts. A control flow part, described by a FSM, and a data flow part, which describes the data paths controlled by the output of the FSM. The states of the FSMs were described with mnemonics, without bothering about the binary representation and transition functions. Later on the only problem is to convert the Final State Machines to the Cadence Design Framework by implementing them with Flip-Flops and combinational logic. The data paths are easy to convert from a register transfer level description to standard cells.

IV.2 - Results achieved during Simulation

Using the high level decription we could easily verify the functional correctness of the single modules. Furthermore we tested the synchronization and the interaction of the modules of the GAG itself and the interaction of the GAG with its peripheral components. Additionally we simulated several conceptional alternatives and so improved our busprotocol and did some first timing estimations. With statistical analysis we checked for the modules spending most time

blocking the others and optimized these parts to get a major improvement in performance.

V - Conclusions

Many applications use generic algorithms. The Xputer is especially designed for such problems and so provides great performance improvements. Essential to the efficiency of the Xputer is a very flexible and powerful address generation unit. The complexity of this Generic Address Generator asks for a high level hardware description language with comfortable and fast simulation tools. Verilog fulfills all these demands. It is similar to the C programming language and contains all high level constructs. But there is no tool to convert Verilog descriptions to real hardware designs. So we had to give ourselves hard restrictions in using Verilog. The restrictions lead us to a high level behavioral description, which was still close to the future hardware. With this design we checked the functionality and the concepts of the whole Xputer system. Statistical and timing estimations made with Verilog gave us the ability to optimize our design efficiently.

All in all we made good experiences with Verilog. It is a good tool for rapid prototyping and for testing the basic concepts and the function of a hardware module. The simulation is fast, comfortable and easy to do. But a great disadvantage is the fact that its results are not portable to the Cadence Design Framework.

References

[1] D. E. Thomas, P Morby: The Verilog Hardware Description Language; Kluwer Academic Publishers, 1991.

[2] N.N. (Gateway): Verilog-XL Reference Manual, Version 1.5a; Gateway Design Automation Corporation, Lowell, USA, 1989.

[3| R. W. Hartenstein, A. Hirschbiel, M. Riedmueller, K. Schmidt, M. Weber: A Novel Pradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware; Future Generation Computer Systems, Elsevier Science Publishers, North-Holland, 1990.