A hardware implementation of the UniSim pipeline model

Radu Andrei Stefan and Koen Bertels
Computer Engineering Laboratory
Electrical Engineering, Mathematics and Computer Science Department
Delft University of Technology
Mekelweg 4, 2628 CD, Delft, The Netherlands
{R.A.Stefan, K.L.M.Bertels}@tudelft.nl

Abstract—

Design space exploration is a component of the product optimization process that confronts the architect with the task of evaluating a large number of design choices. In order to achieve this goal software tools are used to automatically determine the benefits and drawbacks of each proposed implementation. In the field of processor architecture one such tool is UNISIM. The UNISIM environment provides a framework for design space exploration in the form of libraries and tools for evaluating designs through cycle-accurate simulation. UNISIM emphasizes modularity and reusability of modules, by defining a standard interface for inter-module communication.

The simulator is capable of providing accurate results in terms of number of cycles required for the execution of a specific program, however it does not offer any information regarding the power consumption, the occupied silicon area or the clock rate the design is capable of. In order to overcome these limitations, we attempt to produce in this study a hardware implementation that closely follows the structure of the simulator at source code level. In large part, this translation process involved only a change in syntax from the C code to Verilog.

Our study shows that our objective can be achieved at moderate implementation cost while preserving all the features of the original software simulation. We emphasize modularity and a standardized control flow for inter-module communication, which are also the defining characteristics of the software implementation.

Keywords— **Design Space Exploration, UNISIM Simulator, Hardware Description Language, Verilog, FPGA.**

I. Introduction

When developing processor architectures, engineers often use simulators to determine the efficiency the designed processors have in executing benchmark programs. Most often, the accuracy required for these simulations is very high, in particular, in order to correctly determine the scheduling of instructions within the execution pipeline, a cycle-accurate simulator is needed.

Traditionally, simulators were developed in-house by the research groups that were also developing the simulated architectures, but implementing those simulators led to the unnecessary duplication of a large amount of work. The solution to this problem was to develop libraries, frameworks or even new languages that can be used across multiple simulators.

The libraries and frameworks often evolved to cover a broader range of scenarios and circuits they can describe. This has inevitably forced simulators to resemble more closely in structure the designs they model, sometimes to the extent of providing a description that is equivalent to the hardware description, effectively eliminating the boundary between the two, as it is the case with SystemC [6].

The simulation platforms often emphasize modularity and reusability among the advantages they provide. Indeed, thanks to well defined, carefully designed interfaces, it was shown it is possible to combine in a single design, modules developed by different research groups in different parts of the world [1]. Although modularity is not a concept foreign to hardware design, a further step taken by simulation platforms like Liberty [11], further developed into UNISIM [1], was to distribute the logic controlling the flow of data into the modules themselves. This approach would allow modules to perform their function without the need of an external, centralized control, which is highly dependent on the specific architecture, and hence non-reusable.

In our study, we attempt to use this concept of decentralized control in an actual hardware implementation, however, we do not attempt to synthesize the simulation code directly. We acknowledge the fact that having a common source base for both synthesis and efficient simulation would provide an advantage by reducing the development effort involved, however the tools necessary for this automatic translation of C

code to hardware are still immature. Hence, our approach consists of manually creating a separate hardware implementation that preserves the features of its software counterpart with regard to modularity and reusability, but more importantly, it preserves the distributed control mechanism of UNISIM.

II. Related work

Over time, numerous simulation platforms have been developed, many of them targeted at microarchitectural development [2], [5], [7], [11], [1]. Of these we have chosen UNISIM, a recently developed emerging simulation platform as a starting point in our research. We found UNISIM to be an suitable choice because of the way it closely mimics the hardware implementation.

The origins of the UNISIM framework are found in the Liberty Simulation Environment [11] developed at the Princeton University, and its main characteristics, in particular the three way handshake that we will describe later on in this study is inherited from LSE.

UNISIM, as well as Liberty are platforms for creating cycle-accurate simulations of devices, but by themselves do not provide any information about the power consumption or the clock rate of the devices being modeled. The problem has been addressed by using extensions to the platform and external power estimators [3], [4], [13]. In all these cases, the power is only estimated, based on existing models and measurements on similar circuits. Timing is also estimated in Justice [3] by taking into account the wire lengths generated by a simple floorplanner.

Our approach is a radically different one. We aim to produce a synthesizable model, from which the information on power consumption can be extracted either by physical simulation or using the power models of the logic blocks used in the design. Performance figures can also be obtained directly from the synthesis tools.

The advantage we seek is that of obtaining in the hardware implementation the same capability that the software simulator has, that of combining modules without the need of rewriting control code.

III. The UNISIM Environment

Pipelines are a ubiquitous feature of modern architectures, and UNISIM offers a powerful mechanism for modeling them. Under normal operation, data is partially processed in each stage of the pipeline, and then forwarded to the next stage for further processing. A stall can occur when one of the stages is not able to accept incoming data for one or more cycles. As the buffers of previous stages also become full, the stall will propagate upstream.

An effect that occurs simultaneously is that the stage producing the stall, probably being in a busy state due to a longer operation, is also incapable of producing the data required by the following stage. As data is propagated in the pipeline on each stage, this 'lack of data' is also propagated and is called a bubble.

In UNISIM, modules are required to produce a value into each output port, each cycle. When a module is not able to produce any useful data, the output can be marked as 'nothing'. If the module is part of a pipeline, producing 'nothing' on its output is equivalent to inserting a bubble.

In practice, the 'nothing' flag can be modeled as an additional one-bit signal in the interface between modules. Alternatively, the negated 'something' signal has been used in the hardware implementation. Another signal, named 'accept', is transmitted the opposite direction, for the purpose of modeling stalls. It allows receiver to inform the source whether it accepts data during the current cycle or not.

A third signal called 'enable' may be used by modules having multiple outputs. Consider the case of a sender that can provide data to multiple receivers, but to no more than one during a single cycle. The 'enable' signal provides the sender module with the means of informing receivers whether to use or not the data it has provided during the current cycle.

Together, the three signals form a three-way-handshake mechanism, commonly found in asynchronous systems or communication networks.

The distributed control produces an overhead in terms of implementation effort as the control has to be replicated in each module, however, the negotiation scheme is simple and can be reused in multiple designs.

Consider the pipeline stage in figure 1. The module produces useful data on its output if it has data, received during previous stages, that it has not delivered yet.

The module accepts incoming data if either it has a free buffer to store the data, or it can free a slot in its buffers by delivering data to the next module. As a protocol convention, incoming data is not accepted, unless the 'something' signal is asserted. Because the module is a simple pipeline stage with a single output, data is always delivered when accepted.

Let us consider the cost of implementing the de-

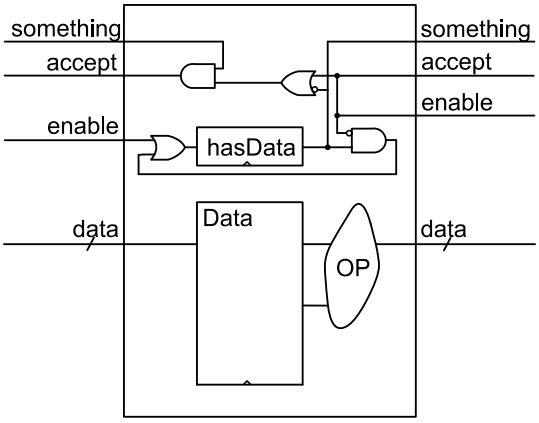

Fig. 1. One pipeline stage

scribed model in hardware. In figure 2 can be observed that the distributed control logic forms a combinational chain that spans the entire pipeline, adding two logic levels for each stage.

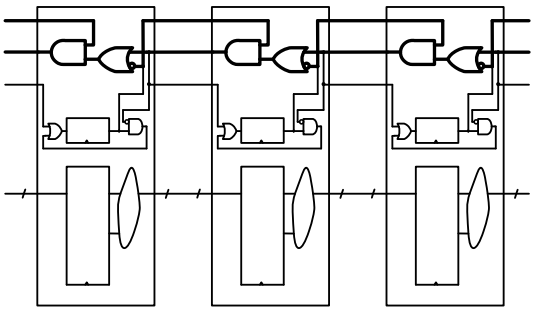

Fig. 2. Combinational logic chain through the pipeline control logic

However, in practice this does not represent a problem because, when taken separately the control logic has a small number of inputs, one for each pipeline stage in addition to the external signals received by the first and last stage as shown in figure 3. Functions with few inputs can be easily optimized by synthesizer tools, or they can be implemented in look-up tables. Techniques exist for breaking the combinational chain in very long pipelines but their discussion is beyond the scope of this article.

IV. Migration toward hardware

In large part, our approach consists of manually translating the simulation code into the Verilog hardware description language. Wherever possible, we preserve module and signal names as well as the constructs used in the simulation code. In the future, we may consider partially automating this process.

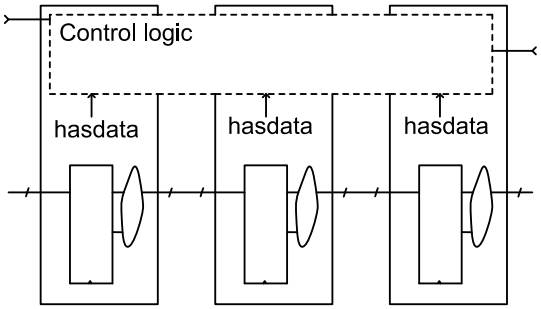

Fig. 3. Isolated control logic

A. Programming language constructs

Although obviously the Verilog language does not provide the same diversity of constructs that are available in the C++ language, in which the software simulator is written, most of the constructs that were actually used have their equivalents. Conditional statements and loops can be found in similar forms in both languages. As an observation, the actual usage of those constructs requires more discipline when describing hardware, and a better understanding of the way the compiler operates.

A notable feature that was absent in Verilog was the ability to define complex data types like structures. A work-around for this issue was found in declaring the data types as simple arrays of bits and using preprocessor directives to assign names to address ranges within these arrays of bits in a similar way member names are used to address the particular fields within the structure. Preprocessor directives were also used to define constants. The disadvantage of this approach is that all member names and constants use the same global namespace.

A comparison of the two languages is exemplified in figure 4 and 5.

B. Combinational logic

One feature inherited from previous generations of HDLs as well as SystemC is the sensitivity list. The programmer is thus required to specify which are the inputs of the combinational blocks.

In UNISIM, the sensitivity list is less strict than in an HDL. It is not necessary to add to the list all signals the output depends on, as long as the programmer ensures the signals are known when they are needed. The 'known()' function is provided for checking whether a specific signal has been generated during the current clock cycle.

By comparison, in the current version of Verilog, the need for sensitivity lists has been entirely elimi-

```
const unsigned int OP_ADD=0x01;
const unsigned int OP_SUB=0x01;

struct instruction
{
   unsigned int adr;
   unsigned int opcode;
};

...
instruction i1;
i1.adr=OP_ADD;
...
if (i1.adr==OP_ADD)
{
   c=a+b;
}
```

Fig. 4. C++ code for structure definition and usage

```
‘define OP_ADD 32’h01
‘define OP_SUB 32’h02

‘define INSTRUCTION [63:0]
‘define OPCODE [31:0]
‘define ADR [63:32]

...
reg ‘INSTRUCTION i1;
i1‘ADR=OP_ADD;
...
if (i1‘ADR==OP_ADD)
{
   c=a+b;
}
```

Fig. 5. Verilog code equivalent to the C structure definitions

nated, as the simulator and synthesizer tools are able to automatically determine which are the signals the output depends on.

C. Registers and clock cycle

For convenience, in the UNISIM model, the clock cycle has been split in two phases, which have been formally associated to the two edges of the clock. During the first phase, the rising edge of the clock, signals are propagated between modules, while during the second phase the internal state of modules is updated.

This separation, although useful for a better structuring of the simulation code, is entirely artificial and has been eliminated in the hardware implementation. Between modules, the signals are propagated through combinational logic, while the internal state is updated on the rising edge of the clock.

D. The great wall of memory abstraction

One of the weak points of hardware synthesizers has been the ability to deal with memory abstractions. In time, the tools have evolved, and in our experiments, they have been able to correctly recognize all architecture elements, including the register bank, memories and related logic without any human intervention. We have still to determine whether the design requires further optimization for the components where the synthesizer tool may not have made the best choices.

V. Experimental results

For our experiments we have used a model of the DLX processor described by Patterson and Hennessy [9]. The software simulator for the described architecture is already part of UNISIM.

We have implemented our design using the Verilog hardware description language, and tested the design using the open-source Icarus Verilog [12] simulation software. For verification, we ran several programs written in assembly language, programs that use the entire range of operations offered by the architecture: memory read and write operations, arithmetic operations, conditional jumps, as well as features like register dependency checking.

Despite extensive verification, the results in this study shall be regarded as preliminary, especially the performance figures generated by the synthesis tools.

We have synthesized the design using the freely-available Xilinx ISE [8], version 9.2i, for the Virtex-4 platform, obtaining running frequencies of the synthesized circuit of up to 166.8 MHz, which corresponds to a propagation delay of 5.9 ns. For comparison, a single 32 bit adder-subtracter like the one found in the execution stage of the DLX pipeline, when synthesized on the same platform and with the same speed grade, presents a propagation delay of 2.36 ns, while a memory module, like the one found in two pipeline stages has a delay of 2.876 ns.

The area occupied by the design was 690 slices, which amounts to 6% of the total number of slices available on the device. However, no attempt was made so far to optimize the area, and the synthesis tools were in fact configured to improve speed in the detriment of the amount of hardware resources used.

Another interesting point of comparison is the speed of simulating the hardware implementation compared to the speed of the original software simulation. Using Icarus Verilog the simulation speed was unfortunately very low, more than three orders of magnitude below the speed of the software simulation, however, according to [10] the Icarus Verilog simulator is not known as one of the fastest simulators available. In the future, we plan to evaluate other simulators as well.

The advantage of producing a synthesizable implementation is that instead of using a software simulator, the design can be uploaded on a reconfigurable device and tested directly, with greatly improved speed. So far, we have not tested our design on a real FPGA board, as we did not have one available with the same model that was used for synthesis, and problems were encountered when trying to synthesize for a Virtex-2 target.

VI. Future directions of research

Our work so far consists of a proof-of-concept translation to hardware of simple simulator written in the UNISIM environment. This translation preserves the features that would help provide module reusability, thus enforcing a discipline in the way the hardware modules are described.

Although our goal is achieved in terms of preserving the interface between modules that is found in the software simulation, but the impact of this choice on the quality of the designed hardware has yet to be analyzed.

We intend to further advance our study by modeling more complex architectures, in order to determine the new challenges imposed by larger designs and give a more clear view of the advantages and disadvantages our approach may have on the design process, and on the final result of the design process.

Ultimately, we can only consider our goal attained when a diversity of modules, possibly developed by third parties, can be seamlessly integrated into an existing design, and new designs can be created using the library of existing modules.

Acknowledgements

This work was supported by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6).

References

[1] David August, Jonathan Chang, Sylvain Girbal, Daniel Garcia-Perez, Gilles Mouchard, David Penry, Olivier Temam, and Neil Vachharajani. Unisim: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Computer Architecture Letters*, PP:1–1, 2007.

[2] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[3] N. P. Carter and A. Hussain. Modeling wire delay, area, power, and performance in a simulation infrastructure. *IBM J. Res. Dev.*, 50(2/3):311–319, 2006.

[4] Ashutosh Dhodapkar, Chee How Lim, George Cai, and W. Robert Daasch. Tem2p2est: A thermal enabled multimodel power/performance estimator. In *PACS '00: Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 112–125, London, UK, 2001. Springer-Verlag.

[5] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[6] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[7] Christopher J. Hughes, Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. Rsim: Simulating shared-memory multiprocessors with ilp processors. *Computer*, 35(2):40–49, 2002.

[8] Xilinx Inc. *Synthesis and Simulation Design Guide*, 2007.

[9] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[10] Wilson Snyder. Verilog simulation benchmarks. http://www.veripool.com/verilog_sim_benchmarks.html.

[11] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. The liberty simulation environment, version 1.0. *SIGMETRICS Perform. Eval. Rev.*, 31(4):19–24, 2004.

[12] Stephen Williams. Icarus verilog. http://www.icarus.com/eda/verilog/.

[13] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.