Lessons Learned from Using Superlog, SystemVerilog’s Predecessor

David Slogsnat, Patrick R. Haspel, Ulrich Brüning
Department of Computer Science and Engineering
University of Mannheim, 68131 Mannheim, Germany

July 24, 2003

Abstract

For high-level simulations of complex systems, classical hardware description languages like Verilog are inefficient. On the other hand, C-based languages and libraries often lack ease of use when it comes to describing concurrency and timing behaviour. Superlog, a language extension to Verilog, closes this gap. This paper describes how Superlog, the SystemVerilog predecessor, was used to implement a shared-memory multiprocessor simulation environment. Superlog allows transaction level simulations, in contrast to Verilog’s RT level methodology. This environment, named GEMS (Generic Environment for Multiprocessor Simulations), is used to explore the design space for shared-memory multiprocessor systems. Therefore, a large number of the simulated system’s properties can be varied. Examples are the number of processors, clock speeds of the system components, cache sizes, interconnect bandwidth and delay and so on. To analyze the performance of applications on the simulated system, traces of their memory accesses drive the simulation. There are two alternatives to model the memory subsystem: a high-level model and, as a very precise model for the memory subsystem, a memory verification IP in combination with a memory controller implementation IP. It is almost impossible to implement such a complex system with reasonable effort in Verilog, especially with respect to extensibility, modifiability and readability of the code. This paper explains how Superlog simplifies the modeling of such complex systems drastically.

1 Introduction

Cluster Computing is a very promising trend in high performance computing. By interconnecting off-the-shelf PC’s with high-speed system area interconnects (SANs) it is possible to create low-cost parallel computers. Currently, the nodes in most clusters are dual processor machines. To analyze to behaviour of SMP nodes with a higher number of processors, we developed a special simulation environment: GEMS (Generic Environment for Multiprocessor Simulations). We use GEMS to analyze the design space for medium scale SMP systems. Since memory bandwidth and latency are usually the bottlenecks in these systems, the focus within GEMS lies on the processor-memory interconnect and the memory subsystem, particularly on the memory controller and the cache coherence protocol. GEMS has been implemented in Superlog, which is the predecessor of SystemVerilog. Most of the Superlog constructs that we used in our project are identical with the corresponding constructs of the SystemVerilog 3.1 Standard [Accb] from
June 2003. However, build in queues and dynamic memory management of Superlog are not part of SystemVerilog. The status of the SystemVerilog development can be tracked on the Accellera Homepage [Acca]. The remainder of this paper is organized as follows. Chapter 2 briefly describes the system that is simulated by GEMS. In Chapter 3 we explain our motivation to use Superlog for the implementation of GEMS. Chapter 4 reports our experiences with Superlog in our project. Chapter 5 concludes this paper.

2 The Simulated System

Figure 1 shows a general block diagram of the simulated node. At its heart is the system interconnect, connecting all components in the system. The system interconnect is implemented as a full crossbar. Since we assume the node to be part of a cluster system, a future extension to it will connect one or more network interfaces (NICs) to the SAN, using the same interface as the other components. A directory-based MOESI-like cache coherence protocol is used to maintain cache coherence within the node. Since our aim is the exploration of the design space, the system is highly configurable. The configuration parameters range from basic properties as memory sizes and clock speeds to very detailed characteristics as the latencies for important hardware operations. Our aim is to analyze the performance of real-world applications on the system. The focus lies especially on numerical applications, like smoothed particle hydrodynamics [Mon92] for example. We assume that on a shared-memory system, the execution time will typically be dominated by memory accesses rather than by actual computation. Therefore, we only simulate memory accesses and not the computation. Simulations can be either trace-driven, or they can be based on a stochastical model. For the trace-driven simulation, memory access trace files of the specific applications must be provided as input to the simulated CPU. To perform simulations using the stochastical model, the simulated CPUs generate memory accesses based on stochastical methods. In this case, the behaviour of the CPUs can be adjusted by various parameters. These parameters can be modified at simulation run-time, which allows to model the different computational phases of a numerical application. More detailed information about GEMS and the simulated system can be found in [Slo02].

3 Superlog and Possible Alternatives

There are already a few projects that provide an environment for computer architecture simulations. But none of those environments could be used “as is” for our purposes, they all require changes to the simulated system that require extensive modifications to the respective code. For example, there is the newly developed ML-RSIM [ml], which is based on the RSIM simulator (RICE Simulator for ILP Multiprocessors [rsi]). ML-RSIM is written in C++. Modules to be included into the simulation, for example a model for a cache, are implemented as C++ objects. Their behavior is not described using plain C++, but using functionality provided by an event-driven simulation library. Therefore, tools like RSIM are reportedly quite complex to handle and require a relatively long period of time to get used to.

Superlog is a relatively new language. It is based on Verilog, and actually includes Verilog 2001 as a subset, with all of Verilog’s features like the PLI for example. It includes also features know from programming languages, like pointers or classes, and features that aren’t available in HDLs or programming languages, but in languages that are used for testbench creation.
Therefore, by using Superlog it should be possible to write a simulation environment that uses high-level constructs that are native to the language, without the need of a special simulation library. Particularly, Superlog allows for transaction level simulations, as described in [GLMS02], in contrast to Verilog’s RT level methodology.

SystemC is another language with a similar intention as Superlog. However, SystemC is based on C++, adding features known from HDL languages. These features are not included as new language keywords and functionality, but as C++ class libraries. This has the disadvantage that SystemC lacks of the simplicity known from HDLs when it comes to describing concurrency. As a result, Superlog was chosen to be the basis for GEMS for the following reasons: It should be much easier to learn Superlog than to learn how to use and modify a C-based simulation environment. Since this project startet as a diploma thesis, which is very limited in time, a short learning phase was important. The compatibility to Verilog allows the integration of existing Verilog components without any modifications. For example, RTL code components can be included, allowing a smooth transition from high-level to RTL simulations.

4 The GEMS Environment

In the following subchapters it is described how the Superlog features significantly helped to develop GEMS.

4.1 C- and Userdefined Datatypes

Superlog incorporates all features known from C. This means that it also contains all C datatypes, and userdefined data types. User-defined data types can be declared using the keyword typedef just like in C. To construct new data types, the keywords enum and struct can be used. A
very important data structure in GEMS are the transactions that are send via the system interconnect. In Verilog, a transaction would be represented by the values stored in an array of registers. In Superlog, transactions can be represented by a single, user-defined data type. The following is the (slightly simplified) transaction data type used in GEMS:

```c
//struct for transactions
typedef struct {
  int receiver_id; //device ID of receiver
  int sender_id; //device ID of sender
  longint address; //memory address
  int unsigned req_id; //request ID
  instruction instr; //instruction
  tag_type tag; //tag
  int data_length; //length of data in bytes
  moesi_state moesi //Cache coherence protocol state information
} transaction;
```

As can be seen, other user-defined data types are used in the struct transaction: instruction, tag_type and moesi_state are all enumeration types. An enumeration is created as follows:

```c
//moesi state data type
typedef enum{M, O, E, S, I} moesi_state;
```

Now, a transaction can be stored in a single variable of type transaction. The different fields of a transaction can be accessed using the "." operator:

```c
//create 2 variables of type transaction
transaction my_transaction, my_transaction2;

//set sender_id, address and moesi fields of my_transaction
my_transaction.sender_id = 0;
my_transaction.address = 'h1234;
my_transaction.moesi = M;
```

### 4.2 Pointers and Dynamic Memory Management

Transactions are used for communication between all modules in the simulation environment. When transactions move between modules, only pointers to the corresponding transaction are passed between the modules, the transaction itself does not have to be copied or moved. Passing pointers is therefore much more efficient than passing whole transaction structs. The following code example demonstrates the use of pointers and dynamic memory management in Superlog:

```c
// create pointers to type transaction tract_ptr, tract_ptr2
ref transaction tract_ptr, tract_ptr2;
// allocate memory for a new transaction
```
tract_ptr = $alloc(transaction);
// set sender_id and address of the new transaction
tract->sender_id = 1;
tract->address = 'hFFFF;
// let tract_ptr2 point to the same transaction
tract_ptr2 = tract_ptr;
// let tract_ptr2 point to my_transaction
tract_ptr2 = ref my_transaction;
// de-allocate memory pointed to by tract_ptr
$delete(tract_ptr);

There are two dynamic memory modes: save mode and fast mode. In save mode, it is assured that pointers always point to a valid address or null, and that no memory leak exists. The fast mode does not perform those checks. When using fast mode, pointer handling and dynamic memory management are just like in C. The save memory mode was a great help during the development and debugging of GEMS. Basically, the save mode does two things: first, it ensures that there is always at least one valid pointer to every dynamically allocated block of memory. If the last pointer to such a block is deleted, a warning is generated. This means that memory leaks can be avoided. For the simulation environment this means that transactions cannot get lost: transactions are moved by moving their pointers. A transaction could get lost in case a pointer is accidentally overwritten. In this case, a memory leak would occur, generating a warning. So, it can be ensured that no transactions are lost by just using the safe mode in a very efficient way. The second feature of safe mode concerns the de-allocation of memory: when a block of memory is de-allocated, all pointers that are pointing to this block are set to null. Optionally, a warning is generated for every such pointer. This feature ensures that pointers always point to a valid location or to null.

4.3 Interfaces

The system interconnect within GEMS is implemented as a Superlog interface instead of a conglomeration of modules and wires. Superlog introduces the interface construct to provide a more abstract level of communication than Verilog’s wires. On the lowest level of abstraction, interfaces are only wires. On the highest level, interfaces are rather classes containing constants, variables, functions and tasks. Interfaces are especially useful to describe buses or interconnection networks on system level, or to build functional or even full bus models. Especially in combination with assertions, they are also useful to check whether implementations are meeting timing and protocol requirements and to do performance analysis. The system interconnect in GEMS is implemented in an interface. It allows a number of devices to be attached to it. Devices can exchange transactions with each other using the interconnect. The interconnect provides a number of channels which can be either virtual or physical channels.
//the interface declaration
interface interconnect (input reg clk, input reg reset);
    parameter int DEVICES = 1;
    parameter int CHANNELS = 2;

    //ports for devices attached to interconnection network
    modport device ( input stop, // forbid to send transaction
                    input valid, // flag showing whether valid data
                    import write, // write transaction to interconnect
                    import read // read transaction from interconnect
    );

Figure 2: The device ports of the system interconnect

The write and read tasks can be called by the devices at any time, but since the system interconnect is a synchronous device, they will wait for the next positive edge of the interconnect clock before they execute. Thereafter, the tasks return to their callers. The tasks require the following parameters: the pointer to the transaction to be sent, the ID of the calling device, and the channel. A status variable shows whether the task has succeeded in writing/or reading (status==0) or not (status==1).

//declarations of the read and write tasks:
//write a transaction for transmission into the interconnect
  task automatic write(output status, input ref transaction tract, input int device_id, input int channel);

//read a transmitted transaction from the interconnect
  task automatic read( output int status, output ref transaction tract, input int device_id, input int channel);

4.4 The “Automatic” Keyword

In Superlog, variables can be declared either static or automatic. A static variable is created upon instantiation and never de-allocated, while for an automatic variable, storage in the stack
is reserved upon entry of the task, function or block in which it is declared, and de-allocated on exit. Thus, a static variable is like a Verilog variable, while an automatic variable is like a C variable. By default, variables are static if not otherwise declared. Tasks and functions can be declared static or automatic too. In an automatic task or function, all local variables are automatic, unless they are declared static. In tasks or functions, static variables are shared among all instances of the task/function (as in Verilog). Automatic variables are not shared, which allows them to run without interferences (as in C). Automatic variables allow for example recursive function calls. Almost all tasks and functions in GEMS are automatic, since many tasks and functions may be called several times in the same time step. The interface’s read and write tasks, mentioned in the previous chapter, are a good example: they will most likely be called from a number of devices in the same time step.

4.5 Gathering Performance Data

To obtain performance characteristics of the simulated system, the first step is to collect the required raw data during the simulation run. A successive processing step will turn this raw data into a more general measure that describes the overall system performance. For example, this processing step could compute the average CPU performance using the performance data for the individual CPUs. In GEMS, performance data is collected and processed by a separate statistics interface. Every element of the simulation environment (i.e. CPUs, memories and the system interconnect) is connected to it. The interface provides a number of statistics variables and arrays that have to be updated by the individual modules and interfaces in the simulation environment. After a simulation run, the statistics interface processes this raw data and writes the performance characteristics to a number of ASCII files. The advantage of this text-file format is that it can not only be viewed by using a simple text editor, but it can also easily be imported into programs like MS Excel or gnuplot. We use gnuplot for a graphical representation of the results. This step can be automated very easily using gnuplot scripts: scripts can automatically open the data files, plot defined sets of data, create corresponding labels, and save the plot in a variety of file formats, including Postscript and FrameMaker formats.

4.6 Integrating the Denali Databahn Memory Controller and the MMAV Verification IP

Superlog includes all of Verilog-2001. This allows for the integration of exisiting Verilog modules, including those that make use of the PLI. This feature has been used in GEMS for the following: To have a very precise model for the memory subsystem, GEMS offers in addition to a high-level model a very detailed memory model. This model consists of two IP cores from Denali Software Inc [6]: the Denali Databahn memory controller and the Denali MMAV memory verification IP. The Denali Databahn is a highly configurable memory controller that supports all major memory standards, including DDRII-SDRAM. An encrypted Verilog RTL source of the controller has been integrated into the Superlog simulation environment without any problems. The Denali MMAV (Memory Modeler - Advanced Verification) verification IP is an industry standard for memory simulation. MMAV consists of a set of C-models describing the behavior of the different types of memory. Features and timing of a particular memory device are described in SOMA files. SOMA files can be downloaded from the eMemory.com web pages. Just like the Databahn verification IP, MMAV supports a broad variety of ROMs, Flashes, SRAMs and
DRAMs. As with the Databahn memory controller, the MMAV has been integrated into the 
Superlog simulation environment without any major problems.

4.7 Queues as a Special Type of Array

As is the case in high-level designs in general, GEMS makes heavy use of queues. While it 
is possible to use queues in Verilog by including some self-made queue-modules, these queues 
are not really straightforward to use. And they especially lack of polymorphism. Polymorphism 
is a great help since GEMS uses queues for a number of different types of variables, especially 
of userdefined types. Superlog supports queues natively, and they were a great help for the 
development of GEMS. In Superlog, queues are a special type of array, and they are created in 
a similar way. The main difference is that a queue does have a variable size, which is indicated 
by a $:

```
// create the request_queue which contains transactions
ref transaction request_queue[0:$];
```

Superlog provides a number of operators that can be applied on the queue or on elements of 
the queue. In the following are only some examples how queues can be used:

```
// insert my_transaction to the right of the queue
request_queue = {request_queue, my_transaction};

// insert my_transaction after queue element i
request_queue = {request_queue{0:i}, my_transaction, request_queue{i+1:$}}

// remove leftmost item of queue
request_queue = request_queue{1:$};

// get the number of items in the queue
int n = request_queue.$num;
```

4.8 Enhanced Random Data Generation

As mentioned before, simulations in GEMS can be based on stochastic assumptions alternatively 
to the trace-driven model. The stochastic model makes use of a random number generator to 
produce a stream of memory accesses. This process can be controlled by a number of parameters 
that have an direct influence on the produced stream. Therefore, they have to be incorporated 
into the random number generation process. This is done by using Superlog’s random data 
generation capabilities: Superlog replaces the $random in Verilog with the more extensive $rand. 
$rand accepts a number of parameters that control the generation of random data. Using these 
parameters, it is very easy to create for example distributions of specified values, distributions 
of values within a given range and weighted distributions and any combination. To give an 
example, the following is a slightly simplified excerpt from GEMS:

```
// previous declarations/definitions
```
typedef enum { READ, WRITE, CODE_READ} instruction;
instruction instr;

// next instruction is a CODE_READ (with a probability of 50%),
// a READ (25%) or a WRITE (25%)
instr = $rand(.seed(42), weighted_values({ {50, CODE_READ}, {25, READ}, {25, WRITE} }));

In this example, the variable instr is assigned a new value. With a probability of 50%, the new value is CODE_READ, otherwise it is READ or WRITE with a probability of 25% for each. A constant value is used as seed.

5 Conclusions

The simulation environment has been implemented as part of a diploma thesis. The design space evaluation, the implementation and all testing took about 5 months for a single person. Although this is a relatively short time for such a project, GEMS does even exceed the aims that have been set beforehand. Our experiences with the Superlog language are very positive: The integration of all C functionality into Superlog combined with additional high-level constructs makes it a very powerful language for high-level design. We can strongly recommend the use of Superlog for high-level design of complex systems. However, it is not clear if SystemVerilog will be suited for transaction-level design and modeling, since it lacks of pointers and a comparable dynamic memory management functionality. The compatibility to Verilog does indeed allow for a seamless integration of existing Verilog modules and IP cores. Compared to other design languages, Superlog is a very natural way to describe complex systems, especially if the designer is already familiar with HDLs like Verilog or VHDL. Due to the high-level constructs, the readability of the code is relatively high, which is a great benefit during the debugging and verification phase. From the authors experience, it can be said that Superlog is much easier to use than C-based languages as SystemC.

6 Acknowledgments

We want to thank Chris Podger, Dave Kelf and Lee Moore from former Co-Design Software (now Synopsys) for providing us with the Superlog products and the valuable support. Thanks go also to Paul Rowley, Beate Page and Peter Prüller from Denali Software Inc for the excellent cooperation.

References

