ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Simon Jäger

A Simulation Framework for Multiprocessor SoC

Student Thesis SA-2003.15
Winter Term 2002/2003

Tutor: Alexandre Maxiaguine

Supervisor:
Prof. Dr. Lothar Thiele

21.2.2003

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Network Laboratory

Abstract

System-on-Chip designs are the answer to the demand of real time multimedia applications for high performance architectures. Such architectures comprise of multiple processing units, memories and input-output devices interconnected by advanced communication infrastructure. Design exploration is the complex process of finding an optimal mapping of the application to the SoC architecture. In order to reduce design cycles, developers rely on sophisticated evaluation tools. The goal of this project was to build a simulation framework which combines an instruction set simulator with a hardware description language in order to analyze multimedia streaming applications. In this work the SimpleScalar instruction set simulator tool set was integrated into a SystemC hardware simulation environment. With this framework, designers are able to run customized applications on the processing units in their simulated system. In addition, this framework can be used to validate simulation results acquired with analytical evaluation techniques.

Table of Contents

Introduction

Today's embedded real time multimedia applications demand for high performance and flexible hardware architectures. System-on-Chip (SoC) designs made these architectures affordable, since they allow to integrate an entire system on a single die. These systems may comprise of multiple processing units, memories and input-output devices, interconnected by sophisticated communication infrastructure. Specializing these architectures for a given application, allows to achieve even higher performance.

Many different designs must be considered when choosing a specific architecture. Many different alternatives must be evaluated in order to find an optimal mapping of the application to the SoC architecture. Thus, this kind of design space exploration is usually a very complex process and relies on advanced performance evaluation techniques. Since building a real system or an emulation for each interesting variant is not feasible due to the high simulation time involved, performance estimation models are commonly used to predict system properties.

Two classes of performance estimation models exist, those based on analytical methods and those based on simulation. Though analytical methods are usually superior in terms of execution time, they tend to be less accurate than simulations, due to the limited scope of system properties they can capture.

Recently at the Computer Engineering and Networks Laboratory an analytical method has been developed based on the theory of real time calculus (RTC) [1]. An analytical framework has been established which can be used in initial stages of the design space exploration. A reference system-level model of a network processor architecture has been chosen and was evaluated both by using a known analytical model and also by detailed simulation. By comparing these results, it was shown that the analytical method was accurate for this simple packet processing application. However, it is not clear whether this also holds true for multimedia real time applications. To answer this question, the analytical results will be compared to data acquired from simulations. The analytical framework is quite flexible and can easily be re targeted to a new application domain. The network processor simulator on the other hand, is not suited for simulation of interesting multimedia streaming applications. A new simulation framework needs to be created. This new framework will also be useful for the comparative studies as well as for multimedia SoC software and hardware architectural research.

Requirements

- **Simulation of multiprocessor architectures**

In order to evaluate System-on-Chip architectures the simulation framework must be able to simulate multiple parallel processing units and peripheral subsystems. Its task is to provide information about the requirements for resources, hot spots and bottlenecks in the system, the extendability with additional hardware and the sensitivity to input traffic [2].

The goal of this project is to present a simulation framework for multiprocessor Systems-on-Chip

- **Efficient Performance Estimation**

During design cycles, fast performance estimation is required in order to evaluate a maximum of possible design variants. Due to the complexity of such design variants simulation is very time consuming. The performance of the evaluation tools have an enormous influence on the length of design cycles.

The goal of this project is to design an efficient simulation framework.

- **Support for variable layers of abstraction**

In order to achieve a reduction of simulation time, the evaluation tool needs to provide the flexibility to change the level of detail. Since more detailed examination requires additional computation, there is always a trade off between accuracy and simulation speed. If the simulator does not fulfill the requirements concerning simulation time, it must be possible to compromise the level of detail for a higher speed. Different layers of abstraction give the designer the opportunity to concern only about the necessary details of the evaluation. Simulation of the entire system with high details is not feasible, and usually not necessary.

The simulation framework should support various levels of abstraction for individual modules.

- **Capability of HW/SW co simulation**

Integrating processor models into hardware simulation environments allows designers to execute their software on the hardware models. This gives the possibility to analyze the effect of the target application on the system. In addition, co-simulation of hardware and software enables the verification of application code running on processor models in conjunction with accurate models of the hardware system. [3]. Co-simulation/verification of hardware and software has been proposed for a number of environments such as [4] [5] and is also offered commercially by companies such as Mentor Graphics [6], Synopsys [7] and others.

The simulation framework should allow execution of custom applications on a hardware simulator.

- **Support for various processor models**

The simulation framework should not be constrained to a particular processor model. It should provide the flexibility to use different instruction-simulators such as: Shade [8], ARMPhetmaine, tARMac, SimARM [9], ARMulator [10] and ARMSim [11]for ARM processors; MINT for MIPS R3000 [12]; Xtensa ISS for Xtensa processors [13] and SimpleScalar[14] which supports multiple platforms. The framework must also support processor simulator of a higher abstraction level, for example simulators based on SystemC.

The simulation framework should allow to integrate various instruction-set simulators.

- **Implementation with SystemC and SimpleScalar**

The idea arose, to write both the instruction set architecture model and the remaining hardware models in the same language. The anticipated executable specifications would possess a very high level of flexibility and facilitate the entire HW/SW co-design. There exist two open source projects which are written in closely related language: SystemC [15] and SimpleScalar [14]

The project required to use SystemC as the hardware modeling language. SystemC is a C++ library with a simulation core based on discrete event systems. It was chosen for this project for its support of multiple layers of abstraction.

Another predefined requirement was to use SimpleScalar instruction-set simulator. It's sequential execution makes it a very fast. SimpleScalar models a variety of instruction-sets and features exchangeable simulation cores with different levels of detail. Since it is written in C, it is easier to combine it with SystemC.

Instruction Set Simulators and Hardware Models

The SystemC modeling language

SystemC is a set of modeling constructs entirely based on C/C++. It allows RTL and behavioral modeling similar to modeling within an hardware description language (HDL) such as Verilog or VHDL. Similar to these HDLs its simulation core is based on a discrete event system. Using modules, ports and signals, users can construct structural design.

Modules can be instantiated within other modules, thus creating a hierarchical design. Different levels of abstraction are possible, ranging from high level abstraction models to low level register transfer level models. Signals connected to ports allow communication between modules. The designer can choose from a rich set of signal types on various levels of abstraction. They can be as detailed as bit-vectors as well as abstract as to represent entire data structures. The SystemC simulation kernel contains routines to trace these signals and dump the waveforms to a file. After completion of the simulation, a typical waveform viewer can display them.[16]

Concurrent behaviors are modeled through processes. A module can contain several parallel processes. Such a process is like an independent thread of control and is sensitive to predefined signals. Whenever the values of these signals change, the process performs some action. Afterward, it suspends execution and transfers the control to the next process.

Communication can be even more generalized by using channels, interfaces and events. A channel is an object, that serves as a container for communication and synchronization. Channels are the implementation of one or more interfaces. The access methods to the channels are specified within the interfaces, but the interfaces do not provide the implementation. An event is a synchronization primitive used to construct other forms of synchronization without physically connecting signals. Events do not have a type or transmit a value, they only transfer control form one thread to another. [17]

The SystemC 2.0 standard also includes elementary library models such as timers, FIFO, buses, etc., which are widely applicable.

SystemC is an open source project under the Open Community Licensing model. Designers are allowed to create, validate and share their models within the community. There are no licensing fees for either internal nor commercial use. The driving force behind SystemC is the Open SystemC Initiative (OSCI), a non-profit organization run by a broad number of universities and companies including Fujitsu, Motorola, STMicroelectronics, Synopsys, CoWare and Cadence[15].

The SimpleScalar tool set

The SimpleScalar instruction-set simulator (ISS) provides an infrastructure for simulation and architectural modeling. It is used to build benchmark applications for performance analysis, micro architectural modeling and hardware software co-verification. SimpleScalar provides the tools to model various platforms, from simple unpipelined processors to detailed dynamically scheduled micro architectures with multiple-level memory hierarchies.

The simulators use interpreters to reproduce the processing units. The interpreters are fully customizable and already support several popular instruction sets such as Alpha, PowerPC, x86 and ARM. Using its own compiler, a variant of the Gnu C compiler, the designer can write his own modeling applications, in order to test the instruction set architecture.

Simulator	***Description***	***Speed***
Sim-Fast	Speed optimized functional simulator	7 MIPS
Sim-Profile	Dynamic Program Analyzer	4 MIPS
Sim-Outorder	Detailed micro architectural timing model	0.3 MIPS

Table 1: SimpleScalar simulator models

Table 1 shows an excerpt of the various simulator cores for SimpleScalar. The fastest yet the least detailed simulator is *sim-fast*. *sim-fast* just executes each instruction sequentially without any accounting for time. It is the fastest simulator, but it supports only rudimentary profiling. In fact, it simply executes the program and counts the number of committed instructions. *sim-cache* and *sim-cheetah* are two functional cache simulators which are ideal for fast simulation of caches if the effect of cache performance on execution time is not needed.

sim-profile generates detailed profiles on instruction classes and addresses, text symbols, memory accesses, branches and data segment symbols. Without taking cache timing into account and support for only in-order issue of instructions, *sim-profile* cannot measure the timing effects of executed programs.

Out of order simulation

The most detailed simulator in the distribution is *sim-outorder*. This simulator supports out-of-order issue and execution based on the register update unit (RUU) [18]. The RUU scheme uses a reorder buffer to automatically rename registers and hold the results of pending instructions. The processor's memory system employs a load/store queue (LSQ). Store values are placed in the queue if the store is speculative. Loads are dispatched to the memory system when the addresses of all previous stores are known. The simulator uses an event queue that holds the operations which can be written back to the register file in the future. *sim-outorder* employs a ready queue which holds all operations that are ready to be issued to a functional unit.

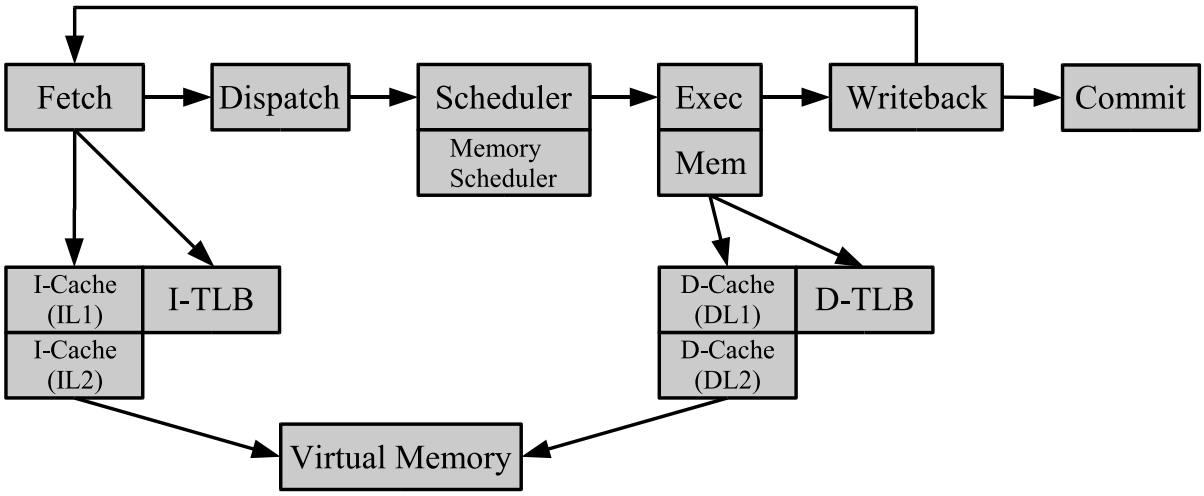

Illustration 1: out-of-order issue simulator

Since this simulation core was used in this project, its functionality is described here with more detail.

A register update unit (RUU) station is a reservation station which captures results from the register. If all operations are ready, the instruction is issued to a functional unit. The load and store queue (LSQ) holds the loads and stores in program order and indicates their status of access.

Figure 2 shows one simulation cycle in *sim-outourder*. The cycle starts by checking both the RUU and the LSQ for completed instructions. These are committed and removed from the queue. In case of stores, the cache is accessed and if a miss occurs, it writes back and replaces the accessed block.

The writeback function then services completed results, such as memory references, from the event queue and puts these operations int the ready queue. It broadcasts the result to consuming operations and updates the branch prediction.

Afterward, memory operations that are ready to be executed are located in the LSQ and are inserted in the ready queue.

If the an operation's register dependencies are satisfied and a functional unit is available, the functional unit is allocated and the operation issued. After scheduling the writeback event, it begins execution. For loads, the latency is calculated through access of the cache, where also cache miss penalties are handled. Store operations are not scheduled, since SimpleScalar assumes an infinite number of write buffers.

The dispatch function decodes and dispatches new operations. Operations with any dependencies are put directly into the ready queue. Otherwise the function updates the input and output dependencies and allocates an register update unit and in case of loads and stores a link in the LSQ.

At the end of a cycle, in case the fetch unit is available, the simulator performs an instruction cache access and fetches new instructions, thereby blocking the fetch unit for the number of cycles according to the memory access latency.

Design

Layers of Abstraction

Simulation Time versus Accuracy

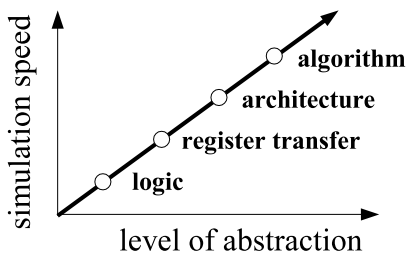

Illustration 2: speed vs. abstraction

During the development of a system, the designer needs to have access to detailed yet slow simulators, in order to detect misbehavior and to evaluate performance. In order to reduce development cycle time, the designer also asks for fast simulators. Higher levels of detail ask also for more computation time. Therefore, a simulator cannot yield a fast simulation time and a high level of accuracy at the same time. Since it is not feasible to model the entire system in detail, it must be possible to model non relevant parts with more abstraction. Details are compromised in order to obtain a faster simulation. If the simulator supports different levels of abstraction for each module, then the designer is able to accurately investigate the point of interests, without wasting simulation time for unimportant parts of the system.

Levels of Detail

During the design cycle of an SoC, designers use a variety of simulators These simulators model the design at different levels of abstraction and become slower as the design progresses to lower levels of abstraction[19].

Design Level	***Description language***	***Primitives***	***Host cycles per simulated cycle***
Algorithm	HLL	Instructions	100 - 100
Architecture	HLL	Functional Blocks	1000 - 10'000
Register Transfer	HLL, HDL	RTL primitives	1M - 10M
Logic	HDL, Netlist	Logic gates	10M - 100M

Table 2: layers of abstraction

The highest level of system design is the specification of the algorithm or protocol that the system is targeted to perform. The designer can verify the correct functionality of the algorithms and protocols. Instruction-set simulators allow to evaluate the effects of the instruction-set on the application's execution time. The integration of an ISS into a hardware model even allows to simulate the effects of the instruction-set on the target system.

The architecture level concerns about the definition of the main functional blocks and their estimation of performance. An example of these blocks the processor interacting with the memory system or processing hardware units.

At the register transfer level the goal is to model all the components in the system in enough detail such that performance and correctness can be modeled to the nearest clock cycle. At this level of abstraction the simulator accurately models all the concurrency and resource contention in the system.

At the logic level all the nets and gates in the system are defined. Such high level of detail requires an enormous amount of computation. This level is not discussed in this project, since the slowdown of the simulation would not be acceptable.

Incrementally Refining a Model

SystemC supports incremental refining of models. First, models are implemented on a high level of abstraction. In case more detail is needed, a refined model substitutes the abstract model. This gradual refinement applies to all models, including communication. Following the interface-based design methodology[20] communication on the system bus is modeled through the passing of tokens. A token represents a complete communication between two or more entities. Through successive refinement of the models higher levels of detail can be achieved. Increasing the level of abstraction gives the option of increasing simulation speed.

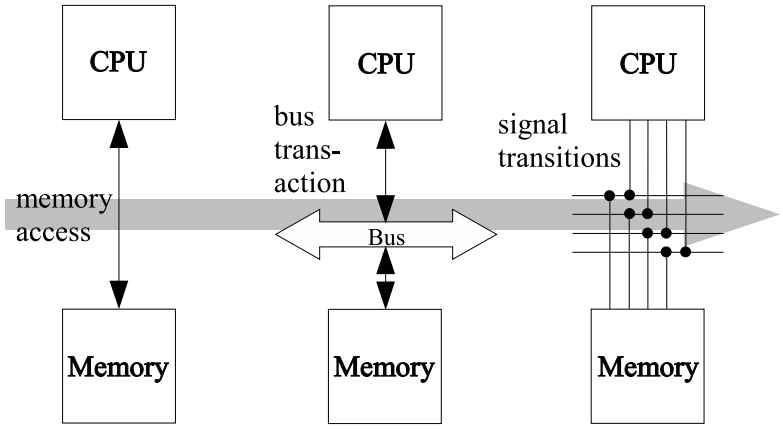

Illustration 3: incremental model refinement

Whenever the instruction-set simulator communicates with the system, it issues a token. This token contains the information about the communication: target address, the length of the data, type of command (read or write), bus access parameters and the id of the issuing cpu. After the communication, the token is marked completed, thereby informing the ISS about the end of the communication.

Interfaces

In order to provide the support for different levels of abstraction, the models in the framework access other models through interfaces. Interfaces provide the access functions to model and hide their implementation at the same time. The user of an interface calls functions, without concerning about their implementation.

One interface defines the communication between the hardware model written in SystemC and the instruction-set simulator. Using this interface allows to integrate many different ISS into the SystemC environment. With this interface, the ISS can be started and clock cycles can be evaluated one at a time. Through this interface, the hardware model receives tokens, which represent a communication on the system. Also, he hardware model signals the ISS when these transactions have completed through this interface.

Another interface models memory references on the system, using tokens. The interface accepts tokens and marks them as completed when the memory access is finished. The interface only provides a read and a write function, and a function to show whether the interface is busy. Without the use of any signals or clocks or similar, the implementation can be very abstract.

A third interface provides access to the simulated memory space. This memory space holds the data of the instruction-set simulator, such as the program code,and the simulated heap and stack. This provides the flexibility to split up the memory space into smaller pieces called memory banks. For the ISS the simulated memory space looks always the same, while the organization of the memory can be changed. The use of such an interface allows to redirect memory transactions to different parts of the memory space, without having to change the instruction-set simulator.

These interfaces allow to change the level of abstraction of certain parts of the system, without having to adapt the rest of the system.

Encapsulation of the Instruction Set Simulator

Combination of cycle accurate ISS with discrete event simulation

This project's approach is to use instruction-set simulators to model the processing units. Usually, instruction set simulators are written in a high level language (HLL). The cycle based ISS do not account for timing constraints, one clock cycle being the smallest unit of time. Typical modeling languages for hardware are hardware description languages (HDL) such as Verilog or VHDL. These modeling languages use a simulation core based on a discrete event simulation (DES). DES are not constrained to a time frame with regular intervals. DES calculate at the occurrence of an event, at what points in the future the system will change its state. Due to the large simulation overhead of DES, a processor model would be too slow for reasonable simulation. The sequential programming of the cycle accurate models produces a much faster simulator. The goal of this project was to combine a cycle accurate instruction set simulator with a hardware simulator based on a DES.

SystemC Wrapper

The wrapper's task is to integrate the instruction-set simulator into the SystemC environment. Since it communicates with the ISS through an interface, it is possible to replace SimpleScalar with a different ISS.

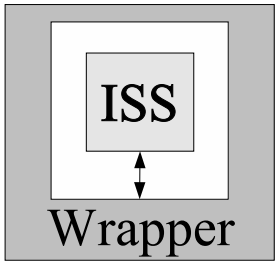

Illustration 4: wrapper

The wrapper synchronizes the ISS with the hardware simulator and guarantees the proper order of events. Basically, a processing unit is a finite state machine. It calculates its next state and its output in dependence of the current state and the input. The ISS executes sequentially and models one clock cycle inside a loop. The wrapper has to break up this loop and tie the execution of the application to the system clock. Each clock cycle, the wrapper translates incoming events from the system and evaluates the next state of the ISS. The question is how the wrapper is suppose to behave in case more than one event is pending at the input, for example more than one token has completed. This two solutions are possible:

- The wrapper schedules events and sends only one to the ISS. The advantage is that the wrapper relieves the ISS of evaluating unimportant events. This improves the ISS's performance, since no time is wasted for ineffective events. Deprecated of the information about the other events, the ISS may not correctly calculate its next state, which results in misbehavior. The ISS can avoid this by referencing the wrapper for the state of the essential ports. This leads to a significant increase of complexity and requires substantial changes to the ISS in order to ensure correct behavior. Such complicated communication between the ISS and the wrapper generates unnecessary simulation overhead.

- The wrapper sends all events to the ISS. With this solution the complexity moves to the ISS. Not the wrapper but the ISS evaluates the signals for their importance. Taking every event into account increases the stability of the instruction set simulator. Although this demands for additional code to the ISS the wrapper is reduced in complexity.

The latter solution was implemented, because it also bears the advantage that the interface does not depend on the implementation of the ISS.

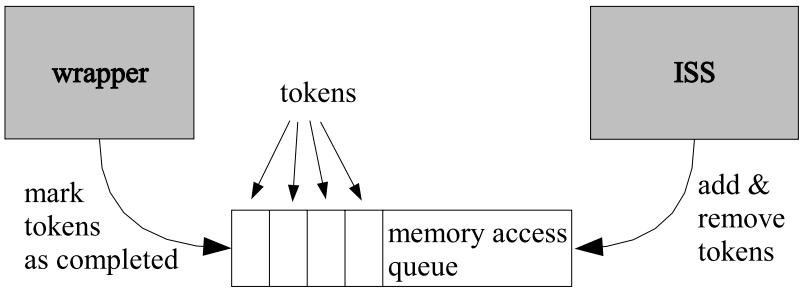

Illustration 5: memory access queue

The instruction-set simulator puts tokens for memory accesses into a queue. The wrapper checks this queue for uncompleted events. Through a memory map the wrapper selects the corresponding memory interface and if it is available, it issues the token. When the transaction has completed, the wrapper marks them as completed.

When the wrapper invokes the ISS, it evaluates one simulation cycle. At the beginning of each cycle, the ISS checks the memory access queue for completed tokens and uses this information to calculate its next state.

Instruction Set Simulator Interface

The evaluation of one cycle on the ISS comprises of four steps:

1. The wrapper sends events from the hardware model to the ISS.
2. The wrapper hands over control to the ISS
3. The ISS calculates the new events and transfers them to the wrapper
4. The ISS returns control to the wrapper

There are three variants to give the instruction set architecture access to the events:

Callback functions

The ISS registers a callback function inside the wrapper. The wrapper calls this function in order to evaluate one cycle of the ISS. The advantage is that this approach yields a high level of flexibility because the ISS can change the callback function during runtime. It is questionable whether this flexibility is an improvement to the overall simulator since there is no reason why the callback function should change during runtime. In any case, it has the disadvantage of increasing the complexity.

Direct access to SystemC ports

The wrapper registers pointers to its ports in the ISS. The the ISS is able to access the ports directly[21] which reduces the simulation overhead. The wrapper and the ISS are not separated anymore and the independence of these two are not provided anymore. This leads to a substantial reduction of flexibility. In addition, whenever a different ISS is being used, significant adaption is required in order to maintain compatibility

Abstract interface class

All access methods are defined within an interface class. The class of the ISS is inherited from this abstract base class Declaring the methods pure virtual constrains the programmer of the ISS to provide the implementation of these methods. The clear definitions simplify the implementation and the pure virtual functions assure consistence throughout the source code. Although due to the predefinition of the methods the flexibility of the implementation of the ISS is reduced, these constraints guarantee the wrapper's independence of the ISS. These are the reasons why this solution has been chosen.

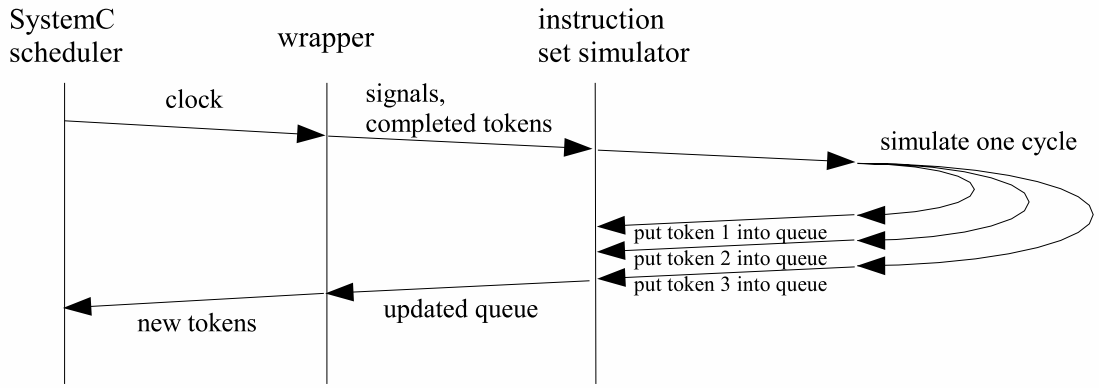

Illustration 6: interface to the instruction-set simulator

SimpleScalar's Emission of Tokens

Whenever SimpleScalar simulates a cache miss, it puts a token into the memory access queue. This token does not contain any data. Its sole purpose is to calculate the memory access latency, the time after which the transaction has completed.

In case of an instruction fetch, SimpleScalar blocks the fetch unit until the load of the instructions has completed.

At the beginning of a cycle, SimpleScalar scans the memory access queue for completed memory transactions.

If an instruction fetch has completed, SimpleScalar frees the fetch unit and fetches a new instruction. If a load instruction has completed, SimpleScalar puts the operation into the event queue in order to write back the result and retire the instruction.

Completed store transactions are ignored and are not handled by SimpleScalar. In order to correctly model behavior of the processor, SimpleScalar must also provide a mechanism to model delays on store instructions. One solution is to block the particular load/store unit, until the write transaction has completed. As long as the load/store unit is blocked, it will not accept a new instruction to execute. However, this mechanism has not been implemented yet.

Memory Space

SimpleScalar's Virtual Memory

SimpleScalar separates the timing modeling of the memory from its functionality. The content of the memory is not simulated in the SystemC memory model. SimpleScalar stores the values of the simulated memory in different memory model. The simulator allocates its simulated memory space in the host's memory. SimpleScalar's page tables translate the logical addresses of the simulated memory references to the host's memory logical addresses. The host's page tables translate these addresses again to physical addresses.

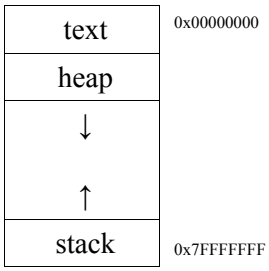

Illustration 7: SimpleScalar's memory organization

SimpleScalar encapsulates its simulated memory space. It is important that the instruction-set simulators have access to a common memory space, since in multiprocessor architectures processors one method of exchanging data is through shared memory. In order to share data through memory, the simulated memory space and its organization is extracted from SimpleScalar and is encapsulated in its own, independent object. The simulated memory space is subdivided into smaller memory spaces called memory banks. One memory bank can represent a shared memory, a tightly coupled memory, a FIFO or a memory mapped I/0.

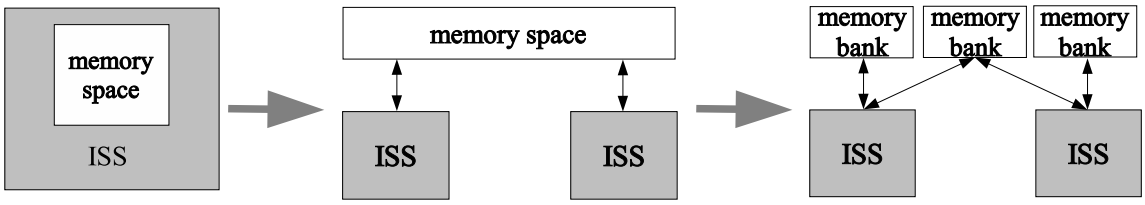

Illustration 8: encapsulating the memory space into an independent class with multiple memory banks

Memory Interface

The interface base class provides the semantics of the memory access functions. By hiding the memory space from the simulator, the simulator does not rely on the implementation of the simulated memory space.

The interface consults a memory map in order to select the appropriate memory bank, depending on the address.

The memory can be organized in two different ways:

- All memory banks reside inside a memory pool. The memory pool assigns an interface to each cpu. This interface also holds the memory map for its assigned cpu. The entire memory space is encapsulated inside this memory pool with the advantage that the wrapper and the simulated memory are separated. The disadvantage is that this also results in limitations to the interface, since each memory map depends on the connected cpu.

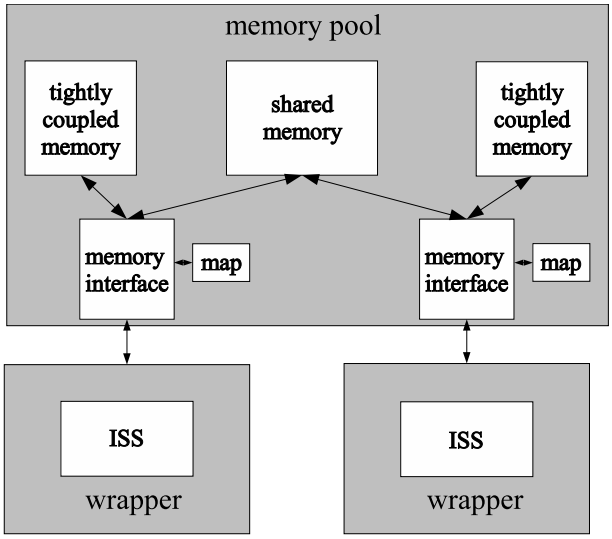

Illustration 9: memory pool

- The memory interface is part of the wrapper. The interfaces to the available memory banks are registered inside the wrapper. The wrapper holds the memory map and selects the appropriate memory bank, depending on the address. The wrapper's memory interface translates the ISS's memory accesses and forwards them to the corresponding memory bank.

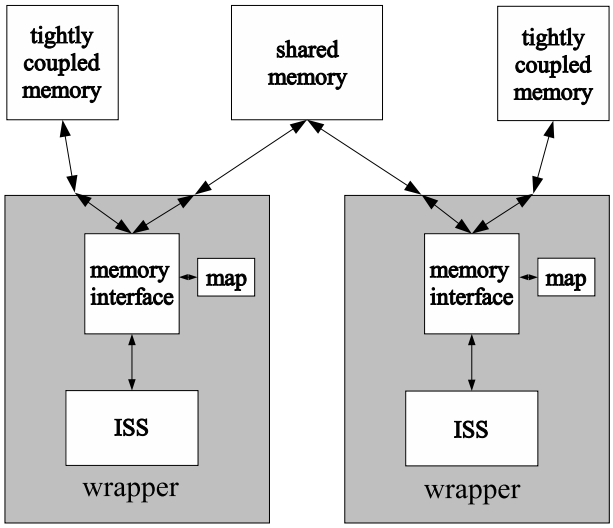

Illustration 10: wrapper holds the memory interface

The second design has been chosen, since it takes into account that each cpu needs its own specific memory map. In addition, with more sophisticated memory layouts, this design proofs to be more flexible. Additional memory banks are more easily integrated into the system.

Memory Map

Two memory interfaces can access the same memory bank. This is called shared memory. The two interfaces can map their addresses to the same range or can overlap only partially.

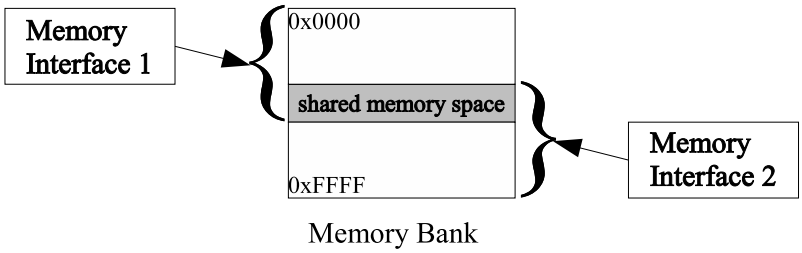

Illustration 11: partial overlapping of address ranges

An address range of the instruction-set simulator does not necessarily have to map to the same address range on the memory bank. Even more, it is possible that two different address ranges map to the identical address range on the memory bank. This memory address translation is not yet implemented in the simulation framework, but it is possible to improve the memory interfaces to support this feature. Figure 12 shows one possible memory mapping in a two processor system with multiple memory banks.

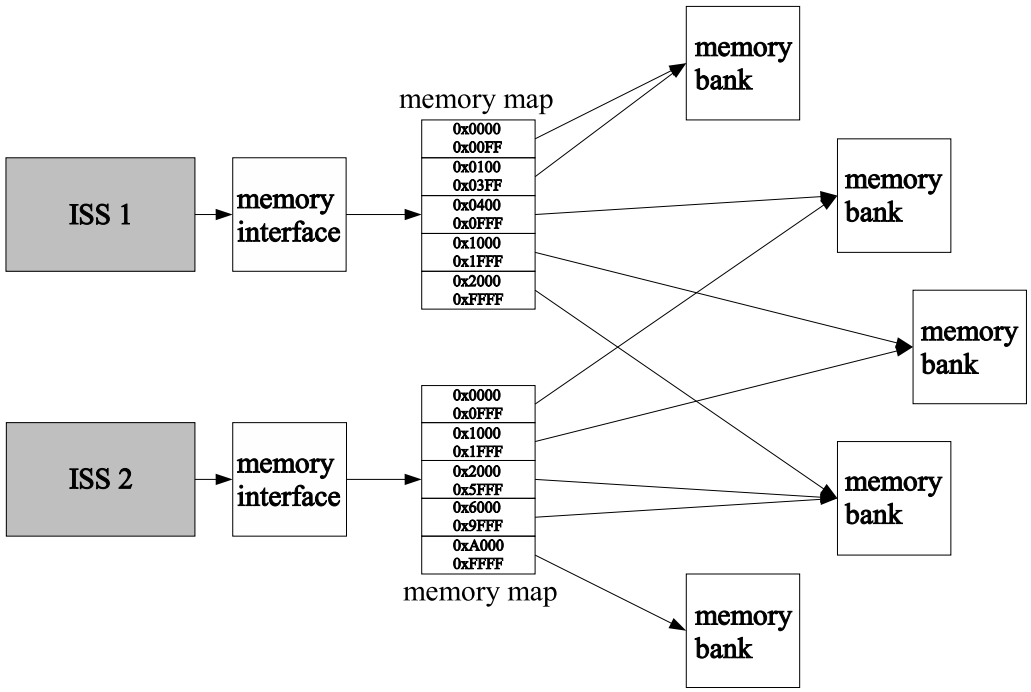

Illustration 12: example of memory mapping

Memory Coherence

The simulation framework's task is to preserve the correct order of events. This also applies to the order of accesses to the memory. The order of accesses to the simulated memory space must correspond to the order of access to the memory modeled with SystemC.

Possible inconsistencies occur when two processors try to access the same memory location during the same cycle. If at least one of these processors is writing to the memory location, there is a chance that the order of access to the simulated memory does not correspond to the order of memory accesses simulated with SystemC. SystemC's simulation core does not guarantee the order in which the modules are executed in the same clock cycle. Thus, whichever processor model is executed first, accesses the simulated memory first. Which module is granted access to the bus and accesses the SystemC memory model, depends on the arbiter. Possibly these two orders are not the same, which causes an inaccuracy in the simulation.

Let us consider three scenarios involving two processors P1 and P2 accessing the same memory location.

Scenario 1: Memory accesses not in the same cycle

This scenario shows the case when P1 and P2 do not access the shared memory during the same cycle. P2 performs the memory transaction after P1.

Cycle	*Processor 1*	*Processor 2*
1	P1 writes to the simulated memory and issues a memory access token. The arbiter grants access to the bus.	
2	P1 accesses the bus and performs a write transaction on the memory.	P2 reads from the simulated memory and issues a memory access token. The arbiter denies access to the bus.
n	P1 completes the bus transaction	
n+1		The arbiter grants access to the bus. P2 accesses the bus and performs a read transaction on the memory.

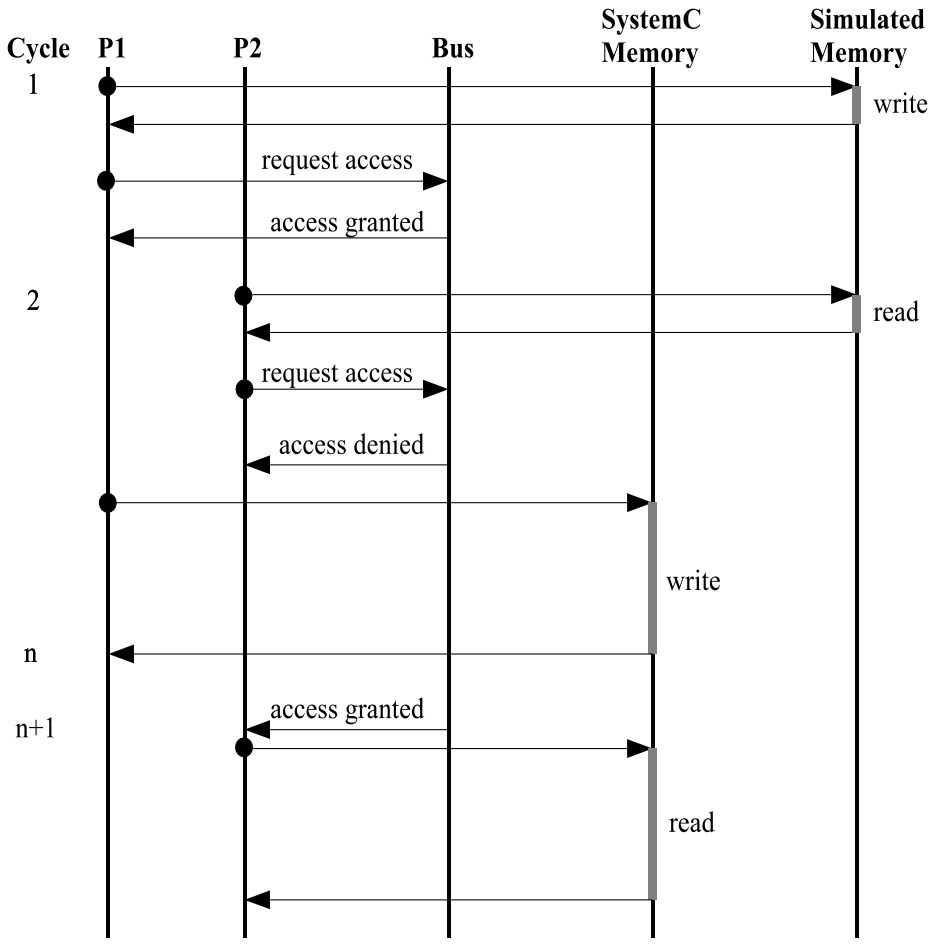

Illustration 13: coherence scenario1: P2 accesses the memory one cycle later

In this scenario the order of events is preserved.

Scenario 2: Memory accesses not in the same cycle, order preserved

In this scenario the memory accesses occur in the same cycle. P1 is called first by SystemC and also has a higher bus access priority than P2.

Cycle	*Processor 1*	*Processor 2*
1	P1 writes to the simulated memory before P2 and issues a memory access token. The arbiter grants access to the bus.	P2 reads from the simulated memory after P1 and issues a memory access token. The arbiter denies access to the bus.
2	P1 accesses the bus and performs a write transaction on the memory.	
n	P1 completes the bus transaction	
n+1		The arbiter grants access to the bus. P2 accesses the bus and performs a read transaction on the memory.

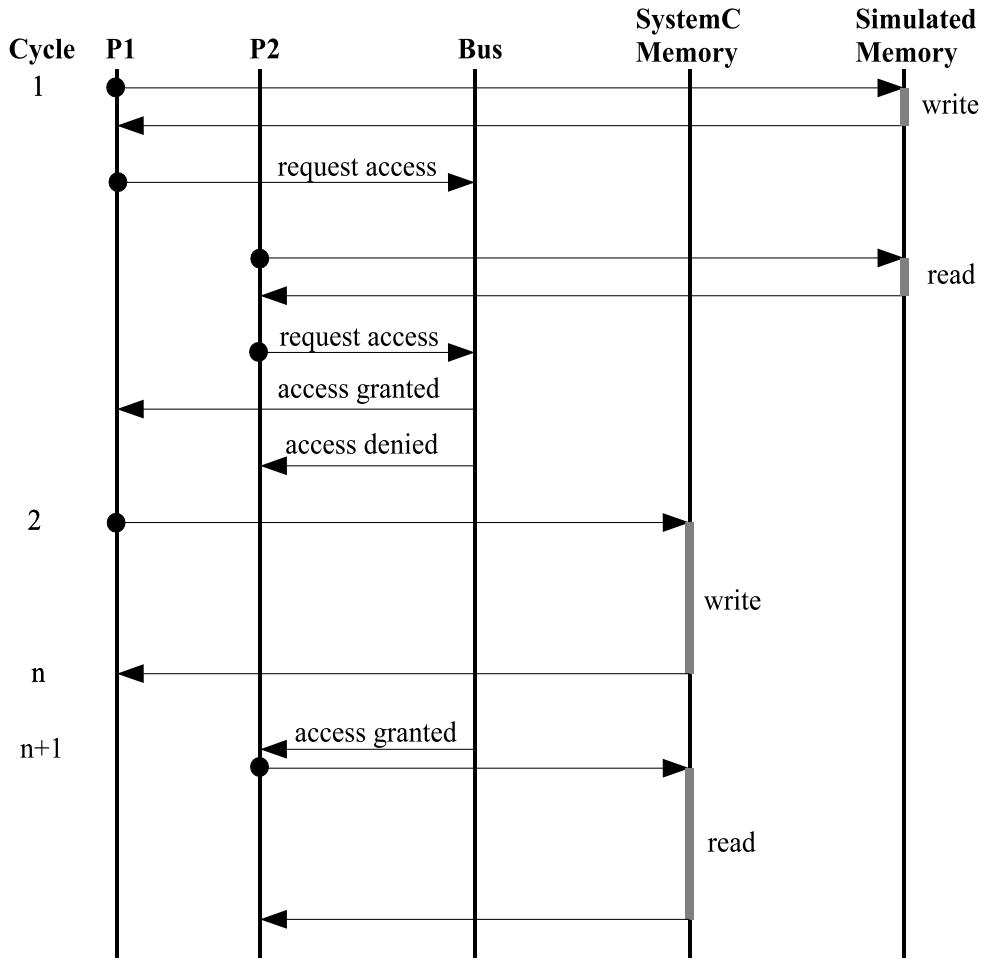

Illustration 14: coherence scenario 2: P1 and P2 access the memory during the same cycle

In this scenario the order of memory accesses is also preserved.

Scenario 3: Memory accesses not in the same cycle, order not preserved

In the last scenario, SystemC calls P1 first and P1 writes to the memory. But P2 has a higher bus access priority and thus performs the bus transaction before P1.

Cycle	*Processor 1*	*Processor 2*
1	P1 writes to the simulated memory before P2 and issues a memory access token. The arbiter denies access to the bus.	P2 reads from the simulated memory after P1 and issues a memory access token. The arbiter grants access to the bus.
2		P2 accesses the bus and performs a read transaction on the memory.
n		P2 completes the bus transaction
n+1	The arbiter grants access to the bus. P1 accesses the bus and performs a read transaction on the memory.	

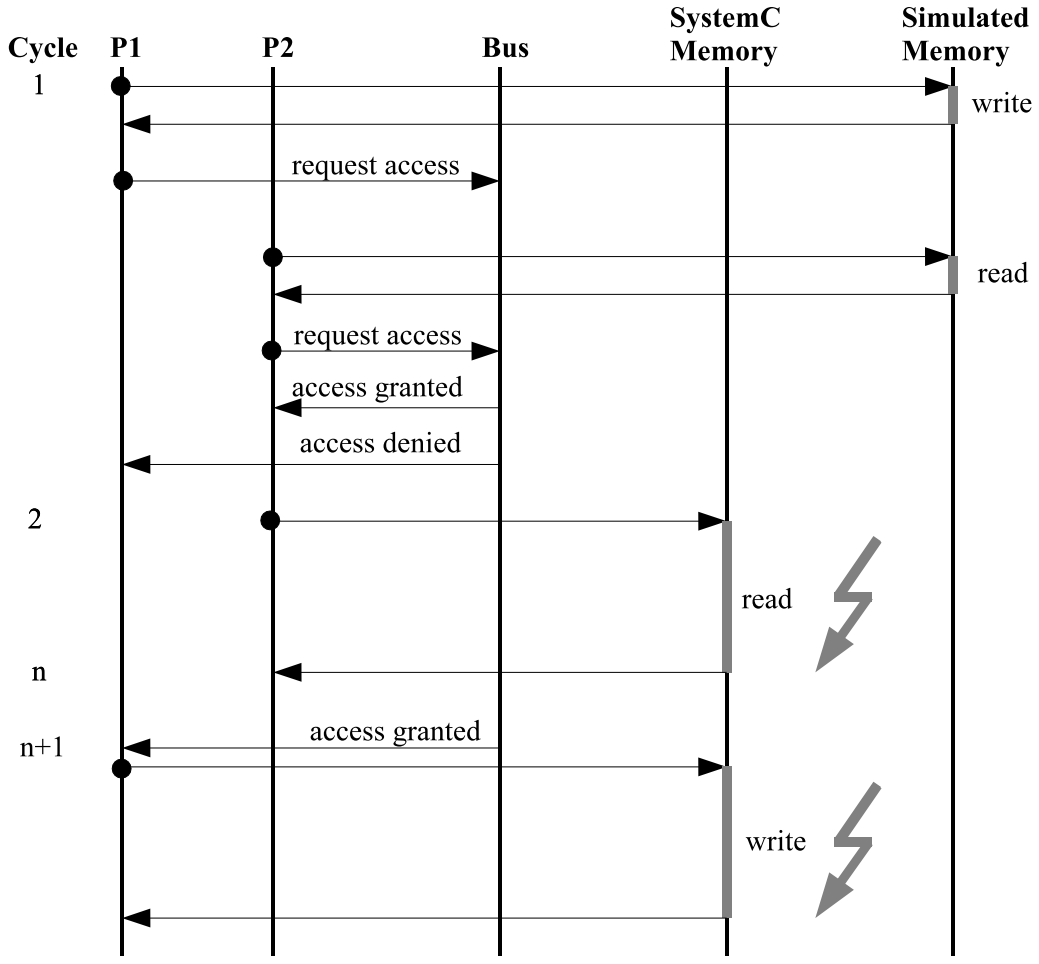

Illustration 15: coherence scenario 3: P1 and P2 access the memory during the same cycle

In this scenario the processors do not access the simulated memory in the same order as they perform the transactions on the SystemC model. Although P2 reads from the memory location after P1 has written to it, in the SystemC model it performs the transaction before P1 and thus causes an inaccuracy in the simulation.

This issue has not been resolved in the presented framework. The hazard can be avoided by omitting that both processors access the memory during the same cycle.

System Model

Bus Model

Multiple master and slave modules connect to the bus. Whereas master modules initiate a transaction, slave modules can only perform an action after being invoked by a master. In case more than one master is requesting a bus transaction, an arbiter selects one according to the arbitration scheme.

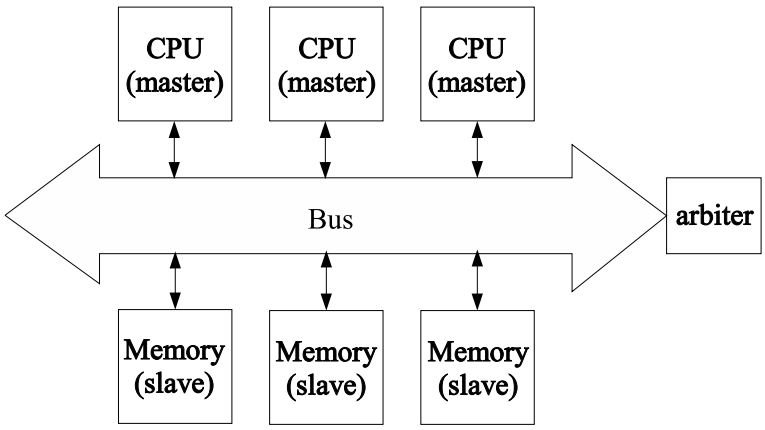

Illustration 16: bus architecture

Since the bus model was not the main focus of the project, the Simple Bus library which is part of the SystemC 2.0 release, has been chosen as sample solution. Its description follows in the implementation chapter.

Processor Model

The different parts of the processor model are collected in one container. This container comprises of the wrapper including the instruction-set simulator, SystemC memory interfaces, interfaces to the simulated memory banks, ports for input and output signals and ports for clock and reset.

SystemC Memory Interface

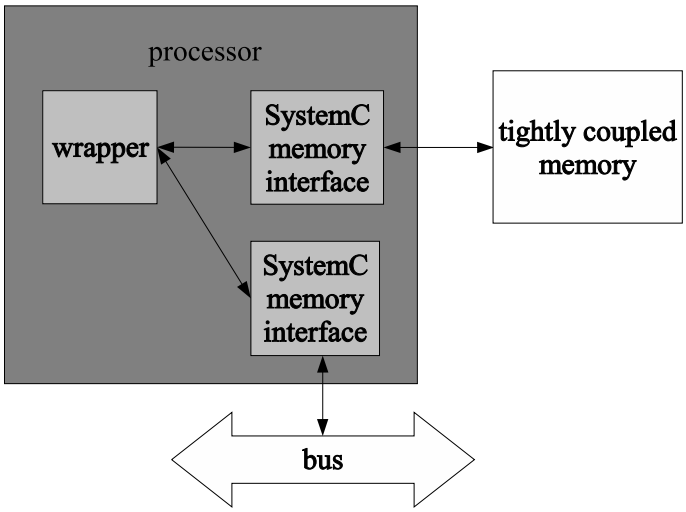

Illustration 17: SystemC memory interface

The SystemC memory interface receives the tokens from the wrapper and accesses the corresponding memory on the system. This memory can be directly connected to the interface, as in case of a tightly coupled memory, or it is connected over a bus or over some other kind of interconnect.

The wrapper does not have to concern about the location of the memory and how it is connected to the processor. From the wrapper's point of view, the interface accepts a token for either a read or a write transaction. Until the interface has completed the transaction, the memory interface signals that it is not ready to accept new tokens. After completion, it marks the token as completed.

Implementation of the Simulation Framework

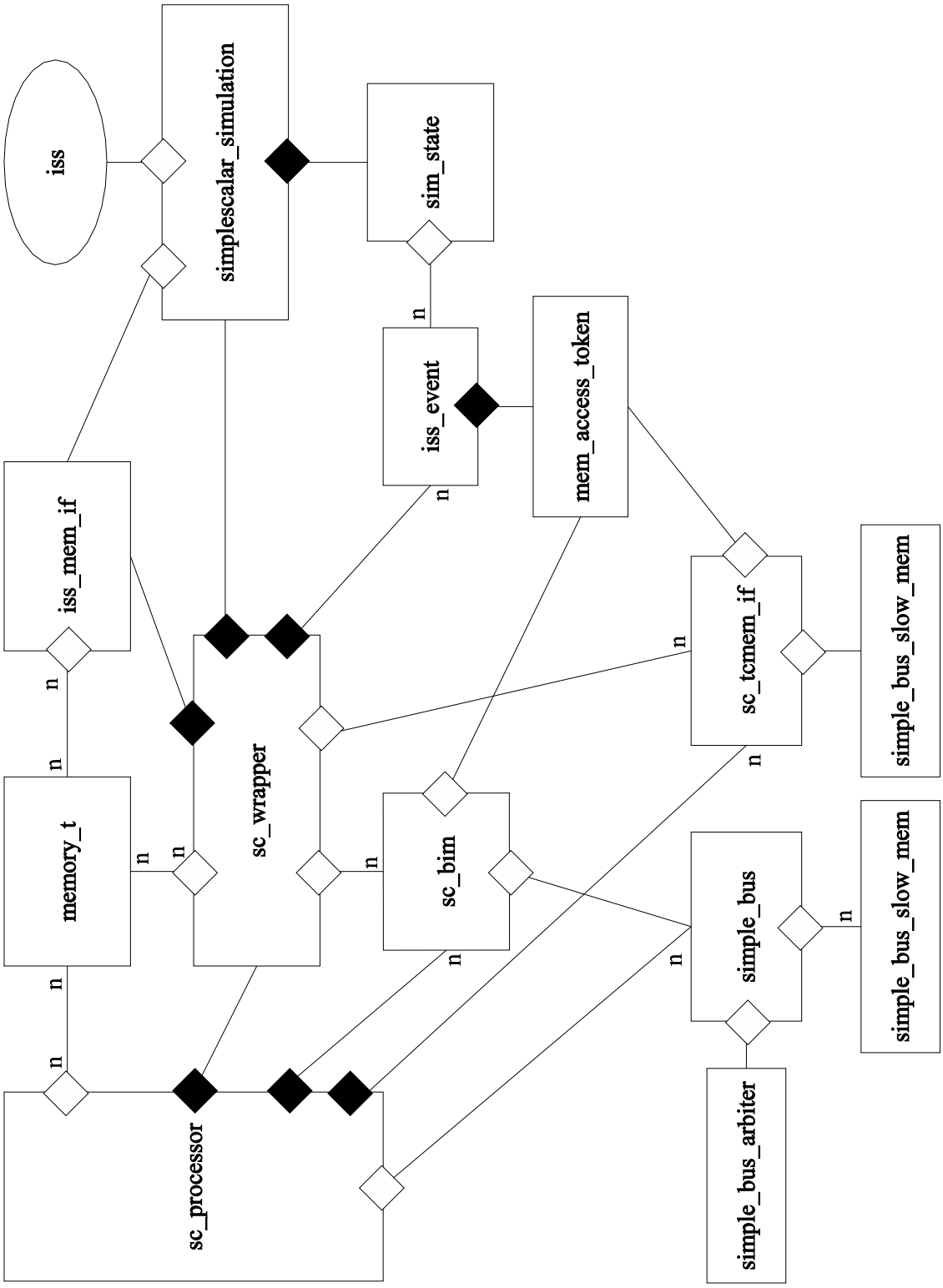

Illustration 18: UML diagram of the simulation framework

Encapsulation of the Instruction-Set Simulator

From C to C++

In order to use the SimpleScalar simulator inside a C++ environment, the source code requires several adjustments. Beside syntax changes, the entire program needs to be put inside a class. This class is called `simplescalar_simulation`. Since the source code is scattered over many files, the functions must have access to global variables and external functions. In C this problem is solved by declaring those functions and variables as external. In C++ this does not apply anymore. There are three alternatives:

1. All functions and global variables are part of a common name space. Because name spaces can be split over multiple files, they do not create a redefinition[22]. Thus the modification of the source code is kept to a minimum. The downside this approach is that only works if the SimpleScalar simulator is singleton[23], the object being the only instance of its class. Since every instance has access to the same name space, they would access the same variables, thus writing into each others memory space.

2. The simulator has its own class and the functions and global variables are members of this class. This way, the simulator is isolated and the different instances do not interfere with each other. The original source code of SimpleScalar uses many header files in order to improve readability. C++, on the other hand, does not allow to spread a class definition over multiple files. Either all header files are copied into a single file, resulting in a not very manageable file. Or somewhat unusual, the header files are included inside the class, which also calls for some restrictions in C++. In addition, the SimpleScalar simulator makes wide use of callback functions, which would involve major changes to the source code in order to maintain the functionality.

3. The solution is the combination of the previous two. This way, the simulator is able to use the methods, defined in the common name space *iss*. The global variables are defined within the class sim_state, which is a member of the `simplescalar_simulation` class. The `sim_state` class preserves the state of the simulator whenever control moves out of its scope. It also initializes the variables with their respective value. The reference to this state is passed to every function within the names space `iss`, because the function cannot otherwise determine which instance of ISS is calling it nor its current state.

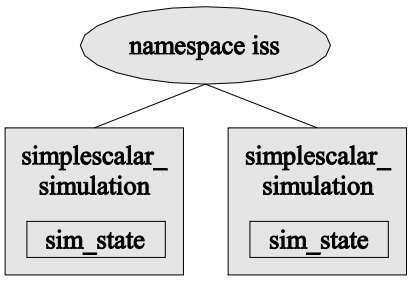

Illustration 19: using name spaces

The source file `main.c` is transferred to the file `simplescalar_main.cpp` and the function `main()` changes its name to `sim_start()`. It is called either right after instantiation or whenever the programmer wishes to start the ISA simulation. The simulator core routines are implemented in the file `sim-outorder.cpp`. Substitution of this source file with other implementations such as `sim-fast.cpp` or `sim-profile.cpp` exchanges the simulator core with the respective implementation. This modular structure facilitates “rolling your own” simulator[24]. However, so far only `sim-outorder.cpp` was integrated in the framework.

The Module sc_wrapper

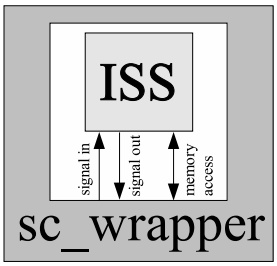

Illustration 20: the module sc_wrapper

The wrapper is a SystemC module with the name `sc_wrapper` which has interfaces to the memory banks and to the instruction set simulator. It also has single signal ports which can be used to flag events such as interrupts and ready signals.

The wrapper contains only one process with the name `HandleEvents`. This SystemC method is sensitive to the positive edge of the clock and to the reset signal. In case the reset signal goes high, the wrapper deletes the instructions set simulator, instantiates a new ISS and restarts the simulation.

Every clock signal, the wrapper forwards all input signals to the ISS. After evaluating one cycle of the ISS, the wrapper maps all outgoing signal from the ISS to their respective ports. If the a signal has not changed since the last cycle, it will not generate an event on SystemC. Then the wrapper checks all tokens in the memory access queue whether their corresponding interface is able to perform a transaction. By issuing memory accesses to all available memory interfaces, the wrapper is able to model concurrent memory transactions.

The wrapper has input ports for clock and reset. It also features input and output signal lines, as well as multiple interfaces to SystemC memory banks. One of these interfaces is the connection to the bus interface model.

The wrapper's constructor requires following arguments: The name of the SystemC module, the pointer to the first element of an array of memory interfaces, the number of memory interfaces, the command line parameters for the ISS, whether to turn on verbose output.

ISS interface

The abstract base class which defines the interface to the ISS is called `iss_if`. The instruction set simulator must provide three functions:

1. The function `start_sim` starts the `simulation`. Its arguments are the ISS parameters `argc`, `argv` and `envp`. They are the same as if SimpleScalar were called on the command line.
2. The function `eval_cycle` evaluates one cycle on the ISS. Its parameters are `signalin_array` and `signalout_array`, the pointers to the single signal arrays; `mem_access_queue_head`, the pointer to the first element of the memory access queue and `mem_access_queue_length`, the reference to the number of elements in the memory access queue.
3. The function `show_stats` displays the statistics of the ISS. It does not require any parameters.

```
class iss_if {
public:
virtual int  start_sim(int argc, char **argv, char **envp)=0;

virtual void
eval_cycle(iss_event<bool> *signalin_array[NUM_INSIGNAL],
           iss_event<bool> *signalout_array[NUM_OUTSIGNAL],
           iss_event<mem_access_token> **mem_access_queue_head,
           int *mem_access_queue_length)=0;

virtual void show_stats(void)=0;
};
```

Source Code 1: Interface to the ISS

The class `simplescalar_simulation`, the encapsulated SimpleScalar instruction-set simulator, is one implementation of the interface class `iss_if`.

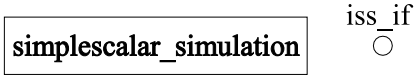

Illustration 21: interface class iss_if

The wrapper and the ISS exchange events using the class `iss_event` to store data. This class is a template depending on the type of data being used. In case of single signals, booleans are used. For the more abstract memory accesses, the type in use is the class `mem_access_token` which contains the information about memory transaction such as address, length of data, id

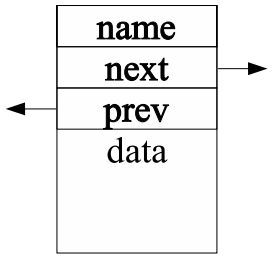

Illustration 22: the class iss_event

of the cpu, bus access parameters and whether it is a read or a write transaction. The token also contains flags that show if the token has been issued to a memory interface and if the transaction has been completed.

The single signal events are organized in two arrays, one for input ports and one for output ports.

This organization simplifies the handling of the signals. The memory access events are stored as tokens in queue called `mem_access_queue`. The class `iss_event` comes with a special constructor which put the instantiated element at the end of the queue.

At the beginning of each cycle, the SimpleScalar checks the event queue for completed memory references. Write transaction are not handled, since SimpleScalar assumes an infinite number of output buffers. If the completed read transaction accessed the instruction memory, SimpleScalar fetches new instructions. If the completed read transaction accessed the data memory, it puts the completed event into its own event queue. The ISS writes changes to the outgoing signals directly into the array. In case of memory accesses it stores the event at the end of the memory access queue. After updating the internal state machine, the ISS returns control to the wrapper.

Simulated Memory Space

Memory Banks

Data, such as program code, heap and stack is stored in the host's memory in a struct, which holds all the page tables. SimpleScalar accesses its virtual memory through functions defined in the source file `memory.c`. The simulator creates page tables where it stores the logical address of its data in the host's memory. More details on SimpleScalar's memory organization are described in SimpleScalar's source code [14].

The outsourced memory must provide the same functionality as the original implementation of SimpleScalar. In order to avoid many changes to the SimpleScalar code, the memory object must provide the same methods that were previously defined in the file `memory.c`. The question was whether to make the page tables public or private.

- Making the page tables public does not require much change to the SimpleScalar code. But there is no separation between the simulator and the implementation of the memory.
- Making the page tables private increases the complexity, since the simulator cannot access the page tables directly, but only by using certain interface methods. This solution increases the safety as well as the flexibility. The implementation of the memory is independent of the simulator and vice versa.

In favor of the option to easily exchange the implementation of the memory, the second solution has been chosen.

With the encapsulation of the memory space, the memory access function is a member function of the memory object. The method's implementation depends heavily on the implementation of the memory bank. Therefore generic memory access functions are no longer supported.

Interfaces

Interface class memory_if

All memory implementations are inherited from the class `memory_if`. This results in a common interface to all memory banks. This interface provides the semantics of the memory access functions.

The memory interface contains the start and the end address of its memory space. Besides accessor functions for memory statistics, the interface provides functions for the translation of an address to the host memory address (`mem_translate`) and a function for the allocation of a new memory page (`mem_newpage`). Most importantly it provides a generic access function (`mem_access`) and several memory accessor routines such as string copy (`mem_strcpy`) and copy of n bytes (`mem_bcopy`).

memory_if
simmem start_adress, end_address mem_ptab mem_bus_width
mem_translate(addr) mem_newpage(addr) mem_access(cmd, addr, *vp, nbytes) mem_strcpy(cmd, addr, *s) mem_bcopy(cmd, addr, *vp, nbytes)

Table 3: members and Methods of the interface class memory_if

Memory access through the wrapper

From the instruction set simulator's point of view, the memory space behaves like a single memory bank. The simulator accesses the memory space through the wrapper. The wrapper contains the memory interface `iss_mem_if`. Although this class is inherited from memory_if, it does not provide the implementation of the memory space. It only decides on behalf of the address, to which memory bank the transaction is forwarded to. By hiding the memory space from the instruction set simulator, the simulator does not rely on the implementation of the memory bank and how the content is stored.

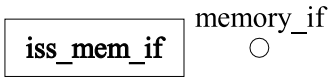

Illustration 23: class iss_mem_if

SimpleScalar's memory implementation

The memory class `memory_t` is inherited from the abstract base class `memory_if`. This class provides the implementation which was previously used in SimpleScalar. This is the place where the simulator finally commits the transaction to the host's memory. More details are provided in the SimpleScalar source code [14].

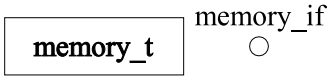

Illustration 24: class memory_t

SystemC system model

The Module sc_processor

The processor model, the module `sc_processor`, does not contain any processes itself. It is a container for the various SystemC components which model the processor: `sc_wrapper` that contains the instruction-set simulator, `sc_bim` which models the interface to the bus, `sc_tcmem_if` the interface to tightly coupled memory, other derivatives of the memory interface `sc_mem_if`, clock and reset ports and ports for input/output signals.

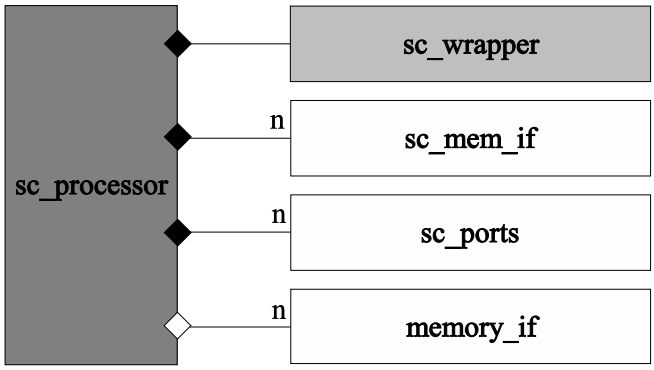

Illustration 25: UML chart of the processor model

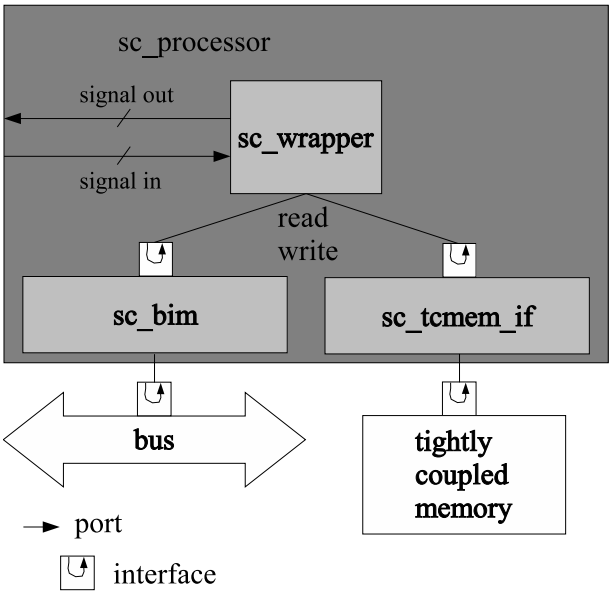

Illustration 26: the module sc_processor

Interface to SystemC memory models

The class `sc_mem_if` defines the interface methods to the memory banks modeled in SystemC. It comprises of following methods: `read(*token)` performs a read transaction, `write(*token)` performs a write transaction and `ready()` shows whether the interface is busy or not.

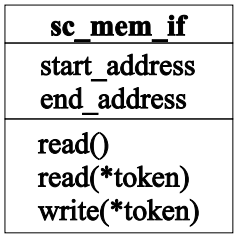

Illustration 27: members and methods of the module sc_mem_if

Bus Interface Model

The purpose of the bus interface model is to connect the wrapper with the bus. The class `sc_bim` is inherited from the class `sc_mem_if`.

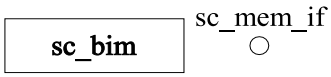

Illustration 28: class sc_bim

When the bus interface model receives a memory access token, it issues a read or a write event respectively. Two threads are sensitive to these events, one to read events and one to write events. After receiving the events they execute the `read_bus` method and the `write_bus` method respectively. These methods perform a blocking or non blocking bus transaction, depending on the mode defined in the token. The bus interface requests access to the bus, and when the arbiter grants access, the interface performs the memory transaction. After completion of the transaction, the bus interface model sets the completed flag of the token.

The module has an input port for the clock and three interface ports to the bus: one for direct access to the memory, one for non blocking transactions and one for blocking transactions. The bus interface model can only use one interface at a time.

Interface model to tightly coupled memory

The interface connects directly to a memory, in the projects case a memory module of the Simple Bus library. It is also inherited from the SystemC memory interface class `sc_mem_if`. The behavior of the module `sc_tcmem_if` is the same as the behavior of the bus interface model. Except that the interface model to tightly coupled memory does not have to issue a request and wait for admission. The interface is able to access the memory model immediately.

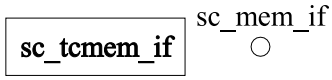

Illustration 29: class sc_tcmem_if

The Simple Bus Library

For the bus and memory model, the Simple Bus library has been chosen. The Simple Bus library is part of SystemC version 2.0 [25]. The Simple Bus model consists of master and slave modules, a bus module and an arbiter module. Modules communicate via channels. Channels are a set of interface methods. They can be hierarchical.

The bus module is such a channel. Multiple master modules connect to the bus through ports. When one or more bus transactions are requested, the bus calls the arbiter in order to select a master. In the Simple Bus library, the arbiter is based on a priority scheduling scheme. The arbiter grants access to the master with the highest priority.

Memories are connected to the bus as slave modules. Slave modules are also channels and feature certain interface methods. In case of a memory, these are `read` and `write`, and `direct_read` and `direct_write`. The bus provides the following interfaces to master modules: a direct interface, a non blocking interface and a blocking interface.

The direct interface allows instantaneous access to the slave without advancing the simulation clock. It should only be used for debugging purposes. The non blocking interface allows a single transaction on the bus. The blocking interface allows burst transactions over the bus, enabling a series of transactions without being interrupted.

Performance of the Simulation Framework

Performance Determinant Parameters

Simulation speed depends heavily on the host's performance. Not only CPU speed is important, but also memory size and access time and because of the use of virtual memory, the hard disk drive throughput. Following parameters of the host computers have been measured:

device	***parameters***		
CPU	clock speed	cache size	MIPS
RAM	capacity	memory bus speed	throughput
HDD	throughput		

Table 4: host computer parameters

Parameters of the simulated system influence also the simulator's performance. Following

module	***parameters***	
application	frequency of memory references	
instruction set architecture	cache properties	number of memory ports
memory space	number of memory banks	latency of the memory banks
system architecture	number of processors	number of busses

Table 5: architecture parameters

parameters can be adjusted on simulator's architecture:

Host computers

Comparison between the two host computers:

Host 1 is a Dell Precision 340 workstation with a Intel P4 processor. Host 2 is a Dell Inspiron 8100 notebook with a Intel PIII mobile processor.

CPU	***clock speed***	***cache size***	***MIPS***
Host 1	2000 MHz	L1: 8kB; L2 512 kB	5300 MIPS
Host 2	1200 MHz	L1 16kB, L2 512 kB	3200 MIPS

RAM	***capacity***	***memory bus speed***	***throughput***
Host 1	512 MB	800 MHz	2400 MB/s
Host 2	256 MB	133 MHz	850 MB/s

HDD	***throughput***
Host 1	22 MB/s
Host 2	8.2 MB/s

Table 6: comparison between the host computers

Measurements

Experiment 1

The simulated system consisted of two processing units with their own memory which was accessed over a shared bus. The benchmark application was a floating point computation called test-fmath which is part of the SimpleScalar tool set.

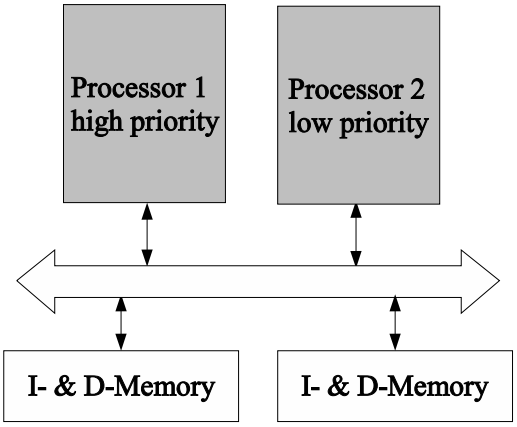

Illustration 30: setup for experiment 1

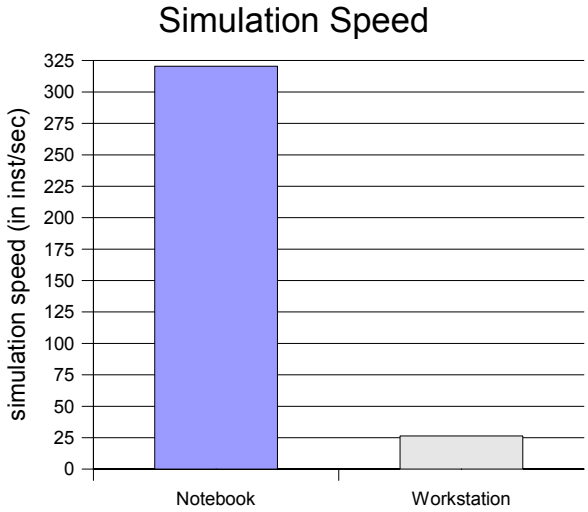

Chart 1: simulation speed for 200'000 simulated cycles

The chart shows the number of simulated instructions per second for Host 1 and Host 2. In total, 200'000 cycles were simulated on the system.

These measurements show that this simulator is between 3 to 4 magnitudes slower than the stand-alone instruction-set simulator.

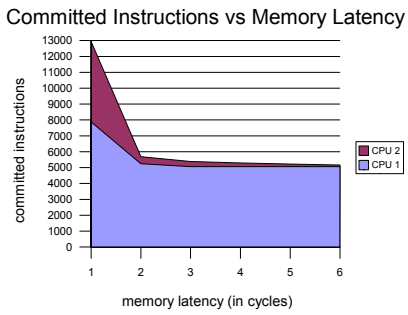

Chart 2

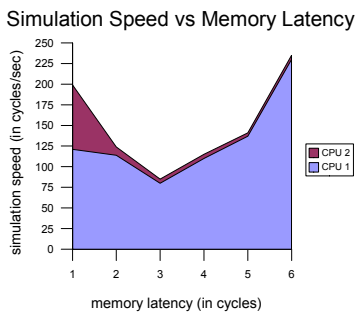

Chart 3

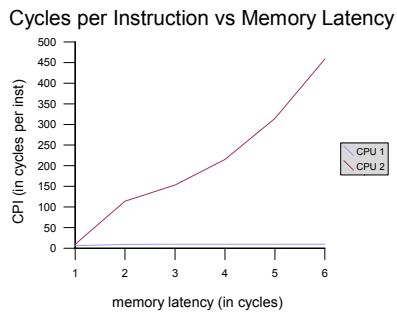

Chart 4

Charts 2-4: the effect of the memory latency on the simulator's performance (simulation time: 50k cycles)

Chart 2 shows that a memory latency of nearly zero, yields a very high simulation speed, since the processors block the bus for only a very short time. Since only few cycles are wasted for memory transactions, the processors work almost in parallel and commit nearly the same number of instructions, as seen in the left chart. A high memory latency also results in a high simulation speed, since the processor with the higher priority (CPU 1) constantly accesses the bus and the requests of the second processor are never handled. The processor with the lower priority is starving and the system behaves like a uniprocessor architecture. In multiprocessor systems, a starvation of a processor cannot be tolerated and therefore memory latencies must be kept to a minimum.

Experiment 2

In the second experiment, each processor has direct access to its dedicated instruction memory. The program used for this experiment is an integer benchmark call test-math, which is also part of the SimpleScalar tool set.

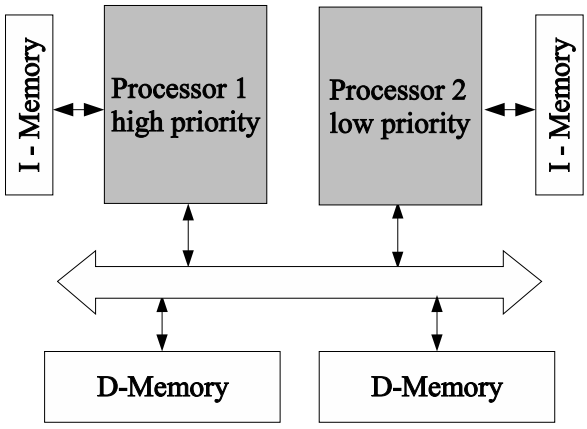

Illustration 31: setup for experiment 2

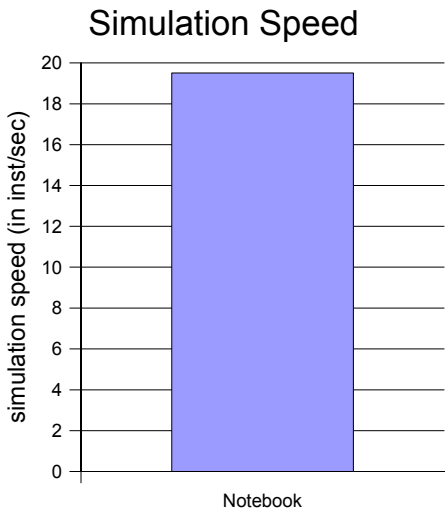

Chart 5: Simulation Speed for 200'000 simulated cycles

In this setup, the processors only have to access the bus for data transaction while instruction fetches are executed in parallel. This reduces bus congestion significantly, but also puts higher demand on the simulator, since up to three memory transactions are performed concurrently.

Compared to the standalone SimpleScalar simulator, the slowdown amounts to approximately 4 magnitudes.

Conclusion

Results

The simulation framework implemented in this project allows designers of multiprocessor Systems-on-Chip to thoroughly explore their architecture. Through the use of interfaces, the framework allows using diverse levels of abstraction for modeling different parts of the system.

The integrated SimpleScalar tool set supports customized applications, thus designers can run their own dedicated programs on the simulated system. The presented framework allows analysis of system behavior with deep insight into the instruction set architecture.

In this project the SimpleScalar instruction-set simulator was integrated in SystemC. The framework is not limited to the use of SimpleScalar, other instruction-set simulators can also be integrated.

Besides co verification of hardware and software, this framework also facilitates measurement of various metrics of the simulated architecture such as bus performance, memory usage and overall throughput. It even supports detailed analysis of processor statistics like branch prediction and cache performance.

Future Work

The presented framework can be used in the future for the analysis of sophisticated Systems-on-Chip. Especially multimedia network processors are of big interest. Application specific benchmarks can be written for the instruction-set architecture such as real time decoding of data streams in multimedia systems or packet routing in network processor.

Follow up projects can integrate other instruction-set simulators and compare their performance with SimpleScalar.

The modules have been designed with the focus on correct behavior of the the simulation framework. Yet, there is still room for a more efficient implementation. Especially memory usage can be improved. By eliminating accesses to the swap space on the hard disk drive, considerable improvement to simulation speed can be achieved.

The simulator's results can be compared to results acquired with analytical methods. Thus it can be verified whether analytical methods are applicable to real time multimedia applications.

Acknowledgments

I would like to thank my tutor Alexandre Maxiaguine and my supervisor Prof. Dr. Lothar Thiele for their help and support during this project. I am also thankful for the detailed documentation of SystemC by the European SystemC Users Group. Finally I thank Todd Austin, for providing the source code of SimpleScalar free of charge to academic users.

Appendix

Appendix A: Command Line Parameters

The compiled simulation executable `sim-outorder.exe` requires two parameters for each processor model. The first parameter represents the path to the configuration file for the instruction-set simulator. The second parameter represents the path to the binary, that is executed by the instruction-set simulator. These two parameters are repeated for each processor model.

```
usage:

sim-outorder config binary [config binary] [config binary] [ ...]

        config    specifies the configuration file
        binary    specifies the executed binary file
```

Command line parameters

Information on the syntax of the configuration file is provided in the SimpleScalar source code [14].

Appendix B: Acronym Dictionary

BIM	Bus Interface Model
DES	Discrete Event Systems
FIFO	First In, First Out
HDL	Hardware Description Language
HLL	High Level Language
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
LSQ	Load/Store Queue
OSCI	Open SystemC Initiative
RUU	Register Update Unit
SoC	System-on-Chip

Bibliography

[1] Samajit Chakraborty, Simon Künzli, Lothar Thiele, "Performance Evaluation of Network Processor Architectures: Combining Simulation with Analytical Estimation" ", Swiss Federal Institute of Technology (ETH) Zürich, Zurich, Switzerland, 2002
[2] Patrick Crowley, Jean-Loup Baer, "A Modeling Framework for Network Processor Systems", University of Washington, Seattle, USA,
[3] Lisa Guerra, Joachim Fitzner, Dipankar Talukdar, Chris Schläger, Bassam Tabbara, Vojin Zivojnovic, "Cycle and Phase Accurate DSP Modeling and Integrationn for HW/SW Co-Verification", DAC 99, New Orleans, USA, 1999
[4] Andreas Hoffmann, Tim Kogel, Heinrich Meyr, "A Framework for Fast Hardware-Software Co-Simulation", IEEE 2001
[5] D. Becker, R. Singh, S.Tell, "An Engineering Environment for Hardware/Software Co-Simulation", DAC, USA, 1992
[6] Mentor Graphics Seamless. www.mentor.com/seamless
[7] Synopsys modeling tools. www.synopsys.com/products/sld/sld.html
[8] Bob Cmelik, David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", ACM, Inc., USA, 1994
[9] Green Hills Software Inc. SimARM. www.ghs.com
[10] ARM ARMulator. www.arm.com
[11] Alpa Shah, "ARMSim: An Instruction-Set Simulator for the ARM processor", Columbia University, USA,
[12] Jack Veenstra, Robert Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors", MASCOTS, USA, 1994
[13] Tensilica Xtensa Instruction Set Simulator. www.tensilica.com
[14] SimpleScalar . www.simplescalar.com
[15] SystemC . www.systemc.org
[16] Joachim Gerlach, Wolfgang Rosenstiel, "System level design using the SystemC modeling platform", University of Tübingen, Germany,
[17] Stuart Swan, "An Introduction to System Level Modeling in SystemC 2.0", Open SystemC Initiative (OSCI), USA, 2001
[18] Gurindar Sohi, "Instruction issue logic for high-performance, interruptible, multiple funtional unit, pipelined computers", IEEE Transactions on Computers 1990
[19] Kunle Olukotun, Mark Heinrich, David Ofelt, "Digital System Simulation: Methodologies and Examples", Stanford University, Stanford, USA,
[20] James Rowson, Alberto Sangiovanni-Vincetelli, "Interface-Based Design", DAC, Anaheim, USA, 1997
[21] Ilia Oussorov, Wolfgang Raab, Ulrich Bachmann, Alex Kravtsov, "SystemC - based development flow from abstract C++ to ISS integration", European SystemC Users Group, Germany, 2002
[22] Bruce Eckel, "Thinking in C++", Mindview Inc., Web Edition, 2000
[23] Erich Gamma, "Design Patterns", Addison-Wesley, USA, 1995
[24] Todd Austin , "SimpleScalar Hacker's Guide", Intel MicroComputer Research Lab, USA, 1997
[25] Ric Hilderink, Thorsten Grötker, "Transaction-level modeling of bus-based systems with SystemC 2.0", Synopsys Inc., USA, 2001