University of Twente
The Netherlands

Implementing and communicating with SHILS

Master's Thesis
by

Remco Blumink

Committee:
ir. P.T. Wolkotte
dr. ir. A.B.J. Kokkeler
ir. P.K.F. Hölzenspies

University of Twente, Enschede, The Netherlands
August 25, 2008

Abstract

Simulation can be used to check whether a design complies to its specifications. Digital hardware designs must be simulated cycle-true, bit accurate to verify timing. Performing such simulations takes prohibitively long for large hardware designs, i.e. a 6x6 NoC design requires 29 hours for simulation.

Simulation on an FPGA platform can be used to shorten the simulation time. However, a large hardware design can not be simulated in a single FPGA as a whole. To be able to simulate a large system using a single FPGA, the system is divided in sections (entities) that are simulated sequentially. The entities in homogeneous systems are identical, the required logic for an entity can be reused for all entities when the state of the entities can be extracted. For extraction, a tool is provided. The resulting simulator performs simulations on a Design Under Test (DUT) sequentially.

To simulate a system in an FPGA, it must be supplied with stimuli and control. This thesis integrates the simulator in an FPGA design, referred to as SHILS. SHILS provides stimuli and control buffers, and a MMIO interface. SHILS is controlled by software in an embedded processor, it is a co-simulation system.

In order to extend the capabilities of SHILS, a design is proposed to link MATLAB to SHILS. The design is based on Xilinx System Generator, which arranges the communication between the MATLAB model and SHILS over Ethernet.

Contents

List of Acronyms

AHB	Advanced High-Speed Bus
ASIC	Application Specific Integrated Circuit
BCVP	Basic Concept Verification Platform
CAES	Computer Architecture for Embedded Systems
DSP	Digital Signal Processing
DMA	Direct Memory Access
DUT	Design Under Test
EBI	External Bus Interface
EDA	Electronic Design Automation
EDIF	Electronic Data Interchange Format
FIFO	First In-First Out
FSB	Front Side Bus
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
GUI	Graphical User Interface
HDL	Hardware Description Language
HIL	Hardware-in-the-loop
IC	Integrated Circuit
IIR	Infinite Impulse Response
ISE	Integrated Synthesis Environment
LUT	Look Up Table
MMIO	Memory Mapped In-/Output
NoC	Network on Chip
OSI	Open Systems Interconnection reference model

SoC	System on Chip
SHILS	Sequential Hardware-In-the-Loop Simulator
RAMP	Research Accelerator for Multiple Processors
RTL	Register Transfer Level
VHDL	Very High Speed Integrated Circuit Hardware Description Language

CHAPTER 1

Introduction

1.1 Motivation

When a system is designed, it is important to know whether the design is *correct*. Knowledge on this matter is evenly important for digital systems and digital hardware designs. Technology for digital systems has evolved greatly because of advances in Integrated Circuit (IC) technology. More flexible chips have become available, like Field Programmable Gate Arrays (FPGAs) and nowadays a complete system is constructed on a single chip (a System on Chip (SoC)) instead of several chips mounted on a Printed Circuit Board(PCB). Also, there is a tendency to design multicore processors, possibly connected to each other via a Network on Chip (NoC).

Development of a SoC is a very complex task, but can be structured by the separation of communication and computation. This can be implemented by means of a NoC [2]. Still, errors and mistakes are introduced in system design. SoCs can be realized in an Application Specific Integrated Circuit (ASIC). Production of an ASIC is extremely costly. Therefore, design errors need to be eliminated. Besides design verification, the facility to optimise the network's parameters is desirable to achieve the best possible results.

Before continuing, several terms should be defined to avoid confusion.

Definition* 1** (Verification):*** *Based on a* ***complete*** *specification (using pre- and postconditions), give a solid proof that the postcondition holds given the truth of the precondition. Thus, if the precondition does not hold, the postcondition needs to fail too. This also holds for the system's specification and its implementation.*

In verification, an important assumption is that a system is accurately specified by means of pre- and postconditions. That is to say, if the precondition is true before an execution of the system starts, then the postcondition must be true after execution. If a system is specified using pre- and postconditions, than its correctness can be *formally verified* (proved): just assume that the precondition holds for all input of the system, and prove that the output satisfies the postcondition. However, in practice this technique is only possible for fairly small components of a system, the system as a whole is often too complex to be proven correctly in a formal way. In real life, the best way to check whether a system is correct, is by simulation:

Definition* 2** (Simulation):*** *To represent the behaviour of a design by using another system, for example by a computer program designed for this purpose. Characteristic is that simulation should imitate the internal processes and not necessarily deliver the results of the design that is simulated for the purpose of detailed analysis and prediction. Also, the execution time of the simulated entity does not necessarily match the real time.*

The term verification by simulation is used when applying simulation to check the correctness of a system. This is not formal verification as only can be proven that in certain conditions a system will fail or pass. Nevertheless, verification by simulation is often used in system design to replace formal verification.

Using simulation, the behaviour of a design can thus be examined. Simulation of a synchronous digital system is possible at multiple abstraction levels, but is mainly performed at two levels: abstract (behavioural) and cycle-accurate (precise). Behavioural simulation focuses on the way a design behaves, the exact number of clock cycles an operation requires are not taken into account. An advantage of this method is that behavioural simulation is reasonably fast to compute on a normal PC. Simulation is completed in an acceptable amount of time.

Another, often used, method is cycle-accurate or cycle-true simulation. The main difference with behavioural simulation is that timing is included in simulation. Each operation takes exactly as many clock cycles as a real ASIC. Calculating all states in each time step creates a greater computational complexity, which eventually leads to prohibitively long simulation times when designs are large. For example, a 6x6 NoC described in the SystemC language requires 29 hours to complete [39].

When simulating a complete system, only system-wide details are required by the user. Internal details are required for precise timing and are thus calculated every simulation step although system-wide data is examined.

1.1.1 Hardware-based simulation: emulation

Due to the problems which arose with the time needed to complete a cycle-accurate simulation, and the availability of parallel hardware, several initiatives started to investigate the possibilities of using FPGAs for simulation. By creating the hardware in the FPGA, the simulated design can be operated fully in parallel instead of sequentially in software on an ordinary PC. This will reduce the total simulation time by several factors.

Performing simulation in physical hardware is sometimes also referred as emulation. For the sake of completeness, emulation is defined in definition 3.

Definition* 3** (Emulation):*** *A system which is behaving exactly as a real system, but is mapped to a different system. The main difference with simulation is that to the system's environment, the system acts fully identically compared to the system it replaces. It is possible to replace a real design with an emulator in a system without notice of the other system parts. For developers/debugging, an emulator often provides for several facilities to monitor variables inside the emulator.*

Several research initiatives started exploring the use of emulation for verification of designs, which are discussed in more detail in section 2.4.

At the University of Twente, several platforms for streaming data applications are developed and researched.

Intermezzo: Streaming data

Streaming data is a model which is used when large quantities of data arrive continuously. Characteristic is that storage of this data is not needed for a long period, as it is either impractical, or just unnecessary. There are several applications which naturally generate vast amounts of data which arrive continuously, instead of simple sets of data arriving accidental, for example multimedia applications (IPTV etc.), financial markets and news organizations [28].
Focus of the University is mainly at multimedia applications.

These streaming applications are mapped on homo- and heterogeneous tiled architectures which often incorporate parallelism and have a more or less regular structure, composed out of a lot of identical components. An example of this is a SoC build of 9 elements interconnected via a NoC as shown in figure 1.1. The processing elements (PE) in this figure are displayed smaller than the router for ease of drawing. This is not necessarily the case in reality.

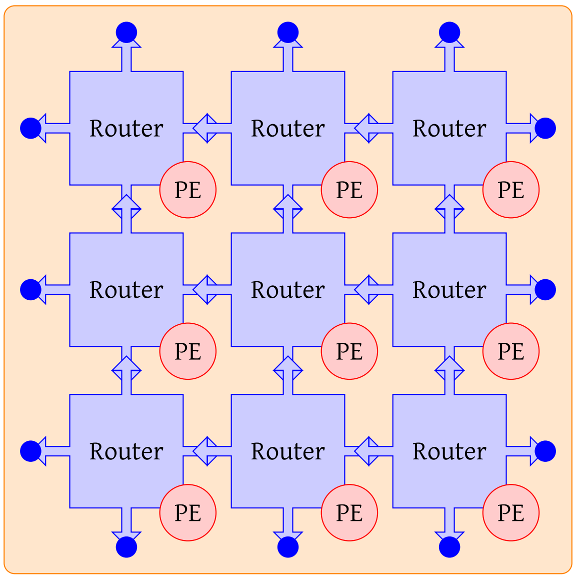


Figure 1.1: System on Chip Example

Systems that are just slightly larger than this example (like a NoC of 6x6 elements), are virtually impossible to simulate in software [38]. Therefore, research effort has been put into the development of a dedicated simulation system.

1.1.2 Sequential Simulation

Systems like the example of figure 1.1 require too many resources to be realized in a single FPGA to be used as emulator. For example, the RAMP project [3] uses a lot of very expensive FPGAs to map a multiprocessor on. To reduce size and cost allocations, several initiatives create a system which operates in a single FPGA by using time-multiplexing [12, 39]. System-level simulations are then performed sequentially in parallel hardware instead of parallel on a sequential system.

A digital system is represented by its state and logic. When logic and state become too large, it is not possible to update the entire state at a single moment due to resource constraints in an FPGA.

It is possible to partially update the state of a system. This is achieved by dividing the state is into several sections, which are then sequentially updated. These sections are referred to as entities. The number of logic needed to update an individual section of the system state is significantly less than the required logic for the entire system. Common functionality of entities can be shared by all state updates of individual sections. The logic can be reused, therefore fewer resources are required. This enables the use of a single FPGA for simulation of the entire system (when the entity requires no more resources than available in a single FPGA). The example system shown in figure 1.1 can be simulated in sections based upon the reuse of logic from single router and processing element. The modification of the figure is shown in figure 1.2.

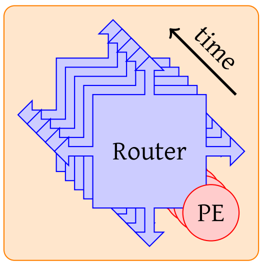


Figure 1.2: Sequential System on Chip Example

The state of digital synchronous systems can alter both on the rising edge and the falling edge of the system clock. Therefore, the entire system should be evaluated for both parts of a clock period. An entity can be simulated during each simulation cycle. A single instance simulation cycle is addressed as delta cycle for readability. An example system consisting of four identical parts requires eight simulator clock cycles to complete simulation of one system clock cycle, which is shown in the example schedule in figure 1.3.

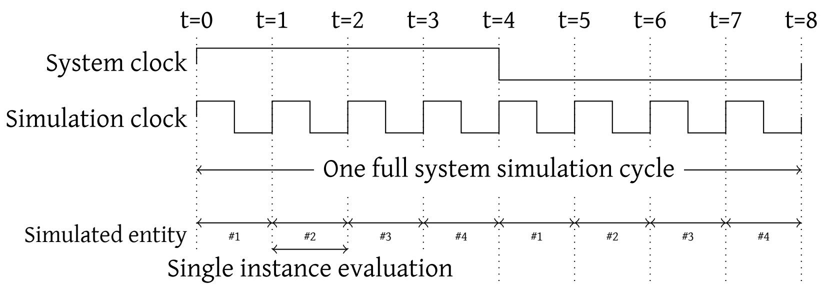
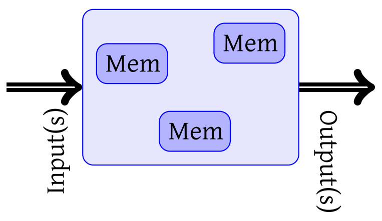


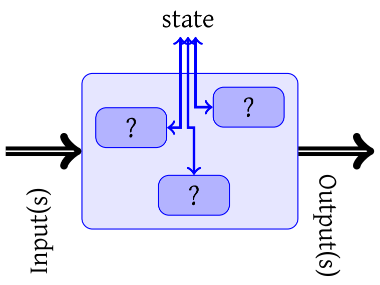
Figure 1.3: Global simulator time line

To simulate an entity with the sequential simulator, the state of the other entities should be stored elsewhere. After all, the logic is reused in simulation. The registers and other memory elements are replaced by custom components that link to an external storage location for their values. Thus instead of an entity with some memory elements shown in figure 1.4a, the memory elements are replaced with a component that fetches and updates the corresponding element of external state as shown in figure 1.4b. By storing the entire state of a system in an external memory, more efficient storage is provided compared to flip-flops as the density of RAM in FPGAs is much higher compared to that of registers.

In this way, at any moment in time, only a small section (a single entity) of the entire system is used in the simulator. Therefore, the number of required resources are much lower than for the complete system. In homogeneous systems the logic for most entities is identical, which lowers the required resources further.



(a) Original design



(b) Design with extracted memory elements

Figure 1.4: State extraction example

Of course, emulating a system sequentially requires more time than a fully parallel system, but this is several orders of magnitude faster compared to simulation in software. To control the simulator, hardware is added to the FPGA.

Efforts from Rutgers [33] at the University of Twente delivered a tool which:

1. Automatically extracts memory elements out of a design.
2. Provides a wrapper around the transformed entity, which provides memory for the extracted state of the simulated entity and the links between entities.

Rutgers has also provided in the simulator, to which is referred as “the simulator” in this thesis.

The simulator requires external control and the simulator should be provided with stimuli. The simulator is designed to operate in a system consisting of an FPGA that houses the simulator and a processor to provide it with stimuli, according to the Hardware-in-the-loop (HIL) principle. However, the external interface of the simulator is not directly linked to any physical interface like a Direct Memory Access (DMA) interface. The simulator is also not equipped with buffering capabilities for both stimuli and the output data of simulated entities. Therefore, the simulator delivered by Rutgers is not yet able to perform simulations.

1.2 Assignment

This thesis project started to provide the simulator with external control and stimuli. It is centred around communication between the simulator and processor. This section discusses the problems dealt with in this thesis.

The main focus of this thesis is:

> What measures are needed to connect a stimuli generator and an analysis program operating on a PC to the sequential simulator operating in an FPGA?

This question is based upon two pillars, which explain why this problem is discussed in this thesis.

First, the simulator delivered by Rutgers was not ready for usage in the FPGA. A major concern is that the simulator and processor operate at different clock frequencies. Also, the frequency of producing and consuming differs. Therefore, buffering is required. The first part of this thesis is thus focused at:

> How to implement the simulator in an actual FPGA, how to connect it to a processor and how to make them communicate?

Second, as noted in [38, 39], there is a significant effort required of a small processor for stimuli generation. Since a PC is equipped with a much greater amount of processing power, it is very interesting to replace this processor with a PC to save more processing power for analysis and control tasks. Also, software is available for data generation and analysis. The second question is formulated as follows:

> How to connect a PC with the sequential simulator on the FPGA, how to make them communicate and process data?

The PC-based simulation system must have better performance than the system using an embedded processor. Also, tool chain integration is important for more easy generation of stimuli and analysis.

For readability, SHILS is used to refer to the complete sequential simulation system hardware. Statements on the simulator exclusively apply to the simulator delivered by Rutgers.

1.3 Thesis Outline

This thesis begins with background information on sequential simulation and related work. To introduce sequential simulation, chapter 2 starts with a discussion of a basic FPGA design flow, and indicates the required adjustments for sequential simulation. Chapter 3 discusses the design of the initial version of the simulator which has been tested on a hardware platform, using an example design. The implementation of this design is discussed in chapter 4. Implementation specific issues for the example are discussed in appendix B. Chapter 5 discusses PC-based tools, which could provide in a method to connect the simulator with a PC, formulates selection criteria and selects the best suitable product. Chapter 6 discusses the design of a co-simulation system based upon the selected tool. The results of the implementation are discussed in chapter 7.

CHAPTER 2

Background

This chapter introduces a number of topics, which are important for the SHILS discussion, they create the context for it.

Before describing the sequential simulator in detail, its role in the system design process should be pointed out. Therefore, the normal system design process is discussed first. The sequential simulator is designed to aid in the verification process of a design. Using the sequential simulator in an existing design does not require many additional steps, but requires attention throughout the entire process.

SHILS is a co-simulation system. Co-simulation is discussed in general in section 2.2.

The discussion of sequential simulation also introduces the sequential simulator design of Rutgers [33] in more detail, focused towards usage of the simulator.

2.1 System design process

The design process is based upon the Waterfall model, which is briefly discussed next.

The Waterfall model (shown in figure 2.1) is a very well known model, introduced in 1970 by Royce [32]. In this iterative model, each successive step adds more detail to a design. The main benefit of this approach is that the number of changes is manageable, and it is possible to return to an earlier phase if unforeseen problems occur. This process is referred to as feedback.

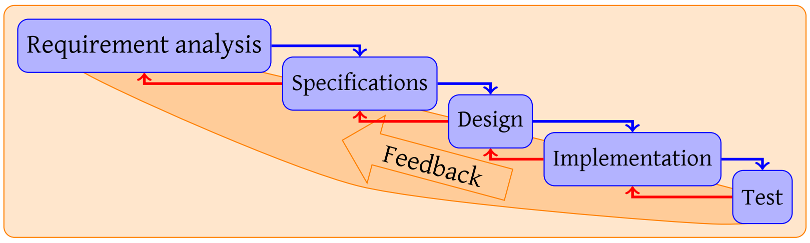


Figure 2.1: Waterfall model

2.1.1 Digital System Design

The methodology of both ASIC and FPGA design are identical, the main difference is the tooling which is used. Therefore, a design that shall be produced in an ASIC, is testable in an FPGA.

A design can be specified at several levels, but most digital systems are designed using a Hardware Description Language (HDL), specified at Register Transfer Level (RTL). The sequential simulator also is designed at RTL. This discussion of digital system design is therefore focused at RTL level.

Globally, the design process can be divided into three steps, as shown in figure 2.2. The Waterfall model is applicable to these steps, and to the internals of the design phase. Verification of a design, the goal of SHILS, takes place prior to the production.

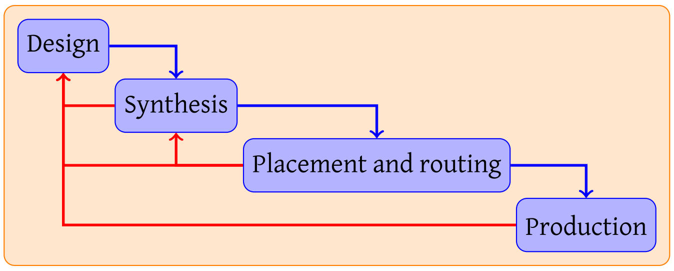


Figure 2.2: Global design process

The synthesis phase translates the RTL design into technology dependent cells that perform a certain function, like flip flops and multiplexers and deals with interconnection of those cells. This is performed in two tasks:

1. Assemble all system parts to create a single integrated system
2. Translate the logic cells in the design to technology dependant cells

Subsequently, the placement and routing phase then performs two tasks:

1. Map the assembly of logic cells to a grid of available resources. In FPGAs the Look Up Tables (LUTs) are mapped into a suitable physical position on the chip.
2. Connect the logic cells which are used in the grid of an FPGA.

In the end, the production phase generates a programming file for an FPGA.

The designer is mostly involved in the first phase. Tooling can solve the other phases. In special cases, the designer influences the solutions of the tooling. Still, designing remains an iterative process with feedback from previous phases, even automated tasks generate feedback that should be used in preceding phases.

2.1.2 Tool chain

Several tools are used to ease the developers' life in system development. This section discusses the tools used for this project. For simulation, QuestaSim from Mentor Graphics is used. This tool, the successor of as ModelSim, is capable of simulating RTL descriptions with a normal PC, which gives maximum flexibility at a reasonable speed for behavioural descriptions. Bit-accurate, cycle-accurate simulations consume quite a lot of time.

PrecisionSynthesis is used for synthesis. This tool, also from Mentor, is used to translate a RTL description to industry standard Electronic Data Interchange Format (EDIF). This describes the logic cells which will be used to physically realise the RTL description.

Finally, the placement and routing phase is performed by Xilinx Integrated Synthesis Environment (ISE). This tool accesses several tools internally:

1. `ngdbuild`, which combines all subparts into a single file combined with constraints.
2. `map`, which maps the logic cells to the actual cells.
3. `par`, which places the cells at positions on the FPGA and interconnects them
4. `bitgen`, which generates the programming file

The tool flow can be highly automated. This saves the designer time and increases the ease of use, and a single command can be used to perform both synthesis, and placement and routing.

2.2 Co-simulation

Verification of a design using SHILS is closely related to co-simulation. Therefore, co-simulation basics are briefly introduced.

Embedded systems are often multidisciplinary, spanning between between pure software design, pure hardware design and mechanics. These disciplines require a combined methodology to create a system. Co-simulation is a powerful technique in virtual prototyping to provide a solution for this [16].

Co-simulation is a simulation which spans across multiple disciplines, and can be performed within a single simulation package capable of performing multidisciplinary simulations, or by connecting several simulators specific for each discipline. For example 20-sim [6] and Ptolemy [4] can be used for multidisciplinary simulations.

A number of simulators specific to a discipline can be linked with a separate infrastructure that deals with interconnection of these simulators [10] or by linking the simulators directly together [26].

Several co-simulation environments already incorporate a form of *distributed simulation*. By distributing the simulation across multiple PCs, it is possible to increase the overall simulation speed as the available processing power is increased. A similar approach could be used to connect the sequential simulator in the co-simulation environment although most co-simulation approaches use only software for simulation.

2.3 Sequential simulator introduction

This section describes sequential simulation in more detail, using the SHILS approach. Also, this section describes the sequential simulator design of Rutgers [33] in more detail. The discussion is focused towards usage of the simulator, the detailed internals are not discussed. SHILS requires a revised approach to system design.

The process of extracting memory elements from a design is extensively discussed by Rutgers [33, chapters 2 and 3]. Usage of the transformation tool is discussed in section 2.3.2.

2.3.1 Simulator design

The simulator framework designed by Rutgers is globally divided into four main blocks. These are depicted in figure 2.3.

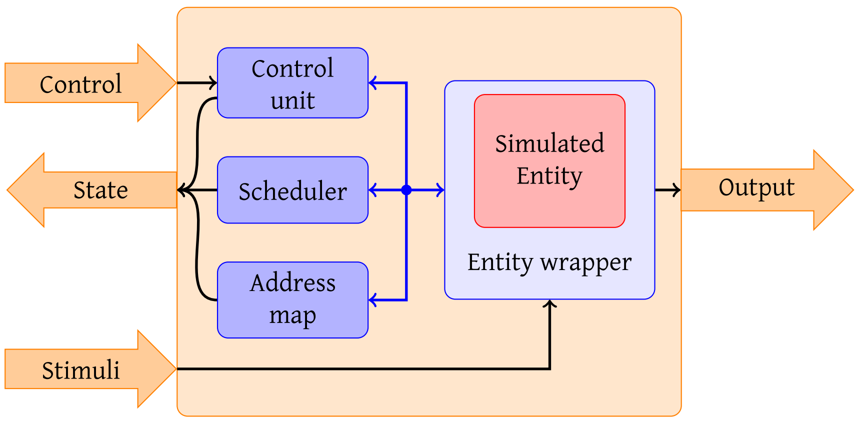


Figure 2.3: Global simulator design

As shown, the simulator is wrapped around a simulated entity. The entity is not an integral part of the simulator, it is purely for simulation of a specific Design Under Test (DUT). To match the entity to the standard interfaces used by the simulator, a separate wrapper is provided by the transformation tool. This wrapper also holds the storage for the DUT's state and deals with linking the entities of the DUT together in simulation; the wrapper is discussed in more detail by Rutgers [33]. The wrapper is controlled by the control unit and signals when the system can advance to a next delta cycle when the DUT has stabilised after data of a next entity has been offered.

Input ports of DUT entities that are not connected internally must be supplied with data. The stimuli input of the simulator provides this data. The simulator is not equipped with any logic to check input data correctness at any moment, it requires data to be ready at the moment it needs the stimuli. The output of entity instances can be examined via the output port. The simulator control state is accessible via an output port.

The simulator is controlled using a set of commands, i.e. "RUN CYCLE", defined by Rutgers. Another command is used to initialize connections between entities in a DUT. For each connection, the arguments of this command specify:

- Which input port reads from what entity instance?
- Which input ports are connected to the stimuli input(if any)?
- Which output port writes to what entity instance?

The information is used by the address map to fetch data from the correct memory addresses of the DUT's state storage in the entity wrapper. The scheduler is provided with a vector of entities that are dependant of the entity instance that is currently simulated by the address map.

The scheduler determines which entity instance is simulated next. The implementation of Rutgers uses a fairly basic Round Robin scheduling mechanism.

The dependency vector is important for the scheduler as an entity instance might change the input of a previously simulated entity instance, that must be re-scheduled for simulation. Information on changed outputs is gained from the entity wrapper. Re-scheduling is depicted in figure 2.4.

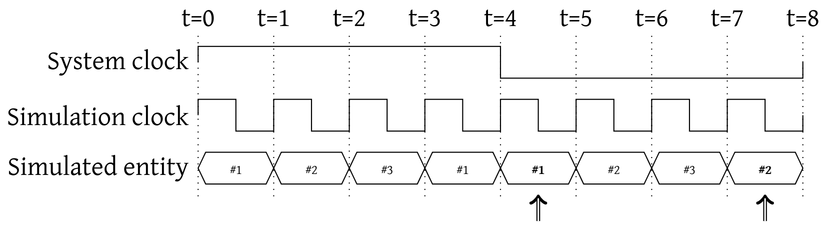


Figure 2.4: Re-scheduling example

To improve performance, the simulator is pipelined. Several steps must be taken before an entity instance can be simulated, which can be performed simultaneously for several instances. Pipelining boosts performance of the simulator, it is capable of processing an entity instance each clock cycle. These simulation cycles are referred to as delta cycles. To complete a system cycle, the simulator must process simulation of all entities for both the “high” period as well as the “low” period of the system clock. This is also depicted in figure 2.4

2.3.2 Using SHILS

To start using SHILS, several steps should be followed. The entire system is too large to simulate in an FPGA. Therefore, the top level description is not usable for simulation, but it is the starting point in the design flow. The system design flow divides into two separate flows, which is depicted in figure 2.5 and discussed next.

The flow of information indicated in figure 2.5 with “manual” must be performed manually for now, as information that is described in the top-level description cannot be extracted automatically yet [33]. The developer should extract the following information manually:

1. How many entities are simulated?

2. How are entities interconnected?

3. What are the global clock and reset signals?

These questions are used in the translation process, and in the actual simulation runs.

After design of the entity, the memory elements can be extracted from it and the entity is transformed using the information gathered manually. That information is also used to integrate the simulator in SHILS, which is implementation specific.

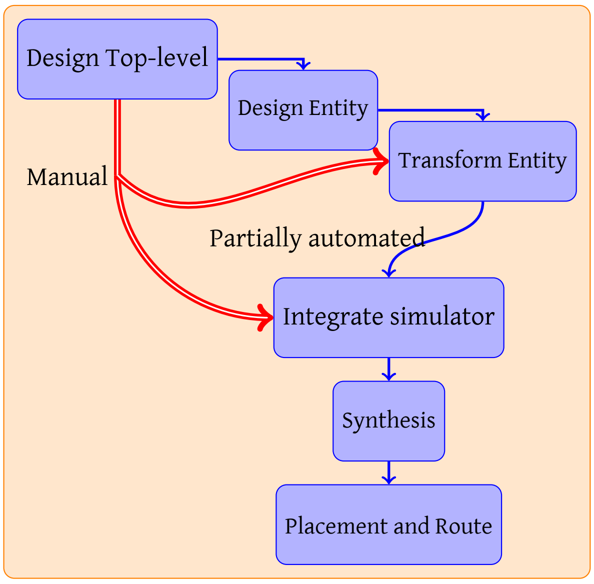


Figure 2.5: Sequential simulator design process

Transform entity

Memory elements in an entity are automatically extracted by a transformation tool delivered by Rutgers [33]. Besides the entity, the tool requires a number of basic arguments:

- The set of possible links between the ports of an entity
- How to order ports internally
- Specification of which signal to use for clock and reset

To specify these arguments, the information gathered in the manual sub flow(section 2.3.2) is used. Rutgers defined additional arguments for several purposes [33, chapter 4].

This tool operates on EDIF files only. Therefore, the design specified in Very High Speed Integrated Circuit Hardware Description Language (VHDL) should be synthesized prior to transformation. The transformation tool requires a number of options to be set in synthesis:

- Do not generate I/O pads
- Preserve hierarchy

This is required for correct integration in SHILS after transformation. Precision-Synthesis, which is used at the CAES chair, uses the following commands to set the required options:

```
setup_design -addio=false
set_attribute -design rtl -name HIERARCHY -value preserve NAMEOFENTITY
update_constraint_file
```

Listing 2.1: Additional synthesis commands for entity transformation

The transformation tool is run using a Makefile, which automatically executes several steps to deliver several items:

- Transformed entity in EDIF and Xilinx-proprietary NGO format[1] as well as VHDL format that can be used for simulation.
- Entity wrapper in VHDL format
- VHDL package with correct bit widths, port specifications etc.

Integrate entity in simulator

After transformation, the entity is integrated in the simulator. Some data widths are not adjusted automatically and should be modified. The simulator integrated in the rest of hardware design, described in chapter 3.

SHILS is then integrated and synthesised using the tool flow described in section 2.1.2. The synthesis tool integrates the transformed entity into the simulator during synthesis of SHILS.

Simulator usage

The synthesis tool generates a programming file which can be inserted in the FPGA. Before simulation, several configuration variables must be set. The transformation tool does not extract these variables automatically, the information gathered in the manual sub flow (section 2.3.2) is therefore used. Configuration is done at run-time, and can be changed if desired (reset is required). These configuration variables are:

Number of simulated entities The simulator is capable of simulating a certain maximum number of entities. This number is set manually in the simulator VHDL code. Per simulation it is selectable how many entities are used for simulation.

Initialize connections between instances Inside the simulator, link memories are used to store data for input of other entities. Per entity, three configuration variables must be set. These are:

- Which input port reads from which entity
- Which output port writes to which entity
- Which input port is connected to external stimuli (if any)

The simulator internally uses different memories to store the read and write actions on entity ports. Therefore, these must be specified separately.

With these configuration steps, the simulation can be run. Of course, for input ports of simulated entities that should be supplied with stimuli, data should be ready.

[1]Basically, the transformation tool could transform any FPGA technology. The current implementation can only translate Xilinx-based designs

2.4 Related work

As introduced in chapter 1, research for simulation of large embedded multiprocessing designs has been extended with (at least partial) hardware-based simulation. Several approaches exist, which are discussed in this section.

The software industry tends towards multiprocessor systems because of the “brick wall” [21]. Compared to research in NoC and SoC verification, there are similarities with ordinary software research in multiprocessing. FPGAs have attracted the attention of software researchers, too due to their flexibility and scalability. According to Chung et al. [12] a slowdown of about 100x in an FPGA compared to a real multiprocessor is acceptable for software research. This figure is supported by figures of Olukotun et al. [29]. Ideas used in this research field are also applicable in simulation and verification of SoC and NoC designs.

2.4.1 Hardware emulation systems

There are several initiatives to port a basic multiprocessor system to an FPGA-based platform and to realise a basic infrastructure for simulating a large number of processors. For example the Research Accelerator for Multiple Processors (RAMP) project [3], can simulate up to 1024 MicroBlaze processors. This project is heavily sponsored by Xilinx and Intel, and has already delivered a few versions of a prototyping board, like BEE2 [9]. This approach is costly, as a single board costs about 20.000 USD and a complete simulation system is constructed out of several of these boards.

2.4.2 Sequential Simulation

Another — cheaper — approach is started by Chung et al. [11, 12] in the ProtoFlex project. The ProtoFlex project is carried out in conjunction with the RAMP project. They present a system which uses time-multiplexing to save FPGA resources and a clever hybrid mechanism to keep implementation complexity low, running at the BEE2 board [9]. The time-multiplexing mechanism used by ProtoFlex is very similar to that of SHILS, an interleaved pipeline is used to sequentially simulate all CPUs which are simulated. The idea behind the ProtoFlex approach is that computation-intensive and often used tasks are performed directly on the FPGA, whereas less used tasks are performed in software. The tasks that are executed by software are either executed in a soft-core processor residing in an FPGA or in an external PC, connected via Ethernet [12] to the FPGA board. A speedup of 39x compared to common used multiprocessor simulation software is obtained [12].

Parashar and Chandrachoodan also present a sequential simulation framework [31]. They present an algorithm intended to simulate synchronous systems of logical processes, using events and a simulator to test their algorithm. The algorithm is for simulation using very basic processing elements, and it eliminates the need to sort the events of single queue event based simulation algorithms. Those processing elements only support simulation of the straightforward Boolean functions AND, NAND, OR, NOR, XOR, XNOR, and NOT, and simulation of an edge-triggered D-flipflop. Nothing is stated about more complex operations. The cycle-true simulator is transaction-based in which scheduling of processing elements occurs at compile time.

The system is implemented using the Verilog Programming Level Interface (PLI) to handle reading and writing of all events. A maximum of 64 processing elements fits in a medium sized FPGA(Xilinx Spartan XC3S1200E) after synthesis. The proposed algorithm is only tested in the ModelSim simulation software, not in physical hardware.

2.4.3 Co-simulation

Several systems create a co-simulation system, a simulation system assembled of both hardware and software. A considerable number of resources is required to implement a complete simulator in hardware. Therefore, several initiatives propose to use the versatility and power of an ordinary General Purpose Processor (GPP) for stimuli generation and analysis combined with the parallel simulation capabilities of FPGAs for the actual simulator.

The aforementioned ProtoFlex project [11,12] applies this form of co-simulation, primarily to save time and resources from implementing rarely used functionality in hardware. The key concepts behind this approach have been discussed in section 2.4.2.

Another example of co-simulation is proposed by Ou and Prasanna [30]. They use co-simulation in an entirely different manner; the novel aspect in their approach is the usage of high-level cycle-accurate abstractions of a low-level implementation to speed up the simulation process. MATLAB is used for the high-level abstractions. The origin of their approach is in the increased usage of soft core processors on FPGAs, and those processors are more customisable than normal processors. Certain parts can easily be performed in hardware, whereas other parts are executed within the soft core. By replacing the FPGA hardware with MATLAB connected to the soft core, complicated calculations can be performed at high speeds with hardware ensuring cycle-accuracy. Simulation of programs executing in a soft core processor is difficult within low-level simulation software like QuestaSim, therefore execution on a target platform is required to benefit fully from the speed-up.

Genko et al. [19] present another approach primarily intended for NoC feature exploration. Their approach uses a Xilinx Virtex 2 Pro (v20) FPGA as hardware platform to emulate NoCs. The system offers designers a platform to quickly characterize performance figures of a NoC, without loosing cycle-accuracy. A speed up of four orders of magnitude compared to cycle-accurate HDL simulation is reached according to the presented results. Different from other approaches is that this platform is suited purely for NoC emulation. Within the FPGA, several NoC topologies can be simulated without re-synthesis by software configuration. Primary results are latency statistics.

The presented results are based on relative small networks; presented figures are for a 2x2 mesh network. Scalability of this approach within a single FPGA is therefore poor.

Other examples of co-simulation are introduced in [16] and [35].

The communication between multiple platforms is often limiting the performance of co-simulation. The improved performance gained by running simulations in hardware can be eliminated by communication overhead easily [13]. Chung and Kyung present an algorithm to reduce the communication overhead between the individual simulators in co-simulation [13]. Their algorithm allows the simulators to run asynchronous if there are no transactions between the simulation models. In order to predict the duration of the interval in which no communication is guaranteed, two methods are introduced by Chung and Kyung. One method focuses at backward tracing of HDL models, whereas the other focuses at software code analysis. Their approach reduces the amount of communication by a factor of 15 to 67, which results in an overall speed-up factor of 4-40 compared to existing lock-step simulation.

Co-simulation systems distribute functionality. One can argue that challenges are similar to those of distributed computing systems; therefore the problems and solutions discussed by Tanenbaum and Van Steen [34] for distributed systems, such as data consistency and data distribution, are also applicable to the co-simulation field.

Co-simulation can be performed by several co-simulation tools. A number of these tools are discussed in chapter 5.

CHAPTER 3

Basic SHILS design

Earlier, it was not yet possible to test the simulator of Rutgers [33] in real hardware, due to the lack of a physical interface with a controller (i.e. a GPP).

To connect the simulator shown in figure 2.3 with the physical world, a unified interface should be used to offer a sturdy connection with simulator control and analysis software.

3.1 System structure

The system is supplied with control and data by an external processor that is connected to the simulator. In this way, the system shown in figure 3.1 is created.

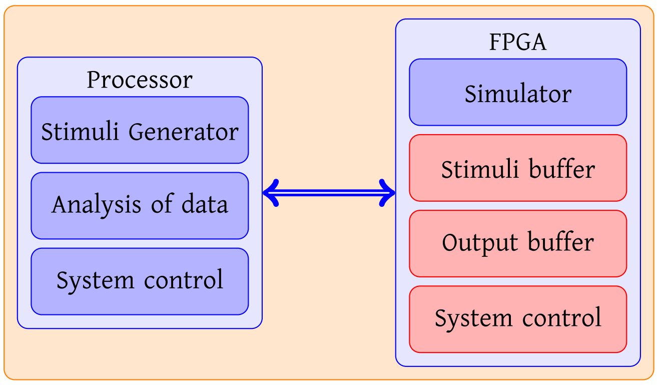


Figure 3.1: Global System structure

The figure shows the division of tasks between the processor and FPGA, as proposed by Wolkotte [39]. This balances both performance as well as flexibility. The interfaces require buffering of the input stimuli, output data and control. These are designed in the FPGA.

Hardware design is related to the targeted technology. The simulator design of Rutgers is not technology dependant, but the transformation tool is currently tailored to Xilinx technology dependant EDIF files.

3.2 Design

The SHILS FPGA design consists of several blocks. This design also incorporates an interface bridge for a GPP's interface. Processors connected to an FPGA often operate at different clock frequencies than the blocks in the FPGA. As a result of this, clock domain transformations are required. This needs to be added to the implementation of the glue logic for certain parts of the system, which is called clock stretching. The blocks in which SHILS is divided are shown in figure 3.2, that also shows the primary flow of data, control and state in the FPGA.

The hardware design of Rutgers [33] is extended with:

- Stimuli buffer
- Output data buffer
- Control buffer
- Interface bridge

The design of these aspects is discussed in the following sections.

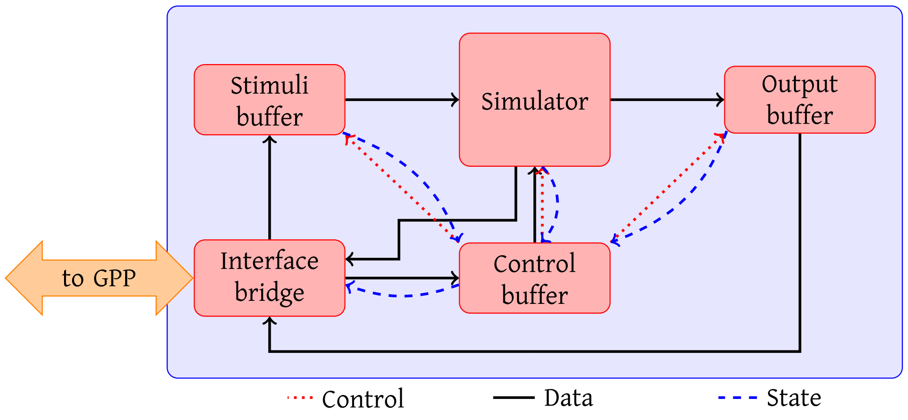


Figure 3.2: System design block diagram

3.2.1 Stimuli

This section discusses general aspects of stimuli which are of importance for the design of the stimuli buffer.

The inputs of simulated entities should be supplied with stimuli, i.e. data. Stimuli can be assembled and offered to the simulator in various methods, which could be divided according the quantity of data which is transported.

The least efficient method of offering stimuli to a system is by sending new data *precisely* at the moment it is required to change. It is very impractical; as the tight coupling requires that the producer and consumer of stimuli operate fully synchronous or use a tight handshake mechanism. A more flexible link is required that decouples production and consumption rates and times.

Stimuli are generated in a GPP as shown in figure 3.1. The operating frequencies of the GPP and FPGA differ; clock domain transition signals that are fully synchronous are hardly possible therefore. Because the amount of communication that is required to keep the system synchronized is very high.

To reduce the communication overhead, several measures are possible:

- Compress stimuli in time
- Produce stimuli in the FPGA
- Compress stimuli in space

Compress stimuli in time

Stimuli will not change every clock cycle, therefore not every clock cycle a sample should be transmitted. To be able to offer data efficiently in this case, a sample is appended with a time stamp. This time code directs the earliest moment at which the stimuli buffer is required to offer the data to the simulator. Compression of stimuli in time is also used by Wolkotte [39].

Adding a time code to stimuli requires additional storage and communication. If stimuli is changing each clock cycle, this is not the best solution to reduce the quantity of date that is transmitted. To reduce the number of bits needed for the time code, relative timing could be used — the time code then signals the number of clock cycles *between* two samples.

Produce stimuli in FPGA

To completely eliminate the clock domain transformation, the stimuli generator could also be moved to the FPGA. This can greatly enhance the system's performance, but significantly limits the flexibility for the designer. This lowers the generator speed, as most FPGAs operate at much lower clock frequencies than GPPs. A less restrictive solution is to generate a time code in the FPGA for stimuli that is generated in software. Time compression is applied in this manner without the additional communication overhead.

Compress stimuli in space

Data compression is a method which is often used to reduce the number of bits required to transmit data. Previously, no data compression methods have been applied to SHILS. As limited experience with data compression is available, this is left as future work.

There is an important aspect on using data compression: the compression and decompression processes require time, which could harm the performance of the system. However, if a compression method is used that is efficiently to compress and decompress, the amount of communication is cut down. This results in less communication overhead and possibly higher SHILS performance.

SHILS approach

As the addition of a timestamp has already been successfully used by Wolkotte [39], it has been chosen to design the system using this approach. Time compression also offers a lot of flexibility and configurability for the stimuli generation mechanism.

3.2.2 Stimuli Buffer design

Figure 3.1 shows that stimuli are generated in a GPP. The link between GPP and SHILS requires the translation of the clock domain. The frequency at which stimuli is generated and consumed differs. The producer and consumer of stimuli must be decoupled for this reason too. Buffering is required to convert clock domains and to decouple producing and consuming frequencies. By decoupling the operating frequency of the simulator and stimuli generator, several issues are introduced, focused towards efficient transport of the data through the buffer.

The buffer has to maintain data consistency — the order in which data enters the buffer should be preserved. This avoids nondeterministic behaviour in simulation and analysis. In many cases generates the stimuli generator blocks of samples, which must be inserted in the buffer rapidly. For optimal data consistency, a useful feature is an interrupt mechanism. When the buffer is almost empty, the processor has to fill it fast. Another option is to temporary pause the simulator in the FPGA, but this has a significant performance penalty.

The simulator has no facilities to control the input stimuli. It just expects data to be available at the moment. Therefore, the buffer also has to be capable of extracting the oldest element out of the buffer and offer it at the right moment in time to the simulator.

Simulated entities can have multiple inputs, for example multiple ports of a router in a NoC. SHILS should be able to supply data for all these inputs. When mutual exclusion is ensured, a single buffer could be shared by multiple ports to reduce resource usage. The simulator knows which port accepts a sample at what moment, therefore the same element can be offered to both ports simultaneously. Still, for input ports which are accessed by the same entity at the same moment, separate buffers or data outputs should be used. This also partially holds for multiple ports within the same clock cycle, an identifier could be used to signal which entity needs which data. The most straightforward method to create a buffer for all inputs is to provide a unique stimuli buffer for each input port. This allows data to be consumed by multiple input ports in parallel. Of course, this approach requires the most resources.

A buffer, using the First In-First Out (FIFO) principle, is chosen for the stimuli buffer. It preserves data ordering and — implementation dependently — offers the buffer to be filled with blocks of data rapidly. This leads to the design shown in figure 3.3. Future choices could lead to stimuli generation in the FPGA. The design already includes facilities for the stimuli generation mechanism and stimuli source selector, these are also shown in figure 3.3. The stimuli buffer is controlled by a controller which manages data consistency and prevents data of being overwritten or incorrectly read. The buffer is provided with separate read and write ports to fully decouple producer and consumer.

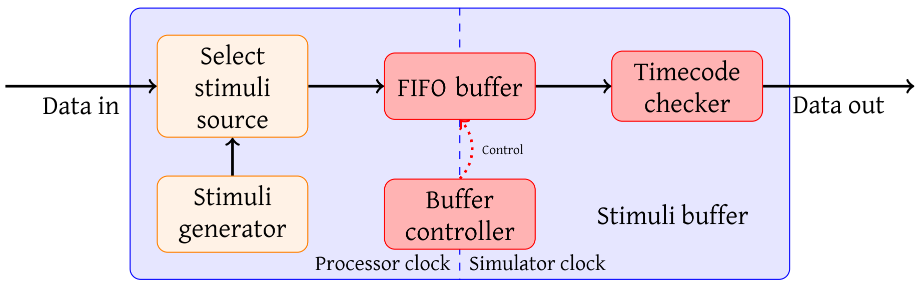


Figure 3.3: Stimuli buffer block diagram

Stimuli buffer output timing

As noted, the simulator cannot control its stimuli input, it expects data to be available. A separate component is introduced to offer stimuli at the designated moment to the simulator: the *time checker*. This component compares the current time with the timestamp of the stimuli element that is the oldest in the FIFO buffer.

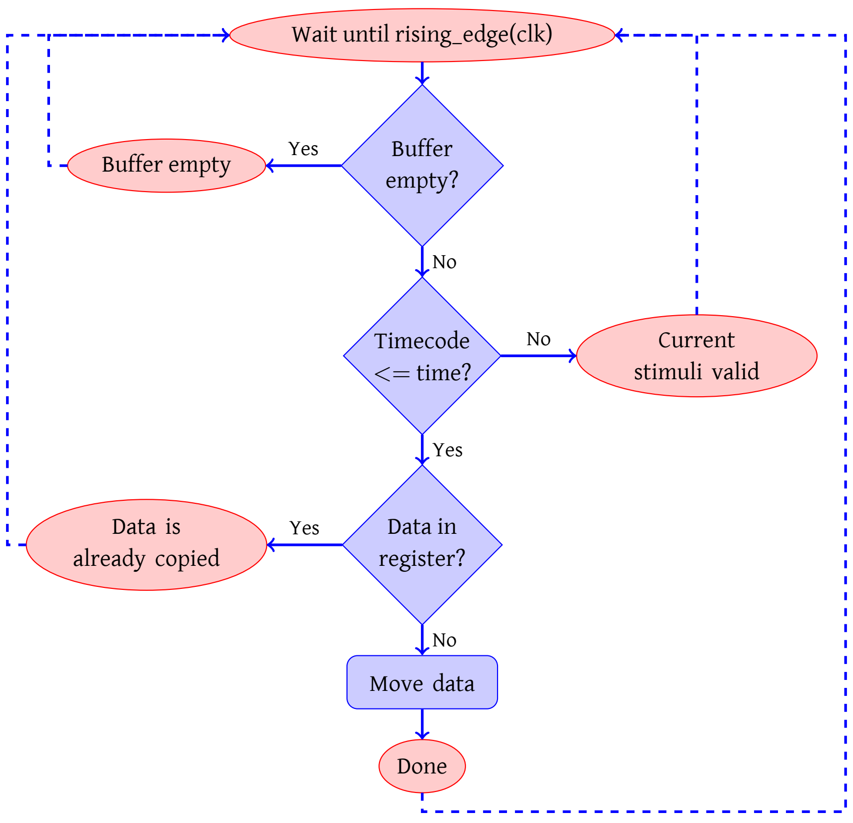


Figure 3.4: Time checker flowchart

It monitors the top register of the FIFO buffer, and copies its value to the output of the stimuli buffer at the intended moment. The process of checking the timestamp is depicted graphically in figure 3.4. To decouple the stimuli buffer and simulator, the checker does not offer stimuli directly to the simulator, it only extracts the correct sample from the buffer. The precise moment at which it is offered is arranged by the interface bridge. Data is ready within one delta cycle.

Timing is important in this stage. Based on [33, figure 5.2], timing of the simulator regarding stimuli is shown in figure 3.5. To clarify the steps, a single flow through the pipeline is shown. In reality, each delta step a new address is generated and thus a sample should be available.

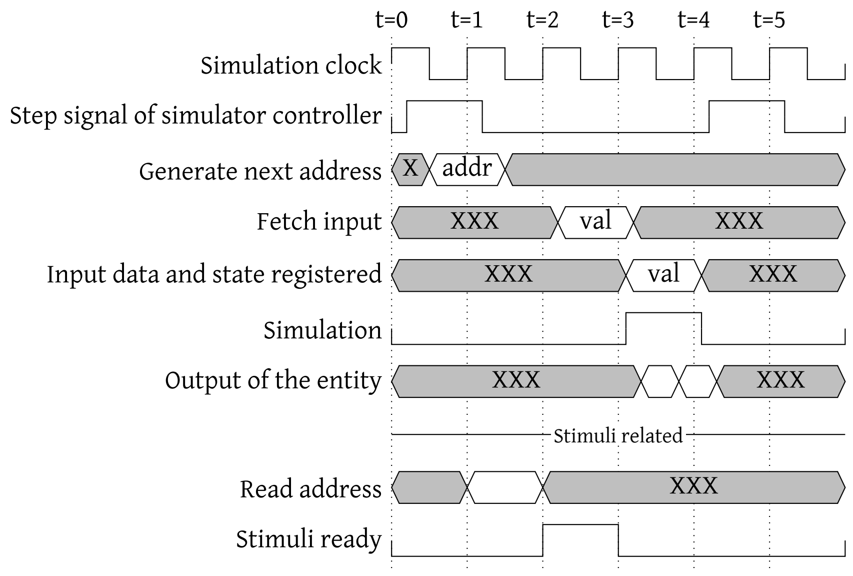


Figure 3.5: Timing diagram based on Rutgers [33]

After the controller of the simulator has given the “step” signal, the address of the instance that is simulated next is generated. It is available at the next rising edge of the clock. If the instance corresponding to the generated address requires stimuli, data is copied from the output register of the stimuli buffer to the stimuli input at t=1. This is determined by the developer. At t=2, is the stimulus ready for the simulator and copied to an internal register. At t=3, all data is ready for simulation, thus the actual simulation is performed at this moment in time. At t=4, the simulation of this delta cycle is done.

The simulator is pipelined. This indicates that stimulus must be buffered for 2 clock cycles in the pipeline.

3.2.3 Output buffer design

The design of the output buffer is similar to the input buffer design, except for the control logic. The simulator is not equipped to control the write cycle, the buffer controller provides this feature therefore. The mechanism is straightforward; it will store the data with the current simulator time in the buffer on data changes.

All data in the buffer should be accessible. Therefore, the entire buffer contents must be readable by the processor, which makes it similar to the producer side of the input buffer.

3.2.4 Interface bridge design

To connect all hardware parts with a GPP, an interconnection block is required. This provides glue logic to translate the data signals, which are used in the hardware, to a unified interface that connects to the GPP.. Also, clock domain translations are performed by the buffers of the interconnection logic. Clock domain transformations for the system's control are performed by the control buffer. This buffer retains the same value for a minimal of one SHILS simulator clock cycle, which implies that this is the maximum frequency at which new commands can be issued by a GPP. In this way it is ensured that the command is received correctly.

The interface bridge connects the external interface to:

- Stimuli buffer
- Output buffer
- Simulator control via clock stretch buffer
- Simulator state

Also, the connector manages the overall system and provides internal connections between certain parts. Therefore, it is referred to as system controller in the implementation.

SHILS incorporates several buffers that are externally addressed using Memory Mapped In-/Output (MMIO). To connect a GPP easily to the internal MMIO interface, the external interface for SHILS is also a MMIO interface. This provides in a direct link between the GPP and the internal buffers. Glue logic provides for integration of the internal MMIO interfaces. It is a common practice to use MMIO in systems consisting of both a processor and dedicated hardware for communication between them. MMIO has proved itself thoroughly in the past in numerous applications.

MMIO interfaces come in several sorts. To provide the simulator with a universal access method for multiple platforms, an additional glue logic block is added that holds the interface for a specific platform.

CHAPTER 4

SHILS Implementation

To gain more feeling with the basic FPGA/ASIC design tool flow and to have a good example of the challenges for the tool developer, an example is used for the implementation of the basic simulator design discussed in chapter 3. An Infinite Impulse Response (IIR) filter is used for this purpose. A filter type often used in digital signal processing. More detail about the filter and its implementation is discussed in appendix B. For a more straightforward implementation of the IIR filter in the simulator, tweaks are used in the system implementation.

For future scaling possibilities, the design is made suitable for a large variety of platforms. This implementation, however, is created for a specific platform, which is discussed next.

4.1 Platform

The simulator is tested on a platform that consists of both hardware *and* software, to provide both flexibility and performance to the developer. The implementation is tailored for a verification platform referred as Basic Concept Verification Platform (BCVP), shown in figure 4.1.

The platform is constructed with a Xilinx Virtex 2 FPGA (3000 series) and two ARM9 processors (an ARM920 and an ARM946 processor). A single processor is used for this first application, as the test simulations are very basic and do not require many resources. The processors and FPGA operate at a frequency of 86 MHz.

The FPGA and processor are connected via the Advanced High-Speed Bus (AHB) bus and External Bus Interface (EBI), providing the processor with a direct memory interface to communicate with the FPGA. This is referred to as MMIO.

Besides the connection to the processor(s), the FPGA also offers a test/debug interface which is connected to LEDs.

For more information on the BCVP platform, see [5].

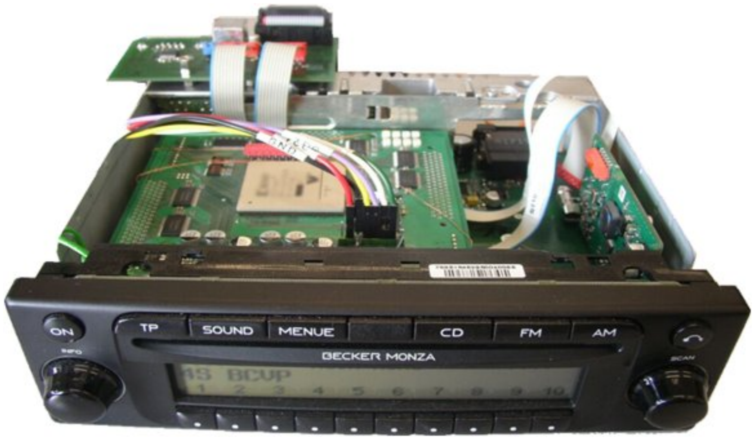


Figure 4.1: BCVP platform

4.2 Hardware implementation

The features of the BCVP platform, which are used by the hardware, are:

- EBI interface
- Test interface
- Clock divider

This leads to a system structure shown in figure 4.2 combined with the simulator design discussed in chapter 3. The simulator functions at a lower frequency than the ARM processor on the BCVP platform. To be more precise, the simulator operates at a frequency 13 times slower than that of the ARM and MMIO interface. The clock division is provided by a clock divider block in the FPGA and used therefore.

As discussed in section 3.2.4, a universal interface is required for easy adapting to a new hardware platform. The BCVP platform specific parts can be replaced easily with a specific interface for any other platform. The topmost level in the implementation is the BCVP specific interfaces. These interfaces are defined in figure 4.3.

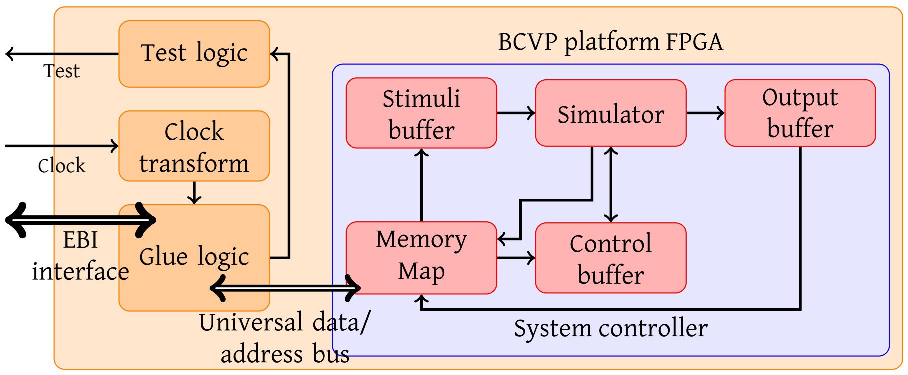


Figure 4.2: FPGA System implementation block diagram

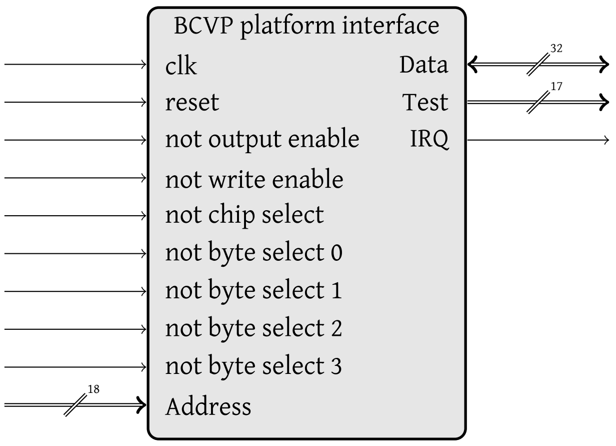


Figure 4.3: BCVP port spec

4.2.1 System Controller

The required universal interface is provided by the system controller, which is connected to the top level entity using the ports specified in figure 4.4.

The EBI interface of the BCVP platform maps easily to this interface by logic that converts the EBI-specific signals to the universal interface. For example, the write enable signal is formed by a logic 'AND' operation of the inverted EBI signals write enable and chip select.

For ease of implementation, the read and write data is divided into two signals instead of a bidirectional bus. It is not possible to simultaneously read and write from the bus, as the data bus of the EBI interface is bidirectional.

This part of SHILS structure is named system controller, as it provides the control of all system parts and interconnects of these parts. Besides interfacing with the hardware platform, the system controller provides interconnection of the other system parts listed on the next page.

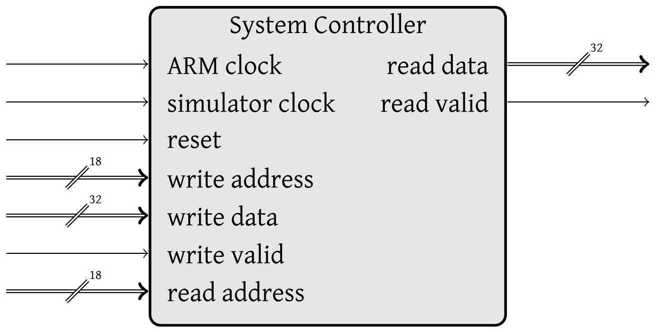


Figure 4.4: System controller port specification

- Simulator
- Stimuli buffer
- Output buffer

The VHDL implementation instantiates these parts from within the system controller as components, which is depicted in figure 4.2.

SHILS is externally is controlled by a GPP over a MMIO interface. The MMIO interface has been chosen as it provides for a direct connection between the GPP and the stimuli and output buffer. The memory interface is used to control memory, which is efficiently and fast in the test platform. This interface is discussed in the next section.

The BCVP platform is equipped with an interrupt connection to the ARM processor. The system controller does not use this interface; this is left as future work.

4.2.2 Memory-Mapped I/O interface

Memory-Mapped I/O provides a robust interface for the processor to connect to the hardware. A dedicated section of the processor's addressing range is assigned to external hardware for this purpose. In the BCVP platform this address range is from `0x30000000` up to `0x301FFFFF`. The sequential hardware-in-the-loop simulator requires only a small number of the available addresses. The addressing space is divided into 16 blocks, of which 4 are used, like shown in figure 4.5. For now, this provides for sufficient addressing space. This division is made by using 4 bits in the upper region of the address vector.

The physical platform used for the tests has a defect in its EBI interface. Bit 13 of the address bus is defect and replaced by shifting bits 18 downto 14 a position to the right. The primary address division is made using bits 16 downto 13 instead of 15 downto 12. By the shift introduced by the physical defect, no awkward transformations should be made in addressing; only a conversion to byte addressing is still required.

For easy programming, read and write of the same data is put on the same address but distinct values are separated. This is shown in appendix A; tables A.1 and A.2.

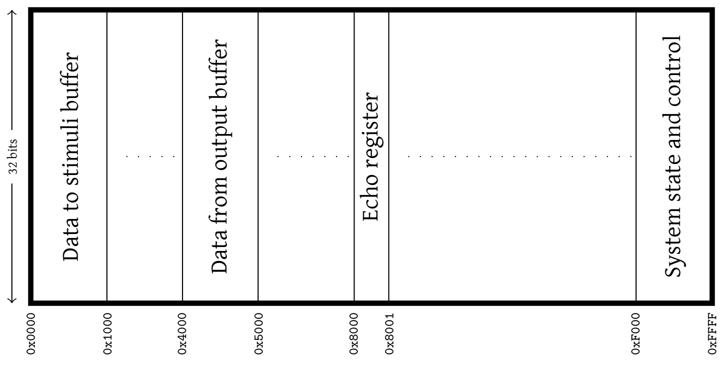


Figure 4.5: Global address space division

4.2.3 Simulator

The simulator connects with the system controller through an interface specified by Rutgers [33], shown in figure 4.6.

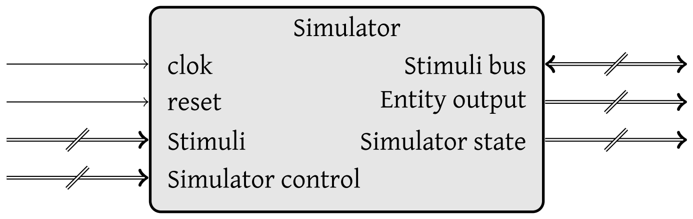


Figure 4.6: Simulator port specification

For more information on port naming and specification, see [33]. Several of the ports are design specific and therefore are not specified in detail. The reset mechanism of the simulator is *active-low*, whereas the rest of the system uses *active-high* reset. Active-high reset is used as this is considered more intuitive.

4.2.4 Stimuli buffer

The FIFO principle is used by stimuli buffer to store stimuli, as noted in section 3.2.2. Circular addressing is used to reduce the amount of required addresses in the buffer. Circular addressing moves the pointer of the buffer to the start address when the end of the buffer has been reached.

Writing new data is performed at the clock frequency of the GPP connected to the FPGA, whereas retrieval of data is at the clock frequency of the simulator. The frequency at which data is produced and consumed also differs. Therefore, the buffer provides for separate read and write ports. Pointers are used to indicate which elements have been written and which elements are read. The buffer controller prevents that the read pointer can pass by the write pointer. To provide the producer (the GPP) with unlimited access to the buffer's memory, the input side of the buffer is fully addressable by the GPP. Several elements can be written to the buffer before the pointer has to be updated. This allows for large quantities of data to be written at once. The GPP must prevent the write pointer from passing the read pointer; this is not done by the controller.

For initial testing of SHILS at the BCVP platform, stimuli generation in the FPGA is not required. Therefore, the implementation does not support selecting the source for stimuli. Also, the interrupt is not implemented, as no interrupt handler is available.

To control the state of the stimuli buffer, two record types are introduced in this implementation, specified in listings 4.1 and 4.2. All elements are data words, 32 bits wide. Using these types, the ports of the stimuli buffer are specified as shown in figure 4.7.

The stimuli buffer instantiates the timecode checker internally. Therefore, it is not depicted in figure 4.2.

```
type fifo_status_t is record
  size  : word;      -- Total capacity
  full  : word;      -- Number of filled positions
  empty : word;      -- Number of available positions
  ptr   : word;      -- Current address position
end record;
```

Listing 4.1: Status type specification

```
type fifo_status_upd_t is record
  ptr   : word;      -- New pointer position
  valid : std_logic; -- Write enable
end record;
```

Listing 4.2: Control type specification

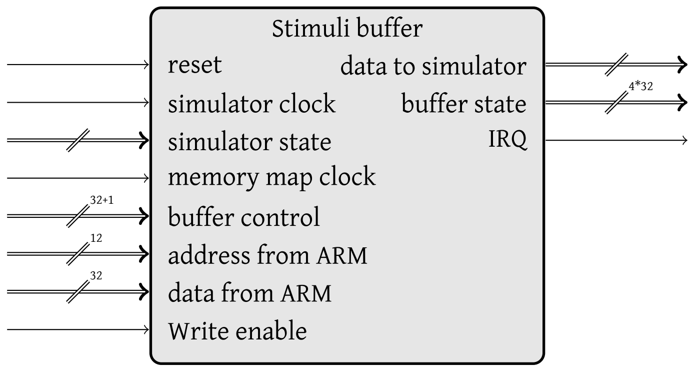


Figure 4.7: Stimuli buffer port specification

4.2.5 Timecode checker

The timecode checker compares the timecode of the topmost sample in the stimuli buffer with the current timecode of the simulator. The timecode is checked using a combinatorial path. The sample is copied to the output of the timecode checker immediately to avoid delay if the timecode matches the current time. The sample is also copied to a register that is used in the following delta cycles if the sample is still valid. This avoids incorrect consumption of data. The ports of the timecode checker are specified as shown in figure 4.8.

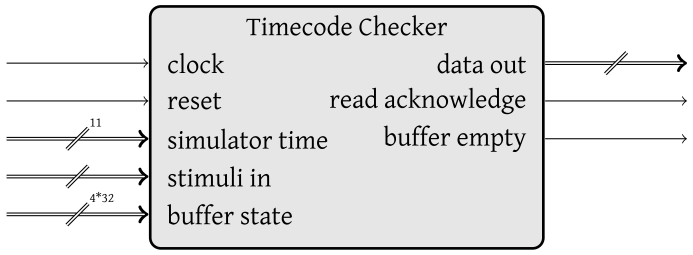


Figure 4.8: Timecode checker port specification

As noted in section 3.2.2, the timecode checker delivers data to the system controller, which determines the precise moment at which the data is offered to the simulator. This moment is determined using two signals in the state port of the simulator, named `prefetch_stimuli` and `prefetch_stimuli_valid`. The first signal identifies the instance that will be simulated after three clock cycles. When the signal is valid (`prefetch_stimuli_valid` becomes high), the stimuli element for the corresponding identifier should be offered to the stimuli port of the simulator.

If A DUT has multiple external inputs, a shift register must be used to offer data at the correct moment to the simulator. The IIR filter only has a single external input. Therefore, the identifier check and shift register are not implemented.

4.2.6 Output buffer

For analysis of the simulation afterwards, the output of the simulated entities is stored in a FIFO buffer also. The implementation of the output buffer is based on the implementation of the input buffer. The controller extracts the current time from the state of the simulator. It is coupled to the output of the current entity and stored in the buffer. For fast consumption, the entire address space of the buffer is accessible by a GPP, but for data validity, only addresses which have been written may be read. The GPP must prevent this by verifying that the read pointer does not pass the write pointer. The controller of the output buffer prevents that data is written before being read. The controller can set the interrupt flag at a certain capacity level, to prevent buffer overflow, but this is not implemented.

The output buffer interface is specified as shown in figure 4.9, using the type definitions of listings 4.1 and 4.2.

4.3 Software implementation

The software is intentionally kept basic, as the test case is very basic. This means that several steps, which could be eased with procedures and functions, are left for future implementation and have to be performed manually for now. An exception is made for the initialisation procedure of the bridge with the FPGA, as this will be used often.

To access the memory portion reserved for the simulator in the FPGA, a structure is used. This is specified in listing C.1.

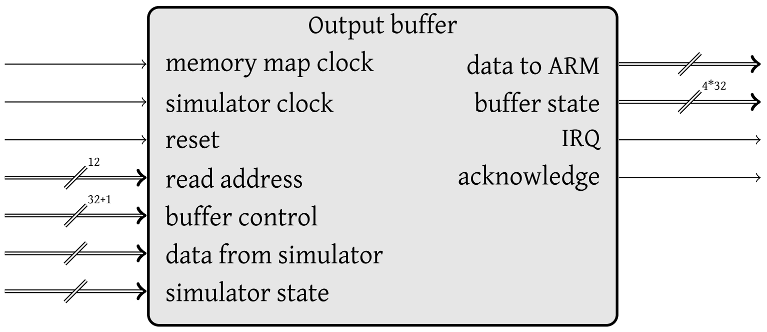


Figure 4.9: Output buffer port specification

The structure is initialized in a precompiler directive, which places the memory map at the correct start address. For readability, several portions are divided in an own structure. For example the simulator state and control are specified as shown in listings C.2 and C.3. For more information on the definition of these variables see [33].

In contrast with the hardware implementation, the variables like `read_from` are not yet assembled of record structures. C is byte-oriented, not bit oriented like VHDL.

To verify the behaviour of the memory map in all regions of the addressing space, the hardware implementation returns debugging data at several positions. The memory position of the test registers is specified in listing C.1.

4.3.1 Configure connection

Before the ARM processor can use the connection with the FPGA, the connection must be configured correctly. The function `bridge_init` configures the interface in 5 steps:

1. Configure (Parallel I/O)PIO clock
2. Enable PIO clock
3. Configure PIO Reset
4. Reset PIO
5. Configure memory controller
    a) Configure Setup time
    b) Configure pulse time
    c) Configure total cycle duration
    d) Configure read/write mode

For a detailed description on these configuration variables, see [24].

4.3.2 Test connection

To verify whether the connection has been configured correctly, the test registers that exist in the memory map listing C.1 are examined and compared with the expected value. This is included in the function `bridge_init()`.

4.3.3 Configure simulation

After initialization of the connection with the FPGA, the simulator is stopped to configure the simulation. The number of entities which are used for this simulation are set by issuing the command `ENABLE` on the address `0x300F0300` and writing a value to memory address `0x300F0304`. After enabling a number of entities, links between entities are created. A function is created for this purpose which is listed in listing C.4. This function issues several commands according to section 2.3.2:

1. Command `CREATE_LINKS` at address `0x300F0300`
2. Instance address at address `0x300F0304`
3. Instance reads from at address `0x300F0308`
4. Instance stimuli mask at address `0x300F030C`
5. Instance writes to at address `0x300F0310`

4.3.4 Run simulation

With all settings created, stimuli should be inserted in the stimuli buffer before performing the actual simulation. Stimuli is copied to the an address in the range `0x30000000` up to `0x30010000`, and the write pointer is updated afterwards by writing the new pointer value to address `0x300F000C`.

Further description of the C code for the IIR filter can be found in appendix B as this is design specific.

CHAPTER 5

Tool evaluation for Hardware/Software co-simulation

The design and implementation discussed in chapters 3 and 4 is actually the design of a co-simulation system. Basic ideas of co-simulation are discussed in section 2.2.

Previous work has shown that stimuli generation significantly increases the load on a processor [39]. A PC is equipped with a greater amount of processing power; the amount of processing power available for other tasks is far greater than in the embedded processor. A PC will be used to replace the embedded processor as shown in figure 5.1. Also enhanced stimuli generation is possible. Using a PC also opens up

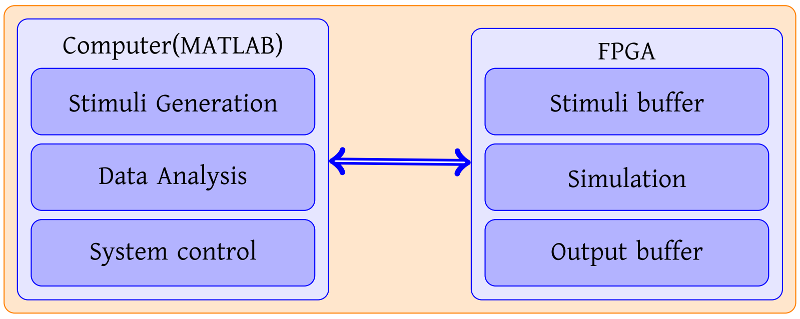


Figure 5.1: Co-simulation system structure

extended capabilities to perform data analysis in various tools. Several tools can be used for the connection of a PC to SHILS, which are compared in this chapter.

MATLAB is sometimes already used in initial design stages for algorithm exploration and other tests. To be able to reuse previously made test benches (in MATLAB), MATLAB is used to generate the stimuli and analyse the output data. Somewhat equivalent to the computing capabilities of MATLAB is GNU Octave [18], but that application is not directly equipped with co-simulation facilities and therefore not discussed in the context of this thesis.

5.1 Requirements

An important aspect is the operating method of SHILS. Implementations so far use a data pushing method to keep the simulator supplied with sufficient stimuli (chapter 3, [39]). This method is used to maximize data throughput. The simulator expects data to be ready for it all the time; therefore the synchronization mechanism is important. To reduce communication overhead, it is desirable to send a large quantity of data at the same time to the simulator input buffer. MATLAB is generally used sequentially, but can function perfectly asynchronous.

The current implementation of the SHILS external interface is a MMIO interface (chapter 3). This enables transmission of data via a unified interface. Minimized effort in migration of the interface in both hardware as software towards the new co-simulation system is an advantage.

The tool discussion assumes that MATLAB is used for stimuli generation, and the tool is used to arrange the connection. However, MATLAB can also arrange the connection. It is discussed as an option for this reason.

5.2 Criteria

The tools to connect SHILS with MATLAB are evaluated according to multiple criteria. These criteria are based on the aforementioned requirements. In the tool comparison is the interface that will be used to connect SHILS not decided.

Damstra has defined key factors which define a good co-simulation system [16]. These key factors are purely focused on software-based simulation. The key factors of Damstra have been used to formulate the tool evaluation criteria, they are modified to apply to hardware connectivity, effort to use and performance. The criteria have influence on each other and partially overlap.

The criteria are evaluated individually. The majority of the criteria are evaluated using an ideal solution that is best for each specific criterion, resulting in ++, +, o, -, -- or --- to indicate how a specific product relates to the ideal. The ideal for each criterion is discussed below. Criterion 3 and 7 cannot be judged in this manner, they are judged upon available features.

1. Hardware connectivity possible This criterion expresses whether the tool can connect with hardware by default. It is desirable that connectivity with hardware can be enabled without great effort. The best case is that connectivity can be automatically arranged/generated.

2. Interaction with multiple simulation tools/platforms (e.g. QuestaSim and hardware) This criterion concerns the possibility of the evaluated tool to interconnect multiple simulation tools or hardware platforms. An example application is the use of QuestaSim for functional and short simulations and SHILS for bit accurate cycle-true long simulations. This feature is an advantage as one could replace QuestaSim with SHILS transparently to the end user. In the best case, the tool arranges the interconnection and replacement without great effort.

3. Supported I/O protocols This criterion concerns the different methods which could be used to connect the tool with hardware. This is closely related with the physical interface used by SHILS, as is must connect with software over this interface too. The evaluation of this criterion only lists the possible protocols, as the comparison of tools is performed independently of the platform used and thus the physical interface used.

4. FPGA integration This criterion concerns the required effort to embed the connection with the tool in the SHILS hardware design. In contrast with criterion 1, this criterion judges whether the SHILS hardware design integrates well with the connection in the FPGA. Integration without a great deal of effort is best, preferably automatically generated.

5. Performance This criterion concerns the performance of the system. Appraisal of this criterion is based on the other criteria.

The performance of the entire system is dependant on the performance of the communication between PC and FPGA and can be measured by data throughput and overall simulation time.

6. Communication overhead This criterion concerns the communication overhead. Some tools introduce significant communication overhead, whereas others handle this efficiently. This criterion is assessed relative to the other tools. Of course, little or no overhead is best.

7. Synchronisation mechanisms This criterion concerns the methods which could be used to provide synchronization between tool and hardware. Synchronization is mainly performed either by time synchronization (tightly coupled) or by using a data-push approach. This criterion lists the options offered by the tools.

8. Initial SHILS deployment effort This criterion concerns the effort that is required to set up the co-simulation system using the tool and SHILS for the first time. Low effort is better.

9. Deployment effort of a new simulation model as DUT for SHILS This criterion deals with the effort that is required to perform simulations using SHILS on a new DUT. Also for this criterion applies that low effort is better.

10. Parameter adjustment effort This criterion deals with the effort that is required to adjust parameters, for example the coefficients of an IIR filter or the topology of a NoC. It is desirable that this can be achieved without synthesis. This is highly implementation dependant, though. Stimuli generation is not considered in this criterion as the interconnection tool does not influence the stimuli generation process.

11. Embedding of co-simulation into the normal design flow This criterion deals with the effort that is required to embed the usage of SHILS simulation in the normal system design flow described in section 2.1.1. Easy embedding is desired, also regarding usage of tools in earlier design phases and reusing testbenches.

12. Cost Although not a primary criterion, this criterion concerns the cost involved in deployment of one of the tools. Both initial purchase cost and recurring costs are considered.

5.3 Selected tools

The tools discussed originate from several disciplines. All tools have possibilities for co-simulation. The discussed tools are:

- The Mathworks: MATLAB & Simulink
- Xilinx: System Generator for DSP
- Altera: DSP Builder
- Chiastek: CosiMate
- UC Berkely: Ptolemy II
- Virtutech: Simics
- Manual solution

The tools provided by Xilinx and Altera are MATLAB toolboxes, which require MATLAB to function properly. Sometimes is constructing a solution manually the best solution. This option is considered too. The manual solution deals with the problems encountered in the discussion of the other tools.

5.4 MATLAB

MATLAB [27], created by The Mathworks, is commonly used in both industry and research for its computing capabilities in several application areas. Its computing capabilities are extended by various toolboxes to connect MATLAB with external tool and hardware, and perform tasks specific for an application area (e.g. Financial Modelling). The collection of toolboxes is quite large. A small selection with self-explanatory names consists of: “Instrument Control Toolbox,” “Image Processing Toolbox,” “Signal Processing Toolbox” and “Control System Toolbox”. Within MATLAB, Simulink offers an environment to apply model-driven design in arithmetic and analysis. Models in Simulink are created in an interactive graphical environment instead of using the scripting language of MATLAB. Simulink comes with a customizable set of block libraries for various application areas. Complicated operations and transformations can be used to graphically create a model, which can be used for calculations.

MATLAB is often used to explore and test the behaviour of systems at high levels of abstraction, where calculations needs to be performed without timing and/or bit accuracy like algorithm testing, signal processing and analysis and much more. At the University of Twente, MATLAB is used for various projects and purposes, in-detail knowledge on MATLAB is therefore available.

The collection of MATLAB toolboxes extends in the field of Digital Signal Processing (DSP) design, and also FPGA design for co-simulation with reconfigurable hardware and hardware oriented simulators. There is a coupling possible with several Electronic

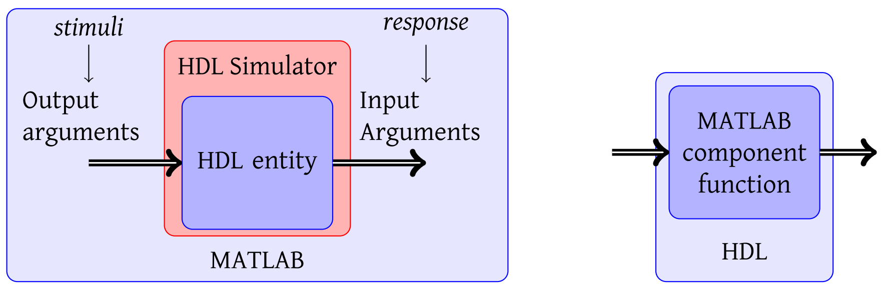
Design Automation (EDA) tools, like QuestaSim (successor of ModelSim) which is discussed in section 5.4.1.

Besides the toolboxes that are sold by The Mathworks, MATLAB is also extended by several FPGA hardware vendors like Xilinx and Altera. These products are discussed in sections 5.5 and 5.6. Both Xilinx and Altera, with respectively System Generator and DSP builder, sell toolboxes which bring FPGA functionality and tools to MATLAB. Both vendor specific tools are primarily targeted at DSP applications, but are also provided with various blocks for HIL simulation.

5.4.1 MATLAB EDA link MQ

EDA link MQ [26], also from The MathWorks, enables a designer to use the simulation capabilities of QuestaSim for hardware designs combined with the computational power of MATLAB for control, signal processing and data analysis. The inputs and outputs of a DUT are connected to structures in the MATLAB workspace, providing the end user with a transparent interface, and a callback routine to control the DUT in QuestaSim from MATLAB.

The features of EDA link MQ enable the creation of test benches (figure 5.2a) for QuestaSim in MATLAB. Furthermore, EDA link MQ enables embedding MATLAB functions in simulations in QuestaSim (figure 5.2b). Wrappers are provided for both options.



(a) MATLAB testbench wrapper (b) QuestaSim wrapper

Figure 5.2: MATLAB QuestaSim toolbox diagram

The toolbox can be used in two methods, a universal method based on TCP/IP and a dedicated one using shared memory. QuestaSim operates as client for MATLAB. The direct memory connection provides the best performance, but the TCP/IP version offers distribution of the simulation across multiple PCs. MATLAB can operate at one PC as server, serving multiple QuestaSim clients. In theory, it should be possible to replace the client (QuestaSim) by a hardware simulator when connecting using TCP. However, there is no specification of the protocol used, which makes engineering an interface for this protocol quite difficult.

A number of disadvantages and pitfalls of this product are discussed by Gestner and Anderson [20]. Creation of test scripts is quite difficult, as users are required to monitor the progress of the simulation by communicating through test vectors, which is not very intuitive. Furthermore, the need to signal QuestaSim every clock cycle introduces a lot of communication overhead. Both of these problems are solved by the open source solution of Gestner and Anderson [20]. As their solution is intended solely for software-only simulation, their connection is using a memory connection, it is not considered further.

5.5 Xilinx System Generator

Xilinx offers several tools to help a developer. One of these is System Generator, which is a MATLAB based DSP design tool, which can *generate* hardware configurations for a design [42]. In this manner, the model-driven design can be used to develop a (sub)system by drag-n-drop methods in Simulink.

A model in Simulink can be constructed out of ordinary DSP building blocks such as adders, multipliers and hardware, but also using more complex blocks or custom hardware [43]. Custom hardware can be added as a black box, so that it is included in synthesis. The Simulink model can communicate using MATLAB connectors with the custom design. The sequential simulator can be viewed as such a black box, a component in the design.

System generator can either generate a HDL file to be included in a HDL design, generate a synthesised netlist (the Simulink model is then the top-level entity) or generate components that apply HIL co-simulation directly. The latter option internally generates HDL code, synthesizes it and provides the FPGA programming file. With System generator, it is possible to realize both the hardware (FPGA/VHDL) and connection to MATLAB at once [17].

The features offered by System Generator can be used to easily create hardware-in-the-loop simulation for various Xilinx-based target platforms. For this task, a GUI is provided (figure 5.3) .

Xilinx System Generator can be used to generate hardware only for Xilinx FPGAs. This is a big disadvantage for flexibility and versatility, but not a very big problem within the CAES chair as primarily Xilinx FPGAs are used. Also, the transformation tool of Rutgers [33] is currently only equipped for Xilinx FPGAs.

5.6 Altera DSP Builder

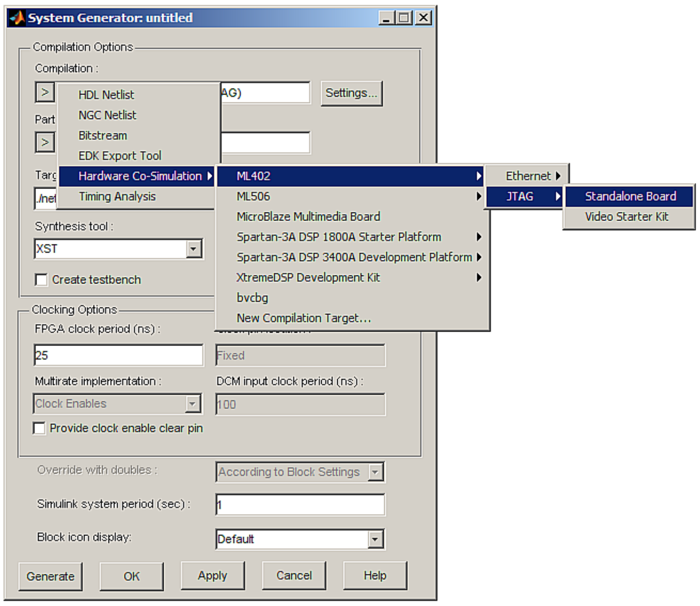
Altera offers DSP capabilities for FPGA developers with the DSP Builder product. Similar to System Generator, a model can be created in Simulink which can be inserted in an FPGA after synthesis, and used for co-simulation. Of course, usage of this tool requires that Altera FPGAs are used. The features offered by Altera's products [1] are similar to those of Xilinx System Generator, no differences have been found in the specifications of both.

5.7 CosiMate

Chiastek offers CosiMate, a product capable of connecting multiple tools together for a co-simulation system using a unified backplane. A GUI is provided to connect the products that are used for a simulation.

By default, CosiMate provides a connection to its backplane for several simulation packages [10], originating from various disciplines:

- The MathWorks: Matlab/Simulink
- Telelogic: Tau G2 (Model-driven software design)


System Generator: untitled

Compilation Options

Compilation :

AG)

Part

Targ

./net

Synthesis tool : XST

Create testbench

Clocking Options

FPGA clock period (ns) : 25

Fixed

Multirate implementation : Clock Enables

DCM input clock period (ns) : 100

Provide clock enable clear pin

Override with doubles : According to Block Settings

Simulink system period (sec) : 1

Block icon display: Default

Figure 5.3: Xilinx System Generator targets

- Telelogic/I-Logix: Statemate Magnum (Graphical embedded system design)
- Synopsys: Saber Designer (Dynamic systems simulator)
- MSC Software: Easy5 (Dynamic systems construction)
- Imagine: AMESim (Mechanics simulation)
- Mentor Graphics: QuestaSim (Digital Hardware design simulation)
- Cadence: NCSim (Digital Hardware design simulation)

CosiMate offers interfaces for several languages:

- C, C++
- Hardware Description Languages: VHDL, Verilog
- SystemC

CosiMate is the master in the system and handles synchronisation issues for the end user, the backplane synchronizes with each simulator individually to allow for each tool to operate at its own frequency. If desired, CosiMate also has the option to operate all tools fully synchronous by tight coupling.

Connections between simulators can be made using TCP, UDP or RCP. Networked simulation is therefore possible, making it possible to open a connection with the sequential simulator over TCP/IP or similar. Backplane connections with other, not supported, simulators can be created using the CosiMate Interface Development Tool. SHILS can be connected either using the Interface Development Tool. Reverse engineering the communication protocol that CosiMate uses is not needed. Still, hardware needs to be developed to connect to the CosiMate interface.

CosiMate should allow for both functional and synchronised co-simulation, but this is not yet possible in current tool releases according to Damstra [16].

Future development is an asynchronous mode to connect sequential and event driven simulators.

5.8 Ptolemy

Ptolemy is a long lasting project of UC Berkely, it can be used without cost for academic and commercial purpose. Ptolemy can be described as "An extremely extensible, heterogeneous simulation and prototyping system with a sophisticated graphical interface" [22]. This description is far from complete, but important to notice is that the objective of Ptolemy II is "to support the construction and interoperability of executable models that are built under a wide variety of models of computation" [4].

Its primary feature is the support for multiple computational models, which can even be used within one system model. This is made possible by the actor-oriented structure of Ptolemy. Examples of actor-oriented languages include hardware description languages, like VHDL and Verilog, and this makes Ptolemy well-suited for embedded system design.

Also, parallel hardware designs can be synthesised directly from Synchronous Dataflow Graph Specifications [37]. Ptolemy can thus generate VHDL code, as FPGA vendor tools are able to generate VHDL code.

For distributed simulation, an architecture is provided that deals with mapping of the simulation processes, and arranges communication and synchronization in a manner that is transparent for the end user constructed on top of the Synchronous DataFlow (SDF) domain called Automated Distributed Simulation (ADS) [15]. This is possible due to the structure of Ptolemy — using the actor formalism. It is implemented using the JINI platform, based on Java just like Ptolemy itself.

To use SHILS for hardware simulation, a client for the distributed simulation will need to be developed according to available specifications. A MATLAB connector is available from commercial vendors that are constructed out of Ptolemy.

For sequential oriented designs, pipelining techniques are offered to benefit from distributed simulation.

5.9 Simics

Virtutech, a spin-off of the Swedish Institute of Computer Science (SICS), has created Simics, a tool that aims at simulation and debugging software operating on multicore and multiprocessor machines [36]. It can simulate networks of processors. Simulations in Simics can be distributed across multiple computers. It is used by the ProtoFlex project in conjunction with an FPGA to increase the overall simulation speed.

Problematic is that the Simics only supports simulation of multiprocessors that are currently on the market, no custom designs are supported. To be able to simulate a new design in Simics requires significant effort as a new processor model must be created for each new hardware design. Also, these extensions can be implemented only by Virtutech, not directly by an end user.

Another disadvantage of Simics is that there is no previous experience with usage of Simics at the CAES chair.

5.10 Manual solution

Each of the tools discussed above has disadvantages. The desired result of the manual solution is a tool that perfectly matches the requirements of SHILS, stimuli generator/analyser and end user. This solution matches all the criteria of section 5.2 perfectly from the performance point of view. However, the manual solution is the worst solution for effort and resources.

The major concern of the discussion of the tools is the communication overhead. There are communication media available for even faster communication between SHILS and MATLAB, like PCI-X, PCI-E [40] and a direct Front Side Bus (FSB) connection [7]. In this manner, a connection capable of transmitting several Gbps is offered. In the FPGA an interface to the SHILS design has to be created, similar to the already existing interface (chapter 3). Besides an interface for the FPGA, MATLAB connectors need to be created. In order to replace QuestaSim transparently with SHILS in MATLAB, additional effort is required.

5.11 Comparison

A decision should be made in favour of one of the discussed solutions based upon the criteria listed in table 5.1. Per tool, a score is added to the criteria. Criterion 3 and 7 are not evaluated according to scores but according to features.

Statements on Xilinx System Generator also apply to Altera DSP Builder and are referred to as FPGA vendor tools in this section.

Hardware connectivity is possible for almost every tool. The FPGA vendor tools connect directly to FPGAs, whereas CosiMate, Simics and Ptolemy provide for a TCP/IP interface for distributed simulation in the software. The FPGA connector must be developed for all tools, except for the FPGA vendor tools. TCP/IP hardware blocks are available, but co-simulation logic still must be created manually. The FPGA vendor tools generate the specialized co-simulation blocks too.

Most tools only support connectivity with a single other tool, except CosiMate which has been designed specifically to interconnect multiple tools, and Ptolemy with its computational domains. Transparent replacement of QuestaSim by SHILS without changes is only possible with CosiMate, as only CosiMate configuration is changed.

Each tool supports communication over the TCP/IP protocol. Most implementations work local at a single machine, but the co-simulation can be distributed across multiple machines. MATLAB is equipped with a large number of connection protocols including USB and serial connection that are sold separately. Therefore, MATLAB is very versatile regarding connections. The FPGA vendor tools can access these connection media too when available, but require changes to the connectors in both the Simulink model and FPGA design.

The FPGA vendor tools generate hardware, and provide several options to integrate this hardware in a design. For this reason the integration with the FPGA design is best for these tools. Code generation of designed cores is possible for Ptolemy, CosiMate and MATLAB, which indicates that integration is possible, but less versatile than the vendor tools.

Literature states performance problems with MATLAB due to significant communication overhead [20]. Communication overhead is caused by a lot of effort required to keep the entire system synchronized. Future versions of CosiMate are said to be capable of performing functional (asynchronous) simulation, a solution with less communication could be developed for the FPGA vendor tools. Ptolemy is equipped with ADS that arranges distribution across multiple clients [15]. This improves performance if computation requires more computational power than communicating by pipelined parallel execution. Simulation in Simics is performed functionally, therefore the performance is acceptable but not suitable for cycle-accurate verification.

Data-pushing synchronisation can be applied by System Generator or CosiMate natively. Ptolemy offers an automated mechanism to distribute actors in simulation(ADS). The other tools use time synchronisation which can involve a lot of communication overhead. Simics allows the simulation hosts to be out of sync slightly, to reduce this overhead. Therefore, tools offering data-pushing synchronisation have an advantage over the rest.

The required effort to create an initial system is large for CosiMate, Ptolemy and Simics, due to the lack of previous experience within the CAES chair with these tools in general and especially for use in co-simulation. System creation is graphical in CosiMate, which indicates that the effort required to create the system is not Experience with MATLAB is available, initial deployment effort is therefore less for MATLAB and the FPGA vendor tools, which use Simulink for modelling. As the FPGA hardware can be generated by the FPGA vendor tools, total effort is lower for initial deployment than for MATLAB.

The easiest approach to deploy a new DUT for simulation is to connect the new model to already existing infrastructure. Therefore, CosiMate has an advantage over the rest. As insertion of a block is pretty straightforward in Simulink, this is considered easy too. In the FPGA, no changes should be required when the implementation has grown to its full potential.

The possibility to easily implement small changes to a design during simulation is highly appreciated. The manner in which this is dealt with is highly implementation dependant. For example, at run-time, SHILS can change the connections between entities and the number of simulated entities. CosiMate only operates at the interconnection level. It therefore does not *directly* provide facilities for this. The tools, which only generate code for FPGAs (CosiMate, Ptolemy and MATLAB), are not suited well, as a rerun of the synthesis process is not desired. This is of course heavily dependant on the specific implementation which is made.

MATLAB is already used within the normal design flow. Therefore, usage of MATLAB requires the least changes to the design flow. Tools which do not operate within MATLAB (e.g. CosiMate), require minor changes to the design flow.

Most of the discussed tools require a license that must be purchased, except Ptolemy, which is available free of charge. Fortunately, academic research licenses are available for several of the other tools. A research licence for MATLAB must be purchased. However, the communication interfaces are provided by a toolbox which is not available at the CAES chair due to licensing specifications[1]. Using the Instrument Control Toolbox require additional funds.

System requirements are comparable for each tool; a faster CPU will result in faster generation of stimuli etc.

[1]The Research license of MATLAB does not include these toolboxes by default, whereas the educational licence does.

Table 5.1: Tool comparison matrix

	MATLAB	System Generator & DSP Builder	CosiMate	Ptolemy	Simics	Manual
1. Hardware connectivity	+	++	o[1]	o[1]	o[1]	+
2. Connect multiple tools	+	+	++	++	--	o
3. I/O protocols	TCP/IP, Serial, USB [2]	TCP/IP, USB , MATLAB IO[3]	TCP/IP	TCP/IP	TCP/IP	Any[4]
4. FPGA integration	+	++	+	+	--	---
5. Performance	-	+	-	+	-	++
6. Communication overhead	--	-	+/-	+	o	++
7. Synchronisation mechanism	time[5]	time	time/data	data	time/data	time/data
8. Initial deployment effort	-	++	+	--	--	---
9. New model insertion	+	+	++	+	+	++
10. Parameter adjustment	+	++	-	+[6]	+	++
11. Embedding in design flow	++	++	+	-	-	+
12. Cost	-	+[7]	+[7]	++	+[7]	++

[1] Not by default

[2] Matlab offers connectivity with various interfaces. For example TCP/IP, USB and Serial, but numerous interfaces are available.

[3] TCP/IP and USB are supported natively. System Generator can use the I/O protocols of MATLAB.

[4] It is possible to implement any desired interface, but it requires development.

[5] There is an open source extension which uses data pushing.

[6] When only used as interface construction tool.

[7] Academic licenses available.

5.12 Conclusion

Based on all arguments in the previous section (summarized in table 5.1), the best solution is to implement the SHILS interface using Xilinx System Generator, and slightly alter the MMIO interface in the FPGA accordingly. It offers all desired functionality at a reasonable effort and performance. When more performance is desired than Xilinx System Generator can offer, the next step is to create a custom system, which requires a far greater effort to implement. When the custom solution is chosen, CosiMate could provide easily for an interface for the interconnection medium.

Ptolemy scores quite well in table 5.1. It is not selected as best as it is not equipped with FPGA connectivity features out-of-the-box. Furthermore, little experience of Ptolemy is available at the CAES chair. Significant effort is thus required to gain knowledge on how to use Ptolemy for co-simulation.

CHAPTER 6

SHILS co-simulation system design

SHILS is intended as an aid in the verification(by simulation) process of large SoC/NoC designs. Verification is performed by doing extensive simulations of a DUT. Previously, a relatively small embedded processor was used to generate stimuli for these simulations [38, 39]. Besides the computational load of stimuli generation on the processor, also the effort to deploy the test bench is considerable.

Software is already used in NoC design for several research tasks, such as analysis and simulation of communication and data processing applications. MATLAB/Simulink is often used for design exploration, but can also be used to perform behavioural analysis of the implementation of such systems combined with QuestaSim and therefore to create advanced test benches. These test benches could be reused in the verification process, but when verification is performed on a different platform, significant effort is still required to port the test benches towards the other platform. For flexibility in the stimuli generation procedure and re-usage of previously developed components, it is desirable that stimuli are generated by software, not by hardware. By replacing the small processor with a standard PC, the performance of stimuli generation can be improved with enhanced features. Software like MATLAB offers advanced data analysis tooling for the analysis of simulation results. By connecting SHILS to a PC, a “new” co-simulation system is created.

The system tasks are distributed across the PC and FPGA to fully benefit from the capabilities of both FPGAs and PCs like shown in figure 6.1. Chapter 5 focuses on the connection of PC and FPGA, whereas this chapter focuses on the entire system using that connection.

The PC runs MATLAB for stimuli generation and analysis afterwards, and the FPGA is charged with performing the simulation to accelerate the verification process. The link between PC and FPGA can be tightly coupled, which can lead to a the PC waiting on the FPGA or the opposite. This will lead to less performance than all parts operating at full speed. To enable the simulator and MATLAB to operate at different operating speeds, buffering is required in the FPGA for stimuli elements and output data, also used in the first design, see chapter 3.

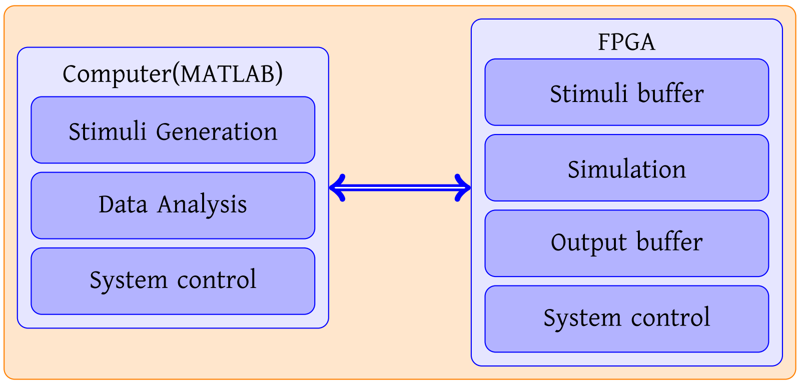


Figure 6.1: Global System structure

The connection between processor and FPGA in earlier implementations of SHILS has been a DMA connection. For connectivity between PC and FPGA, several other media are possible (discussed in section 6.3.3). Using these media, the connection can be made using several tools (discussed in chapter 5), or manually.

The rest of this chapter discusses the high-level communication first. Afterwards, low-level communication using Xilinx System Generator is discussed, which came up best out of the tool comparison.

6.1 Requirements

Criteria for co-simulation tools are defined in section 5.2, which are also applicable for the created co-simulation system. The design should profit from general knowledge of the target platform, based upon platforms available in the CAES chair. Also, the system has to deal with requirements described in chapter 3 on reuse of previously developed parts.

Another important requirement is that the new design is more efficient than previous implementations, especially in the field of communication overhead as this is quite often a major bottleneck. The precise elements that are communicated and the number of transmitted elements should be considered with great care for this reason.

Earlier implementations show that, with significant effort in a processor (about 25% of the time purely to generate new elements), it is possible to keep up with the simulator [38, 39]. The link between PC and FPGA must be loose without causing problems with buffers or the rest of the system's integrity. Stopping the simulator to wait for data to be transmitted is not a good option for maximal performance.

6.2 Global system structure

The global system structure is depicted in figure 6.1. This shows the division of functionality between the PC and FPGA. The global structure of this design is similar to the design discussed in chapter 3.

Within the system, the PC is the master, the FPGA slave. To loosen up the link between PC and FPGA, buffering is used for stimuli and output data. Previously, a MMIO interface was used to control this. The MMIO interface was also used for system control. Usage of a MMIO interface is a bit different than the data transfer method used by MATLAB.

6.3 Data transfer method

Usage of the MMIO interface in the new system is evaluated for the data intensive stimuli and output flows. The linear buffer (FIFO), implemented using circular addressing is not discussed at this point. Only the methods to transfer data to and from the buffer are treated.

Within the system, options are available for placing data in the FIFO buffers, these are discussed below.

6.3.1 Reuse MMIO interface in FPGA

The option that is easy to deploy in the FPGA is to keep the existing interface to the FIFO buffers. MATLAB will then need to generate addresses where to store the elements . This requires that MATLAB maintains an up-to-date administration of the state of the buffer, involving a significant quantity of communication.

6.3.2 Add address generation to FPGA

Stimuli are generated as a sequence of elements most of the time. Therefore, no advanced address generation mechanism is required, and this could easily be implemented in hardware. Generally, it is better to execute such tasks in hardware to gain better performance. Also, moving the task of address generation from MATLAB, reduces the communication overhead. The FPGA design is extended with an address generation mechanism to reduce communication overhead, increasing overall performance and reducing interface complexity in MATLAB.

Synchronized operation is required, even with loose coupling of PC and FPGA. When buffers are full, no more elements may be transmitted to avoid buffer overflow problems, and empty places in the buffer must be filled as soon as possible to keep the simulation from stalling. Stalling the simulator, of course, leads to a loss of performance, but provides a safety net when the unlikely event might occur that a buffer is empty. The FPGA design should be extended with logic that temporarily halts the simulator when stimuli buffers are empty.

The moment until which the simulation in the FPGA can run undisturbed is easily determined at the moment that time codes are generated. The PC can use this period of time to generate new stimuli. Knowledge about the time it takes to generate elements must be available. The FPGA control can be extended by a “Run for n cycles” command using this principle. A parameter of the command is “n”, the number of cycles that the simulator will run. This is a more sophisticated method of pausing the simulator after a predetermined number of cycles, instead of using the emergency break when a buffer is empty. Some administration is used in the PC that sets the “Run for n cycles” counter when new data has arrived in the FPGA. With these measures taken, the control data is significantly reduced. Instead of sending a command each system cycle, the number of transmitted commands is divided by n.

Performance degrades slightly, but simulating with incorrect data leads to a simulation run that is completely useless.

The amount of data, which arrives in the FPGA differs with the medium used for connection. A single stimulus sample could arrive in one frame or several thousands of elements in one frame. The communication interface buffers these elements shortly as it is not possible to copy multiple elements instantaneously to the stimuli buffers, and of course, the buffers are large enough to accommodate this number of elements in the buffer. Large buffers allow a more loosely coupling of PC and FPGA, simply due to the larger time both parts can operate independently. Also, communication overhead is naturally lowered when transmitting more data in the same package, according to Chan et al. [8]. This is not possible for all interface media, like serial communication. Considering these aspects, a good balance has to be found between buffer size, packet size and stimuli generation frequency.

The amount of data generated for stimuli is far greater than the amount of control data. Although the MMIO interface is well-proven, there is a lot of additional data communicated for addressing. Reducing the number of addressing bits reduces this overhead. Instead of using 26 bits for addressing(BCVP platform), this could be reduced to only a few (2-4) bits to select between stimuli and control, which is easier to implement in MATLAB and requires less resources in the FPGA.

However, when implementing the new SHILS using a DMA interface, argumentation above is invalid, as the connection is already equipped with the MMIO interface. Thus, this is largely dependant on the used interconnection medium.

MATLAB is not equipped best for generating data/address bus like structures. The mapping to physical addresses needs to be made in the background, by programming MATLAB to write to certain predetermined addresses.

6.3.3 Interconnection medium

A number of features are available for some connection media only. These need to be discussed. Based upon available platforms at the CAES chair, possible interfaces are:

- Serial
- USB
- TCP/IP (Ethernet)
- Direct memory, i.e. PCI-X

Several other interfaces exist, possibly offering better throughput/performance. PCI-Express is an example. Even a direct Front Side Bus link is available [7]. Focus of this thesis is on *available* platforms, therefore, these kind of interfaces are not considered.

To implement the connection between PC and FPGA, several tools are available, which are discussed in chapter 5. As noted in chapter 3, the implementation is equipped with a MMIO interface for connection to the BCVP platform. This may be changed upon requirements of the connecting interface.

Serial interface

The serial interface is an often applied interface for connection of embedded systems, due to its simplicity and ease of implementation. However, SHILS will generate a lot of data at high frequencies. For most serial interface implementations, this is not practical. Therefore is it not considered as an option for data communication. For high-priority, low latency, communication it can be a good option, i.e. IRQ signals.

Serial interfaces operate at low layers in the Open Systems Interconnection reference model (OSI) model, for example the RS232 defines only the physical layer (layer 1).

USB interface

For several applications, USB has superseded the serial interface, primarily due to the higher bandwidth. In contrast with basic serial interfaces (i.e. RS232), several protocol layers are offered on top of the physical layer. There are four types of transfer defined [14]:

- Control Transfers
- Interrupt Transfers
- Asynchronous Transfers
- Bulk Transfers

These types allow a better match to the requirements of the client software. Interesting for SHILS are Interrupt transfers, which offer bounded-latency communication at low frequencies and Bulk transfers for transferring a large quantities of data at once.

TCP/IP over Ethernet interface

Several low level implementations of the TCP/IP protocol are available dealing with levels 1 to 4 of the OSI model. IP is the Network layer. TCP is the Transport layer, both operate on top of several physical media. Ethernet is most often used for this purpose. The upper levels are left to the developer, and application-specific implementations. The required implementation effort is lower when standard connectors are available for higher level communication.

The best solution for maximal throughput is Ethernet according to Xilinx System Generator. System Generator can both operate in a normal TCP/IP network as well as a point-to-point connection over Ethernet. This point-to-point connection only implements the Physical and Data link layer for maximal throughput.

Direct memory

Theoretically, Direct Memory Access (DMA) for the processor offers maximum performance, which is used in the first implementation of SHILS, see chapter 3. Another direct connection is using a PCI interface, or its successors. Available for usage are the EBI connection for the BCVP platform. A different platform provides a PCI-X interface which offers a maximum throughput of 4.3 GB/s.

A hazard when using DMA is that a major part needs to be developed especially for a specific target platform with its specific interconnection, which requires significant effort.

6.4 Xilinx-based design

This section discusses a co-simulation system design based on Xilinx System Generator. System Generator can generate all required elements to use an FPGA in co-simulation over Ethernet. The created system is depicted graphically in figure 6.2.

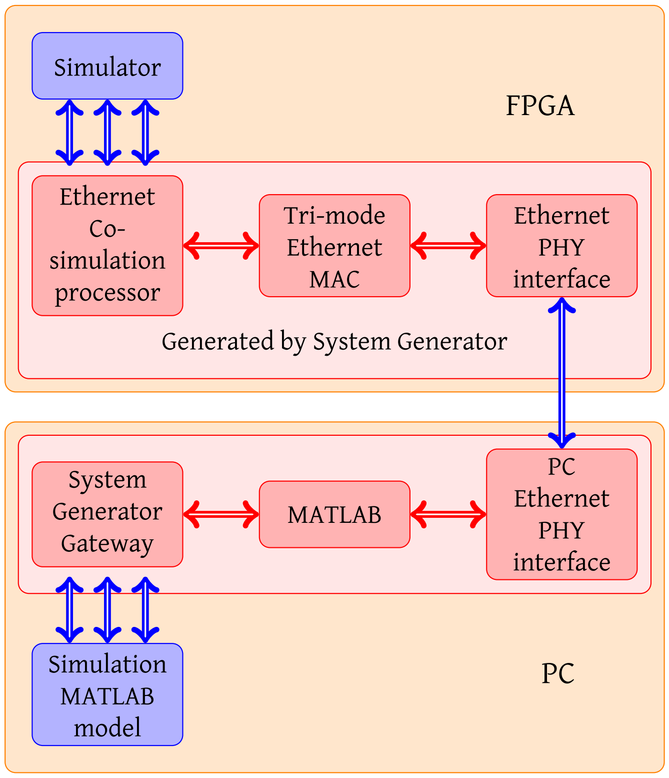


Figure 6.2: System design using Xilinx System Generator

Inside the FPGA are several cores generated by System Generator. The Ethernet co-simulation processor takes care of the communication process with the PC. For the Ethernet communication, an Ethernet MAC core is generated and a PHY interface. The co-simulation processor offers a number of ports to the simulator in the FPGA. These ports are defined in MATLAB using “Gateway” blocks which convert MATLAB data types to System Generator fixed point data types (when applicable) and takes care of the instantiation of the corresponding ports in the FPGA design.

The ports in the FPGA connects to registers for control and state or FIFO buffer ports for the stimuli and output data. The output gateways of MATLAB write to the input registers and stimuli buffers, the input gateways read from state registers and output buffers. Of course, the amount of data that is transported to the FIFO buffers is much larger than the amount of data for control and state.

The Ethernet connection delivers large packets of data at one moment for low communication overhead. Therefore, the buffer capacity has to be large enough to accommodate for the number of elements that are delivered. There are IO buffers provided by System Generator, allowing delay to clock in the data.

6.4.1 Performance

According to Xilinx, connecting System Generator over point-to-point Ethernet offers the best performance and throughput, as it can operate at speeds up to 1Gbps and use jumbo Ethernet frames (up to 8MB per frame) [8]. The communication overhead, as the system of course requires management of the co-simulation processor, is minimized when transmitting large quantities of data at a time. In creating large frames of data, large buffers are required. By using System Generator to arrange the communication, throughput is not considered a bottleneck, which is also substantiated with figures on throughput [25].

In the unexpected event that communication throughput becomes a problem, the system could be redefined to use either PCI-X, which offers a maximal bandwidth of more than 4 GB/s, or create a system directly coupled to the FSB of a processor like [7]. As consequence, Xilinx System Generator cannot be used to arrange the communication for those platforms; it can be used to regulate the gateway structures.

If time it takes to generate new stimuli and perform analysis on the results becomes a problem, can this be solved by using computationally cheaper algorithms, or by distributing stimuli generation across multiple PCs.

6.4.2 Deployment effort

If system parts could be generated, it requires less effort to implement. This has been one of the primary reasons to choose Xilinx System Generator. Deployment in the FPGA will be fairly easy, although caution should be taken to correctly design the control and state interfaces which are structures that require multiple 32-bit words. These interfaces are highly design specific.

6.4.3 Resources

The design has been based on hardware platforms that are already available at the CAES chair. Therefore, the major cost is MATLAB, including the connectivity toolboxes and Xilinx System Generator. To examine the features of System Generator, a limited demo version has been used. Academic licensing is available for System Generator. The toolboxes for MATLAB need to be purchased for research purposes.

System Generator runs fine in MATLAB on a up to date PC (Intel Core 2 duo processor at 3 GHz, 2GB RAM).

6.5 Conclusion

This chapter discussed the redesign of SHILS using Xilinx System Generator. It is possible to design a loosely coupled system over Ethernet, with enough bandwidth available for large amounts of data. In case that the interface does not offer enough throughput, alternatives are proposed which will require larger deployment effort, as System Generator could not be used for that purpose. An available platform has a PCI-X interface. This interface is generally not supported in a normal PC. To gain more throughput, significantly more resources are required to purchase dedicated hardware.

CHAPTER 7

Results

To provide more insight in scalability of SHILS, the resource requirements are examined. These results are obtained from synthesis using Xilinx Integrated Synthesis Environment (ISE).

The implementation of SHILS uses stimuli buffer that is 2048 elements deep and 32 bits wide. The output buffer is implemented identical. The IIR filter entity is used as DUT.

SHILS is synthesized for a Xilinx Virtex-II series FPGA, type 3000, this FPGA has resources shown in table 7.1 [41].

To be able to compare the number of required resources across multiple technologies, 4 figures are used:

- the number of 4-input Look-Up Tables (LUTs)
- the number of synchronous memory (distributed RAM)
- the number of Flip flops
- the number of Block RAMs

Table 7.2 shows that 2.3% of the available 4-input LUTs are used, 3.5% of the number of available Flip Flops and 15.6% of the available Block RAMs are in use in the Virtex-II FPGA. A distributed RAM 2 LUTs in a Virtex-II FPGA [41]. The number of LUTs used for distributed RAM is not added to the number of 4 input LUTs in table 7.2.

Table 7.1: Xilinx Virtex-II available resources

	4 input LUTs	Flip Flops	Block RAMs
Virtex-II 3000ff1152	28672	28672	96

Table 7.2: SHILS LUT resource consumption

Component	4 input LUTs	Percentage LUTs	Distributed RAMs	Flip Flops	Block RAMs
Simulator + DUT	177	26.8 %	12	92	7
Stimuli buffer	86	13.0 %	0	106	4
Output buffer	53	8.2 %	0	89	4
Control buffers	132	20.0 %	0	323	0
System controller	182	27.5 %	0	291	0
SHILS	630	95.3 %	12	901	15
Platform Logic	31	4.7 %	0	89	0
Total in FPGA	661	100 %	12	990	15
% of FPGA capacity	2.3 %			3.5 %	15.6 %

Table 7.3: SHILS Simulator resource consumption

Component	4 input LUTs	Percentage LUTs	Distributed RAMs	Flip Flops	Block RAMs
Design Under Test	65	36.7 %	0	0	0
Entity wrapper[1]	102	57.6 %	0	34	3
Simulator	75	42.4 %	12	58	4
Simulator total	177	100 %	12	92	7

[1] Including DUT

7.1 Simulator Scalability

For better insight in the overhead, the required resources for the simulator are split out in table 7.3. The percentages for the DUT give a distorted view as the currently implemented DUT is very small. A resource estimation for a larger DUT is listed by Rutgers in [33, chapter 6].

7.2 Scalability of stimuli buffer

Other DUTs probably have a different configuration of inputs and outputs. An example is a NoC with 256 routers and thus 256 inputs which should be buffered 32 elements deep. The inputs do not need to be provided with stimuli in parallel. This example is used to compare with the current SHILS implementation that has a single input which is buffered 2048 elements deep. The resource consumption for the individual parts of the stimuli buffer are shown in table 7.4. Scalability is important to increase the number of stimuli buffers, and decrease its depth.

Table 7.4: Stimuli buffer specified resource consumption

Component	4 input LUTs	Percentage LUTs	Distributed RAMs	Flip Flops	Block RAMs
Controller	61	70.9 %	0	89	0
Buffer	0	0	0	0	4
Timecode checker	25	29.1 %	0	17	0
Total	86	100 %	0	106	4

7.2.1 Storage scaling

The current implementation of both the stimuli buffer and the output buffer store 2048 samples of 32 bits. These require 4 block RAMs each, as one block RAM can store up to 512 elements of 36 bits wide. The alternative example stores 32 elements of 32 bits wide for 256 buffers, a total of 8192 elements, which can be stored easily in 16 block RAMs. The resources required to store the larger number of elements scales linearly.

The data can be either stored in one large memory, or in 256 smaller memories. Control of a large memory is important. The best solution is to divide the addressing space into blocks for each input to avoid searching for stimuli for input X. Due to the larger address width of a large memory, more LUTs are required and other interconnection logic for implementation are required, especially when the memory is divided into blocks, although data density in the block RAMS is higher when using a large memory.

7.2.2 Control scaling

When the entire stimuli buffer is instantiated multiple times for multiple inputs, the number of resources required scales linearly. For selection of the correct buffer, additional logic is necessary of course.

A controller for multiple buffers could be created, but would increase the complexity and would thus increase the combinatorial path. Therefore, it is best to use a separate controller for each stimuli input, of which the data is either stored in a single large memory or individual memories. As noted above, a division of addressing space in blocks eases access to the contents in an FIFO oriented fashion. Using multiple controllers is possible, as a controller requires little resources(0.3% resource consumtion in Virtex-II 3000 FPGA).

7.3 Control buffer scalability

The control buffers are implemented as 32 bits wide buffers. Therefore, they consume the same amount of logic when used in larger systems (with larger bit widths). Also, the amount of resources required for all control buffers is marginal compared to the available resources.

CHAPTER 8

Conclusion

The research for this final project has focused on two main issues as stated in the introduction:

> How to implement the simulator in an actual FPGA, how to connect it to a processor and how to make them communicate?

and

> How to connect a PC with the sequential simulator, how to make them communicate and process?

For the implementation of the simulator in an FPGA, several components are designed to provide the simulator with a sturdy external interface and buffers for clock domain conversion. The external interface provides the simulator with a connection to a processor, so that the processor and simulator can communicate using Direct Memory Access over an MMIO interface. This interface theoretically offers maximum throughput as the processor can access the FPGA directly. The design is made suitable for a large variety of platforms. The implementation is created for the BCVP platform. Resources for the stimuli buffer scale proportionally, the resources consumed by the control buffers are constant. The stimuli buffers should be scaled with care, implementation using fully parallel buffers is easiest.

The initial design and implementation for the BCVP platform were created with limited features to provide a solid base for further research. Creating this implementation consumed much more time than was anticipated. Therefore, the focus remained on the main topic of this final project — communicating with the simulator.

To provide for a more versatile test platform, the ARM processor is replaced by a PC. The PC can take care of the set-up of link memories in the simulator before simulation. This is done manually up till now.

The combination of an FPGA and a PC create a new co-simulation system. The interconnection is often a major performance bottleneck in co-simulation systems. To create the interconnection between PC and FPGA without much effort, several tools can be used. Xilinx System Generator is evaluated as the best tool for SHILS, in particular as it generates facilities the for synchronization and communication.

The required effort to deploy SHILS using System Generator is lower than for other options, and communication overhead is not significant. Ethernet is used as communication medium over which System Generator can transmit raw Ethernet frames in a point-to-point connection to obtain the best possible throughput. Transmission speeds of 1Gbps are theoretically possible according to Xilinx.

8.1 Recommendations and Future work

SHILS is work in progress, several issues arose in the design and implementation phases. Besides optimizations for SHILS and its tooling, also new ideas are mentioned.

8.1.1 Improve conversion to simulator

When a design is made, the designer will need to manually define the entity which will be simulated sequentially. These manual changes imply a serious risk of introducing additional errors in the process of linking the entities together. This is undesired in the test and verification of a design.

8.1.2 Better simulation control in FPGA

Control of the simulation in the FPGA is easy, but should be done with care. Registers keep the same value when set once. It often is easier to allow a fixed number of system cycles before issuing another command. Tight coupling of FPGA and processor is not desired, the processor cannot monitor exactly at what moment the simulation is done. Therefore, the simulator control should be extended such that the command register in the FPGA is reset each time a command is accepted. This ensures that the simulator stops after a full system cycle, instead of running to completion.

8.1.3 Data compression of stimuli and output data

For now, all data is transmitted and stored without any form of compression. With limited bandwidths and storage capacity, using compression poses an opportunity to enhance throughput and storage capability. As compression and decompression requires resources and time, this requires more thorough examination.

8.1.4 Replace QuestaSim with SHILS

The new design can be constructed in such a method that one can choose whether to simulate using QuestaSim, or using SHILS with the same set of test data.

Behavioural simulations can be done easily in QuestaSim, and SHILS can be used for cycle-true bit-accurate simulations that in QuestaSim last prohibitively long.

APPENDIX A

Memory Map

Figure A.1 shows the address space which is available for SHILS as designed in the FPGA. The addresses which are mentioned indicate the range which is reserved for a certain block. The figure shows the word address. In the connection to the processor is the most significant byte one position shifted left. This is caused by physical defects on the test platform (section 4.1) The processor uses byte addressing. For these reasons, an address translation table is created, depicted in tables A.1 and A.2.

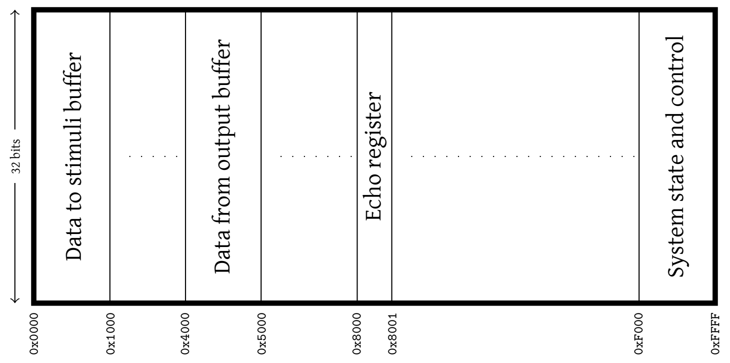


Figure A.1: Global address space division

The system state and control address space is expanded in figure A.2. The address space for simulator control and state is in the range of `0xF0C0` up to `0xF0FF`, which is expanded in figure A.3. Figure A.3 uses naming in accordance with Rutgers [33]. All these figures show the address space as designed in the FPGA, which is a sort of word addressing.

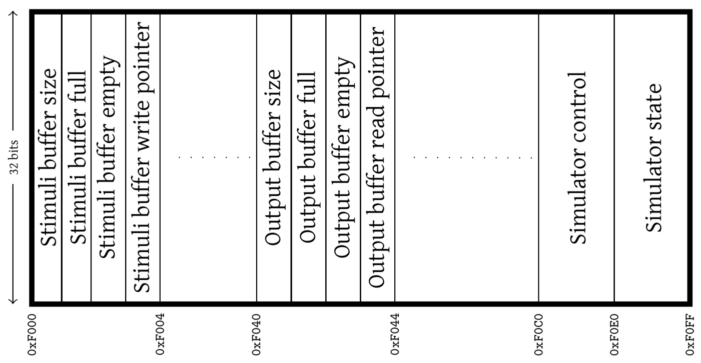


Figure A.2: System control address space

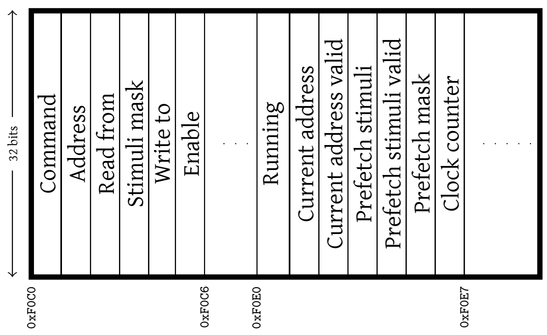


Figure A.3: Simulator state and control address space

Table A.1: Write memory map

GPP	FPGA addressing			Description
Byte address	bits 16-13	bits 7-4	bits 3-0	
`0x00000`	`0x0`	X[1]	X[1]	Write data to FIFO
`0x80000`	`0x8`	X	X	Write data to echo register
				Buffer control
`0xF000C`	`0xF`	`0x0`	`0x3`	Update FIFO input write pointer
`0xF000C`	`0xF`	`0x4`	`0x3`	Update FIFO output read pointer
			bits 4-0[2]	Simulator control
`0xF0300`	`0xF`	`0xC`	`0x0`	Command
`0xF0304`	`0xF`	`0xC`	`0x1`	Address
`0xF0308`	`0xF`	`0xC`	`0x2`	Read from
`0xF030C`	`0xF`	`0xC`	`0x3`	Stimuli mask
`0xF0310`	`0xF`	`0xC`	`0x4`	Write to
`0xF0314`	`0xF`	`0xC`	`0x5`	Enable

[1] Actually, these bits determine the address at which the data is stored in the FIFO buffer.
[2] To reserve a larger addressing space for simulator control, 5 bits are used.

Table A.2: Read memory map

GPP	FPGA addressing			Description
Byte address	bits 16-13	bits 7-4	bits 3-0	
`0x00000`	`0x0`	X	X	Return `0xDEADBEEF`
`0x10000`	`0x1`	X	X	Return `0xBEEFDEAD`
`0x40000`	`0x4`	X[1]	X[1]	Return FIFO output data
`0x80000`	`0x8`	X	X	Read data from echo register
				Buffer State
`0xF0000`	`0xF`	`0x0`	`0x0`	Return input size
`0xF0004`	`0xF`	`0x0`	`0x1`	Return input full
`0xF0008`	`0xF`	`0x0`	`0x2`	Return input empty
`0xF000C`	`0xF`	`0x0`	`0x3`	Return input write pointer
`0xF0100`	`0xF`	`0x4`	`0x0`	Return output size
`0xF0104`	`0xF`	`0x4`	`0x1`	Return output full
`0xF0108`	`0xF`	`0x4`	`0x2`	Return output empty
`0xF010C`	`0xF`	`0x4`	`0x3`	Return output read pointer
			bits 4-0[2]	Simulator state
`0xF0380`	`0xF`	`0xE`	`0x0`	Running
`0xF0384`	`0xF`	`0xE`	`0x1`	Current_address
`0xF0388`	`0xF`	`0xE`	`0x2`	Current_addr_valid
`0xF038C`	`0xF`	`0xE`	`0x3`	Prefetch_stimuli
`0xF0390`	`0xF`	`0xE`	`0x4`	Prefetch_addr_valid
`0xF0394`	`0xF`	`0xE`	`0x5`	Prefetch_mask
`0xF0398`	`0xF`	`0xE`	`0x6`	Clk_cnt

[1] Actually, these bits determine the address at which the data is read from the FIFO buffer.

[2] To reserve a larger addressing space for simulator state, 5 bits are used.

APPENDIX B

Test case: IIR filter

A test case is used to verify whether there are difficulties with the test flow, and to test the challenges for a system designer.

The test case is very basic, but nevertheless involves these aspects:

1. Homogeneous structure[1]
2. Easily adaptable number of taps, thus increasing the number of entities and the complexity.
3. Contains both forwards *and* backwards loops.

B.1 IIR filter

The example is an *Infinite Impulse Response (IIR) filter*. The equation for an IIR filter is:

$$y_n = \sum_{i=0}^{n} b_i x_{n-i} - \sum_{i=0}^{n} a_i x_{n-i} \tag{B.1}$$

This equation expands to:

$$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} - a_1 y_{n-1} - a_2 y_{n-2} \cdots \qquad with\ n = 0, 1, 2, \cdots \tag{B.2}$$

The equation above clearly shows several recognisable items of a regular structure. Graphically, this is even more clear (see figure B.1). An IIR filter can be designed in several structures [23]. Figure B.1 shows the direct-form structure of an IIR filter that is used for tests.

The regular structure of an IIR filter is easily recognisable in the forwards path, but somewhat less straightforward in the backwards part. The single entity which should be processed by the tool of Rutgers is shown in figure B.2, which is a very small piece of hardware.

[1]A number of changes to the structure are necessary to create an entirely homogeneous structure, however.

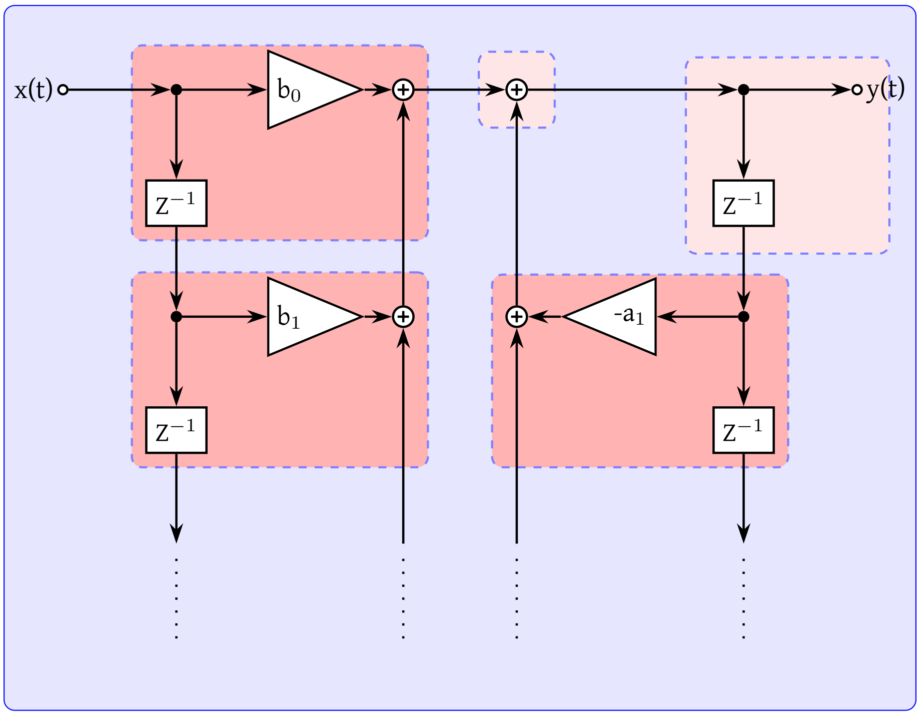


Figure B.1: Basic IIR filter

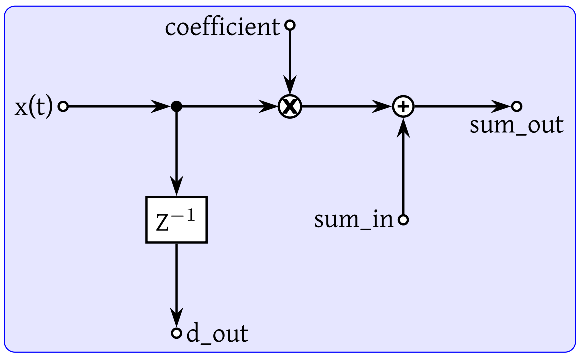


Figure B.2: IIR entity design

A first order IIR filter design has been made to test the outcome of the transformed design. The design for the IIR entity shown in figure B.2 is specified in listing B.2. The entity is equipped with several ports, specified in listing B.1.

The implementation of the asynchronous and synchronous part are explicitly separated for clarity. The IIR entity is used by the IIR filter top-level entity. The top-level entity uses the `generate` statement of VHDL for automated instantiation of a multi-taps IIR filter.

```vhdl
entity iir is
  port (
    reset       : in  std_logic;
    clk         : in  std_logic;
    x           : in  std_logic_vector (15 downto 0);
    coefficient : in  std_logic_vector (15 downto 0);
    sum_in      : in  std_logic_vector (15 downto 0);
    d_out       : out std_logic_vector (15 downto 0);
    sum_out     : out std_logic_vector (15 downto 0)
    );
end iir;
```

Listing B.1: IIR entity definition

```vhdl
-- ========= asynchronous ========
process(x,sum_in, coefficient)
   variable mult_out : signed(31 downto 0);
   variable sum_int  : signed(15 downto 0);
begin
   --calculate product
   mult_out := signed(x) * signed(coefficient);

   --calculate sum
   sum_int := mult_out(30 downto 15) + signed(sum_in);
   sum_out <= std_logic_vector(sum_int);
end process;

-- ========= synchronous  ========
--delay input sample with one clock cycle and provide reset circuitry
process (clk,reset)
begin
   if reset ='1' then -- ATTN: Active high reset!
      delay <= x"0000";
   elsif rising_edge(clk) then --force fdc flip-flop by if statement.
      delay <= x;
   end if;
end process;
```

Listing B.2: IIR entity behaviour VHDL code

B.2 Homogeneous structure

Figure B.1 shows that an IIR filter is not entirely homogeneous by the components filled light. To generate the IIR filter, a number of different conditions are used. The code to generate an IIR filter is added in listing B.3.

The transformation tool of Rutgers [33] is not able to process systems that are not fully homogeneous. The structure of the IIR filter must be changed for this reason. Two additional instantiations of the basic IIR entity were added to replace the irregular components.

Mathematically, the multipliers can be "bypassed" using coefficient 1. The IIR filter is constructed in a fixed-width system, which can not represent 1. A solution, which is correct, is to multiply two times by -1, as -1^2 is *exactly* 1. This requires the addition of more entities, which results in a total of seven entities for a first order IIR filter. To scale the filter to a second order only two entities need to be added and 4 to create a third order filter.

In the end, the design is built out of 7 blocks, which are tied together by a number of wires as shown in figure B.3. For readability, the coefficient input is not named, though the values for it are depicted.

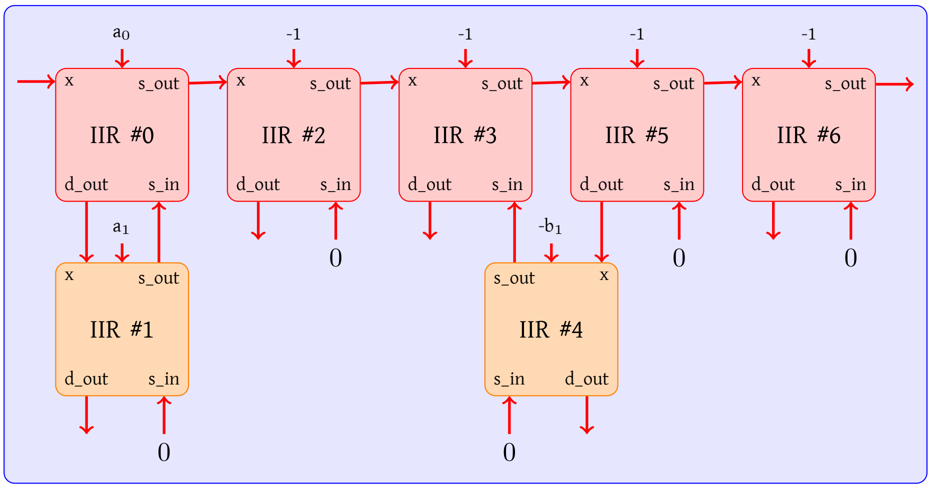


Figure B.3: IIR filter as block, homogeneous

B.3 Challenges

Usage of the transformation tool has posed challenges, which are discussed in this section. A common mistake on signal naming is discussed.

As specified by Rutgers [33, section 5.5.3], the tool can connect only one output to a single input. However, the IIR filter requires that both the output of the adder *and* the delay output can be connected to the 'x' input of an IIR entity, as shown in figure B.3. The hatched components are connected via different ports than the other components. In simulation, multiple link memory outputs try to write to the same signal simultaneously. This pitfall causes conflicts in the signal, which becomes invalid. Therefore, a manual change has been made to the wrapper entity; an additional multiplexer was added. This is suggested by Rutgers too. The transformation tool is not adjusted to support this feature.

A common practice in system design is to name a signal 'reset' when the reset condition is active when the signal is 'high', instead of 'low'. However, the transformation tool uses the name 'reset' for its active-low reset condition, which is misleading. Listing B.2 uses active-high reset for this reason.

```
--Instantiate FIR parts
fir: for tap in 0 to LENGTH_IIR generate
  fir_in : if (tap=0) generate -- Irregular: upmost block
    fir_part_1: entity work.iir port map( x,
                fir_delay(0),
                fir_sum_out(0),
                fir_sum_out(1),
                fir_coeff(0),
                clk, reset);
  end generate;
  fir_others : if (tap>0) generate -- Regular structure (tap 1 to end)
    fir_part : entity work.iir port map(fir_delay(tap-1),
                                        fir_delay(tap),
                                        fir_sum_out(tap),
                                        fir_sum_out(tap+1),
                                        fir_coeff(tap),
                                        clk,reset);
  end generate;
end generate;
-- Regular part
iir:  for tap in 0 to LENGTH_IIR+1 generate
  iir_irr : if (tap=0) generate
    total_part: entity work.iir port map(fir_sum_out(0),
                iir_delay(0),
                iir_sum_out(0),
                iir_sum_out(2),
                iir_coeff(0),
                clk,reset);
    iir_part_1: entity work.iir port map(iir_sum_out(0),
                                       iir_delay(1),
                                       iir_sum_out(1),
                                       zero,
                                       iir_coeff(0),
                                       clk, reset);
  end generate;
  iir_reg : if (tap>1) generate
    iir_part: entity work.iir port map( iir_delay(tap-1),
                                      iir_delay(tap),
                                      iir_sum_out(tap),
                                      iir_sum_out(tap+1),
                                      iir_coeff(tap-1),
                                      clk, reset);
  end generate;
end generate;
```

Listing B.3: IIR filter generation code

Table B.1: IIR filter entity resource consumption

Precision Synthesis			
Resource	Used	Available	Utilization
Function Generators	16	28672	0.06%
CLB Slices	8	14336	0.06%
Dffs or Latches	16	30832	0.05%
Block RAMs	0	96	0.00%
Block Multipliers	0	96	0.00%

B.4 Results

Synthesis results are listed in table B.1.

APPENDIX C

C code listings

This appendix contains the definitions of the structures that are used to describe the memory map for the processor in the BCVP platform. Similar to appendix A, the simulator control and state are expanded in individual structures (listings C.2 and C.3)

```
struct Hw_bridge{
  /* 0x00000000 */ volatile long data2fifo[IN_FIFO_SIZE];
  /* 0x00000040 */ fill(reserved, sizeof(data2fifo),0x10000); //Initial test size
  /* 0x00010000 */ volatile long test1;
  /* 0x00010003 */ fill(reserved0,0x10004,0x40000);
  /* 0x00040000 */ volatile long data2arm[OUT_FIFO_SIZE];
  /* 0x00040040 */ fill(reserved1,0x40000+sizeof(data2arm),0x80000); //Initial test size
  /* 0x00080000 */ volatile long echo;
  /* 0x00080004 */ fill(reserved2,0x80004,0xF0000);
  /* 0x000F0000 */ Fifo_state in_fifo;
  /* 0x000F0010 */ fill(reserved3,0xF0010,0xF0100);
  /* 0x000F0100 */ Fifo_state out_fifo;
  /* 0x000F0110 */ fill(reserved4,0xF0110,0xF02C0);
  /* 0x000F02C0 */ volatile long test2;
  /* 0x000F02C4 */ fill(reserved5,0xF02C4,0xF0300);
  /* 0x000F0300 */ Sim_ctrl control;
  /* 0x000F0318 */ fill(reserved6,0xF0318,0xF0380);
  /* 0x000F0380 */ Sim_state state;
  /* 0x000F0398 */ // End  address
};
```

Listing C.1: Structure describing memory map in processor

```
struct Sim_ctrl{
  /* 0x0000 */ volatile long cmd;
  /* 0x0004 */ volatile long addr;
  /* 0x0008 */ volatile long read_from;
  /* 0x000C */ volatile long stimuli_mask;
  /* 0x0010 */ volatile long write_to;
  /* 0x0014 */ volatile long enable;
};
```

Listing C.2: Simulator control struct

```
struct Sim_state{
  /* 0x0000 */ volatile long running;
  /* 0x0004 */ volatile long current_addr;
  /* 0x0008 */ volatile long current_addr_valid;
  /* 0x000C */ volatile long prefetch_stimuli;
  /* 0x0010 */ volatile long prefetch_addr_valid;
  /* 0x0014 */ volatile long prefetch_mask;
  /* 0x0018 */ volatile long clk_cnt;
};
```

Listing C.3: Simulator state struct

```
extern unsigned char link(long address, long read_from, long stimuli_mask,
              long write_to){
  hw_bridge->control.cmd = CMD_CREATE_LINKS;
  hw_bridge->control.addr = address;
  hw_bridge->control.read_from = read_from;
  hw_bridge->control.stimuli_mask = stimuli_mask;
  hw_bridge->control.write_to = write_to;
  return 1;
}
```

Listing C.4: Instantiate link memory function

Bibliography

[1] Altera. *DSP Builder User Guide*, 8.0 edition, May 2008.

[2] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, 2002.

[3] UC Berkeley. Research Accelerator for Multiple Processors (RAMP) project. `http://ramp.eecs.berkeley.edu/`. Last checked August 15, 2008.

[4] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2008-28, EECS Department, University of California, Berkeley, April 2008.

[5] W. Brugger, M. Hofinger, and 4S partners. *4S BCVP specification*. Atmel Germany, April 2004.

[6] Controllab Products B.V. 20-sim. `www.20-sim.com`. Last checked August 15, 2008.

[7] Allan Cantle and Robin Bruce(Nallatech). An introduction to the Nallatech Slipstream FSB-FPGA Accelerator Module for Intel Platforms & the Nallatech High-Level Toolset. White paper NT 405-0342, September 2007. NT 405-0342 `http://www.nallatech.com/mediaLibrary/images/english/6615.pdf`.

[8] Ben Chan, Nabeel Shirazi, and Jonathan Ballagh. Achieving high-bandwidth DSP simulations using Ethernet Hardware-in-the-Loop. *DSP Magazine*, 2:42–44, May 2006. Xilinx.

[9] C. Chang, C. Chang, J. Wawrzynek, and R.W. Brodersen. BEE2: a high-end reconfigurable computing system. *IEEE Design & Test of Computers*, 22(2):114–125, 2005.

[10] ChiasTek. Cosimate: Heterogeneous co-simulation in distributed environments. `http://www.chiastek.com/products/cosimate.html`, 2008. Last checked August 15, 2008.

[11] Eric S. Chung, Eriko Nurvitadhi, James C. Hoe, Babak Falsafi, and Ken Mai. Virtualized Full-System Emulation of Multiprocessors using FPGAs. In *Workshop on Architectural Research Prototyping (during ISCA2007)*, volume 2, New York, NY, USA, June 2007. ACM.

[12] Eric S. Chung, Eriko Nurvitadhi, James C. Hoe, Babak Falsafi, and Ken Mai. A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs. In *FPGA '08: Proceedings of the 16th international*

ACM/SIGDA symposium on Field programmable gate arrays, pages 77–86, New York, NY, USA, February 2008. ACM.

[13] Moo-Kyoung Chung and Chong-Min Kyung. Enhancing performance of HW/SW cosimulation and coemulation by reducing communication overhead. *IEEE Transactions on Computers*, 55(2):125–136, February 2006.

[14] Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, and Koninklijke Philips Electronics N.V. Universal serial bus specification revision 2.0, 2000. downloaded from `www.usb.org`.

[15] Daniel Lázaro Cuadrado, Anders P. Ravn, and Peter Koch. Automated Distributed Simulation in Ptolemy II. In *Proceedings on the 25th IASTED International Multi-Conference Parallel and Distributed Computing and Networks (PCDN'07)*, pages 139–144, Anaheim, CA, USA, February 2007. ACTA Press.

[16] A.S. Damstra. Virtual prototyping through co-simulation in hardware/software and mechatronics co-design. Master's thesis, Control Laboraty, University of Twente, April 2008.

[17] Daniel Denning(Nallatech). Accelerate and verify algorithms with the XtremeDSP development kit-II. *Xcell Journal Online*, (49), Summer 2004.

[18] John W. Eaton and Community. GNU Octave. `http://www.gnu.org/software/octave/`, 2008. Last checked August 15, 2008.

[19] Nicolas Genko, David Atienza, and Giovanni De Micheli. NoC Emulation on FPGA: HW/SW Synergy for NoC Features Exploration. In *Proceedings of the International Conference on Parallel Computing (ParCo2005)*, volume 33, pages 753–760, Jülich, Germany, September 2005. John von Neumann Institute for Computing (NIC).

[20] B. Gestner and D.V. Anderson. Automatic generation of ModelSim-Matlab interface for RTL debugging and verification. In *Proceedings of the 50th Midwest Symposium on Circuits and Systems MWSCAS 2007*, pages 1497–1500, Los Alamitos, CA, USA, August 2007. IEEE.

[21] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Third edition, May 2003. ISBN 1-55860-596-7.

[22] K. J. Hines. Pia: A framework for embedded system co-simulation with dynamic communication support. Technical Report TR-96-11-04, University of Washington, October 1996.

[23] Douglas L. Jones. IIR Filter Structures. Connexions `http://cnx.org/content/m11919/latest/`, December 2004. Last checked August 15, 2008.

[24] A. Lafage. *HSMC3 functional specifications.* Atmel Coorporation, April 2003.

[25] V. S. Lin and D. E. Pansatiankul. Hardware-accelerated simulation tool for end-to-end communication systems. In *Proceedings of the IEEE Global Telecommunications Conference GLOBECOM '06*, pages 1–5, Los Alamitos, CA, USA, November 2006. IEEE.

[26] The Mathworks. EDA Simulator Link™MQ. `http://www.mathworks.com/products/modelsim/`, March 2008. (version 2.4) Last checked August 15, 2008.

[27] The Mathworks. MATLAB®- the language of technical computing. `http://www.mathworks.com/products/matlab/`, March 2008. (version 7.6) Last checked: August 15, 2008.

[28] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. Streaming-data algorithms for high-quality clustering. In *Proceedings of the 18th International Conference on Data Engineering*, pages 685–694, Los Alamitos, CA, USA, February 2002. IEEE Computer Society.

[29] K. Olukotun, M. Heinrich, and D. Ofelt. Digital system simulation: methodologies and examples. In *Proc. Design Automation Conference*, pages 658–663, Los Alamitos, CA, USA, June 1998. IEEE Computer Society.

[30] Jingzhao Ou and V.K. Prasanna. MATLAB/Simulink Based Hardware/Software Co-Simulation for Designing Using FPGA Configured Soft Processors. In *Proc. 19th IEEE International Parallel and Distributed Processing Symposium*, pages 148b–148b, Los Alamitos, CA, USA, April 2005. IEEE.

[31] K.N. Parashar and N. Chandrachoodan. A novel event based simulation algorithm for sequential digital circuit simulation. In *Proc. International Conference on Field Programmable Logic and Applications FPL 2007*, pages 792–795, Los Alamitos, CA, USA, August 2007. IEEE.

[32] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, March 1987. IEEE Computer Society Press.

[33] J. H. Rutgers. Automated sequential hardware-in-the-loop simulator generation. Technical report, University of Twente, November 2007.

[34] A.S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002. ISBN 0-13-121786-0.

[35] C. A. Verhaar. An integrated embedded control softwaredesign case study using Ptolemy II. Master's thesis, Control Laboratory, University of Twente, May 2008.

[36] Virtutech. Simics for multicore software. White paper, June 2007. `www.virtutech.com`.

[37] Michael C. Williamson. *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*. PhD thesis, EECS Department, University of California, Berkeley, 1998.

[38] P. T. Wolkotte, P. K. F. Hölzenspies, and G. J. M. Smit. Fast, Accurate and Detailed NoC Simulations. In *Proceedings of the 1st ACM/IEEE International Symposium on Networks-on-Chip, Princeton, NJ, USA*, pages 323–332, Los Alamitos, May 2007. IEEE Computer Society Press.

[39] P. T. Wolkotte, P. K. F. Hölzenspies, and G. J. M. Smit. Using an FPGA for Fast Bit Accurate SoC Simulation. In *Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS'07) - 14th Reconfigurable Architecture Workshop (RAW 2007), Long Beach, CA, USA*, page 167, Piscataway, March 2007. IEEE Computer Society Press.

[40] Xilinx. Technology solutions - connectivity. `http://www.xilinx.com/products/design_resources/conn_central/index.htm`. Last checked August 15, 2008.

[41] Xilinx. Virtex-II platform FPGAs: Complete data sheet, November 2007. DS031, version 3.5.

[42] Xilinx. *System Generator for DSP User Manual*, 10.1.1 edition, April 2008.

[43] Xilinx, Justin Delva, Adrian Chirila-Rus, Ben Chan, and Shay Seng. Using system generator for systematic HDL design, verification and validation. White Paper, January 2008.