Masterarbeit

A Combined Approach for Debugging Hardware and Software in SystemC Instruction Set Simulations

Martin Lang

Institut für Technische Informatik
Technische Universität Graz

Begutachter: Univ.-Ass. Dipl.-Ing. Dr. techn. Christian Steger

Betreuer: Dipl.-Ing. Johannes Loinig

Graz, im Dezember 2012

Abstract

In the last decades systems consisting of hardware and software modules have become a part of our daily life. With the increasing complexity of these applications also the debugging process is becoming more and more complicated. To cope with this development it is necessary for the designers to be supported by new tools that make it possible to debug the new and complex systems. Most tools nowadays help the developers to debug either hardware or software. As many errors only occur when hardware and software are developed together this is often still a problem for debugging.

This master thesis gives an overview about the existing ways for debugging applications. Also the design, implementation and results of a new debugging system for hardware/software components are presented. With this system it is possible to debug hardware and software at the same time using traditional debugging methods. Therefore SystemC models of the hardware components are used. Using different *Integrated Development Environments* (IDE) this new debugging environment is also able to work in a distributed system. This is possible because all connections between the different components are realised using sockets. To verify the implementation and to show how a new processor model can be added to this new debugging environment, a *Transaction Level Modeling 2.0* (TLM 2.0) model of an 8051 microprocessor was integrated. To demonstrate the abilities of the new debugging framework and to show the integration process of new modules, also other SystemC models have been evaluated.

Kurzfassung

In den letzten Jahrzehnten haben Systeme, die sowohl aus Hardware als auch Software Modulen bestehen, Einzug in nahezu alle Bereiche des täglichen Lebens gehalten. Durch die immer größere Komplexität dieser Programme gestaltet sich auch die Fehlersuche immer schwieriger. Um mit dieser Entwicklung Schritt halten zu können, ist es notwendig den Entwicklern neue Tools zur Verfügung zu stellen, um die neuen Produkte auch testen zu können. So gibt es zurzeit sowohl für Software als auch für Hardware verschiedene Hilfsmittel für das Debugging. Das Problem dabei ist, dass viele Fehler erst dann auftreten, wenn Software und Hardware zusammen entwickelt werden.

Diese Masterarbeit gibt nun einen Überblick über bestehende Möglichkeiten der Fehlersuche. Zudem wird das Design, die Implementierung und die Ergebnisse eines Debugging Systems beschrieben mit dem man Software und Hardware gleichzeitig, mit traditionellen Debug Methoden, testen kann. Dabei werden SystemC Modelle der Hardware Komponenten verwendet. Im Zusammenspiel von verschiedenen Integrated Development Environments (IDE) ergibt sich somit ein Debugging System welches, durch die Vernetzung der einzelnen Komponenten mit Sockets, auch in verteilten Systemen eingesetzt werden kann. Um die Implementierung zu verifizieren und um zu zeigen welche Schritte notwendig sind um einen anderen Mikroprozessor in das neue Debugging System einzufügen, wurde ein Transaction Level Modeling 2.0 (TLM 2.0) SystemC Model eines 8051 Mikroprozessors integriert. Außerdem wurden verschiedene SystemC Modelle untersucht, um damit die Möglichkeiten des neuen Debugging Systems und den Integrationsprozess von neuen Modulen zu präsentieren.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Danksagung

Im Rahmen dieser Danksagung möchte ich mich bei all jenen Personen bedanken, die mir den Abschluss dieser Masterarbeit ermöglicht haben.

Bedanken möchte ich mich bei Univ.-Ass. Dipl.-Ing. Dr. techn. Christian Steger für seine Unterstützung und seine Geduld. Mein Dank gilt auch meinem Betreuer Dipl.-Ing. Johannes Loinig, der mir oft mit seinem fachlichen Ratschlägen weiterhelfen konnte und immer ein offenes Ohr für meine Fragen hatte.

Besonders möchte ich mich auch bei meinen Eltern bedanken, die mich während meiner Studienzeit gefördert haben, ohne zu fordern, und auch in schwierigen Zeiten immer für mich da waren. Außerdem möchte ich mich bei meiner Frau Marlene bedanken, die immer hinter mir gestanden ist und mich mit aller Kraft unterstützt hat.

Ohne euch alle wäre diese Masterarbeit nicht zustande gekommen und daher noch einmal ein großes: Danke!

Graz, im Dezember 2012 Martin Lang

Contents

List of Figures

1. Introduction

Electronic systems have become an essential part of our everyday life. Especially in the last decades the number of hardware/software systems has increased very much. Every new generation of electronic devices has a bigger range of functionality, resulting also in an increasing complexity of the systems. Like Gordon Moore already predicted in 1965 in his thesis, later known as the famous *Moore's law*, the number of components on an integrated circuit doubles every two years [25]. This high complexity makes it harder for the designers to implement, but also debug the applications. It gets even worse when hardware and software are developed together, because many errors occur only when both parts are interacting with each other. Also when dealing with embedded systems the level of difficulty increases (see Chapter 2.7.5) even more [19].

One way to cope with this rising complexity is by using a hardware/software co-design approach (see Chapter 2.3). Using such a design flow allows the developers to implement hardware and software at the same time. Therefore the system is split up into different parts that are either implemented in hardware or software. By using simulations (see Section 2.1) through the whole design process the hardware and software side are verified together. Also the requirements of new applications, regarding for example power comsumptions, can be integrated into this design flow.

The problem of finding errors is not new. Even long time before the first computers were built people had to find the source of errors in their machines. Shortly after the first programs have been implemented and electronic machines have been created a new discipline for software/hardware developers has emerged: Debugging, the hunting for bugs (see Chapter 2.7). The debugging consumes a huge part of the time for creating a new product. As the time to market pressure is getting higher and higher for the companies, often also the testing and therefore also the debugging time gets reduced to finish a project faster. This can cause a huge problem as it is very important to find bugs as early as possible. The later you find a design error the more expensive it is for the whole project to correct it. Another problem are products with many unresolved bugs in them because of too less testing time. There exist of course tools that can help the developers to find the errors faster, but often even those tools are not able to deal with complex and dynamic failures. Therefore new tools and debugging systems are necessary to enable hardware and software debugging at the same time. The tools will help the developers to find even complex bugs and to reduce the overall debugging time.

Even more difficulties arise when hardware and software have to be debugged together to find the errors in both parts. As real hardware components have only very limited access possibilities once they are built, one way to cope with this problem is to simulate the hardware in software. This enables developers to use debugging methods known from the

software development also on the hardware components. Especially when developing in a hardware/software co-design flow the simulation of components is very important to have a possibility for debugging.

This master thesis shall give an overview of the state of the art methods and tools to debug hardware and software simultaneously. Also the problems testing of embedded systems and the design, implementation and results of a new debugging system to enable hardware/software co-design debugging will be presented.

1.1. Objective

To do a research on state of the art methods to debug hardware, software and both in combination was one topic of this master thesis (Chapter 3). The main goal was to design and implement a new debugging system that can handle the following objectives:

- Ability to debug software and hardware modules at the same time
- Usage of traditional debugging methods

This new debugging system will help developers to debug hardware and software simultaneously using existing debugging IDE tools and supports the integration of different models. In a preliminary work a TLM 2.0 SystemC model of an 8051 microprocessor was implemented together with a group of students [12] (see section 5.4). This model and some other open source models are used in this master thesis to verify the design and demonstrate the necessary steps for an integration into the debugging environment.

1.2. Thesis Outline

This section shall give an overview about the following chapters of this master thesis: In Chapter 2 the theoretical background is explained, demonstrating the ideas of hardware/software codesign, the basics of the modeling language SystemC and the Transaction-Level Modeling standard. The second part of this chapter takes a closer look on the debugging technology, showing a bit about the history and explaining different debugging methods in detail. One focus lies thereby on embedded systems and how to find the errors in them (see Chapter 2.7.5).

The chapter afterwards presents the related work in this area (see Chapter 3). Different approaches are presented, ranging from software based solutions to additional hardware debugging features.

In Chapter 4 and Chapter 5 the design and implementation of the new debugging system created for this master thesis are presented. The design is examined from different design perspectives, e.g. logical, process and development view. The implementation chapter describes the environment that is necessary to run the system (see Section 5.1), specifies

the interfaces used to connect the single parts (see Section 5.2) and finally presents the new modules that have been implemented (see Section 5.3).

Then the results are presented and discussed in Chapter 6, showing the integration of an 8051 TLM 2.0 microprocessor model into the new debugging system. This is done to verify the design and implementation and to show the necessary steps for such an integration. Also the evaluation of different SystemC open source components together with the new debugging system is part of this chapter.

The master thesis is closed with the last chapter, giving a conclusion about the whole topic and also pointing out possible future work for the new debugging system and hardware/software debugging in general (see Chapter 7).

2. Background

This chapter shall give an overview about the topics of this master thesis, presenting the theoretical background and describing the main concepts that are used. The focus of this master thesis lies on the design and implementation of the new debugging system, so the basic information for the concepts can be found here. More detailed descriptions about those topics can be found in the corresponding references. But before the main concepts of creating the design for a system consisting of hardware and software will be explained, this section shall give a short introduction to the concept of simulations in principal.

2.1. Simulation

In this section the main idea behind simulations is explained and the advantages and disadvantages of this approach will be presented.

But what exactly is a simulation? In the book *Processor description languages: applications and methodologies*[2] the term *simulation* is defined as:

> A Simulation is the imitation of the operation of a real-world process or system over time.

This means that a simulation is used to gain information about a system, to describe it and to assist the developers with an early verification of the new design. Especially the last point, the early verification of the design, is very important for the developers. It enables them to find faults very early in the design flow. This is good because the later a fault is found, the more expensive it is to fix it. Especially when dealing with design faults this can be very critical for the whole project.

Other advantages when using simulations are:

Receiving insight information of the system: When dealing with real components, for example a new hardware, it is often very complicated and sometimes even not possible to receive information about the inside behaviour of it. With simulations and models of those components the developers can receive this data and analyze it. This is necessary to understand the detailed behaviour of the system and to be able to debug it.

Exploring possibilities: When using simulations it is also easy to test your new component with different settings. This can be for example another operating policy, a different timer setting or any other changes to the models.

Establishing requirements: With the insight information and the possibility to change the models it is also possible to define requirements for the new system.

Share knowledge: When working in a team simulations can be a good way to share the knowledge about the system among the team members. Also when new people join the team, simulations can be a good starting point for them.

But developers also have to deal with some disadvantages when they are using simulations for their projects:

Creating the models: Before a simulation can be started all the necessary models have to be created in advance. This needs special knowledge of the developer to create them properly. Also it is complicated to compare two different models of the same component with each other, as they are strongly depending on their implementation.

Time consuming: To analyze the results and to run the simulation itself is often a time consuming task. Especially in the last decades the components have become more and more complex and it takes more and more effort to run a simulation for them.

More information about simulations can be found in the book of Jerry Banks: '*Processor description languages: applications and methodologies*' [2]. This master thesis will focus on the debugging aspect in such simulations and therefore concentrates on the hardware and software design and implementation.

2.2. Hardware Emulation

When developing new systems that consist of hardware and softare parts, the developers also have to think about the debugging in these systems. Especially the debugging of the hardware component parts can be difficult and time consuming. One way to cope with this problem is to use hardware emulators. As described in the book '*Electronic Design Automation For Integrated Circuits Handbook*' written by Luciano Lavagno [11] an emulator is a hardware component that imitates the behavior of another hardware part, for example a microprocessor. The big advantage of this approach is, that the developers can access debug data inside the emulator directly by reading registers and signals. So for the system everything seems to be normal as the functionality of the hardware is imitated by the emulator, but for the developers this hardware component is under full control. This is also a huge advantage in comparison to a protoype of the new hardware where this debug functionality is not available in most cases. The main disadvantages of this approach is the lower performance that can be achieved when using an emulator.

Emulators are from importance because they can be used as in-circuit emulators directly on the chips. This enables the developers to get information from the hardware component which can be used for the debugging process. More information about this topic and other research projects can be found in the related work section of this master thesis (see Chapter 3).

2.3. Hardware/Software Co-Design

As described in the article by Wayne Wolf [30], the name hardware/software co-design has been established in the early years of the 1990s. Back then it was soon clear that microprocessors would also play an important role for *integrated circuits* (IC). Since that times, following Moore's law [25], more and more microprocessors and subsystems have been implemented onto a microchip, combining hardware and software in one system. The hardware/software-co design approach tries to cope with the problems of designing such systems:

> Hardware/software co-design tries to increase the predictability of embedded system design by providing analysis methods that tell designers if a system meets its performance, power and size goals and synthesis methods that let researchers and designers rapidly evaluate many potential design methodologies [30].

When designing a hardware/software co-design system the following tasks are important for the developers [31]:

Partitioning: splitting up the main task into smaller parts.

Allocation: decision at wich location the parts from the partitioning will be implemented (hardware/software)

Sheduling: timing the execution of the single system parts.

Mapping: implementing the system parts on the specified components.

This can also be seen in Figure 2.1 [10].

Everything starts with a high level description of the complete system. When this model is established the next step is the partitioning of the system. In this phase the high level model is split up into hardware and sofware parts. This is important as during the partitioning also the first decision about the physical design of the final product is made. The next steps use those software and hardware models to implement the software and to create a hardware implementation using a HDL. This phase is called *mapping*. The final step in the design flow is the compilation of the software and the synthesis of the hardware parts. Togehter they form the final system. Between every step in the design flow, also co-simulations of the hardware and software side are executed. The intention of those simulations is a constant check if the achieved system is valid and matches the desired requirements.

Other interesting requirements for the hardware/software co-design flow are created by the low-power and power-aware designs. Co-synthesis can help the developers to optimize their systems to the low power profile and stay within the desired power limits. This trend has also started in the mid 1990 and still goes on. With an increasing number of mobile devices, the power consumption has become a very crucial part for many systems.

All these requirements and the physical aspects of embedded system do not only influence the complete design and implementation of them, but also have an impact on the debugging processes. More information about the debugging of embedded systems can be found in Chapter 2.7.5.

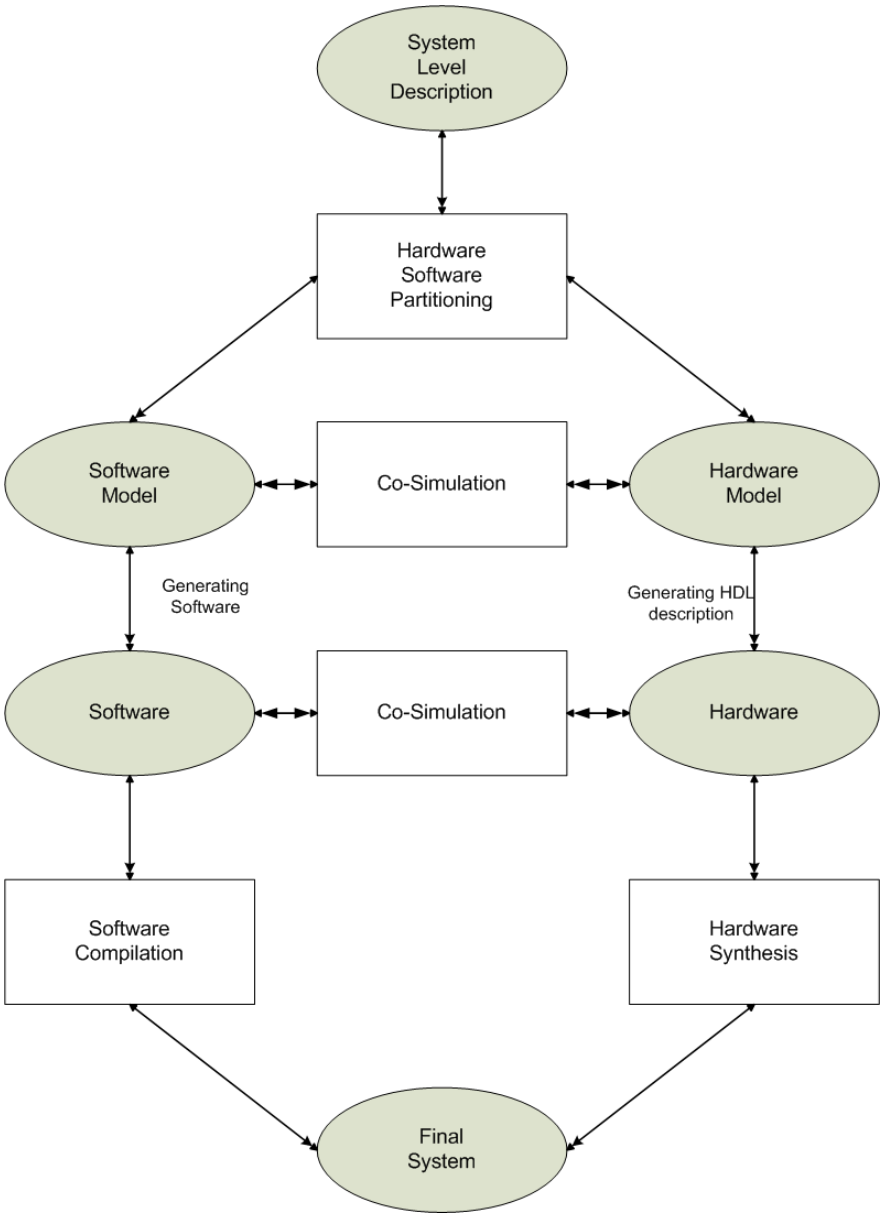

Figure 2.1.: Hardware/Software Co-Design flow

There is one more reason why hardware/software co-design was so successfully and still is: the field-programmable gate arrays (FPGA). With the invention of the FPGAs it was possible for the designers to configure the FPGA after the manufacturing. By using a hardware description language (HDL) and logical blocks it was now possible to create everything from simple logical gates to new complex functions. This allows the designers to create much more flexible solutions and also verify their implementations much faster.

Since the early years of the new millennium also new system-level design languages have been created. One of them is SystemC, an extension to the C++ programming language. See also the following Section 2.4 for more information about SystemC. To know details about this HDL is important as it is used for modeling the system components (microprocessor and additional modules) in the new debugging system presented in this master thesis. Also the models which are used for the evaluation are open source SystemC components (see Section 6).

2.3.1. Embedded Systems

A very good example for systems that combine hardware and software in one application are *Embedded Systems*. One of the first article that gives an insight and an overview of the hardware/software co-design flow for embedded system can be found in [31]. The first embedded systems have been used in the banking sector for handling transactions and their storage. Since back then the systems have evolved in almost all aspects. The product range of embedded systems is very wide nowadays, including all kinds of electronical devices. Example devices are:

Simple systems: simple comsumer electronics with an input interface, for example a microwave oven

Portable devices: for example mobiles phones

Industrial systems: controllers for factories

Critical controller: security controllers in a production unit

For portable devices some constrains have a higher priority: The power consumption of the system and the size of it. Those products are used on the road and therefore it is not possible to recharge them again easily. And also the size is an important factor when carrying the system around. Another constrain is very important for the industrial systems: the reliability and maintainability. As those controllers are often used in factories to ensure the security of machines and product lines it is very important that they have a very low fault rate and downtime.

Because of the strong conncetion between the hardware and software parts they have to be designed together. This is necessary to implement the desired functionality of the system, but also to be able to meet certain restrictions for the new product. This includes limits for the energy comsumption, the performance of the system and costs of the developement.

In the beginning, when the first embedded systems have been created, it was possible to implement the complete product in advance and then test it. Nowadays this is impossible. Even the small and simple applications include often a few microprocessors, together with the software running on them and the communication infrastructure between the single parts.

2.4. SystemC

One of the main problems in describing hardware and software at the same time is how to cover up the big differences between them. One solution for this is to transfer the hardware modules into software components. Therefore a way to rebuild the hardware features (parallel processing, timing...) in software is needed. At this point HDLs come into play [23].

At the end of the 1990s some big companies formed the Open SystemC Initiative (OSCI)[1]. The goal of this initiative was to develop a new way to exchange code modules from different companies more easily and to enable a hardware/software co-design flow. The result of their work was SystemC. The new HDL uses the programming language C++ as its basis with additional support to define and implement hardware modules. This new functionality is added to the normal C++ environment with additional header files. All the normal object oriented features of C++ (definition of classes, inheritance, templates,...) are therefore available, including the new features to simulate the timing, parallel execution, hardware data types and synchronous/asynchronous processes. Main components of SystemC are:

Module: Modules can be compared to blackboxes because from the outside one cannot see what is happening in their internals. They are used to split up the system into smaller parts and can contain other modules or signals as well. The benefits are the reduced complexity of the system by splitting it up and the easier maintenance of code in a module. As long as the ports do not change, the interface to the outside stays the same.

Port: The ports of a module represent the interface to the outside. They can either handle input data (sc_in), output data (sc_out) or both directions (sc_inout).

Channel: In SystemC channels are used to connect one port to the other and therefore also one module to the another one. This connection can be established for example by using a signal or a more complex mechanism. More complex connection methods are for example buffers or semaphores.

Process: In SystemC there exist mainly two different types of processes: methods and threads. They represent different kinds of functionalities for the modules. Both kinds are usually created statically and represent a code block that is processed sequentially. There are a few differences between them. Threads for example can only be started once and are called automatically at the startup of the simulation, while methods can be used more than once. Another big difference lies in the execution: While threads can be suspended during their execution, methods cannot be stopped once they are started. Both of them can be controlled via static or dynamic sensitivity. The difference is that

[1] http://www.systemc.org/home

the dynamic sensitivity, as the name implies already, can change the sensitivity of a method or thread on the fly during runtime. A special kind of process is the clocked thread. In contrary to the normal thread process, the clocked thread does not have a sensitivity list but is triggered on the rising or falling edge of the clock signal.

Datatypes: In addition to the datatypes known from other programming languages, SystemC introduces a set of new, more hardware oriented types (see Tabel 2.1).

Simulation kernel: The simulation kernel of SystemC works in a cyclic way. During one clock cycle all processes that had a changing signal are computed. Their signal values are then updated with the next changing edge of the clock. The kernel is also implemented in an event-driven approach. This allows the execution of parallel processes, which is necessary to simulate hardware correctly.

datatyp	description	possible values
sc_bit	1 bit value	0 or 1
sc_bit	1 bit value	0 or 1
ets sc_logic	1 bit value	0, 1, X (undefined) or Z (high resistance)
sc_int	64 bit signed value	user defined
sc_int	64 bit unsigned value	user defined
sc_bigint	signed int with various size	user defined
sc_biguint	unsigned int with various size	user defined
sc_bv	bitvector	0 or 1
sc_lv	bitvector	0, 1, X or Z

Table 2.1.: SystemC datatypes

2.4.1. Transaction-level Modeling 2.0

The time is an important factor for a simulation. Depending on the abstraction level and the complexity of a system, simulations can be executed in only a few seconds, but can also need days to be finish. Therefore it is very important to choose the right abstraction level for the simulation. TLM 2.0, a new standard from the OSCI, is describing the systems on a very high abstraction level. The benefits choosing the TLM 2.0 standard are a very low simulation time and also an early possibility to verify the new design [13].

Another important key aspect of transaction level modeling is the seperation of the actual communication mechanism from its implementational details. The communication plays a big role in most systems that consist of different components. When using a TLM approach the communication is reduced to a very abstracted view, where it is more important to know from where and to what destination the data has to be sent. The low level implementational details of those connections are hidden by interfaces. This enables the developers to focus on the implementation of the system and the arrangement of the components in the communication network. Because of the interfaces that are used, it is also possible to try different communication mechanism easily. The complete communication systems can be replaced by

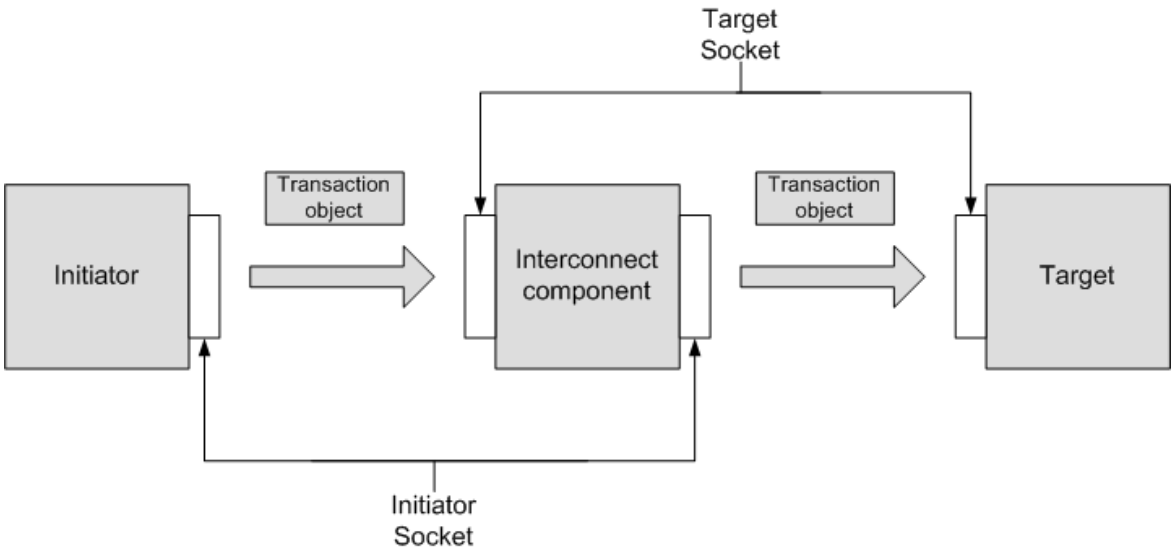

Figure 2.2.: TLM 2.0 example

another technology, for example a different bus system, and the system model will still work normaly as long the interfaces are implemented correctly.

TLM 2.0 is created on the base of SystemC and is used to transfer data packets between the single modules in the system. The main modules of TLM hold the functionality and are accessed via interfaces using TLM 2.0 sockets. This moves the communication of modules to a very abstract layer, resulting in a high simulation speed and an easily exchangeable design. For example different kinds of bus systems can be evaluated by only replacing the bus component as the sockets stay the same.

To establish a connection in a system between the components two different sockets are used in TLM 2.0. The initiator socket and the target socket. They enable the foward and backward path for sending the data. Also included in them is the possibility to use a blocking or non-blocking transport mode. Another important part is the generic payload which is used to enable the interoperability of the data transmissions in the system. If the exact details of the used bus protocol are not known or not important, the generic payload can be used as an abstract memory-mapped bus model. On the other hand, if a specific protocol is required, it is easy to implement it onto the base of the generic payload.

As described in the reference manual for TLM 2.0 [17] an example system can be seen in Figure 2.2.

In this example the system consists of three components: The *Initiator* that is used to send the information, the *Interconnect Component* which is used as a link between the other two components and the *Target* which represents the destination for the transmitted data. Of course this is just a simple example and therefore in a real system much more interconnect components can be used as links between the initiators and the destinations. In Figure 2.2 also the two socket types that are used to connect the components are presented.

While the *initiator socket* is always used to start the transmission and send out the data, the corresponding *target socket* handles the incoming transaction object. By connecting the components with these two socket types, the complete communication network can be established in a TLM 2.0 system.

2.5. Joint Test Action Group

The IEEE *Joint Test Action Group* (JTAG), also known as *IEEE 1149* standard, is a collection of different methods to debug and test integrated circuits. One of the most common methods is for example the *Boundary Scan Test* [6]. The IEEE 1149 standard defines the access, control and testing of digital circuits via a serial communication. Therefore five pins on the chips are used. As the JTAG pins are the only way to access an IC and to get information out of it after it has been built, those pins are also often "hijacked" to be used for enabling debugging support. This standard is also from importance for the related work of this master thesis, because a few projects described in this chapter use it to implement the debugging functionality in their systems.

2.6. Instruction Set Simulation (ISS)

To enable an early debugging of a new processor model special simulation devices are needed. An instruction set simulator can be one of them. An ISS is also used in the new debugging framwork presented in this master thesis and therefore this section shall explain the debugging device ISS a bit more in detail.

Instruction set simulators are often implemented in a high level programming language. They are used to simulate the behavior of a microprocessor. Therfore the ISS has to implement the complete instruction set of the corresponding processor. As described in the book '*Processor Description Languages*' written by Prabhat Mishra and Nikil Dutt [14] there are mainly four different ways to implement this simulation:

- Interpretive simulation
- Compiled simulation
- Just-in-time cache compiled simulation
- Hybrid simulation

The first kind of implementation, the *interpretive simulation*, simulates the behavior of the processor at runtime. Each instruction is loaded from the memory, decoded and executed when it is needed. The advantage of this approach is a reduced memory consumption and also a high flexibilty as the complete simulation is implemented in software. This is also the main disadvantage for this approach as it reduces the performance of the system.

In contrary to the implementation style before the *compiled simulation* uses a different technique. When using compiled simulations the instructions are fetched in advance, decoded

and then stored back into the memory. This time the huge amount of memory that is needed is the main disadvantage, while the execution speed can be increased with this approach.

Just-In-Time Cache Compiled Simulation is the third way to implement a instruction set simulator. It is a combination of the two perviously discussed implementation types. By combining the advantages of the other two methods a lower memory space and a higher performance can be achieved. This is established by using a cache for a defined amount of instructions. Before a new instruction is fetched and processed the cache is checked if this instruction is already stored in it. If this is the case, the values can be used as they are and no further processing is necessary. This approach is always a tradeoff between the cache size, and therefore the memory consumption, and the performance.

The last approach is called *hybrid simulation*. When developing a new system there are parts in this system that are more important then others. Therefore it makes sense to simulate those parts more detailed than the other parts. The idea of the hybrid simulation approach is to seperate the simulation into two different parts, depending on their importance. While the important parts are simulated as described in the previous ISS implementation, the less important parts are implemented as functions in the application.

For the 8051 microprocessor model, presented in this master thesis, the instructions were implemented in software completely. A detailed description of them can be found in Chapter 5.

2.7. Debugging

There are different theories where the names *bug* and *debugging* came from in the beginning [19]. The common thinking is that they first appeared after the incident that happened to Grace Hopper, one of the first compiler engineers. Her computer, at that time a room filling machine, was broken because of a bug that crawled into it. The bug was attracted by the cathode ray indicator of the system and then caused a short circuit. From that day on, the name *bug* was related to errors in computer hardware and software, while *debugging* is describing the process of finding and removing those faults. To assist the designers to find those bugs in their programs and/or hardware many tools have been created. A description about the techniques can be found in Section 2.7.4.

As already mentioned in the introduction of this master thesis, in the last decades many developing tools to debug hardware or software have been implemented. As both areas are very different from each other, also their debugging tools vary very much (see the following Sections 2.7.1 and 2.7.2 for more information regarding these topics) [19].

But tools cannot do anything without the people who operate them. And this is also one of the 9 rules from David J. Agans's book: *Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems* [1]:

> Understand the System: Know your tools

Only if the designers know how their tools work, how they have to be operated and in what situations they can be used, the results will be correct and useful. Agans compares it to the use of a thermometer. If the one operating it knows which side of the thermometer has to be put into the mouth of the patient it will work correctly. Otherwise the result will be only the temperature of the room.

Tools can be a really big help for a developer. They can, in the case of debugging, increase the developing speed very much and can help to find bugs faster. But at least as important as a good tool is a good designer that operates it. See also Section 2.7.4 where the main methods for debugging tools are described.

2.7.1. Hardware Debugging

Depending on the phase of the hardware development, different kind of tools can be used for debugging. In the design phase mostly tools are used that are able to simulate the hardware components and their states during execution. As more and more hardware is created in a digital way the simulation and testing has become one of the main parts in the development process. The goal is to remove the bugs already in the design and implementation phase. As the hardware is only simulated at that time, it is much easier to find and to remove the faults there. Later, when the component is already built, one way to obtain values from it is by using measurement devices like oscilloscopes and multimeter. That makes debugging of course much more difficult. Another problem for hardware debugging is the simulation of the hardware itself. Many devices are needed to establish a correct simulated hardware component. For example emulators (see Section 2.2) and FPGAs have to be used to model all features of a real hardware, like parallel processing or the timing behaviour.

2.7.2. Software Debugging

Since the first software has been designed and implemented, designers had to struggle with finding and resolving bugs in their programs. One of the first techniques, and still used nowadays, to find the faults were *code reviews*. By looking at each line of code manually bugs can be found and also design errors can be detected. As this is a very time consuming work, nowadays various tools support the designers to do their jobs more efficient. The range of different tools is very wide and some of them have become very powerful. Besides standard debugging methods, like stepping through the code and setting breakpoints, the debugging tools allow the designers also to read, change or trace the values in the programs (see Section 2.7.4).

One important tool for testing software that is created for special processors is the instruction set simulator (ISS). With the help of an ISS it is possible to test a new program even if the hardware does not exist at this moment in time. The ISS simulates the behaviour of a processor at an instruction-set level and is therefore very important in an early design phase. Also the development of new ISS implementations is still an interesting topic in different research projects (see for example [24]).

2.7.3. Debugging Modes

Debugging tools can be divided into two groups, depending on their way of work: the stop-mode and the run-mode technique. The difference between them is the way they interact with the *device under test* (DUT).

The stop-mode, as the name implies already, stops the system at a specified point during execution. This is done for example with so called *breakpoints* (see Section 2.7.4). The stop-mode is suitable when the error is supposed to be in a certain part of the DUT. But if the system gets too complex or is not allowed to be suspended, for example when dealing with real-time applications, the second mode is more appropriate.

The run-mode is used when the DUT is not allowed to be stopped, for example because the system would be damaged if it was suspended at a wrong moment during execution. The debug data is created on-the-fly during runtime of the system. The easiest way is by using print statements to display the information on the screen. More advanced debugging tools collect the debug data, process and prepare them in a graphical way to the user. To be able to get the data from the DUT, additional software or hardware has to be added to it. This can be a problem, as every added debug functionality also affects the simulation (additional memory usage, different timing behavior) and therefore the results too. This problem is also known as the *Heisenberg Problem* [19].

2.7.4. Debugging Methods

This section shall give a short overview and a short description of the most commonly used software debugging methods:

Breakpoints: Special markers that are set onto lines in the source code to control the execution flow are called (*Breakpoints*). When the DUT reaches such breakpoint during execution, the complete system is suspended. This gives the designers the chance to have a closer look at the system, to read out values from variables or to set new ones. This can be necessary for example to force an error and test how the design behaves afterwards.

Watchpoints: There is only one difference to distinguish a *watchpoint* from a breakpoint. A watchpoint is not set onto a line of code but onto a variable. Every time that variable is read from or written to, the watchpoint triggers and forces the system into the debug mode.

Single step/step over: Stepping trough the lines of code, after a breakpoint was reached, is often used during debugging. Also to *step over* or *step into* functions is one of the traditional debugging techniques.

On chip debugging: Especially in embedded systems often on chip debugging is used to gather debug data from the system. The debug device is therefore integrated on the chip itself and measures and receives the needed information directly from the system.

This creates of course influences for the DUT, as described in Section 2.7.3. Another example for this kind of debugging method can be found in the paper of H.Yue li [21].

Tracing: An often used method is *tracing* [20]. When tracing a system, every time a function is entered or left, a message about this is stored in a file. The address of the function is stored and also any referenced data. This helps the designer to verify if the system is working correctly or not. With the help of the tracing methods it is also possible for the developers to find even more bugs than with traditional debugging methods. Especially for randomly appearing faults this technique is very promising. A more detailed description about the advantages of tracing, especially the tracing of hardware components, can be found in Chapter 7.

For the rest of this mater thesis those methods will be referred to as *debug methods*.

Each of these debug methods can be asigned to one of the debugging modes described in Section 2.7.3 before. While breakpoints, watchpoints and the step commands are traditional stop-mode techniques, the tracing of values and the on chip debugging modules are examples for run-mode debugging.

2.7.5. Debugging Embedded Systems

Embedded systems are a very good example for devices where hardware and software are developed together (see Chapter 2.3.1). This raises some new problems during the development and effects of course also the debugging of such systems. Many bugs appear only when the prototype of the new device is used together with its software. As it is very hard to gather debugging data from an embedded system after it has been built, ways to deal with it have to be found.

One way to obtain the desired data from the embedded system is by using the *Design For Debug* (DFD) [29] approach. This means that additional debugging support is added to the new embedded system in advance. It is clear that there will be some bugs in the system after the first prototype is established. The debugging support gives the developers the opportunity to get an insight view of the system. For example when developing a new microprocessor for a mobile phone, additional debug code is added to the chip during implementation. During the execution of the complete chip the developers are then able to force the chip into a certain debug mode. In this mode the chip is in a known state and it is possible to receive the debug information from it.

But there are also restrictions to this approach. Embedded systems are getting more and more complex and the number of debugging information inside them can reach easily some terabytes per second. This is of course way too much data to be transferred. The pins on the chip are limited in number and speed. Another problem is little size of the systems, so the additional features have to be as small as possible. For example if the ROM of the system has a fixed size, the software has to be implemented to fit into the available memory. This is often complicated and needs a special designed code, which can lead also to more errors too.

Another disadvantage, mentioned also in Section 2.7.3, is the Heisenberg problem [19]. Every measurement or additional function added to a system can have an impact on the results.

The simulation of a complete embedded systems in software is another approach for enabling the debugging. The problem with this idea is, apart from the high complexity of those systems, that hardware and software components are created on very different layer of implementation. SystemC and TLM 2.0 can be one alternative to combine both parts on a high abstraction level. More information about SystemC and TLM 2.0 can be found in the Chapter 2.4.

At the moment many debugging tools exist for either hardware or software (see Chapter 3), but there are tools missing that can debug both at the same time. The design and implementation of a new debugging environment, presented in this master thesis, shows up another possibility. By using different existing and well established IDEs and tools it is possible to extract the best parts of them and create a completely new debugging system (see Chapter 4).

3. Related Work

This chapter shall give an overview of different approaches towards debugging hardware, software and both parts together in a co-design. Especially in the last years new and innovative ideas have been established. Those projects use different approaches to do their debugging. For example by integrating new debug modules into existing design or by extending existing parts with new debugging functionality.

3.1. In-Circuit Emulator

The first project presented in this chapter is improving a well known and often used debugging tool: the *In-Circuit Emulator* (ICE) [8]. ICEs represent a copy of a microprocessor that has extra debugging support added to it. This means for example to be able to monitor internal values and to provide the debugging techniques described in Chapter 2.7.4. One problem with the ICEs is the fact that their implementations vary very much among different systems. Mostly an ICE is specially created for a certain microprocessor and its software when there is a need for a debugging functionality. It is very complicated to reuse an ICE in another system. So the goal for this scientific group was to design a retargetable and low-cost ICE on the *Register Transfer Level* (RTL). Therefore they implemented their new ICE in Verilog RTL and used the *JTAG Standard* 2.5 and expanded it for a debugging support. A diagram of the ICE can be seen in Figure 3.1 One of the main components in their design is the *Breakpoint Detection Unit* (BDU). This unit enables breakpoint support on the chip. It watches the address/data bus for certain values and is able to stop the microprocessor if a breakpoint or watchpoint is reached.

3.2. On-Chip debug module

Another idea to manage debugging on a microchip is presented in [21]. Nowadays the systems are getting more and more complex, which makes also the implementation of a traditional ICE much more complicated. Therefore many manufacturers try to integrate debugging support in advance on their chips as additional modules. These On-Chip debug modules mostly also use the JTAG pins, which were explained in the previous project, to send and receive the debugging data. The debugging module implemented in this project [21] is created for an 8051 microprocessor. In their project they used an 8051 compatible microprocessor model from the University of Shanghai to integrate the new module. The main components in their design are the microprocessor itself, with the On-Chip debugging module, the host

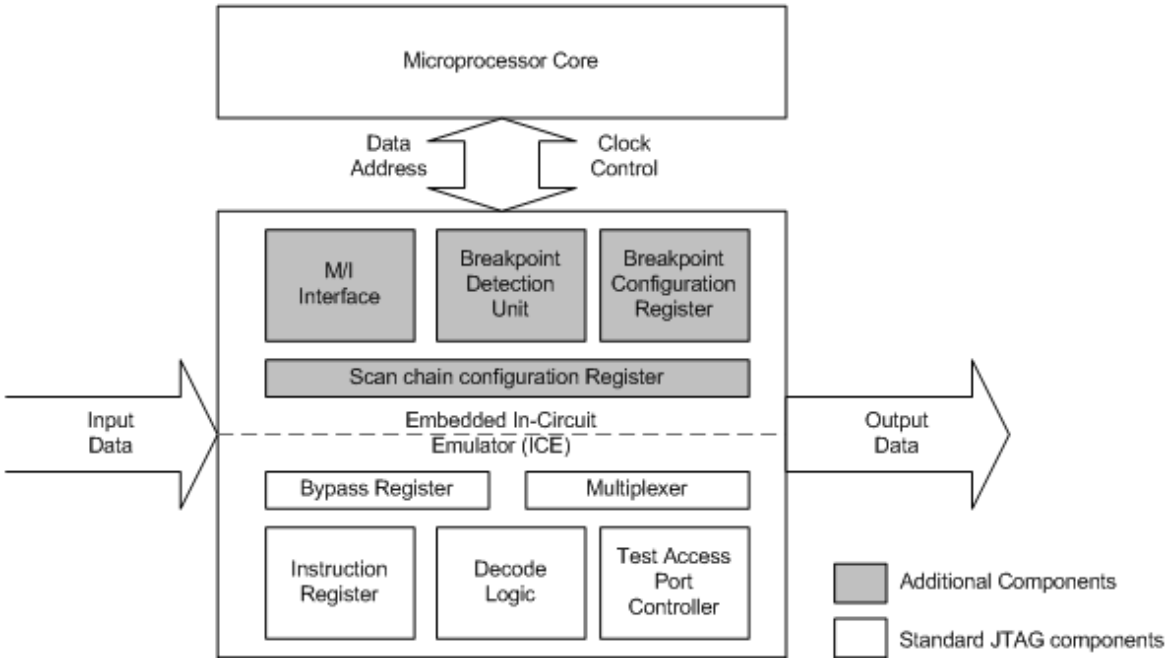

Figure 3.1.: In-circuit emulator based on JTAG

computer that is used for running the software to process the debugging data and a protocol converter to connect the other two parts. A picture of the system can be seen in Figure 3.2

The most interesting part in this design is the debugging module. It uses a communication sub module for interacting with the host computer and also implements the traditional debugging methods described in Chapter 2.7.4. As the Intel 8051 microprocessor type is also used to demonstrate the integration of a processor model into the new debugging environment their project is also from special interest for this master thesis.

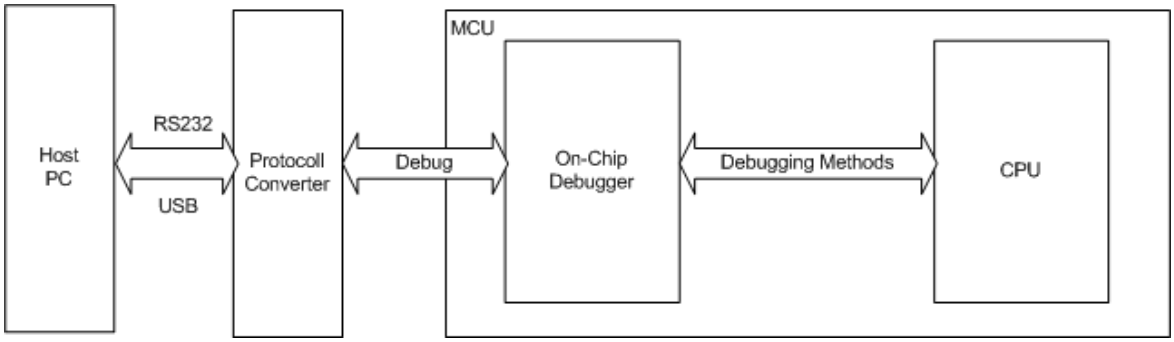

Figure 3.2.: On-Chip Debug Module and Environment

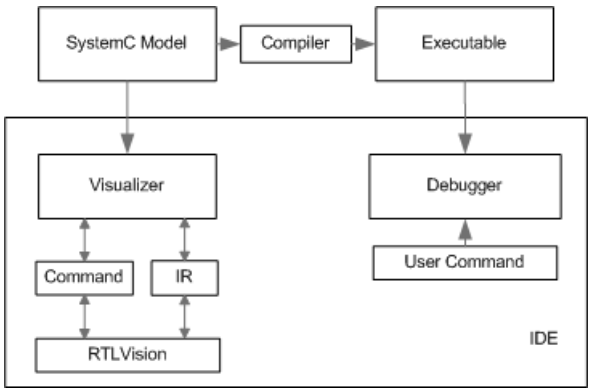

Figure 3.3.: Overview IDE

3.3. Integrated SystemC debugging environment

It is also possible to manage debugging when the design is implemented on a more abstract implementation layer than RTL, for example when using a system description language like SystemC? SystemC enables a cycle accurate simulation and hardware datatypes (see Section 2.4), but it is also possible, with the help of TLM (see Section 2.4.1), to implement parts of the model in an object-oriented way and on a functional level. This improves the simulation speed and helps to find design errors much earlier in the design flow. A problem with the SystemC standard is, that no debugging interface is defined for it and also no visual representation is supported. With implementing additional code in the model the users are able to access the values of the signals, but therefore the designers have to have a detailed knowledge about the system and its internals. To cope with this problem an integrated SystemC debugging environment was developed by a group from the University of Bremen [5] which is presented in Figure 3.3. Their system provides SystemC with debugging and visualization support on the functional and system level layer. Also the new parts do not influence the existing SystemC simulation kernel. The system is build upon the *GNU Debugger* (GDB)[1] and the visualization is done with the tool *RTLVision* from *Concept Engineering*[2]. The goals of their work can be summarized as:

- Use the OSCI SystemC kernel
- Enable high-level debugging
- Extend SystemC with the posibility of visualization
- Be non-intrusiveness to prevent alteration of the model, the SystemC kernel and libraries
- Create commands that implement a high-level-debugging interface

The SystemC model description is the starting point for the integrated debugging system. This system description is then compiled into an executable and used as input for the GDB

[1] http://www.gnu.org/software/gdb
[2] http://www.concept.de

debugger. At the same time the SystemC code is transmitted to the visualizer where it is analyzed into an *intermediate representation* (IR). The IR is used to render the visualization of the SystemC model. The debugging part can be split up into two parts: the functional debugging, finding errors in the implementation, is done using the GDB debugger. On the system level the debugging is done with the help of the visualization to find errors in this abstract layer. This includes for example faults in the communication between the modules.

3.4. Debugging using transactions

The abstraction level of a design is strongly correlated to its simulation speed. The more abstract the implementation of a system is chosen, the faster it can be simulated. Following Moor's law, the complexity of systems on chip increases every year and with it also the lines of code needed to control and interact with the systems grow with high rates. One way for the designers to cope with this trend is a high abstraction level for the design. This helps to isolate errors to certain parts of the application. A project where an environment was developed, that is able to handle debugging in such an abstract application, is presented in [4]. Using the transaction level for the design the system is broken down to read and write operations. To transfer these operations, transactions are created and a communication infrastructure is used to transfer them from one module to the other. The communication unit is also the target for this debugging approach. The idea is simple but effective. By including the communication system into the debugging control the designers are able to use debug techniques also for the transactions directly. For example the debug method of breakpoints can be established very easily. The transactions are monitored for certain values (source address, data values...) and, like a traditional breakpoint, the system is suspended when one of these values occurs. Then the communication control refuses any further transaction requests. This way all modules in the design reach an idle mode automatically, which allows to stop also the system clocks. For the implementation a *Network-On-Chip* (NOC) was chosen, as it is one of the promising communication infrastructures for large systems. The experimental results show, that the debugging approach is working for small SoC designs and that the transaction layer is a good level to be used for applying such debugging methods.

3.5. Simulation environment for hardware-software codesign

Another interesting project is presented in [15]. The project described in this paper was written shortly after the first hardware/software co-design flows were created. It is not dealing with debugging in general, but still it is from special interest for this master thesis. The project is dealing with the problem of simulating hardware and software parts in a co-design. The simulation is an important part in the co-design work flow as it shows the functionality of the design. In [15] the design flow is split into two parts (hardware and software) that are processed completely seperated from each other. Therefore their simulation environment also supports different specifications for both sides. The communication in the simulation system

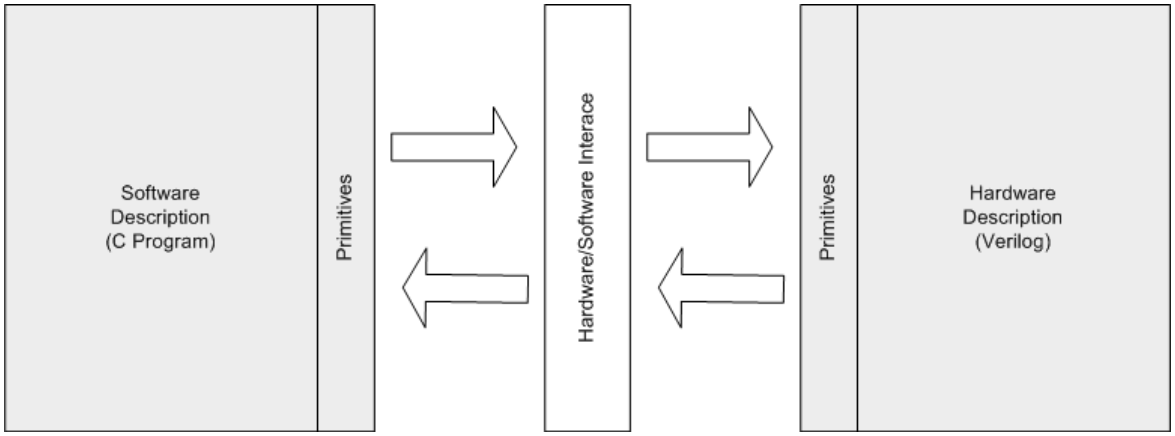

Figure 3.4.: Hardware/Software Simulation environment

is established using a messaging system. Thereby they communicate not directly with each other but via a hardware/software interface. This setup is illustrated in Figure 3.4:

The software part is implemented using the programming language C, while the hardware models are created as Veriolg Modules. The design of this project is similar to the new designed debugging environment created for this master thesis. Both IDEs seperate the hardware and software modules strictly from each other and commuicating via a messaging system. The project from [15] has its focus therefore on the simulation part.

The new framework presented in this master thesis offers a new approach for debugging hardware/software co-design systems. All projects that were presented in the related work had new and innovative ideas how to enable debugging in a hardware/software design. They want to assist the developers to do their debugging work faster and to shorten the developement time of new systems. Therefore new approaches are used or well known techniques are refined and improved. The idea for this master thesis was to extract the advantages from the different approaches and combine them in a new debugging environment.

The design of the new debugging framework presented in this master thesis has similar goals, but also presents new ways to deal with the debugging of hardware and software components. The goals of this new approach are:

- The design and implementation of a high-level debugging framework
- To establish a way to debug hardware and software components at the same time using SystemC modules and existing debugging IDEs

The new designed framework will therfore the following advantages for the developers:

- Developers will be able to find complex bugs that can only be found when hardware and software are used together
- Enable the debugging of hardware/software systems in a distributed environment

- Establishing a modular framework that allows to exchange and integrate components from different abstractional layers

A detailed description of the new debugging framework can be found in the following chapter.

4. Design

In the following sections the design of the new debugging framework will be presented. Therefore the framework will be surveyed from different views. This includes for example the physical view, logical view and development view. Also the different components that are building the new system will be explained. But before that the requirements for this master thesis are described in the section below.

4.1. Requirements

Before the design of the new debugging framework will be explained in details, the requirements for it are listed below. They describe the desired features of the new framework as well as the necessary programming languages which need to be supported. Another important part is the possibility to integrate other existing models into the framework. Therefore different open source projects have to be evaluated. The detailed requirements are:

1. Design a debugging framework that shall be able to debug hardware and software models at the same time
2. It shall be possible to use the debugging framework in a distributed environment
3. The framework shall be easily extensible with other models
4. Hardware models written in SystemC TLM 2.0 shall be supported
5. Software models written in C/C++/assembler shall be supported
6. Support common debugging methods like breakpoints and step commands
7. The integration process for other open source models shall be evaluated

The results of these requirements and how they were established can be found in Chapter 6.

4.2. Debugging Framework

In this section the design of the new debugging framework, that was created for this master thesis, is presented. An overview of the new framework can be seen in Figure 4.1.

The system can be split up into the following main components:

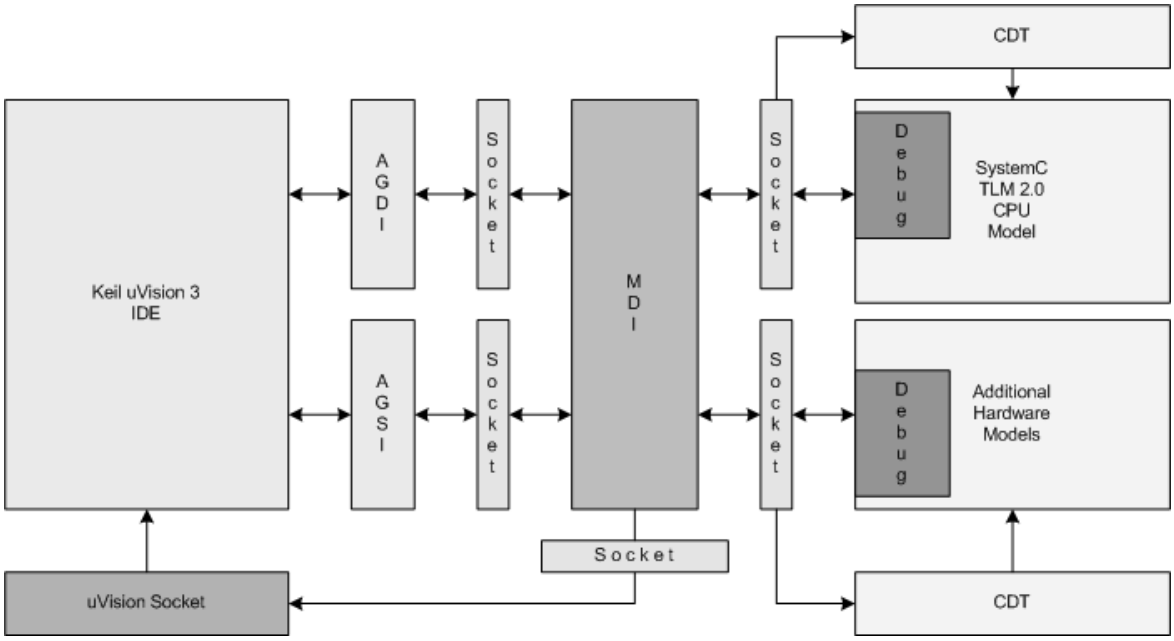

Figure 4.1.: Detailed overview of the debugging system

Keil μVision 3 IDE: The Keil μVision 3 IDE[1] is often used when developing software for hardware components (see Chapter 5.1.4). It combines compilers, debuggers and real-time kernels in one system, enables the simulation of models and supports the creation of integrated environments. Also many microprocessors are supported for simulations in this environment. Keil μVision 3 offers different interfaces to connect new devices to the IDE. For this master thesis the *Advanced Generic Debugger Interface (AGDI)* (see Chapter 5.2.1), *Advanced Generic Simulator Interface (AGSI)* (see Chapter 5.2.2) and the *μVision Socket* (see Chapter 5.2.3) are used. In the μVision IDE the software, that is implemented for the hardare components, is developed and executed. It is possible to use all traditional debugging methods (see Section 2.7.4) on that software modules. The μVision IDE also offers the possibility to watch the code in a disassebly window. In this view the code is displayed as a mixture of source code and the corresponding assembler commands. Another advantage of μVision is the support of instruction sets for different processor architectures. This makes it easier for the developers to implement software for a specific hardware model.

Model debug interface (MDI): This module establishes and manages the connection between the other components. It is the main part for the commuication in the new debugging system, holding client and server sockets for all the other modules. New data that arrives at one of the server sockets at the MDI is then routed to the right destination via the correct client. More about the implementation can be found in Chapter 5.3.2).

Microprocessor model: The microprocessor model is building the base component for the

[1] http://www.keil.com/uvision

simulation of the hardware. The new debugging system is not bound to one specific microprocessor type. So it is possible to use different CPUs as they can be integrated easily into the system. To show the integration of such a model see also Chapter 6, where an 8051 microprocessor model is used to show the necessary steps. That 8051 model was created beforehand this master thesis as a group project togehter with two other students.

C/C++ development tooling (CDT): CDT is a plugin for the Eclipse platform that enables C and C++ debugging in this IDE and therefore also the debugging of SystemC hardware modules[2]. Like the whole Eclipse IDE also the CDT is an open source project. During this master thesis the necessary functionality was added to it, which enables external applications to operate the CDT debug methods directly. Therefore it is also possible to control the debugging process of the hardware models from the outside. More about the implementation can be found in Section 5.3.3.

To connect the different parts of the debugging system the *windows sockets* are used (see Section 5.2.3). With clients and servers data can be transmitted from one component to the other. Another advantage of this approach is the possibility to run the applications in a distributed system over a network. So for example the expensive Keil μVision 3 IDE license can be used as a shared resource and also the hardware models can be simulated on different machines. This is an advantage for example when simulated models need a lot of performance and therefore it makes sense to execute them on different machines.

By using interfaces in the whole design of the debugging system, it is also possible to exchange parts with new components. For example a new processor model or new hardware components. But not only different types of models can be integrated, also models implemented using different programming languages or abstraction layer can be used. Therefore a *Wrapper* function has to be inserted between the interface and the new module, translating the interface functions into the new programming language and vice versa. So neither the debugging IDE nor the new model have to care about each others different implementation.

A more detailed view on the design of the main components is presented in the following subsections.

4.2.1. Eclipse C/C++ Development Tooling

One interesting topic when designing the new debug system was the question how to enable debug methods for the hardware models. Of course there are built in debugging methods in most IDEs, but the challenge was to find a way to control those methods also from an external program. In the first design the Microsoft Visual Studio IDE was used for that purpose, but it was soon clear that this was not the right IDE to use. The problem with Visual Studio was the lack of possibilities to extend its source code and add the required changes to call the debug methods directly. The design was changed to use the Eclipse IDE for the hardware models instead.

[2] http://www.eclipse.org

The CDT plugin manages and provides the debugging methods for C and C++ programs. This was also the starting point for the implementation of the additional debugging control for the hardware modules.

The changes that are necessary to extend the CDT for the new debug system are described in Section 5.3.3.

4.2.2. Hardware Debug Module

One of the main tasks for this master thesis was the design of a debug module for hardware models: the *Hardware Debug Module (HDM)*. The HDM enables the model to react on the requests of external debugging IDEs, working like an In-Circuit debugging module. It is able to

1. Gather data from the model and transfer it to the external IDE
2. Change values in internals of the model, for example the registers or the RAM
3. React other debug requests: setting/removing breakpoints and resetting the module

In this compact form it is easier to access the values for a stored debug tasks. The implementation of the *Hardware Debug Module* is described in detail in the Section 5.3.4.

4.3. Design Views

The following chapters will describe the design of the debugging system in a detailed way by using different views. Looking at the system from different sides helps to present the idea and to demonstrate its usage. The available views are:

- Physical View
- Development View
- Logical View
- Process View

The *Physical View* is used to present the physical characteristics of the debugging environment. It displays the distributed nature of the framework and explains what models and applications belong to those separated parts. Also the connection between the physical components is described.

The second view is called *Development View*. It is used to present the software applications in the debugging environment and to show the interaction points that are offered to the users. With these interfaces the users can control the execution of the applications. Also the purpose of the applications is described in this chapter.

The next view presents the logical description of the design: The *Logical View*. This section describes the user interfaces in more details and also explains the functionality of the system

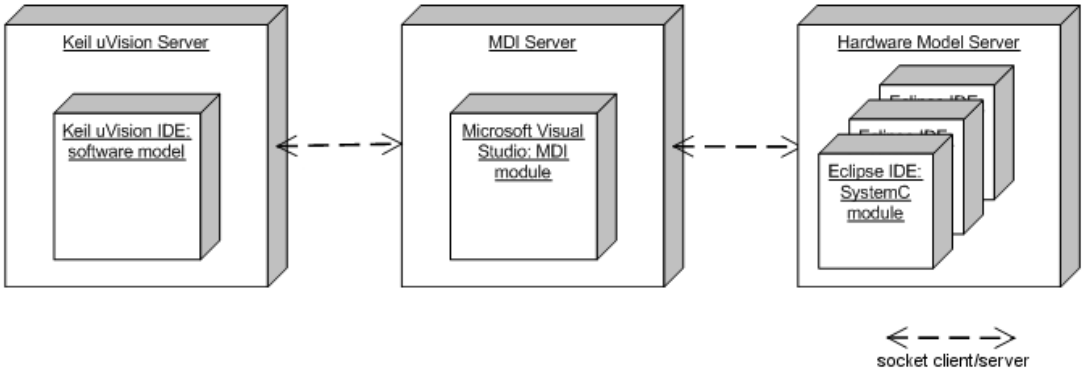

Figure 4.2.: Physical components of the debugging system

that is offered to the users. Therefore use cases are presented to demonstrate the systems reaction on an interaction of an user. After that, timing sequences are used to present the execution of a debug task in details. This is necessary to show the timing relations between the components in the debugging system.

The final section presents the *Process View* . This view is used to describe the communication between the processes in the system. This includes the setup of the communication infrastructure and the reaction of the modules on incomming debug tasks. This view is important to show the behaviour, synchron or asynchron, of the components when dealing with the different kinds of debug tasks.

4.3.1. Physical View

The new debugging system is designed to work also in distributed environments. Figure 4.2 shows the physical blocks of the design.

It is possible to run every block on a seperate computer: One PC for developing and debugging the software components. Therefore the Keil μVision 3 IDE is used. The second machine contains the communication server for the MDI. The last server is the hardware model server, which is used to run and debug the different models of the hardware components. This includes first of all the model of the CPU, but also all additional hardware models that are integrated into the system. In this design the hardware models are created and executed with the Eclipse IDE while the CDT plugin is used to debug them.

The connection between the servers and components in the design is established using windows sockets. In the Keil μVision 3 environment the AGDI, AGSI and the μVision Socket interfaces are used to access the socket clients and servers. The users can interact with the Keil and Eclipse side via the corresponding IDEs.

The separation of the three components (Keil μVision 3 IDE, MDI, Eclipse System) is necessary as it can happen during a debug session that either the Keil μVision 3 or the Eclipse IDE side is suspended completely. The MDI module is always running, so it can manage the restart or reactivation of the other subsystems if that is necessary. Another reason for splitting the debugging environment into three parts is the different IDEs that are used. Every part uses its own environment:

- Keil μVision 3 IDE
- MDI using Microsoft Visual Studio[3]
- Eclipse CDT

This is necessary as also the different components, hardware and software, need different environments to be simulated. For example when dealing with hardware components, their simulation depends on the used hardware description language and the instruction set that is used.

More about the implementation of those parts can be found in Chapter 5.

4.3.2. Development View

As mentioned in the section before, the user has two possibilities to interact with the debugging system. This is also displayed in Figure 4.3. The access point for the users are the two IDEs, the Keil μVision 3 IDE and the Eclipse framework with the CDT plugin. The users can therefore access the models with the user interfaces they are already familiar with and there is no need to learn how to handle a new *Graphical User Interface* (GUI) component.

The Keil μVision 3 IDE represents the software generating part of the design. Using the μVision IDE the user can create the software that is later executed on the hardware modules. The environment supports the implementation of software for hardware components, for example by providing instruction set compatible registers that can be accessed like variables. It is also possible to watch the software code in a *Disassembly* view. In this view the single instructions executed by the processor can be seen.

The hardware models are created with the help of the Eclipse environment. As Figure 4.2 shows, the design of the new debugging system is able to work with different hardware modules at the same time. The modules can be created in any HDL and it is also possible to use models from different abstraction layers, for example RTL modules. As long as the interfaces to the MDI and the CPU debug module (see Chapter 5.3.4) are implemented correctly the system will be able to use the new components.

The integration process of a new model is explained in details in the Chapter 6 and more information about the interactions between the parts and with the system can be found in Chapter 4.3.3.

[3] http://msdn.microsoft.com/en-us/vstudio/default.aspx

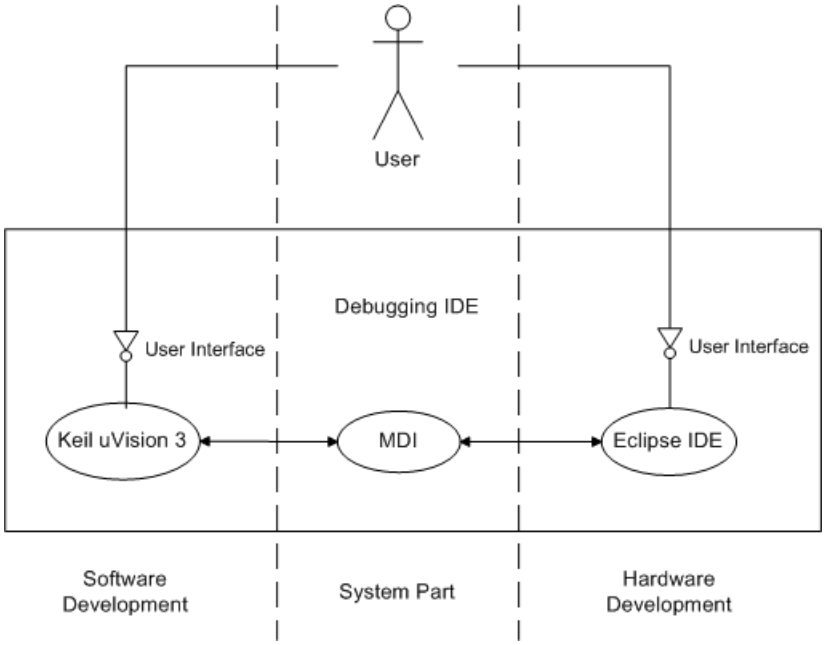

Figure 4.3.: Component diagram of the debugging system

4.3.3. Logical View

To understand a system it is important to know how the different parts of environment interact with each other and what functionality is offered to the users. In this debug environment the user has two possible access points to operate the system. The first one is the Keil μVision 3 tool set and the second one is the Eclipse IDE with the CDT plugin. In Figure 4.4 and Figure 4.5 two usecases for starting a *debugging task* are displayed. This task can be any kind of debugging activity, for example stepping through the code or setting a breakpoint.

The two uses cases explain the dataflow in the design of the debugging environment. In Figure 4.4 the first data flow, from the Keil μVision 3 toolset towards the Eclipse IDE and the hardware modules that are running there, is presented. If a user activates a debugging task in the Keil μVision 3 IDE it is transferred via the interfaces AGDI or AGSI to the MDI module. From there the debug commands and data are transmitted to the right destination. This can either be the Eclipse CDT plugin, the microprocessor model or one of the additional hardware modules.

The other use case works similar and is explained in Figure 4.5. A debug task is send from the Eclipse IDE to the Keil μVision 3 environment when a user debugs the hardware models with the CDT plugin from Eclipse. The CDT sends the data to the MDI module, which then forwards the request to the μVision socket interface.

A debug task of course not only occurs when a user is involved. Also the componentes themselves (Keil μVision 3 IDE, hardware modules) can create such a task on their own. For example if they need to send or receive certain information to update tbe values in themselves

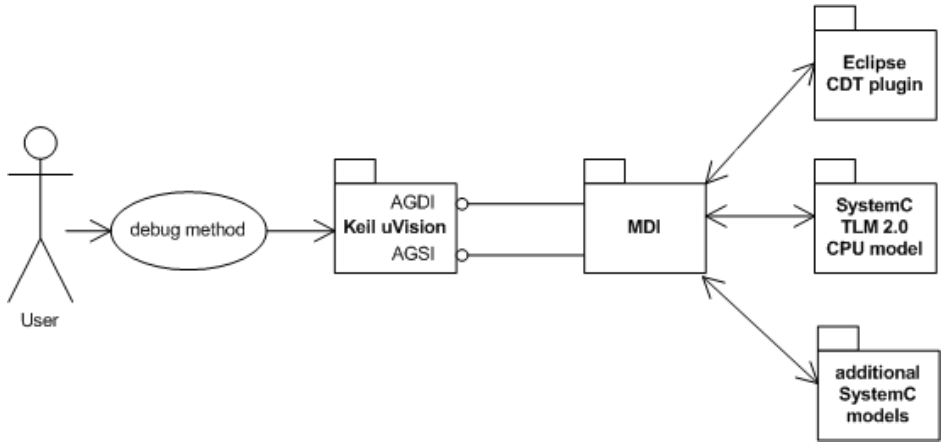

Figure 4.4.: Usecase1 - Debugging task from Keil μVision 3 to Eclipse

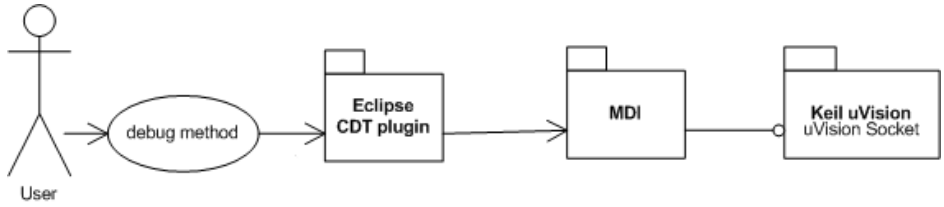

Figure 4.5.: Usecase2 - Debugging task from Eclipse to Keil μVision 3

or in another modules. This happens often during execution to keep the values in the tools up-to-date and to display the right values to the user.

Figure 4.6 and Figure 4.7 present a more detailed view on the necessary interaction steps during a transmission of a debug task:

These timing diagrams are examples to demonstrate the different types of interaction sequences. There are different debugging task in the debugging environment but they can be separated into three categories:

- Read debug tasks
- Write debug tasks
- Command debug tasks

Read and write debug tasks focus on the data transfer in the framework. They are used for updating and modifying data in the different models and also for receiving certain values for further processing. The command tasks on the other hand are used for different debugging techniques. It can be for example a command to suspend a certain model or to start its execution again. Also debug methods like the single-step or the step-over methods are part of the command debug tasks.

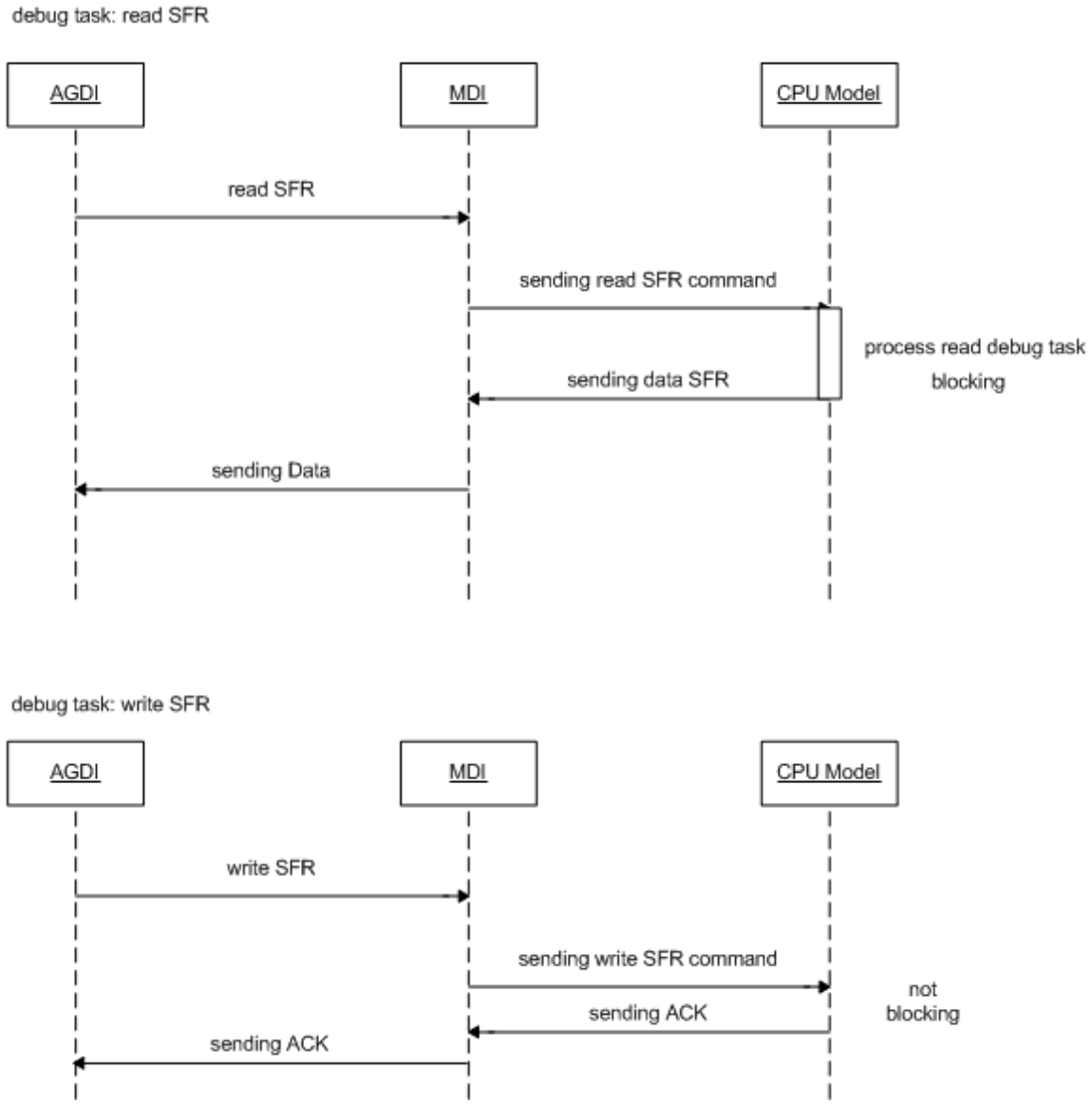

Figure 4.6.: Timing sequence for read and write debug tasks

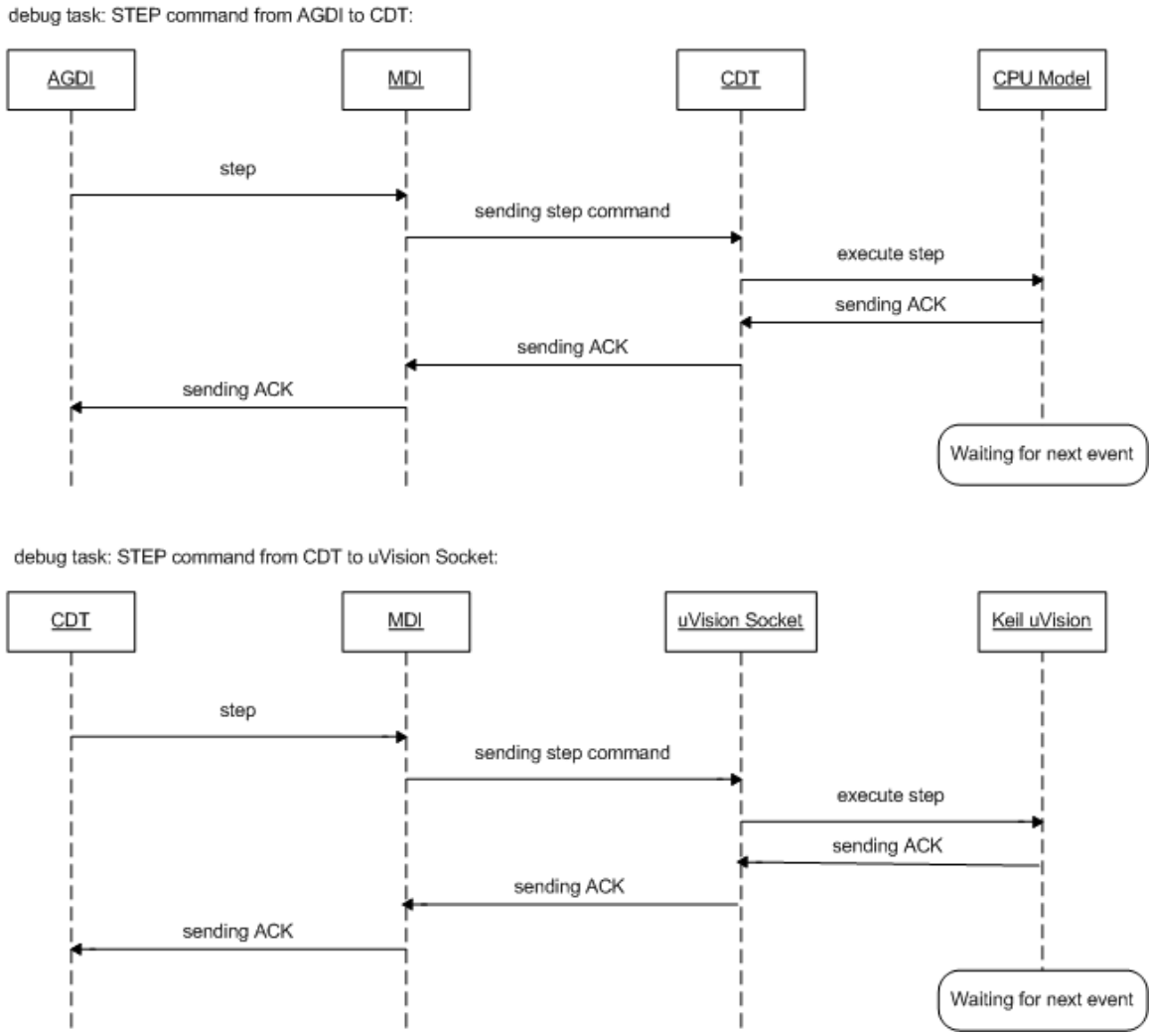

Figure 4.7.: Timing sequence for command debug tasks

A read and a write debug task can be seen in Figure 4.6. In the example the usecase of accessing a register in the microprocessor is presented. There is a big difference between a write and a read debug task. The write debug tasks handles the usecase in a straight forward and non-blocking way. The debug task is sent to the MDI module which forwards it to the right component. In this usecase the destination is the microprocessor model. The write debug task is then stored into a list in the microprocessor module and processed later. If everything worked fine and the transmission was successfully, a confirmation message (ACK) is sent back.

A read debug task is working differently. The transmission of the debug task to the destination component is done the same way as explained in the write debug task example. But instead of sending only an ACK message back, the read debug tasks requests more information. Therefore the task is blocked and has to wait while the microprocessor module is executing the request and preparing the desired data. When the data is ready the data is sent back to the Keil μVision 3 IDE via the MDI module.

The reason for this different behaviour of a write and a read debug task is a performance improvement. Many of those tasks occur during a debugging session. When a write debug task is created in a component, the main goal is to update a value in another part of the debugging system. Therefore it is not necessary for the component which created the debug task to be blocked till the task is processed. As soon as the write debug task reaches its destination and the ACK message returns, the component can continue its normal execution. No more information is needed and the values in the destination will be updated as soon as the stored request is processed. This is not possible for a read debug task. When a component creates a read debug task it also needs to obtain a certain information from another part of the debugging environment. So it is blocked and has to wait till the read debug task is processed by the destination module and the response data is received. The requested values are needed and can have a direct impact on the execution, therefore it is necessary to handle read debug request in a blocking way.

The command debug task behaves the same way as a write debug task and can be seen in Figure 4.7. The only difference is that it is not sent to the 8051 TLM model directly but to the CDT plugin, as this plugin controls the debugging methods of the Eclipse environment. In the other direction the debug task is created at the CDT and is then transferred to the μVision Socket which controls the Keil μVision 3 IDE. Like the write debug task also the command debug tasks can be processed in a non-blocking way and only stop the execution till the verification ACK message is returned.

4.3.4. Process View

As the communication is a crucial part in the debugging environment, it is very important to make clear how the different parts of the system interact with each other. For the communication the most important component in this design is the MDI module. It handles all incoming debug task requests, either from the Keil μVision 3 or the Eclipse IDE side. If a new debug tasks is established in the framework the procedure is always the same: The module in which the task was created uses its client socket to send a debug request to the

next server socket. Every new communication between two different modules is always started like that, with the client socket as the initiator. As described in the chapter before there can arrive different debug tasks. In Figure 4.8 the activity diagram of the MDI for handling debug tasks is presented.

Most of the debug tasks that occur during a debugging session are read or write requests. The MDI receives the necessary data from the AGDI, AGSI interface or the SystemC modules. As different tasks require different data to be sent or received, the MDI gathers the desired data and waits till it received the complete information. Then it sends the data to the correct target component. When dealing with a read debug task the MDI has to wait also for the answer from the target module. Then the MDI receives the response data and, if the data is complete, transfer it back to the module that created the initial read debug task. The write and command debug tasks are just forwarded to the right destination and if the transmission was successfully an ACK verification message is sent back to the initiator target. This different handling for the debug tasks is necessary because of their blocking or non-blocking behavior, which is explained in Chapter 4.3.3.

The next figure shows the activity diagram of another important component in the debugging environment, the *Debug Module* that is located in the hardware models (see Figure 4.9).

When a new debug task arrives, the *Hardware Debug Module* (HDM) behaves similar to the MDI. The incoming debug task is identified and the corresponding data is received. After that a *Debug Element* is created. This class is used to store the debug task data in a compact way that makes it easier to process it later. Therefore debug tasks are also saved into separated lists. One for storing the command and write debug tasks and one list for holding the read debug task. This is necessary due to the non-blocking and blocking behavior of the different types of debug tasks. For a write debug task, as soon as the complete data is received, an ACK message is sent back to the MDI module. For a read debug task the necessary response data has to be prepared first and if it is ready, it is sent back also to the MDI. A more detailed view on the execution process of those debug tasks can be seen in Figure 4.10.

The debug module checks if a new write debug task is available in the stored write debug task list. If there is a request stored this debug task is processed. After the task is finished and the data has been written, the debug module continues searching for write debug tasks till all of them have been executed completely. Then the same procedure is done for the stored read debug task, if there is one available. When the read debug task is processed completely, the created data is transmitted to the MDI module, as described before.

As mentioned at the beginning of this section, the communication system builds the main component in the new design. This fact is also used to establish another important method for debugging: Breakpoints. This debugging methods is explained in details in Section 2.7.4. When a breakpoint is set in a hardware/software co-design framework it is necessary to guarantee that the complete system is stopped at the same time. Therefore it is possible for the developers to compare the data from the different componentes with each other as they describe the same state. To establish this goal the communication system of the framework is inlcuded into the debugging process. When a new breakpoint is reached this information is also passed to the MDI module which immediately forwards a stop debug task to all other

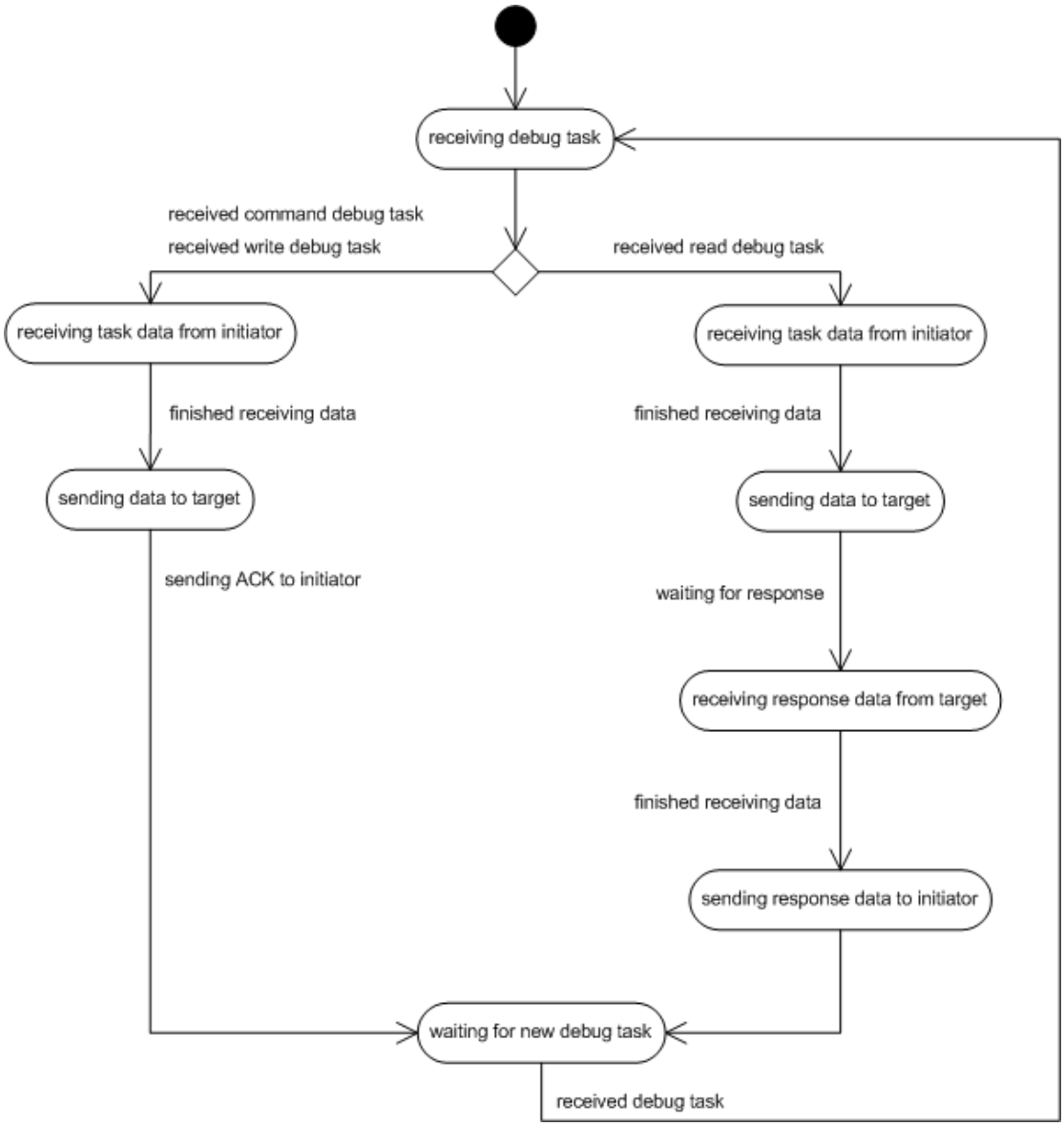

Figure 4.8.: Incoming read and write tasks in the MDI module

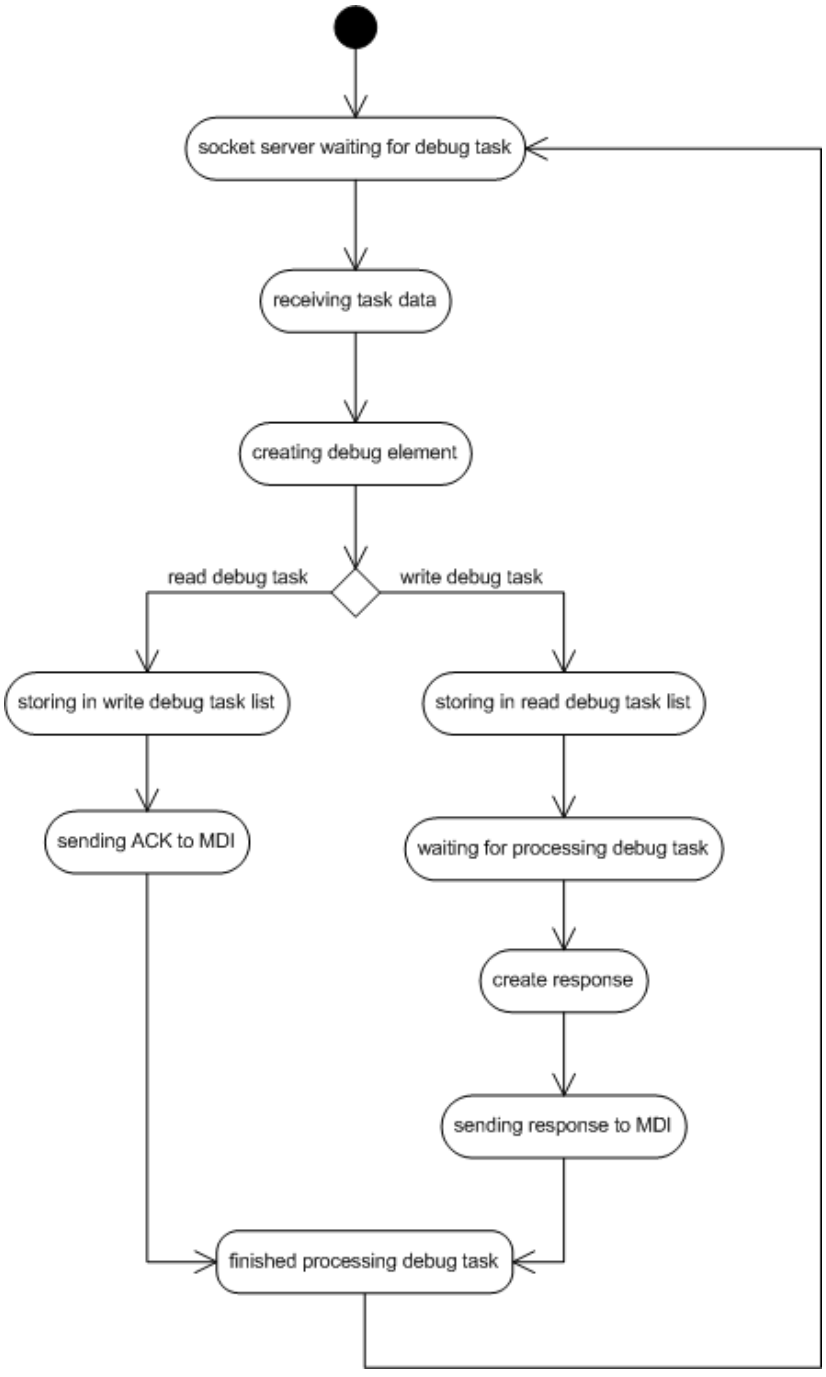

Figure 4.9.: Incoming debug task at 8051 TLM 2.0 model

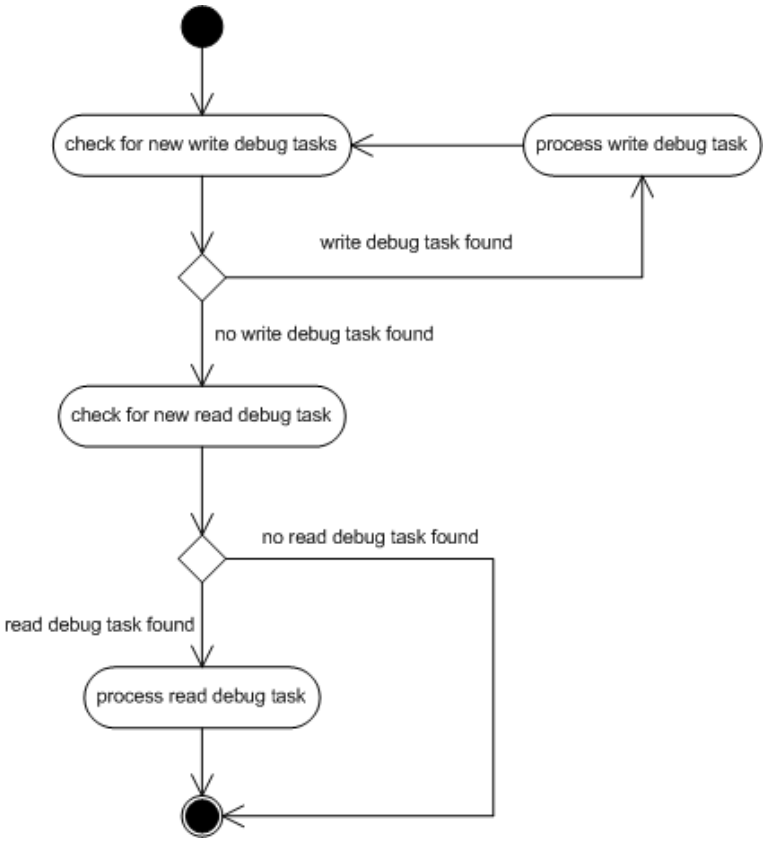

Figure 4.10.: Processing of debug tasks in the controller

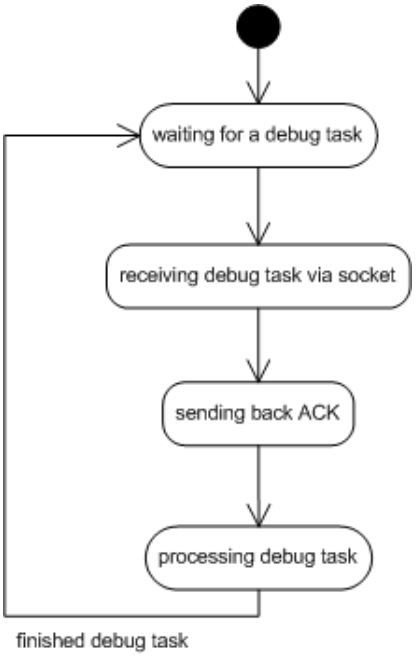

Figure 4.11.: Processing of debug tasks in CDT and μVision socket

components. For the Keil μVision 3 environment this is done via the Keil μVision 3 socket interface, stopping the whole debugger with a command. On the SystemC side the debug module takes care of the suspending the component. Therefore it prevents the execution of the next instruction. Apart from that also no new debug requests are accepted from the MDI module till the execution is continued.

The other components, like the CDT and the μVision Socket, handle incoming debug tasks like explained in Figure 4.11. Those requests are always command debug tasks that are used to operate the debugging functions in the corresponding environment (Keil μVision 3 and Eclipse). As soon as an command task is received completely an ACK verification message is sent back to the MDI module to signal that the transmission was succesfully. The debug task is then identified by the module and then processed. After that the module waits for the next incoming task on its server socket.

5. Implementation

This chapter will explain the components of the implemented debugging environment in more details and how they interact with each other. In Chapter 5.1 the underlying environment IDEs are presented. After that, in Chapter 5.2, the important interfaces that enable the interaction in the systems are explained. In the final Section 5.3 the internals of the system modules are presented. This includes the debug module in the microprocessor as well as the CDT plugin and the *Model Debugging Interface*.

5.1. Environment

In this section the used environment tools and IDEs are explained. This part is important because it shows the wide range of different applications that have to work together to build the new debugging environment. Those IDEs are created by three different companies. The reasons why exactly those applications have been chosen is explained in the following subsections.

Figure 5.1 shows the three IDEs that are part of the debugging framework. Each of them is used for a special context. Their advantages and the reasons why they have been chosen are explained in the following sections. The Keil μVision 3 IDE (see Chapter 5.1.4) is containing the software models, the Eclipse CDT environment (see Chapter 5.1.2) is used for developing the hardware models. More details about these two IDEs can be found in the following sections. The third development environment that is used is the Microsoft Visual Studio 2008 framework. It contains all the communication parts and is explained in the following subsection.

5.1.1. Visual Studio 2008

Visual Studio 2008 Professional Edition (VS) from Microsoft[1] is one of the most popular IDEs for software development on Windows operating systems. It supports different programming languages. For this master thesis the C++ development environment of Visual Studio was needed. Although that the documentation provided by Keil for the AGDI/AGSI interface and the μVision socket is very limited (see also Chapter 5.2.2, 5.2.1 and 5.2.3) there are some example projects available that explain the interfaces and therefore can be used as a basis for a new implementation. Those examples are available as Visual Studio projects so it was decided in the designing phase that the Microsoft IDE would become a part of the

[1] http://www.microsoft.com/visualstudio/en-us/

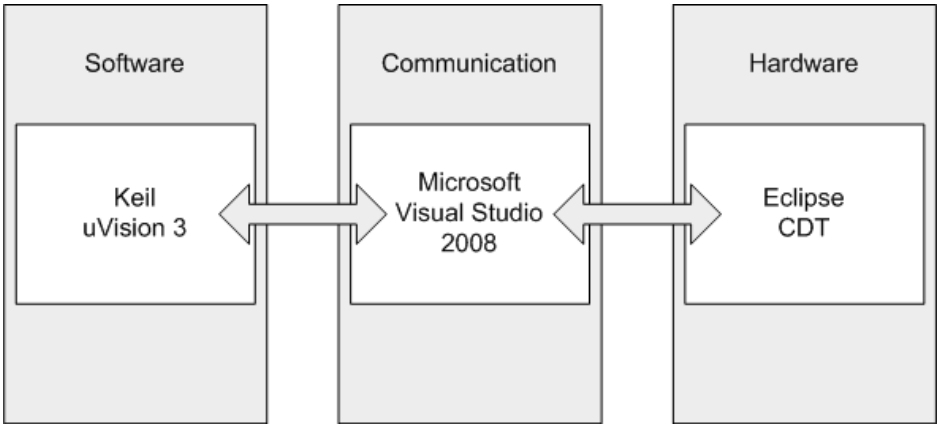

Figure 5.1.: Overview of the different IDEs

new debugging environment. Therefore it was possible to extend the existing example and implement the new communication structure around them. The 8051 TLM 2.0 model, that is used to demonstrate the integration of a new CPU model in Chapter 6, was developed using the Microsoft IDE. As VS has many advantages, for example the easy to use user interface and many of debugging functions and tools to assist the developers, it has also a big disadvantage. During the design phase it was soon clear that VS does not provide the necessary possibilities (for example via interfaces or open source code) to access and modify the debugging methods of Visual Studio. Therefore this IDE is not suitable for running the hardware models, but still VS is used in some other parts of the new debug environment: The projects for implementing the AGDI and AGSI interfaces as well as the MDI communication module.

5.1.2. Eclipse/CDT

Similar to the Visual Studio IDE from Microsoft, presented in the section before, is the Eclipse[2] environment. This IDE is a very popular toolset for software developers. The development framework is written in Java and supports also many different programming languages (C, C++, Perl,...). For this master thesis the C++ package of Eclipse was chosen, which is called *Eclipse CDT*. The CDT is the plugin that supports the Eclipse IDE with the debugging capabilities for C++ programs. The CDT plugin offers all traditional debugging methods and acts as a frontend to the GNU Project Debugger (GDB) (see Chapter 5.1.3).

The Eclipse IDE and its plugins are all open source projects, which is a big advantage in comparison to the Visual Studio IDE. Due to the availability of the source code it is possible to adjust the debugging methods directly. This means that it is possible to add the necessary code that allows an external application to access those functions from the outside. That was also the reason why the Eclipse framework was chosen for the microprocessor model, the additional hardware modules and the CDT plugin.

[2] http://www.eclipse.org

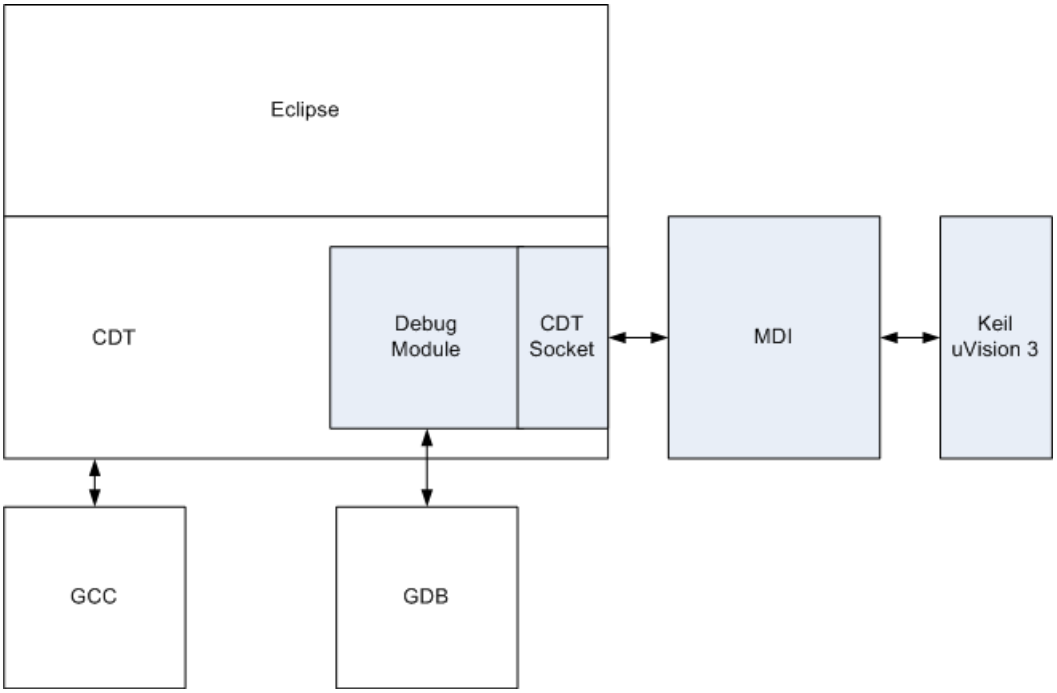

Figure 5.2.: Overview of the Eclipse environment and the additional modules

Also the big support community of the Eclipse tools is a big plus for this framework, offering a lot of insight information and solutions for problems and errors.

In Figure 5.2 an overview of the Eclipse and the necessary modules for this environment are presented. It can be seen that the Eclipse IDE builds the main part of the debugging framework. Also the additional modules and plugins are displayed, as well as the parts that were implemented during this master thesis. The GCC is used for compiling the source code while the CDT plugin enables Eclipse to debug it afterwards. The new implemented modules are located in the CDT plugin. The socket is used to establish a connection to the MDI module (see Chapter 5.3.2) for details about this module). The received debug tasks are then processed by the debug module located in the CDT, which then calls the corresponding debugging methods in the GDB. Like this it is possible to control the debugging process of a hardware model from the outside. The results of the executed debugging method, for example a single step, are displayed in the Eclipse window debugger window.

Many open source SystemC projects are implemented using the Eclipse IDE. So this is another advantage of this IDE and a reason why the Eclipse environment was chosen also for the debugging system presented in this master thesis. Using existing project files makes it easier to use the open source systems because the setup can be done in a much faster way.

5.1.3. GNU Project Debugger (GDB)

The GNU Project Debugger (GDB)[3] is a software debugger that was developed in the end of the 1980s [16] by the GNU Project (GNU is not Unix Project). It can be used for different programing languages (C, C++, Pascal, Java) and is also available for many different operating systems. Important for this master thesis are the interfaces that enable users to interact with the GDB remotely. The debugger offers two interfaces for that purpose: the CLI (Command Line Interface) and the MI (Machine Interface). While the CLI uses the command line to read in debug commands and display the results, the MI is implemented to interact with external applications. The MDI interface is also used by the CDT plugin (see Chapter:5.3.3) to connect the GDB to Eclipse and to enable the IDE to debug C and C++ programs. Using the interface it is possible to control the debug functions of the GDB directly, which is a necessary feature for the debugging system created in this master thesis. The GDB also offers the possibility to be used in a *remote mode*. This means that the GDB is running on a host computer and other programs can connect to it via a serial or a TCP/IP connection.

5.1.4. Keil μVision 3 IDE

The last IDE that is used in the new debugging environment is the Keil μVision 3 framework[4]. It can be used for software editing, to debug programs with traditional debugging methods and to run a complete instruction set simulation. Also many different processor types are supported for those simulations. There exist tools to measure specific characterstics during the simulations, for example to analys the performance. As this is all packed into one toolset, Keil μVision 3 is a very powerful IDE. One fact that was very important for this master thesis, is the possibility to receive the debugging information from the Keil μVision 3 framework and also to operate its debugging methods from the outside. For the first issue Keil μVision 3 offers two interface: The AGDI and the AGSI interfaces, which are explained in the Chapters 5.2.1 and 5.2.2 in detail. The control of the debugging methods of μVision is achieved with the help of the μVision Socket interface (see Chapter 5.2.3).

In Figure 5.3 an overview of the Keil μVision 3 framework and its components can be seen. The important parts of μVision that play a role for this mastethesis are the three interfaces that are provided by this IDE. The AGDI, the AGSI and the μVision Socket interface. They are used to connect the Keil μVision 3 IDE with external components and remote control the IDE itself (see Section 5.2). In the new debugging framework, illustrated in Figure 5.3, all interfaces of the μVision are connected directly with the MDI module. The MDI manages then the transmittion of the debug tasks from and to the Keil interfaces.

[3] http://www.gnu.org/software/gdb
[4] http://www.keil.com/uvision/

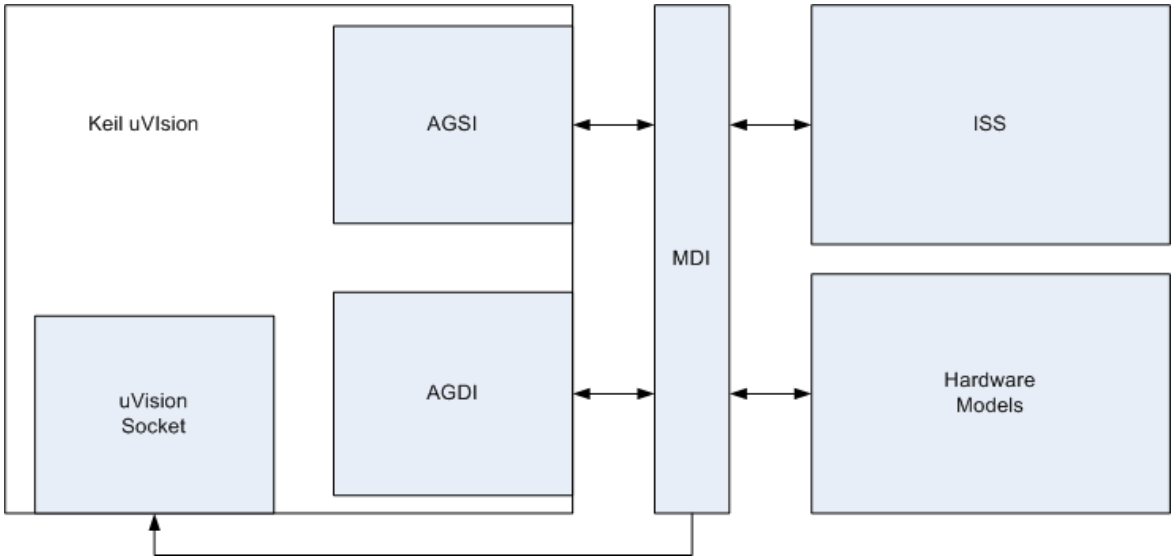

Figure 5.3.: Overview of the Keil μVision 3 framework

5.2. Interfaces

In this section the interfaces used in the debugging environment are presented. Only with the help of those interfaces it is possible to interact with the different IDEs. For this master thesis the three interfaces from the Keil μVision 3 IDE are especially important: The AGDI, the AGSI and the μVision Socket. An overview of the different interfaces and how they are linked in the system can be seen in Figure 5.4.

Each interface has its own special purpose. The AGDI interface is used to connect the Keil μVision 3 framework to a hardware processor model (see Chapter 5.2.1) which is used for the instruction set simulation. As described in the results of this master thesis (see Chapter 6) a SystemC 8051 TLM 2.0 microprocessor was used to demonstrate the integration of such a processor model.

The second main interface that is available in the Keil μVision 3 environment is the AGSI interface (see Chapter 5.2.2). This interface enables the integration of additional hardware models into the debugging system. This can be a completely new component, for example a analog-to-digital converter, or an additional unit for the processor module, like a secondary timer or UART unit.

The last interface described here is the μVision Socket (see Chapter 5.2.3). This interface works the other way around, making it possible to control the Keil μVision 3 environment and its debugging methods from an external application.

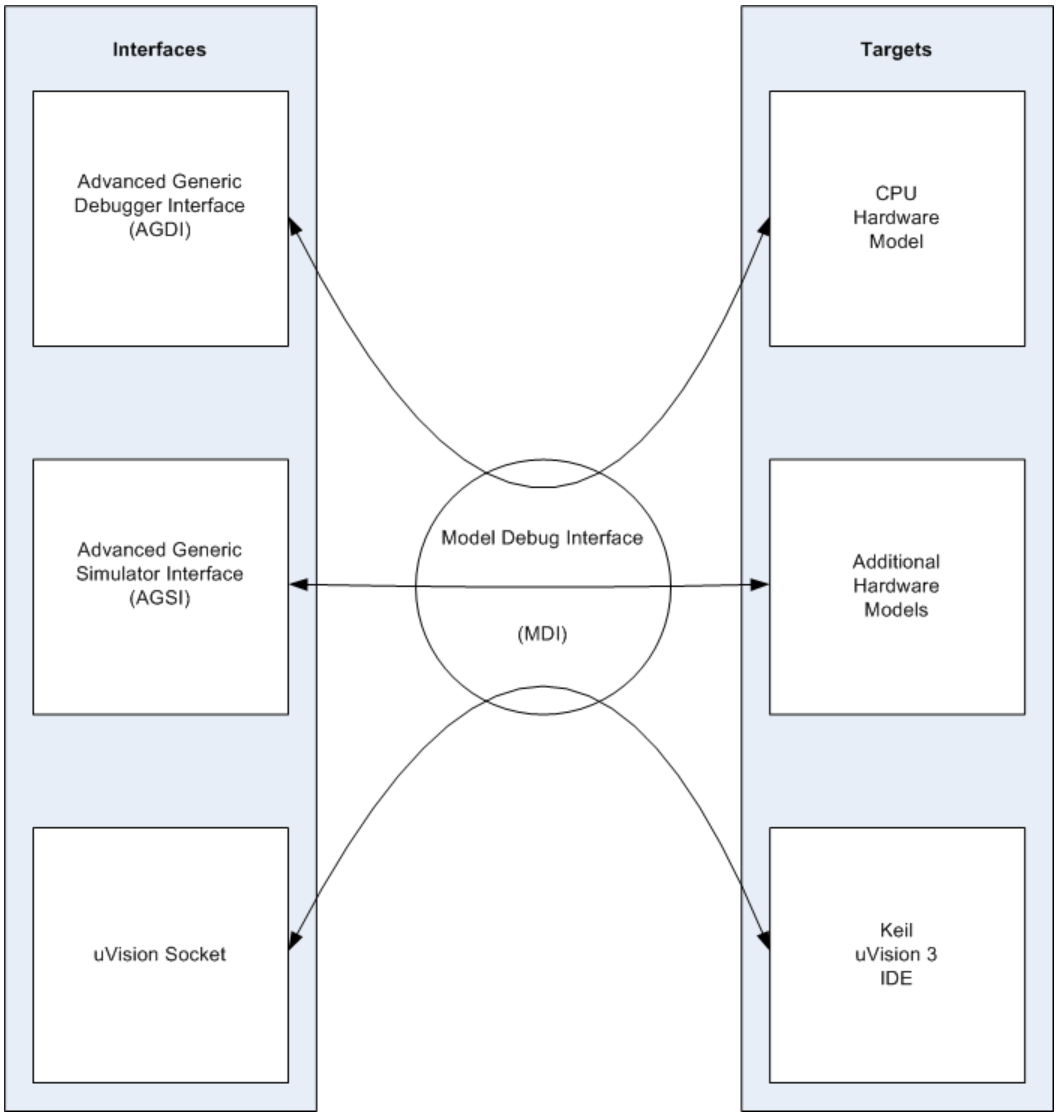

Figure 5.4.: Overview of the interfaces in the debugging environment

5.2.1. Advanced Generic Debugger Interface

The first interface that is discussed here is the *Advanced Generic Debugger Interface*, or in short form: the AGDI interface [27]. This interface allows developers to add their own hardware emulators and monitors directly to the Keil μVision 3 framework. The AGDI interface offers all traditional debugging methods for the external models and can also be adjusted with hardware specific commands. As the interface is independent of the hardware architecture of the integrated hardware models it is also very flexible. Building up onto the available sample DLL project in Visual Studio, the interface provides the function bodies to connect to and to work with an external application.

A description of the most important functions can be found below [27]. A complete list of all AGDI interfaces is presented in the appendix of this master thesis (see Appendix A.1).

- `UL32 ReadData (BYTE *pB, DWORD nAdr, DWORD nMany)`: reading data bytes from the microprocessor model
- `UL32 ReadSFR (BYTE *pB, DWORD nAdr, DWORD nMany)`: reading SFR byte from the microprocessor model
- `UL32 WriteData (BYTE *pB, DWORD nAdr, DWORD nMany)`: writing Data bytes to the microprocessor model
- `UL32 WriteSFR (BYTE *pB, DWORD nAdr, DWORD nMany)`: writing SFR byte to the microprocessor model
- `UL32 ReadPC (void)`: read the PC from the microprocessor model
- `void WritePC (UL32 nPC)`: write the PC to the microprocessor model
- `void GetRegs (void)`: read all registers from the microprocessor model
- `void SetRegs (RG51 *pR)`: write new values to all registers in the microprocessor model
- `UL32 Step ()`: execute a step command
- `void GoCmd ()`: execute a go command
- `int SetClrBp (int set, AG\_BP *pB)`: set/clear a breakpoint

Also additional helper functions have been implemented to establish the connection to the MDI module (server and client) and to combine similar tasks into one function to keep the design clear, for example the readData() and writeData() functions.

- `void readData (char comId[1], BYTE *pB, DWORD nAdr, DWORD nMany)`: used for reading data from the microprocessor model
- `void writeData (char comId[1], BYTE *pB, DWORD nAdr, DWORD nMany)`: used for writing data to the microprocessor model

- `void createServerToMdi (void)`: starting the creation of the server to the MDI module
- `void createClientToMdi (void)`: starting the creation of the client to the MDI module

5.2.2. Advanced Generic Simulator Interface

The μVision 3 environment provides another interface for developers to integrate external peripherals into the Keil μVision 3 framework: The *Advanced Generic Simulator Interface* (AGSI) [26]. AGSI therefore offers functions for simulating the behaviour and also methods to display the peripherals data in the μVision environment. Similar to AGDI there are also two example projects, an analog-to-digital converter and a timer similar to the original 8051 CPU timer, available for this interface. They can be used as a starting point for creating and adding a new peripheral.

There are different functions available in the AGSI interface that handle the configuration setup as well as the execution of the different debugging operations. The only one that is called by μVision itself is the AgdiEntry function. It is used to initialize the debugging session and to set up the peripheral simulation. The other functions can be categorized into the following groups:

- Defining values (special function registers (SFR), virtual registers (VTR, timer,...) during the initialization phase
- Read/Write operations on the memory, SFR or VTR
- Receiving status information
- Controlling the simulator
- Storing and retrieving configuration data
- Retrieving symbol names and values

A description of the most important functions for the different groups can be found below [26]. A complete list of all AGSI interfaces is presented in the appendix of this master thesis (see Appendix A.2).

- `DWORD AGSIAPI AgsiEntry (DWORD nCode, void *vp)`: This function holds pointers to all other AGSI functions. It is the only function that is called directly from the Keil μVision 3 framework to intialize the additional hardware modul. The nCode parameter is used to identify the necessary AGSI function.
- `bool AgsiDefineSFR(const char* pszSfrName, AGSIADDR dwAddress, AGSITYPE eType, BYTE bBitPos)`: During initialisation this function can be used to define a SFR or a special bit in an SFR.

- `bool AgsiSetWatchOnSFR(AGSIADDR SFRAddress, AGSICALLBACK pfnReadWrite, AGSIACCESS eAccess)`: This function is used to set a new watch point onto a specific SFR to notice all accesses.
- `bool AgsiWriteSFR(AGSIADDR SFRAddress, DWORD dwValue, DWORD dwMask)`: This function is used to write a new value to a specific SFR.
- `bool AgsiReadSFR (AGSIADDR SFRAddress, DWORD* pdwCurrentValue, DWORD* pdwPreviousValue, DWORD dwMask)`: This function is used to read the value from a specific SFR.
- `bool AgsiReadMemory (AGSIADDR Address, DWORD dwCount, BYTE* pbValue)`: This function is used to read a value from a specific memory address.
- `bool AgsiWriteMemory(AGSIADDR Address, DWORD dwCount, BYTE* pbValue)`: This function is used to write a value to a specific memory address.
- `AGSIADDR AgsiGetProgramCounter(void)`: Returns the program counter of the model.
- `void AgsiStopSimulator(void)`: With this function it is possible to stop the complete simulation.

A big problem in a simulation can be a poor performance, leading to long simulation times. That can arise for example when the values of the peripherals are updated with every CPU instruction. In that case much data is generated and will result in a slowdown of the complete simulation. For that reason, μVision uses an event driven approach. Only when one of those events (also called watches) occurs, that can be a read/write access to a special register or a timer that expires, the values from the peripheral are updated. For example if a new analog-to-digital converter (A/D converter) shall be connected to the μVision framework via the AGSI. As the A/D converter is not working most of the time it would make no sense to simulate it when it is inactive. Therefore *access watches* (AgsiSetWatchOnSFR and AgsiSetWatchOnVTR) have to be set during the initialization, for example on registers or external inputs (pins). The pins are later used to initialize and start the A/D converter. Now μVision can observe the activities of the new component and only simulate it when it is processing and executed.

5.2.3. μVision Socket

The last interface that is presented in this section is the *μVision Socket* [28]. It is used to control the Keil μVision 3 IDE from an other program. As the other two interfaces, AGDI and AGSI, are used to integrate external sources into the μVision framework, the μVision Socket is working the other way around. It allows external programs to control and monitor μVision remotely. This includes not only the controlling of the debugging mechanism that are available in Keil μVision 3. It is used to interact with the framework itself. For example it is possible to launch the IDE remotely, load a specific project and switch on the debugging mode automatically. Therefore a TCP/IP connection is provided in two ways: As a direct TCP/IP connection or a C interface that uses the TCP/IP interface internally. For this master thesis the second possibility was chosen. This is also the recommended solution from Keil because it

hides the TCP/IP details from the developers. The C interface is available inside a Windows Dynamically Linked Library (DLL) and also embedded into a sample application. For this master thesis that application is extended with windows client and server sockets that are also used in the other modules of the debugging environment. With the help of these sockets the μVision Socket is connected to the MDI module.

To present the μVision Socket interface in more details, the accessible functions which are also explained in the μVision Socket application note of Keil [28], are listed and described below. As this interface offers a lot of different possibilities to interact with the Keil μVision 3 framework only the functions that are important for this master thesis are presented here. A detailed list of μVision Socket interfaces is presented in the appendix of this master thesis (see Appendix A.3).

- `UVSC_STATUS UVSC_Init(int uvMinPort, int uvMaxPort)`: this is the first function that has to be called during the initialization phase of the μVision client. The minimum and maximum ports for the TCP/IP connection are specified with this function.
- `UVSC_STATUS UVSC_DBG_START_EXECUTION (int iConnHandle)` : starting the software code.
- `UVSC_STATUS UVSC_DBG_STOP_EXECUTION (int iConnHandle)` : stoping the execution of the simulation.
- `UVSC_STATUS UVSC_DBG_STATUS (int iConnHandle, int *pStatus)` : receiving the current status of the simulation. for example if it is running or suspened.
- `UVSC_STATUS UVSC_DBG_STEP_INSTRUCTION (int iConnHandle)` : used to step one instruction at the assembler layer.
- `UVSC_STATUS UVSC_DBG_STEP_INTO (int iConnHandle)` : used to step into a function in the high level software description.
- `UVSC_STATUS UVSC_DBG_STEP_OUT (int iConnHandle)` : used to step out of a function in the high level software description.
- `UVSC_STATUS UVSC_DBG_RESET (int iConnHandle)` : reset the simulation. This includes the software aswell as the simulated hardware parts.
- `UVSC_STATUS UVSC_DBG_MEM_READ (int iConnHandle, AMEM *pMem, int memLen)` : reading data from a memory address in the Keil μVision 3 memory.
- `UVSC_STATUS UVSC_DBG_MEM_WRITE (int iConnHandle, AMEM *pMem, int memLen)` : writing to a memory address in the Keil μVision 3 memory.
- `UVSC_STATUS UVSC_DBG_CREATE_BP (int iConnHandle, BKPARM *pBkptSet, int bkptSetLen, BKRSP *pBkptRsp, int *pBkptRspLen)` : creating a breakpoint in the simulation.

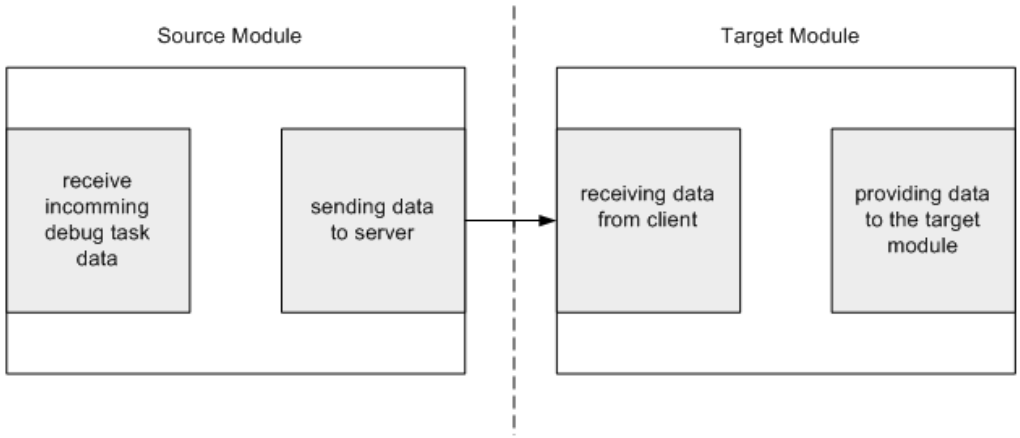

Figure 5.5.: Communication procedure between client and server socket

5.3. System Modules

In this section the modules that are used to build the new debugging environment are explained in detail. An overview of the complete system and its components can be seen in Chapter 4. The focus thereby lies on the parts that have been implemented and added during this master thesis.

5.3.1. Client/Server Sockets

Client and server sockets are used in every single component in the debugging environment. They enable the communication and data transfer in the framework. In Figure 5.5 the communication procedure between a client socket and a server socket can be seen.

First the new debug task request is received in the client socket, identified and then sent to the corresponding server socket. The server socket then receives the data, processes it and provides it to the correct traget modul.

The following set of functions are available in the client and server to interact with each other:

- `bool Close()`: closes the socket
- `bool RecvAck()`: waiting to receive an Ack from the server via the socket
- `bool SendAck()`: sending an ACK to the server via the socket
- `bool SendString(char *pStr)`: sending a string to the server via the socket
- `bool SendInts(int *pVals, int iLen)`: sending integers to the server via the socket

- `bool SendBytes(char *pVals, int iLen)`: sending bytes to the server via the socket
- `bool SendFloats(float *pVals, int iLen)`: sending floats to the server via the socket
- `bool SendDoubles(double *pVals, int iLen)`: sending double to the server via the socket
- `int RecvString(char *pStr, int iMax, char chTerm)`: receiving a string from the server via the socket
- `int RecvInts(int *pVals, int iLen)`: receiving integers from the server via the socket
- `int RecvFloats(float *pVals, int iLen)`: receiving floats from the server via the socket
- `int RecvDoubles(double *pVals, int iLen)`: receiving doubles from the server via the socket
- `int RecvBytes(char *pVals, int iLen)`: receiving bytes from the server via the socket

They enable the client/server sockets to send and receive data from various types (integer, double, string, bytes) and also to manage and establish connections to other client or server sockets. For synchronization issues during a connection the `RecvAck()` and `SendAck()` functions are used.

The client and server sockets are used to set up connections to the different modules. One example, a server and a client class to the MDI module can be seen in Figure 5.6.

Such classes exist for all other modules (AGDI, microprocessor model, MDI, CDT, μVision Socket) in the debug environment. The modules start up their clients and servers as threads so that they can run independently from the rest of the program. This is necessary as especially the servers continuously have to check for new requests from their clients.

5.3.2. Model Debugging Interface

The Model Debugging Interface is the main component for the communication in the design. This module holds client and server sockets for all other modules. If one component, for example the AGDI, wants to send data to another module it sends the data via its client to the MDI. The MDI decides, depending on the command id that is transferred together with the data, to which module it has to send the data. The MDI and all of the client/server connections are illustrated in Figure 5.7. During the initialization phase of the debugging environment, the MDI is creating the initial connections to all servers and clients. First the servers for each module are started in the MDI, waiting for a connection attempt from their client. As soon as they get a connection, the corresponding client in the MDI tries to connect to the server in the initiating module. If this connection attempt was successfully for all clients/servers the system communication is correctly established.

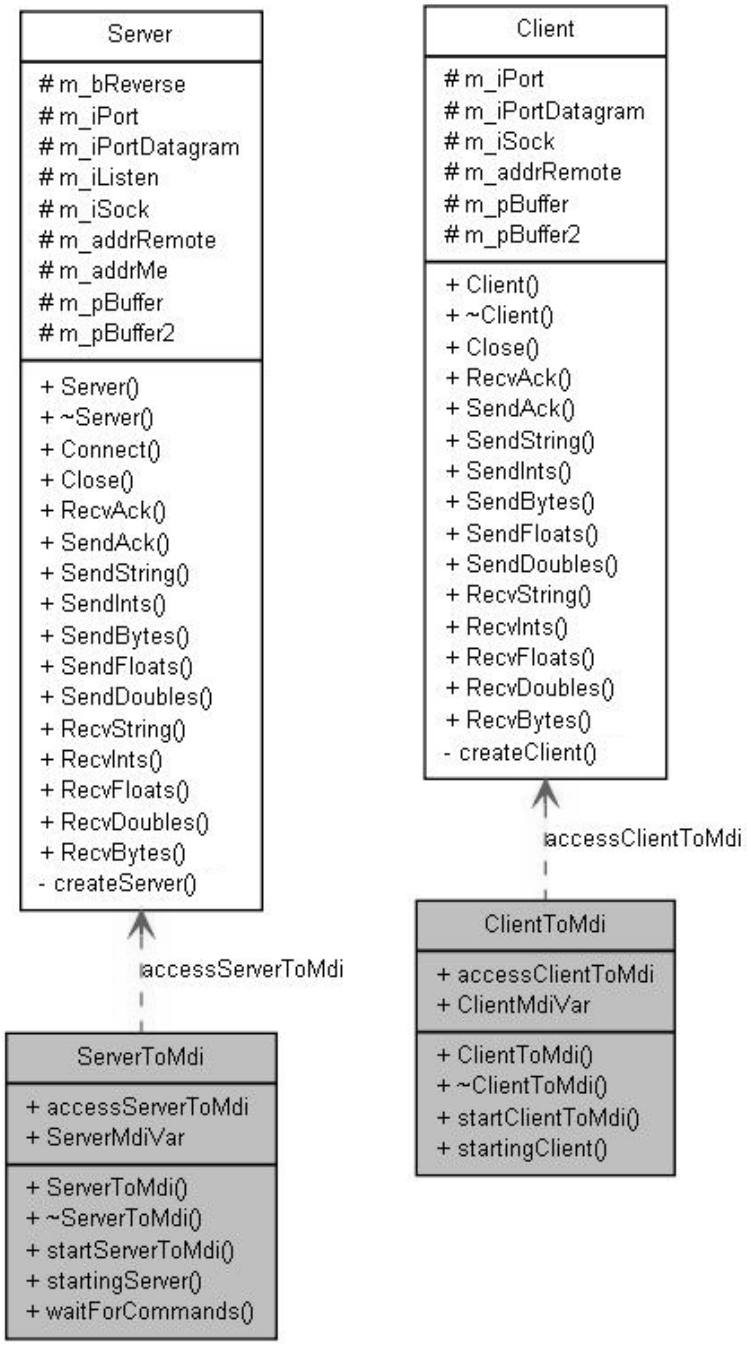

Figure 5.6.: A client and server class to the MDI module

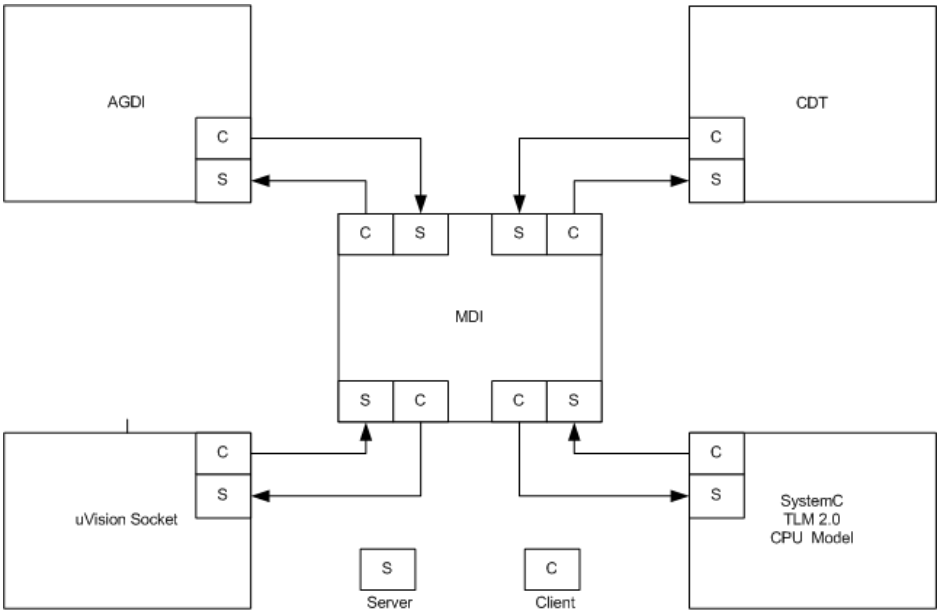

Figure 5.7.: Overview of all Clients and Servers

When a new debug task request reaches one of the servers in the MDI the following steps are performed by the MDI module:

1. Receive the command ID to identify the request
2. Receive the desired data (depending on the command ID)
3. Send the command ID to the target module
4. Send the data to the target module
5. In case of a read request: wait for the return data and send it to the initiating module

The beginning of the procedure is the same for all types of debug tasks that can occure in the debugging framework. To process the task in the correct way it has to be identified first. This is done via the Command ID, which is a unique ID in the debugging environment and identifies one specific debug task. After that the necessary data and information are loaded from the client into the MDI and are stored there temporarily before they are passed to their destination module. Again the command ID is transmitted to the destination component in advance to identify the task to the target module.

Still it is important to notice that a *read debug task* is treated differently than the other debug tasks. As already described in Chapter 4.3.3 the read requests require a response data to be send back. So the MDI waits till the target module transmits the response data and forwards it then to the initiating module.

The command IDs identify a debug task and are shared among the whole debugging environment with a unique identification number. This number is stored in a definition file and used by all classes to keep it consistent in the framework.

The MDI is also for another reason very important for the debugging environment. It is the constant factor in the system that keeps running the whole time. As it can happen during the debug process that the Keil μVision 3 IDE and the Eclipse environment are suspended. This can happen for example if one of them reaches a breakpoint and is suspended. Then this information is transferred across the debugging environment to stop the other IDE too. The MDI and therefore the communication system stays alive during that time to enable afterwards the restart of the system.

5.3.3. Extending Eclipse C/C++ Development Tooling

The necessary code changes have been implemented in the Target.java file of the CDT[5]. There exist functions for executing every debugging method available in the CDT. This code is also used when a debug task from an external component arrives at the CDT via its server socket. Then, depending on the command ID, the right debug method is called.

To be able to change the implementation of the CDT plugin and to extend it with new functionality it is necessary to launch the CDT source code in the Plug-in Development Environment (PDE) perspective of Eclipse. Therefor the Java development package of the Eclipse IDE has to be used. The source code itself can be downloaded from the Concurrent Version System (CVS) of the Eclipse project. A detailed description for the single steps can be found on the homepage of the Eclipse website[6].

To enable the communication a server and a client socket (see Section 5.3.1) haven been added to the CDT, similar to all other modules in the debugging environment. The server waits for incoming debug tasks and identifies them with their command ID. It uses the same debug functions as the original CDT plugin to control the software.

5.3.4. Implementation Hardware Debug Module

In Figure 5.8 the collaborational diagram of the HDM is presented.

It can be seen that the HDM uses two other classes and one interface. The two classes are the ClientToMdi and the ServerToMdi, which have been discussed already in Chapter 5.3.1 in detail. They handle the interaction to the external debugging program and the HDM uses them to send and receive data.

It also implements the Debug Interface, using the checkForDebugRequests function to check the lists for new debug tasks. The lists hold all debugElements that have arrived at the model and need to be processed. A debugElement is created by the server socket when a new debug task is received. The debugElement is a class that contains variables for important debug task values:

1. Command ID: identifying the debug task

[5] org.eclipse.cdt.debug.mi.core/cdi/org/eclipse/cdt/debug/mi/core/cdi/model
[6] http://wiki.eclipse.org/Getting_started_with_CDT_development

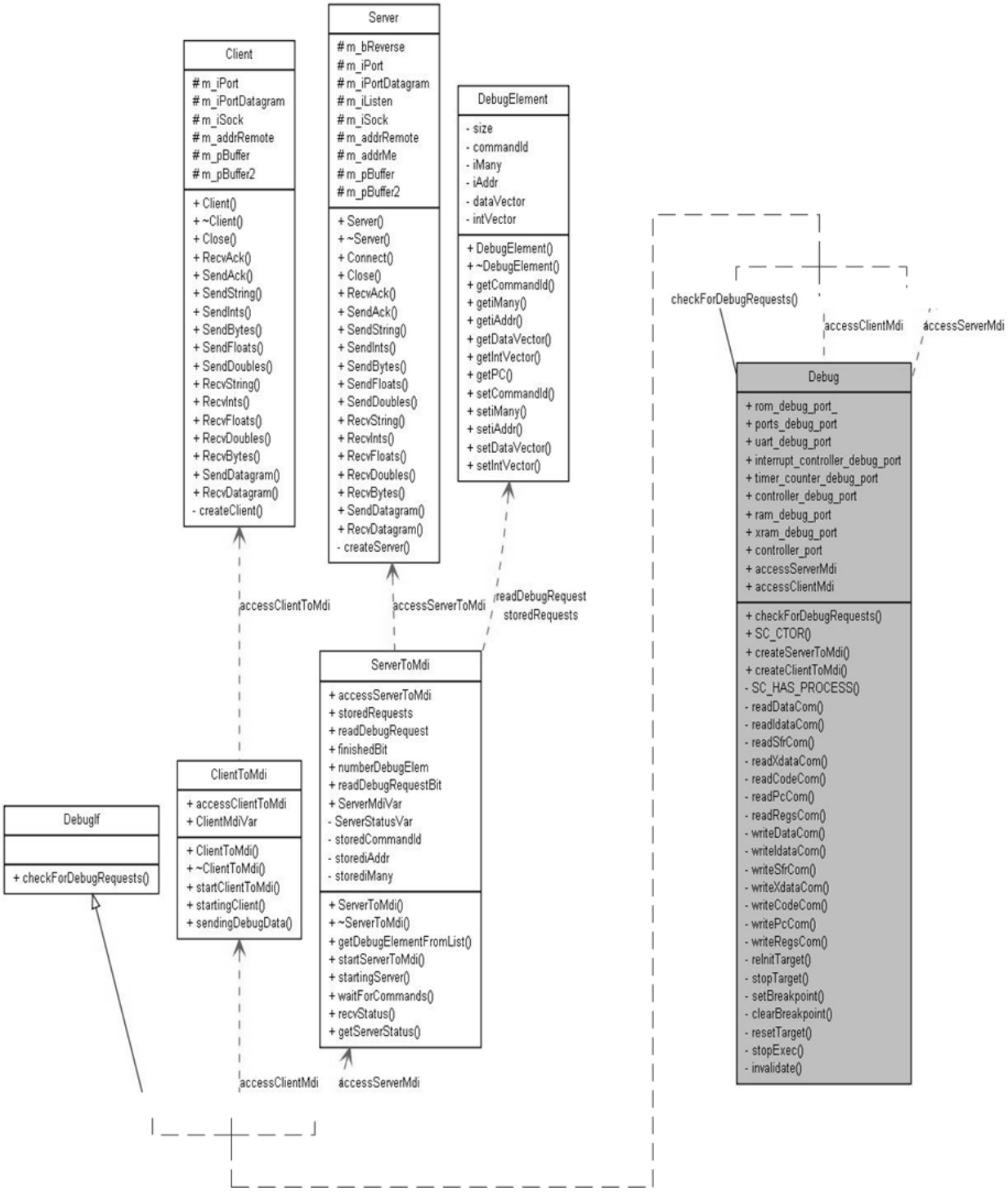

Figure 5.8.: Collaboration diagram for the Hardware Debug Module:

2. Address: used when accessing a value in the memory
3. Number to specify the usage of a debug task: length of data or number of necessary executions
4. Data vector: storing the data from the debug task

Debug Functionality Interface

To integreate a new model into the debugging environment it is necessary that its modules implement a special debugging interface: the *Debug Functionality Interface* (DFI).

The DFI features the following functions for debugging a task in one of the modules:

- readDebugTask()
- writeDebugTask()
- commandDebugTask()

The interface functions are used for the three different debug tasks (write, read and command). Each of those functions can be called by the debug module. Then the corresponding target module has to handle those requests. After that the active debug element is processed, containing all information that are needed for a special debugging task. Therefore functions to get and set the values in the debug element are provided. Further processing of the data and the creation of the response data, in case of a read debug task, lies in the area of responsibility of the single modules. As long as they implement the interface functions, they are able to interact with the debug module.

An example how to use the DFI and how the design effects the usage of the DFI can be found in Chapter 6.1.3

5.4. 8051 Processor Model

The 8051 SystemC TLM 2.0 microprocessor model is used to demonstrate the integration of a hardware model into the new developed debugging system. The model was created as a team project together with two other students at the university of Graz [12]. The processor model is used as the basis for the ongoing projects of each team member and therefore also for this master thesis. The idea was to develop a SystemC model of an 8051 microprocessor in a TLM 2.0 style. The advantages of TLM 2.0 are explained in Chapter 2.4.1. The model implements the original Intel 8051 microprocessor but the design is based on the open source microprocessor model of *Oregano Systems* [7]. The Oregano core is available as a synthesizable VHDL code. It implements the complete instruction set of the original Intel processor but has some slightly different design changes to improve the performance of the system. The biggest change to the original implementation is the management of the registers. In the

[7] http://www.oregano.at/ip/8051.htm

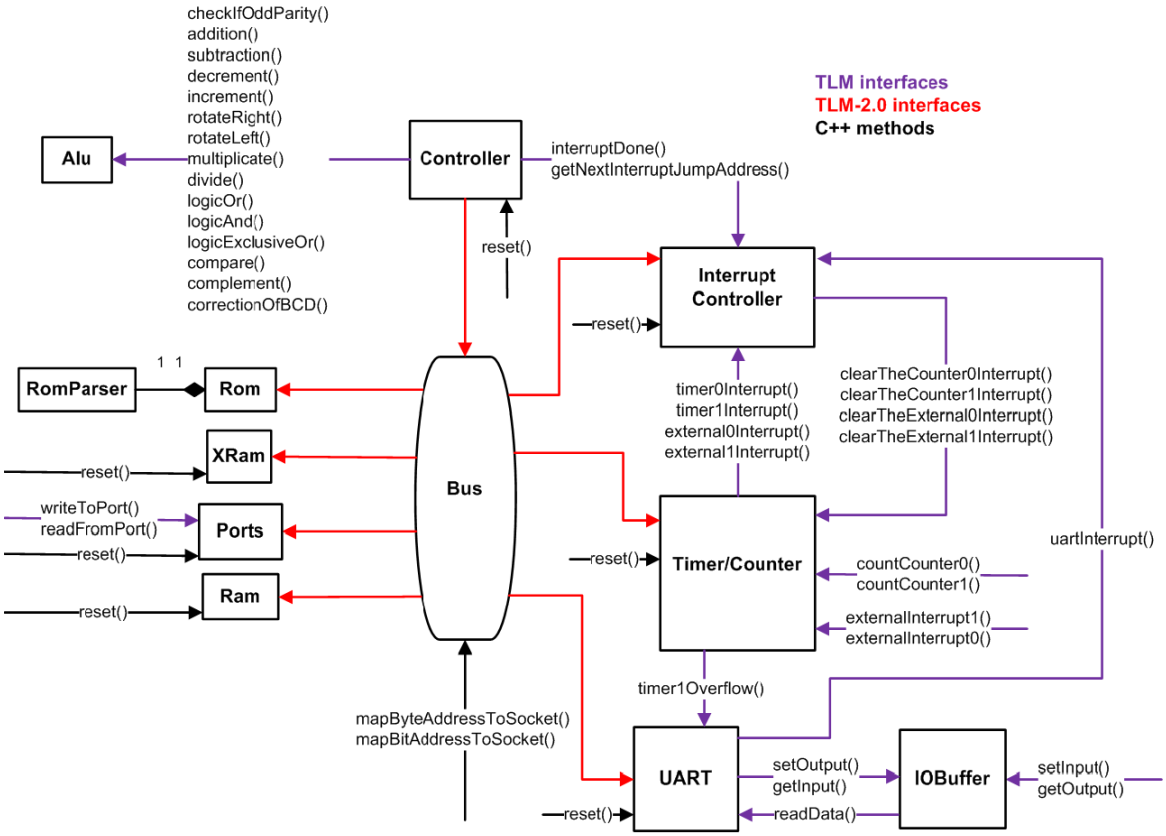

Figure 5.9.: Detailed block diagram of the 8051 TLM 2.0 model [12]

original design all registers of the microprocessor are located in the RAM at certain addresses. In the Oregano design, and therefore also in the design of the new 8051 TLM 2. model, the registers are stored directly in the components that are using them. The advantage of this approach is a benefit in performance speed and an easier integration of additional modules. Also the activity on the bus is reduced as every component has direct access to its registers. In Figure 5.9 the architecture of the 8051 processor can be seen.

Except the direct connection between the arithmetic logic unit (ALU) and the controller all other modules are connected to the controller via the bus. The bus module is implemented in SystemC TLM-2.0. When the controller wants to send new data to another module, it creates a payload. This payload consists of the following three parts: the data, the address and the extension. The extension can be either a RAM-extension, a ROM-extension or a XRAM-extension. This helps the bus to decide to which component it has to send the payload. Is it a RAM-extension the bus also has to send it to all other modules one after the other. They check the address in the payload then and decide, as they know their own addresses, if the payload was meant for them or not. To be able to connect to the bus, the components have to implement the b_transport() method that is specified in TLM 2.0 too.

As displayed in Figure 5.9 the main component in the 8051 microprocessor model is the

controller. It is connected to the ALU directly and it is also the bus master component. This means the controller can access the bus and create transactions for the other modules. Also the complete instruction set (255 methods) of the original Intel 8051 microprocessor is implemented in the controller. Every instruction is a method that is accessed via a function pointer array. To simulate the timing, wait() statements have been inserted using the timing information of the original 8051 instruction set.

Another important module is the RAM component of the model. It can hold up to 128 bytes of data and is byte and bit addressable. The option can be set in the extension of the payload. If the byte addressing option is set, all values in the RAM can be accessed. Using bit addressing, only the values between byte addresses 0x20 and 0x2F can be used. Similar to the RAM is the ROM module of the 8051 model. The only difference is the size, as it can hold up to 0xFFFF bytes, and that it is only able to be read from and not written to. The last storage module in the model is the XRAM component and it can also store 0xFFFF bytes of data.

The interrupt controller module holds five variables that represent the five different interrupt sources (UART, timer 0 and 1, external 0 and 1). If a new interrupts occur, they are saved into a list and processed one after the other by the controller, depending on their priority.

As mentioned before, one source that can create two interrupts is the timer/counter. It also includes a logic for handling the external interrupts for the system.

To send and receive data from the outside a Universal Asynchronous Receiver Transmitter (UART) is used. As its name already implies, the UART consists of two parts: the sender and the transmitter. Both are implemented as threads, waiting for incoming or outgoing requests.

The last module, that consists also of two parts, is the ports module. It contains pins and registers. Other classes can use them via their readFromPort() and writeToPort() interface functions to read or store values into the model. During a read or write access, a bit-wise conjunction of the pin and register value is performed. So only when the right bit value is set to 1 by the user program, a valid data can be generated. Otherwise the value will stay 0.

To be able to use the microprocessor model in the new debugging design, it was necessary to migrate the 8051 CPU model from the original Visual Studio project into an Eclipse compatible version.

6. Results and Evaluation

In the chapters before this master thesis is giving background information about the research topics in this area, the design and the implementation of the new debugging system are described and the parts of the framework are presented in details. This chapter will describe the process of adding a new model to the system. This is important as new developers want to debug their own modules with the debugging environment. This example is also used to demonstrate the usage of the new design and evaluate the results. Therefore the 8051 TLM 2.0 model, which is described in Section 5.4, is used. Also other open source models are evaluated to demonstrate how and if it is possible to integrate them into the new debugging framework.

When talking about integrating new models into the debugging environment, the Keil μVision 3 IDE distinguishes between two different kinds:

- CPU models
- Additional components

The μVision framework offers two different interfaces for them: The AGDI interface for connecting CPU models and the AGSI interface for integrating additional components. Those interfaces are described in Chapter 5.2.1 and 5.2.2. The new debugging system is taking care of establishing the connection to Keil via the MDI module. All necessary steps that have to be done are explained in Chapter 6.1. This includes the implementation of the DFI interface (Chapter 5.3.4), the usage of the server and client sockets and the debug module.

6.1. Integration Process

This section will focus on the integration of a microprocessor model. Therefore the 8051 TLM 2.0 model, that is described in the section before, is used as an example. The integration can be split up into three parts:

- Establishing the communication (see Section 6.1.1)
- Integration of the Debug Module (see Section 6.1.2)
- Implementation of the Debug Functionality Interface (DFI) (see Section 6.1.3)

Those three parts can be compared to steps in the lifetime of a debug task. First the task is received in the module and therefore the communication system has to be set up. Then, in the second step, the received data has to be transmitted to the correct address in the module. This is done using the integrated debug module. And finally, the implementation of the debugging interface functions. They enable the modules to process the debug task and, if necessary, create the required response data. This is the case when processing a read debug task.

The required implementations effect not all modules in the debugging environment. The CDT, the MDI, the μVision socket and the interface in the Keil μVision 3 IDE do not have to be modified at all. Details about the necessary steps can be found in the following sections.

6.1.1. Establishing the Communication

To connect the new model to the communication system the server and client sockets, explained in Chapter 5.2.3, are used. Therefore both sockets have to be created and started. They are integrated into the debugging module so that, as soon as this module is instantiated, also the communication is started and connected to the MDI module. This is all done automatically with no need for the developers to change anything themselfs.

6.1.2. Integration of the Debug Module

The main block that is responsible for enabling the debugging in the new model is the Debug Module. This module uses the received and stored debug tasks to transmit them to the correct destination modules. Therefore the debug module accesses a stored debug element and obtains its command ID. This ID is a unique number to identify a debug task. Depending on this ID the debug module chooses the right function and sends the active debug element to the destination module. This is also the place where the developers have to add model specific code. Every function represents one specific debug task. As different CPU models have very different architectures and designs, their components and instruction sets vary very much. For example the registers are stored at different places or there exist multiple timers or more than one UART in the model. That applies also for the additional hardware modules. Not only the architecture of the models can be different, but also the abstraction layer of their implementations. For example the Register Transfer Level layer or hardware description languages are often used for implementations of new hardware modules.

The debug tasks are the same for all of those different types of models. So as long as the interfaces are implemented correctly and the modules are connected correctly to the debug module, the new debugging framework is able to handle various kinds of architectures and processor types. On one hand the AGDI, which defines the different debug tasks for the CPU model and on the other hand the AGSI interface, which specifies the functions needed for the additional hardware models. In the debug module every debug tasks needs to know which module it needs to contact. Therefore the developers have to insert the code to call the DVI functions. The DVI interace is explained in the following section. Then the debug element

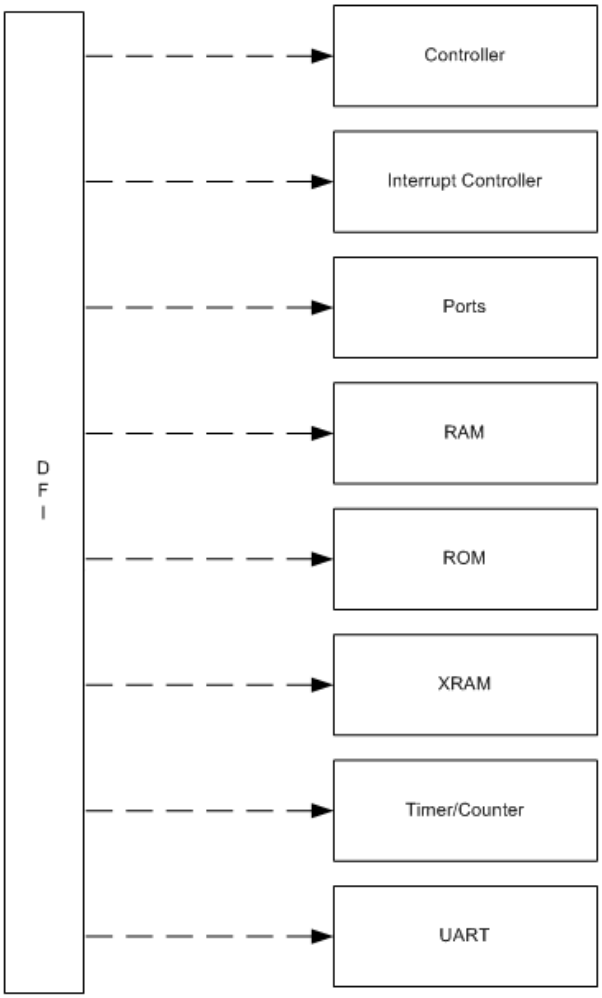

Figure 6.1.: Overview of models implementing the DFI interface

is transfered to the target module and, in case of a read debug task, the response data is returned to the debug module. The transmission to the MDI is done automatically.

6.1.3. Implementation of the DFI interface

The DFI interface, as described in Section 5.3.4, has to be included in all modules of the model which have a necessary function for the debugging process. For example this can be a function to store data in the RAM or the ROM module. Those modules hold values values that are important for the debugging methods. It is also necessary to add the DFI if a module is used for processing or executing a command debug task. In case of the 8051 TLM 2.0 microprocessor model, the modules including the DFI can be seen in Figure 6.1

It can be seen that most of the modules from the microprocessor model have to implement the DFI. The reason for this lies in the design of the 8051 TLM 2.0 model. As every module holds its own special function registers, also every module needs to implement the interface to the debug module. In comparison to the original 8051 microprocessor design, which stores all registers in the RAM, much more modules are involved during debugging. This means that the integration work, that has to be done by the developers who want to integrate their new models, is strongly depening on the design of the model.

6.2. Evaluation

As the design of the new debugging system is completely base on the usage of different interfaces, new models and modules can be easily added to the framework. Therefore the integration process for a 8051 TLM 2.0 microprocessor model was presented in the section before.

This section will focus on the integration of other components into the new debugging framework. Therefore opensource SystemC models have been investigated to show how the framwork can be used to debug them and what necessary changes have to be done to accomplish the integration. Those new models include different types of components, for example a processor with a different architecure, a new peripheral module or even a completely new communication simulator. The evaluated models are:

- AES/DES cryptoProcessor [9] (see Section 6.2.1)
- Reed-Solomon decoder [3] (see Section 6.2.2)
- NoC simulator (network-on-chip) [18] (see Section 6.2.3)

The results of the evaluation for each of the three new components is presented in the following subsections

6.2.1. AES/DES CryptoProcessor Model

The first component that was evaluated for an integration into the new debugging framework is a SystemC model of a cryptoprocessor. The processor was developed by a group of hardware/software developers at the University of Rey Juan Carlos (Madrid) [9]. The new model was designed to work as a cryptoprocessoer taking over the caculation of the AES and the DES encyption and decryption. The model is implemented in the SystemC system description language and was used as a starting point for the development of their new silicon coprocessor. Another reason why this project is from interest for this master thesis is the way the SystemC model is implemented. Similar to the processor model presented in Section 5.4 the spanish research tem also used the TLM approach to specify their new processor. This helped them to receive an early verification of their design.

In Figure 6.2 an overview of the cryptoprocessor model system can be seen. The setup consists of four main components. In the testbench the test data for the cryptoprocessor is

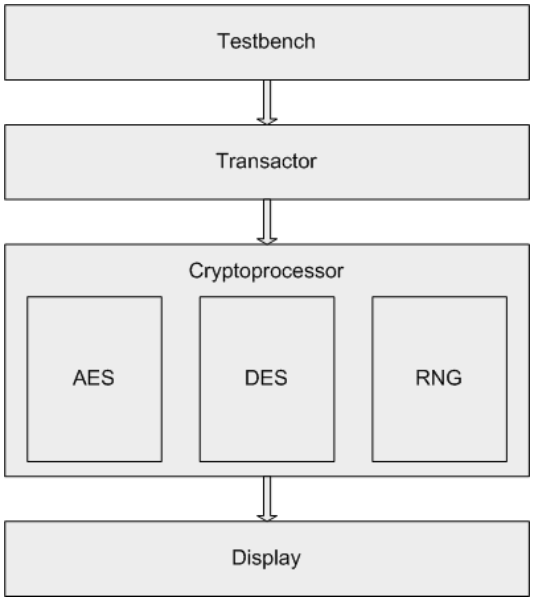

Figure 6.2.: Overview of the cryptoprocessor model system

generated. This data is then prepared with the help of the transactor module. Then the processor model is able to process these requests. The transactor also assures the re-usablility of the system as it can be adjusted to other data sources by simply changing the transcator itself. With this approach the implementation of the system behind it can stay as it is. The third important module in the system is the cryptoprocessor component. It is implemented as C++ functions with a SystemC wrapper surrounding them. It is able to calculate *Advanced Encryption Standard* (AES) and *Data Encryption Standard* (DES) operations. Therefore it also includes a random number generator that is used for these cryptographic calculations. To be able to show the results of the operations the system is connected to a display.

Results Integration Evaluation

The most important part to enable the evaluation of the cypto-processor and its components is the possibility to get access to the source code of this system. The SystemC source code files for the implementation of the DES calculation were received from the OpenCores initiative[1]. Different to the project described in the paper [9], this source code includes only the implementation of the DES algorithm. For the evaluation of the cryptoprocessor system the missing AES is not important as it does not change the structure of the processor but only reduces the cryptographic potential of it. Apart from the SystemC implementation of the DES encoder and decoder also the Verilog sources for a synthesis are provided.

As described in the section before and presented in Figure 6.2, the first module in the cryptographic system is the testbench component. It is necessary to generated test data for

[1] http://opencores.org

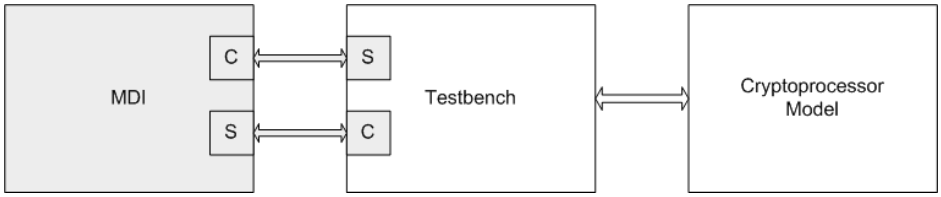

Figure 6.3.: Evaluation cryptoprocessor model system

the system and therefore this testbench is implemented in SystemC. During evaluation of the source code of the complete cryptoprocessor it was soon clear that this testbench would also be a very good connection point for the new debugging framework that was designed for this master thesis.

The necessary changes for integrating the cryptoprocessor into the debugging framework are illustrated in Figure 6.3. Similar to the integration of the microprocessor model described in Section 6.1, the process for the cryptoprocessor is similar. To connect the new component with the debugging system, a client and server architecture is established between the model and the MDI component. For the connection Windows sockets are used for passing data and commands to the cryptoprocessor and for returning the data to the MDI module. To be able to handle the incomming data via the server socket the testbench module also has to implement the DFI interface (see Section 5.3.4).

6.2.2. Reed-Solomon decoder

When transmitting digital data, special decoders are used to encode and decode the values. The *Reed-Solomon* decoder is a device that is integrated in various kinds of applications. This includes for example audio cd-players, mobile communication devices and digital television. Beside the decoding the Reed-Solomon decoder is also able to recognize and correct errors that have occured during the data transmission. Due to its wide range of applications where the Reed-Solomon codes can be used, a number of different scientific groups are working on new ways to improve the performance and minimize the enegery consumption of these decoders. For example the project from a group from the university in Cairo [7]

To be able to demonstrate that also components with cryptographic operations and algorithm can be used together with the debugging framework, the Reed-Solomon decoder was chosen for this evaluation.

An overview of the decoder architecture can be seen in Figur 6.4.

After the input data has been passed to the decoder a padding is applied to the received values. The padding is necessary to be able to use the algorithm in the decoder. The main part in the component is the error corrector. It calculates the corrected values if an error has occured. Therefore the output is different depending on the error correction. Also the padding is removed at the end to receive the plain data again.

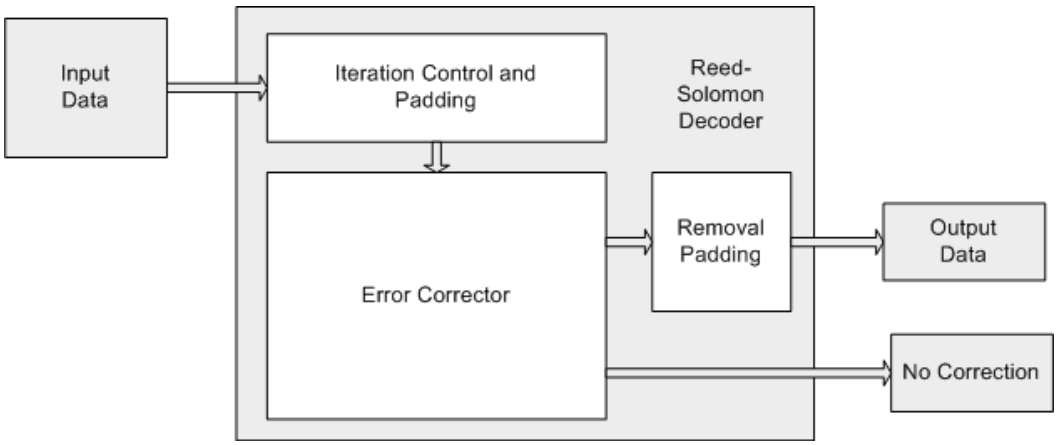

Figure 6.4.: Architecture of the Reed-Solomon decoder

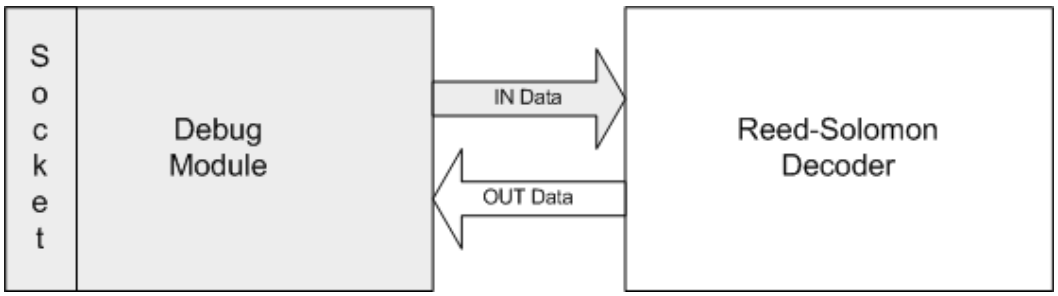

Figure 6.5.: Evaluation of the Reed-Solomon decoder

Results Integration Evaluation

The Reed-Solomon decoder simulation has two main access points. The first one is the input channel where the data, that has to be processed, is put into the decoder. The second one is the output port where the result data can be received again. These two ports are also important when adding the decoder simulation to the debug framework.

As displayed in Figure 6.5 the debug module is connected to input and output of the simulation.

The data is received via the socket and the debug module is then using the input port of the decoder simulation to fill it with the new values. Once the data is processed the debug module receives the result and then returns o via the socket. During the processing of the data the debugging of the simulation can be controlled by the debug module and the Eclipse CDT plugin. This demonstrates that the debugging framework, presented in this master thesis, is easily adjustable also for small additional components.

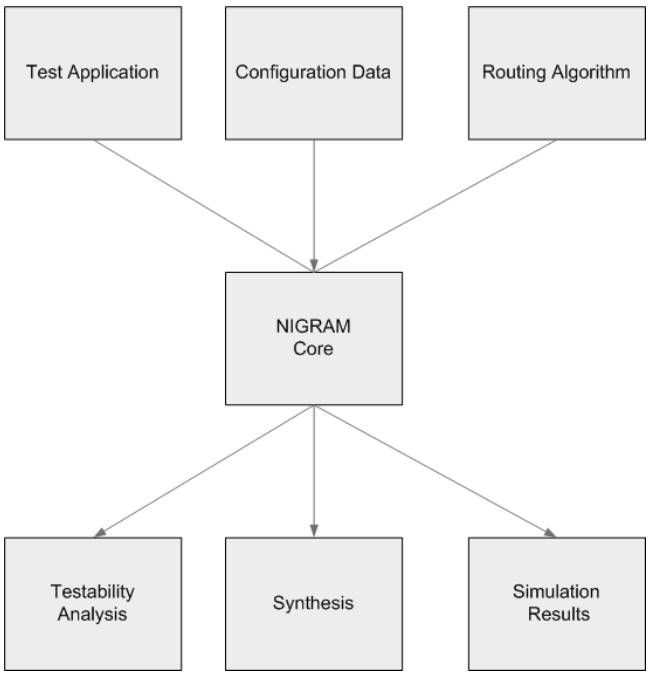

Figure 6.6.: Overview of the NIGRAM Network-on-Chip simulator

6.2.3. NoC Simulator (Network-on-Chip)

One of the most important features for a debugging framework is the posibillity to extend the framework with new components. Therefore the integration of a bus system or a network structure is an important task. An open source project implementing such a NOC component has been chosen for a detailed evaluation and is explained in this section.

By using a Network-on-Chip infrastructure it is possible to combine multiple components into one system. One interesting project regarding this topic is the *NIGRAM Simulator* [18]. NIGRAM is a SystemC base simulator that allows cycle accurate simulations. The system supports different kind of 2D network topologies and different routing algorithms. In Figure 6.6 an overview of the simulator framework can be seen. The main module and the heart of the system is the NIGRAM core component. It uses the input data and the mathematical background, provided by the other modules, to create the desired results. In NIGRAM the input is generated by three modules. One is providing the routing algorithm that can be chosen for the simulation. The second input module is used to modify the system to the special requirements and provides therefore configuration possibilities to the user. The last, and for this masterthesis also most interesting input module, is the *test application* component. With its help it is possible to interact with the Simulator and use it together with test programs to evaluate different routing algorithm and network issues. After the processing by the main module is finished the NIGRAM simulator provides three different result options. For example the test and simulation data can be used to calculate the percentages of fault detection and fault isolation. In the NIGRAM framework this is done in the testability analysis component. Of course it is also very important to know if the results of the simulation

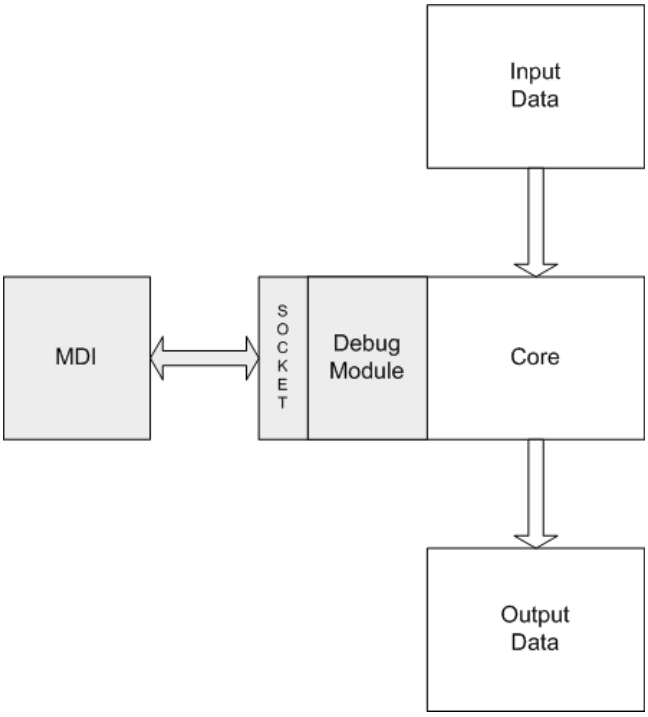

Figure 6.7.: Evaluation extended NIGRAM Network-on-Chip simulator

would also work as a low level implementation. Therefore also a procedure to generate the logic synthesis from the simulation is available. The last output of the simulation are the simulation results in general. As they have been used also in the testability component the NIGRAM framework provides even more data to the user. This includes values that were created by the simulation, graphs of the different simulation results as well as performance analysis evaluations that were made during the simulation.

Results Integration Evaluation

When integrating the network-on-chip into the debugging framework it is important to establish full control over the simulation. The evaluation of the NOC simulation code showed that the best place to integrate the debugging module is the core of the simulation itself. As explained before, the core is the central point in this design. In Figure 6.7 the extended simulation can be seen.

The connection to the MDI is established via the sockets and the debug module, which is implementing the DFI interface. With this approach it is possible to control both directions, incoming and outgoing data, of the simulation. This is necessary for starting new calculations with different inputs and for beeing able to return the caculated results. The evaluation showed that the changes only have to be done in the `main.cpp` which is executing the main process loop of the NOC simulation. This is also the place where the debug module has to be

integrated to allow direct access to control, modify and receive the necessary data for the simulation.

7. Conclusion

The area of hardware/software co-design has been a very interesting topic in the last decades and also will be a promising area for research in the following years to come.

As described in the chapter about the related work (see Chapter 3) there exist very different approaches to provide solutions for this topic. The common goal is the development of debugging systems and tools that help the developers to do their work faster and to find more bugs. Time is money and so a faster development process has become an important factor when creating a new product. This trend will also continue in the future for sure.

7.1. Results

The new debugging system, created during this master thesis, combines the advantages of all the other approaches presented in the related work in this master thesis. Well known and often used IDE frameworks (μVision, Eclipse, Microsoft Visual Studio) are used in this new design. This helps the developers to work with the environment, because they are already familiar with the underlying IDEs and the debugging methods that are offered to them.

One of the main tasks for this master thesis was to design a debugging system and to find the right applications to establish it. Interfaces, like the AGDI and the AGSI interface from the Keil μVision 3 IDE, are great for adding new sources to the IDE. As Eclipse and the CDT plugin are open source too, adding the desired functionality into those parts was possible. This was also the reason why Eclipse was chosen instead of Visual Studion from Microsoft. The possibilities to extend the code of this framework are very limited and also the integration of new tools or models is hardly possible. Especially for this master thesis, it would have been necessary to access the debug methods directly, either via an interface or by extending the debug methods themselves. Both ways would not have been possible with the Microsoft Visual Studio IDE. This is in general often a problem with commercial software applications. The companies have no interest in allowing other people to have a closer look on the insights of their implementations or supporting them with new and additional functionalities. Therefore this master thesis is a combination of both parts. Depending on the requirements of the parts in the system either commercial applications or open source tools are used.

The high abstraction layer of the debugging environment enables a fast simulation speed and also makes it possible to have an early look on the complete system. This is important for the designers as they can get an early overview of the whole systems to find faults in the design as early as possible.

The design of this debugging system is based on interfaces in almost all parts. This makes the system very flexible and enables the developers to replace single modules easily. Therefore, as the system is not focused onto only one architecture, it supports various kinds of processor types and additional hardware components. Although those components have different layouts and different instruction sets, the basic debug methods are always the same. An evaluation of different open source SystemC models has proofed that the debug system can be integrated also in existing projects. The evaluation therefore covers different kind of models, for example a cryptographic component or a NOC simulation.

7.2. Future Work

The integration process of the 8051 TLM 2.0 microprocessor model is a good example for showing the steps to include a new hardware model. A future work could be the creation of a *Dynamic Linked Library*(DLL) including the debug module and the sockets for the new hardware components. This will combine them into one package which will make it more comfortable to include for the developers. Another topic of interest could be a detailed performance analysis, comparing the simulation time of the debugging environment with the results of compareable systems.

As mentioned already at the beginning of this chapter, the future for hardware/software co-design looks promising. Not only when dealing with debugging, but also in the verification process of complex embedded systems. This is a very interesting scientific area for the future, but it will need new approaches, tools and techniques to cope with the rising complexity of the systems.

One promising approach for future debugging IDEs is presented in [22]: Hardware tracing. As this technique monitors and stores all instructions a processor executes, the developers are able to gain a detailed view into the internal operations of the system. So it would also be an interesting addition for the debugging system created for this master thesis, especially when dealing with bugs that only appear randomly during the execution.

Future work, using this debuggin framework, will also focus on the creation of new hardware models. This can either be a completely new processor model that is replacing the 8051 TLM 2.0 CPU model presented in Chapter 6 or new additional hardware modules extending the design. Those new models can be implemented in the SystemC system description language, like the 8051 processor model, but they can also be written in any other programming language. Those new models can be used to build up a pool of different components. So if something else has to be tested, the developers can choose from a stored model database which component they want to use. This way a new component can be set up really quickly.

Establishing an IDE framework that can debug hardware and software modules at the same time helps the designers to do their work faster and to find more bugs in a shorter time. The system designed and presented in this master thesis, is therefore a good starting point for further researches and the testing of new components.

A. Appendix

A.1. Advanced Generic Debugger Interface Functions

List of AGDI interface functions (see also Section 5.2.1):

- `UL32 ReadIdata (BYTE *pB, DWORD nAdr, DWORD nMany)`: reading Idata bytes from the microprocessor model
- `UL32 ReadXdata (BYTE *pB, DWORD nAdr, DWORD nMany)`: reading Xdata bytes from the microprocessor model
- `UL32 ReadCode (BYTE *pB, DWORD nAdr, DWORD nMany)`: reading Code bytes from the microprocessor model
- `UL32 WriteCode (BYTE *pB, DWORD nAdr, DWORD nMany)`: writing Code bytes to the microprocessor model
- `UL32 WriteXdata (BYTE *pB, DWORD nAdr, DWORD nMany)`: writing Xdata bytes to the microprocessor model
- `UL32 WriteIdata (BYTE *pB, DWORD nAdr, DWORD nMany)`: writing Idata bytes to the microprocessor model
- `U32 ReInitTarget(void)`: reset the communication
- `U32 InitTarget (void)`: initialize the communication to the MDI module
- `void StopTarget (void)`: stops the communication to the target
- `void ResetTarget (void)`: reset the microprocessor model
- `U32 StopExec (void)`: stop execution of the user program
- `void Invalidate (vp)`: invalidate everything that is necessary after GO or Step

A.2. Advanced Generic Simulator Interface Functions

List of AGSI interface functions (see also Section 5.2.2):

- `AGSIVTR AgsiDefineVTR(const char* pszVtrName, AGSITYPE eType, DWORD dwValue)`: During initialisation this function can be used to define a VTR.
- `bool AgsiDeclareInterrupt(AGSIINTERRUPT *pInterrupt)`: To add a new interrupt source from the hardware model this function is used.
- `bool AgsiSetWatchOnVTR(AGSIVTR hVTR, AGSICALLBACK pfnReadWrite, AGSIACCESS eAccess)`: This function is used to set a new watch point onto a specific VTR to notice all accesses.
- `bool AgsiSetWatchOnMemory(AGSIADDR StartAddress, AGSIADDR EndAddress, AGSICALLBACK pfnReadWrite, AGSIACCESS eAccess)`: This function is used to set a new watch point onto a specific address in the memory to notice all accesses.
- `AGSITIMER AgsiCreateTimer(AGSICALLBACK pfnTimer)`: If a software timer is needed, this function is called to create one. Everytime the timer expires a specific function is executed.
- `bool AgsiDefineMenuItem(AGSIMENU *pDym)`: This function is used to create a new menu entry for the hardware module or anything that belongs to it. The menu can be found in ther peripherals menu of μVision.
- `bool AgsiWriteVTR(AGSIVTR hVTR, DWORD dwValue)`: This function is used to write a new value to a specific VTR.
- `bool AgsiReadVTR (AGSIVTR hVTR, DWORD* pdwCurrentValue)`: This function is used to read the value from a specific SFR.
- `bool AgsiSetSFRReadValue(DWORD dwValue)`: In some cases it is necessary to replace the value that was read from a SFR with an instruction for the processor. For example when reading values from the ports, as they are read from the register and not from the single pins. This is done to simulate the original behaviour and therefore this function is used.
- `AGSIADDR AgsiGetLastMemoryAddress(void)`: To identify which access watchpoint has been reached this function is used.
- `bool AgsiIsSimulatorAccess(void)`: Used to distinguish a simulation and a callback call.
- `bool AgsiSetTimer(AGSITIMER hTimer, DWORD dwClock)`: With this function it is possible to access a software timer and change its expiration time.
- `UINT64 AgsiGetStates(void)`: Returns the number of steps during the simulation.
- `DWORD AgsiIsInInterrupt(void)`: This function returns the current interrupt level that is used.

- `bool AgsiIsSleeping(void)`: Checks if the CPU is in a sleep mode.
- `void AgsiTriggerReset(void)`: As the name already implies, this function is used when the CPU has to be reset.
- `void AgsiUpdateWindows(void)`: To keep the values up to date in the windows of μVision, this function is called to update them. If this function is called too often it can slow down the simulation.
- `void AgsiHandleFocus (HWND hwndDialog)`: To forward accerlerator keys, for example the tabulator, to the dialog messenger this function is used.
- `DWORD AgsiGetExternalClockRate(void)`: If the module uses an external clock rate, this function is used to retrieve this rate.
- `DWORD AgsiGetInternalClockRate(void)`: The internal clockrate of the CPU can be retrieved with this function. It can also be calculated with the external rate divided by the clock factor.
- `double AgsiGetClockFactor(void)`: Returning the clock factor of the external clock.
- `void AgsiMessage(const char* pszFormat, ...)`: For debugging purposes this function can print a text to the command window of Keil μVision 3.
- `bool AgsiSetTargetKey(const char* pszKey, const char *pszString)`: Configuration data can be stored as a text
- `const char * AGSIAPI AgsiGetTargetKey(const char* pszKey)`: To retrieve the text that was stored with AgsiSetTargetKey, this function can be used.
- `DWORD AgsiGetSymbolByName (AGSISYMDSC *vp)`: As symbols have a value, this function is able to retrieve the values.
- `DWORD AgsiGetSymbolByValue(AGSISYMDSC *vp)`: This function works exactlly the other way around as the function before. It is possible to reveive the name of a symbol only holding a value of the symbol.

A.3. μVision Socket Interface Functions

List of μVision Socket interface functions (see also Section 5.2.3):

- `UVSC_STATUS UVSC_UnInit(void)`: reverse operation of UVSC_Init. used to remove the Uninitialise UVSC.
- `UVSC_STATUS UVSC_OpenConnection (char *name, int *iConnHandle, int *pPort, char *uvCmd, UVSC_RUNMODE uvRunmode, uvsc_cb callback, void *cb_custom, char *logFileName, xBOOL logFileAppend, log_cb logCallback)`: used to open a connection to the Keil μVision 3 framework. The stored connection handle is later used to access this session again. Also a connection name can be set with this interface function.

- `UVSC_STATUS UVSC_CloseConnection (int iConnHandle, xBOOL terminate)`: closing the connection to the session that is identified via its connection handle.
- `UVSC_STATUS UVSC_ConnHandleFromConnName (char *name, int *iConnHandle)` : if the connection handle is needed it can be received with the connection name of this session.
- `UVSC_STATUS UVSC_GetLastError (int iConnHandle, UV_OPERATION *msgType, UV_STATUS *status, char *str, int maxStrLen)` : used to receive informations on the last error that occured while using this interface.
- `UVSC_STATUS UVSC_LogControl (int iConnHandle, xBOOL enableRaw, xBOOL enableTrace)` : setting up the log control of the μVision socket connection.
- `UVSC_STATUS UVSC_GEN_UVSOCK_VERSION (int iConnHandle, UINT *pMajor, UINT *pMinor)` : returning the version number of the μVision socket interface.
- `UVSC_STATUS UVSC_GEN_HIDE (int iConnHandle)` : used to hide the main window of μVision.
- `UVSC_STATUS UVSC_GEN_SHOW (int iConnHandle)` : showing the μVision IDE window.
- `UVSC_STATUS UVSC_GEN_MAXIMIZE (int iConnHandle)` : fitting the μVision IDE window to the maximum screen size.
- `UVSC_STATUS UVSC_GEN_MINIMIZE (int iConnHandle)` : minimizing the μVision window.
- `UVSC_STATUS UVSC_GEN_RESTORE (int iConnHandle)` : used to show the μVision IDE window again.
- `UVSC_STATUS UVSC_PRJ_LOAD (int iConnHandle, PRJDATA *pProjectFile, int projectFileLen)` : by providing the path to a μVision project file a certain project can be loaded.
- `UVSC_STATUS UVSC_PRJ_CLOSE (int iConnHandle)` : closing the currently active μVision project.
- `UVSC_STATUS UVSC_PRJ_BUILD (int iConnHandle, xBOOL rebuild)` : building the currently activ μVision project.
- `UVSC_STATUS UVSC_PRJ_CLEAN (int iConnHandle)` : cleaning up the currently active μVision project.
- `UVSC_STATUS UVSC_DBG_ENTER (int iConnHandle)` : starting the debug mode of μVision. In this mode all debug methods can be accessed and used.
- `UVSC_STATUS UVSC_DBG_EXIT (int iConnHandle)` : this function is used to exit the debug mode of μVision.
- `UVSC_STATUS UVSC_DBG_STEP_HLL (int iConnHandle)` : used to step one code line in the high level software description.

- `UVSC_STATUS UVSC_DBG_CHANGE_BP (int iConnHandle, BKCHG *pBkptChg, int bkptChgLen, BKRSP *pBkptRsp, int *pBkptRspLen)` : modifying an existing breakpoint.
- `UVSC_STATUS UVSC_DBG_ENUMERATE_BP (int iConnHandle, BKRSP *pBkptRsp, int *pBkptIndexes, int *pBkptCount)` : enumerating all breakpoints in the active µVision project.
- `UVSC_STATUS UVSC_DBG_ENUM_VTR (int iConnHandle, iVTRENUM *piVtrEnum, AVTR *paVTR, int *pVtrIndexes, int *pVtrCount)` : enumerating all virtual registers in the current project.
- `UVSC_STATUS UVSC_DBG_VTR_GET (int iConnHandle, VSET *pVSet, int vSetLen)` : receiving the value of a virtual register.
- `UVSC_STATUS UVSC_DBG_VTR_SET (int iConnHandle, VSET *pVSet, int vSetLen)` : setting the value of a virtual register.
- `UVSC_STATUS UVSC_DBG_WAKE (int iConnHandle, iINTERVAL *piInterval)` : waking up the simulation from sleep mode. This is only possible when working with a simulator.
- `UVSC_STATUS UVSC_DBG_SLEEP (int iConnHandle)` : sets the simulation into sleep mode. This is only possible when working with a simulator.

List of Abbreviations

IDE	Integrated Development Environment
TLM	Transaction-level Modeling Standard
IC	Integrated Circuit
HDL	Hardware Description Language
OSCI	Open SystemC Initiative
DUT	Device Under Test
ICE	In-Circuit Emulator
RTL	Register Transfer Level
GDB	GNU Debugger
NOC	Network-On-Chip
SoC	System on Chip
CDT	C/C++ Development Tooling
MDI	Model Debugging Interface
SFR	Special Function Register
HDM	Hardware Debug Module
CPU	Central Processing Unit
SFR	Special Function Register
VTR	Virtual Register
TCP/IP	Transmission Control Protocol and Internet Protocol
DFI	Debug Functionality Interface

Bibliography

[1] D. J. Agans. *Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems.* American Management Association, http://www.amacombooks.org, 2002.

[2] J. Banks. *Processor description languages: applications and methodologies.* Morgan Kaufmann, 2008.

[3] A. Agarwal et al. Reed solomon decoder. Technical report, Computation Structures Group, 2007.

[4] B. Vermeulen et al. Communication-centric soc debug using transactions. *12th European Test Symposium ETS 07, IEEE*, pages 69–76, 2007.

[5] F. Rogin et al. An integrated systemc debugging environment. *Embedded Systems Specification and Design Languages: Selected contributions from FDL 07, Springer*, page 59 71, 2008.

[6] H. Chun et al. Es-debugger : the flexible embedded system debugger based on jtag technology. *Advanced Communication Technology, ICACT 2005. The 7th International Conference on*, pages 900–903, 2005.

[7] H.A. Ahmed et. al. A low energy high speed reed-solomon decoder using decomposed inversionless berlekamp-massey algorithm. *Systems and Computers (ASILOMAR),IEEE*, 2010 Conference Record of the Forty Fourth Asilomar Conference on Signals:406–409, 2010.

[8] I. Huang et al. A retargetable embedded in-circuit emulation module for microprocessors. *Design & Test of Computers, IEEE*, Volume: 19 Issue: 4:28–38, 2002.

[9] J. Castillo et al. Systemc design flow for a des/aes cryptoprocessor. *WSEAS Transactions on Information Science and Applications*, pages 193–198, 2004.

[10] J. Castillo et al. An open-source tool for systemc to verilog automatic translation. *Latin American applied research*, vol.37:53–58, 2007.

[11] L. Lavagno et. al. *Electronic Design Automation For Integrated Circuits Handbook.* CRC Press, 2006.

[12] M. Lackner et al. Design and implementation of a system level 8051 microcontroller in systemc using tlm-2.0. Technical report, Institute for Technical Informatics, Graz University of Technology, 2010.

[13] N. Hatami et al. Tlm 2.0 simple sockets synthesis to rtl. *Design and Technology of Integrated Systems in Nanoscal Era 2009, 4th International Conference on*, pages 3–8, 2009.

[14] P. Mishra et al. *Processor Description Languages*. CRC Press, 2006.

[15] S.L. Coumeri et al. A simulation environment for hardware-software codesign. *Computer Design: VLSI in Computers and Processors, IEEE International Conference on*, pages 58–63, 1995.

[16] Free Software Foundation. *Debugging with gdb: the gnu Source-Level Debugger*. http://sourceware.org/gdb/current/onlinedocs/gdb.html, 2010.

[17] Open SystemC Initiative. Osci tlm-2.0 language reference manual. Technical report, Open SystemC Initiative (OSCI), 2009.

[18] L. Jain. Nirgam: A simulator for noc interconnect routing and application modeling. Technical report, University of Southampton UK,Malaviya National Institute of Technology India, 2007.

[19] J. Langer. *Testing, tracing und debugging bei Embedded Systems*. Trauner Verlaf, www.trauner.at, 2008.

[20] J. R. Larus. Efficient program tracing. *Computer, IEEE*, Volume 26 , Issue 5:52–61, 1993.

[21] H.Yue li et al. Design of an embedded on-chip debug support module of a mcu. *High Density Microsystem Design and Packaging and Component Failure Analysis, HDP'06. Conference on*, pages 5–8, 2006.

[22] M. Lindahl. The device software engineer's best friend. *Computer*, vol. 39, no. 5:95–97, 2006.

[23] H. Muhr. Einsatz von systemc im hardware/software-codesign. Technical report, Institute of Industrial Electronics and Material Science, Vienna University of Technology, 2000.

[24] s. Beyer et al. Generating an efficient instruction set simulator from a complete property suite. *IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 109–115, 2009.

[25] R.R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34 Issue:6:52–59, 1997.

[26] Keil Software. Application note 154: Implementing uvision dlls for advanced generic simulator interface. Technical report, Keil Software, Inc and Keil Elektronik GmbH, 2001.

[27] Keil Software. Application note 145: Implementing uvision2 interface dlls to hardware debuggers. Technical report, Keil Software, Inc and Keil Elektronik GmbH, 2003.

[28] Keil Software. Application note 198 - using the uvision socket interface. Technical report, KEIL - An ARM Company, 2008.

[29] B. Vermeulen. Functional debug techniques for embedded systems. *Design and Test of Computers, IEEE*, Volume: 25 Issue: 3:208–215, 2008.

[30] W. Wolf. A decade of hardware/software codesign. *Computer, IEEE*, vol. 36, no. 4:38–43, 2003.

[31] W. H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, Volume: 82:967–987, 1994.