Cycle Accurate Simulator Improvements for the FireWorks/SideWorks Architecture

(Master Thesis Extended Abstract)

João Carlos dos Santos Ferreira
Department of Electrical and Computer Engineering
Instituto Superior Técnico - Technical University of Lisbon, Portugal
joao.dos.santos@gmail.com

Abstract— The work realised in the scope of this thesis consisted in the improvement of a cycle accurate simulator for the Fireworks processor, previously developed by the author of this thesis, with the objective of emphasizing the debugging capabilities and performance analysis. The simulator was implemented in Java.

In simulator mode it operates in assembly instruction mode and is cycle accurate, obeying a modular architecture, permitting the usage of processor and peripheral modules previously developed. The simulator supports the software loading in Executable and Linking Format (ELF) format and the configuration of the system to simulate through Extensible Markup Language (XML) files.

In debugger/GNU Debugger (GDB) server mode, the protocol GDB Remote Serial Protocol (GDB RSP) is supported and operates as a GDB server.

In performance analyser mode, it produces performance reports in terms of instructions and in terms of function of the simulated software.

To assess the correct operation and performance of the developed simulator, various test programs simulations were made.

Index Terms— Microprocessors, Executable and Linking Format (ELF), Simulator, Debugger GNU Debugger (GDB), Performance analyser.

I. INTRODUCTION

A. Motivation

Since the beginning of this century there has been a constant growth in the use of embedded systems, continually posing new challenges to the development of innovative solutions and new applications. The embedded systems are found in all aspects of contemporary life, there are many examples of its use in telecommunications and computers, medicine and offices, industry, defense and space, as well as consumer electronics and automotive industry. The market for embedded systems is extremely competitive with a large impact on the economy. The quality, innovation, functionality and price are the main factors of choice of an embedded system.

The profits obtained through the business of embedded systems provide both from the hardware and software. The hardware consists of system boards and integrated circuits; software consists of development tools, applications and operating systems. Although the majority of the revenue comes from the integrated circuits, the highest growth rate of revenue comes from software. The value added by embedded software to the end product is generally of a higher cost than the device itself [1].

Many embedded systems used in machines which are expected to continuously operate for years, are used in critical systems or even in hard to reach operating machines such as robots and space satellites. For these reasons the software of embedded systems is usually developed and tested more carefully than the software for personal computers.

This thesis focuses on the improvement of an accurate cycle simulator with new features to simplify the development of embedded software for systems, based on the FireWorks architecture from Coreworks [2], in particular the features of debugging and performance analysis.

B. Context

The work developed within this thesis was carried out in collaboration between the author of the thesis and the Portuguese startup technological company Coreworks, owner of FireWorks processor [3]. The author aimed to obtain the academic master's degree in electrical and computers engineering at Instituto Superior Técnico (IST), supporting the Coreworks company in its effort to develop tools and applications for its reconfigurable architecture FireWorks/SideWorks [4].

C. Objectives

This thesis aimed for the improvement of an accurate cycle simulator for supporting the development of software for embedded system. Although the simulator can be easily adapted to systems with other processors, its development had in mind systems based in FireWorks architecture.

Development requirements:

- The simulator should allow the simulation of the accurate cycle of systems based in the FireWorks architecture;
- The simulator must be developed in Java to facilitate its portability on several operating systems;
- The simulator must be able to read in Executable and Linking Format (ELF) the assembly code of the programs to load;
- The simulator must have an architecture that facilitates connecting peripherals and memories to the system buses that will be simulated, for example, the SideWorks digital signal processor from Coreworks;
- The simulator has to allow the configuration of the entire system, from the processor, the latency of instructions, set the peripherals and memories that are part of the system, and the mapping configuration and latency of memories and peripherals used;
- The embedded system supported by the simulator include memories, Universal Asynchronous Receiver/Transmitter
(UART), and preconceived timers that enable the interconnection to the system and its simulation;

- From the developed simulator make improvements in terms of features allowing the simulator to operate as performance analyser and debugger/GNU Debugger (GDB) server.

D. Applications

The simulator of FireWorks processor based systems designed within this thesis had practical applications. In the following lists there are examples of studies that used or use this simulator:


In both of these works the simulator was used to generate reports with the trace of the programs route. Within the doctoral thesis of João Bispo, a version of the simulator has been used to identify certain instruction sequences in traces of execution.

E. Outline

This paper is organized as follows: Section II: Presents the architecture and features of the FireWorks processor and the systems based on it. Here are presented the simulation tools, debugging and the environments currently used in the implementation and development of the software and hardware of systems based in the FireWorks processor. Section III: Is devoted to the accurate cycle simulator developed in this thesis. Here are explained the reasons for the development of a simulator of this type and described its features, functions and architecture. Section IV: The improvements made to the simulator and the reasons for such improvements are presented. Here are described the changes made in the simulator considering new modes of performance analysis operation and debugger/GDB server. Section V: The experimental results tests of the simulator tests are provided. Section VI: Are presented the conclusions and future works that can be developed from the work developed in this thesis.

II. FIREWORKS PROCESSOR-BASED SYSTEMS

A. FireWorks Processor

The FireWorks processor is developed by Coreworks [2] with reference to the 3.0 version of the OpenFire processor [5], to be used in embedded systems. It is synthesizable and optimized for implementation in Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC). Its features allow the design of systems taking into account specific requirements as well as the cost/performance.

The following list shows the characteristics shared by FireWorks and OpenFire processors:

- Reduced Instruction Set Computing (RISC) architecture;
- Pipeline architecture;
- 32 bits instructions;
- Instruction Set Architecture (ISA) identical to the MicroBlaze processor from Xilinx;
- Thirty two general purpose registers of 32 bits;
- Architecture totally orthogonal;
- Harvard architecture;
- 32 bits addressing bus;
- Big-endian format is used for data representation and addressing;
- Without Memory Management Unit (MMU);
- Without caching system;
- Without power manager.

In Figure 1 the block diagram of FireWorks processor-based system.

1) Instructions

One of the features that the FireWorks processor shares with the OpenFire processor is the support of the same ISA. The supported ISA is identical to one from Xilinx MicroBlaze v6.00 processor, with the exception of the instructions that operate with breaks, with the cache, with the Fast Simplex Link (FSL) interface and the Floating Point Unit (FPU); all other instructions are supported and fully implemented.

The instructions are grouped into the following five categories: arithmetic, logical, branches, special and reading/writing. These are represented by 32 bits and a set in one of two formats: Type A or Type B.

Type A is used for register - register instructions. Contains the operation code, a destination register and two source registers.

Type B is used for register - immediate instructions. Contains the Ocode, one destination register, an origin register and an immediate source value with 16 bits. The immediate value can be extended to 32 bits, but it is necessary to precede the Type B instruction with the \texttt{LDC} instruction.

2) Registers

The processor architecture is totally orthogonal, that is, the instructions have the same base format, allowing the use of any register and addressing mode. Contains thirty-two general purpose registers of 32 bits, numbered from R0 to R31 and eighteen special usage registers of 32 bits.

3) Pipeline

The processor comprises a pipeline architecture divided into three-stages: Fetch (IF), Decode (OF) e Execute (EX).
Most instructions take three clock cycles to be executed, although the pipeline operation enables to achieve close to one clock cycle per instruction performance. There is a minority of instructions that do not obey to the previous situation. These specific instructions occupy multiple clock cycles on the Execute (EX) stage. For this to be achieved the pipeline operation is blocked until the instruction on the Execute (EX) stage is terminated; this block is known as stall.

On each stage there is an internal register Program Counter (PC), where the address value of the instruction to execute is stored.

Pipeline stall, branches and data forwarding [6] are automatically resolved by the processor.

4) Memory Architecture

The FireWorks processor is based on the Harvard architecture, which means that the memory access is made via two buses: the data and the instructions. Both buses utilize 32-bits addressing, allowing the 4 Gbytes of address per bus (data and instructions). The overlap of the buses is possible. For such to happen the mapping must be performed on the same physical memory.

The data bus of the processor supports three kinds of memory access by different sizes: byte (8 bits), halfword (16 bit) and word (32 bits).

The instructions bus only supports the word (32 bits) memory access.

The big-endian format is used in both the address and data representation, that is, the bytes are ordered by descending order of their weight in successive memory locations.

Peripherals used in embedded systems with the FireWorks processor are mapped in areas of data addressing and accessed by its data bus.

5) Reset, Exceptions and Interrupts

The FireWorks processor supports the following exceptions: reset, interruptions and exceptions. The processing of these exceptions is carried out taking into account their priority. So they are ordered in descending as follows:

- Reset;
- Hardware exception;
- Interruption;
- Software exception.

Certain memory locations are used to store vectors for the processing of exceptions. The vector is no more than the address (direct or indirect) of a portion of code for which the program execution is diverted to the processing of the respective event.

B. Memory and Peripherals

1) Standard Bus Interconnects

The FireWorks processor uses two buses for memory access; a bus is used to access the data and another to access the instructions. The buses used are: Local Memory Bus (LMB) and On-chip Peripheral Bus (OPB).

The LMB bus is simple, efficient, synchronous, ensuring access in a single clock cycle to Block Random Access Memory (BRAM) type memories and single-master bus. This bus is only used for connections to the fast internal memory of the system, which is comprised of BRAM type memory. The processor comprises two LMB buses, one is used for the instructions memory access and the other for the data memory.

For connecting peripherals and external memory to a FireWorks processor is used the 2.0 version of the OPB bus. This bus is 32-bit multi-master multi-slave. The external memory is intended only for data memory; stored instructions in this type of memory cannot be fetch directly by the processor.

2) Memory

For a system based on the FireWorks processor can use the external memory the usage of a memory controller is required. The controller function to access and manage memory accesses and it’s placed between the OPB bus and external memory. Following are some types of supported controllers:

- Flash memory controller;
- Static Random Access Memory (SRAM) memory controller;
- Synchronous Dynamic Random Access Memory (SDRAM) memory controller;
- Double Data Rate Synchronous Dynamic Random Access Memory (DDR SDRAM) memory controller;
- BRAM memory controller;
- Zero Bus Turnaround (ZBT) memory controller.

3) Peripherals

The peripherals may comprise control registers, status registers, data registers and memory, etc. To be able to access these resources is necessary to map them in the OPB bus of the data bus. All mapped registers have the word size (32 bits).

There is a wide range of peripherals that can be used in a FireWorks system. Following are listed some of the existing peripheral features:

- Controller interruptions (is being developed);
- Ethernet Media Access Control (MAC);
- Universal Asynchronous Receiver/Transmitter (UART);
- Serial Peripheral Interface (SPI);
- FPU;
- Digital Signal Processing (DSP), for example, the SideWorks digital signal processor;
- Inter-Integrated Circuit (PC);
- Temporisers/Counters;
- Memory controllers.
C. Environments and Tools for Simulation and Debugging

The FireWorks processor was designed to take full advantage of the development tools designed by Xilinx for the MicroBlaze processor.

Xilinx [7] provides the development package Embedded Development Kit (EDK) for the implementation of systems based on the MicroBlaze processor on Xilinx FPGAs. The EDK package consists of a range of tools that enable the development of the hardware and the software. The EDK is available for Windows, Solaris and Linux platforms.

The entire development process of the system hardware and the FPGA implementation follows a sequence of steps. The first step is the synthesis of the system from its description in VHDL Hardware Description Language (VHDL) or Verilog. Carried out mapping, placement and routing, and the generation of the FPGA configuration (bitstream) in the next step. In the last step we proceed to load the bitstream generated in the FPGA.

There are two ways to perform the steps of the software and hardware development process with EDK: one manual and another through the Xilinx Platform Studio (XPS) tool included in EDK package. In the manual, all files and scripts used in steps of the development process have to be created and edited by the user, requiring advanced knowledge to do so. The XPS tool acts as a graphical user interface over the EDK and allows a simpler and interaction with the user in the process of development and implementation of the hardware. Through the XPS it is possible to create and configure a system in totality: configure the MicroBlaze processor, add Intellectual Property Blocks (IP Blocks) previously designed by the user or libraries, interconnect the components of the system, making the loading of the software, automatically generate the set of scripts necessary to perform the synthesis and implementation, and proceed to load the bitstream generated in the FPGA.

For programming and debugging software, the EDK package includes a modified version of the standard GNU toolset, known as GNU Toolchain [8]. The changes made are designed to adapt and provide additional options that are specific of the systems based on the MicroBlaze processor. This toolkit contains the compiler C/C++ GNU Compiler Collection (GCC) [9] and the GDB [10], among other tools. With the function of debugging the software loaded on the physical system, the EDK package includes the Xilinx Microprocessor Debugger (XMD) tool. With the XMD tool it is possible to connect the computer where the development of programs is made to the MicroBlaze microprocessor, physically implemented on an FPGA development board via interface Joint Test Action Group (JTAG), thus enabling the loading of the software in the system memory, monitoring the implementation of software, access to processor registers, among other features.

Also allows the debugging through the simulator of the cycle accurate MicroBlaze processor Instruction Set Simulator (ISS) available in EDK package.

The user can use the graphical version of GDB client, also available in EDK package to connect to the XMD tool, thereby allowing further simplification of interaction loading, running and debugging the software.

D. Documentation

Until the completion of the development tool to design within this thesis there was no documentation by the Coreworks regarding the FireWorks processor. During the development of the tool was used the documentation related to the OpenFire processor and the MicroBlaze processor and the technical support by Coreworks employees.

The documentation available for the MicroBlaze processor and its development tools are very informative, can be obtained from Xilinx website [7]. The documentation for OpenFire processor is available on the internet in the webpage from the project of the processor [5]. Though limited, for more detailed information consult the documentation related to the MicroBlaze processor. The source code written in Verilog, also available on the project website, is a source of information to consider.

The FireWorks processor and the MicroBlaze processor support exceptions and interrupts something that OpenFire processor does not support. However, the documentation of these features is incomplete, thus making it very difficult to develop a simulator or debugger to an independent MicroBlaze processor [11].

III. SIMULATOR

A. Why Simulate?

The most used method for debugging in embedded systems software is to proceed with his execution controlled directly on the system hardware. Monitoring of implementation is carried out using a Personal Computer (PC) connected to the embedded system via a communication interface. But for the control to be possible is necessary to add to the software one monitoring application (debug stub) when creating the executable. This application has the function to communicate with the PC, proceed to the internal control of the implementation of the software, read and write registers in the processor, and other features.

This debugging method brings inconveniences. Due to the extra presence of the monitoring application, the software is loaded in different memory locations for which it was designed, which can cause changes in the normal performance of the software, such as access to different memory locations for which it was programmed, which may hide other errors. Another drawback of this method is not to provide full control over the execution environment, if the execution is interrupted for some reason and is subsequently triggered an exception will not be handled by software, another consequence is to be hard to replicate the occurrence of certain types of exceptions.

The use of simulators for the detection of errors in software allows to overcome these drawbacks and to add functionality not attainable by the previous method. An accurate cycle simulator, focused on embedded systems is a program that performs partial or complete modeling of all aspects of the hardware on personal computers. Allows the modification and access to all the states of a system and the control execution, without the use of monitoring applications, namely, non-intrusive way. This feature allows its use for detection of errors in the software running on the simulator without the possible consequences caused by the use of a monitoring application or non-processing of exceptions due to stops in software implementation.

The use of simulators has implications not only on the software level. The implications are also felt on the hardware level, because it allows the evaluation of different projects without the expense in building physical systems.

Although the initial cost of developing a simulator, its use allows access to non-existing systems or peripherals and the
development and testing in advance, dramatically reducing the duration of the project and significantly reducing the initial costs of development and maintenance of the software for embedded systems.

B. Description of the Simulator

The simulator currently used in the simulation of systems based on FireWorks architecture is the simulator of systems based on MicroBlaze architecture available in the package EDK from Xilinx. The simulator is ISS, accurate cycle and does not allow access to peripherals. Their use is only possible through the XMD tool and its source code is not available to the public. The software to simulate can only reside in memory mapped in the LMB and its size sets the maximum memory size.

Under this thesis is proposed a fully designed simulator developed in Java to facilitate its portability between different operating systems and to enable a modular architecture by using dynamic class loading mechanism. Its design allows the loading of executable files in ELF and operating autonomously, not depending on existing tools. The source code is freely accessible thus enabling future modifications.

The simulator performs the modeling of the overall system. Both the functional point of view and from a temporal perspective. The temporal modeling allows the simulator to be accurate cycle and run the simulation cycle to cycle. This way allows access to all the states of the system, obtain results of the software performance, number of instructions and cycles processed.

At startup the simulator loads an XML file with all the system configuration to simulate. Through this file you can set the processor and the peripherals that constitute the system, configure the latency of instructions, mapping and latency of each peripheral registers and define the type, latency, size and mapping of memories.

In terms of the simulator architecture the processor and the peripherals are not part of the simulator, but separate modules. Your upload is done during the simulator startup as the system configuration to simulate. This feature allows the further development of new modules, both processors and peripherals.

In order to demonstrate and assess the simulator functionalities was designed a system based on the FireWorks architecture, comprising a FireWorks processor with three-stages pipeline, one peripheral OPB Timer/Counter [12] and a peripheral OPB UART Lite [13], with partial modeling.

C. Simulator Architecture

Systems based on the FireWorks architecture can have different configurations, depending on the intended application. Given this characteristic, the development of the simulator had in mind a modular architecture.

The structure of the simulator consists of two parts, the core and the modules. The core has three functions:

- Simulator configuration, proceeding to the desired system configuration via an XML configuration file and loading the executable ELF from the software to simulate;
- Simulation engine operation;
- System memory functional and temporal modulation.

The modules define and shape functionally and temporally the peripheral and the processor.

1) Configuration

To be able to perform the simulation of the desired system, first is required to configure the simulator. This action occurs during the execution initialization and is composed of four phases.

In the first phase it’s carried out the loading and validating of the system configuration XML file. In this file is set the processor and the latency of its instructions, the peripherals and the mapping and latency of its registers, the memories defined as to type, mapping and latency. The XML file is validated and verified the consistency of the memories mapping and the peripherals of the system to simulate with the FireWorks architecture.

The second phase consists in the loading and validation of XML configuration files of the peripherals modules and processor, which are referenced in the system configuration file.

In the third phase is carried out the system configuration. The memory is initialized with the definition of its boundaries for both the LMB bus and the OPB. The processor started up is performed by the dynamic loading of the corresponding modules classes and by registering the ISA in the central core memory modeling system, in order to decode the instructions. Peripherals are initialized by performing the dynamic loading of the corresponding modules classes and by allocating the registers in the system memory.

In the fourth and final phase it’s performed the validation and loading of the ELF executable file to simulate.

2) Simulation Engine

The control and execution of the whole system simulation are performed through the simulation engine. The simulation engine is a state machine in which the transitions are triggered by the processor and the simulation execution performed cycle by cycle.

The simulation engine processes at each system cycle one processing cycle for the processor and one processing cycle for each peripheral in the system. With the processing cycles of the processor and peripherals being performed sequentially it is necessary to keep the simulation temporal context, since these operations occur simultaneously.

The implementation of a written register must contain a temporary attribute and a real attribute. The real attribute is the attribute used for the peripheral internal processing, since this saves the consistent value with the simulation context. When a processor writes in a peripheral writing register the value is stored in the temporary attribute of the register, and when the completion of the peripheral processing cycle occurs is performed the real attribute value updating with the temporary attribute value.

In Figure 2 is shown an UML activity diagram of the simulation engine from the FireWorks system based architecture.

3) Memory

The functional and temporal modeling of the system memory is carried out in the simulator central core. The modeling comprises the LMB bus, the OPB bus and the interconnection with the OPB bus with the peripherals registers, allowing data reading and writing operations and instructions.

In access and data transfer operations, the modeling is carried out by a state machine. The states define the memory behavior during operations carried out by the instructions. The state transitions are related to the access times of different memories
and registers that constitute the system and the way to access the data. Reading and writing methods that implement the various transfers are supported by the FireWorks architecture. Since the loading of instructions by a processor in a system based in the FireWorks architecture can only occur from the memory mapped in the LMB bus, this operation taking one clock cycle, there is no need to formally implement a state’s machine.

The class that models the system memory contains three instances of the class java.util.HashMap: one saves the data, the other stores the instructions and the other saves the classes instances of the peripherals registers. When writing certain data to a memory position in the LMB bus the central core stores the data value in the data class, and from the same data performs the instruction decoding, creating an instance of the instruction class.

4) Processor

The temporal and functional modeling of the processor is implemented in the simulator via a module developed for that purpose. The processors implementation through the modules enables the development of multiple versions of a given processor, varying its features, such as support or no support of interruptions, support or no support of exceptions, number of pipeline stages, ISA, without the need to make any simulator modifications. Modules usage also allows the development of a processor modules library enabling the testing of several hardware solutions for a given application.

The modeling of the processor operation is performed both temporally and functionally via a state’s machine. The state’s transitions are related to the instructions execution. For this the instructions behavior is modeled in terms of cycle and pipeline. The processor modeling supports the interruptions and exceptions processing.

The processor instructions in the simulator are implemented through classes, one class for each instruction. The instruction classes are instantiated by inheritance. The processor performing the loading of a given instruction does not perform its decoding, since this is already pre-decoded in the central core memory system of the simulator.

In the simulator the processor is not only composed by instructions belonging to the ISA, it’s also formed by special instructions. The special instructions are intended to define states that the processor can operate and are not triggered by the ISA instructions.

The class that defines the instruction contains reference to a class of the same type. This feature allows an instruction to have the same behaviour of other instruction. This feature is used in the implementation of special instructions, has the same behaviour of the instruction referenced but cause different transition state.

5) Peripherals

In a system based on the FireWorks architecture the number and type of peripherals vary depending on the configuration used. The temporal and functional modeling is implemented through modules. The use of modules allows the development of a peripheral modules library, thus enabling the testing of several solutions and hardware configurations.

Peripherals can contain multiple registers. During the configuration process, the dynamic loading of peripheral module classes the classes concerning registers are allocated in central core memory system.

In Figure 3 is shown the UML activity diagram of the processing of one cycle concerning a generic peripheral with the feature to generate interruptions. In the first phase is performed the peripheral processing. In the second phase are performed the written registers modified by the processor updates, thus maintaining the context of the simulation. In the last phase the peripheral interruption is generated if the conditions are fulfilled.

IV. IMPROVEMENTS

A. Performance Analysers Mode

1) Motivation?

Depending on the function for which given software is designed, there may be specific requirements regarding the performance that necessarily has to be long. If these requirements are not achieved the software might not properly perform the function for which it was developed. For embedded systems, due to limited resources and the type of applications they are intended, this is extremely important, as is the case of multimedia systems and real-time. To perform the validation of these requirements refers to the use of a development tool, known as performance analyser.

The performance analyser is a tool that collects information about the behaviour of a program during its execution, with the aim of measuring the frequency and duration of calls to functions, and thus determine which sections of the software should be optimized.

With the information gathered is possible to know whether the number of times the functions have been performed were as...
expected and can therefore detect possible errors whose origin could be hard to detect otherwise. As the performance analyser uses information gathered during the program execution, can be used in very large and complex programs whose source code is extensive and difficult to analyse. It is a dynamic form of analysis, as opposed to other forms of static analysis of the source code.

The performance analysers use a wide variety of techniques for data acquisition, which includes:

- Instrumentation: insert code in the program for data acquisition;
- Statistical: use sampling methods for data acquisition;
- Exceptions: exceptions used for data acquisition;
- Simulator: data acquisition using an instructions simulator.

All these techniques have their advantages and disadvantages, their usage depends on the purpose it is intended and the answers to be reached. There are analysers that use a mixture of some of these techniques to collect the information.

2) \textit{Description of the Performance Analyser Mode}

For the performance analysis the tool currently used is the gprof\cite{14} from the GNU. The gprof provides performance analysis by sampling and intrusive. In order for this tool to generate the results of performance analysis is necessary to add code to the software to analyse to perform data acquisition. The inclusion of the code for performance in software data acquisition can cause different responses for which it was programmed. Additionally, the use of the sampling method for data acquisition leads to the inaccurate results.

With the features of the simulator already mentioned above, the ability to identify which instruction was executed and which memory location occupies, the ability to perform the count of the number of cycles performed and the support for loading executable files from the ELF\cite{15}, has been developed the performance analyser operation mode.

From the information contained in the ELF executable file regarding the functions that compose the software, was developed a new simulation engine with the ability to carry out the count of functions and instructions executed and the number of cycles performed. Thus carried out the data acquisition of performance to the simulator level, namely, non-intrusive, without the need to enter code to do this acquisition. Another consequence of the data acquisition of performance to the simulator level is the accounting of all occurrences, thus obtaining accurate data.

The results are presented through a performance analysis report based on the format of the reports produced by the GNU gprof tool and provided to the user as a text file. The report consists of four sections: flat profile, general profile, instruction profile and function profile.

3) \textit{Performance Analyser Mode Architecture}

The architecture of the performance analyser mode is generally identical to the architecture of the simulator mode, except for the configuration and the simulation engine.

In the configuration the difference resides in the phase of loading the executable file in ELF format. In performance analyser mode is not only carried out the loading of the software executable code; is also performed the loading of the information describing the functions that belong to the software. This information is also available through the ELF file, contains references to all functions as well as the memory space they occupy.

From the simulation engine was developed a new simulation engine for performance analyser mode in order to enable the counting of cycles and instructions executed per function of the simulated software. All the data collection is performed at the simulator level and not at the simulated software level, so non-intrusive. The method developed for the data collection in the simulation engine allows the counting of all occurrences, which allows accurate analysis.

At the end of each processed cycle and if an instruction was executed, the address of the instruction is crossed with the information provided by the ELF file about the functions in order to acknowledge to which functions belongs the instruction. The result is stored in tables for later processing. At the end of the simulation the data is processed and the result is provided as a analyses report of the software in text format performance.

B. Debugger/GDB Server Mode

1) \textit{Motivation?}

In the software development process for embedded systems, there is much time spent correcting errors, validation and elimination of defects in the program. This procedure it’s called debugging.

Debugging is time consuming and with some degree of complexity. All these factors depend on the development environment used, which are part of the programming language and development tools, which include the debugger.

The debugger is a computer program whose function is to verify and control the execution of other programs in order to detect their errors and defects. The control allows to examine/modify values of registers and memory, set positions in the program where the execution should be suspended (breakpoints), reboot or rewind the execution, etc. All these characteristics vary from debugger, its characteristics can vary depending on the system to which it is implemented.

The importance of these tools can be a decisive factor for the choice of a given system or programming language. Its absence or existence of low capacity may lead to the same situation to omit a system or programming language best suited for the intended task, to other less suitable.

One should not underestimate the impact of a tool like a debugger on an embedded systems software design project. This causes the decrease of time spent in development and support, and reduces the number of errors and defects of the program. Reducing development costs and increasing the quality of support and the final product.

2) \textit{Description of the Debugger/GDB Server Mode}

Currently software debugging for the systems based in FireWorks architecture are used two tools, the \textit{mb-gdb} tool and the XMD. The operating principle of both tools follows a client/server architecture with the communication made through the GDB Remote Serial Protocol (GDB RSP)\cite{10} on the Transmission Control Protocol/Internet Protocol (TCP/IP). The \textit{mb-gdb} tool is an adapted version of the client GDB tool for the MicroBlaze processor. The XMD tool acts as GDB server. Both tools run on the PC being connected to the hardware (FPGA Board) by a Universal Serial Bus (USB) cable from the PC side, attached to a physical USB converter/Joint Test Action Group
(JTAG) which in turn is connected to the JTAG connector on the FPGA board. This assembly is illustrated in Figure 4.

In simulator mode the XMD tool in conjunction with the MicroBlaze processor of accurate cycle simulator ISS from Xilinx allows non-intrusive debugging. As already stated, the simulator of this approach has the following limitations:

- You cannot access peripherals;
- The software can only reside in the internal memory of the system;
- The size of the software sets the maximum memory size that the simulator can access;
- It is only possible through the use of XMD tool;
- Its source code is closed.

For the mb-gdb tool can provide debugging tool at the level of source code, the executable ELF should contain information in Debug With Attributed Record Format (DWARF), version 2 [16] format. This information is added automatically during the compilation of the software through the –g option.

One of the changes made to the simulator was the elimination of the XMD tool in the debugging software process. The change was implemented by designing a new operating mode of the simulator: debugger/GDB server mode. The architecture used continues to allow the usage of the tool mb-gdb GDB client in the debugging process. The client/server architecture is maintained: the mb-gdb tool continues to operate as GDB client and the simulator in debugger/GDB server mode operates as server. Interconnection is also carried out by the GDB RSP protocol over TCP/IP protocol. In Figure 5 illustrates the interconnection between the simulator and the mb-gdb tool.

The commands implemented from protocol GDB RSP enables a normal execution of the software, the execution instruction by instruction, the insertion and removal of breakpoints, and viewing and modification of registers, memory and variables.

The simulator was adapted to allow the software stops running through breakpoints. The technique used is made to the simulator of a non-intrusive way, since there are no changes to the software, this technique has no restrictions on the number of breakpoints that can be used in a system. An advantage of this technique is to suspend all the simulation of a breakpoint system, thereby eliminating the possibility of untreated exceptions occurs.

3) Debugger/GDB Server Mode Architecture

The architecture of the debugger/ GDB server mode, is a GDB server type integrated with the FireWorks simulator.

The implementation of the GDB server consisted in the development of an communication interface between the mb-gdb tool and simulator.

With the exception of the final phase, the whole procedure of the simulator in debugger/GDB server mode configuration is performed the same way as configuration in simulator mode. The phase intended to perform the loading of the executable file in ELF was removed in the debugger/GDB server mode, since the mb-gdb tool already includes this feature. A new phase was added to the end of configuration to perform the communications setup and put the interface into standby at the end.

Two of the implemented protocol commands GDB RSP are related to the software execution control. One package makes the continuous execution of the software stopping at breakpoints. The other package carries only the execution of one software instruction.

For the simulator to operate in both these two ways the architecture related to the simulation engine has been modified to support two simulation engines. A motor for continuous execution and another for the execution of a single instruction. The architecture was redesigned to allow the use of several simulation engines, but only one can operate at the same time. The architecture allows during the software execution stoppage to exchange the simulation engine, this swap is only possible to achieve because the simulator saves every moment the whole context of the simulation system.

Because the instructions are causing the change in the processor state, it was necessary to develop a new special instruction, the breakpoint instruction, it takes the processor to the breakpoint state. The state was added to implement the stoppage of the system in a breakpoint while executing the software. This causes the stoppage of the entire system.

The insertion and removal of breakpoints in the system memory are incurred by the interface between the GDB server and the simulator, therefore sending commands from the mb-gdb tool. This type of command contains information about the memory position where the breakpoint should be placed. When removing a breakpoint the interface puts in its memory position the referenced instruction by the breakpoint instruction.

V. EXPERIMENTAL TESTS

A. Gauging of the Simulator

Due to the complexity of implementing a simulator of a microprocessor-based system it is not possible to prove its functionality. It is only possible to give a certain degree of confidence about their operation through benchmarking tests. These tests consist of performing tests in the simulator with reference to the real system. That is, a set of appropriate tests are executed in the real system and in the simulator, the results are compared in order to detect errors. With the absence of errors increases the confidence in the functionality of the simulator.

The real system used in the context of this thesis was provided by Coreworks through a development board FPGA from Digilent [17]. The board model provided Spartan-3 Board consists of the

![Fig. 4. Interconnection between the hardware and XMD and mb-gdb tools.](image)

![Fig. 5. Interconnection between the simulator and the mb-gdb tool.](image)
Spartan-3 FPGA from Xilinx [7]. The FireWorks system was configured as follows:

- Three-stages pipeline FireWorks processor;
- Peripheral OPB Timer/Counter;
- Peripheral OPB UART Lite;
- 64kBytes fast internal memory, BRAM type;
- 64kBytes of external memory, SRAM type;
- System clock frequency 50MHz.

All test programs used in the simulator tests were compiled with the mb-gcc 3.4.1 version of the GCC present in the Microblaze GNU Toolchain (Xilinx EDK 8.1.0i).

The benchmarking tests are divided into two groups. The first, made up of eight algorithms and the second of five versions of the Dhrystone Benchmark differentiated by the level of optimization of the generated executable code. The Dhrystone benchmark is used because is one of the most used benchmark by the industry. Table I shows the algorithms used in the tests.

In order to validate the benchmarking tests, the tests conducted for this purpose should produce information both at a functional and temporal level. In the case of algorithms that requirement is not completely satisfied, especially regarding the temporal level. In order to get around this situation the algorithms programs have suffered some changes. In the main() function of each algorithm a system of counting clock cycles was introduced, the count begins to be performed before the beginning of the execution of the algorithm itself and ends after its completion.

By comparison between the results obtained in the tests on the simulator and the real system under the benchmarking tests we can conclude that the results are equal, thus validating the operation of the simulator for this set of tests.

B. Processor Functional Tests

The goal of processor functional testing is to demonstrate the implementation and operation of interruptions and exceptions modulated in the simulator.

The results of processor functional tests gave the expected results. The interruptions test is just a clock with the processor to treat and perform the counting of the number of generated interruptions by the peripheral OPB Timer/Counter. The peripheral OPB Timer/Counter was configured to generate one interruption every second. To test the exceptions was used the Exceptions test, one exception of each type supported is caused by the test program and treated by the processor.

C. Performance Analysers Mode Tests

The purpose of testing the performance analyser mode is to show the operation of the simulator to gather detailed and accurate data for the analysis of performance. And to produce a performance report with the same data.

The tests showed the simulator capacity to generate detailed performance reports without the need of insert code in the program to analyse.

D. Debugger/GDB Server Mode Tests

The purpose of testing the debugger/GDB server mode is to demonstrate the functionality of the simulator when applied in software debugging. In order to perform the tests two programs were designed for this purpose. Each program has two versions differentiated by the configuration used in the compilation. In both versions the build configuration used in the test of the other modes of operation of the simulator program was used. Being in one added to that configuration the compile option –g, as previously mentioned this option gives indication to the compiler to add debugging information to the program.

The simulator in debugger/GDB server mode works as expected allowed to the client program mb-gdb the control of the execution of the tested programs. All commands implemented in the simulator with the exception of the command printf (returns the register value n...) have been successfully tested. It was not possible to test the command because the client program mb-gdb in all tests never performed with the command.

The test programs versions without the debugging information despite having worked properly only allowed the monitoring of the execution through the assembly code. While test programs versions with debugging information enabled the monitoring of the execution both through the assembly code and though the source code written in C language. As shown by the sizes of the executeables of both test programs versions adding debugging information in a program will not cause any code insertion, thus adding debugging information will not influence the proper functioning of the same.

The intention to carry out tests with the interruptions program is to test if the simulator in debugger/GDB server mode when making a stoppage to the software execution due to a breakpoint the entire system operation is suspended, including peripheral (in this case the peripheral OPB Timer/Counter). Thereby eliminating the possibility of exceptions untreated occurs.

The behaviour of the simulator in debugger/GDB server mode to run the test interruptions program was as expected. A breakpoint in order to stop the program execution after reaching the count of the first minute was inserted. When reached the breakpoint the simulator suspended the execution of the program, stopping the processor and peripherals that are part of the system.

VI. CONCLUSIONS AND FUTURE WORK

A. Conclusions

The work within this thesis was to improve an accurate cycle simulator for FireWorks architecture based systems.

A study on the benefits of using simulators was presented, analysing the simulators currently used in order to propose a simulator with a flexible architecture, enabling the configuration of the desired functionality.

The architecture designed enables the usage of processors and peripherals modules later developed, and performs the modeling of the overall system. The simulation is carried out at the instruction level, with accurate cycle, which allows to obtain performance results. The software is loaded on the ELF, and the system can be configured using XML files. The source code is open and allows future modifications.

<table>
<thead>
<tr>
<th>ALGORITHMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autocorrelation</td>
</tr>
<tr>
<td>Discrete Cosine Transform Algorithm</td>
</tr>
<tr>
<td>Fibonacci Number</td>
</tr>
<tr>
<td>Intel/DVI ADPCM Coder</td>
</tr>
<tr>
<td>Intel/DVI ADPCM Decoder</td>
</tr>
<tr>
<td>Quicksort Algorithm</td>
</tr>
<tr>
<td>Tak Benchmark</td>
</tr>
<tr>
<td>Tower of Hanoi</td>
</tr>
</tbody>
</table>

TABLE I. THE EIGHT ALGORITHMS USED IN THE TESTS
The MicroBlaze processor of accurate cycle ISS contains the following limitations:

- You can’t access peripherals;
- The software can only reside in the internal memory;
- The size of the software sets the maximum memory size that the simulator can access;
- It is only possible through the usage of the XMD tool;
- Your source code is closed.

Improvements made to the simulator allow the operation as performance analyser and debugger/GDB server in a non-intrusive manner, and enabling the development of accurate values in the analysis of results, without the problems usually found in other methods.

In the performance analysis the tool currently used, gprof, adds code to analyse the software and thus can cause side effects, and performs sample analysis, thereby not obtaining accurate results. The debugging tool currently used, XMD, adds to the software to debug a section of code with the name stub in order to control the execution of the software and make the communication with the mb-gdb client program, which can also cause side effects in the implementation of software.

The results obtained in tests on a real system and on simulator by executing benchmarks adapted for the purpose, allowed to assess and demonstrate the functionality of the designed simulator. The work achieved all the objectives that were proposed and can be enhanced to integrate the development tool package of FireWorks processor based systems.

B. Future Work

Although the thesis objectives have been achieved, various features can be improved and many other added. Regarding the simulation mode, several improvements can be implemented. Following are presented the more relevant to the needs observed at Coreworks:

- The simulation engine can be improved to support systems with multiple processors, operating at different frequencies;
- Working with files stimuli, thus simulating the interaction with the outside world;
- Monitoring the stack of the simulated software in order to debug problems associated with violations of the size of the stack.

The analysis performance mode could be improved to support the partial analysis of programs instead of global analysis now implemented, allowing focus on a particular problem area of the problem. The extent of this mode to multiprocessors systems would also be useful.

The debugger/GDB server mode could be improved to be able to access and view registers of peripherals. The support systems with multiple processors is also useful in this mode.

In order to facilitate its usage, the simulator could be integrated in the development environment Eclipse [18]. Like other development environments, would thus be possible to make the whole process of developing embedded software from a single program: development of the source code, compile, the simulator configuration for the desired system, control the simulator, debugging, analysis performance reports and reports visualization.

It could also adapt the simulator to the dynamics simulator environment Simulink [19], thus allowing the simulation of software embedded in interaction with dynamic systems.

REFERENCES