Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://cas.et.tudelft.nl/

CAS-MS-2009-13

Master of Science Thesis

Design of a Portable and Customizable Microprocessor for Rapid System Prototyping

Tamar Kranenburg

Abstract

Due to the increasing number of processors which are integrated in System On Chips (SOCs) the need for robust, highly configurable processors emerged. Preliminary research showed that commercial processors are not suited for many research projects due to the fact that they are closed source and they can not be modified. Open source processors on the other hand appeared to be of too low quality. Within this thesis a light-weight instruction and cycle compatible implementation of the MicroBlaze architecture called MB-LITE is presented to fill the need for a fast, reliable processor suitable for Field Programmable Gate Array (FPGA) and semi-custom implementation. This was accomplished by using a proven design methodology and using high-level VHDL abstractions.

Experimental results showed that MB-LITE is able to obtain much higher performance than existing open source processors, while using very few hardware resources. MB-LITE can be easily extended with existing Intellectual Property (IP) components due to a wishbone bus adapter and a modular, easily configurable multiplexed memory bus. All components are thoroughly tested for compliance, and their functionality was proven to be correct using timing back-annotated simulations. Continued work on MB-LITE will focus on including the design in a reconfigurable fabric as well as fabrication in a 90 nm semi-custom Integrated Circuit (IC) technology.

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Design of a Portable and Customizable Microprocessor for Rapid System Prototyping

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Tamar Kranenburg
born in Naarden, The Netherlands

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

TU Delft

Delft University of Technology

Copyright © 2009 Circuits and Systems Group
All rights reserved.

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Design of a Portable and Customizable Microprocessor for Rapid System Prototyping"** by **Tamar Kranenburg** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 23-09-2009

Advisor:

Dr.ir. T.G.R.M. van Leuken

Committee Members:

Prof.dr.ir. A.J. van der Veen

Dr.ir. N.P. van der Meijs

Prof.dr. K.G.W. Goossens

Abstract

Due to the increasing number of processors which are integrated in SOCs the need for robust, highly configurable processors emerged. Preliminary research showed that commercial processors are not suited for many research projects due to the fact that they are closed source and they can not be modified. Open source processors on the other hand appeared to be of too low quality. Within this thesis a light-weight instruction and cycle compatible implementation of the MicroBlaze architecture called MB-LITE is presented to fill the need for a fast, reliable processor suitable for FPGA and semi-custom implementation. This was accomplished by using a proven design methodology and using high-level VHDL abstractions.

Experimental results showed that MB-LITE is able to obtain much higher performance than existing open source processors, while using very few hardware resources. MB-LITE can be easily extended with existing IP components due to a wishbone bus adapter and a modular, easily configurable multiplexed memory bus. All components are thoroughly tested for compliance, and their functionality was proven to be correct using timing back-annotated simulations. Continued work on MB-LITE will focus on including the design in a reconfigurable fabric as well as fabrication in a 90 nm semi-custom IC technology.

Acknowledgments

In June 2008 I entered the office of René van Leuken to discuss the opportunities for a graduation project. Although many different projects were available, the development of a microprocessor immediately got my attention. This assignment gave me the feeling that I could really contribute to research and make the lives of researchers a lot more pleasant. The idea that there were plans to make a real chip based on my work was exactly the challenge I was looking for.

When I started working on this project in September 2008 the idea was to select one of the many available open source processors and improve it wherever necessary. It quickly turned out that the existing processors were not documented at all, so the answers to many questions had to be found in code. This was a tough job, but fortunately my advisor constantly pointed me into the right direction by asking the right questions.

I would like to thank my advisor René for all of his assistance during this project. When I was writing this thesis the idea developed to submit a paper about our achievements. Unfortunately there was not very much time left to write this paper, but you generously sacrificed some time during the weekend to help finishing it before the deadline.

Many other people contributed indirectly to this project by giving distraction whenever I needed it. My roommates kept me awake during the late hours by serving coffee on the right moment. To this extend we have set up a small community which we called "the black gold alliance". Special thanks goes to Steven Mulder since he made coffee most often, but especially because he refurbished the LaTeX thesis template of Computer Engineering and often assisted me in solving LaTeX related issues.

Many other people helped me to find distraction whenever I needed it most. Many thanks goes to my *jaarclub* Speer, with whom I have spend countless evenings with games, dinners and other parties. Furthermore, my soccer team "Galaxy Frankfurt" helped me to get some exercise during Monday evenings. Without all you guys I would have never been able to finish my study without a mental breakdown.

I would also like to express my thanks to my parents and brothers, who have always supported me during my study and graduation project. Even educational differences did not withhold you to struggle through my texts and to give feedback on my work. Without their support it would have been impossible to finish my study successfully.

Tamar Kranenburg
Delft, The Netherlands
23-09-2009

"*Everything should be made as simple as possible, but not any simpler.*"
—Albert Einstein

"*There's ways to amuse yourself while doing things
and thats how I look at efficiency.*"
—Donald Knuth

Contents

List of Figures

Introduction 1

Many scientific research in the field of Computer Engineering use one or more soft core processors within a single SOC to implement new and innovative concepts. Researchers who use processors like MicroBlaze experience that simulation and integration within their project is a time consuming and exhausting task. Within this thesis a solution is proposed to make the life of researchers—and in the second place hardware developers—a lot easier.

To this extend a platform is developed in which the focus is not only on speed, but also on usability. An open source 32-bit General Purpose Processor (GPP) is required which can be used for quick-prototyping of research projects. Projects who can have major benefit of simple multi-processor configurations can be found in research on Wireless Signal Networks (WSNs) and Network On Chips (NOCs), but it is likely that many other projects can take advantage of a reliable, easy to use processor platform.

Research topics change rather quickly and every topic will probably have different requirements and expectations of a microprocessor in terms of features, clock frequency and processor size. In order to take advantage of experience with a processor in one project, it is desirable that we can use this experience also in many other projects. Due to these differences in requirements a processor needs to be configurable. The selection of a platform which can fill all different requirements is therefore a challenging task.

A rather different design metric concerns the portability of a design, which describes qualitatively the versatility in platforms a design can be implemented on. Three major technologies can be targeted in a design. A Programmable Logic Device (PLD) is a component of which the functionality can be programmed at least once. A FPGA is of this type and can be quickly reprogrammed many times. The second platform which can be distinguished are semi-custom ICs and consist of a library of basic building blocks of which a design can be composed. These designs can not be programmed like PLDs but have to be manufactured in a factory—hence these designs are much more expensive. Finally, full-custom ICs processes offer the most flexibility but are very time-consuming and complex to work with. Within this thesis focus will be on the first two classes. Both of these classes are based on platform dependent libraries which contain all basic building blocks.

Generally a design is first developed and tested on a PLD before an IC is fabricated. This relation shows the importance of a design which can be easily ported to both technologies. A PLD is the platform of choice for quickly prototyping a system and prove correct functionality. Subsequently, steps can be taken towards IC fabrication while the original design serves as a reference.

An infinite list of design goals and requirements can be defined in this way. This would become an unfair quest if all goals are equally important. The implementation proposed and the methodology used in this thesis will focus on designers and researchers. Therefore high usability, portability, simulation speeds and synthesis speeds will be more important than performance metrics like resource utilization and processor speed.

Within this thesis a solution to many of these requirements is proposed. In Section 1.1 it is motivated how a widely applicable processor can improve the development speed of state of the art concepts. In Section 1.2 the thesis goals will be formulated. Section 1.4 gives an overview of related work while subsequently in Section 1.5 an overview is given of basic concepts used within this project. This chapter is concluded in Section 1.6 with an overview of the organization of this thesis.

1.1 Motivation

Many applications include at least one—but often many more—GPPs. Due to the increasing capacity and performance of FPGAs more and more processors can be used in a design without violating resource constraints. The standard MicroBlaze tools do not support multi-processor configurations—a topic which is gaining more and more interest nowadays. Experience with manually inserting these processors in a project becomes an exhaustive, time consuming task. Experiences within different research projects also showed that there is a need to quickly change the design—for example to test a different bus or a new co-processor. It is therefore also important that the bus is simple enough to support quick prototyping. In order to provide the best achievable flexibility in hardware an open source alternative is required.

Many processors have been designed and implementations might be expected to be widely available. The available designs can be separated in two groups, i.e. processors with a commercial license and processors with an open source license. Commercial processors are generally provided as firm-core implementations which are technology dependent and are practically unmodifiable. An important requirement for processors which will be used in research is that they can be changed—for example to implement a different bus—see for example [1, 2, 3]. Furthermore, research depends heavily on simulation of components, but simulating firm-core designs is very time consuming compared to the simulation of a behavioral model.

Since most available processors are closed source, only a fraction of the total available processors remain which can potentially fill our need for flexibility. Due to the nature of the open source designs quality and support become major concerns in the selection of an open source processor for research purposes, especially when it is considered that at some moment the design must be implemented in a rather expensive IC fabrication process.

It is concluded that it is far from trivial to select an open source processor which can satisfactorily meet all requirements. A preliminary research will point out if such a processor exists. However, first a thorough list of requirements need to be defined.

1.2 Thesis goals

The goal of this research is to develop an open source platform centered around a processor, which is suitable for a wide range of research projects in the field of Computer Engineering. A processor is ought to be suitable if the following design goals are met.

- Open source, to be able to make changes to several parts of the design
- High quality and reliability, in order to simplify maintenance, promote design reuse and increase its durability
- Standard compliant, in order to reduce the modifications necessary to replace the processors used in current projects
- High configurability, in order to use the same processor in different configurations and projects
- High usability, in order to be able to quickly build prototypes of novel concepts. Standard interfaces and components must be available.
- High simulation and synthesization speeds, since research designs are always "under test" and need to be build repeatedly
- High portability, in order to be able to implement it in a 90 nm process technology
- Small implementation size, in order to reduce production costs or use many equivalent components within a design
- High performance, in order to avoid the processor to become the bottleneck in high-performance designs
- Availability of components, to avoid having the need for designing standard components or interfaces

Many of these design goals can be achieved by following a well-defined design methodology. To the best of our knowledge the methodology summarized in Section 1.5.3 is the only available method. This model will therefore serve as reference for the qualitative aspects enumerated above. Additionally, design recommendations from Section 1.5.2 will also be used as starting point to compare existing designs with.

1.3 Achievements

None of the existing processors filled the needs satisfactorily for different reasons. The most important shortcomings have to do with the lack of a design methodology, bad performance or low overall quality. Therefore a new processor called MB-LITE was designed according to the MicroBlaze processor specification in which all these problems were removed. A design strategy and methodology was followed to obtain a light-weight cycle- and instruction compatible MicroBlaze clone with outstanding performance. A paper has been submitted for acceptance in the Proceedings of the Design, Automation and Test in Europe 2010 [4].

Many of the design goals can be achieved by applying a good, generic coding style. The two-process methodology is a design strategy used by many developers to obtain high quality designs. A strict separation between behavioral and sequential logic makes it possible to take the most advantage of optimization strategies within compilers—it has been proven that this strategy can be applied to obtain fast designs using less resources than compared with other methodologies. Furthermore, readability is greatly improved since the algorithm is completely determined by the behavioral part of the component. Therefore projects designed using the two-process methodology have improved maintainability than other projects.

Compliance with MicroBlaze specification was proven by designing a test bench which runs programs compiled using the standard toolkit from Xilinx. A program was assembled which—after compilation—resulted in a test environment with full statement coverage. Many different programs are generated using different libraries to make sure that the design correctly implements the specification. To prove correct behavior on a real platform, MB-LITEs synthesized netlist was simulated including time annotations (post place-and-route). The simulation results were compared with the behavioral model to prove equivalent behavior.

High usability is achieved by including a very flexible, highly configurable multiplexed bus which can be connected to the MB-LITE data bus in a plug-and-play fashion. The bus contains an address decoder which can be completely configured using two generic parameters: the number of outputs and its address ranges. Usability and availability of components is improved even more by another plug-and-play component: a wishbone bus adapter. Using these two components a memory topology can be created without knowledge of MB-LITE itself.

Portability has been optimized by exclusively making use of synchronous, generic, inferred components—an approach proven to be feasible in [5]. Simple basic memory structures are used since these are generally the most challenging components to implement in a semi-custom process. Several important parameters are included in order to configure the processor with an optional multiplier or barrel shifter or to enable or disable support for interrupts.

This research resulted in a reliable framework for projects using one or several embedded microprocessors. Since MB-LITE has been designed with the intention to obtain very high usability, very good design quality and high performance this processor is suited for many different applications.

1.4 Related work

Although many soft core microprocessors exist, very few research have been done on aspects like usability and portability. Most research focus exclusively on the design for specific FPGAs and do not take portability into account. In [6] the five most used soft core processors are evaluated. Commercial products like Nios II, MicroBlaze, PicoBlaze and Xtensa of the vendors Altera, Xilinx and Tensilica respectively and the two open source processors LEON3 and OpenRisc1200 are evaluated and compared. The presented results are incomplete and the conclusion is quite shallow.

In [7] a clear overview is given of synthesizable Central Processing Unit (CPU) cores, namely the LEON2, MicroBlaze and OpenRisc1200. Many aspects are taken into account ranging from speed, area usage to synthesization and documentation. The focus was primarily on “comparing” these processors in terms of performance, while insufficient attention have been spend on the integration of custom IP blocks.

In [8, 9] the performance of different configurations of an Altera NIOS clone, called UT NIOS is presented and compared with the original. No relation have been made with other processors, so the results are quite shallow. Only performance and size metrics were taken into account in their comparison.

The authors of [10] noticed that no reliable and small microprocessor implementation existed which was small enough for their research on configurable processor arrays. A small implementation was desired since they needed to include many of them on a single chip. The author therefore wrote a MicroBlaze compatible clone called OpenFire.

1.5 Background

Several ideas and concepts about system modeling, microprocessor design as well as FPGA and Application Specific Integrated Circuit (ASIC) development are frequently used within this thesis. These concepts are used as the foundation for several lines of thought. Furthermore an short introduction into the classic Reduced Instruction Set Computer (RISC) pipeline is given. It is assumed that the reader has some basic knowledge of instruction set architectures. A brief overview of these concepts are given as well.

1.5.1 Soft-, firm- and hard-core models

Three levels of hardware descriptions are generally distinguished: behavioral, structural and physical descriptions. A behavioral model is an abstract description of the algorithm of a model (i.e. how a model behaves), by defining the outputs as a function of the inputs. A behavioral model can be synthesized using automated tools to obtain a structural model which contains information about the implementation of each component. Finally, a physical model provide most details about the implementation in terms of selected components, wires and its delays.

Traditionally, the design flow starts using the behavioral description. After the functionality has been verified, subsequent design steps are gradually taken towards physical implementation. Synthesis is the process to derive a structural model from a behavioral model. To obtain a physical model using the structural model requires two more steps: mapping and routing.

Three different names are used frequently to specify how an IP core is provided (e.g. by a vendor). A piece of hardware can be provided as soft-, firm- or hard-core. According to Vahid et al. these terms correspond with the three levels of hardware description (i.e. behavioral, structural and physical) [11].

All process steps depends on large libraries of available components, which are specific for a platform. Therefore only a purely behavioral description can be technology independent. Unfortunately, behavioral and structural models are often mixed by instantiating specific components from a behavioral description. This reduces portability and must be avoided. A better approach is to “infer” a component, i.e. writing the model in such a way that the synthesizer is able to recognize a structure and select the correct implementation. Nevertheless, this does not always give good results, in which case solid workarounds have to be used to solve these problems while maintaining platform independence.

1.5.2 ASIC design recommendations

The recommendations presented in this section are derived from the book “Advanced ASIC Chip Synthesis” [12]. These recommendations in turn originate from recommendations from Synopsis as well as experiences with Synopsis Design Compiler.

A good coding style is very important for several reasons. When working together with a team on a project, other team members should be able to use and adapt the design for their needs. This promotes the reuse of components which in turn will greatly reduce development time. Secondly a good coding style will improve the synthesis process which in turn results in faster logic and a reduction of chip area. Finally the design will be more suited for different synthesization processes which simplifies platform portability.

Components should be modeled while keeping the desired hardware implementation or construct in mind. Since Very High Speed Integrated Circuit Hardware Description Language (VHDL) is a hardware description language which is based on templates, different synthesization processes can infer different components with alternative area and timing specifications. Therefore all basic hardware components like memories and arithmetic operations should be inferred from a general structure, and for the same reason a ‘case’ statement is preferred with respect to an if statement. Explicit instantiation of components should be avoided whenever possible. In this way the system can be implemented on different technologies fairly straightforward. If this is not suitable for the desired function the technology dependent parts should be implemented in separate modules. The use of latches within a design is strongly discouraged since this makes static timing analysis cumbersome.

Clock gating logic and reset generation should be captured within a single module. Avoid using multiple clocks in a single module since this relieves the burden and tedious task of managing clock skew. Furthermore it is recommended that the complete circuit uses the same name for the clock signal.

The top-level component should not contain any combinational glue logic. This will greatly reduce the compile time of the block since all components can be simply put together. In a later design phase this will also simplify the place and route stage which will result in improved area utilization. Another related recommendation is that unnecessary design hierarchy should be avoided since Synopsis Design Compiler is unable to optimize efficiently across hierarchies.

All outputs of a module should be registered. Unfortunately this is not always practical, for example when a memory transaction has to occur in the same cycle. Adhering to this recommendation improves circuit optimization and prevent timing problems to occur.

Use of these recommendations is important since it contributes in making a design comprehensible and portable. It covers the most frequent problems which occur in the development of hardware for ASIC implementation. Although these requirements are more important for ASIC designs than for FPGA design these guidelines are equally applicable for designing programmable logic.

1.5.3 Structured design and the two-process methodology

This section is based on the two-process methodology as explained and motivated in "Fault-tolerant Microprocessors for Space Applications" [13].

Traditional "ad-hoc" VHDL designs are frequently used to model the behavior of a component. These designs typically have a large number of signals defined which are related by many small processes. All processes are active concurrently which makes it hard to understand its behavior without thorough knowledge of the "big picture". Another characteristic of these designs is that many lines of code are necessary to describe the behavior of basic components like multiplexers and registers, since coding is done at the lowest level by assignments with logical expressions. Abstract data structures are generally not used. Due to the many concurrent statements and the large number of lines of code these designs are hard to read, difficult to understand and virtually impossible to maintain. Furthermore, large port declarations are necessary in entity headers and the execution is slow due to the large number of signals and processes. Therefore the ad-hoc method is not scalable.

Ideally behavioral VHDL models are easy to understand and can be synthesized without making modifications. Simulation and synthesis should be as fast as possible while no discrepancies are introduced between the behavioral and the synthesized model. An ideal model can thus be applied for small as well as very large designs, or in other words they need to be scalable.

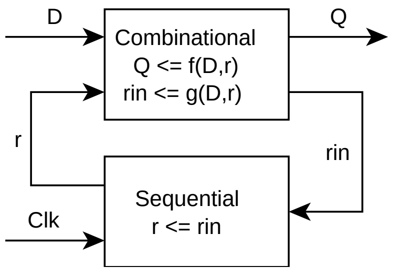


Figure 1.1: General structure of a two-process component

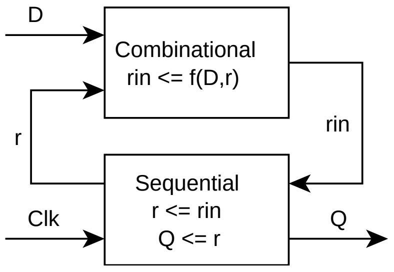


Figure 1.2: Modified structure of a two-process component

The two-process methodology is based on a strict separation between combinational and sequential logic. The combinational process contains the algorithm to be implemented and is determined based on the current inputs and values stored in temporary registers. These registers are instantiated in the sequential process. Due to this strict separation the functionality is completely determined by the combinational part.

The output of the combinational block Q and rin are computed in an combinational process using the current input (denoted D) and the register values (denoted r). The register inputs rin are stored in registers (r) in the synchronous process. The structure of the two-process component component is shown in Figure 1.1.

The signals (D, Q, r and rin) can be collected in records. If a signal needs to be added, only the record needs to be changed while the component instantiations do not need to be modified. Note that in this implementation the outputs are not strictly registered, since logic can be inserted between the combinational inputs D and r and output Q. Furthermore, this approach requires three different records. By introducing a modified structure the number of records can be reduced to two and the outputs become strictly output registered. This structure is shown in Figure 1.2. In this structure the output Q is connected to register r.

A complication is encountered when memory contents need to be fetched during a sequential process. This would imply the use of asynchronous memories. However, as indicated in previous section this is not always favorable and could

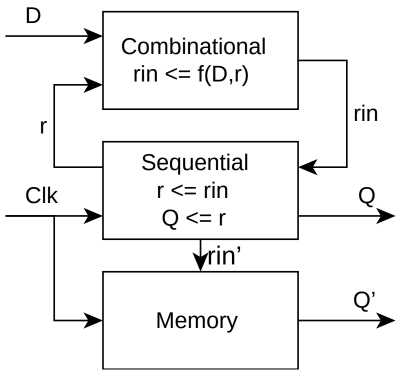


Figure 1.3: Structure of a two-process component including memory

potentially lead to problems when implementing large memories. Therefore the outputs of a memory need to be combined in the outputs of the top level component. Asynchronous control signals can be used internally to control the memory. Logic which need to be inserted after the memory output need to be postponed to subsequent stages. The structure of a two-process component which includes synchronous memory is shown in Figure 1.3.

1.5.4 Introduction to the classic RISC pipeline

All modern microprocessors use pipelining as the primary technique to increase performance. Using this technique we make advantage of parallelism between subsequent instructions. A pipelined architecture provides overlapped instruction execution to improve hardware utilization and eventually improve the processors throughput. The RISC pipeline can be used for many different architectures. MIPS and DLX are two of the most famous RISC architectures which are implemented using this pipeline technique. We will start by separating the execution of a single instruction in several functions.

The classic RISC pipeline consists of five stages which are consecutively active. The first stage is called Instruction Fetch (IF) and handles the retrieval of new instructions from instruction memory. When a new instruction is available it is passed to the Instruction Decode (ID) phase. During this stage the instruction is inspected and decoded in a set of control signals. The control signals determine for example if an Arithmetic Logic Unit (ALU) needs to do an addition or multiplication, if a branch needs to be taken and if that branch depends on the result of the ALU and finally if memory should be read or written. Finally, if the instruction requires operands located in a register file than these are read from the register file.

After the ID stage the instruction is fed to the Execute (EX) stage where arithmetic operations are performed and branch conditions are evaluated if necessary. Subsequently the instruction goes through the Memory (MEM) stage where results

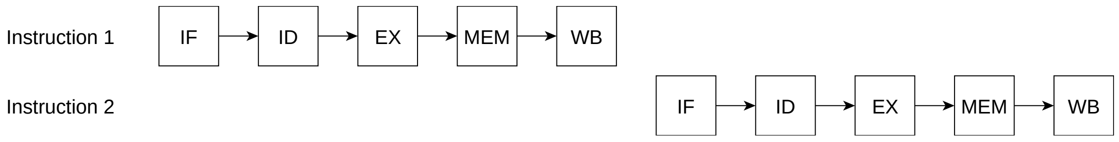


Figure 1.4: Sequential execution of the RISC architecture

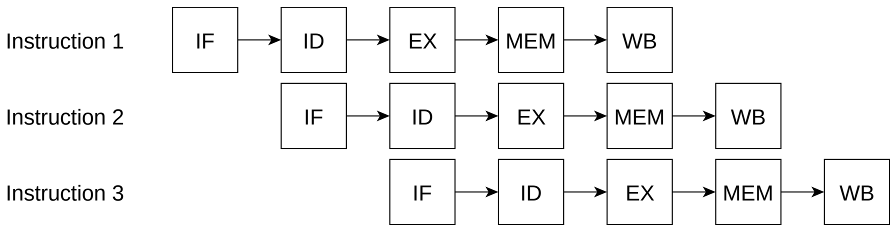


Figure 1.5: Pipelined execution of the RISC architecture

are retrieved from or written to data memory. Finally the instruction is transferred to the WriteBack (WB) stage where the execution result or the memory value is written back to the register file.

The term memory can be confusing for a novice reader. In a memory mapped computer model every device connected to the processor can be seen as memory with a private memory space. These spaces can be physical memory where data is stored or retrieved, or a device such as for example Universal Asynchronous Receiver/Transmitter (UART), Ethernet or Universal Serial Bus (USB). The addresses of all devices are logically distributed and named using a memory map.

All pipeline stages could be executed one after the other during a single clock cycle. This would result in a long execution path because every instruction needs to reach the end of path before the next instruction can begin. Figure 1.4 shows an example of this situation. In this organization only one functional unit will be active at every point in time so this solution is inefficient because expensive hardware resources are being spilled.

Pipelining allows us to get more performance by improving hardware utilization since logical units are now used concurrently. Due to the improved hardware utilization instructions can finish faster and hence the processors frequency is improved. The time line for pipelined instruction execution is shown in Figure 1.5. This organization might increase the clock frequency by a factor which is equal to the number of pipeline stages if the workload of each stage is equal which is generally not the case. The bottleneck is determined by the slowest (i.e. most complex) stage and the stages which are finished earlier must wait for the other stages to complete. This limits the ideal speedup which can be obtained by pipelining.

There are however three problems which need to be taken into account. Subsequent instructions often use the result of previous instructions resulting in dependencies within the execution of the pipeline. The register file might be used by the writeback stage to write values into it while at the same time the decode stage needs to read values from the register to feed the pipeline. Such hazards are called structural hazards.

Since several instructions are in the pipeline and as a result are not finished yet old values might be used since the dependent instruction have not passed the memory or writeback stage yet. The instruction which is in the execute stage might use old register values and the program does not run as expected. These type of hazards are called data hazards.

Control hazards arise from the fact that a branch decision is taken in the EX stage while the foregoing IF and ID stages have started working on the instructions which were originally next in order. If the branch is not taken this is good, but if the branch is taken these two instructions should not have been executed. Therefore if a control hazard occurs the pipeline should be flushed.

Solving these problems can be done rigorously by removing all erroneous executed instructions and issue them again when the dependencies have disappeared. This would obviously lead to many stalls (i.e. the execution of an instruction is delayed until all dependencies have been solved) and flushes (i.e. the pipeline is emptied). Fortunately a better solution can be implemented for all three hazards to reduce the stalls and hazards to a minimum.

A Harvard architecture has the advantage that a structural hazard will not occur if the register file supports concurrent reads and writes so during the same clock cycle a value can be written and a new value can be read from memory. The register file can be still involved in causing a data hazard if a value is read from and written to the same logical address. This can be solved by using read-after-write (i.e. write-first) memories, in that case the updated value is always used as the read result. If such memory is not available on the targeted platform dedicated logic must be added to make sure that correct values are used in subsequent stages.

A combinations of instructions can cause a data hazard if the destination register is equal to one of the two operands of the next or subsequent instruction. In these situations the updated destination register is in the EX or MEM stage. Forwarding can be implemented to solve many situations without the need for stalling the pipeline. Nevertheless one situation that can not be solved by forwarding is when an operand is read in the MEM stage while in the same clock cycle the EX stage needs this value. In this particular case we have to stall the pipeline and make use of the forwarding unit but a stall is unavoidable. It is up to the compiler of the software to reduce the number of such dependencies to a minimum by rearranging the instructions.

Control hazards can not be easily solved by additional logic. The stages IF and ID need to be executed so the branch decision can be taken at its earliest occasion in the EX stage. As already mentioned if the branch is not taken the instruction could in fact be ignored and execution continues normally. However, if the branch decision evaluates to logic true the branch should have been taken although the IF and ID stages have fetched and decoded the next two instructions respectively.

These instructions must be removed from the pipeline which is called flushing the pipeline.

Obviously control hazards are the most expensive type of hazards since branches occur very frequently. A technique to reduce the delay penalty to only one cycle does exist and involves small hardware changes and compiler support to get the most advantage of it. If the instruction immediately following the branch is always executed and a useful instruction can be inserted in this slot one out of two cycles is saved. It is often not very hard to modify the software in such a way that efficient use can be made of this so called branch delay slot.

An important property of pipelining is the number of cycles the execution of a single instruction will take on average. This indication is expressed as

$$\text{CPI} = \frac{\text{Clock cycles}}{\text{Instruction}}$$

and is called Cycles Per Instruction (CPI). An ideal pipeline will finish one cycle per instruction and has a CPI equal to 1 while a realistic CPU can be expected to have a CPI somewhere around 1.4.

1.6 Thesis organization

The organization of this Master thesis is as follows. Chapter 2 existing CPU cores are evaluated. A first selection is made of existing open source processors which have the potential to serve as the core of the platform to be developed. In Chapter 3 it is tried to adapt the selected processor to our needs by applying certain modifications. Efforts will be spent on improving the design, especially considering the requirements on portability. Unfortunately, this approach leads to a dead end. Within the second part of this chapter, a new design called MB-LITE based on the MicroBlaze architecture is presented in order to overcome the drawbacks imposed by all other open source designs.

In Chapter 4 MB-LITE is tested for compliance with the MicroBlaze specification and size and performance results are compared with the CPUs of Chapter 2. Chapter 5 will give a reflection of what has been achieved during this research and is finished with an overview of future research and recommendations.

2 Evaluation of CPU cores

Several implementations of open source soft-core microprocessors are available for free on the Internet. First a quick selection is made of open source processors which will serve as a starting point for further exploration. Processors are selected based on its maturity, features, available bus interface and its reliability (e.g. is it used in different projects, has it recently been updated). Subsequently each of the selected processors will be described and deficiencies will be discussed. The performance of the processors will be analyzed in terms of speed and size. Finally, several quality aspects will be reviewed in relation to the requirements as presented in Sections 1.5.2 and 1.5.3. In Section 2.6 our findings will be discussed and a processor will be selected to serve as reference design.

2.1 Exploration of available processors

In this chapter we analyze the available processors which comply with the requirements described in Section 1.2. Our main source of LGPL licensed processors are the websites of OpenCores [14], Aeste [15], Gaisler Research Laboratories [16], OpenSPARC [17] and Lattice Semiconductor Corporation [18]. Currently 95 microprocessors are listed on OpenCores. Since this is a little bit too much to take into account we will start by reducing this list to a set of candidate processors, based on several basic requirements involving architecture, bus-width and development stage. The initial selection of processors together with some features are shown in Table 2.1.

The processors are gradually more thoroughly explored. Therefore some sections are much more elaborate than other sections and provide more details about the design.

Table 2.1: Candidate processors and features

Processor name	Architecture	HDL	Wishbone
AeMB	MicroBlaze EDK 6.2	Verilog	yes
LEON3	Sparc V8	VHDL	no
OpenFire	MicroBlaze EDK 7.1	Verilog	no
OpenRisc 1200	ORBIS	Verilog	yes
Plasma	MIPS	VHDL	no

2.1.1 AeMB

The aeMB 32-bit microprocessor is under development of Aeste Works. The processor uses a wishbone interface for both data- and instruction memories. It has a five stage pipeline and separate instruction and data buses (Harvard architecture). The organization is based on the classic RISC pipeline, see Section 1.5.4. Both instruction and data memory comply with the wishbone bus protocol and provides support for a single external interrupt. Additionally a Fast Simplex Link (FSL) bus is present. The processor contains 32 x 32-bit general purpose registers.

AeMB is written in Verilog. The width of the buses are fixed within the design. Parameters are available to enable a hardware multiplier or barrel shifter. The FSL interfaces as well as the caches can not be disabled. The available documentation gives little insight in the design, so making changes can be quite challenging and time consuming. Refer to Table A.1 of Appendix A for a detailed list of features.

The organization of the processor without external connections is shown in Figure 2.1. The signal names are not descriptive and the suffixes _ex, _mx, _if and _of are pretty confusing. The data path—and the timing of the pipeline—is not clear from the implementation or documentation. Component delays are defined within the components which makes verification of an implemented design unreliable, since discrepancies can exist between the behavioral and the synthesized model.

The microblaze tool chain can be used to compile software for this processor, but this appeared to be a tedious task. This is due to the implementation of a two-threaded system, which performs a context switch after every cycle. For no obvious reasons some programs compiled for this platform did not execute as expected. Making the header files compatible with the ANSI C solved most—but not all—problems.

The development status as depicted on opencores is indicated as 'stable'. It has been tested using software simulation tools and synthesized on a FPGA. The Verilog code synthesizes without significant problems using Xilinx ISE 10.1. A shell script is provided to compile the software, synthesize the hardware and simulate the software using Iverilog.

Many efforts were required to successfully run the provided tests. At first the memory allocation routine failed. Rewriting the software and libraries according to the ANSI C standard solved this issue. More reliable verification results are not available, but reports can be found about successful project integration.

In order to use this processor several adjustments are be required. The design uses asynchronous memories to implement the register file and instruction cache. Since synchronous components improve the insight into timing issues and asynchronous memories can not be implemented efficiently on FPGAs it might be interesting to replace these components with synchronous equivalents. Furthermore the signals should be grouped more clearly and named more descriptively to increase the insight for future users.

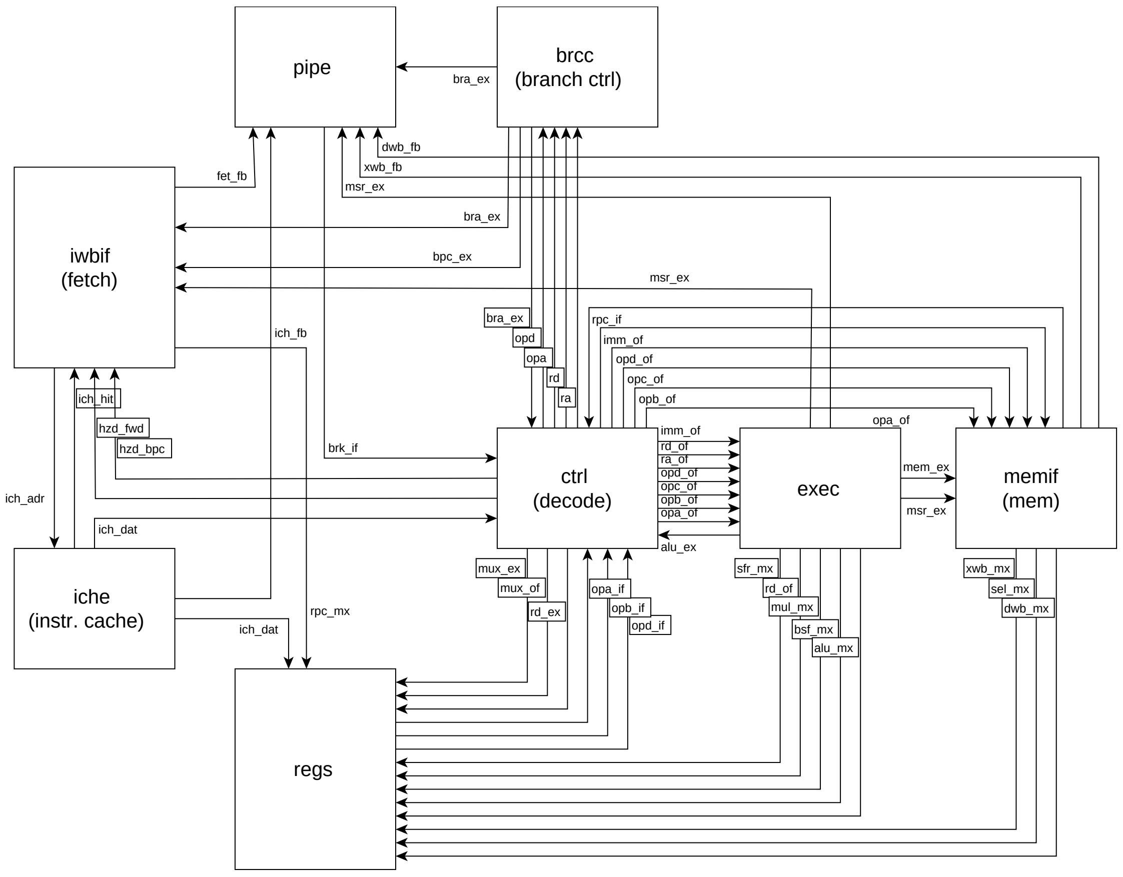


Figure 2.1: aeMB top level connections

2.1.2 LEON3

The LEON family of processors is designed and maintained by Gaisler Research. This processor family is an implementation of the Sparc V8 architecture which is non-proprietary and fully open. The latest processor in this family is the 32-bit LEON3 microprocessor [19, 20]. Its implementation is based on a Harvard architecture and uses the AMBA Advanced High-performance Bus (AHB) as its main on-chip communication bus [21] which is free of royalties as well.

A special feature of the Sparc architecture is that it uses register windowing to increase the performance of context switches. Generally when a procedure is called the register values are stored in memory which results in a considerable amount of overhead. Using register windowing a new 'set' of local registers is selected upon a procedure call which makes the time consuming process of storing register contents in memory redundant. In the Sparc architecture window switching is done by hardware and is therefore transparent to the software developer. This comes at a price since extra logic and memory is necessary to achieve this efficient way of context switching. A default implementation has 8 global registers and 8 sets of register windows, and each window consists of 24 registers. At every time instant

32 registers will thus be visible for a program, while a total of 200 registers exist.

Custom peripherals can be added to the system bus. The AHB supports split transactions which will result in better interconnect performance when many different IP-blocks need to be connected and controlled concurrently. Designing AMBA AHB compatible peripherals is far more challenging than designing wishbone compatible devices. The addition of existing peripherals to the system bus is more complicated since a complex controller needs to be modified. A great advantage of the LEON3 processor is that it uses a structured organization of packets, folders and VHDL records, which influences the usability in a positive sense.

All components are claimed to be written according to the two-process methodology. When a closer look is taken at the integer unit—which is in this research considered as the top-level component—many constants, procedures and functions are defined within the same file. Depending on the configuration, at least 6 processes are defined. The fact that the file consists of almost 3000 lines of code, the structure and relations are just as unclear as the "ad-hoc" design method. Therefore the claim that the processor is written according to a well-defined methodology is questioned.

According to the documentation of the Sparc architecture the integer instruction timings are strictly dependent on the implementation. Nevertheless most instructions take a single clock cycle to execute except for load and store instructions. An advanced feature is that the LEON3 processor can be dual clock to run the AHB bus on a lower clock speed than the processor core. Additional logic needs to be added to cross these clock boundaries.

The design is very modular, so it should be relatively easy to disable certain parts of the processor and save resources while possibly gaining performance. A configuration utility is used to assemble a complete SOC and to include and configure many optional peripherals. Up to eight microprocessors can be attached to the AHB bus. The package also comes with implementations for many different platforms (i.e. several FPGAs and ASIC libraries). Nevertheless it is impossible to obtain a light-weight (integer unit only) implementation of this processor since the memory management unit, caches and AHB bus controllers can not be disabled.

The LEON3 processor is published under the GNU GPL license and can be used free of charge for educational and research purposes. We can safely assume that this processor is very reliable since it is used in military and space applications. Refer to Table A.1 of Appendix A for a detailed list of features.

2.1.3 OpenFire

The OpenFire 0.3b is a clone of the MicroBlaze Embedded Development Kit (EDK) 7.10 architecture. It has 32-bit instruction- and data words and includes all basic arithmetic operations. It is designed for research on configurable soft processor arrays. Features like interrupts, exceptions and special registers are not implemented since the objective of the designer was to keep the core small and simple.

The OpenFire processor connects directly to the instruction memory while the

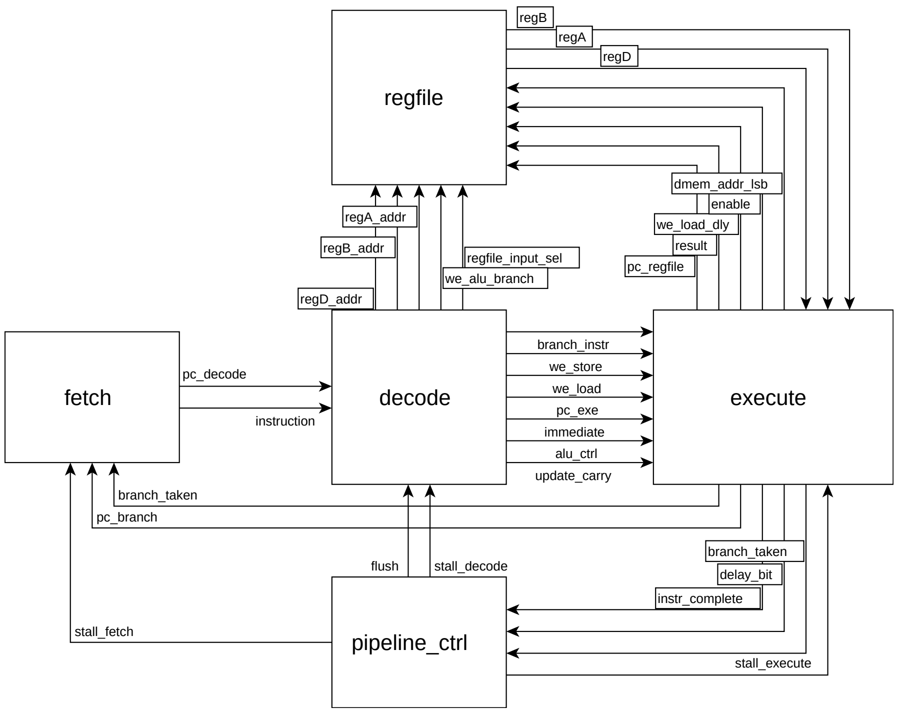


Figure 2.2: Internal top level connections of the OpenFire microprocessor

execution stage connects directly to the data memory interface. The instructions are pipelined in three stages which deviates from the MicroBlaze architecture specification. Loads and stores take more than 1 cycle to execute, so this processor is not cycle compatible with the MicroBlaze specification. These differences had to be found in the source files and are not documented.

The organization of the processor is shown in Figure 2.2. All components—except for the pipeline controller and the register file—are output registered. Signals names are clear, and their use is described in comment. To implement the register file the designer has chosen to use two dual port memories with asynchronous read and synchronous write capability. Several processors can be cascaded using a MicroBlaze FSL bus but unfortunately a wishbone bus is not included in the design.

If the multiplier or the FSL bus was disabled problems occurred because synthesization did not complete. The current implementation is targeted at Xilinx FPGAs and has not been designed with portability in mind. The documentation is very poor and does not give any insight into the design. The documentation mentions that the break instruction does not function correctly, but no efforts are spend to solve this issue. A detailed list of processor features can be found in Table A.1 of Appendix A.

2.1.4 OpenRisc 1200

The OpenRisc 1200 is part of the OpenRisc 1000 family of embedded microprocessors. It contains a five stage pipeline and implements the OpenRisc Basic Instruction Set (ORBIS) architecture. This Harvard architecture has 32 bit instructions and can operate on 32- or 64-bit data. Several extensions to this architecture exist to enhance vector processing (OpenRisc Vector/DSP eXtension (ORVDX)) and add floating point instructions (OpenRisc Floating Point eXtension (ORFPX)). These extensions are not implemented in the OpenRisc 1200 implementation.

Both instruction and data buses have a wishbone interface. Instruction as well as a data cache is available to improve the performance of the operations involving memory transactions. Other components like Joint Test Action Group (JTAG) or a Programmable Interrupt Controller (PIC) are also available. The memory management units for data- and instruction side of the processor can be enabled or disabled but some components are not completely removed and quite some logic remains implemented. The power management feature, debug unit, PIC and tick timer can not be disabled. This suggests that we can not obtain a light weight implementation, e.g. to execute Digital Signal Processor (DSP) algorithms.

The design does not have a hierarchical file organization nor conventional module names and all files are in a single directory. Furthermore, the components are not generalized so the modularity of the design is not optimal. This might lead to problems when we want to add custom peripherals to the CPU, since the interface of the CPU needs to be modified. Therefore the design is not modular, which limits design reuse.

A GNU Compiler Collection (GCC) port for this Instruction Set Architecture (ISA) is provided and maintained by voluntary developers of the OpenRisc project. As a consequence, we rely on volunteers both for software as well as hardware issues. Although this is common for all open source projects, it is more important for this processor since they maintain the architecture, its implementation as well as the tool chain. Therefore this project might have a reduced durability—although there are no signs of this yet. A list of features can be found in Appendix A.

2.1.5 Plasma

The Plasma 32-bit microprocessor has a Von Neumann architecture and is an implementation of the MIPS instruction set architecture which is classified as RISC processor. This core is published on OpenCores and is one of the few written in VHDL. Since VHDL offers better possibilities for abstraction than its competitor Verilog. Therefore it seems that this processor is worth a short review.

Depending on configuration and instruction type the instructions are pipelined in two, three or four cycles. It contains an interrupt controller, several interfaces (i.e. UART, Static Random Access Memory (SRAM), Double Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM)) and an Ethernet controller.

The architecture has several instructions which are patented by Mips Computer Systems Inc and can not be implemented therefore. The most important of these instructions are the unaligned data access instructions. According to the documentation the software needs to be manually inspected and altered in order to see if one of the patented instructions (LWL, LWR, SWL, or SWR) is generated by the compiler. The tool chain is thus not fully automatic and the generation of large programs might become a time consuming task.

The Von Neumann architecture has a fundamental memory bottleneck, since both instructions and data need to be fetched from the same memory. Structural hazards are therefore likely to occur[22]. This will increase the CPI which decreases processor performance. Furthermore, all components are connected through a non-standard system bus. The functionality of the bus is not documented either, so designing a interface might become a difficult task.

A very poor processor description is given on the opencores website and good documentation does not exist. Since it does also not have a standard data memory interface this architecture need big additions. Detailed information about the features of this processor can be found in Table A.1 of Appendix A.

2.1.6 Other processors

Several well known processors are not included in our discussion. This section gives the reasons why they are not included in our research.

The Xilinx MicroBlaze, Altera NIOS and the Altium TSK3000A are closed source and targeted at specific platforms. Lattice provides the LatticeMico32 processor with an open IP core license. However, few software and peripheral devices (i.e. drivers) are available for this architecture. OpenSPARC is an open source implementation of SUNs UltraSPARC architecture and is used for desktop and servers. This processors is incredibly large and complex (e.g. it consists of eight cores and each core can execute eight threads), and therefore not suited for embedded applications.

Several 8 bit processors exist, but these are unable to address sufficient memory for our needs. These are amongst others the 8051 and AVR. ZPU claims to be the smallest CPU with a GNU tool chain. At first sight it seems like an interesting CPU, however it was found that it is a stack based CPU and is therefore not very suited for many tasks. Furthermore it does not provide other advantages compared with those included in our discussion.

2.2 Synthesization results

The performance of the selected processors was estimated with Xilinx ISE 10.1.03 using a Xilinx Virtex 5 (XC5VLX110-3FF1760) development board. Since all processors have a different set of peripherals we aimed to reduce the CPU configurations to a minimum, preferably implementing the integer unit only. This will give results which can be reasonably compared. All results will be discussed in this section and are summarized in Table 2.2.

Table 2.2: Numerical performance results

	Flipflops	LUTs	Max frequency (MHz)
AeMB	711	926	279
LEON3	1133	3448	183
OpenFire	207	752	198
OpenRisc 1200	1577	3802	185
Plasma	1297	2457	73

The performance of aeMB appeared to be very high. While consuming only few logic elements the clock frequency is around 279 MHz when the multiplier and barrel shifters are disabled. It occupies 711 flipflops and 926 lookup tables. Configurations with or without the barrel shifter and multiplier resulted in negligible changes in the amount of logic and clock frequency.

LEON3 has commercial implementations and is used in applications with high demands (e.g. space and military). The CPU was configured without the multiplier, AHB and Random Access Memory (RAM) controllers, interrupts or other peripherals like JTAG and UART. This configuration results in an implementation running at 183 MHz while using 3448 Lookup Tables (LUTs) and 1133 flipflops. Furthermore, it uses 10 Block Random Access Memory (BRAM) elements which is quite a lot. It was noted that the implementation uses six global clocks. In Section 1.5.2 it was motivated that multiple clocks should be avoided to avoid problems with clock skew which are incredibly difficult to solve. However, because ASIC implementations have been created using this architecture this should not be a problem.

A stand-alone version of OpenFire does not compile immediately with ModelSim SE nor Xilinx. Nevertheless, after some efforts a successful synthesization was performed. It was estimated that the processor obtains a maximum clock frequency of 198 MHz using relatively few hardware resources, only 752 LUT and 207 flipflops. The hardware requirements are thus very low, but quite some important instructions and features are not implemented.

OpenRisc 1200 processor synthesizes without problems. The clock frequency on our target FPGA is estimated to be around 185 MHz. The implementation of the core uses 3802 LUT and 1577 flipflops.

Plasma synthesizes easily without modifications or configuration changes. Despite the fact that only few features are implemented, it uses 2457 LUTs and 1297 flipflops which is quite a lot for this processor. The processor speed is very low and estimated at 73 MHz on our FPGA.

The measurement results are presented in Figure 2.3. It can be seen that aeMB obtains the highest clock frequency while OpenFire uses the least hardware resources. This trade-off between processor speed and implementation size is shown in Figure 2.4 where the amount of Logic Elements (LEs) is shown divided by the

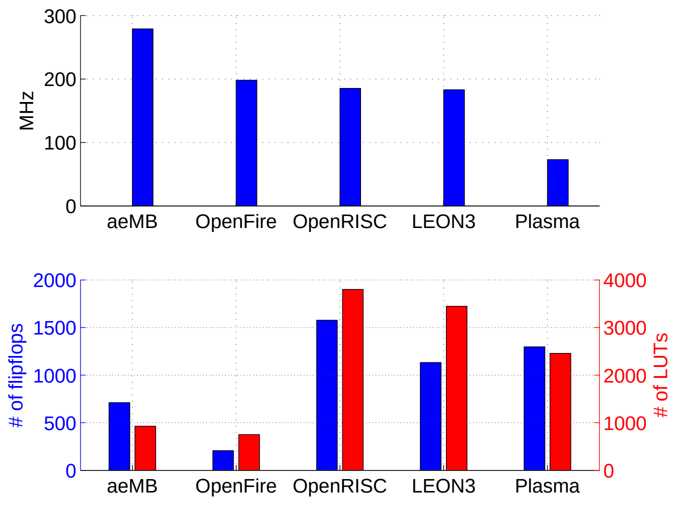


Figure 2.3: Processor speed and size

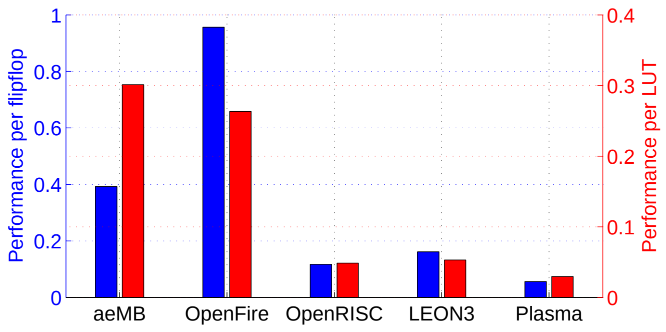


Figure 2.4: Relative processor performance

performance in MHz. It is assumed that the amount of logic is equally important as the resulting performance so an unweighed division is applied.

The trade-off between cost and performance between all different processors is visualized in Figure 2.5. Using this plot a trade-off between size and performance can be easily made. The fastest processor can be found on the right edge of this figure, while the smallest processors can be found at the bottom. Often both performance as well as resource utilization have tight requirements. In those cases the processors found on the bottom-right of the figure are favored since they use least hardware resources while obtaining significant speeds.

In conclusion to previous research it has been found that LEON3 and aeMB are the most complete and efficient implementations until so far. Plasma is at this moment discarded due to the very limited performance in terms of speed, while OpenRisc is too large for the amount of features it offers. Finally, OpenFire is too limited since there is no solid data bus nor interrupt. Since none of these processors have been designed using a well-defined methodology, it will be hard to adapt any

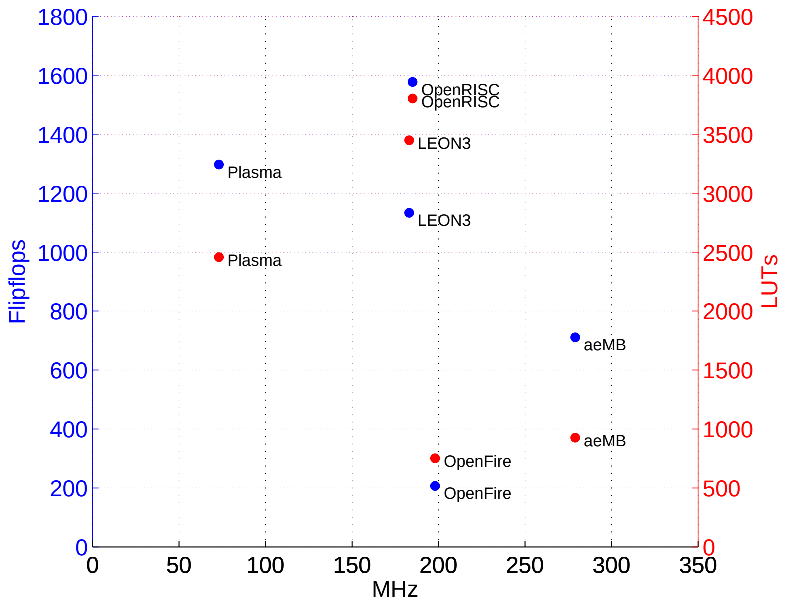


Figure 2.5: Scatter plot of processor performance

of these designs to our needs. Therefore this option is discarded as well.

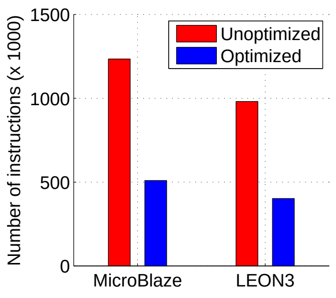
The rest of our preliminary study will focus on LEON3 and aeMB since these are the most promising candidates for our research. In subsequent sections the performance and quality of the tool chains for the MicroBlaze and Sparc architectures will be included to obtain an indication for the real execution time of several programs.

2.3 Tool chain results

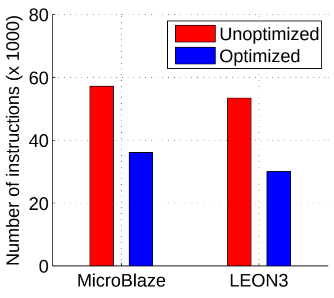
The ISA and corresponding tool chain of LEON3 and aeMB play an important role in the performance of the processor as a system. Therefore several instruction set simulations were executed. Two simple benchmarks were used to compare the performance of the aeMB and LEON3 microprocessors. The Dhrystone benchmark is a well-known indication for processor performance developed in 1984 by Reinhold Weicker. This benchmark is representative for integer operations and has been used extensively as an indication of processor performance for embedded applications.

To test iterative function calls a Fibonacci formula was used to evaluate performance on these areas. This benchmark computes the fifteenth Fibonacci number. Due to these iterative calls it is expected that LEON3 will perform slightly better, since it can make use of its register windows and thereby reduces the load on the data memory interface.

The benchmarks were compiled using the compilers sparc-elf-gcc and mb-gcc. Cycle accurate instruction simulators for these processors were used to measure the number of instructions and cycles. The LEON3 measurements were performed



(a) Dhrystone instructions



(b) Fibonacci instructions

Figure 2.6: Number of instructions for both processors

using the evaluation version of TSIM distributed by Aeroflex Gaisler [16] while the aeMB tool chain was tested using the build in Instruction Set Simulator (ISS) of Xilinx Microprocessor Development of EDK 8.1i[1]. Using an ISS makes it superfluous to assemble a design which might be a time consuming task.

According to the project description of the aeMB microprocessor on the Open-Cores website the processor is "almost cycle and instruction compatible with the MicroBlaze tool chain". It is assumed that these processors will perform roughly the same. Therefore the MicroBlaze ISS was used to obtain performance results [2].

The instruction count, cycle count and CPI of the Dhrystone and Fibonacci benchmarks are shown in Figure 2.6, 2.7 and 2.8 respectively. The benchmarks were compiled with and without optimizations. The unoptimized versions give the best indication of relative performance since modern compilers might apply rigorous optimizations so that the benchmarks are not representative for general applications.

Comparing the unoptimized benchmark results we can see that the Sparc V8 compiler needs less instructions for the same result for both benchmarks. As was already expected this has most likely to do with the register windows used in the Sparc architecture. LEON3 has on the other hand a slightly worse CPI.

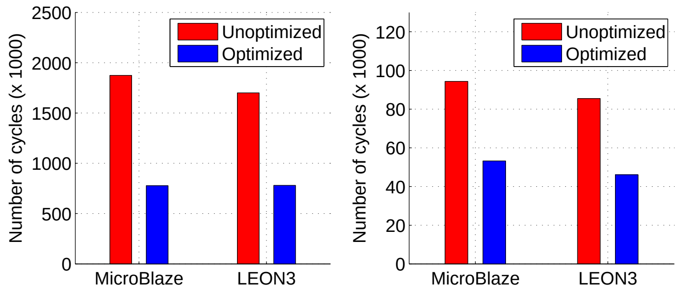
In the end, the most important metric is how long the execution of a program takes, so the clock frequency of the processor needs to be taken into account. The formula for the computation of the execution time is

$$T_{\text{total}} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Clock cycles}}{\text{Instruction}} * \frac{\text{Seconds}}{\text{Clock cycle}}$$

and this figure was computed for both processors and benchmarks. The total execution time for both benchmarks is shown in Figure 2.9.

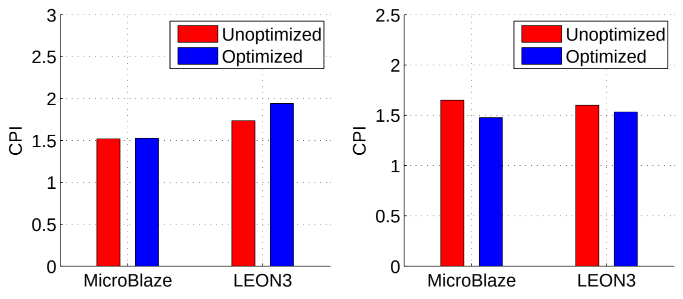
[1]It appeared that subsequent versions of Xilinx ISS have a bug or did not support advanced statistics. Therefore an older version of ISS had to be used.

[2]In the next chapter we will see that aeMB is not cycle compatible at all. Performance results as mentioned in this section might therefore be interpreted incorrectly. Better performance indications can be found in Section 3.1.3



(a) Dhrystone cycles (b) Fibonacci cycles

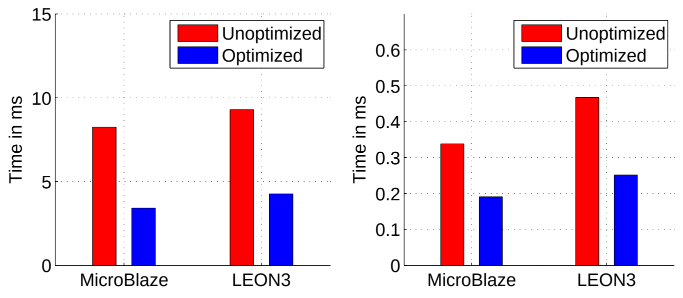
Figure 2.7: Measured number of cycles for both benchmarks and processors



(a) Dhrystone CPI (b) Fibonacci CPI

Figure 2.8: Measured CPI for both benchmarks and processors

Although LEON3 has a smaller instruction count, this has been compensated by the higher CPI and lower clock frequency. The Microblaze tool chain and processor therefore performs better than LEON3.



(a) Dhrystone execution time (b) Fibonacci execution time

Figure 2.9: Execution times of the benchmark set

2.4 Configurability

As explained in Section 1.2 the processor will be used in a variety of projects which makes configurability an important requirement. We evaluate both processors with respect to this requirement by testing different configurations while keeping an eye on size and performance.

The aeMB core provides several parameters to adapt the core to different needs. Several CPU parameters can be changed, although other parameters are absent or not implemented. It is impossible to disable the instruction cache. A parameter to implement the FSL bus is ignored so we can't disable this part of the microprocessor without modifying the design. The widths of the three buses (data, instruction and FSL) can be changed without any problems to values smaller or equal than 32, but larger values are not supported and result in synthesis errors. Size nor performance advantages are obtained when scaling the width of the data memory.

To facilitate the configuration of a LEON3 microprocessor a graphical tool to assemble the CPU with some peripherals is provided. Many parameters for Floating Point Unit (FPU), Memory Management Unit (MMU), data- and instruction-cache can be changed, even the target technology (several FPGAs, ASIC) can be selected. Makefiles have been added to automatically create synthesizable projects for different development environments. It is however not possible to extract a very small processor with only basic arithmetic operations only. Therefore it is unlikely that this processor can be used as DSP.

The width of the instruction and data paths are 32-bits and are fixed within the design. The AHB cannot be used to connect to another peripheral directly so a controller is necessary to successfully make use of this bus. Unfortunately this controller is rather large and consumes a considerable amount of resources.

Regarding the configurability of both processors a trade-off need to be made between a light-weight processor without many features and a large, complex processor providing many features. The considerable amount of configurability embedded in LEON3 comes at a price of being hard to use and modify. For research projects a small implementation will be easier to handle and debug. More importantly the simulation speeds of aeMB will be most likely much higher than LEON3 which is important for a design that needs to be modified constantly.

2.5 Design quality

Several general recommendations and requirements are presented in Section 1.5.2 and 1.5.3 to improve usability and portability of a design. Following these recommendations leads to a design which can be used and modified by a team in stead of being dependent from an individual.

Projecting the mentioned design recommendations on aeMB leads to the following considerations. The design is synchronous on most component interfaces, but contains a considerable amount of asynchronous glue logic. Within the code many delays are inserted for simulation purposes which is generally a bad idea. There is no common modeling style which makes the code hard to read. The functionality of the components is not documented and needs to be extracted from the code. The header file does not contain significant information and the signals are not documented nor follow any comprehensive naming convention.

No clear and intuitive design style is followed. This ad-hoc implementation is therefore even more difficult to understand. The lack of a record-like structure in Verilog results in large lists of signals to instantiate and connect internal components. The existence of an ASIC implementation of the aeMB could not be found and since several ASIC design guidelines are violated it is doubted if this design can be implemented in a semi-custom process.

The aeMB microprocessor needs some rigorous improvements and additions to increase the overall quality. A common design methodology is needed to improve the readability, and the use of parameters can be improved. Since it was not an easy task to generate software that worked on this processor, this needs to be investigated more. Since it is known that there is a working uClinux[3] version available for aeMB, it is expected that this will not be a problem.

LEON3 and its peripherals is very portable, which is proven by the fact that designs are available for many development boards. A preconfigured implementation targeted at ASIC technology is available. New designs and configurations can be easily added and configured due to the hierarchical structure of files, packages and libraries. Technology dependent components are separated from the design and placed in technology dependent libraries. Therefore the design can be very portable.

When LEON3 is used as a stand-alone component (and not inside a simulation environment), several methods are available to load and start a program. These methods are based on RS232, JTAG, Ethernet or USB connections. The microprocessor uses a boot loader to configure hardware and start normal program execution. A utility is available to encapsulate the boot loader and program in a Programmable Read Only Memory (PROM). Alternatively large programs can be stored on a compact flash card and loaded using the boot loader.

A drawback of LEON3 is that the data memory bus uses the quite complex AHB protocol. This reduces the opportunity to quickly add and test custom peripherals [20]. It will be a time consuming task to design a wishbone adapter in order to connect existing co-processors to this bus. This reduces the usability aspect of this processor.

The number of features of aeMB are very limited and the processor need rigorous optimizations. On the other hand, LEON3 is too large complex to be quickly integrated in research projects—especially due to the AHB bus, but offers a quite complete set of features and peripherals.

[3]uClinux is a Linux based operating system targeted at embedded processors which does not have a memory management unit

2.6 Conclusion

In this chapter five open source processors are compared and evaluated: LEON3 (Sparc), aeMB (MicroBlaze), OpenFire (MicroBlaze), OpenRisc (ORBIS) and Plasma (MIPS). Since a considerable amount of projects use one of these processors, the development of these processors are regarded to be mature enough to rely on. First the features of the architectures and implementations were reviewed. For each processor it was determined whether it suits our needs our if it needed modifications or additions. It turned out that LEON3 and OpenRisc offer the most complete set of features. OpenFire, aeMB and Plasma might be a little too limited.

Subsequently processor speeds and resource requirements were evaluated. It was found that Plasma uses many LEs for a processor without any additions and that it is terribly slow. OpenFire and aeMB offered the best performance when taking both the required hardware resources as well as clock frequency into account. It appeared that OpenRisc did not give any advantage with respect to LEON3 and that OpenFire did not give any advantage with respect to aeMB. Furthermore, Plasma appeared to be an expensive solution with very few features. At this point Plasma, OpenFire and OpenRisc were excluded from further research.

The tool chain and instruction set efficiency of the two most promising candidates—LEON3 and aeMB—were taken into account. It turned out that the program size for MicroBlaze is structurally larger than the program size for the Sparc architecture. This is probably due to the register windows of Sparc. The execution of the program also takes less time on the Sparc architecture. The better performance of the Sparc tool chain is completely compensated by the differences in clock speed. In the end, MicroBlaze obtains lower execution times than LEON3.

Both tool chains are based on GCC and instruction set simulators are available for both architectures as well. Standard embedded libraries like LibC are freely available as well. The inclusion of the tool chain and instruction set efficiency did not lead to a better insight in the selection of a processor core.

LEON3 has outstanding configurability and portability. It offers by far the most features and additions of all free and open source processors available today. Nevertheless it is unfeasible to remove AHB completely from the design and replace this by another bus. AeMB does not offer additional interfaces and controllers, and does not offer many configurable options besides a barrel shifter and multiplier. AeMB needs quite some improvements to make it usable in our projects.

Finally the quality of both processors was evaluated with respect to the recommendations presented in Sections 1.5.2 and 1.5.3. LEON3 is a nice example of efficient use of VHDL constructs like libraries, packages and records—although the use of generics might have been pushed to the limit. Generic parameters are used for nearly everything, which still results in incredibly long component instantiation lists. AeMB does not provide a common organization since additional plug-and-play components are not included. It is also not designed using a well-defined modeling style.

An important aspect of an embedded processor for research purposes is that it can be reduced in size when necessary. This contributes to a better insight in the behavior of the design, and reduces simulation time. LEON3 appears not to be able to meet this requirement. The AHB which is difficult to configure is hard to use in an environment aimed at quick prototyping. Due to these aspects it is concluded that best results can be obtained by improving aeMB instead of reducing LEON3.

Design and implementation 3

Within this chapter the design and implementation of MB-LITE is presented. Due to several problems with aeMB it turned out that this processor could not meet all design goals satisfactorily and that a more structured design approach was required. In order to overcome the issues imposed by aeMB, it was unavoidable to design a completely new processor from scratch. Our design goals have been achieved by following a proven strategy: the two-process design methodology.

In Section 3.2 the implementation of MB-LITE will be discussed. The approach which was followed to achieve all goals is presented in Section 3.2.2. Subsequently the MicroBlaze architecture will be briefly reviewed in Section 3.2.3. Using the description of the architecture, the four basic components of the pipeline will be introduced and their functionalities and responsibilities will be defined. Details about the implementation of each component will be given in Section 3.2.4.

Before MB-LITE is introduced, a short overview is presented in Section 3.1 of what needs to be changed to aeMB in order to achieve our design goals and which restrictions are encountered.

3.1 Improving aeMB

AeMB is a light-weight processor and has quite high performance. The wishbone buses on both the instruction as well as data side of this Harvard architecture can be used to easily connect existing devices. The MicroBlaze architecture has a single interrupt which is implemented in the aeMB as well. The package comes with a small test bench which tests several basic functions like arithmetic operations, the FSL bus and the interrupt handling system.

In Section 3.1.1 issues concerning software are discussed and the necessary changes while in Section 3.1.2 hardware issues are reviewed and improvements are discussed. Several difficulties and limitations were encountered during this process, these will be discussed in Section 3.1.3.

3.1.1 Software improvements

The software provided in this package consists of a single program which invokes six test routines. First the interrupts are enabled which is followed by the interrupt test. A loop is entered while an interrupt is triggered from within the test bench to exit that loop. Next the memory allocation is tested by calling the library function MALLOC. After this is finished the FSL bus is tested by using PUT and GET instructions. Subsequently a Fibonacci number is computed using two methods and the results of these two functions are compared and the Greatest Common Divisor is computed using the Euclidean algorithm. Finally the computation of floating point numbers are tested to find the root of a number using Newton-Raphson's method. Since floating point numbers are not natively supported these are emulated in software by using library functions.

After executing the test bench all tests finished successfully except for the memory allocation test. This would be the first point of attention for improvement of the aeMB. Compiling other software appeared to be a complex task. Several header files needed to be included to be able to generate software which works on this processor. Failing to include the appropriate header files resulted in compilation warnings or software which compiled without any hassle but did not execute as expected. Such problems are very hard to find so a solution had to be found for this.

A shell script was provided to compile the test bench while invoking the tool chain with the correct compiler options. To make the tool chain independent from the platform it is desired to use a Makefile to generate the software. It was furthermore found that a mix of C and C++ files were used throughout the test bench. Since C++ requires additional headers to construct and destruct objects this is not an efficient choice for embedded application development. We had to get rid of the C++ files and convert these to plain C files, preferably adhering to the ANSI ISO C standard.

A last issue concerning software was that to enable hardware threads to work properly code optimization is necessary. The inability to generate software without optimizations has several consequences. First of all it is hard to check for hardware problems concerning correct instruction execution because it is hard to trace the code manually. Secondly a good comparison with other processors could not be done properly: enabling optimizations in a benchmark (e.g. Dhrystone) will damage its structure which reduces the reliability of the tool.

3.1.2 Hardware improvements

Since aeMB is written in Verilog and we like to take advantage of the abstraction levels offered by VHDL such as libraries, packages and records we tend to rewrite the processor in this language. This should improve the processor structure and provide better insight in the design. The processor is rewritten in accordance with the two-process methodology as presented in Section 1.5.3 to obtain even more insight into the design and especially in the pipeline timing.

To save hardware resources the FSL bus will be made configurable since it is generally not needed: the memory on FPGAs (e.g. BRAM on Xilinx devices) is capable of feeding the pipeline fast enough without compromising for speed. Additionally this will save some more hardware resources.

The register file is implemented using three dual port asynchronous memories, which cost many LUTs on Xilinx devices. It will be examined if these can be changed in synchronous equivalents to make advantage of the available BRAM.

This is desirable since synchronous memories provide the best results with respect to portability and additional performance might be gained.

While synthesizing the microprocessor a considerable amount of warnings were generated. Most warnings point out that there are inconsistencies in the design like unused signals. By getting rid of these warnings more severe issues will be easier to find.

For the first step an automated translation tool was used to convert the Verilog code to VHDL (X-HDL from X-Tek Corporation). Nevertheless it took some effort to obtain synthesizable code out of these results. And worse, the processor did not work as anymore. Finally the translation of the test bench had to be done completely by hand since this component uses constructs which are very specific for Verilog like printing text on the console screen.

After it was discovered that the processor did not work after the translation was performed a structural method was used to repair the processor. A mixed design using Verilog and VHDL was created and the components were replaced one at a time while maintaining correct functionality. Differences were found and repaired on the level of signals by comparing the functionality of the hardware with Verilog equivalents using compare functionality of ModelSim. Within several weeks a working VHDL implementation of aeMB was completed.

In the next step the ad-hoc code was structurally optimized and clarified with comments. All components were optimized by adhering to the two-process design methodology. Furthermore records, libraries and packages were introduced to improve portability and to reduce the length of the component instantiations. The FSL bus was made configurable to save hardware resources.

3.1.3 Limitations

While translating the Verilog aeMB processor several issues were encountered. The information about the design which was provided in the documentation, the headers of the files and the comments included in the code was insufficient to fully see through the complete design. This limited the speed of the translation and made several adjustments quite impossible. It turned out that some components used asynchronous logic at the interfaces, especially the modules related to cache, pipeline control and instruction decoding could not be easily modified nor efficiently translated in a two-process architecture.

More serious issues were encountered later on. The processor gave the impression of reaching very high performance in terms of clock frequency. As can be seen in Figure 3.1 the acknowledge signal repeats every five cycles, and this holds for a large part of the simulation. AeMB is a single issue architecture which alternately executes two threads—a so called barrel processor. Execution time of single-threaded software is likely to degrade in performance since the unused thread is not occupied.

An instruction cache is used to minimize the latency of the wishbone bus and optimize the instruction throughput. It had been tried to measure the CPI this

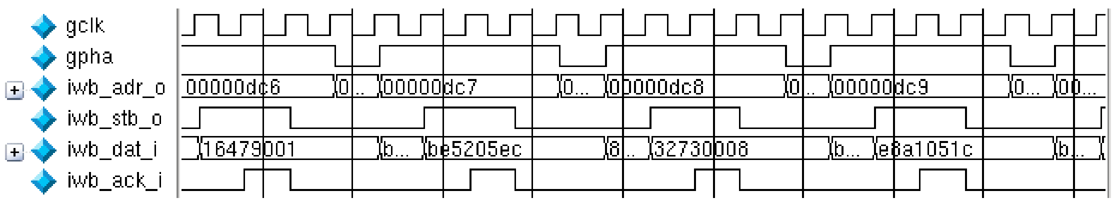
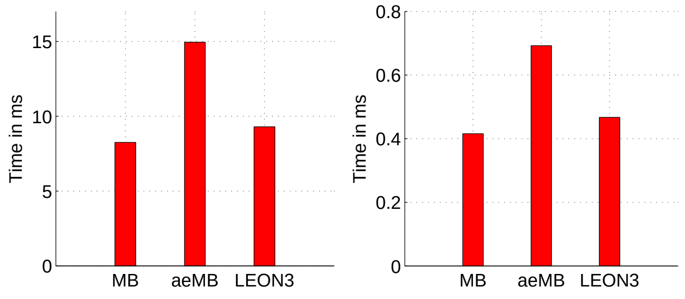


Figure 3.1: Instruction bus timing of aeMB microprocessor



(a) Dhrystone execution time (b) Fibonacci execution time

Figure 3.2: Corrected Dhrystone benchmark results

processor would obtain in real. As already pointed out it is hard to compile working software for reasons that are unknown. The Dhrystone benchmark could not be executed on the aeMB since the execution fails in the software emulated division procedure divsi3. Therefore an estimation has been made using the original aeMB test bench (refer to Section 3.1.1). During execution of this test bench the phase signal which controls the thread toggles 93153 times. Because this test bench only makes use of a single thread half the number of cycles are wasted. Using this approximation it can be estimated that effectively $\frac{93153}{2} \approx 46577$ instructions have been executed. Executing these instructions takes 157450 clock cycles. The CPI of this single-threaded program is thus $\frac{157450}{46577} \approx 3.38$ which is more than twice the CPI a MicroBlaze processor would generally obtain. The corrected performance results of the MicroBlaze, aeMB and LEON3 processors are shown in Figure 3.2.

Another issue was found during the translation of the Verilog code to VHDL. While trying to transform the structure in a two-process design many glitches were found in the control signals as shown in Figure 3.3. This might not only result in hardware which is not working correctly but this makes the timing of the components incredibly hard. To get rid of these problems it was tried to combine all these components into a single module so the interfaces were synchronous again. Unfortunately this did not prove to be a successful strategy. The lack of documentation together with the use of asynchronous signals made it impractical to modify the structure.

When evaluating the wishbone system buses other irregularities were encoun-

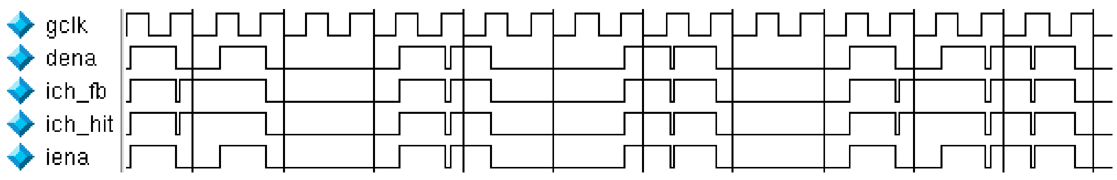


Figure 3.3: Glitches in the control signals of the aeMB microprocessor

tered. When the acknowledge signals of one of the buses was delayed the bus did not behave as required. The acknowledge signal had to be deasserted within three cycles or the program execution would crash. This violates rule 3.55 of the wishbone specification revision B3. Although this might not be a problem for most designs the wishbone bus is not very robust.

3.1.4 Conclusion

When aeMB was modified in order to reach the goals as described in Section 1.2 several pitfalls were encountered which could not be easily solved. Due to insufficient documentation several architectural properties were found to be disadvantageous instead of useful for a simple microprocessor design. The inflexibility of the control and cache mechanism made it impractical to modify the design in accordance with our requirements. Moreover these issues could not be solved satisfactorily and the deliberately generated glitches might lead to even more problems later on. The wishbone buses appeared not to be very robust.

It is concluded that aeMB will not meet the requirements without making rigorous and time consuming changes. Furthermore it is assumed that designing a microprocessor from the ground up will lead to a working implementation much quicker. Therefore the aeMB design is left as is and the focus was put on the design of a portable, customizable microprocessor for rapid system prototyping.

3.2 MB-Lite

The MB-Lite microprocessor is an implementation of the MicroBlaze architecture. This RISC processor closely resembles the well known DLX and MIPS architectures.

The MicroBlaze tool chain is well developed and widely available. Many software packages are available for this architecture. Petalogix provides amongst others a free MicroBlaze GCC compiler and debugger, a port of the uClinux operating system together with all common standard libraries. This architecture is therefore regarded as a reliable base to provide a reference design for an embedded microprocessor.

3.2.1 Design methodology

In order to achieve the design goals a structured design approach is necessary. To this extend all components have been designed using the two-process design methodology. Extensive research and experience on this methodology has shown that better synthesization results can be obtained because synthesizers can better apply their optimization strategies which will eventually result in smaller and faster designs. Due to the improved component structure designs become much more readable—which results in better maintainability and in a more durable design.

High-level language constructs in VHDL offer more abstraction and are widely supported by modern simulation and synthesis tools. A good application of high-level constructs result in even better readable code. Functions, procedures and type definitions in shorter code fragments and code that can be easier tested for correct functionality. This also improves the quality and reliability of the design.

3.2.2 Implementation goals

To achieve a processor which can be used in all projects, standard toolkit programs need to work with MB-LITE without modifications or additions. Our target is to design a processor which can run C or C++ programs compiled with MB-GCC - the standard GCC based compiler for the MicroBlaze platform - which is available in the EDK of Xilinx. Interrupts can be used in the same way as the original MicroBlaze implementation by defining a function with the type declaration VOID __ATTRIBUTE__ ((INTERRUPT_HANDLER)).

The key of the MB-Lite design is to create a simple and well defined structure in hardware as well as in the file and library organization. This will enhance productivity of users and makes sure that the design can be maintained and improved in the future by other contributors in which way the open source concept is used to the largest extend. The data path and control units of the MIPS processor will be implemented as described in the popular books of John L. Hennessy and David A. Patterson about computer architecture [22, 23]. The naming conventions will be chosen in accordance with these books whenever applicable. Several changes need to be made to make this organization compatible with the MicroBlaze architecture, but it is tried to follow the concepts as applied in that architecture as close as possible. In this section it will be described what is to be achieved and how this can be achieved. First the memory organization of the MicroBlaze is discussed and a more generalized model is presented. Second the implementation methodology is presented which will prove why and how small deviations to the MIPS processor needs to be made. It is not attempted to design a replacement part for the commercially licensed MicroBlaze processor. Therefore we will define which subset of the MicroBlaze will be implemented.

The MicroBlaze architecture has up to three interfaces for memory access. The Local Memory Bus (LMB) provides low-latency access to a small amounts of memory. The CoreConnect bus is used to connect peripherals to the processor. This bus is a fully synchronous, non multiplexed, pipelined bus topology and supports concurrent read and write transactions. A complex memory topology can be build to efficiently sustain devices with different throughput and latency requirements. The Processor Local Bus (PLB)[24] is a high-bandwidth, low-latency bus and can be used to connect high performance demands such as memory access. For peripherals with less throughput and latency requirements the On-Chip Peripheral Bus (OPB)[25] can be used. These devices can be accessed through the OPB to PLB bridge unit. The last interface provided by the MicroBlaze architecture is the Xilinx Cache Link (XCL) which is used as streaming interface between caches and external memory controllers.

Within the MB-Lite concept the memory bus interface is generalized as much as possible to a simple core interface. A minimal set of data memory control wires is provided which can be used to connect immediately to a standard memory component. The "bare" core thus provides a standard memory interface as primary bus to make it possible that a range of interfaces can be designed for the MB-Lite. A fully functioning Wishbone Bus Interface is provided in the form of an adapter to show the easily extensible processor features.

A minimal subset of the MicroBlaze architecture will be implemented. Features which can be disabled by compiler parameters will be avoided as much as possible. Special hardware features like a FPU, multiplier, barrel shifter and compare instructions will not be implemented. Whenever necessary these can be replaced by software libraries as provided by Xilinx.

3.2.3 MicroBlaze architecture

The RISC processor as described in Section 1.5.4 will be used as a base to implement the MicroBlaze architecture. This section contains an introduction to the MicroBlaze architecture (for detailed information refer to [26]) as well as the description of the tasks of the four main components which are used to implement the five pipeline stages. In this section we will take a look at the interaction between these four components and the responsibilities of each of these processor parts.

Two basic instruction types are defined which are called Type A (i.e. R-Type or Register Type) and Type B (i.e. I-Type or Immediate Type). The operation to be performed is identified by a 6-bit field and called the operand field (i.e. opcode). Type A instructions contain up to two source register operands and one destination register operand. Type B instructions have one source register, one destination register and a 16-bit immediate operand. The instruction format and corresponding bit numbering of both instruction types is shown in Figure 3.4. The MB-Lite implementation of this architecture prefers linearly ordered bit numbers instead of reversed bit numbers since this is common practice in hardware design.

If an operation requires more than the available 16 bits immediate value (e.g. for accessing a high memory address or adding large numbers) the instruction can be preceded by the IMM instruction to preload the upper half of the immediate value. The IMM instruction thus only affects the immediate value of the subsequent

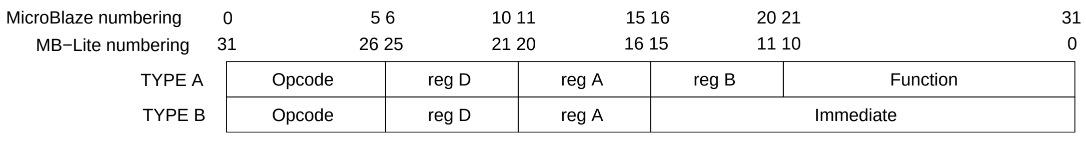


Figure 3.4: Instruction Format of the MicroBlaze architecture

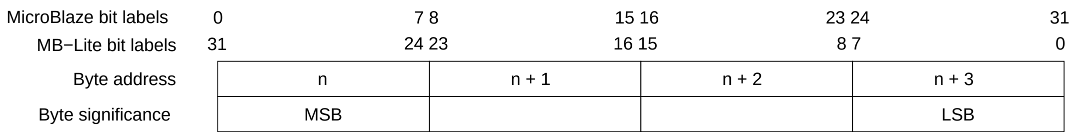


Figure 3.5: Data Format of the MicroBlaze architecture

instruction. By using the IMM instruction all 32 bits of the memory can be addressed effectively.

Thirty two general purpose registers are available. Some of them are “reserved” for special purposes such as return addresses of interrupts. MB-Lite obviously follows these conventions to keep the hardware specification compatible with the Application Binary Interface (ABI). Furthermore thirty two special purpose registers are defined. These contain amongst others information about the current configuration of the processor and the Program Counter (PC). The current MB-Lite version does not support instructions related to special registers since these instructions are generally not needed.

MicroBlaze uses Big-Endian bit-reversed form to number individual bytes and bits. Memory accesses are one byte wide (i.e. bit-addressing is not supported) and the memory reads and writes must be aligned. Therefore we can also safely change the bit numbering of the MicroBlaze architecture to our preference. Bit zero thus represent the Least Significant Bit while bit 31 is regarded as the Most Significant Bit. The data format, byte addressing as well as the bit and byte numbering for the data format is shown in Figure 3.5.

All instructions implemented in MB-Lite have the same latency as defined in the MicroBlaze architecture specification. Most instructions have a single cycle latency. Taken branches with a delay slot have two cycles latency and taken branches without a delay slot three cycles.

The IF module feeds the pipeline with the requested instruction and stores the current program counter. The PC corresponds to the address of the instruction divided by four and is also referred to as the instruction number. The PC is incremented automatically every clock cycle. The IF stage is responsible for handling the system reset, branch and hazard signals as well.

During the ID stage (also called the Operand Fetch (OF) stage) it is first determined which instruction needs to be executed. The instruction to be executed is determined using the reset, hazard, and hazard recovery signals. The operands corresponding to the instruction are fetched from the register and the instructions are decoded in control signals. The control signals travel along with the execution operands in subsequent stages. This simplifies the control of the pipeline and makes sure that the implementation can be easily debugged: at every time instant it can be easily determined what needs to be done and what the final results are. The decode stage takes care of interrupts by evaluating if the normal execution flow can be interrupted without causing problems and the control signals will be overloaded with a branch to the interrupt routine.

If a data hazard is encountered which can not be solved by forwarding, a stall will be executed which will delay the pipeline. Control hazards which are caused by taken branches require the pipeline to be flushed, i.e. instructions that should not have been executed will be erased. Both hazards are detected and handled in the ID stage. Both situations cause a NOP instruction to be inserted instead of the originally fetched instruction.

In the EX stage all necessary arithmetic is done. Due to data dependencies as explained in Section 1.5.4 the correct values of registers A, B and D can come either from the execution unit itself (single level forward), from the memory unit (second level forward) from the writeback stage (register bypass or third level forward) or from the register file. The third level forward is required to reduce the requirements on the memory used in the register file. Very few memories can read and write the same address during a single cycle, which could potentially lead to memory hazards. If a register value is forwarded from memory, the result still needs to be aligned.

The EX stage also checks if a branch condition is met and takes care of asserting the appropriate branch control signals. The branch decision depends on the relation between the value of register A and zero (e.g. greater than, smaller than or equal to zero). Therefore the ALU can be used to compute the branch target address while simple compare logic is used to evaluate the branch condition.

It is the task of the MEM module to control the interaction with the data memory. It generates the byte select signals (i.e. a 4-bit enable signal), enable and write enable signals. In case of a branch the current program counter needs to be stored the ALU result must be replaced by the program counter. In organizational terms it would made more sense to do this during the EX stage. However, because the logic depth of this stage is already quite high it was decided to postpone this selection to the memory stage. The pipeline will be flushed anyway if a branch is taken, no problems concerning forwarding will be introduced.

The inputs and outputs described previously are drawn in Figure 3.6. Three types of signals are distinguished: data wires (green), address wires (blue) and control wires (red). On the top of the figure the writeback signals are drawn while on the bottom the signals for data- and control hazards are shown.

3.2.4 Detailed design

Now it is known which components are needed and which functionality they must provide we can focus on the implementation of each component. Pipelining is an effective method to exploit some of the instruction level parallelism inherent to a

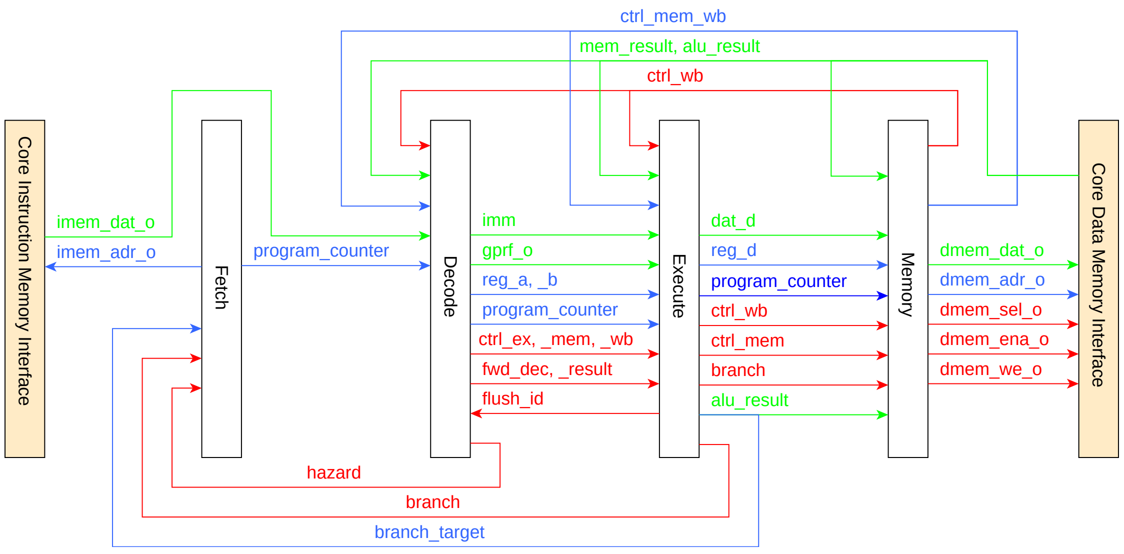


Figure 3.6: Interface signals of the MB-Lite processor

micro architecture. To increase the frequency of the processor it is important to take advantage of parallelism within each component by carefully evaluating all dependencies and reduce the internal dependencies to a minimum.

3.2.4.1 Instruction Fetch

The IF stage generates the signals which are necessary to fetch an instruction from memory. The control signals going into the instruction memory needs to be available before the next clock-edge. At the positive clock edge the information will be available immediately.

Three address sources needs to be distinguished in order to fetch the required instruction. The primary source is obviously the PC but the hazard and branch signals need to be taken into account as well. At the positive clock edge the PC will be incremented automatically. Secondly the external reset signal needs to be taken into account. When a reset signal is asserted the PC is set to zero and the corresponding instruction is fetched. Since the memory output is synchronous (positive edge triggered) it is directly connected to the input of the ID stage. The ID stage has to make sure that when the reset signal is asserted NOP instructions are inserted in the pipeline instead of continuously feeding the first instruction.

If a branch occurs the PC is set to the requested branch target and the corresponding instruction is immediately fetched from instruction memory. If a hazard occurs the PC remains unchanged. Since a hazard is detected in the decode stage and available at the end of a clock cycle the IF module has already loaded the next instruction. The ID stage is thus responsible for reissuing the instruction which caused the hazard. A block diagram of the IF stage is shown in Figure 3.7.

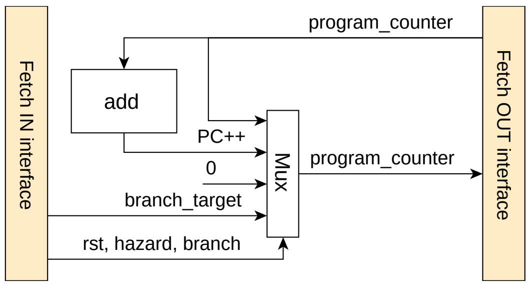


Figure 3.7: Block diagram of the MB-Lite fetch stage

3.2.4.2 Instruction Decode

First the program counter and instruction to be executed is determined by evaluating the reset, flush and stall signals. In case of a reset signal the instruction and program counter will become zero. Subsequently it is determined if the sequence of the previous instruction followed by the current instruction results in a hazard. If this is the case the current instruction and program counter are latched and a NOP operation is inserted. In subsequent cycle the latched instruction and program counter will be automatically issued again. If all these exceptions are not true execution will continue normally, i.e. the instruction which is fed into the ID module will be selected and executed.

Subsequently it is determined if an interrupt is triggered and can be handled. Several conditions have to be evaluated to see if this can be done without causing problems for the general program flow. The interrupt is latched if it can not be taken care of immediately. The instructions that prohibit immediate handling are a branch, return statement or an instruction in a delay slot. Furthermore the instruction must not be preceded by an IMM instruction or the interrupt will be delayed.

If an interrupt can be handled safely a branch is executed to the interrupt handling routine which is located at instruction memory address X10. The writeback register address is set to 14 and the writeback signal is asserted. This will force the current program counter being written into the interrupt return address as specified by the microblaze architecture. After the interrupt routine has been executed normal program execution will continue.

If there is no interrupt to be handled, the instruction is further decoded in control signals. The immediate value to be used in subsequent stages is determined. The most significant bits of the immediate value might have been set by a previous IMM instruction, or otherwise the immediate value will be sign-extended from the lower 16 bits of the instruction.

While the immediate value is determined the control signals for subsequent stages can be determined as well. Since this is a straight forward translation of the architecture in control signals only a description of the purpose of each signal is given in Table 3.1. The control signals are grouped in three records representing

Table 3.1: Description of the decoded control signals used in MB-Lite

Signal name	Module	Values	Description
alu_op	Ex	ALU_ADD, ALU_OR, ALU_AND, ALU_XOR, ALU_SHIFT, ALU_SEXT8, ALU_SEXT16, ALU_MUL, ALU_BS	Set the ALU operation
alu_src_a	Ex	ALU_SRC_REGA, ALU_SRC_NOT_REGA, ALU_SRC_PC, ALU_SRC_ZERO	Set the input operand A of the ALU
alu_src_b	Ex	ALU_SRC_REGB, ALU_SRC_NOT_REGB, ALU_SRC_IMM, ALU_SRC_NOT_IMM	Set the input operand B of the ALU
operation	Ex	NONE, CMPU	Special instruction
carry	Ex	CARRY_ZERO, CARRY_ONE, CARRY_ALU, CARRY_ARITH	Set the carry input of the ALU
carry_keep	Ex	CARRY_NOT_KEEP, CARRY_KEEP	Carry behavior
delay	Ex	0, 1	Delay slot
branch_cond	Ex	NOP, BNC, BEQ, BNE, BLT, BLE, BGT, BGE	Branch condition
mem_read	Mem	0, 1	Read from memory
mem_write	Mem	0, 1	Write to memory
transfer_size	Mem	WORD, HALFWORD, BYTE	Size of data element
reg_write	WB	0, 1	Write register back to register file

the pipeline stage where they are used.

The instruction STORE WORD requires three register *values*. The memory location where the value will be stored is composed of the sum of the values of register A and B while the value which needs to be written is in register D. Therefore the register file must be able to read three values from the register at the same time[1]. Furthermore a value might be stored while the same or another register value is read. In case the a new value is written while the same register is read it has to be made sure that the most recent value is used, i.e. the one that is being stored.

The default MB-Lite register file consists of three dual port synchronous 32x32 bit memories (1 clock, 1 write port and 1 read port). The read and write behavior does not need to be specified explicitly since it can be transparently bypassed using the forwarding logic in the EX stage. The memory file is therefore less dependent on the implementation platform. A block diagram of the organization of the decode stage is shown in Figure 3.8.

[1]This is another deviation from the MIPS implementation since MIPS requires at most two register values: one for the data and one for the address.

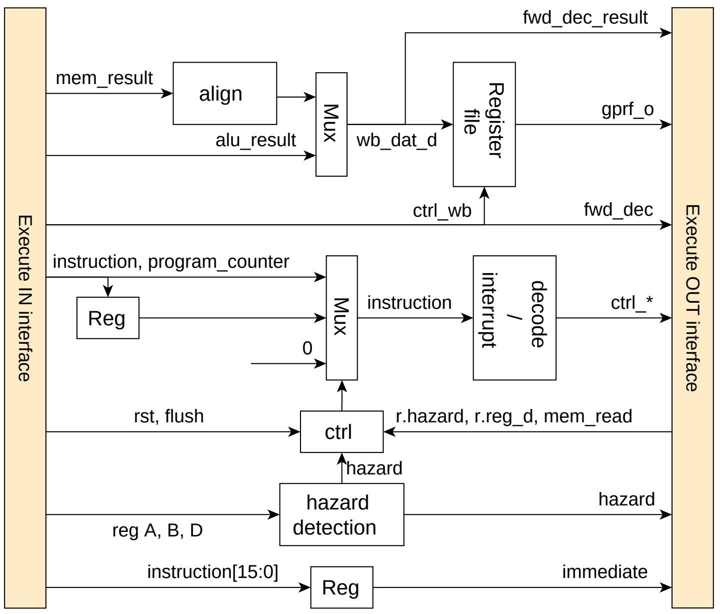


Figure 3.8: Block diagram of the MB-Lite decode stage

3.2.4.3 Execute

During the execution stage all necessary computations will be performed. Before the actual computation can be executed the correct operands have to be selected first. Since a fully synchronous design is demanded the data values depend on the results of all other stages except the IF stage. Forwarding needs to be applied to all three register values because a store operation requires up to date values of all registers.

First the register values from the ID and WB stage are selected using the function SELECT_REGISTER_DATA. If the register address is zero the data will be cleared. If a register other than zero is requested, the value which has been read from the register file or written back to the register file will be used. Now the forwarding conditions for the other stages are evaluated. If one of the conditions is met the most recent value will be used, otherwise the original value from the register file will be selected.

Four different values can be used as carry in for the ALU. We can use a previously generated carry, the most significant bit of operand A, a zero or a one. A carry-keep instruction makes sure that the carry will be spared for another clock cycle.

Care has to be taken when the CMP instruction is executed. Besides the computation of the difference of registers A and B the Most Significant Bit needs to be modified to represent equality or inequality. In case of signed comparison no changes need to be made, but when unsigned comparison and addition is being performed the Most Significant Bit must be inverted if one of the two—but not

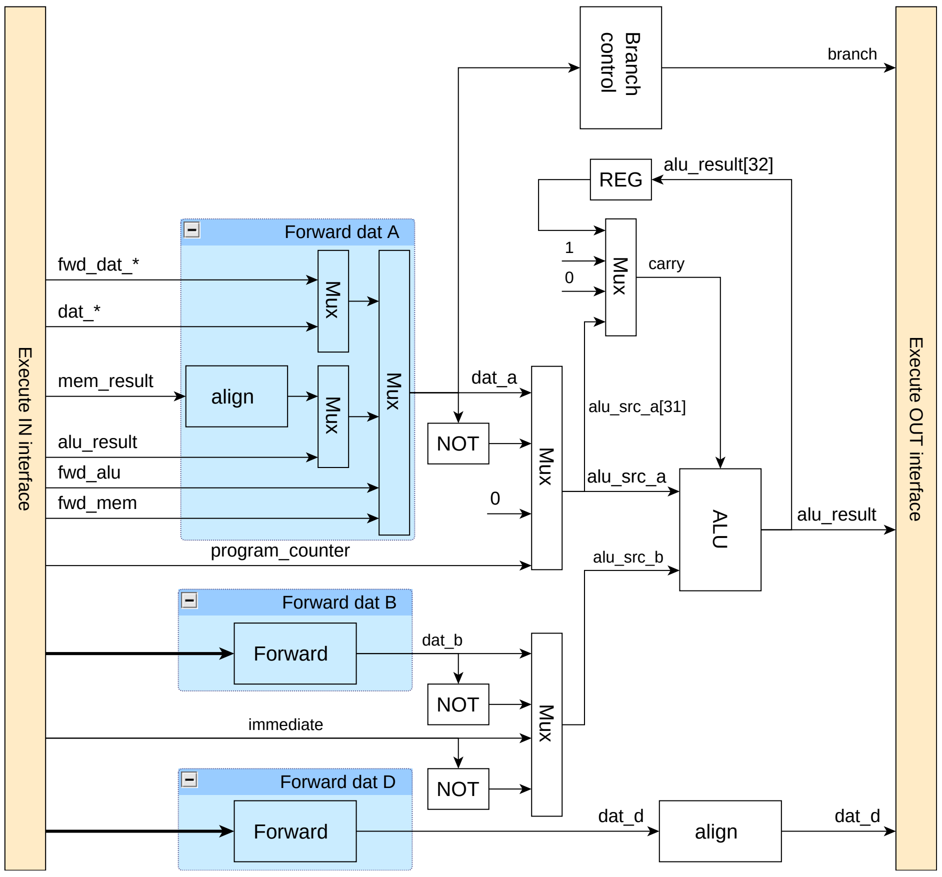


Figure 3.9: Block diagram of the MB-Lite execute stage

both—operands is negative.

A block diagram of the execute stage without the control signals is shown in Figure 3.9. The flush action which clears all control signals is not shown either to keep the image as clear as possible. Forwarding is drawn only for the data value of register A since this process is exactly the same for register B and D. The compare instruction is considered to be in the ALU.

3.2.4.4 Memory

The 32-bit memory interface generates the control signals for a standard memory component. The interface consists of an address bus, two separate data buses for reading and writing, an enable and write enable signal and a separate 4-bit signal SEL_O to indicate which byte or bytes of the 32-bit word is requested. A synchronous memory component with four write enable signals can be connected directly. The only difference is that a regular memory component have a 4-bit write instead of a byte select signal. The select signal needs to be combined with the write enable signal to obtain a valid 4-bit write enable signal suitable for block

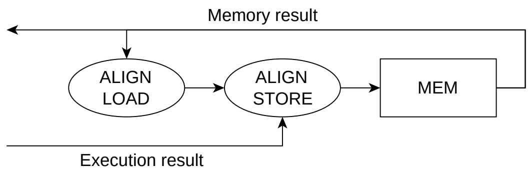


Figure 3.10: Block diagram of the MB-Lite memory stage

ram. Furthermore the data memory input has an enable signal to be able to halt the processor.

The wishbone interface signals correspond closely with the MB-Lite data memory interface signals. Only the handshake protocol signals (ACK, STB, CYC) are missing. This allows memory transactions to complete within a single clock cycle which is not possible when using the classic wishbone protocol. The byte select signal is generated using the function DECODE_MEM_STORE. The alignment of a data element which is being stored in memory is already handled in the execution stage. Alignment of the loaded data elements is distributed to the decode and execute stages to avoid asynchronous components in the design.

When a load instruction is immediately followed by a store instruction a data hazard might occur if a register which has just been loaded is immediately written to a new memory location. Forwarding is successfully implemented to solve these hazards as shown in Figure 3.10. However, this requires additional forwarding and alignment logic. Applications with tight resource constraints can take benefit of a parameter to insert a stall instead of including additional forwarding logic. Since these hazards are quite rare only a small reduction in performance should be expected.

3.2.5 Address decoder

To simplify connecting multiple devices to the MB-Lite core a highly configurable address decoder was added to the design. This address decoder can be directly attached to the data memory interface and splits the data bus using multiplexers in a configurable number of slave interfaces. Not only the number of slaves can be specified during design, but also the memory map can be given transparently to the decoder by using VHDL generics. This makes the addition of additional peripherals an extremely simple task.

The topology of the memory decoder is straight forward. First the address requested by the master for reading or writing is decoded using the generic memory map. This results in a signal which had been called chip-enable and has the property that always one bit of the vector is asserted. After the address has been decoded one of the slaves is activated by asserting the corresponding enable and write enable signals. The other signals going into all slaves (DAT_O, ADR_O and SEL_O) can be directly connected since these don't influence the state of the slaves.

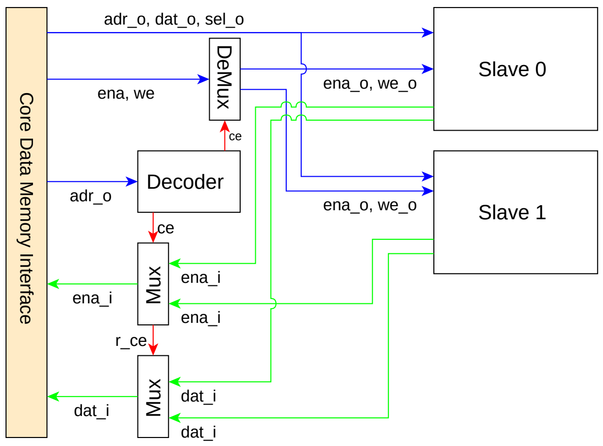


Figure 3.11: Design of the MB-Lite address decoder implemented with two slaves

Connecting the correct slave signals to the memory signal is a little bit more challenging. Since the MB-Lite core expects the DAT_I signal to be registered and the chip-enable signal is not registered it has to be made sure that the registered output signal is connected to the master. Therefore the chip-enable signal is stored in a register and the DAT_I signal will be forwarded using this registered value. A block diagram of the decoder is shown in Figure 3.11.

3.2.5.1 Wishbone bus adapter

Memory transactions are in general the most frequent instructions to be executed by a processor and account roughly for 30 percent of all executed instructions. The classic wishbone bus protocol uses a synchronous handshake protocol to interchange data and will take at least two clock cycles to complete. Implementing this as standard interface for all components would therefore result in a major performance degradation. Therefore the MB-Lite core can be optionally connected to a wishbone bus adapter. This gives the designer the freedom to design a fast memory interface with single cycle latency while giving the possibility to add wishbone compatible components with multiple latency cycles.

The modularity of the design is unimpaired due to this decision. Complete memory topologies can be build on top of the bus interface and other bus interfaces can be easily designed using the wishbone interface as example.

The adapter is responsible for disabling the core until the slave has acknowledged correct transmission of a data element. The wishbone interface is implemented according to the classic wishbone bus specification revision 3b [27]. The acknowledge signal which comes from the slave is evaluated every clock cycle as is suggested in the wishbone specification. Only the strobe and cycle signals need to be generated by the adapter using the enable and write enable signals from the core and the acknowledge signal from the slave. The Wishbone bus as implemented in MB-Lite is thus fully synchronous.

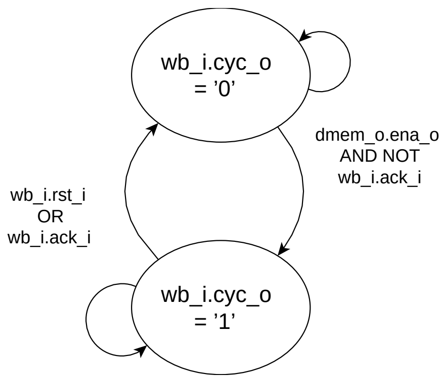


Figure 3.12: Finite State Machine of the MB-Lite wishbone adapter

The structure can be regarded as a state machine with the state being equal to the current cycle output signal. In case the bus is "available" for a transmission state zero is asserted. If the core requests a transaction by asserting its enable signal the bus goes in to state one on the premise that previous acknowledge signal is deasserted. If either the reset or the acknowledge signal is asserted while the bus is in state one (i.e. a valid transaction is in progress) the transaction will be terminated. In all other cases the states won't change. This allows for slow slaves being connected to the wishbone bus. The state machine which is used to implement the wishbone bus is shown in Figure 3.12.

During a wishbone transaction the data which is sent to a slave is stored in a register for multi-cycle transactions. Due to timing complexities it is much easier to store the data here than forcing the memory to keep the data in a valid state.

4 Results

Within this chapter we will focus on the results which have been obtained. First it will be proven that the MB-LITE implementation complies with the MicroBlaze specification in Section 4.1. In Section 4.2 numerical results on performance and resource utilization will be presented.

4.1 Verification and compliance testing

A good hardware verification process improves the quality and reliability of a design drastically. To prove that the specification is correctly implemented the design has been thoroughly simulated using many different configurations and programs. Other methods for formal verification are regarded to be unfeasible due to the state explosion problem of large designs [28].

4.1.1 Compliance testing methodology

In order to test the design for compliance with the MicroBlaze architecture a large C program was designed and compiled using the MicroBlaze tool chain (i.e. MB-GCC and MB-AS from EDK 10.1). This results in the generation of a binary in ELF format. Using the program MB-OBJCOPY a binary is generated. Using a simple utility BIN2MEM the binary format is converted into a MEM file which can be easily used with ModelSim. All software provided with MB-LITE contain a Makefile which automatically executes these steps.

The results are verified using a standard Input/Output (IO) interface which reads characters from a bus and writes these to the console screen using the package TEXTIO. The results can than be checked by inspection, or a similar program can be compiled and run on a local computer.

4.1.2 Test bench design

The software which is loaded in the instruction memory is responsible for putting the processor in many different states. This can be measured using coverage analysis to show that all possible statements have been executed. Due to the great usability of this method a test bench was designed with complete statement coverage. The design of the test bench went through an iterative process to obtain 100% statement coverage.

First a set of five standard programs are sequentially started. First the correct behavior of the interrupt signal is evaluated by entering a infinite loop. If an

Table 4.1: Coverage report summary of the test bench

Component	Number of States	Number of Hits	% Covered
Fetch	11	11	100.0
Decode	160	160	100.0
Execute	72	70	97.2
Memory	21	21	100.0

interrupt occurs the interrupt routine is called which will make sure that the infinite loop is terminated.

Second some integer arithmetic tests are being executed to test iterative function calls (fib1), loops, additions, comparisons and branches (fib2) and the Greatest Common Divisor of two numbers is computed using the Euclidean algorithm (GCD) to exercise modulo instructions. Along the way intermediate results are printed to the console screen which is not only for convenience because several standard libraries as published by Xilinx are tested as well. Printing a character to the standard output results in a sequence of library calls to determine string size, writing strings or characters and to interpret the string format constant and insert the correct data.

After the arithmetic instructions have passed the test the memory allocation routine is invoked by the function memoryTest. The floating point library is emulated in software since there is no hardware support available yet. The square root of a certain number is computed and if it corresponds with a known value this test has been successfully completed.

The Dhrystone benchmark appeared to be an outstanding tool to add to our test bench. It not only contains a complex mix of instructions and function calls, but the result is printed to the console together with the expected results. This makes the verification of correct execution very straight forward.

It appeared practically impossible to let the compiler generate all possible instructions. Some instructions are simply never used by the tool chain. A function with uncommon instructions was created to improve the code coverage of the microprocessor. During the execution of this function the instructions BEQ, BSRA, ANDN, ANDNI, MULI and SRC are invoked and the results are checked against the expected values.

The console output of the test bench included in Appendix B.1. The coverage result of this test bench is summarized in Table 4.1, while the complete report is included in Appendix B.2. It can be seen that in the execute state two statements have not been reached—which is a good thing since these are error states (c.f. WHEN OTHERS in a case statement). It is concluded that every statement has been executed at least once, which in turn proves that every instruction have been executed at least once.

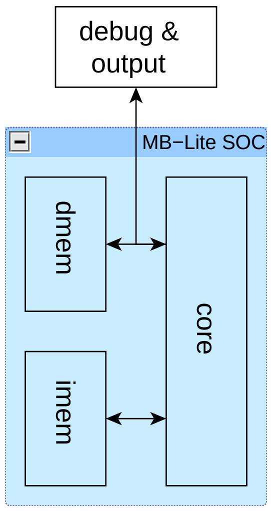


Figure 4.1: Final configuration of the verification test bench

4.1.3 Verification of the post place-and-route simulation

To check the design for errors after placing and routing the design a post place-and-route simulation have been performed using the core design. Such simulations contain detailed and pretty reliable information about delays of components and wires. The behavioral model of the core has been replaced by the VHDL model generated by Xilinx. The simulation has eventually successfully been performed using Mentor Graphics ModelSim SE 6.5 and the post place-and-route model has been generated using Xilinx ISE (EDK 10.1). The core was replaced by a timing accurate VHDL model and ideal memory and standard output was connected.

During post place-and-route simulation several issues were encountered which needed to be solved. The simulator must be set up to simulate with a resolution of picoseconds. However, the simulator refused to load the design using this setting. The solution was to disable all optimizations, which is also turned on by default.

After this issue had been solved less problems were found, but the simulator report dozens of setup and hold timing violations despite the fact that during synthesization no problems were found. Changing the clock to very low frequencies (around 10 MHz) still did not solve the issue. It was noticed that all problems occurred during the interaction with the data and instruction memories. Therefore the simulated model was extended by including the memories in the synthesized design as shown in Figure 4.1. To simplify the verification it was desired that the hello world application could be successfully executed. Due to the libraries this simple program was already quite large, it required 2048 instructions and data memory space.

After these modifications the design still did not execute as expected, although it came again a lot further without any problems. Although the memory timing warnings had disappeared the execution crashed after some time. Than it was realized that the memory topology was the only component which included asynchronous logic to align the data read from memory. It was designed in this way because memory alignment is clearly a function of a memory component. This criterion had been dropped and the alignment functionality had been distributed to both the decode as well as execute components as described in Section 3.2.4.2 and 3.2.4.3 respectively.

Finally another memory issue had come across. The block RAM templates provided by Xilinx in the language template library were used to make sure that the memory synthesized to the correct modules. The provided process sensitivity list is incomplete since the address to be read or written was not listed in the process sensitivity list. As a result values were randomly written to memory during read operations. Strangely enough the Xilinx compiler did not notice this and did not issue a warning about the incomplete list. After this final issue was solved the hello world application was successfully simulated using the post place-and-route model.

4.1.4 Verification of the wishbone bus

The verification of the wishbone bus is important since this will allow other people to attach their peripherals easily. The bus allows both the master and the slave to determine the speed of the transaction. This can be accomplished by allowing both the master as well as the slave to keep their control signals asserted for unspecified time. Therefore a slave needs to be designed which the wishbone behavior can be easily modified and verified.

A device which is easily verified is a character device which shows the result on the console screen. It will be immediately apparent if a transaction has failed. To this extend the core is connected to the address decoder which splits the memory bus in two separate buses with their own enable and write enable signals. The lower memory part is connected to regular data memory while the second port is connected through the wishbone adapter to a standard IO interface which can write data to the console screen. We can now easily modify the interface properties like bus timings without disturbing the program execution. This greatly simplifies the verification of the bus. Wishbone cycles with immediate acknowledge as well as late acknowledge have been simulated. The resulting bus cycles for reading data with a delayed acknowledge is shown in Figure 4.2.

The cycle signal (CYC_O) remains asserted from the beginning until the acknowledge signal is deasserted. The strobe signal (STB_O) remains asserted if valid information is written to the bus and will probably terminate earlier than the cycle signal. This allows the master IP core to continue execution while the bus is still being in use. The bus interface is implemented as a synchronous finite state machine so the design is fully synchronous.

Block reads and writes are currently not supported. This could be advantageous if many bus transactions take place in a short time. The compiler however has the tendency to insert many instructions before and after reading a specific port.

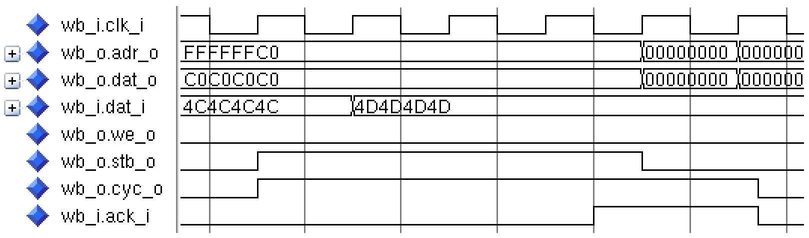


Figure 4.2: Wishbone bus read cycle with delayed acknowledge

Therefore not a lot of time will be saved if block transactions are implemented. Most of the time the bus will be free when the next transaction needs to be started.

The wishbone rule 3.55 states that an acknowledge signal may be hold in the acknowledge state and that this may not influence normal operation of the master. However, when the transmission is acknowledged the processor immediately continues with the execution of the program. If an acknowledge signal is kept asserted so long that the bus is still busy when a next transaction needs to start, the processor needs to wait first until the bus is free. This was successfully tested by creating an acknowledge delay of 100 clock cycles.

4.2 Results on performance and resource utilization

The performance of an application is formed out of the 3-tuple speed, efficiency and program size. These three ingredients are traditionally quantified as maximum clock frequency, CPI and the total number of instructions. First the focus is put on the maximum clock frequency which is probably the most important component of the performance equation.

Another primary design aspect which largely determines the implementation cost of a design is the amount of resources required. Often a trade-off have to be made when a processor needs to be selected for a project. The MB-Lite processor uses just over 834 LUTs and 355 registers, and obtains a clock frequency of 222 MHz on the same Virtex 5 FPGA as used in Chapter 2. In Figure 4.3 all processors discussed so far are shown in a scatter plot.

In terms of clock frequency the MB-Lite processor is comparable with existing designs like OpenRisc and OpenFire. AeMB and LEON3 still seems to obtain the highest clock frequency. However if the number of resources are taken into account this processor can be regarded as a very light-weight MicroBlaze implementation. Especially if the number of options are taken into account such as interrupt support and a highly configurable and extensible local memory bus and wishbone bus.

The Dhrystone and Fibonacci benchmarks are executed on these processors as well. The same files as used during the preliminary research have been executed on the MB-Lite. The number of cycles have been counted by using ModelSim. The results of these measurements are shown in Figure 4.4. The number of execution cycles of both programs on the MB-Lite is lower than on the original MicroBlaze.

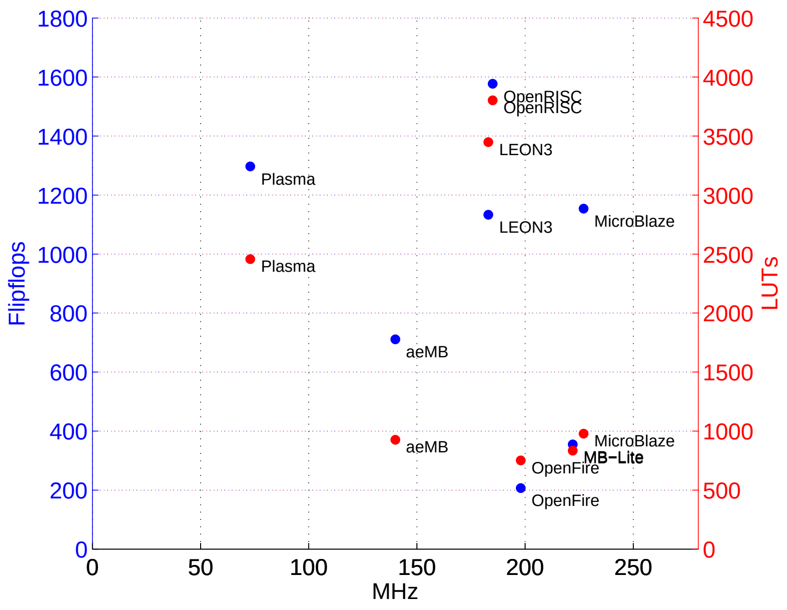
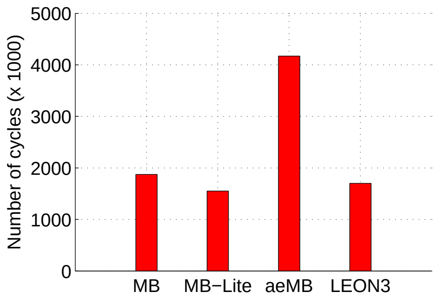
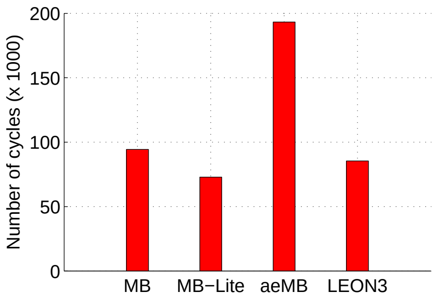


Figure 4.3: Scatter plot of processor performance including MB-Lite



(a) Dhrystone cycles

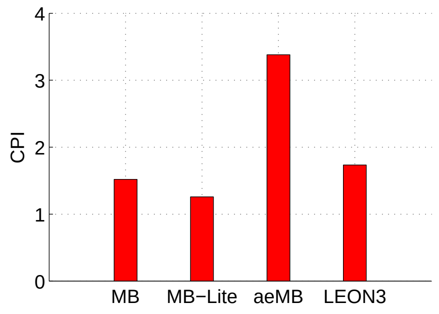


(b) Fibonacci cycles

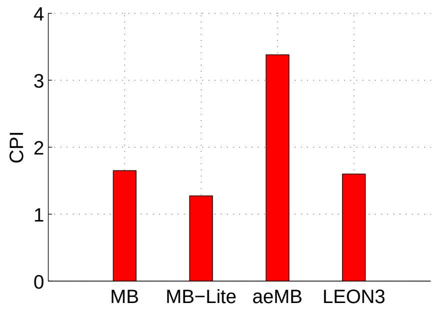
Figure 4.4: Benchmark results in number of execution cycles

Although the implementation details of MicroBlaze are not available it is probably due to the two-cycle delay of the LMB which is used for fetching instructions. It is expected that this introduces a performance penalty of 21%. It is likely that the performance of the prefetch mechanism depends mostly on the number of taken branches. This statement is supported by noting that around 20% of all executed instructions is be a branch. Other discrepancies might arise due to the fact that MB-Lite introduces a stall when a load-store hazard occurs.

If the number of execution cycles are combined with the number of instructions of Figure 2.6 the CPI of MB-Lite can be determined as was done for the other processors in Section 2.3. The results of this computation are shown in Figure 4.5. This figure clearly shows the lower (i.e. better) CPI of the MB-Lite compared with

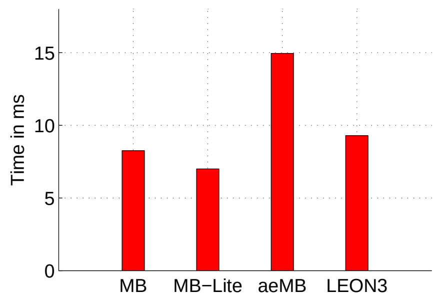


(a) Dhrystone CPI

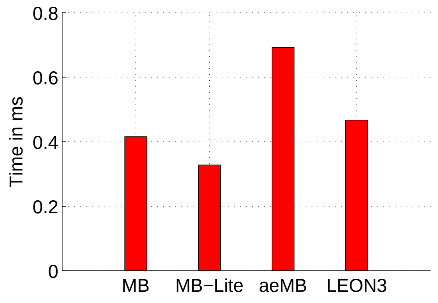


(b) Fibonacci CPI

Figure 4.5: Benchmark results in CPI



(a) Dhrystone execution time



(b) Fibonacci execution time

Figure 4.6: Benchmark results in execution time

aeMB and MicroBlaze.

If the information about the processor performance in terms of clock frequency is combined with the CPI and the program size the execution time of the applications can be determined which is shown in Figure 4.6. The higher clock frequency of both LEON3 as well as the MicroBlaze and the lower CPI have leveled up in terms of execution time. Therefore the MB-Lite performs around 20% worse than the LEON3 and MicroBlaze processors.

Although it was not our first concern to design a fast microprocessor, the MB-Lite comes quite far in matching the performance of the commercial MicroBlaze and LEON3 processors.

Conclusion 5

Within this thesis research has been done to quality and performance aspects of synthesizable CPU cores which can be easily integrated with research on Very Large Scale Integrated Systems On Chip. Commercial processors are mainly aimed at providing many additional functions and obtaining high speeds. Metrics like design reuse, portability and usability are of less importance. Within research projects the same metrics are used but the emphasis is completely different. A summary on the design goals as formulated in Section 1.2 is repeated here.

- Open source
- High quality and reliability
- Standard compliant
- High configurability
- High usability
- High simulation and synthesization speeds
- High portability
- Small size
- High performance
- Availability of components

In order to fill this extensive list of requirements as best as possible an approach was defined using a collection of many well-known strategies which was eventually used to design a processor which filled all needs.

5.1 Proposed strategy

Many requirements can be partially filled by using a two-process design methodology together with a well-defined component framework. According to the two-process design method a component is separated in a behavioral and sequential part. All outputs are connected to the sequential part, and are therefore output registered. Due to this structured design approach, synthesizers can do a very good job when optimizing designs by reducing logic levels, and thus obtain high performance. Since the behavior of the design is determined as a function of the inputs and the registers (c.f. state machines) the functionality of the algorithm can be easily evaluated resulting in more robust implementation.

In traditional ad-hoc design methods many small processes are defined. This greatly reduces readability and complicates maintenance, which in turn decreases reliability since it is hard to verify the functionality of the complete circuit. Types and records are used excessively to reduce code size and improve readability even more. Since only two processes are used for each component, hardware can be simulated and synthesized far more efficiently hence increasing simulation speeds.

Besides an improved hardware structure other means can be used to fill all requirements. Configurability and portability can be improved by making use of a framework in which a clear separation exist between general components and design specific components. A hierarchy of related libraries and packages can also be effectively used to promote design reuse and improve portability.

A wishbone bus is a highly desirable addition to improve usability and availability of components. Since this is a simple, well-defined open source bus protocol, many standard components have been designed using this specification. Additionally, a modular bus topology can be applied to promote the open source character of the system and makes it possible that other buses with different features can be connected. Finally, the design can be made scalable by introducing a bus which can be easily extended and configured.

5.2 Proposed implementation

It was determined that commercially licensed processors were unable to solve the most elementary requirements, amongst others due to the fact that they are distributed as firm-core implementations and thus can not be modified. Therefore the focus of the preliminary research was put on open source processors. It turned out that also none of the open source designs could fill all needs due to different reasons. All of the evaluated designs—except LEON3—were modeled using the old fashioned ad-hoc method and did not take advantage of the increase in abstraction offered by modern design tools. Therefore the requirements concerning portability and reliability could not be proven. Furthermore, the performance of all open source designs left much to be desired.

In order to overcome the problems which were encountered in all open source designs, a light-weight implementation of the MicroBlaze architecture was developed and named MB-LITE. Based on the strategy which was outlined previously, a processor with five pipeline stages was developed. Besides, the processor was modeled according to a well-known pipeline implementation to improve reliability. For those who have some basic knowledge about processors this design will look familiar, hence using this processor will become even more easy. A highly configurable and modular multiplexed bus was developed as well as a wishbone bus adapter.

The processor was thoroughly tested using many different configurations and software algorithms. A test bench was developed with complete coverage which indicates that every state of the processor is accessed at least once. The functionality of many different libraries was verified. Subsequently this test bench was also used to verify the functionality of the bus components. Finally a post place and route (PAR) model was synthesized and verified. Due to the huge program size, the memory requirements of the original test bench were too high for PAR simulation, so a smaller test bench was used to verify the functionality of this model including wire and gate delays.

5.3 Recommendations and future research

Future research on MB-LITE will focus on the implementation in a reconfigurable fabric as well as a 90 nm process technology. Therefore MB-LITE is designed to be easily portable, but no efforts have been spend yet to prove the applied concepts. Nevertheless, because this requirement has been included in the development during an early design stage some precautions were taken. During preliminary research it was found that memory is generally the most challenging component to implement in an IC process. Therefore all memory components have been implemented using standard synchronous components and are collected in a separate library with standard components.

It is recommended that different memory models as well as other components are added as separate VHDL architectures, while configurations are used to select the appropriate model for the targeted design. Traditional methods use generics in order to instantiate platform specific components. This method has an important drawback since the list of generics during component instantiation might increase dramatically in size, and the organization becomes unclear. This would tremendously reduce the readability of the design.

With respect to the use of configurations it might be necessary to replace components which are currently inferred from within the code with a separate entity (e.g. adders, shifters and multipliers). In this way the same method of configurations can be applied to select different implementations of equal components while maintaining the structure of the design.

It is furthermore highly recommended that all modifications will be gathered and published using a revision control system like SVN. In this way everybody can take advantage of user contributions which will greatly improve design reuse.

Nomenclature

ABI Application Binary Interface

AHB Advanced High-performance Bus

ALU Arithmetic Logic Unit

ASIC Application Specific Integrated Circuit

ASIP Application Specific Instruction Set Processor

BFM Bus Functional Model

BRAM Block Random Access Memory

CPI Cycles Per Instruction

CPU Central Processing Unit

DDR Double Data Rate

DSP Digital Signal Processor

EDK Embedded Development Kit

EX Execute

FPGA Field Programmable Gate Array

FPU Floating Point Unit

FSL Fast Simplex Link

FSM Finite State Machine

GCC GNU Compiler Collection

GDB GNU Debugger

GPP General Purpose Processor

HDL Hardware Description Language

IC Integrated Circuit

ID Instruction Decode

IF Instruction Fetch

IO Input/Output

IP Intellectual Property

ISA Instruction Set Architecture

ISS Instruction Set Simulator

JTAG Joint Test Action Group

LE Logic Element

LMB Local Memory Bus

LSB Least Significant Byte

LUT Lookup Table

MEM Memory

MIPS Million Instructions Per Second

MMU Memory Management Unit

MSB Most Significant Byte

NOC Network On Chip

NRE Nonrecurring Engineering

MPSOC Multi Processor System On Chip

OF Operand Fetch

OPB On-Chip Peripheral Bus

ORBIS OpenRisc Basic Instruction Set

ORFPX OpenRisc Floating Point eXtension

ORVDX OpenRisc Vector/DSP eXtension

PC Program Counter

PIC Programmable Interrupt Controller

PLD Programmable Logic Device

PROM Programmable Read Only Memory

RAM Random Access Memory

PLB Processor Local Bus

PLD Programmable Logic Device

ROM Read Only Memory

RISC Reduced Instruction Set Computer

RS232 Recommended Standard 232

SDRAM Synchronous Dynamic Random Access Memory

SOC System On Chip

SPI Serial Peripheral Interface

SPP Single Purpose Processor

SRAM Static Random Access Memory

TTM Time To Market

TVM Transaction Verification Model

UART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

VHDL Very High Speed Integrated Circuit Hardware Description Language

WB WriteBack

WSN Wireless Signal Network

XCL Xilinx Cache Link

XPS Xilinx Platform Studio

Bibliography

[1] W. Zhang, G. Du, Y. Xu, M. Gao, L. Geng, B. Zhang, Z. Jiang, N. Hou, and Y. Tang. Design of a hierarchy-bus based mpsoc on fpga. *Solid-State and Integrated Circuit Technology, 2006. ICSICT '06. 8th International Conference on*, pages 1966–1968, 2006.

[2] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, 2005.

[3] R. Holsmark, A. Johansson, and S. Kumar. On connecting cores to packet switched on-chip networks: A case study with microblaze processor cores. 7th IEEE Workshop DDECS 04, april 2004.

[4] T. Kranenburg and T.G.R.M. van Leuken. MB-LITE: A robust, light-weight soft-core implementation of the microblaze architecture. *Proceedings of the Design, Automation and Test in Europe*, 2010 (submitted).

[5] Alessandro Balboni and Loris Valenti. Asic design and fpga design: A unified design methodology applied to different technologies. In *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 356–360, 1996.

[6] D. Sheldon, R. Kumar, F. Vahid, D. Tullsen, and R. Lysecky. Conjoining soft-core fpga processors. *Computer-Aided Design, International Conference on*, pages 694–701, 2006.

[7] D. Mattsson and M. Christensson. Evaluation of synthesizable CPU cores. Master's thesis, Chalmers University of Technology, 2004.

[8] F. Plavec, B. Fort, Z. G. Vranesic, and Stephen D. Brown. Experiences with soft-core processor design. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 167.2. IEEE Computer Society, 2005.

[9] F. Plavec. Soft-core processor design. Master's thesis, University of Toronto, 2004.

[10] S. Craven, C. Patterson, and P. Athanas. Configurable soft processor arrays using the OpenFire processor. *Proceedings of 2005 MAPLD International Conference*, pages 250–256, 2005.

[11] F. Vahid and T. Givargis. *Embedded System Design.* Morgan Kaufmann Publishers, 1st edition, 2001.

[12] H. Bhatnagar. *Advanced ASIC Chip Synthesis*. Kluwer Academic Publishers, 2nd edition, 2002.

[13] J. Gaisler. Fault-tolerant microprocessors for space applications. http://www.gaisler.com/doc/vhdl2proc.pdf.

[14] OpenCores. http://www.opencores.org.

[15] Aeste Works. http://www.aeste.net.

[16] Aeroflex Gaisler. http://www.gaisler.com.

[17] OpenSparc. http://www.opensparc.net.

[18] Lattice semiconductor corporation. http://www.latticesemi.com.

[19] J. Gaisler, S. Habinc, and E. Catovic. *GRLIB IP Core User's Manual*, 2008.

[20] J. Gaisler, S. Habinc, and E. Catovic. *GRLIB IP Library User's Manual*, 2008.

[21] ARM. *AMBA^{TM} Specification (Rev 2.0)*, 1999.

[22] J.L. Hennessy and D.A. Patterson. *Computer Organization and Design: The Hardware/Software Approach*. Morgan Kaufmann Publishers, 3rd edition, 2005.

[23] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[24] IBM. *Processor Local Bus*, May 2007.

[25] IBM. *On-Chip Peripheral Bus*, April 2001.

[26] Xilinx. *MicroBlaze Processor Reference Guide*, January 2008.

[27] OpenCores. Wishbone system-on-chip (soc) interconnection architecture for portable ip cores. http://www.opencores.org/downloads/wbspec_b3.pdf.

[28] Y. Lu and W. Li. A semi-formal verification methodology. *Proceedings of the 4th International Conference on ASIC*, pages 33–37, 2001.

A Processor features

This chapter lists a subset of features of several processors. The chosen features are selected at importance for this project. The size and maximum clock frequency are based on the synthesis results with Xilinx ISE 10.1 on a Virtex 5 (XC5VLX110-3FF1760) device. All processors are configured with as few as possible additions (i.e. no cache, FPU, multipliers, dividers or other optional peripherals).

Table A.1: Processor features

	AeMB	LEON3	OpenFire 0.3b	OpenRisc1200	Plasma
Architecture	Harvard	Harvard	Harvard	Harvard	Neumann
Number of registers	32	24 (2-32 windows)	32	32	32
Data bus size (bit)	32	32	32	32	32
Instruction size (bit)	32	32	32	32	32
Pipeline depth	5	7	3	5	2, 3 or 4[a]
License	LGPL	GPL	MIT License	LGPL	Free to use
HDL	Verilog	VHDL	Verilog	Verilog	VHDL
ISA	MicroBlaze v6.2	Sparc V8	Microblaze 7.10	ORBIS32	MIPS
Compliant	Partially [b]	Fully	Partially [c]	Fully	Partially [d]
Debug interface	no	optional	no	yes	no
Number of Slice Flip Flops	711	1133	207	1577	1297
Number of 4-input LUT	926	3448	752	3802	2457
Max clock frequency	279 MHz [e]	183 MHz	198 MHz	185 MHz	73 MHz
System bus interface	Wishbone	AMBA 2.0 AHB	FSL	Wishbone	UART
Additional interfaces	FSL	UART, RS232, JTAG	no	JTAG	-
Memory controllers	RAM, SRAM	SRAM, SDRAM	no	no	DDR SDRAM
FPU	no	optional	no	no	no
Barrel shifter	optional	optional	no	no	no
Integer multiplier	optional	optional	yes	yes	yes
Integer divider	optional	optional	no	no	yes
Timer	no	optional	no	yes	no
Interrupt	yes	optional	no	yes	yes
Interrupt controller	no	optional	no	yes	no
MMU	no	optional	no	yes	no
Cache	yes, Instruction only[f]	yes	no	yes	yes

[a]Depends on the configuration and the instruction type

[b]The aeMB microprocessor is instruction compatible with MicroBlaze except for the instructions WIC,WDC,IDIV,IDIVU

[c]The OpenFire microprocessor is instruction compatible with MicroBlaze except for the WIC, WDC, IDIV*, BS, BSI, floating point and pattern instructions

[d]The unaligned load and store operations are patented and therefore can not be implemented.

[e]AeMB is a barrel processor. Hence, it alternately executes two threads. Single thread performance is therefore much lower than might be expected from this figure.

[f]Can not be disabled by configuration

B Simulation reports

B.1 Reference output of the test bench

```
Welcome to the MB—Lite Testbench

1. Testing Interrupt...
Handling interrupt routine
OK!

2. Testing Integer Arithmetic
OK!

3. Testing memory allocation
OK!

4. Testing Floating Point Arithmetic
OK!

5. Testing uncommon instructions
OK!

6. Executing dhrystone benchmark

Dhrystone Benchmark, Version 2.1 (Language: C)

Program compiled without 'register' attribute

Execution starts, 1 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    11
```

```
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          39088
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          39088
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

OK!

The testbench is now finished.
** Failure: FINISHED
   Time: 1046120 ns  Iteration: 0  Process: \
        /testbench/timeout File: ../../designs/core/testbench.vhd
```

B.2 Coverage report of the test bench

```
Coverage Report Summary Data by instance

Instance: /testbench/core0/fetch0
Design Unit: mblite.fetch(arch)
    Enabled Coverage        Active      Hits % Covered
    ----------------        ------      ---- ---------
    Stmts                       11        11     100.0
    Branches                     7         6      85.7
    Conditions                   0         0     100.0
    Fec Conditions               0         0     100.0
    Toggle Nodes                 3         1      33.3

Instance: /testbench/core0/decode0
Design Unit: mblite.decode(arch)
    Enabled Coverage        Active      Hits % Covered
    ----------------        ------      ---- ---------
    Stmts                      160       160     100.0
    Branches                    77        76      98.7
    Conditions                  44        41      93.1
    Fec Conditions              64        56      87.5
    Toggle Nodes                 3         1      33.3

Instance: /testbench/core0/execute0
Design Unit: mblite.execute(arch)
    Enabled Coverage        Active      Hits % Covered
    ----------------        ------      ---- ---------
    Stmts                       72        70      97.2
    Branches                    57        55      96.4
    Conditions                  14        11      78.5
    Fec Conditions              18        15      83.3
    Toggle Nodes                98        97      98.9

Instance: /testbench/core0/mem0
Design Unit: mblite.mem(arch)
    Enabled Coverage        Active      Hits % Covered
    ----------------        ------      ---- ---------
    Stmts                       21        21     100.0
    Branches                     7         6      85.7
    Conditions                   4         4     100.0
    Fec Conditions               6         6     100.0
    Toggle Nodes                34        33      97.0
```

Manuals and reference material C

C.1 Hardware Development Manual

C.1.1 Quick start guide

To simulate a design using ModelSim...

Go to one of the design directories in (MBLITE/DESIGNS/CORE*), and type:

```
%> make all
%> vsim &
```

Subsequently load the test bench design. It is recommended to disable optimizations and set the simulator resolution to PS to avoid warnings being generated. From ModelSim's command line type:

```
%> do start.do
```

to initialize the simulation, or

```
%> do run.do
```

to initialize and run the simulation.

The design CORE_SYN is a little more special since it contains templates to be used for post place and route simulations. These can be generated using a synthesis tool like ISE. Use one of the following actions:

```
%> make syn
%> make par
```

to compile the post-synthesis and post-place and route model, respectively. The memories are initialized already so do-scripts are not available.

To create a new design...

It is recommended to duplicate one of the design directories. If the directory is under version control, e.g. an (SVN) repository, type:

```
%> svn cp mblite/designs/core mblite/designs/my_design
```

Subsequently modify the makefile by changing the variable DESIGN_NAME in the name of the current directory.

To clean up compilation files...

Obviously, you can clean up your current working directory by typing

```
%> make clean
```

Table C.1: Description of the implementation parameters

Parameter	Description
CFG_INTERRUPT	Enable (1) or disable (0) the interrupt.
CFG_USE_HW_MUL	Enable (1) or disable (0) the hardware multiplier.
CFG_USE_BARREL	Enable (1) or disable (0) the barrel shifter.
CFG_DEBUG	Enable (1) or disable (0) debugging mode. In debugging mode more registers are cleared to enhance readability. Disabling debug mode gives a little more performance.
CFG_DMEM_SIZE	Data memory size in number of address bits. Maximum addressable memory is $2^3 2$ bytes (4Gb). Default: 64 Kb
CFG_IMEM_SIZE	Instruction memory size in number of address bits. Maximum addressable memory is $2^3 2$ bytes (4Gb). Default: 64 Kb
CFG_BYTE_ORDER	Little endian byte order (0) or Big endian byte order (1, default). Little endian byte ordering is not supported by the MicroBlaze tool chain.
CFG_REG_FORCE_ZERO	Explicitly clear register R0 when it is read
CFG_REG_FWD_WB	Forward the writeback register value. Forwarding this value reduces the requirements on the memory used to implement the registers. Most FPGA memories require this option to be enabled
CFG_MEM_FWD_WB	Forward the memory result internally instead of introducing additional stalls.

To synthesize a design using ISE...

Start a new project, and compile all design files in MBLITE/DESIGNS/CORE* except config_Pkg.vhd to library work. All other files, including config_Pkg.vhd should be compiled to a library called MBLITE. It is recommended to assign the locations by reference instead of copying the files to the current working directory.

C.1.2 Implementation parameters

Several parameters are added to the design to control the implemented features and components. Table C.1 lists the available parameters and contains a short description where the parameter is used for. The parameter names are chosen in accordance with the MicroBlaze specification.

The parameters are defined in the config_Pkg.vhd file which is in the design folder. The parameters CFG_INTERRUPT, CFG_USE_HW_MUL, CFG_USE_BARREL and CFG_DEBUG are defined generic with the default values coming from the configuration file. The advantage of using generics is that different parameters can be specified for different instantiations of that component. Within the same design we can create a processor with a multiplier and barrel shifter while another component on the same fabric does not have these features.

C.2 Dependencies between VHDL packages

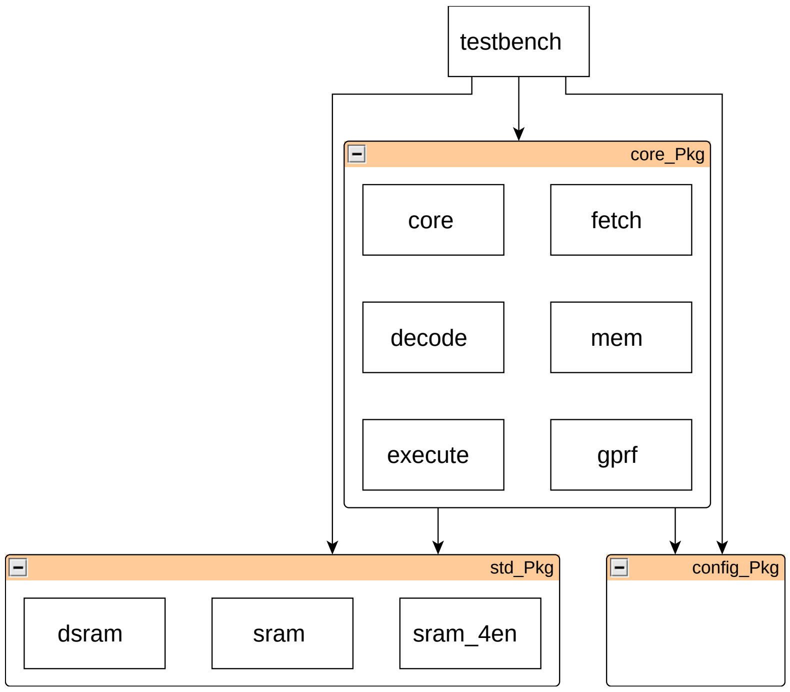


Figure C.1: Dependencies between the MB-Lite VHDL packages

C.3 Contents of the MB-Lite package

C.3.1 VHDL components

The directory MBLITE/HW contains all VHDL descriptions of the packages and components. The contents of these files are described in Table C.2. The MB-Lite core uses several standard generic functions which have been defined in the package std. A description is given in Table C.3.

Table C.2: VHDL components in the package "core"

core.vhd	This top level component consists of instantiation of the pipeline modules and their connections.
core_wb.vhd	This top level component instantiates a core and connects the wishbone wrapper to the data bus.
core_Pkg.vhd	This package contains all MB-Lite component descriptions, the data types as used in the processor components and several functions used in the modules.
core_address_decoder.vhd	This generic address decoder splits the MB-Lite data bus in a configurable number of sub-buses. It accepts a linear memory map to initialize the controller.
core_wb_adapter.vhd	This component contains the description of the MB-Lite bus to wishbone adapter.
fetch.vhd	First pipeline component which fetches the instructions.
decode.vhd	Second pipeline component for decoding the instruction, reading the registers and writing back new register contents.
gprf.vhd	Used within decode to abstract the memory organization used to implement a memory with one write input and three read outputs.
execute.vhd	Contains all components necessary for computing an instruction result.
mem.vhd	The memory stage aligns the memory read and write data and controls the data bus of the MB-Lite processor.

Table C.3: VHDL components in the package "std"

std_Pkg.vhd	Description of components in the and several generic functions.
dsram.vhd	Dual port synchronous RAM component.
sram.vhd	Synchronous RAM component.
sram_4en.vhd	Synchronous RAM component with four write enable ports.

C.3.2 Example software and tools

The directory MBLITE/SW contain the benchmark and verification software. Table C.4 gives a short description of all software. All software packages contain almost identical makefiles to compile software for the MB-Lite processor.

Table C.4: Software used in MB-Lite simulation and verification

dhrystone	The Dhrystone benchmark Version 2.1 written in C. It has been sanitized to obtain an ANSI C compatible version.
fibonacci	Computes the fifteenth Fibonacci number using iterative function calls. Requires a considerable stack for proper execution.
hello_world	Prints hello_world to the standard output.
stdio	Test files for reading and writing memory. Might not be compatible with all MB-Lite example designs due to differences in the test bench.
testbench	Test bench used in verification of the MB-Lite design.
util/bin2mem	This program is used to convert binary compilation images to mem files suitable for ModelSim's MEM command.

C.3.3 Example designs

The directory MBLITE/DESIGNS contains pre-assembled designs and include configuration files, test benches, makefiles and ModelSim scripts for a specific design. Four example designs are included as described in Table C.5. In Table C.6 all files specific for these designs are described. All of the designs have a standard IO connected at address 0XFFFFFFC0 in order to write to the console screen.

Table C.5: Design examples

core	This design instantiates a bare core and connects standard memory components to the instruction and data buses.
core_wb	This design instantiates the wishbone wrapped top level core module. The test bench connects to a wishbone compatible memory device and to a wishbone compatible standard output.
core_decoder	In this design an address decoder is inserted between the MB-Lite core and the attached peripherals. The decoder takes care of routing the control and data signals to and from the appropriate devices.
core_decoder_wb	Same design as core_decoder, except that a wishbone compatible standard output interface is now connected using the wishbone adapter and decoder to the MB-Lite core. The memory is connected directly to the adapter.
core_syn	Same design as core, but both the instruction as well as the data memories are gathered in the top level component mblite_soc. In stead of standard memories, initialized memories are used since currently there is no boot loader or anything available yet.

Table C.6: Design example files

Makefile	This file can be used with ModelSim to compile or clean up the design. Dependencies are solved automatically and all components are compiled to the appropriate libraries.
config_Pkg.vhd	This file contains constants for initialization. Note that some of these constants can be overwritten during instantiation by using generics.
rom.mem	This ascii file contains the program instructions and can be generated using the MicroBlaze tool chain.
run.do	Can be used from within ModelSim to run a design for unspecified time.
start.do	Can be used from within ModelSim to load a design without running. The registers and memories will be loaded with the rom.mem file.
test bench.vhd	Instantiates and attaches all components within the design like the MB-Lite core, memories and standard IO interfaces.
mblite_stdio.vhd	Standard IO to be connected to the MB-Lite data bus.
wb_stdio.vhd	Wishbone compatible standard IO core to be connected to the wishbone adapter.