Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2007

MSc THESIS

Memory organization of the Molen prototype

Sebastiaan Dirk Breijer

CE-MS-2007-06

Abstract

Field Programmable Gate Arrays (FPGAs) are commonly found in prototyping of embedded devices as well as in commercial embedded products. Typical fields of applications for these programmable devices also include the embedded reconfigurable computing domain, where FPGAs provide a platform which is able to be re-programmed (configured) according to the users needs. The support of large applications or even an Operating System (OS), can expand the flexibility introduced by the embedded reconfigurable processing paradigm even further.
The Molen machine organization is one of these embedded reconfigurable architectures, constituted by a core processor and a reconfigurable processor. The Molen prototypes memory organization is based upon on-chip memories, which have a limited capacity. Therefore, in order to support large applications or even an OS, the Molen's memory organization can be improved by extending it or replacing it by using off-chip memories.
In this Thesis, we examine the different options of adapting the memory organization of the Molen prototype, in order to support the execution of applications with a large memory footprint. To realize the use of off-chip memories, we introduce an external memory controller which shares one physical memory for both data and instructions, based on the On-Chip Memory (OCM) buses. Moreover, this memory controller introduces a performance improvement from 6% to 13% over the traditional Processor Local Bus (PLB) based design. Furthermore, in this Thesis, we investigate the possibilities of executing the Linux OS on our newly introduced design. This experiment revealed a limitation of the OCM buses with respect to virtual-to-physical address translation which was undocumented until now. Finally, we recommend and describe a detailed design which is not affected by these limitations, and therefore, enables the execution of the Linux OS on the Molen prototype.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Memory organization of the Molen prototype

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Sebastiaan Dirk Breijer
born in Roosendaal en Nispen, The Netherlands

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

Memory organization of the Molen prototype

by Sebastiaan Dirk Breijer

Abstract

Field Programmable Gate Arrays (FPGA) are commonly found in prototyping of embedded devices as well as in commercial embedded products. Typical fields of applications for these programmable devices also include the embedded reconfigurable computing domain, where FPGAs provide a platform which is able to be re-programmed (configured) according to the users needs. The support of large applications or even an Operating System (OS), can expand the flexibility introduced by the embedded reconfigurable processing paradigm even further.

The Molen machine organization is one of these embedded reconfigurable architectures, constituted by a core processor and a reconfigurable processor. The Molen prototypes memory organization is based upon on-chip memories, which have a limited capacity. Therefore, in order to support large applications or even an OS, the Molen's memory organization can be improved by extending it or replacing it by using off-chip memories.

In this Thesis, we examine the different options of adapting the memory organization of the Molen prototype, in order to support the execution of applications with a large memory footprint. To realize the use of off-chip memories, we introduce an external memory controller which shares one physical memory for both data and instructions, based on the On-Chip Memory (OCM) buses. Moreover, this memory controller introduces a performance improvement from 6% to 13% over the traditional Processor Local Bus (PLB) based design. Furthermore, in this Thesis, we investigate the possibilities of executing the Linux OS on our newly introduced design. This experiment revealed a limitation of the OCM buses with respect to virtual-to-physical address translation which was undocumented until now. Finally, we recommend and describe a detailed design which is not affected by these limitations, and therefore, enables the execution of the Linux OS on the Molen prototype.

Laboratory : Computer Engineering
Codenumber : CE-MS-2007-06

Committee Members :

Advisor: Dr.ir. J.S.S.M Wong, CE, TU Delft

Chairperson: Dr.ir. K.L.M. Bertels, CE, TU Delft

Member: Prof.dr.ir. A.J.C. van Gemund, ST, TU Delft

Member: F. Duarte MSc., CE, TU Delft

To Saskia, for her love and support.

Contents

List of Figures

List of Tables

Acknowledgements

For the past year I have worked on this MSc. thesis project with great pleasure at the Computer Engineering Laboratory in Delft. I would like to thank Stephan Wong, for guiding me and helping me take important decisions. Also, I would like to thank Filipa Duarte, she prepared this project and guided me throughout the year. Moreover, she invested many hours in correcting and commenting on the different versions of this thesis and my paper, for which I am very grateful.

The support of Bert Meijs, Lidwina Tromp and Stefan Raaijmakers (Bugs) should not go unnoticed. Bert and Stefan helped me configure the tools I used to realize this project, they also provided me with support every time something was malfunctioning. I would also like to thank Marc Grootjen for being a nice roommate, a good friend and providing useful comments on this thesis. Moreover, he inspired me to start my study at TU Delft for which I am grateful.

And last, but not least, I would like to pay my gratitude to my parents and to the rest of my family, for supporting me throughout my period of study, in all possible ways.

Sebastiaan Dirk Breijer
Delft, The Netherlands
August 16, 2007

1 Introduction

1.1 General Introduction

When the term computer systems is mentioned, the majority of people will think about the computer they use at home, or at the office. A smaller group may think about the computers used merely to provide data or services to a larger amount of people, so-called servers. The computers used as server, personal computer (at home) or desktop computer (at the office), are computers which can be used for almost any job. However, the underlaying hardware is alike, a system may be optimized for a specific purpose. For example, to play games, it may have some more internal memory or a different graphics card. These computers are often referred to as general purpose computers.

Apart from the well-known general purpose computers, another branch of computers are embedded computers or embedded systems. In comparison to general purpose computers, the hardware of an embedded system varies from system to system. These embedded computers, can be anywhere: in a car, a coffee machine, a washing machine, a mobile phone, etc. Most embedded systems are hidden in a way you might not even identify them as computer systems. Because of the physical location where an embedded system is utilized, a developer may have to deal with strict constraints, like: power consumption, heat production, physical dimensions, performance, etc. For these reasons, the development of embedded systems differs from the development of general purpose computers.

Besides the difference in hardware, embedded systems, also the software utilized in both types of systems is different. For general purpose systems, the use of an Operating System (OS) is common, for embedded systems, the use of an OS is rare. Although an OS enables flexible development of software for different systems, embedded system software is commonly developed specifically for each system. The main reason for not applying an OS on embedded systems is the unsolicited overhead. However, with advances in technology, it becomes profitable to accept this overhead, and save on development time of specific software. Shifting towards embedded OSs can be found in our direct environment. For instance, the newer generations of mobile devices, like mobile phones, rely on the use of an embedded OS.

1.2 Research Scope

In embedded systems gaining performance or reducing power consumption, heat production, area, etc. are main topics of concern. In this thesis, we will focus on the design of embedded hardware which improves performance without sacrificing on any of the other metrics. To be more specific, we will focus on performance of the connection between main memory and the main processor, without sacrificing area. This research intents

to widen the range of possible applications, utilizing the prototype of an embedded test system based on the Molen machine organization introduced by the Computer Engineering laboratory of the Delft University of Technology. We will also focus on the options and limitations of using an embedded OS in combination with this prototype.

1.3 Problem Statement

Any computer system has at least a processor, some main memory and input/output devices. However, the exact configuration of one system may vary a lot compared to another, e.g. the amount of memory used, and the type of peripherals connected may differ. Independent of the hardware configuration, the behavior of a system is determined by the software executed on such a system. Moreover, the performance of the software executed on a system is dependent on the hardware configuration. If a program has a large memory footprint (i.e. requiring a lot of memory), but is executed on a system with only little physical memory, the performance will be bad, or the program will not run at all. For desktop and personal computers, performance related problems are becoming more rare, but for embedded system development, performance issues are still a daily hassle.

In embedded systems, three different types of processors are used. Most common are the General Purpose Processors (GPPs), these processors allow easy software development and are flexible with respect to future changes. Another concept used, is to design an Application Specific Integrated Circuit (ASIC). An ASIC can be referred to as a specific purpose processor. The functionality of an ASIC is limited to one, or several functions needed by the system the ASIC is designed for. The performance, and power consumption, of an ASIC is good compared to a general purpose processor, but there is no flexibility for future changes. The third option for embedded developers is to use reconfigurable hardware. A reconfigurable device, can be configured to operate as an ASIC would, but still provides the flexibility comparable to a GPP. Performance of modern reconfigurable devices is comparable to the performance of an ASIC, but power consumption is slightly worse.

The Molen machine organization introduces a combination of both the programmability of a GPP and the advantages of an ASIC. By connecting a reconfigurable coprocessor to a GPP, the Molen machine organization combines the programmability of a GPP with the performance of an ASIC. To prove the concept of the Molen machine organization, a prototype has been built [8]. This prototype allows small programs to be executed on the GPP, while specific functions are executed in the reconfigurable part of the architecture, namely in the Custom Computing Unit (CCU). However, the memory architecture of the current prototype imposes limitations on the applicability for a wide range of applications.

Is it possible to find a solution to widen the range of applications able to benefit from the Molen machine organization by adapting the hard- and software? Furthermore, can we find a way to provide support for running applications with a large memory footprint by connecting external Double Data Rate (DDR) memory to the Molen machine organization?

To obtain insight in the possible solutions, the hardware organization of the current

Molen prototype will be examined. Moreover, optional combinations of hardware and software changes are to be studied. A solution, which can be a combination of hard- and software, shall be adopted to provide support for application with a large memory footprint on the Molen prototype.

1.4 Thesis Overview

The remainder of this document is organized as follows. Chapter 2 provides the background and the related work. Two development boards with the Field Programmable Gate Arrays (FPGAs) used throughout this thesis are introduced, the PowerPC is introduced and the Molen machine organization is explained

Chapter 3 presents the options for adapting the Molen prototype to support applications with a large memory footprint and concludes with a motivation of the choice for one of the options.

Chapter 4 presents the development of a Data-Side On-Chip Memory (DSOCM) DDR controller, an Instruction-Side On-Chip Memory (ISOCM) DDR controller and a shared On-Chip Memory (OCM) DDR controller. The designs are specified and implementation details are discussed.

Chapter 5 discusses the verification and simulation results of the controllers presented in Chapter 4. Moreover, the performance results obtained using the shared OCM DDR controller are presented.

Chapter 6 presents the considerations and limitations of running Linux on a design based on the shared OCM DDR controller presented in Chapter 4. This attempt reveiled, thus far, undocumented limitations on the use of the PowerPCs memory management in combination with the OCM buses.

Chapter 7 reviews the options presented in Chapter 3 taking the undocumented limitations presented in Chapter 6 into account and concludes by recommending towards future development.

Chapter 8 provides a summary of our conclusions, an overview of the main contributions and presents directions for future research.

2 Background

In embedded system design, reconfigurable hardware is used to provide a way to implement and test a prototype. A commonly used reconfigurable device is a FPGA. The Xilinx corporation [9] is the developer of a wide range of FPGAs, the Virtex family being part of them.

In this chapter we will provide the necessary background on the tools and techniques used throughout this thesis. We introduce two development boards, each based on a different FPGA. Next, we provide details about the Virtex-II Pro and Virtex-4 FX FPGAs. Both these FPGAs have an embedded GPP, namely the International Business Machines Corporation (IBM) PowerPC. Details on the PowerPC are provided. General issues on the use of the Xilinx Intellectual Property Interface are introduced. The Molen machine organization is introduced and the organization of the Molen prototype is explained. Finally, we present related work on memory controllers, reconfigurable architectures and embedded Linux.

2.1 Xilinx Development Boards

Because of the reconfigurable nature of FPGAs, they are well suited and often used for prototyping. To provide the necessary tools for FPGA based designs, development boards are introduced. In this section, we will discuss two development boards, used to implement the designs presented in this thesis.

2.1.1 Xilinx University Program Embedded Development Board

The Xilinx University Program (XUP) embedded development board, is a development board based upon a the Virtex-II Pro FPGA (short: XUPV2P board). The XUPV2P board provides a the following main components:

- XC2VP30 Virtex-II Pro FPGA with two PowerPC405 cores embedded
- A DDR Synchronous Dynamic Random Access Memory (SDRAM) slot, which supports DDR modules up to 2 GB
- RS-232 serial port
- 10/100 Ethernet Physical layer protocol (PHY) device
- Universal Serial Bus (USB) configuration port
- System Advanced Configuration Environment (ACE) configuration chip

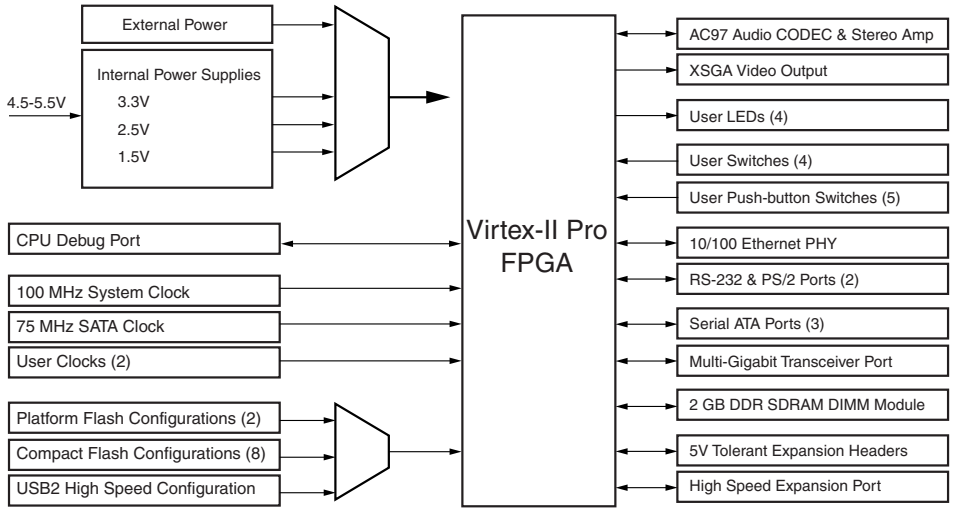


Figure 2.1: XUPV2P system block diagram [1].

Figure 2.1 depicts the block diagram (For the full feature list, see [1]). Each of the connections depicted in Figure 2.1 can be connected using the routing resources of the FPGA. The physical placement of the peripherals on the board introduce limitations on what Input/Output Blocks (Input/Output Block (IOB)) can be used. The routing resources of the FPGA can be used to overcome these limitations.

2.1.2 Xilinx ML403 Embedded Development Board

The ML403 development board, is a development board based upon a Virtex-4 FX FPGA. The ML403 board provides a the following main components:

- XC4VFX12 Virtex-4 FX FPGA with one PowerPC405 core embedded
- 64 MB on-board DDR SDRAM memory
- RS-232 serial port
- 10/100/1000 Ethernet PHY device
- Joint Test Action Group (JTAG) configuration port
- USB host and target ports
- 4 MB flash memory and a Complex Programmable Logic Device (CPLD) to use the flash memory to program the FPGA
- System ACE configuration chip

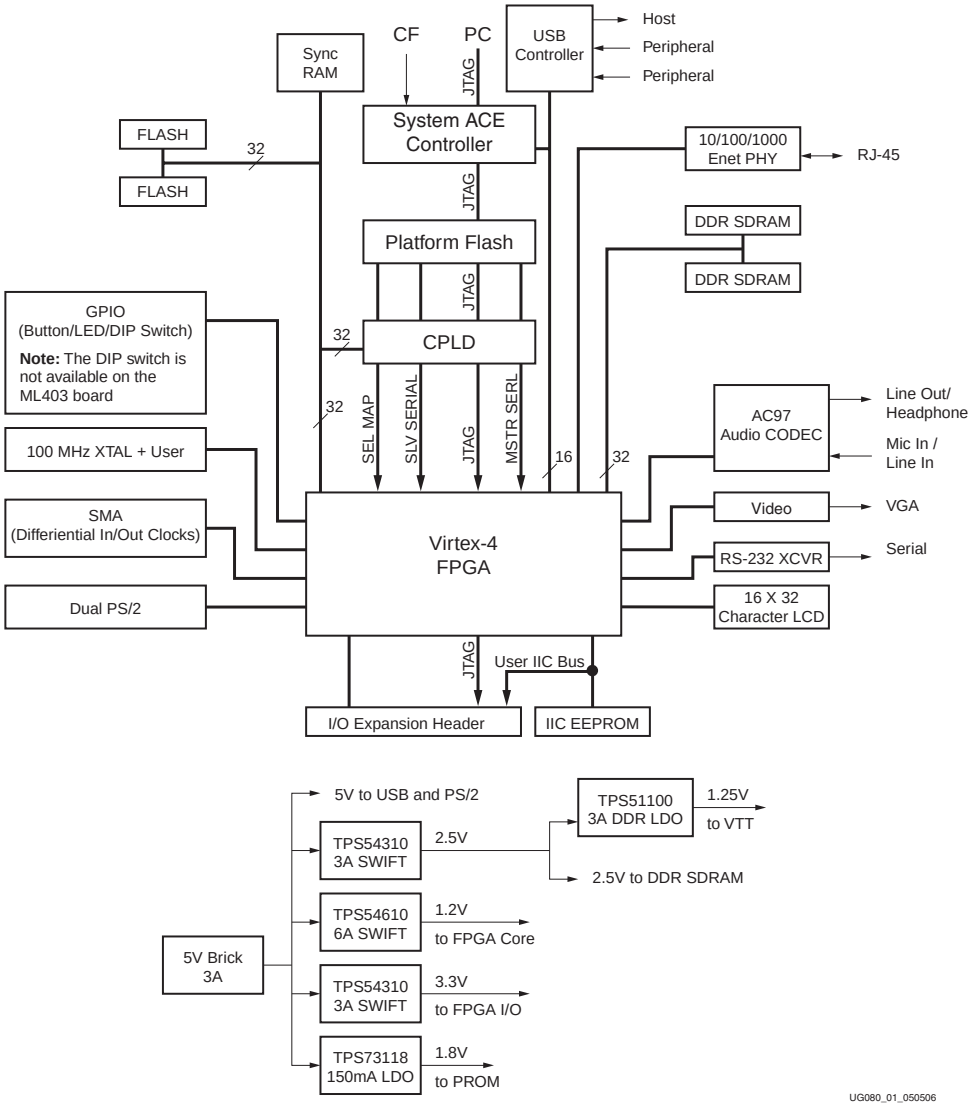


Figure 2.2: ML403 system block diagram [2].

The full feature list can be found in [2]. The ML403 development board also provides on-board flash memory and facilities to use the flash memory to store the FPGAs bitstream and automatically program the FPGA on power-on. Figure 2.2 depicts the block diagram of the ML403 development board and shows the optional connections which can be made using the routing resources of the FPGA. Similar to the XUPV2P board, each peripheral is connected to only one set of physical IOBs of the FPGA. This introduces constraints on the routing of Input/Output (IO) related signals. Moreover, some of the ML403's peripheral share IOBs and can therefore only one of them can be used at the same time.

2.2 Xilinx FPGAs

Both development boards have FPGAs as explained previously. In this section, we present the basic internal organization of a Xilinx FPGA.

An FPGA in general is a reconfigurable device. Like the name states, it is a gate array which is programmable by applying an electric field. Once a FPGA is programmed, the device behaves like "normal" hardware, and its performance is only slightly worse than the performance of an ASIC.

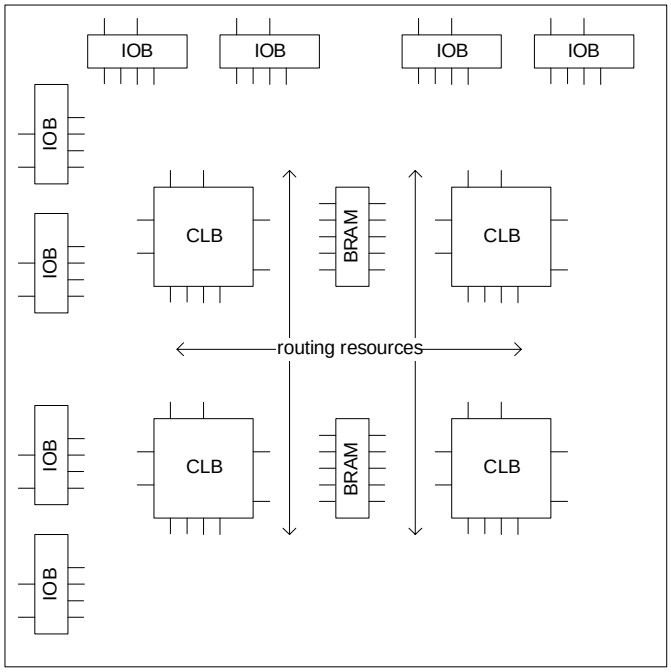


Figure 2.3: Abstract overview of the Xilinx FPGA internal components.

FPGAs of different manufacturers differ in general layout [10]. The Xilinx FPGAs mainly consists of IOB, Configurable Logic Block (CLB) and routing resources. Figure 2.3 depicts an abstract view of the Xilinx FPGA internal components. Each IOB can be configured to be an input, output or both (tri-state). A CLB consists out of four slices, each slice consists of two Random Access Memory (RAM) memories, which can be used a Look-Up Table (LUT), RAM memory or Read Only Memory (ROM) memory. Moreover, a slice also contains two sequential blocks which can be configured as Flip-Flop (FF) or as Latch. The Xilinx FPGAs also provide additional Block Random Access Memory (BRAM) blocks. Note that BRAMs are situated outside of a CLB and have a large capacity (18 KB) compared to the memory segments inside each slice (16 bits). Apart from these common features, the Virtex-II Pro and Virtex-4 FX have one or more PowerPC blocks. A detailed description of the features can be found in [11] for the Virtex-II Pro and in [12] for the Virtex-4.

Modeling the behavior of a FPGA is done by using a Hardware Description Language

(HDL). For the designs presented in this thesis, Very high speed integrated Hardware Description Language (VHDL) is used to create the structural or behavioral model of the hardware presented. The VHDL source code is synthesized and mapped to the Virtex devices using the Xilinx Synthesis Technology (XST) synthesis compiler and the Xilinx tool chain.

2.3 Embedded PowerPC 405 Architecture and Organization

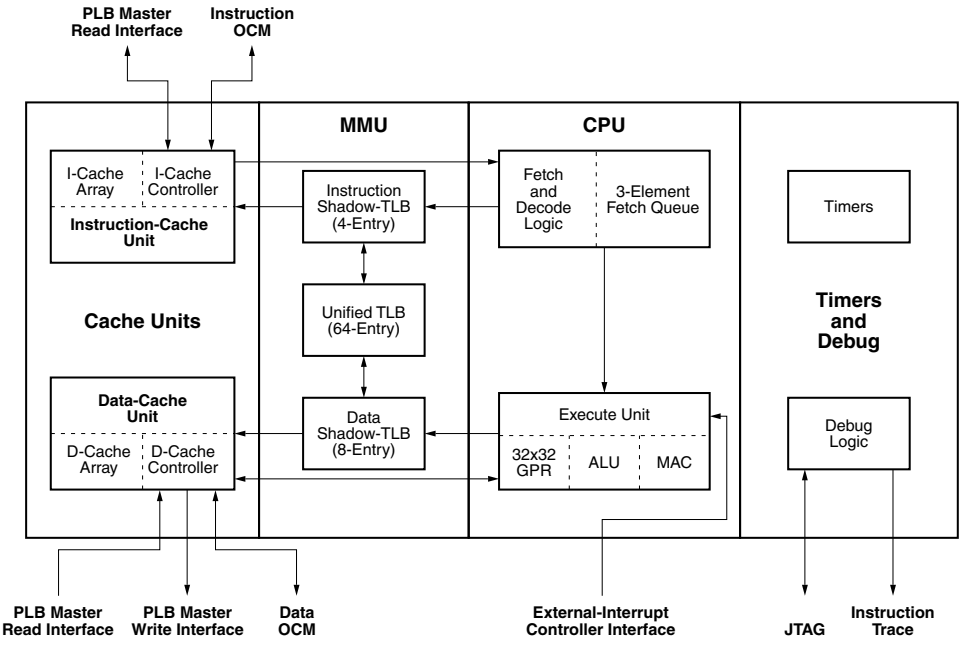


Figure 2.4: Internal PPC405 organization [3].

As the IBM PowerPC is widely used in our design, in this section, we present a brief overview of the PowerPCs general capabilities and internal organization.

The embedded IBM 405 Reduced Instruction Set Computer (RISC) is a general purpose computer compatible with the IBM PowerPC architecture and instruction set. The processor is generally referred to as PPC405. Figure 2.4 depicts the internal organization of the PPC405 processor. The PowerPC supports two operating modes, privileged mode and user mode. In user mode, not all instructions are available.

The PPC405 processor is a Harvard architecture. All data requests are handled by the Data Cache Unit (DCU), the DCU is connected to a 16KB two-way set-associative cache, a DPLB interface and a DSOCM interface. All instructions are fetched using the Instruction Cache Unit (ICU), which is connected to a 16KB two-way associative cache, an Instruction-Side Processor Local Bus (IPLB) interface and an ISOCM interface,

similarly to the DCU organization.

2.3.1 PowerPC bus structure

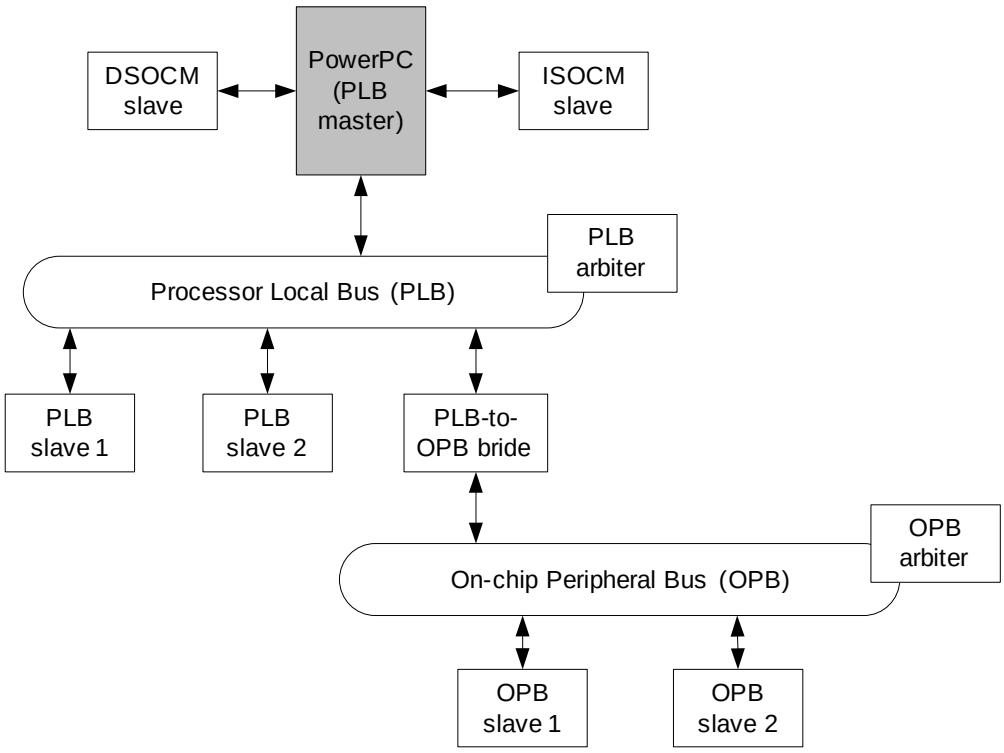


Figure 2.5: Xilinx implementation of the IBM CoreConnect bus system.

The PowerPC uses two different buses to connect memory and peripherals, namely the Processor Local Bus (Processor Local Bus (PLB)) and the On-Chip Memory (OCM) bus. The PowerPC also supports the IBM CoreConnect bus system [13]. The core connect system introduces two new main buses, the On-chip Peripheral Bus (OPB) and the Device Control Register (DCR) bus. The OPB is used to connect relatively slow peripherals and typically runs on a frequency lower than the PLB. The DCR bus is mainly used to transfer device configuration data and settings, the DCR is not used to transfer data or instructions. Figure 2.5 depicts the IBM CoreConnect bus system as used in the Xilinx implementation of the PowerPC core. The OCM buses are not part of the IBM CoreConnect system, but are depicted to give a complete overview.

The OCM bus and the PLB will be discussed in more detail.

2.3.1.1 On-Chip Memory Bus

The OCM bus is divided in an ISOCM and a DSOCM bus. Both ISOCM and DSOCM are designed to connect on-chip BRAM blocks to the PowerPC. Moreover, the buses support one master (the PowerPC) and one slave. On the Virtex-II Pro implementation

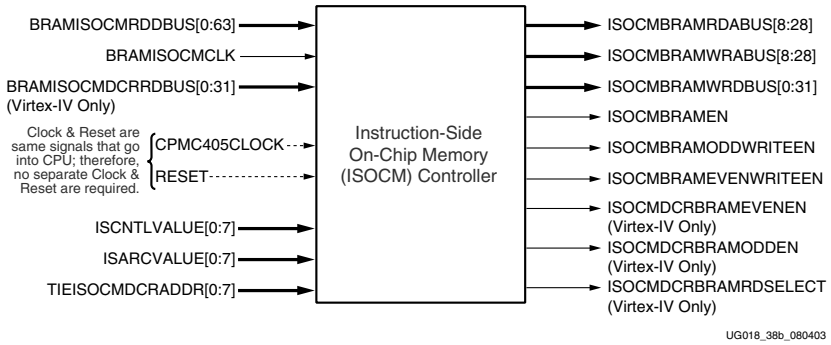


Figure 2.6: ISOCM bus controller block diagram [4] (Virtex-4 implementation).

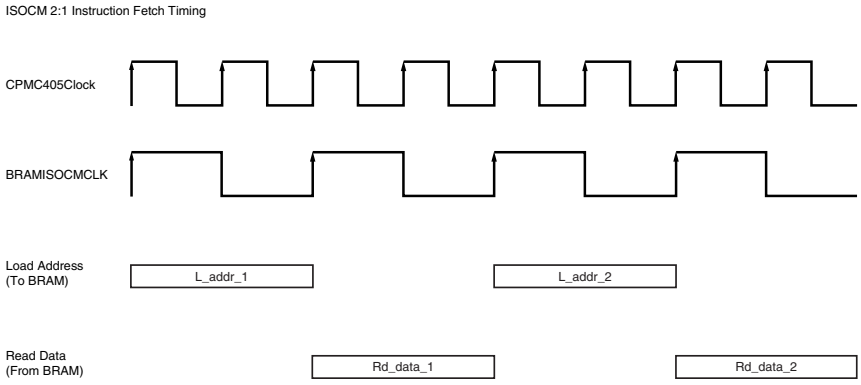


Figure 2.7: ISOCM instruction fetch timing [3].

of the PowerPC, both buses only support the interfacing with the BRAM blocks which have a latency of one clock cycle. On the Virtex-4 implementation of the PowerPC, the DSOCM is extended with a facility to support variable latency. The behavior of both ISOCM and DSOCM are deterministic, meaning that the bus latency is constant for every operation.

The ISOCM bus is conceptually an unidirectional bus, the bus supports double word (64-bit) read operations. For convenience, the ISOCM also supports single word write operations. Figure 2.6 depicts the block diagram of the ISOCM bus controller. The ISOCM bus consists of a 21-bit address bus (`isocmbramrdabus`), which allows 16 MB to be addressed. Internally, the PowerPC uses 32-bit addresses, to verify if memory request is meant for theOCMM bus, the upper 8-bit of the 32-bit address are compared to the `isarcvalue` which determines the offset of the ISOCM address space. The lower three address bits of the ISOCM bus address are implicitly zero. To select which word to write on a single word, the ISOCM introduces two signals, the `isocmbramoddwriteen` and the `isocmbramevenwriteen`. All ISOCM accesses are guarded by the `isocmbramen` signal.

The `iscntlvalue` is used to set the properties of the ISOCM bus, for instance, the clock ratio of the ISOCM clock compared to the PowerPC clock is set using this register. Figure 2.7 depicts the timing of a typical ISOCM instruction fetch (read operation).

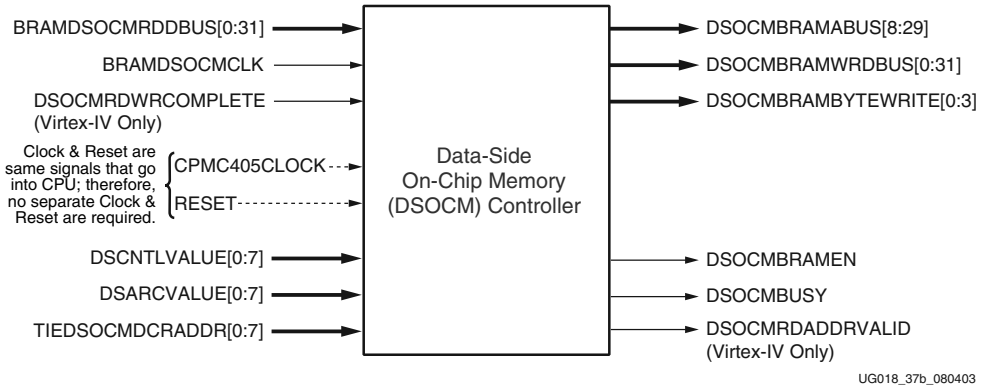


Figure 2.8: DSOCM bus controller block diagram [5] (Virtex-4 implementation).

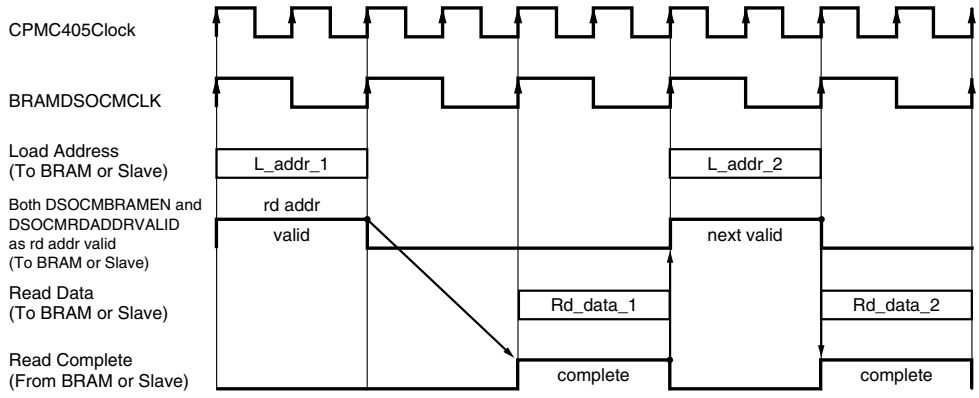


Figure 2.9: DSOCM data read timing [3].

The DSOCM bus is a bi-directional bus. The bus supports single word (32-bit) read and write operations. In the Virtex-4, a handshaking protocol is introduced. Figure 2.8 depicts the block diagram of the DSOCM bus controller. The `dsocmrdwrcomplete` signal should be asserted by the device connected to the DSOCM bus once its operation is completed. The DSOCM bus has a 22-bit wide address bus, allowing 16 MB to be addressed. The upper 8-bits of the internal PowerPC address are compared to the `dsarcvalue` and if matched, the request is sent to the DSOCM. The lower 2-bits of the DSOCM address are zero implicitly. To provide a facility to write separate bytes to the memory, the DSOCM bus provides a byte-write signal, namely the `dsocmbrambytewrite` signal. This 4-bit signal can be used to identify which bytes are to be written. All DSOCM operations are initiated by the PowerPC by asserting the `dsocmbramen` signal.

The `dscntlvalue` is used to set the properties of the DSOCM bus, for instance, the clock ratio of the DSOCM clock compared to the PowerPC clock is set using this register. Figure 2.9 depicts a typical read operation using the variable latency mode of the DSOCM bus.

2.3.1.2 Processor Local Bus

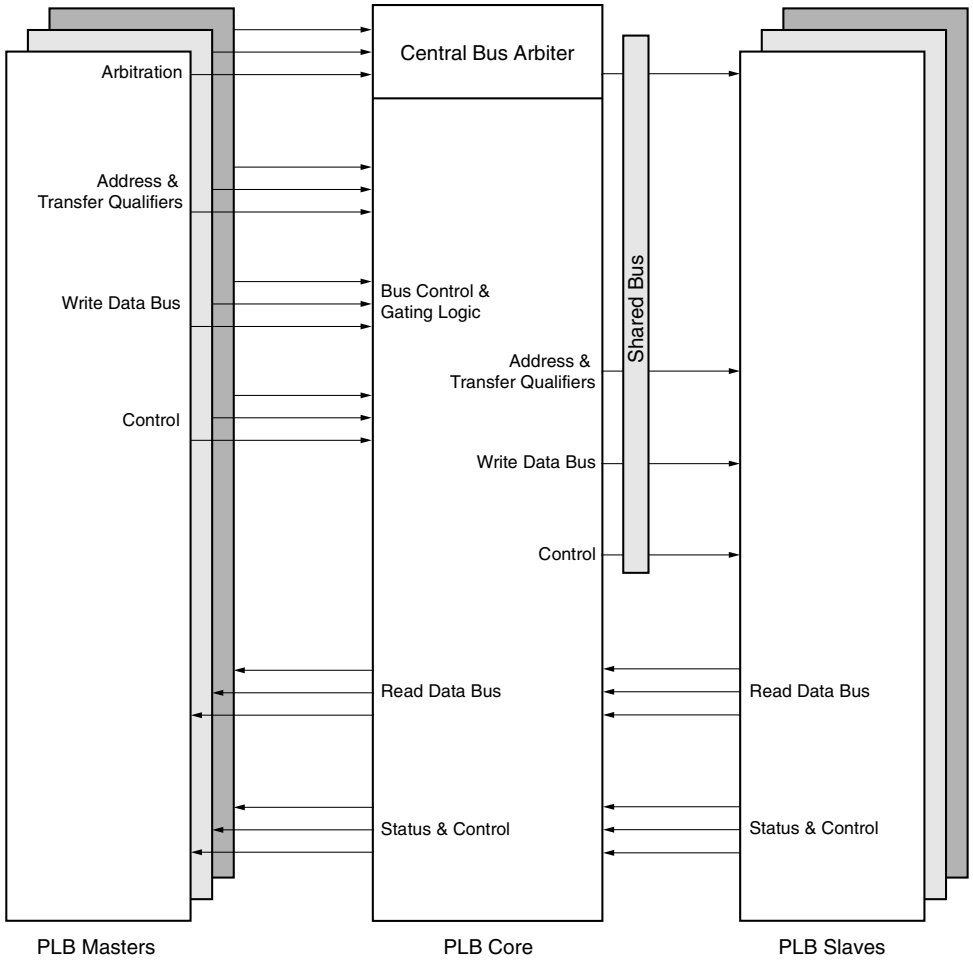


Figure 2.10: PLB arbiter interconnections [6].

The PLB is the main interface to connect peripherals to the PowerPC. The PLB supports multiple masters and multiple slaves. In a typical configuration, the ICU and DCU are connected to the PLB as two master interfaces and several peripherals can be connected using a PLB-slave connection. The PLB uses a double word (64-bit) data bus and a 32-bit address bus. This implies an address space of 4 GB. The PLB is a decoupled bus with bus arbitration to handle requests. Each master has its own set of wires to communicate with the PLB arbiter. All slaves share a set of wires. Figure 2.10 depicts

the master/slave connections from and to the PLB arbiter. The PLB also supports data transfers of multiple words [14]. The PLB is an indeterministic bus, meaning that the bus latency may vary, depending on if an operation is occupying the bus. The PLB also supports Direct Memory Access (DMA). A DMA able master can initiate a data transfer autonomously.

2.3.2 PowerPC exceptions and interrupts

The PowerPC embedded in the Virtex-II Pro and Virtex-4 FPGAs supports interrupts and exceptions. Exceptions are events that often occur if the PowerPCs hardware needs the attention of the software. Exceptions can have an internal or external source. As a result of an exception, the normal program flow is interrupted. This interrupt causes the program flow to be transferred to a pre-defined ISR. The address of the ISR is identified using the exception vector offset and the value stored in the Exception Vector-Prefix Register (EVPR). The PowerPC supports 16 different interrupt causes [7].

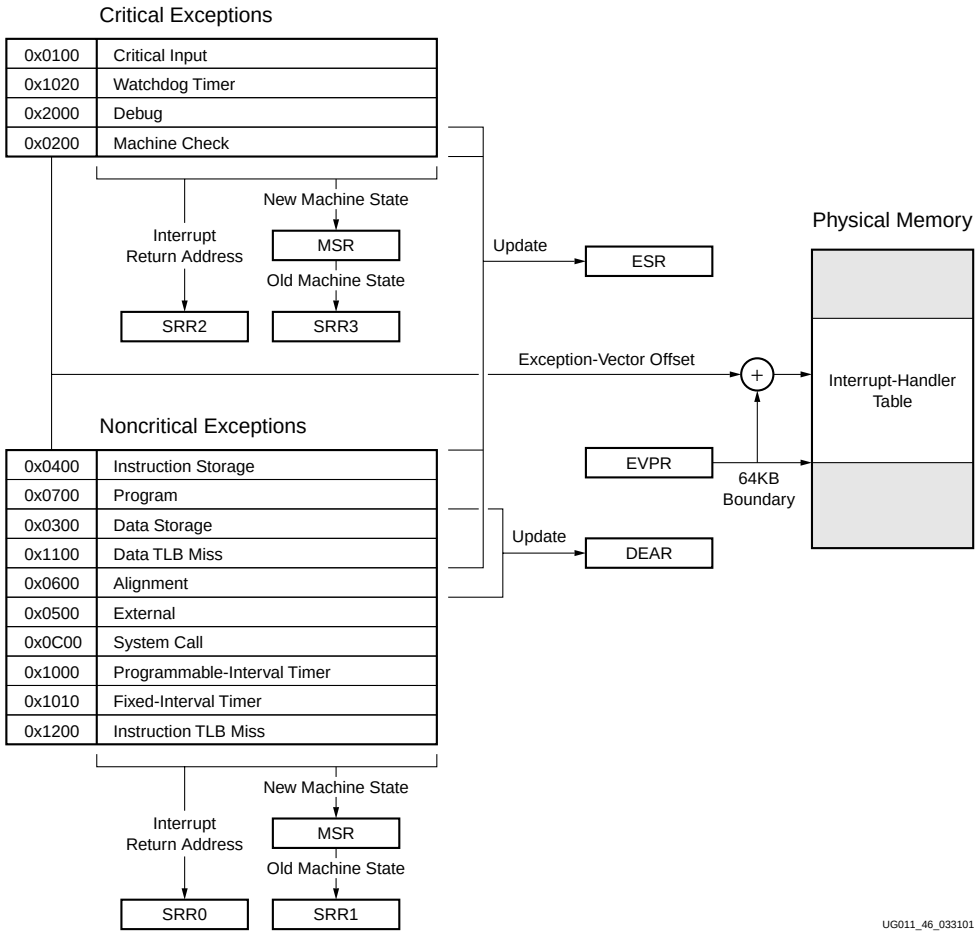


Figure 2.11: PowerPC interrupt mechanism [7].

The PowerPC supports a two-level interrupt mechanism, supporting critical and non-critical interrupts. Once the processor is servicing a non-critical interrupt, and a critical interrupt occurs, the execution of the non-critical Interrupt Service Routine (ISR) is interrupted and control flow is transferred to the critical ISR. To preserve the processor state on an interrupt, the current state is saved in a set of Save and Restore Registers (SRRs). Critical and non-critical interrupts each have their own set of SRRs. Figure 2.11 depicts the PowerPC interrupt mechanism and the use of the exception vector table.

2.3.3 PowerPC Memory Management

The PowerPC supports two memory modes, virtual mode and real mode. In real mode, effective addresses are directly mapped to the physical address space. In virtual mode, effective addresses are translated to physical addresses using the MMU which is part of the PowerPC core.

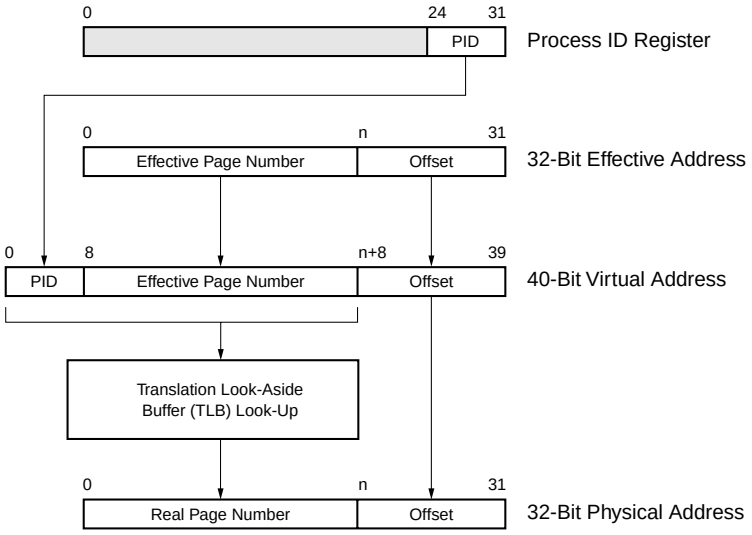


Figure 2.12: Address translation in the PowerPC MMU [7].

Operation in virtual mode allows the applications (or operating system) to map different effective addresses to the same physical address. Translation of an effective address uses lower part of the Process IDentification (PID) register to translate a 32-bit effective address to a 40-bit virtual address. A virtual address is then translated to a 32-bit physical address using the Translation Look-Aside Buffer (TLB). If a TLB request fails, meaning the entry requested is not in physical memory, the Memory Management Unit (MMU) raises an exception. This mechanism can be used by an operating system or application to implement a swapping system. Figure 2.12 depicts the translation process.

The TLB consists of a number of TLB entries. A TLB entry is created by the software executed on the PowerPC. Each TLB entry represents one region in physical memory. Figure 2.13 depicts the layout of a TLB entry. The TAG entry represents the effective

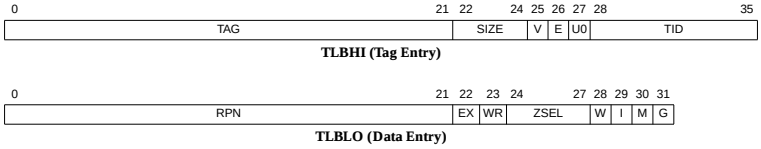


Figure 2.13: Contents of a TLB entry [7].

page number, the size denotes the size of the memory region and is used to determine the number of bits used as offset. The Translation IDentification (TID) field stores the lower part of the PID of the current process and is used to determine if the current process is allowed to access the memory region. In virtual memory mode, the PowerPC also provides memory access protection. The use of the Zone-Protection system overrides the properties set in a TLB entry. The ZPR allows the identification of 16 zones with different properties. For a detailed description on the memory translation and memory protection system, we refer to [7].

2.4 Xilinx Intelectual Property Interface

The Xilinx Coorporation provides a large amount of Intellectual Property to be used with their FPGAs. Most of the Intellectual Property (IP) is distributed as part of the Embedded Development Kit [15] (EDK). The Xilinx Intellectual Property InterFace (IPIF) is an interface which is used internally by the IP provided by Xilinx. The EDK toolset also provides the ability to connect custom hardware, supporting the IPIF to one of the PowerPC's buses (PLB or OPB).

The IPIF supports a large number of features, in this section we will discuss a simple read and write operation using the PLB IPIF and the IPIF DDR controller which constitute to the Xilinx PLB DDR controller [16].

The IPIF uses a naming converting which makes it easy to identify the source and destination of a signal.

- The *bus2ip_* prefix denotes signals originated in the PLB IPIF controller
- The *ip2bus_* prefix denotes signals originated in the IPIF DDR controller

An IPIF read is initiated by supplying a valid value on the address bus (`bus2ip_addr`), the byte enable bus (`bus2ip_be`) and asserting the chip-select (`bus2ip_cs`), the read not write (`bus2ip_rnw`) and the read request (`bus2ip_rdreq`) signals. The address and byte enable must be kept valid if the read request is asserted. The DDR memory will respond by acknowledging the address (`ip2bus_rdaddrack`) and acknowledging the read (`ip2bus_rdack`). The read data (`ip2bus_data`) becomes valid together with the read acknowledge. Figure 2.14 depicts the waveform of an IPIF read.

An IPIF write is initiated by supplying a valid value on the address bus (`bus2ip_addr`), the byte enable bus (`bus2ip_be`), the data bus (`bus2ip_data`) and asserting the chip-select (`bus2ip_cs`) and the write request (`bus2ip_wrreq`) signals. The

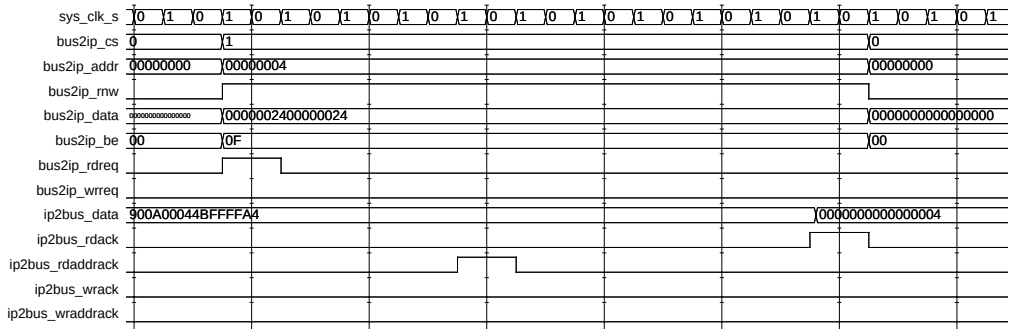


Figure 2.14: IPIF read operation waveform.

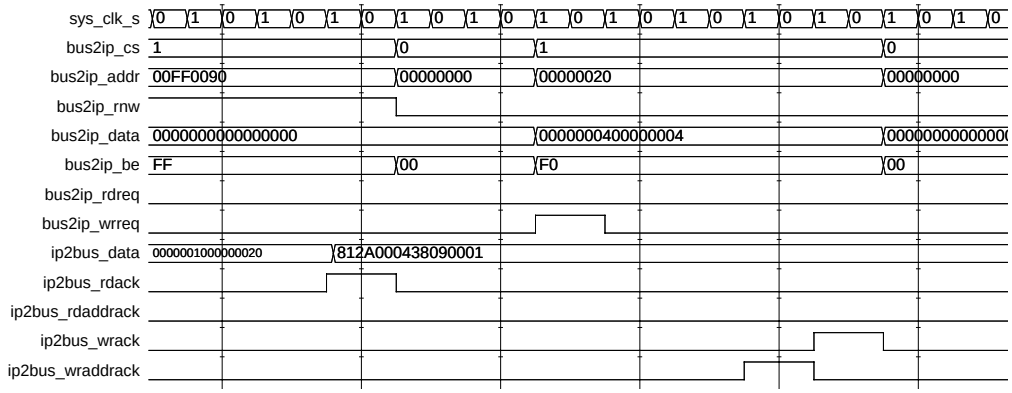


Figure 2.15: IPIF write operation waveform.

address, data and byte enable must be kept valid if the write request is asserted. The DDR memory will respond by acknowledging the address (`ip2bus_wraddrack`) and acknowledging the write (`ip2bus_wrack`). Figure 2.15 depicts the waveform of a IPIF write.

2.5 Molen Prototype

After introducing the basic concepts of the use of FPGAs and the embedded PowerPC processor, we now introduce the Molen prototype. The prototype is as starting point for the work presented in this thesis.

The Molen prototype [8] is an architecture which uses a standard GPP combined with a reconfigurable unit, the CCU. A prototype of the Molen machine organization is implemented on the XUPV2P development board [1]. The PowerPC embedded in Virtex-II Pro FPGA is used as GPP and extended with a CCU which resides in the rest of the FPGA.

The current organization of the Molen prototype uses the ISOCM bus to connect the instruction memory to the PowerPC and the Data-side Processor Local Bus (DPLB) bus

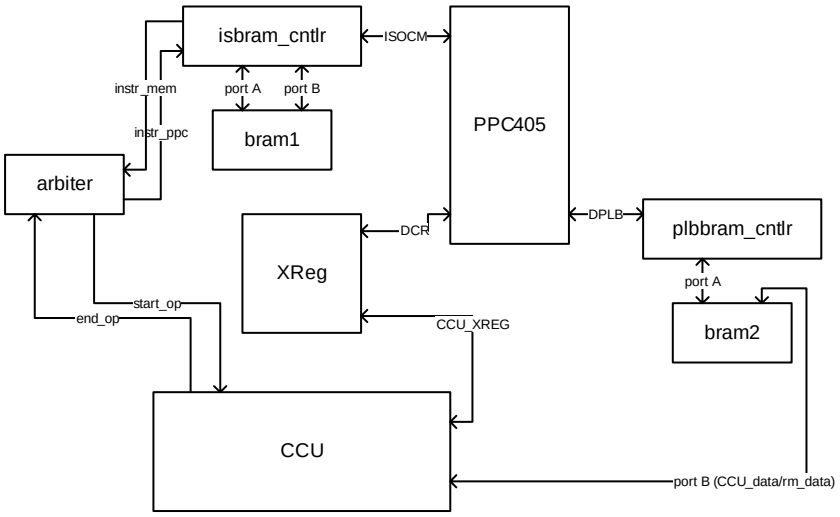


Figure 2.16: Architectural organization in the Molen prototype.

to connect to the data memory. The instruction memory is implemented using 64KB of BRAMs connected to the ISOCM bus, using a modified version of the `isbram_cntlr` supplied by Xilinx. The data memory is implemented using 64 KB of BRAMs connected to the DPLB bus using the `plb_bram_if_cntlr` supplied as part of the Xilinx IP. The BRAM blocks are connected to the `plb_bram_if_cntlr` using only port A, this allows port B of the BRAMs to be connected directly to the data bus on the CCU. The Molen prototype only uses the on-chip BRAMs for data- and instruction- memory. This limits the memory size to number of available BRAMs on the FPGA. Figure 2.16 depicts a schematic overview of the organization in the Molen prototype.

A CCU can be configured with a custom hardware design, which performs partial functionality of a program. To transfer the control from the GPP to the CCU, the program source code can be annotated. Using the Molen compiler [17], the annotated source code is compiled into a sequence of instructions which loads the operands of the function configured in the CCU into the eXchange REGisters (XREGs). To start execution inside the CCU, a reconfigurable instruction is used. To select between normal and reconfigurable instructions, the Molen prototype implements an arbiter. If a reconfigurable instruction is detected, the arbiter sets the appropriate control signals to start execution in the CCU and stalls the PowerPC. To stall the PowerPC, the Molen arbiter [18] provides the PowerPC with the branch-and-link (`bl`) instruction. The branch-and-link instruction sets the Link Register (LR) of the PowerPC. As long as the arbiter detects reconfigurable instructions, the arbiter provides the branch-to-link-register (`blr`) instruction to the PowerPC, causing the processor to loop infinitely. The infinite loop is exited by supplying the branch-to-link-register-and-link (`blrl`) instruction twice. This causes the LR to be set to the next instruction address and therefore makes the PowerPC continue its normal execution.

2.6 Related Work

In this section, we will discuss previous research on the topics discussed in this thesis. We will first discuss several reconfigurable architectures. Next, we will discuss issues related to memory controllers and performance improvement using the OCM bus. Furthermore, we will discuss memory arbitration in general, and more specifically, arbitration utilizing the Xilinx Intellectual Property Interface (IPIF). Finally, we will discuss several projects which are related to executing Linux on an embedded system.

In [19], a survey of different techniques of reconfigurable computing is given. The use of reconfigurable logic as standalone processor, attached processing-unit or co-processor are discussed. The latter is applied in the Molen machine organization [20] as previously explained in this chapter. In [21] the Reconfigurable Streaming Vector Processor (RSVP) is introduced. The RSVP is a reconfigurable processor supporting only vector operations, the shape of a vector can be defined by setting the stride, span and skip values. The authors of [22] introduce an array based reconfigurable co-processor, namely, the Garp. The organization of the Garp relies on the internal organization of an FPGA, and utilizes blocks which can be interconnected to form logical units. In [23] a hardware/software subsystem is presented, REDEFinable Instruction Set (REDEFIS), this system relies on redefining the instruction set of a processor. In [24], a S-bus based co-processor is introduced, the co-processor consists of several FPGAs which are managed by a S-bus controller. The National Adaptive Processing Architecture (NAPA) is presented in [25], the NAPA utilizes a co-processor connected to a GPP through a separate bus. In [26], the impact of different memory interfacing techniques on performance is discussed. Moreover, the impact on memory coherence involving the different systems is also discussed.

Improving memory performance is a well-discussed topic. Improvements are achieved by access scheduling [27], prefetching [28, 29], or pipelining if separate banks are addressed [30]. In addition, caches are generally utilized to improve performance [31, 32]. Possible performance improvement of using the OCM bus instead of using the PLB is presented in [33]. The use of the OCM bus to connect DDR memory is only considered in [34], the performance gain by changing the external memory from the PLB to the OCM bus is presented. In [34] only the DSOCM bus is utilized to connect DDR memory.

When multiple controllers are connected to the same physical memory, memory arbitration is involved. In [35], three important metrics for memory arbitration in general are introduced. Memory arbitration must be fair, a memory arbiter must have a low overhead and a memory arbiter must be easy to insert. In [36], a multiprocessor system using IPIF arbitration is implemented.

The Linux OS [37] is an open-source OS. Therefore, it is well-suited to be ported to support embedded devices and development boards. There are several project presenting guidelines on how to port and execute the Linux operating system on embedded development board, similar to the ML403 and XUP board which are discussed previously. In [38] a guide is presented on how to port the Linux 2.4.26 kernel to Xilinx FPGA development boards. In [39] the MontaVista [40] Linux distribution is ported to the Xilinx University Program (XUP) board.

2.7 Conclusion

In this chapter, we introduced the background on the systems and concepts used throughout this thesis. Firstly, we gave the description of two Xilinx development boards and their FPGAs. Secondly, we explained the internal organization of IBMs PowerPC, which is embedded in the FPGAs utilized. Thirdly, we introduced the Xilinx IPIF, this interface allows the easy interconnect of intellectual property. Fourthly, we detailed the Molen machine organization and the Molen prototype which utilizes this organization. Finally, we referred to work related to topics discussed in this thesis. We refer to other reconfigurable architectures and how they differ from the Molen machine organization. We also referred to techniques which are used to increase performance and techniques for memory arbitration. In our design, we utilize the OCM buses to increase performance, because we use both OCM buses, the arbitration of memory requests is required. Previous research shows that the utilization of the OCM buses can introduce performance improvement. Furthermore, we introduced work on the use of the Linux OS as an embedded operating system, which we will use in combination with our design.

Memory configurations for large applications on the Molen prototype

3

In Section 2.5, we gave a summary of the architecture of the Molen prototype. The use of, for instance, on-chip BRAM blocks introduces limitations on the scalability and therefore on the more general applications of the prototype. The memory organization and the limited memory size, which is implied by this organization introduces the main bottleneck.

In this chapter, we present the results of a study performed to examine the possible memory configurations. We present several options, based on changing hardware or software. Finally, we motivate and choose the most advantageous solution.

3.1 Molen with standard Xilinx PLB DDR controller

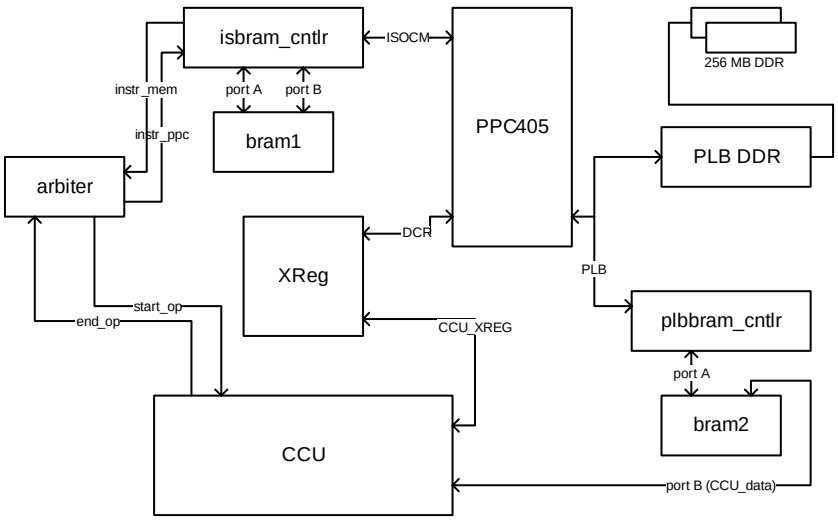


Figure 3.1: Molen machine organization for a software based selection.

One option of expanding the memory size of the Molen machine organization is by connecting a DDR memory controller to the PLB of the PowerPC. This can be realized using the PLB DDR controller provided by Xilinx [16]. The Molens CCU requires a direct connection to the data memory, in the current prototype, this connection is currently established by using the second interface port on a BRAM block. However, if the PLB DDR controller is used, this direct connection is not feasible and the main memory can only be accessed by using the PowerPC if applied in the current Molen prototype.

Therefore, when using the PLB DDR controller, we propose the use of a software based selection mechanism to increase the memory size of the Molen prototype. Figure 3.1 depicts the complete architecture overview with the PLB based DDR controller in place. The option presented in this section uses the BRAM blocks to store instruction sequences which contain reconfigurable instructions, and use the DDR memory to store instruction sequences which do not contain reconfigurable instructions.

The drawback of the option depicted in Figure 3.1 is that, if large applications containing reconfigurable instructions are executed, a software based mechanism must be introduced to move instruction blocks (functions) from DDR memory to the BRAM blocks. This buffering scheme introduces a large performance drawback. The discussion is the same for data as instructions, data blocks used by the CCU must reside inside the BRAM blocks, and must be copied there from DDR memory before use and copied back to DDR memory once the CCU has finished its operation.

Summarizing, using the PLB DDR controller supplied by Xilinx and leaving the rest of the Molen prototype as designed originally implies:

- Add the Xilinx PLB DDR controller to the Molen prototype.
- Develop a software system which allows the identification of programs which use the Molen specific hardware and reallocate the correct blocks of instructions and data into the BRAM blocks accessible by the Molen hardware.

3.2 Molen based on a PLB only bus structure

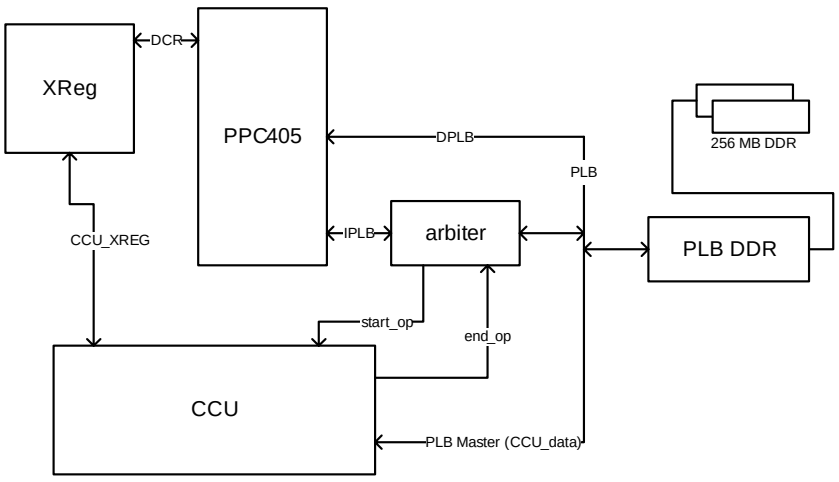


Figure 3.2: Molen machine organization based on only the PLB.

In this section, we introduce an option to increase the memory size of the Molen prototype by porting the Molen prototype to the PLB bus.

The memory size is expanded by using the Xilinx PLB DDR controller. In the original Molen prototype, the Molen arbiter is connected to the ISOCM bus, the drawback of the

ISOCM bus is that it can only connect to on-chip BRAM blocks, which have a limited size. The option presented relies on moving the arbiter from the ISOCM bus to the instruction interface of the PLB (IPLB). Also the BRAM blocks used to store the data required by the CCU are omitted. Instead of using BRAM blocks, we introduce a PLB to CCU controller, able to retrieve data from every peripheral addressable through the PLB. The use of such a controller implies the addition of a handshaking protocol to the CCU to data memory interface. Figure 3.2 depicts the proposed organization of the Molen prototype.

Summarizing, the use of a Molen prototype based only on the PLB implies:

- Add the Xilinx PLB DDR controller to the Molen prototype.
- Port the Molen arbiter to the Instruction PLB.
- Change the CCU to data memory interface. (add handshaking)
- Develop a complex PLB to CCU controller.

3.3 Molen with ISOCM DDR and DSOCM DDR memory

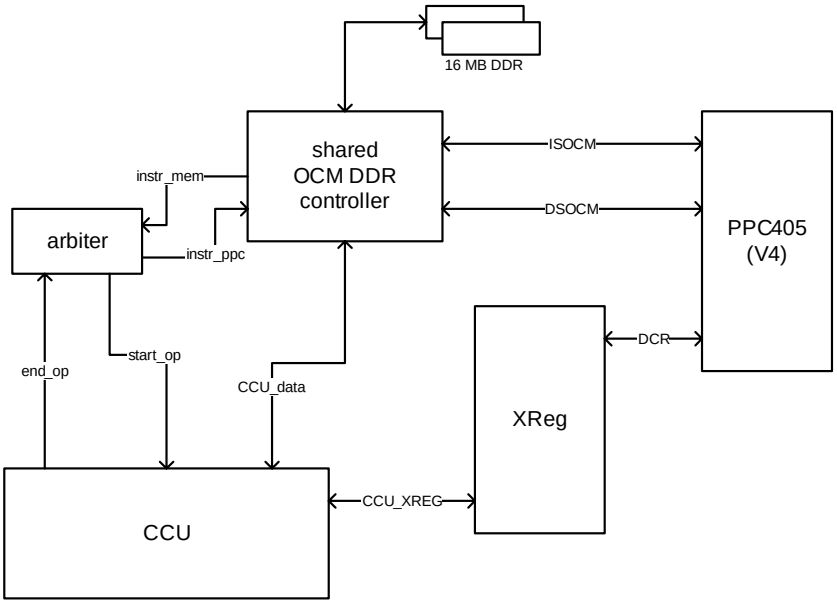


Figure 3.3: Molen machine organization with a shared OCM DDR controller.

Another option to expand the memory size of the Molen prototype is to use the ISOCM and the DSOCM bus to connect to DDR memory. In comparison to the one cycle latency of on-chip BRAM blocks, DDR memory has a variable latency which is greater than one cycle. Because a DSOCM bus on the Virtex-II Pro does not support variable latency operations, the complete Molen machine organization should be ported

to a Virtex-4 board first. The CCU data memory interface does not support variable latency operations either, to interface with DDR memory, a handshaking protocol should be added to this interface. No changes to the Molen arbiter are needed if the developed ISOCM DDR controller includes an interface to the arbiter.

The ISOCM bus on the Virtex-II pro, as well as the ISOCM bus on the Virtex-4 do not support variable latency read/write operations. Although the ISOCM bus does not support variable latency operations, a controller can be created supplying stall operations to the PowerPC as long as no valid instruction data is available.

Since there is one physical DDR interface, the controller to be developed should include an arbitration between accesses on the ISOCM, DSOCM and CCU data bus, because all three modules will access the same physical DDR memory. Figure 3.3 depicts an overview of the proposed Molen prototype architecture.

Summarizing, the use of a Molen prototype based only on the OCM buses implies:

- Port the Molen prototype to a Virtex-4 development board.
- Develop a shared ISOCM/DSOCM/CCU to DDR controller.
- Change the CCU to data memory interface.

3.4 Molen implemented using Virtex-4 APU

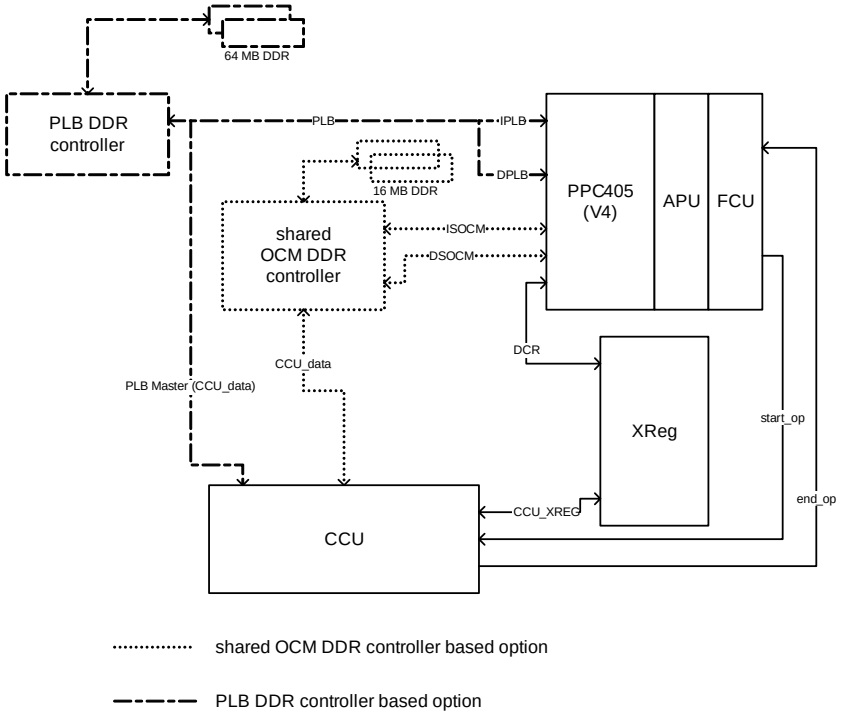


Figure 3.4: Molen prototype architecture using the APU.

A final option to increase the memory size of the Molen prototype is the Virtex-4 Auxiliary Processing Unit (APU). The Virtex-4 has a built in option to connect a Fabric Co-processor Unit (FCU) to the PowerPC [3]. The APU manages the connection between the PowerPC and the custom made FCU. This organization is very similar to the Molen machine organization and the build in features of the APU can possibly be exploited to simplify the Molen machine organization.

To adapt the Molen machine organization using the APU-FCU structure, the arbiter must be merged into the FCU decode unit because the FCU decode unit is a built in unit with comparable functionality as the Molen arbiter. The instructions used by Molen arbiter to signal reconfigurable execution must be changed to correspond with the instructions supported by the APU/FCU decoding scheme, because the opcodes used by the Molen arbiter and the FCU decode unit differ.

Apart from moving arbitration from the Molen arbiter to the FCU decoder, DDR memory must be connected. For connecting the DDR memory, the PLB DDR controller provided by Xilinx can be used or a shared OCM DDR controller as proposed in Section 3.3 can be used.

Using the PLB DDR controller, the direct CCU to data memory interface must also be implemented using the PLB. This implies the development of a PLB to CCU controller as proposed in Section 3.2. Using the shared OCM DDR controller as proposed in Section 3.3, the CCU data memory interface also needs to be modified to handle the variable latency of the DDR memory. Therefore, a handshaking protocol should be added to the CCU data memory interface. Figure 3.4 depicts the proposed Molen prototype architecture based on the PLB DDR controller or the shared OCM DDR controller (Section 3.3). Summarizing, the use of a Molen prototype based only on the Virtex-4 APU implies:

- Port the Molen prototype to a Virtex-4 development board.
- Change the CCU to data memory interface.
- Change the Molen software interface (opcodes used to identify reconfigurable execution).
- Develop a shared ISOCM/DSOCM/CCU to DDR controller or a complex PLB to CCU interface.

3.5 Motivation

The Molen prototype, as implemented on the XUPV2P development board (a detailed description is given in Section 2.5) is a proof of concept, but the used memory organization is too small to support an application with large memory footprint or even an operating system like Linux. In order to support applications with a large memory footprint, the memory size should be increased, which implies re-thinking the Molen memory organization. The options presented in this chapter propose solutions to achieve this goal.

The option proposed in Section 3.1 is based on a minor change in hardware, and major changes in software. The advantage of this option is that the changes proposed are backwards compatible (older CCUs can be used without any problems). The disadvantage of this option is the need to develop a complex software mechanism to reallocate data used by the CCU into the BRAM blocks, causing a performance decrease. Summarizing, this option introduces a way to increase the memory size of the Molen prototype, but does not introduce a structural improvement.

The option presented in Section 3.2 involves major changes in hardware and minor changes in software. The advantage of this option is the use of the Xilinx IP available for the Virtex-II Pro as well as the Virtex-4 FPGAs. The disadvantages of this option are the need to develop a complex PLB to CCU interface which also requires the CCU interface to change and therefore lose backwards compatibility. Also the Molen arbiter must be re-designed in order to deal with instructions transferred using the PLB.

The option presented in Section 3.3 also involves major hardware changes, and minor software changes. The advantage of this option is that the created memory organization is based on the OCM buses, like the original Molen prototype. Which implies that no changes to the Molen arbiter are needed. The option presented in Section 3.3 also utilizes the benefits of the OCM buses as mentioned in [33] and [34]. The disadvantage of this option is the need to change the CCU to data memory interface, which results in loosing backwards compatibility.

The option presented in Section 3.4 involves major hardware and software changes. An advantage of this option is the use of existing embedded features in the Virtex-4 FPGA, which leaves more chip resources for a custom design. The disadvantages of these option are the design of a complex PLB to CCU interface; the development of a shared OCM DDR controller in combination with mapping the Molen arbiter to the FCU decode unit, and changing the software interface.

	Hardware	Software	Performance
Standard PLB DDR	++	−−	−−
PLB only	−−	+	+−
ISOCM DDR and DSOCM DDR	−	++	++
Virtex-4 APU	−	−−	+
	Molen arbiter compatibility	Backwards compatibility	
Standard PLB DDR	++	++	
PLB only	−−	+−	
ISOCM DDR and DSOCM DDR	++	+−	
Virtex-4 APU	n.a.	−−	

Table 3.1: Advantages / disadvantages of proposed Molen prototype organizations (++ = advantageous, −− = disadvantageous)

Table 3.1 gives an overview of the advantages and disadvantages of the options proposed in this chapter. In this table, we consider the expected development effort for both

the hardware and software portion of the proposed design. Moreover, we consider the expected performance of the Molen prototype if the solution proposed is incorporated. Finally, we give an indication on the backwards compatibility, and therefore re-usability, of the Molen arbiter [18] and existing CCUs designs, if the proposed option is adapted.

Developing an integrated shared OCM based DDR controller (discussed in Section 3.3) implies minor changes to existing CCU implementations, since a CCU only needs to be extended with a handshaking protocol to interface with variable latency DDR memory. The design also implies no changes to the software interface, since the position of the Molen arbiter does not change. The design can also be used outside of the scope of the Molen prototype to provide a flexible and fast external memory controller for applications with a memory footprint between 1MB and 16MB, or applications which need a von Neumann like memory system (i.e. shared instruction and data memory). For these reasons, we chose to develop a shared OCM DDR controller which we will explain in detail in Chapter 4.

3.6 Conclusion

In this chapter, we proposed four options for expanding the memory size of the Molen prototype. These options are based upon a combination of hardware and software changes which utilize external DDR memory. The first two options utilize an existing PLB DDR controller supplied by Xilinx. The third option introduces a DDR controller utilizing the OCM buses of the PowerPC. The forth option utilizes the APU incorporated in the embedded PowerPC of the Virtex-4 FPGAs. This APU is used in combination with either the existing PLB DDR controller or the newly introduced shared OCM DDR controller.

In Section 3.5, we gave an overview of the advantages and disadvantages of each proposed solution. We also considered the impact of each option on the performance of the Molen prototype, the re-usability of existing CCU designs and the changes to the Molen arbiter. Finally, we motivated the choice for the development of a shared OCM DDR controller. The design and implementation of this controller is detailed in Chapter 4.

Shared On-Chip Memory Double Data Rate Memory controller

4

In Chapter 3, we presented several solutions for adapting the memory organization of the Molen prototype. As concluded, the shared OCM DDR controller is a solution which allows us to create a prototype, able to run standalone applications or an operating system without major software development work involved.

In this chapter, we present the development of three external DDR controllers for the PowerPC embedded in the Virtex-4 chip. The controllers utilize the DSOCM, the ISOCM and both ISOCM and DSOCM to implement the interface between the embedded PowerPC and the external DDR memory.

Unlike the option discussed in Chapter 3, the controller discussed in this chapter does not include a dedicated interface to a CCU. Nevertheless, the controller is designed in a way it can be easily extended to incorporate an interface to a CCU, the implementation is not discussed in this thesis. The design presented in this chapter is to be published in August 2007 [41].

4.1 General On-Chip Memory Double Data Rate Memory controller

Designing a memory controller, can be done from scratch or we can use existing IP. The OCM bus protocol, covered in Section 2.3.1, is a relatively simple protocol. On the other hand, DDR specific control requirements like pre-charging and initialization make interfacing with DDR quite troublesome and error prone. Therefore, we would preferably reuse existing IP to interface with the DDR memory and possible reuse existing IP to

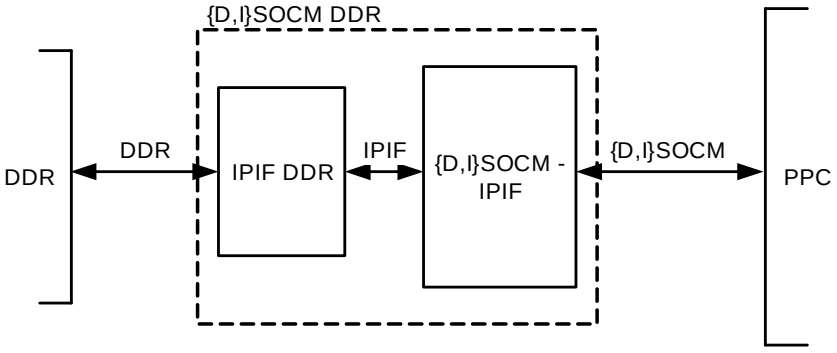


Figure 4.1: General OCM DDR controller organization.

interface with the OCM buses.

The Xilinx Coorporation provides a PLB DDR controller as part of its EDK (Section 2.4). This controller consists out of two main parts, namely: a PLB to IPIF controller and a IPIF to DDR controller. In order to design an OCM based controller, we can use the existing IPIF DDR controller to handle the complex interface with the DDR memory and design an OCM to IPIF controller to interface with the OCM bus.

An overview of the internal organization of a {Data,Instruction}-Side OCM DDR controller is depicted in Figure 4.1. The organization is similar to the internal PLB DDR controller organization.

4.2 Data-Side OCM DDR Memory controller (DSOCM DDR)

In this section, we will discuss the design and implementation of the DSOCM DDR controller. Section 4.2.1 focuses on the design consideration, while Section 4.2.2 discusses the implementation details.

4.2.1 Design considerations

The DSOCM DDR controller is based on the design described in [34]. The controller proposed in [34] is targeting the Xilinx Virtex-II Pro FPGA ([11]), the controller presented in this chapter is targeting the Xilinx Virtex-4 FPGA. The reason to use the Virtex-4 is because the DSOCM bus implemented on the PowerPC embedded in the Xilinx Virtex-4, includes a handshaking protocol. This protocol allows us to connect memory with variable latency, such as DDR memory, to the DSOCM bus.

In this section, we will focus on the design issues related to the DSOCM to IPIF block of the DSOCM DDR controller. As mentioned earlier, the design of the IPIF to DDR controller is standard by Xilinx. The DSOCM to IPIF controller design forwards a request on the DSOCM bus to the IPIF interface. The 22-bit value read on the DSOCM address bus is translated to a 32-bit IPIF address. However, the DSOCM bus uses 32-bit data words, whereas the IPIF bus uses 64-bit data words. To select between the upper or lower 32-bit word of the IPIF bus, the byte enable signal of the IPIF bus is used. A DSOCM bus read is always 32-bit, a DSOCM write can be one to four bytes. The byte write signal combined with the least significant bit of the requested address is used to correctly set IPIF byte write signal, the least significant bit of the requested address is used to select the correct word read from memory. The DSOCM read data bus must be valid when the handshaking (complete) signal is asserted.

4.2.2 Implementation

The controller is implemented using a Finite State Machine (FSM), an address path and a data path. Figure 4.2 gives an overview of the general implementation.

Address path: The address path consists of combinatorial logic, and it provides information to the FSM, if the request is aligned on a 64-bit border.

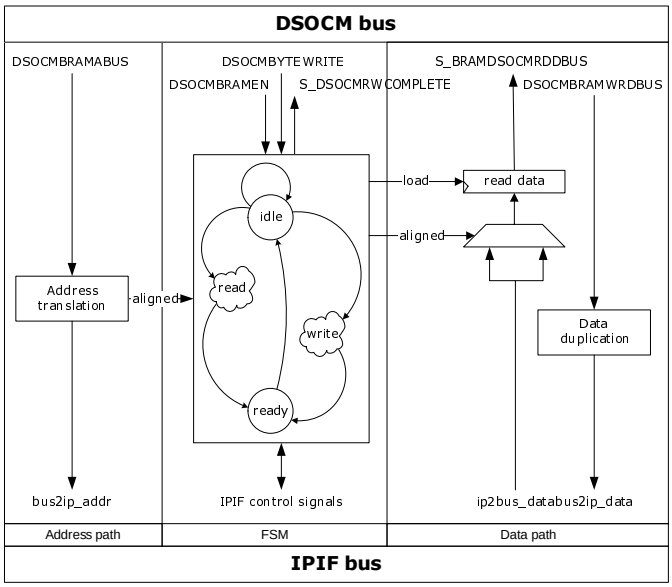


Figure 4.2: General DSOCM to IPIF controller implementation

Data path: The data path consists of a read data register, which stores the 32-bit data selected a multiplexer controller by the FSM. The read data register is necessary to provide a stable signal on the DSOCM read data bus.

FSM: The control contains four basic operations, idle, read, write and ready. Figure 4.3 depicts the FSM state transition graph.

Idle: The idle state is used to differentiate between a read and a write operation. In this state the byte mask is converted to the byte mask used by the IPIF bus. Using the least significant bit of the address, the controller determines if the read/write data is 64-bit aligned.

Read/Write: A read/write operation consists out of three separate states, namely: {READ,WRITE}_INIT, {READ,WRITE}_ADDR_WAIT and {READ,WRITE}_WAIT. The {READ,WRITE}_INIT state initiates the read/write request on the IPIF bus by asserting the `bus2ip_cs`, the `bus2ip_{rd,wr}req` signals and in case of a read, the `bus2ip_rnw` signal. The next clock cycle, the state is updated to the {READ,WRITE}_ADDR_WAIT state. The `bus2ip_cs` is kept asserted throughout the complete operation. For a READ operation, the `bus2ip_rnw` will also be kept asserted. The `bus2ip_{rd,wr}req` will only be asserted during the {READ,WRITE}_INIT state.

The {READ,WRITE}_ADDR_WAIT state is used to wait for the connected IPIF device, in this case theDDRR memory, to respond with asserting the `ip2bus_{rd,wr}addrack` signal. Once the signal is asserted, the state is updated to the {READ,WRITE}_WAIT state.

The READ_WAIT state is used to store the data read (assert read register load signal) and select the upper or lower word of data using the alignment signal calculated in the IDLE state. Note that only the data stored in the last cycle of the READ_WAIT will be used. Once the IPIF DDR controller asserts the `ip2bus_rdack` signal, the read

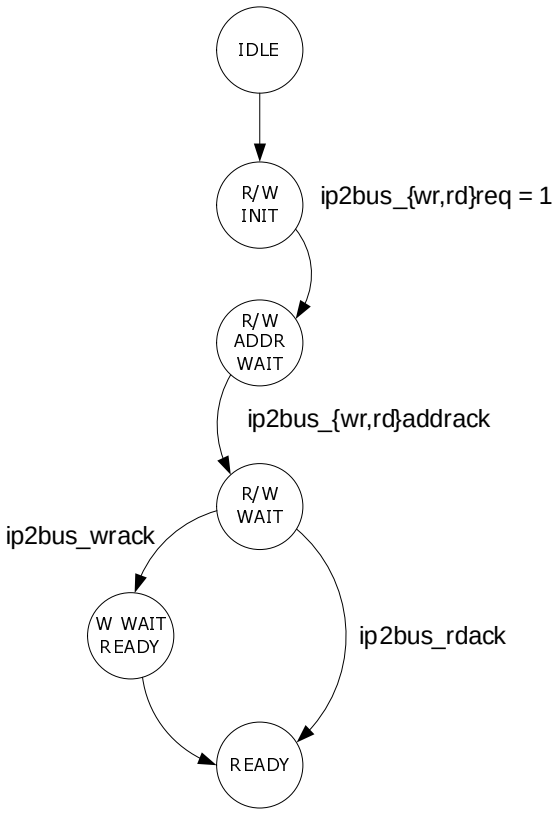


Figure 4.3: DSOCM controller FSM state transition graph.

operation is finished and the state is updated to the main READY state.

The WRITE_WAIT state is used to wait for the connected IPIF device to respond with asserting the `ip2bus_wrack` signal. Once the signal is asserted, the state is updated to the WRITE_WAIT_READY state. To allow the DDR memory to finish the write operation, the WRITE_WAIT_READY state is introduced. This state waits a number of clock cycles equal to the memory latency before updating the state to the READY state.

Ready: The ready operation uses the READY state to set the `s_dsocmrwcomplete` signal on the DSOCM bus. The state is updated to the IDLE state after one clock cycle.

4.3 Instruction-Side OCM DDR Memory controller (ISOCM DDR)

In this section, we will discuss the development of the ISOCM DDR controller. We will introduce the design considerations in Section 4.3.1, and elaborate on the implementation details in Section 4.3.2.

4.3.1 Design considerations

The ISOCM DDR controller consist of two modules: the IPIF DDR controller and the ISOCM IPIF controller. The design of the latter is discussed in this section. Because, as mentioned in Section 4.1, the former is provided by the Xilinx.

The ISOCM bus has no facility to stall or idle the processor because the bus is designed to interface only with the on-chip BRAM blocks. The on-chip memory provides valid data one clock cycle after the address is put on the ISOCM address bus. Since the connected DDR memory is slow compared to the on-chip memory, the controller must keep the processor idle until the DDR memory provides the valid instructions. To keep the processor idle, a scheme like the one used in the Molen arbiter [18] is chosen.

The scheme used in the Molen arbiter keeps the processor in an infinite loop using the `bl` instruction, while execution is transferred to the CCU. To return from the loop, the `blr` instruction is used. In our design, the internal state of the processor must remain unchanged. Moreover, the register file and thus the link-register must remain unchanged. Therefore, the `bl` instruction cannot be used in our design. The `b` instruction is used instead, because it does not change the internal state of the processor, nor the register-file. The processor is provided a relative address offset of zero to be used in the branch. This causes the processor to re-request the same instruction until the DDR memory provides a valid instruction and the program counter is incremented. All memory accesses to OCM connected memory are not cacheable, therefore, the processor will not dead-lock.

If the processor is interrupted in the middle of an instruction fetch, special care must be taken. For reference, the interrupt mechanism of the PowerPC is explained in Section 2.3.2. On an interrupt request, the processor stops execution, saves the return state and current program state [42]. Once the state is stored, the processor fetches the first instruction of the corresponding ISR. The ISOCM DDR controller must be able to detect an interrupt, because the ISOCM bus does not support a facility to signal the occurrence of an interrupt.

To signal if the main control flow is interrupted by an interrupt request, the address of the requested instruction is monitored and compared to the address used for the request sent to the DDR memory. If the address changes, the ISOCM to IPIF controller assumes the program flow is interrupted and the requested instruction is no longer needed. The controller waits for the DDR memory to finish and discards the retrieved data. Once the DDR memory is back to its idle state, the request for the first instruction of the ISR is sent to the memory. During the completion of the active request and until the valid instruction of the ISR is available, the controller provides the `b` instruction to the processor.

4.3.2 Implementation

The ISOCM to IPIF controller is implemented using three main blocks, namely: the address path, the data path and a FSM to control both. Figure 4.4 depicts the general overview of the controller.

Address path: Because the address only needs to be valid on the rising edge of the clock, the requested address is buffered by a rising edge triggered flip flop. If no other

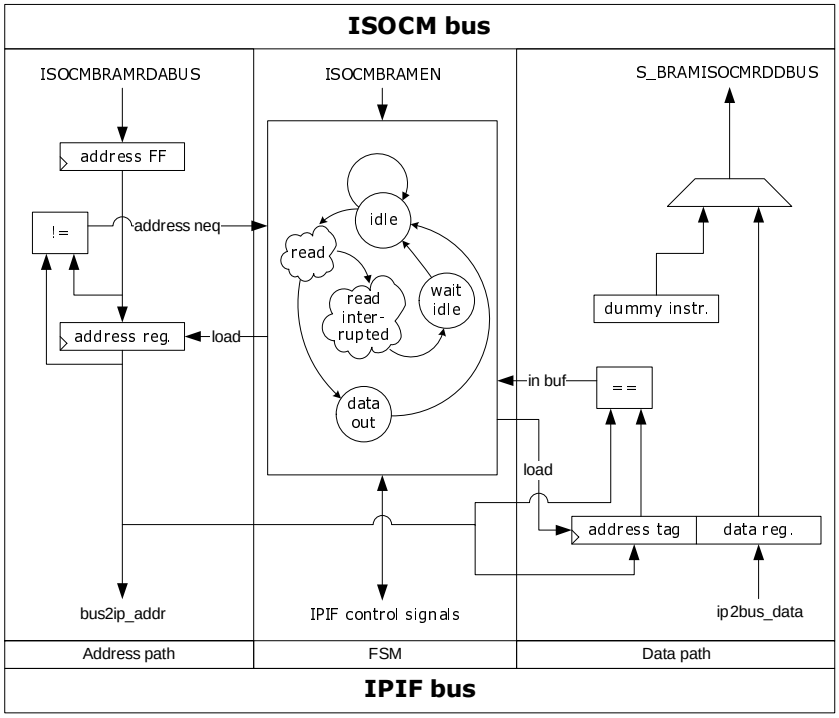


Figure 4.4: General ISOCM to IPIF controller implementation.

request is pending, the requested address is stored in the address register and is kept stable throughout the request sent to the IPIF device.

The stored address is compared to the latched incoming address, to signal an interrupt request. Once an address change is signaled, the FSM continues the request using the interrupted read state sequence.

Data path: Once valid data from the DDR memory is available, the `ip2bus_rdack` signal is asserted. The data is stored in the data register together with the corresponding address read from the address register.

If the requested data is stored inside the data register, the control state machine will select the buffer as source for the `s_bramisocmrddbus` signal. If an interrupt was signaled during the read from DDR memory, the data read from the memory (and its address) is stored in the data register, but the data is not selected for output. Therefore, the `s_bramisocmrddbus` signal still has the dummy instruction as data source.

FSM: The FSM is used to control both address and data path. The FSM uses two main paths, namely: a normal read operation and an interrupted read operation. Figure 4.5 depicts the state transition graph of the normal and interrupted read flow. For simplicity, the conditions where a state does not change and the WAIT_IDLE state are omitted in Figure 4.5.

While the `isocmbramen` signal is deserted, the controller remains in IDLE state. If the controller is serving an active request while the `isocmbramen` signal is deserted, the state machine is set to the WAIT_IDLE state and returns to the IDLE state as soon as

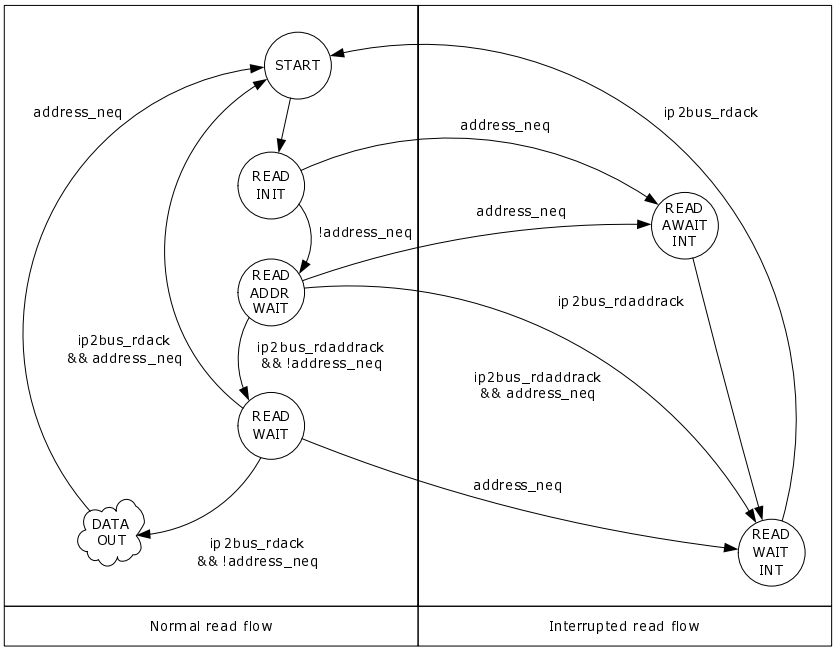


Figure 4.5: FSM transition graph of normal and interrupted flow.

the `ip2bus_rdack` signal is asserted.

Generally, as soon as the `isocmbramen` signal is asserted, a request is started by entering the START state. In the START state, the address is loaded into the address register and the state is updated to DATA_OUT, if the data is already inside the buffer, or the state is updated to READ_INIT, which will start a read request at the connected DDR memory. In the READ_INIT state, the `bus2ip_rdreq` signal is asserted and the state is updated to the READ_ADDR_WAIT state.

Once the connected DDR memory responds by asserting the `ip2bus_rdaddrack` signal, the state machine updates the state to the READ_WAIT state. The READ_WAIT state is used to wait for the DDR memory to respond with the `ip2bus_rdack` signal. Once the read acknowledge signal is asserted, the state is updated to the first DATA_OUT state. While in the READ_WAIT state, the data is stored inside the data register. Note that only the last value stored will be used for read data on the ISOCM bus.

The first and second DATA_OUT states are used to redirect data to the ISOCM bus, the state machine leaves the second DATA_OUT state as soon as an address change is detected.

An interrupted read is signaled by monitoring the address comparison, once a address change is detected, the state machine enters an “interrupted” flow, which is equal to the normal flow, only without the DATA_OUT stages. On an interrupted read, the state machine returns to START state to handle the next request.

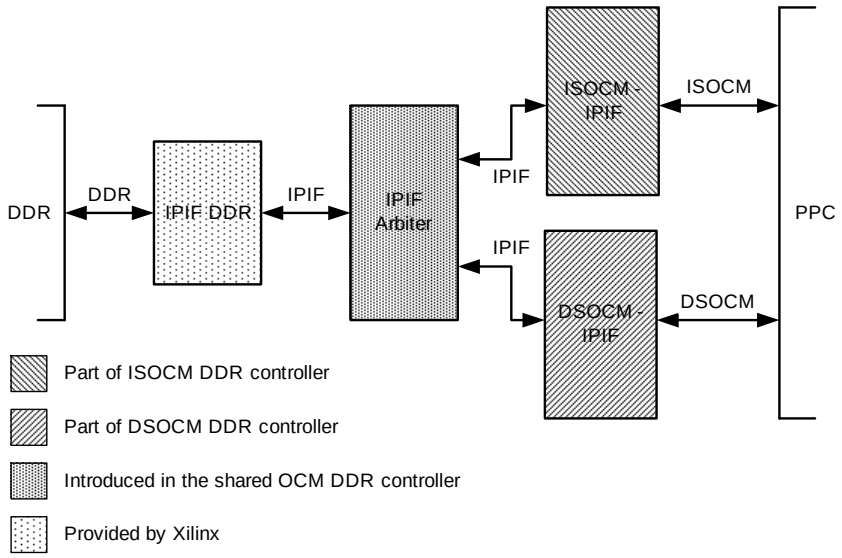


Figure 4.6: Internal organization of the shared OCM DDR controller.

4.4 Shared OCM DDR Memory controller

This section will discuss the integration of the DSOCM DDR and ISOCM DDR controllers, detailled previously, into a shared OCM DDR controller. The DSOCM DDR and ISOCM DDR controller both use the Xilinx IP interface internally (see Section 4.1 for details), this allows us to reuse the DSOCM to IPIF and ISOCM to IPIF modules. Figure 4.6 depicts the internal organization of the shared OCM DDR controller using the DSOCM to IPIF and ISOCM to IPIF modules. The organization as depicted in Figure 4.6 implies the development of an IPIF arbiter. Apart from designing an IPIF arbiter, the integration of the DSOCM DDR and ISOCM DDR controllers into a shared OCM DDR controller, consists of connecting the internal IP interfaces properly. Therefore only the design of the IPIF arbiter will be discussed in this section.

4.4.1 N-way IPIF Arbiter

In this section, we will discuss the development of the IPIF arbiter which is used in the shared OCM DDR controller. The development is split into two parts, Section 4.4.1.1 discusses the design considerations and Section 4.4.1.2 discusses the implementation of the arbiter.

4.4.1.1 Design considerations

The IPIF arbiter has to selects to whom access to the DDR memory is granted. The basic idea for the IPIF arbiter is inspired by a DDR multiplexer introduced in [36].

Each connected controller can send a request at any time, even if access to the DDR memory is granted to another controller. To monitor the requests sent by each

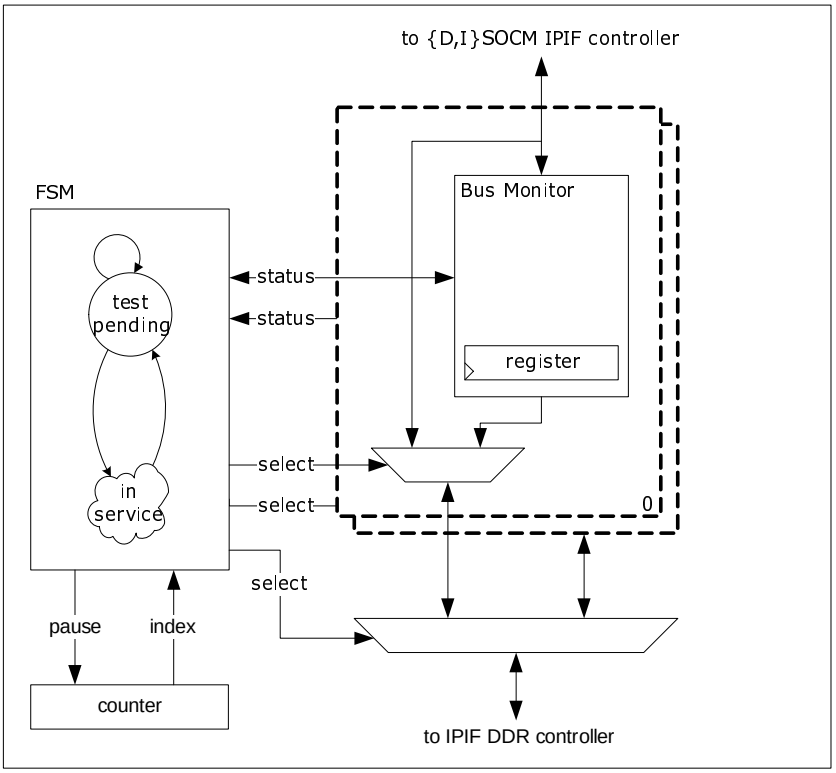


Figure 4.7: General IPIF arbiter design.

controller, they are connected to a Bus Monitor. A Bus Monitor detects a request on the IP interface and buffers the first cycle of the request in order to preserve a request sent by the controller.

The IPIF arbiter uses a FSM to test for pending requests and to grant DDR memory access if the DDR memory is not used by another controller. To ensure a fair selection mechanism, the FSM uses an up-counter which overflows as soon as the number of connected controllers is reached (this number is equal to the number of connected bus-monitors). If a request is selected to be served, the counter is paused and the count value is used to select the signal to route the IP interface of the DDR memory to one of the controllers. The use of a up-counter results in a round-robin like scheme for granting the DDR memory IP interface to a connected controller. Figure 4.7 depicts a general overview of the IPIF arbiter design.

4.4.1.2 Implementation

Bus Monitor: A read or write request to the DDR memory is started by asserting the corresponding request signal on the IPIF, `bus2ip_rdreq` for read, `bus2ip_wrreq` for write. For a multi-word request, these signals are kept asserted for a period depending on the number of words requested. For a single-word request, the signal will only stay asserted for one cycle. To make sure a request is preserved the way it is sent by the

connected controller, all `bus2ip_*` IPIF signals are stored in a register inside the Bus Monitor.

Once the Bus Monitor detects a request made by the connected controller, the `pending` signal is asserted. The internal state is updated to a wait state.

The waiting state is used to wait for the main arbiter control to grant device access to the controller and assert the `in_service` signal. Once this signal is asserted, the IPIF bus monitor goes to a servicing state to wait for the connected device to finish. The monitor signals the end of an operation by monitoring the `ip2bus_rdack` and `ip2bus_wrack` signals. If a completed operation is detected, the IPIF bus is kept idle for one cycle and the `finish` signal is asserted.

FSM: The FSM of the IPIF arbiter has to detect a pending request from one of the connected controllers and grant the device IPIF bus (DDR memory) to a controller with a pending request. The FSM consists of two main states, namely: TEST_PENDING and IN_SERVICE. The FSM starts in the TEST_PENDING state, this state is used to test each connected bus monitor for a pending request by checking the `pending` signal corresponding to the controller indexed by the counter. The control checks a bus monitor for a pending request each clock cycle. If no pending request is detected, the counter is incremented.

If a pending request is detected, the `in_service` signal for that device is asserted. The first cycle after a pending request is detected, the `bus2ip_*` signals stored inside the IPIF bus monitor are connected to the device IPIF bus. The subsequent cycles, the complete controller IP interface is routed to the device IPIF bus. Once the FSM enters the IN_SERVICE state, the counter is paused.

4.5 Conclusion

In this chapter, we presented the design of three DDR controllers using the OCM buses of the PowerPC embedded in the Virtex-4 chip. We first presented general issues relating the development of DDR controllers and introduce the advantage of using the IPIF utilized by Xilinx IP. Secondly, we have presented the design of a DSOCM DDR controller, which can be used separately if needed. Thirdly, we introduced a mechanism to overcome limitations introduced by the ISOCM bus implemented on the PowerPC's embedded in the Virtex-series. This mechanism, based on the scheme utilized in the Molen arbiter, allows us to keep the PowerPC in the same state while waiting for DDR memory, to provide the next valid instruction. Finally, we presented a shared OCM DDR controller which utilizes the Xilinx IPIF internally. We introduced an IPIF arbiter to determine which OCM request should be serviced

The designs presented in this chapter can be utilized to increase the memory size of embedded systems using the OCM buses. Moreover, the shared OCM DDR controller design can easily be extended to incorporate an interface to the CCU which is part of the Molen prototype.

5 Results

In this chapter, we will present the test, verification and performance evaluation results of the designs presented in Chapter 4. We verified the correct functionality of the DSOCM DDR, ISOCM DDR and shared OCM DDR controller using Modelsim, Chipscope and a test-program which extensively utilizes the controllers. Second, we performed benchmarking on both the shared OCM DDR controller and compared the results to results obtained using a PLB DDR controller based design.

5.1 DSOCM DDR controller

To verify the correct functionality of the DSOCM DDR controller (detailed in Section 4.2), a functional simulation is performed using Modelsim [43]. The simulation is performed by continuously reading and writing a word to the DDR memory and comparing the obtained waveform to the expected waveform.

Once the DSOCM DDR controllers behavior is verified using Modelsim, the design is implemented on the ML403 development board [44] (for more information, see Section 2.1.2). A Xilinx Chipscope [45] logic state analyzer is included in the design. The logic state analyzer is used to verify the behavior of the controller. Waveforms obtained using Chipscope are compared to the waveforms obtained using Modelsim. Both waveforms correspond to the expected waveforms which are depicted in Figure 5.1 and Figure 5.2.

Finally, the DSOCM DDR controller is implemented without the Chipscope logic state analyzer. To extensively test the DSOCM DDR controller, a memory test program is used.

The memory test program consists of six main phases of testing, each separated by a debug statement to monitor the progress. The memory test consists of the following stages:

1. Set every location to zero.
2. Compare data read from every location to zero.
3. Write constant value to a random address.
4. Read constant value from a random address.
5. Write random value to a sequential address.
6. Read value from a sequential address and compare to random value.

Random values are generated using the random function implemented in memtest86 [46]. The process is repeated ten times. The software test described, intensively uses the

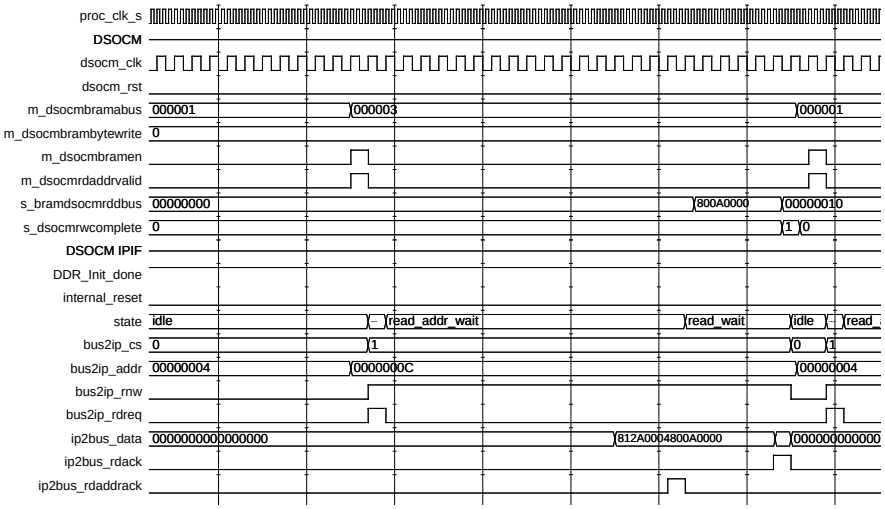


Figure 5.1: DSOCM DDR Read operation simulation waveforms.

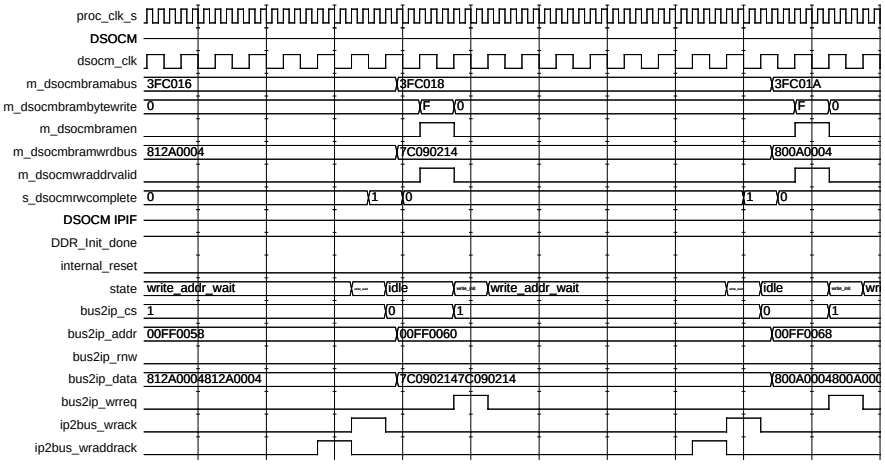


Figure 5.2: DSOCM DDR Write operation simulation waveforms.

DSOCM bus. To verify a correct functionality of the DSOCM to DDR path, we stored the program in BRAM blocks connected to the PLB. This test was successful for the complete address range of the DSOCM bus (16 Megabytes).

Table 5.1 gives the resource utilization and maximum operating frequency of the DSOCM DDR controller implemented on the Virtex4-FX12 device.

5.2 ISOCM DDR controller

To verify the correct behavior of the ISOCM DDR controller (detailed in Section 4.3), a functional simulation is performed using Modelsim [43]. To allow instructions to be

	Total available	DSOCM DDR	PLB DDR
Slices	5472	432 (7%)	597(10%)
Slice Flip Flops	10,944	537(4%)	781(7%)
4 input LUTs	10,944	574(5%)	746(6%)
Maximum Frequency:	169.874MHz		

Table 5.1: Resource utilization of the DSOCM DDR controller

```
here: li  r0,0
      li  r0,4
      li  r0,8
      li  r0,12
      b   here
```

Figure 5.3: Assembly program to verify the ISOCM DDR controller.

fetched from memory, the instructions must be written there first. A simple assembly program is used to perform a functional simulation of the ISOCM DDR controller, the instructions used by the program are written to the DDR memory before the the DDR completes its initialization cycle. Figure 5.3 depicts the assembly code of the simple program used to verify the correct behavior of the ISOCM DDR controller.

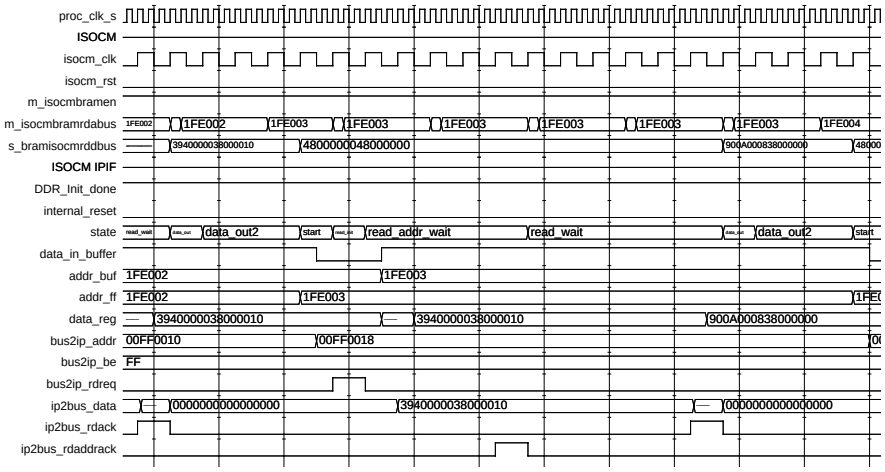


Figure 5.4: ISOCM DDR read operation waveform.

Once the ISOCM DDR controllers behavior is verified using Modelsim, the design is implemented on the ML403 development board [44] (for more information, see Section 2.1.2). A Chipscope [45] logic state analyzer is included in the design. The logic state analyzer is used to verify the behavior of the controller. Waveforms obtained using Chipscope are compared to the waveforms obtained using Modelsim. The waveform corresponds to the expected waveform which is depicted in Figure 5.4.

	Total available	ISOCM DDR	PLB DDR
Slices	5472	424 (7%)	597(10%)
Slice Flip Flops	10,944	515(4%)	781(7%)
4 input LUTs	10,944	520(4%)	746(6%)
Maximum Frequency:	189.672MHz		

Table 5.2: Resource utilization of the ISOCM DDR controller

Table 5.2 gives the resource utilization and maximum operating frequency of the ISOCM DDR controller implemented on the Virtex4-FX12 device.

5.3 Shared OCM DDR controller

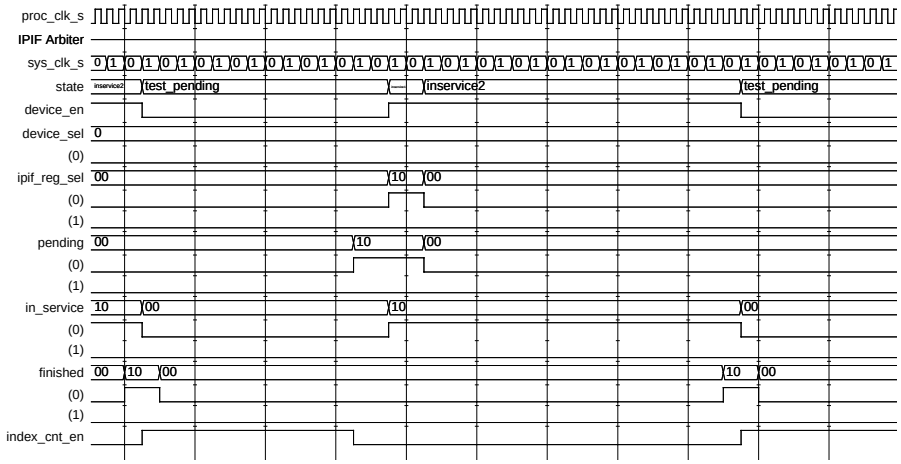


Figure 5.5: IPIF Arbiter simulation waveform.

To verify the correct functionality of the shared OCM DDR controller (detailed in Section 4.4), a functional simulation is performed using Modelsim [43]. The simulation is performed by writing a small assembly program into memory (the program is detailed in Section 5.2) using the DSOCM to DDR path and next executing the program using the ISOCM DDR path. Once these two separated programs are loaded into memory, a program containing memory read/write operations is loaded into memory (this program will be discusses later in this section). The obtained waveforms are compared to the expected waveform. Figure 5.5 depicts a selection operation of the IPIF arbiter. The actual read and write operation waveforms correspond to the waveforms as discussed in Section 5.2 and Section 5.1.

Once the shared OCM DDR controllers behavior is verified using Modelsim, the design is implemented on the ML403 development board [44] (for more information, see Section 2.1.2. A Chipscope [45] logic state analyzer is included in the design. The logic state analyzer is used to verify the behavior of the controller. Waveforms obtained using

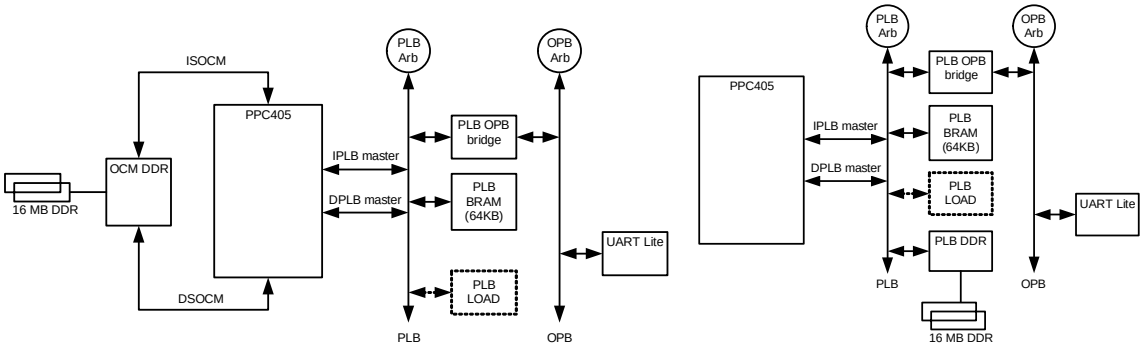
Chipscope are compared to the waveforms obtained using Modelsim, and match the expected waveform.

	Total available	OCM DDR	PLB DDR
Slices	5,472	622(11%)	597(10%)
Flip Flops	10,944	778(7%)	781(7%)
LUTs	10,944	952(8%)	746(6%)
Maximum Frequency:	123.203MHz		

Table 5.3: Resource utilization of the shared OCM DDR controller

```
for (j = 0; j < 10; j++){
  p_ddr = &ddr[0]
  for (k = 0; k < SIZE; k++){
    *p_ddr = k+j
    ddr[k] = *p_ddr
    p_ddr++
  }
}
```

Figure 5.6: Synthetic benchmark pseudocode.



(a) Shared OCM DDR-based. (b) PLB DDR-based.

Figure 5.7: Reference system overview.

Table 5.3 gives the resource utilization and maximum operating frequency of the shared OCM DDR controller implemented on the Virtex4-FX12 device.

For performance analysis, we utilized a synthetic benchmark based on two nested for-loops. The equivalent pseudocode is depicted in Figure 5.6. The boot procedure and timing functions from the Xilinx Board Support Package (BSP) are utilized. The complete program source code is included in Appendix A. Both instructions and data

should reside in the DDR memory. Therefore, instructions should be loaded into memory by means of a boot-loader or manually. In this case, due to the small code segment, we decided to load the code manually.

SIZE	OCM DDR	PLB DDR	Improvement
128 KB	3485841	3706177	6.32%
256 KB	6968810	7412237	6.36%
512 KB	13940600	14824359	6.34%
1 MB	27884687	29648602	6.33%
2 MB	55745890	59297088	6.37%
4 MB	111528345	118594062	6.34%
8 MB	223048473	237188006	6.34%
15 MB	418315051	444727411	6.31%

Table 5.4: Benchmark results w/o PLB load (x1000 cycles)

SIZE	OCM DDR	PLB DDR	Improvement
128 KB	3485841	3950069	13.32%
256 KB	6968810	7900061	13.36%
512 KB	13940600	15800044	13.34%
1 MB	27884687	31600012	13.32%
2 MB	55745890	63199947	13.37%
4 MB	111528345	126399818	13.33%
8 MB	223048473	252799557	13.34%
15 MB	418315050	473999103	13.31%

Table 5.5: Benchmark results w/ PLB load (x1000 cycles)

The program is executed on two reference designs implemented on the Xilinx ML403 development board. A Universal Asynchronous Receiver-Transmitter (UART) Lite peripheral, connected to the OPB, is used to print debug and timing information. Since part of the DDR memory contains instructions and the local variables are also stored inside DDR memory, the benchmark can not utilize the entire 16MB. Figure 5.7 depicts both the shared OCM DDR controller and the PLB DDR controller based reference system. For comparison, the results obtained by executing the synthetic benchmark on a reference system based on the shared OCM DDR controller are compared to results obtained utilizing a PLB DDR based memory system. To emulate the difference in load on the PLB, a PLB master is connected to the bus which will request the bus if it is available.

Table 5.4 and Table 5.5 give an overview of the obtained results using eight different sizes of the occupied memory. If peripherals connected to the PLB are not used, the OCM based design introduces an average performance improvement of 6.34% compared to the PLB based design. If the peripherals connected to the PLB request the bus every

cycle it is available (worst-case), the improvement is 13.34% on average. All results are obtained with the DDR burst mode and with the PowerPCs hardware caches disabled.

5.4 Conclusion

In this chapter, we presented the simulation and verification of the ISOCM DDR, DSOCM DDR and shared OCM DDR controllers. We showed that the functionality of these controllers corresponds to the expected functionality using the Modelsim simulator [43]. Moreover, we verified the correct implementation of the controllers using the Xilinx Chipscope [45] logic state analyzer. We also presented the performance evaluation of the shared OCM DDR controller implemented and tested on the Virtex-4 FX chip. We achieved an improvement from 6% (best-case) to 13% (worst-case) over a PLB based design. The best-case scenario is based on a reference system which only connects memory utilizing the PLB. The worst-case scenario is based on a reference system with memory and other peripherals connected to the PLB. In this scenario, a request is constantly sent if the PLB is available.

The resources used to implement the ISOCM DDR and DSOCM DDR controller are less than with the PLB DDR controller. Furthermore, the resources used to implement the shared OCM DDR controller are comparable to the PLB DDR controller. In our design, arbitration is included in the resource utilization, while for the PLB design arbitration is excluded.

6 Linux on an OCM based design

In Chapter 3 we presented four different options to expand the memory size of the Molen prototype. We chose to develop a shared OCM DDR controller because of the promising performance benefits and the limited number of changes to the existing Molen prototype.

In Chapter 4 and Chapter 5 we presented the development, verification and performance analysis of the shared OCM DDR controller as proposed in Chapter 3. In this chapter we will present the results of porting the Linux 2.4 kernel to a reference system based on the shared OCM DDR controller. We will first introduce the main advantages of using an operating system, second we will present the porting of Linux 2.4 with a PLB based reference system. Finally we will present the porting of Linux 2.4 with an OCM based reference system.

6.1 Linux on the ML403 board with PLB memory system

MontaVista provides a Linux distribution [40] based on the Linux 2.4 kernel. This distribution includes the board dependent files needed to run Linux on the Xilinx ML403 development board. The use of this version of Linux on the ML403 or on the XUP development board is straight forward. The main difference between the XUP board and the ML403 board is the need to apply a patch to the Linux source tree in order to solve problems involving certain versions of the PowerPC embedded in the Virtex-4 FPGA. MontaVista Linux can be used with the ML403 or the XUP board by following the steps listed below (the steps are described in detail in [39]).

1. Create and synthesize a hardware design using the Xilinx EDK [15].
2. Set up the cross-compiler toolchain using Cross Tool [47].
3. Download a copy of the MontaVista Linux 2.4 [40] development three.
4. Patch the Linux source tree using the patch supplied as part of the ML403 PowerPC reference design [48]. (No patch needed for the XUP board)
5. Generate the hardware specific driver files using Xilinx EDK.
6. Configure and compile the Linux kernel using the cross-compile environment.
7. Configure the FPGA.
8. Upload and start the Linux kernel on the board.

Instruction-cache
`icbi` (Instruction Cache Block Invalidate)
`icbt` (Instruction Cache Block Touch)
`iccci` (Instruction Cache Congruence Class Invalidate)
`icread` (Instruction Cache Read)
Data-cache
`dcba` (Data Cache Block Allocate)
`dcbf` (Data Cache Block Flush)
`dcbi` (Data Cache Block Invalidate)
`dcbst` (Data Cache Block Store)
`dcbt` (Data Cache Block Touch)
`dcbtst` (Data Cache Block Touch for Store)
`dcbz` (Data Cache Block Clear to Zero)
`dccci` (Data Cache Congruence Class Invalidate)
`dcread` (Data Cache Read)

Table 6.1: Instruction- and Data-cache related instructions

6.2 Linux on the ML403 board with OCM memory system

Using MontaVista Linux as discussed in Section 6.1 on an OCM based system is, in theory, straight forward. In this section, we will discuss the theoretical changes to the Linux kernel; next, we will discuss the problems encountered while applying these changes and we will present the consequences to these problems.

Since the OCM bus provides non-cachable access to both instruction-side and data-side memory [3], all cache related behavior in the Linux kernel should be taken out. The cache related instructions are well specified in [7] and are only used in the parts of the Linux kernel source code which are written in assembly. Therefore, these instructions are only used in the architecture dependent part of the kernel source three.

Cache instructions can be separated into two groups, instructions which are related to the data-cache and instructions related to instruction-cache. Table 6.1 lists the data-cache and instruction-cache related instructions. The use of one of the cache storage instructions listed in Table 6.1 on a non-cacheable memory area (OCM memory) leads to a instruction- or data-storage interrupt (except `dcbt` and `dcbtst`, which result in no-operation).

In order to identify the source of the interrupt, we developed a disassembler based on the Xilinx Microprocessor Debugger (XMD). XMD is included in the Xilinx EDK toolset, and can be utilized to examine the internal state of the processor. A description of the XMD disassembler can be found in Appendix B.

Once the cache related instructions where stripped from the Linux kernel, the boot process of the Kernel halted as soon as the `start_kernel()` function was called. The jump to the `start_kernel()` function is the first use of the virtual address range assigned to the Linux kernel. The kernel halted by raising a machine check interrupt, which is caused by an invalid address translation.

Physical address (OCM)	Virtual address	Behavior
0x00000000 - 0x00FFFFFF	0x00000000 - 0x00FFFFFF	Normal
0x00000000 - 0x00FFFFFF	0xC0000000 - 0xC0FFFFFF	Machine check exception
0xC0000000 - 0xC0FFFFFF	0x00000000 - 0x00FFFFFF	Machine check exception
0xC0000000 - 0xC0FFFFFF	0xC0000000 - 0xC0FFFFFF	Normal

Table 6.2: Address translation behavior.

Since the kernel halted at the first use of the virtual address space, we decided to run some simple tests involving the use of virtual-to-physical address translation. Table 6.2 lists the tested physical and virtual address combinations and the observed PowerPCs behavior. The results of these tests lead us to suspect limitations on the use of virtual-to-physical address translation in combination with the OCM buses. In [5, 4, 12, 42, 3, 7] no limitations to address translation related to the OCM buses are mentioned. Figure 6.1 suggests that every memory access is handled by the MMU and therefore every memory access is translated. Because our test results indicated a

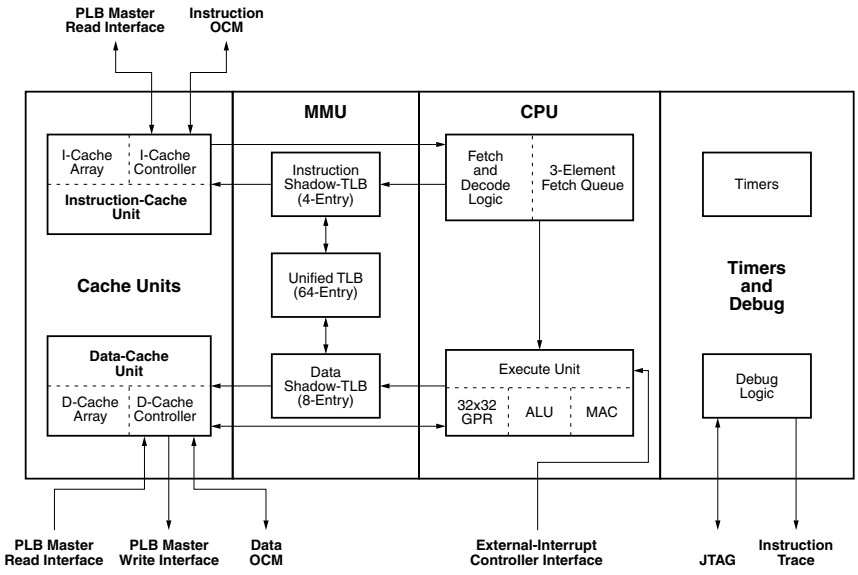


Figure 6.1: Internal PPC405 organization [3].

mismatch between documentation and physical implementation, we contacted the Xilinx Corporation [9] which confirmed these limitations. Summarizing, this limitation implies that when using the OCM buses to connect memory, we can only use virtual memory as long as the virtual address is the same as the physical address, which is conceptually the same as using a system without MMU. Therefore porting MontaVista Linux to a system based on an OCM memory system is not possible.

6.3 Conclusion

In this chapter, we described steps to execute the MontaVista Linux on a reference system with the main memory connected to the OCM buses. While debugging the Linux kernel with the OCM memory system in place, we observed problems with the virtual-to-physical address translation. These problems lead us to suspect limitations on the virtual-to-physical address translation, if used in combination with an OCM bus based memory system. No such limitations were found in documentation. Therefore, we verified the existence of these limitations by contacting the Xilinx Corporation. Due to these limitations, the functionality the PowerPCs memory management in combination with an OCM memory system is comparable to a processor with no memory management at all. Porting the Linux kernel to a system without virtual memory, imposes large restrictions on its capabilities. Without the use of a MMU, the Linux kernel will not support functionalities like dynamic memory allocation, dynamic linked libraries, dynamic stack growth, memory access protection, etc. Therefore the option chosen in Chapter 3 is a good solution for applications which do not require the hardware memory management system (MMU) of the PowerPC, but introduces large limitations if an application demands the MMU.

7 Recommendations

In Chapter 3 we presented four options to expand the memory size of the Molen prototype, we concluded by choosing the shared OCM DDR controller as the best option. We executed the MontaVista Linux 2.4 kernel on a system with OCM based memory. Results of this attempt are presented in Chapter 6. These results lead to the conclusion that the design, as proposed in "Molen with ISOCM DDR and DSOCM DDR memory" (Section 3.3) is not suitable if applications require a MMU. The proposed memory system is, therefore not suitable to execute an OS with full functionality.

In this chapter we will elaborate on the "Molen based on a PLB only bus structure" option as presented in Section 3.2. This option introduces a minimum deviation from the contemporary way of connecting external modules to the PowerPC and therefore, enables the execution of the Linux kernel on the Molen prototype.

7.1 A PLB based design for Operating System support

In this section, we will present the design considerations of the PLB to CCU controller and the CCU data memory interface as discussed in Chapter 3. Apart from being utilized in the "Molen based on a PLB only bus structure", the PLB to CCU controller and the CCU data memory interface can also be utilized in "Molen implemented using Virtex-4 Auxiliary Processing Unit (APU)" (Section 3.4).

7.1.1 General design

To provide a common interface to the PowerPC for all CCU related connections, we propose to integrate the connection to the XREGs in the PLB CCU controller. In the current Molen prototype, the XREGs are addressed using the DCR bus, which does not use the common read/write instructions, but uses specific DCR instructions which are only available in the private mode of the PowerPC [7]. Therefore, transferring data to the XREGs outside of the Linux kernel space (which is the common case), has to be handled by a specific kernel device driver. If the connection to the XREG is included in the PLB CCU controller, the XREG can be assigned their own address space and can also be remapped using the hardware MMU.

The current CCU design does not support data memory with a variable latency to be connected, therefore, the interface between the CCU and data memory should be extended with a facility to support variable latency read/write operations. To preserve as much backwards compatibility as possible, we propose to use the current CCU to BRAM connection as basis for the PLB to CCU connection and expand this connection with a *read/write complete* signal to inform the CCU of the completion of a PLB operation.

7.1.2 Detailed design

For rapid development, Xilinx provides the IPIF (detailed in Chapter 2). The IPIF can also be utilized to implement a PLB CCU controller (Figure 7.1 depicts its detailed organization.) Xilinx provides a generic PLB IPIF controller [49], which is both PLB

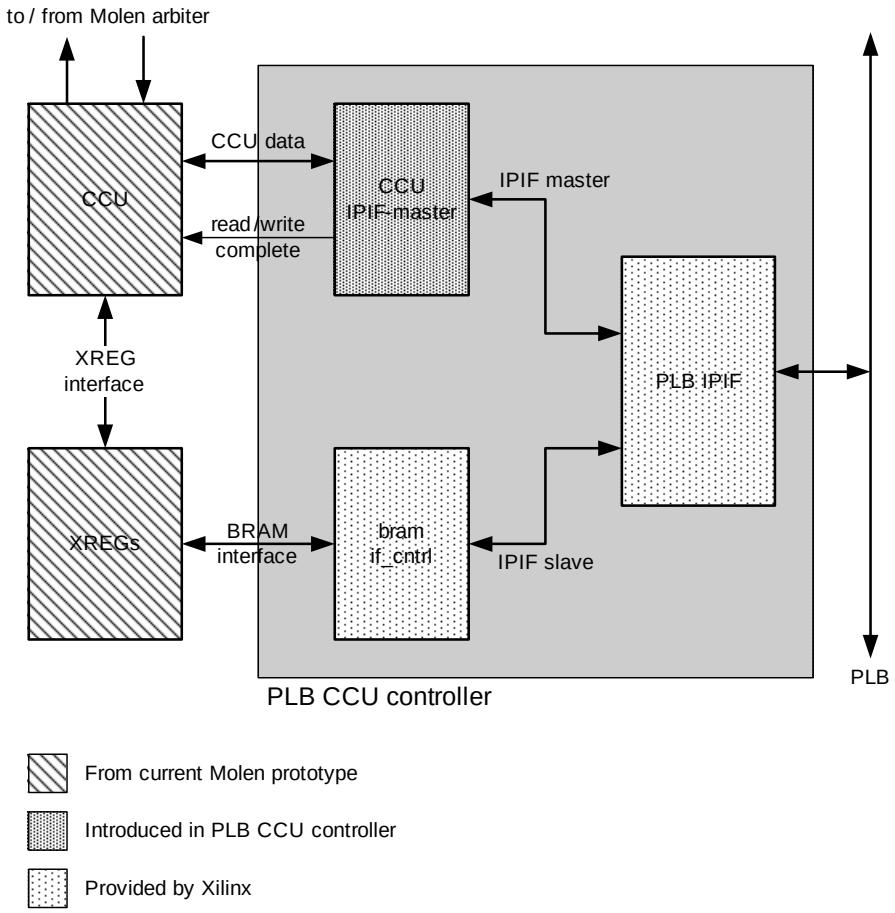


Figure 7.1: Detailed PLB CCU controller design using the IPIF.

master and slave capable. The IPIF slave signals are compatible to the `bram_if_cntrl` interface, and can therefore directly be connected to the BRAM blocks containing the XREG register file. The IPIF master signals are connected to the CCU IPIF-Master controller. The CCU IPIF-Master controller translates read/write requests from the CCU to an IPIF master request. Moreover, the controller should also signal the CCU by asserting the read/write complete signal once an operation is completed.

7.2 Conclusion

In this chapter, we introduced a detailed design of a PLB CCU controller. In spite of a decrease in performance compared to the shared OCM DDR controller presented in Chapter 4, the use of a PLB based memory system for the Molen prototype allows utilizing the capabilities of the hardware MMU incorporated in the PowerPC. Using the MMU, an OS can be used without any adaptations. Moreover, accessing the CCU from within the operating system can be done using memory mapped IO. This even allows virtual remapping of the XREGs address space.

If the Molen prototype is utilized to execute a standalone application, a memory organization based on the shared OCM DDR controller (as presented in Chapter 4) will be the most advantageous. If the Molen prototype is utilized to execute applications requiring virtual memory, like an OS, the memory organization based on the PLB CCU controller (as introduced in this chapter) is recommended.

8 Conclusion

In this thesis, we argued that expanding the memory size of the current Molen prototype is inevitable, if the prototype is to be applied in a wider range of applications. We examined different options to expand the memory size and adapted the, at the time, best solution. Moreover, we proved that our solution introduces a performance benefit compared to the PLB based design provided by Xilinx. Our attempts to execute the Linux kernel on the design were presented. The attempts, revealed limitations on virtual-to-physical address translation which were undocumented until now. Finally, we recommend and specify a solution which enables the execution of the Linux OS on the Molen prototype without the interference of the limitations mentioned earlier.

In this section, we will summarize our conclusions, we emphasize the main contributions introduced by our research, and finally, we will present directions towards future research.

8.1 Summary

In Chapter 2, we introduced the background on the systems and concepts used throughout this thesis. Firstly, we gave the description of two Xilinx development boards and their FPGAs. Secondly, we explained the internal organization of IBMs PowerPC, which is embedded in the FPGAs utilized. Thirdly, we introduced the Xilinx IPIF, this interface allows the easy interconnect of intellectual property. Fourthly, we detailed the Molen machine organization and the Molen prototype which utilizes this organization. Finally, we referred to work related to topics discussed in this thesis. We refer to other reconfigurable architectures and how they differ from the Molen machine organization. We also referred to techniques which are used to increase performance and techniques for memory arbitration. In our design, we utilize the OCM buses to increase performance, because we use both OCM buses, the arbitration of memory requests is required. Previous research shows that the utilization of the OCM buses can introduce performance improvement. Furthermore, we introduced work on the use of the Linux OS as an embedded operating system, which we will use in combination with our design.

In Chapter 3, we proposed four options for expanding the memory size of the Molen prototype. These options are based upon a combination of hardware and software changes which utilize external DDR memory. The first two options utilize an existing PLB DDR controller supplied by Xilinx. The third option introduces a DDR controller utilizing the OCM buses of the PowerPC. The forth option utilizes the APU incorporated in the embedded PowerPC of the Virtex-4 FPGAs. This APU is used in combination with either the existing PLB DDR controller or the newly introduced shared OCM DDR controller.

In Section 3.5, we gave an overview of the advantages and disadvantages of each

proposed solution. We also considered the impact of each option on the performance of the Molen prototype, the re-usability of existing CCU designs and the changes to the Molen arbiter. Finally, we motivated the choice for the development of a shared OCM DDR controller.

In Chapter 4, we presented the design of three DDR controllers using the OCM buses of the PowerPC embedded in the Virtex-4 chip. We first presented general issues relating the development of DDR controllers and introduce the advantage of using the IPIF utilized by Xilinx IP. Secondly, we have presented the design of a DSOCM DDR controller, which can be used separately if needed. Thirdly, we introduced a mechanism to overcome limitations introduced by the ISOCM bus implemented on the PowerPC's embedded in the Virtex-series. This mechanism, based on the scheme utilized in the Molen arbiter, allows us to keep the PowerPC in the same state while waiting for DDR memory, to provide the next valid instruction. Finally, we presented a shared OCM DDR controller which utilizes the Xilinx IPIF internally. We introduced an IPIF arbiter to determine which OCM request should be serviced.

The designs presented in Chapter 4 can be utilized to increase the memory size of embedded systems using the OCM buses. Moreover, the shared OCM DDR controller design can easily be extended to incorporate an interface to the CCU which is part of the Molen prototype.

In Chapter 5, we presented the simulation and verification of the ISOCM DDR, DSOCM DDR and shared OCM DDR controllers. We showed that the functionality of these controllers corresponds to the expected functionality using the Modelsim simulator [43]. Moreover, we verified the correct implementation of the controllers using the Xilinx Chipscope [45] logic state analyzer. We also presented the performance evaluation of the shared OCM DDR controller implemented and tested on the Virtex-4 FX chip. We achieved an improvement from 6% (best-case) to 13% (worst-case) over a PLB based design. The best-case scenario is based on a reference system which only connects memory utilizing the PLB. The worst-case scenario is based on a reference system with memory and other peripherals connected to the PLB. In this scenario, a request is constantly sent if the PLB is available.

The resources used to implement the ISOCM DDR and DSOCM DDR controller are less than with the PLB DDR controller. Furthermore, the resources used to implement the shared OCM DDR controller are comparable to the PLB DDR controller. In our design, arbitration is included in the resource utilization, while for the PLB design arbitration is excluded.

In Chapter 6, we described steps to execute the MontaVista Linux on a reference system with the main memory connected to the OCM buses. While debugging the Linux kernel with the OCM memory system in place, we observed problems with the virtual-to-physical address translation. These problems lead us to suspect limitations on the virtual-to-physical address translation, if used in combination with an OCM bus based memory system. No such limitations were found in documentation. Therefore, we verified the existence of these limitations by contacting the Xilinx Corporation. Due to these limitations, the functionality the PowerPCs memory management in combination with an OCM memory system is comparable to a processor with no memory management at all. Porting the Linux kernel to a system without virtual memory, imposes

large restrictions on its capabilities. Without the use of a MMU, the Linux kernel will not support functionalities like dynamic memory allocation, dynamic linked libraries, dynamic stack growth, memory access protection, etc. Therefore the option chosen in Chapter 3 is a good solution for applications which do not require the hardware memory management system (MMU) of the PowerPC, but introduces large limitations if an application demands the MMU.

In Chapter 7, we introduced a detailed design of a PLB CCU controller. In spite of a decrease in performance compared to the shared OCM DDR controller presented in Chapter 4, the use of a PLB based memory system for the Molen prototype allows utilizing the capabilities of the hardware MMU incorporated in the PowerPC. Using the MMU, an OS can be used without any adaptations. Moreover, accessing the CCU from within the operating system can be done using memory mapped IO. This even allows virtual remapping of the XREGs address space.

If the Molen prototype is utilized to execute a standalone application, a memory organization based on the shared OCM DDR controller (as presented in Chapter 4) will be the most advantageous. If the Molen prototype is utilized to execute applications requiring virtual memory, like an OS, the memory organization based on the PLB CCU controller (as introduced in this chapter) is recommended.

8.2 Main contributions

In this section, we list the most important contributions of our research.

- We have introduced an external memory controller which allows external DDR memory to be connected to the ISOCM bus. The ISOCM IPIF module of this controller can be utilized to connect other types of memory or devices with a variable latency.

- We have introduced an external shared data and instruction DDR memory controller which combines the ideas presented in [34] with the ISOCM DDR controller into a shared OCM DDR controller. We have shown that the shared OCM DDR controller introduces an improvement of 6% to 13% over a PLB DDR controller based design. The development of this controller will be published in August 2007 [41].

- We have introduced an IPIF arbiter, which allows arbitration of various sources simultaneously requesting acces to one target. The arbiter is internally based upon the Xilinx IPIF.

- We have presented the results an attempt to run the Linux OS on an OCM based memory system. These attempts resulted in clear knowledge on, thus far undocumented, limitations of using the OCM buses in combination with the PowerPCs MMU.

8.3 Future Work

In this section, we present directions for future research. The directions listed below are originated from the idea to execute the Linux OS on an architecture based on the Molen prototype.

- **Extend the CCU data memory interface to support variable latency memory operations.**
  The data memory interface of the CCU relies on the latency of the BRAM blocks of the Virtex FPGAs. If the memory required by the CCU is larger than the available on-chip BRAM blocks, the latency of the memory will not match the latency of the on-chip BRAM blocks. Therefore, we propose to extend the CCU data interface with a handshaking facility.

- **Extend the shared OCM DDR controller with a CCU IPIF interface.**
  The design of the shared OCM DDR controller which is introduced in this thesis does not include an interface to the CCU. Therefore, we propose to develop a CCU IPIF controller and integrate this controller with the shared OCM DDR controller. Note, that for the CCU to interface with external DDR, this interface should support handshaking.

- **Port the Molen arbiter to the IPLB.**
  By porting the Molen arbiter to the IPLB, it will be possible to arbitrate instructions which are stored on the PLB addressable range of the PowerPC. Therefore, we propose to port the Molen arbiter to be able to arbitrate IPLB requests.

- **Develop and implement a CCU IPIF-master controller and integrate this controller into a PLB CCU controller.**
  In the current Molen prototype, the CCU interfaces with on-chip BRAM blocks only. If a controller is introduced with enables the CCU to interface with the PLB, the CCU will be able to request data from every location accessible through the PLB by means of a DMA transfer. This even allows the CCU to directly fetch data originating in a different peripheral. Therefore, we propose to develop a CCU IPIF-master controller and integrate this controller into a PLB CCU controller as recommended in Chapter 7. Note, that for the CCU to interface with the PLB, this interface should support handshaking.

- **Develop a PLB CCU device driver and test MontaVista Linux on a PLB CCU based design.**
  Once the PLB CCU controller in realized, the Molen prototype can be used for a wide range of applications. The Molen prototype can be used as base system to execute the Linux OS. For the Linux OS to exploit the advantages introduced by the Molen machine organization, we propose to develop a kernel device driver. This driver should be able to handle the CCUs start/stop operations, the transfer of argument data, the reconfiguration and other management functions of the CCU.

- **Map the Molen design into the Virtex-4 APU.**
  The organization of the APU embedded in the Virtex-4 is very similar to the

Molen machine organization. Therefore we propose to map the Molen machine organization to the APU embedded in the Virtex-4 FPGAs. The Molen arbiter should be mapped to the FCU decode unit, the current reconfigurable instructions should be mapped to the custom instructions supported by the decode unit and a FCU CCU wrapper should be introduced. The direct CCU data memory interface should be adjusted to interface with the PLB bus, or CCU development should omit this interface and use the data fetch instructions supported by the APU.

Bibliography

[1] "Xilinx University Program Virtex-II Pro Development System," Xilinx Corporation, March 2005.

[2] "ML401/ML402/ML403 Evaluation Platform v2.5," Xilinx Corporation, 2006.

[3] "PowerPC 405 Processor Block Reference Guide v2.1," Xilinx Corporation, 2005.

[4] "Instruction-side OCM bus," Xilinx Corporation.

[5] "Data-side OCM bus," Xilinx Corporation.

[6] "Processor Local Bus (PLB) Arbiter design specification," Xilinx Corporation.

[7] "PowerPC Processor Reference Guide," Xilinx Corporation, September 2003.

[8] G. K. Kuzmanov, "The Molen Polymorphic Media Processor," Ph.D. dissertation, TU Delft, December 2004.

[9] "Xilinx Corporation." [Online]. Available: http://www.xilinx.com

[10] R. H. Katz and G. Borriello, *Contemporary Logic Design*, 2nd ed. Pearson Prentice Hall, 2005, no. ISBN 0-201-30857-6.

[11] "Virtex-II Pro and Virtex-II Pro X Platform FPGAs:Complete Data Sheet," Xilinx Corporation, October 2005.

[12] "Virtex-4 Family Overview v1.6," Xilinx Corporation, October 2006.

[13] "IBM CoreConnect Bus Cores," IBM, g224-7587-01.

[14] "Processor Local Bus (PLB) v3.4," Xilinx Corporation.

[15] "Embedded Development Kit (EDK)," Xilinx Corporation. [Online]. Available: http://www.xilinx.com/edk/

[16] "PLB Double Data Rate (DDR) Synchronous DRAM (SDRAM) Controller," Xilinx Corporation, December 2005.

[17] E. M. Panainte, K. Bertels, and S. Vassiliadis, "The PowerPC Backend Molen Compiler," in *14th International Conference on Field-Programmable Logic and Applications (FPL)*, September 2004, pp. 434–443.

[18] G. Kuzmanov and S. Vassiliadis, "Arbitrating Instructions in an $\rho\mu$-coded CCM," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, September 2003, pp. 81–90.

[19] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[20] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53, pp. 1363– 1375, November 2004.

[21] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The Reconfigurable Streaming Vector Processor (RSVPTM)," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium*

on Microarchitecture. Washington, DC, USA: IEEE Computer Society, 2003, p. 141.

[22] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 1997, p. 12.

[23] V. M. G. Ferreira, L. Gauthier, T. Kando, T. Matsuo, T. Hashinaga, and K. Murakami, "REDEFIS: a system with a redefinable instruction set processor," in *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*. New York, NY, USA: ACM Press, 2006, pp. 14–19.

[24] S. Kimura, Y. Itou, M. Hirao, K. Watanabe, M. Yukishita, and A. Nagoya, "A Hardware/Software Codesign Method for a General Purpose Reconfigurable Co-Processor," in *CODES '97: Proceedings of the 5th International Workshop on Hardware/Software Co-Design*. Washington, DC, USA: IEEE Computer Society, 1997, p. 147.

[25] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture," in *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 1998, p. 28.

[26] J. A. Jacob and P. Chow, "Memory Interfacing and Instruction Specification for Reconfigurable Processors," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 1999, pp. 145–154.

[27] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 2000, pp. 128–138.

[28] W.-F. Lin, S. K. Reinhardt, and D. Burger, "Designing a Modern Memory Hierarchy with Hardware Prefetching," *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1202–1218, 2001.

[29] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a Smarter Memory Controller," in *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 1999, p. 70.

[30] B. Jacob, "A Case for Studying DRAM Issues at the System Level," *IEEE Micro*, vol. 23, no. 4, pp. 44–56, 2003.

[31] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[32] S. Prybylski, M. Horowitz, and J. Hennessy, "Performance Tradeoffs in Cache Design," in *ISCA '88: Proceedings of the 15th Annual International Symposium*

on Computer architecture. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 290–298.

[33] K. Lund, "PLB vs. OCM Comparison Using the Packet Processor Software," Xilinx Corporation, July 2002, xAPP644.

[34] B. Donchev, G. Kuzmanov, and G. N. Gaydadjiev, "External Memory Controller for Virtex II Pro," in *Proceedings of International Symposium on System-on-Chip 2006*, November 2006, pp. 37–40.

[35] I. Ouaiss and R. Vemuri, "Efficient Resource Arbitration in Reconfigurable Computing Environments," in *DATE '00: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM Press, 2000, pp. 560–566.

[36] E. Vlachos, "Design and Implementation of a Coherent Memory Sub-System for Shared Memory Multiprocessors," Master's thesis, Computer Science Department, University of Crete, July 2006.

[37] "The Linux Operating System." [Online]. Available: http://www.linux.org

[38] "Brigham Young University Linux on FPGA Project." [Online]. Available: http://splish.ee.byu.edu/projects/LinuxFPGA/

[39] J. W. Donaldson, "Porting MontaVista Linux to the XUP Virtex-II Pro Development Board," August 2006.

[40] "Monta Vista Linux." [Online]. Available: http://www.mvista.com

[41] B. Breijer, F. Duarte, and S. Wong, "An OCM based shared Memory controller for Virtex 4," in *17th International Conference on Field-Programmable Logic and Applications (FPL)*, to appear in August 2007.

[42] "PPC405Fx Embedded Processor Core - Users Manual," IBM, January 2005, sA14-2764-00.

[43] "Modelsim," Mentor Grapics. [Online]. Available: http://www.model.com/

[44] "Virtex-4 ML403 Embedded Platform," Xilinx Corporation. [Online]. Available: http://www.xilinx.com/ml403/

[45] "Chipscope Pro Logic State Analyzer," Xilinx Corporation. [Online]. Available: http://www.xilinx.com/chipscope/

[46] "Memtest86." [Online]. Available: http://www.memtest86.com/

[47] D. Kegel, "Crosstool toolchain." [Online]. Available: http://kegel.com/crosstool/

[48] "ML403 PowerPC Reference design," Xilinx Corporation. [Online]. Available: http://www.xilinx.com/products/boards/ml403/files/ml403_emb_ref_ppc_81.zip

[49] "PLB IPIF," Xilinx Corporation.

[50] "Embedded System Tools Reference Manual," Xilinx Corporation.

List of Acronyms

ACE Advanced Configuration Environment

APU Auxiliary Processing Unit

ASIC Application Specific Integrated Circuit

BRAM Block Random Access Memory

BSP Board Support Package

CCU Custom Computing Unit

CLB Configurable Logic Block

CPLD Complex Programmable Logic Device

DCR Device Control Register

DCU Data Cache Unit

DDR Double Data Rate

DMA Direct Memory Access

DPLB Data-side Processor Local Bus

DSOCM Data-Side On-Chip Memory

EDK Embedded Development Kit [15]

EVPR Exception Vector-Prefix Register

FCU Fabric Co-processor Unit

FF Flip-Flop

FPGA Field Programmable Gate Array

FSM Finite State Machine

GPP General Purpose Processor

HDL Hardware Description Language

IBM International Business Machines Corporation

ICU Instruction Cache Unit

IO Input/Output

IOB Input/Output Block

IP Intellectual Property

IPIF Intellectual Property InterFace

IPLB Instruction-Side Processor Local Bus

ISOCM Instruction-Side On-Chip Memory

ISR Interrupt Service Routine

JTAG Joint Test Action Group

LR Link Register

LUT Look-Up Table

MMU Memory Management Unit

NAPA National Adaptive Processing Architecture

OCM On-Chip Memory

OPB On-chip Peripheral Bus

OS Operating System

PID Process IDentification

PHY Physical layer protocol

PLB Processor Local Bus

RAM Random Access Memory

RISC Reduced Instruction Set Computer

ROM Read Only Memory

REDEFIS REDEFinable Instruction Set

RSVP Reconfigurable Streaming Vector Processor

SDRAM Synchronous Dynamic Random Access Memory

SRR Save and Restore Register

TID Translation IDentification

TLB Translation Look-Aside Buffer

UART Universal Asynchronous Receiver-Transmitter

USB Universal Serial Bus

VHDL Very high speed integrated Hardware Description Language

XMD Xilinx Microprocessor Debugger

XST Xilinx Synthesis Technology

XREG eXchange REGister

XUP Xilinx University Program

Benchmark program

A

```
#include "xtime_l.h"
#include "xexception_l.h"
#include "xcache_l.h"
#include "xparameters.h"
#include "xutil.h"

#define ISOCM

#define KILO (256)
#define MEG  (1024*KILO)

static int a_size[] = {128*KILO, 256*KILO, 512*KILO, MEG, 2*MEG, 4*
    MEG, 8*MEG, 15*MEG};
#define SIZES (sizeof(a_size)/sizeof(int))

#ifdef XPAR_OCM_DDR_0_IS_HIGHADDR
 #ifdef XPAR_PLB_PAYLOAD_0_BASEADDR
  #define DESCRIPTION "OCM DDR w PAYLOAD"
 #else
  #define DESCRIPTION "OCM DDR"
 #endif
#else
 #ifdef XPAR_PLB_PAYLOAD_0_BASEADDR
  #define DESCRIPTION "PLB DDR w PAYLOAD"
 #else
  #define DESCRIPTION "PLB DDR"
 #endif
#endif

#define NUM  (10)
#define RUNS (10)

#define BASE_ADDRESS  0x0 // MAX 0xFFFF, must be multiple of 4

#define PROG_START_ADDR 0x00FF0000

int (*func_ptr) ();

#ifdef ISOCM
void store_program(unsigned int *base, int size, int num){

  /* Store ASM program to *base.
```

```
42      sets r10 to 0
43      Assumes arr is located: 8(r10)
44            Assumes p_arr is located: 12(r10)
45            Assumes j is located: 0(r10)
46            Assumes k is located: 4(r10)
47            Assumes DDR is located at 0x0000
48            Branches are relative.
49      for (j = 0; j < NUM; j++){
50        p_arr = &arr[0];
51        for (k = 0; k < SIZE; k++){
52          *p_arr = k+j;
53          arr[k] = *p_arr;
54          p_arr++;
55        }
56      }
57  */
58
59  /* set r10 = 0 */
60  base[0] = 0x39400000 | BASE_ADDRESS; // li r10,0
61  /* store arr =  12(r10)*/
62  base[1] = 0x38000010; // li r0,16
63  base[2] = 0x900A0008; // stw r0,8(r10)
64  /* j=0 and check */
65  base[3] = 0x38000000; // li r0,0
66  base[4] = 0x900A0000; // stw r0,0(r10)
67  base[5] = 0x800A0000; // lwz r0,0(r10)
68  base[6] = 0x2F800000 | (num-1); // cmpwi cr7,r0,1 (loop bound)
69  base[7] = 0x419D0084; // bgt-  cr7, +132 <0x84>
70
71  /* loop increment: j++ */
72  base[36] = 0x812A0000; // lwz r9,0(r10)
73  base[37] = 0x38090001; // addi r0,r9,1
74  base[38] = 0x900A0000; // stw r0,0(r10)
75  base[39] = 0x4BFFFF78; // b -136 <FFFFFF78>
76
77  /* p_arr = &arr[0];*/
78  base[8] = 0x800A0008; // lwz r0,8(r10)
79  base[9] = 0x900A000C; // lwz r0,12(r10)
80  /* k = 0 and check */
81  base[10] = 0x38000000; // li r0,0
82  base[11] = 0x900A0004; // stw r0,4(r10)
83  base[12] = 0x800A0004; // lwz r0,4(r10)
84  base[13] = 0x3D200000 | ((size-1) >> 16); // lis r9,0xF (size-1
        upper)
85  base[14] = 0x61290000 | ((size-1) & 0xFFFF); // ori r9,r9, 0xFFFF (
        size-1 lower)
86  base[15] = 0x7F804800; // cmpwi cr7,r0,r9 (loop bound)
87
88  base[16] = 0x419D0050; // bgt-  cr7, +80 <0x50>
89  /* loop increment: k++ */
90  base[32] = 0x812A0004; // lwz r9,4(r10)
```

```
91    base[33] = 0x38090001; // addi r0,r9,1
92    base[34] = 0x900A0004; // stw r0,4(r10)
93    base[35] = 0x4BFFFFA4; // b -92 <FFFFFFA4>
94
95    /* *p_arr = k + j*/
96    base[17] = 0x816A000C; // lwz r11,12(r10)
97    base[18] = 0x812A0004; // lwz r9,4(r10)
98    base[19] = 0x800A0000; // lwz r0,0(r10)
99    base[20] = 0x7C090214; // add r0,r9,r0
100   base[21] = 0x900B0000; // stw r0,0(r11)
101   /* arr[k] = *p_arr */
102   base[22] = 0x800A0004; // lwz r0,4(r10)
103   base[23] = 0x5409103A; // rlwinm r9,r0,2,0,29
104   base[24] = 0x800A0008; // lwz r0,8(r10)
105   base[25] = 0x7D690214; // add r11,r9,r0
106   base[26] = 0x812A000C; // lwz r9,12(r10)
107   base[27] = 0x80090000; // lwz r0,0(r9)
108   base[28] = 0x900B0000; // stw r0,0(r11)
109   /* p_arr++ */
110   base[29] = 0x812A000C; // lwz r9,12(r10)
111   base[30] = 0x38090004; // addi r0,r9,4
112   base[31] = 0x900A000C; // lwz r0,12(r10)
113
114   /* add a blr to be able to call the program as a function */
115   base[40] = 0x4E800020; // blr
116   return;
117 }
118 #endif
119
120 int main(int argv, char **argc){
121   int i;
122   int j = 0;
123   int size;
124   unsigned long long *start = (unsigned long long *)(PROG_START_ADDR)
        ;
125   unsigned long long *end = (unsigned long long *)(PROG_START_ADDR+
        sizeof(unsigned long long));
126   unsigned long long diff, avg = 0;
127
128   xil_printf("\%s start.\r\n",DESCRIPTION);
129   for (j = 0; j < SIZES; j++){
130     size = a_size[j];
131 #ifdef ISOCM
132     store_program((int*)(PROG_START_ADDR+2*sizeof(unsigned long long)
          ), size, NUM);
133 #endif
134     func_ptr = (void *)(PROG_START_ADDR+2*sizeof(unsigned long long))
          ;
135
136     for (i = 0; i < RUNS; i++){
137           int l,k;
```

```
138                     int *p_arr;
139                   int *arr = (int*)(PROG_START_ADDR+2*sizeof(unsigned
                        long long));
140         XTime_SetTime(0);
141         XTime_GetTime(start);
142
143 #ifdef ISOCM
144                   func_ptr();
145 #else
146       for (l = 0; l < NUM; l++){
147         p_arr = &arr[0];
148         for (k = 0; k < size; k++){
149           *p_arr = k+l;
150           arr[k] = *p_arr;
151           p_arr++;
152         }
153       }
154 #endif
155
156
157         XTime_GetTime(end);
158         diff = *end - *start;
159               avg += diff;
160       }
161       avg /= RUNS;
162       // Print value hexadecimal because xil_printf does not support
            printing of unsigned long long.
163       xil_printf("\%s: bytes: \%d avg: \%08x\%08x\r\n",DESCRIPTION,
            size*sizeof(int), *((int *)&avg), *(((int *)&avg)+1));
164     }
165     xil_printf("\%s end.\r\n\r\n",DESCRIPTION);
166
167     return 0;
168 }
```

Listing A.1: Benchmark program C-code

XMD Disassembler

B

The Xilinx Microprocessor Debugger (XMD) is distributed as part of the Embedded Development Kit [15] (EDK). XMD allows us to connect to the PowerPC using the Joint Test Action Group (JTAG) interface. Once the connection is created, we can start/stop execution, read registers and memory locations. A detailed description of the functionality of XMD can be found in [50].

The XMD debugger allows the definition of custom commands using the Tcl script language. The custom commands can be placed in the `.xmdrc` file in the `$HOME` directory. Commands must be defined as procedure and may also be defined to accept parameters.

Section B.1 discusses the external program used by the XMD disassember, Section B.2 discusses the definition of the Tcl script and shows an example of how to use the XMD disassembler.

B.1 Disassembler program

The disassembler is written in C and uses the PowerPC opcodes definition as used by the 2.4.x version of the Linux kernel. Listing B.1 gives the full C-code of the disassembler. The disassembler relies on five files included in the Linux 2.4.x kernel, the files have to be linked together with the C file mentioned earlier. The kernel file locations are:

```
arch/ppc/xmon/ansidecl.h
arch/ppc/xmon/nonstdio.h
arch/ppc/xmon/ppc-dis.c
arch/ppc/xmon/ppc.h
arch/ppc/xmon/ppc-opc.c
```

```
1  #include "ppc.h"
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  extern int print_insn_big_powerpc(FILE *, unsigned long, unsigned);
8
9  void print_usage(char **argc){
10   printf("Usage: %s instruction\r\n", argc[0]);
11 }
12
13 int main (int argv, char** argc)
14 {
15   unsigned long instruction;
```

```
16
17   if (argv < 2){
18     print_usage(argc);
19     exit(0);
20   }
21
22   instruction = strtoul(argc[1],(char **)NULL, 16);
23
24   print_insn_big_powerpc (stdout, instruction, 0);
25   printf("\r\n",(unsigned int)instruction);
26
27   return 0;
28 }
```

Listing B.1: Disassembler C-code

B.2 XMD Tcl script

To include the disassembler in XMD, the function given in Listing B.2 needs to be included in `$HOME/.xmdrc`.

```
1  proc memrd {addr {num 10}} {
2    set max [expr \$num * 4]
3    for {set i 0} {\$i < \$max} {incr i 4} {
4      set data [mrd \$addr]
5      set location [lindex \$data 0]
6      set instr [lindex \$data 1]
7      set asm [exec ppc405dasm \$instr]
8      puts ''\$location \$asm (\$instr)''
9      set addr [expr \$addr + 4]
10   }
11 }
```

Listing B.2: Disassembler Tcl-function

Curriculum Vitae

Sebastiaan Dirk Breijer was born in Roosendaal, The Netherlands on May 3, 1981. He obtained his VWO diploma in 2000 at the K.S.G. de Breul, Zeist. In August 2004, he obtained the Bachelor of Engineering grade by completing the Electronic System Engineering program at the Zeeland University of Professional Education. During this period, he successfully completed internships at the Royal Netherlands Navy and Thales Netherlands B.V. In September 2004, Bas joined the Computer Engineering Department of the Delft University of Technology and will graduate in August 2007 by completing his MSc. thesis titled: *"Memory organization of the Molen prototype"*.

His research interests include embedded systems design, memory organizations and embedded Linux.