Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

Implementation and Performance of Synchronization Methods for Dual-Core Engine Control-Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Christian Stoif
Matrikelnummer 0025375

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: o. Univ.-Prof. Dipl.-Ing. Dr. techn. Herbert Grünbacher
Mitwirkung: Univ.-Ass. Dipl.-Ing. Dr. techn. Martin Schöberl

Wien, 14. 11. 2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

Chapter 2

Background

2.1 Fundamentals

2.1.1 Overview

In this section an overview of the most important terms used in this thesis is given. The depth of the background knowledge presented here should be enough for any reader to gain enough insight into the topic to interpret the results of this thesis.

Most elementary terms come from the field of operation systems and have their roots in single-processor systems. In many cases the terminology was just extended to cover also systems that allow concurrent processing. Nevertheless, *concurrency* itself introduces some new problems that are simply *inexistent* in a system with strictly sequential execution. This section covers the theory vital for this thesis by following this historical development from nonparallel to parallel systems.

2.1.2 Atomicity

A sequence of commands is said to be executed *atomically* if they are executed as if they were a single instruction.

2.1.3 Processes

According to [Sta01] the term *process* is used here as a synonym for a program in execution, that is all its data whether it is part of the program itself or the operating platform that executes that process. The system-dependent part of the process is called the *context* of the process, all the data regarding a process are the *working set* of this process.

Implementation and Performance of Synchronization Methods for Dual-Core Engine Control-Systems

Multiple parallel processing cores are about to conquer embedded systems as well - the roadmaps of leading semiconductor companies certify this: it is not the question of *whether* they are coming but *how* the architectures of the microcontrollers should look in respect to the strict demands in the automotive sector. In this thesis the step from one to multiple cores is presented, establishing coherence and consistency for different types of shared memory by soft- and hardware means. Also support for point-to-point synchronization between the processor cores is realized implementing different methods. Though the theoretical approach using simulations is independent of the number of processing units, the practical examinations focus on the logical first step from single- to dual-core systems, using an FPGA-development board with two hard PowerPC - processor cores. Best- and Worst-case results, together with intensive benchmarking of all synchronization primitives implemented, show the expected superiority of the hardware solutions. It is also shown that dual-ported memory outperforms single-ported memory if the multiple cores use inherent parallelism by locking shared memory more intelligently using a locking-method developed in this thesis. Simple global locking of the whole shared memory alone prevents any parallel access on principle. In the worst case multiple-ported memory degenerates in performance to single-ported memory. However, the conditions that must be fulfilled for this worst case to occur do not seem to be realistic for practical applications in the field.

Implementierung und Performanz von Synchronisationsmethoden für Antriebssoftware auf Zweikernsystemen

Mehrere parallele Prozessorkerne sind auch bei den eingebetteten Systemen im Anmarsch - die Pläne von führenden Halbleiterherstellern bezeugen dies: es ist nicht die Frage *ob* sie kommen sondern vielmehr *wie* die Architekturen der Mikrocontroller unter Berücksichtigung der strengen Anforderungen im Automobilsektor aussehen sollen. In dieser Diplomarbeit wird der Schritt von einem zu mehreren Prozessorkernen unternommen, unter Verwendung von Mechanismen in Software und Hardware wird Kohärenz und Konsistenz von unterschiedlichen Typen von geteiltem Speicher hergestellt. Auch Unterstützung für Punkt-zu-Punkt Synchronisation zwischen den Prozessorkernen wird durch verschiedene implementierte Methoden realisiert. Obwohl der theoretische Ansatz mithilfe von Simulationen unabhängig von der Anzahl der Prozessorkerne ist konzentrieren sich die praktischen Untersuchungen auf den logischen ersten Schritt von einem Ein- zu einem Zweikernsystem, unter Verwendung einer FPGA-Entwicklungsplatine mit zwei PowerPC-Prozessorkernen. Resultate des besten und schlechtesten Falles zeigen, zusammen mit intensiven Vergleichstests aller implementierten Synchronisationsprimitiven, die erwartete Überlegenheit der Hardwarelösungen. Es wird auch gezeigt wie ein Speicher mit zwei Ports einem Speicher mit nur einem Eingang überlegen ist, falls die mehreren Prozessorkerne inhärente Parallelität unter Verwendung einer in dieser Diplomarbeit vorgestellten Methode zur Speicherreservierung intelligenter nutzen. Allein eine simple globale Reservierung des gesamten geteilten Speichers verhindert eine parallele Nutzung grundsätzlich. Im schlimmsten Fall degeneriert die Performanz eines Speichers mit mehreren Ports zu der eines Speichers mit nur einem Eingang, doch die Bedingungen die zu diesem schlimmstmöglichen Fall führen scheinen für Anwendungen in der Praxis nicht realistisch zu sein.

Contents

Contents iv

List of Figures vii

List of Tables ix

1 Introduction 1
1.1 Outline of Thesis . . . 2

2 Background 3
2.1 Fundamentals . . . 3
2.1.1 Overview . . . 3
2.1.2 Atomicity . . . 3
2.1.3 Processes . . . 3
2.1.4 Resources . . . 4
2.1.5 The Memory Hierarchy, Caches . . . 4
2.1.6 Process Interaction . . . 5
2.1.7 Mutual Exclusion on Single-Cores . . . 6
2.1.8 From Multiprogramming to Multiprocessing . . . 6
2.1.9 Synchronization . . . 7
2.1.10 Mutual Exclusion using Locks . . . 8
2.1.11 Semaphores . . . 9
2.1.12 Event Synchronization by Barriers . . . 9
2.1.13 The Parallel Random Access Machine (PRAM) Model . . . 9
2.1.14 Real-Time Systems (RTS) . . . 10
2.1.15 Fault-Tolerant Systems (FTS) . . . 11
2.1.16 Architectures of Multiprocessor Systems . . . 11
2.1.17 Caches . . . 12
2.1.18 Coherence . . . 13
2.1.19 Consistency . . . 13
2.1.20 Scalability . . . 15
2.1.21 Transactional Memory (TM) . . . 15
2.2 Related Work . . . 17

2.2.1 Mutual Exclusion 17
2.2.2 Event Synchronization 20
2.2.3 Transactional Memory (TM) 21
2.2.4 A Glance at Scalable Multiprocessors 22

3 Software Synchronization **25**
3.1 The Hardware Platform 26
3.2 Low-Level Coherence 29
3.3 Coherence by Locking 31
3.4 External Bus-Slaves as Locks 34
3.5 Event Synchronization 36
3.6 Locking Performance 38
3.6.1 Direct Access to the Lock 38
3.6.2 Locking, Best Case 38
3.6.3 Locked Single Access, Best-, Worst- & Average-case 39

4 Hardware Synchronization **43**
4.1 The Problems 43
4.1.1 PLB- and OPB-Lock, no Hardware Support 44
4.1.2 BRAM-Lock, no Hardware-Support 44
4.1.3 PLB- and OPB-Lock, Hardware-Support 44
4.2 The Roots of the Problems 44
4.3 From SW- to HW-Synchronization 46
4.4 The OCM-Access-Controller 47
4.4.1 Features 47
4.4.2 Description and Programming 49
4.4.3 Structure and State Machine 56
4.4.4 Coherence Ensurance - Implicit Locking in Hardware 59
4.4.5 Consistency Ensurance -
Explicit Spin-Free Global Locking by Hardware 65
4.4.6 Event Synchronization - Simple Barrier 65
4.4.7 Event Synchronization - Extended Simple Barriers 67
4.4.8 Event Synchronization - Complex Barriers 68
4.4.9 Precision of Event Synchronization 71
4.5 Worst-Case Performance 72
4.5.1 Direct-Access Performance 72
4.5.2 Prevention of Starvation in the Worst-Case 73
4.5.3 Conclusion 76

5 Benchmarking **77**
5.1 Worst-Case Benchmarks . . . 77
5.2 Non-Worst-Case Benchmarking . . . 79
5.2.1 Benchmark System Configuration . . . 80
5.2.2 The Global Locking Counter . . . 80
5.2.3 Heavy-Load Single Access Benchmark . . . 82
5.2.4 Heavy-Load Paired Access Benchmark . . . 83
5.2.5 Heavy-Load Floating Windows Benchmark . . . 84

6 Conclusion **88**
6.1 Summary . . . 88
6.2 Future Work . . . 90
6.3 Acknowledgement . . . 92

A Digital Flow **93**
A.1 The Xilinx Development Board ML410 and the Xilinx FPGA Virtex-4 FX60 . . . 94
A.2 Hardware Flow . . . 95
A.2.1 Setup of a Multi-core System with Xilinx EDK 9.2i . . . 96
A.2.2 Adding Individual IPs in Xilinx EDK . . . 103
A.2.3 Adding Individual IPs Manually: from EDK to ISE . . . 104
A.3 Software Flow . . . 110
A.3.1 Using the Xilinx Microprocessor Debugger (XMD) . . . 111
A.3.2 Upload of Applications to the PowerPC Cores . . . 113
A.3.3 The Xilinx Software Development Kit (SDK) . . . 113
A.3.4 Single-Core Application-Management by the SDK . . . 117
A.3.5 The Usage of Assembler . . . 119
A.3.6 The Directory-Structure . . . 120

References **122**

List of Figures

2.1 Architectures of multiprocessor systems, simplified 11

3.1 Ideal communication stays on-chip 26
3.2 Architecture of present dual-core system 27
3.3 Dual-ported B-RAM without low-level coherence 29
3.4 Worst-case non-coherent access: no single correct value read 30
3.5 Atomic access by locking 31
3.6 From non-coherent test&set to coherent test&set&test 33
3.7 State machines of both OPB-locks 1 and 2 34
3.8 State machines of both PLB-locks 1 and 2 36
3.9 Dual-core event synchronization with busy-waiting 37
3.10 Single access performance using spinning locks (no values: starvation) 40
3.11 Unfairness in relaying requests for the lock causes starvation: stores to locked lock are ineffective 42

4.1 Broken chain of control over accessing bus-slaves 45
4.2 The OCM-access-controller takes control over the BRAM 46
4.3 The OCM-access-controller delays requests for the BRAM if necessary 47
4.4 Core-side addressing of the OCM-access-controller 49
4.5 Explicit locking, principle 50
4.6 Pseudo-code of explicit software vs. implicit hardware locking . . . 50
4.7 Pseudo-code for global locking 51
4.8 Control- and Status-register of the OCM-access-controller 51
4.9 Concurrent access by disjunctive shared memory windows 52
4.10 Pseudo-code for locking an address-window 54
4.11 Event-synchronization by hardware-barriers 54
4.12 Abstracted schematic of OCM-access-controller v4.0 for multiple cores 56
4.13 Core-side state machine of OCM-access controller v3.16 57
4.14 Core-side state-machine for the beyond-dual-core case (v4.0) 58
4.15 Principle of a fair and efficient race for access 60
4.16 Worst-case race 60
4.17 Fixed priority scheme: slices of an invariable register 61
4.18 Variable Priority Scheme: Slices of a rolling register 62

4.19 Detailed single-access race for the shared memory using core-priorities 63
4.20 Race for the global lock of the OCM-access-controller 64
4.21 Simple Barrier, principle: two cores synchronize temporally 66
4.22 Generation of the transition signal *sb_cond_met* to leave simple barrier 66
4.23 Multiple selective synchronization at extended simple barrier 67
4.24 Multiple cores synchronize by extended simple barrier 68
4.25 Generation of the release-signal for extended simple barriers 69
4.26 Complex Barrier Implementation, abstracted 70
4.27 Temporal overlapping of worst-case store-accesses 73
4.28 Analysis of starvation caused by dual-core worst-case 74

5.1 Performance of single accesses to the BRAM using all methods . . . 78
5.2 All modules available for system benchmark configuration 79
5.3 Benchmarking: single accesses (no contention: only one core active) 82
5.4 Benchmarking: paired accesses (no contention: only one core active) 83
5.5 Benchmarking by moving memory-windows on a collision course . . 85
5.6 Results of benchmarking with floating windows 86

A.1 High-level block diagram of the Xilinx ML410, taken from [Xil07e] . 94
A.2 Virtex-4 FX family, partial overview, taken from [Xil07h] 94
A.3 Picture of the Xilinx ML410, from [Xil07e] 95
A.4 Creating a new project in the EDK 96
A.5 Enabling debugging over JTAG for one core 97
A.6 draft of basic system-architecture 97
A.7 deactivate port-filtering in the EDK 98
A.8 OCM-chain of IPs 98
A.9 ISOCM-chain, core-private 99
A.10 DSOCM connections creating shared memory! 100
A.11 Complete shared-memory system configuration 101
A.12 address-ranges of all IPs in design 102
A.13 The EDK peripheral wizard 102
A.14 Xilinx ISE 104
A.15 Synthesis of *IP ocm_access_cntrl* using XST 106
A.16 Programming FPGA with iMPACT 109
A.17 *xbash*-prompt as universal tool to work with 110
A.18 Connect to dual-core design using XMD 112
A.19 Essential dual-core working environment 114
A.20 Working environment of the Xilinx SDK 115
A.21 References essential to core0- and core1- programs 115
A.22 Important project-properties for core0 and core1 117
A.23 Generation of a linker script in the SDK 118
A.24 Asymmetrical linker scripts for core 0 and core 1 118
A.25 Run-configuration of the SDK (only for single-core applications) . . 119
A.26 The final directory structure evolved over this thesis 121

List of Tables

3.1 Time for direct access to different lock-locations [clock-cycles] . . . 38
3.2 Theoretical *best case* (contention-free) locking results [clock cycles] . 39
3.3 WC single locked accesses - code structure 39
3.4 Spinning and locked single-access performance 40

4.1 Core-Priorities determined by significance of single set bit 61
4.2 Access-performance for single accesses in the dual-core case 72

5.1 Benchmarking: single accesses, exact values 84
5.2 Benchmarking: paired accesses, exact values 84

Chapter 1

Introduction

Parallelism is a rising star in the automotive sector, its advent is inevitable and is happening right now. The road map of leading semiconductor companies shows that multi-core chips will be introduced within the next couple of years. It is almost certain that in the next decade vehicles will be controlled by multi-core microcontrollers following the development of microprocessors in other areas.

The subject for this thesis originates in the growing demand for resources in the automotive sector. Electronic architectures in automobiles face a continuous increase in functionality, and the correspondingly increasing code requires additional memory. To handle the new functions in time, additional computational power is needed too. Single-core solutions are now the standard in automobiles, but due to the stringent environmental conditions the electronics must fulfill, increasing the core frequency of the processor cores is much more limited than in other fields.

In this thesis we gather information on the performance of multi-core architectures that seem usable for microcontroller designs in the automotive field. With the results, not only is judging conventional architectures easier, but it is also possible to interpret the results to suggest new hardware designs. Of course it is clear that the most efficient hardware architecture might not be the cheapest one. But since the issue here concerns safety-critical systems, cost will or should not be the sole criterion when looking at the results. Architectures that are quite common in the commercial field might not yield enough improvement in performance to give a significant advantage over a single-core system. Exploiting inherent parallelism in the current engine control software for single-cores is a promising possibility to improve the computational power. However, the complexity of today's engine control software makes it difficult to give an accurate estimate about the improvement to expect. The change from single- to multi-core is not without pitfalls and requires prudence. With an automobile being a safety-critical system, the testing alone may prove tedious and unveil unexpected problems.

The advent of parallelism is notoriously renowned in the field of computer science and there are enough examples that show how parallelism - in all its undeniable benefit - introduces totally new kinds of problems which, unfortunately, are non-trivial in the majority. However, with ever increasing miniaturization, introducing parallelism is a natural next step in the evolution of any microprocessor architecture (e.g. leading from the UltraSPARCI in 1995 to the dual-core 64b UltraSPARC in 2003 [ea04c]).

1.1 Outline of Thesis

We explain the background including all relevant basic concepts and give a broad overview of research done in the field of multiprocessing in chapter 2.

In chapter 3 the hardware platform used for this thesis is described briefly, then our system is supplemented by *software synchronization*. We then try to improve efficiency by using bus-slaves in hardware and analyze the results.

In the chapter 4 we complement the PowerPC dual-core system at hand with hardware facilities to improve synchronization. The benchmarking results and their interpretation is presented in chapter 5. A conclusion is given afterwards to sum up the results and usefulness of this thesis' results.

The appendix covers the vendor-specific step-by-step guide of the developed and used digital flow both for the hardware and software.

Please note here that none of the ANSI C-, Assembler- nor VHDL-code is printed in this thesis due to size limitations.

2.1.4 Resources

In any computer system there are resources that are needed by processes to execute until their completion. Any operating system acts essentially as a *resource controller*, managing the resources of a given system to guarantee correct operation. The resources that must be managed are

- processor time
- memory
- devices

Also in systems that have only one processor there may be more than one process at a time to use the time the processor must wait for relatively slower operations like input-/output(i/o)-operations carried out by i/o-devices. The method that introduced the sharing of the resources of as system between more than just one process at a time is known as *multiprogramming*. With more than one program in execution on a single processor the complexity of the operating system increases significantly. The operating system must cope now with newly arisen problems (→ 2.1.6), even though the system still has only one processing unit that must be switched between the active processes.

2.1.5 The Memory Hierarchy, Caches

The term *memory hierarchy* comes from the fact that a system normally comprises different types of memory where the fastest memory coincides with the highest costs and thus smallest amount ([Sta01], 1.5). Due to today's high memory requirements there is almost always a high-capacity but slow memory present. The slower the memory accessed the longer the waiting time introduced by *waiting cycles*. To avoid such time-consuming access a small but faster memory called a *cache* is used to buffer read values for further accesses or even buffer writes (*write-back* cache) to the slower memory. Caches improve the average performance of a system significantly. The reason for this is the inherent locality of data due to the structure of the data used in programs (*spatial locality*) and the iterative, step-wise nature of programs (*temporal locality*).

To learn more about locality and the memory-hierarchy see [Sta01].

2.1.6 Process Interaction

Processes that are active at the same time might not be aware of each other. According to [Sta01] this makes them simply competitors for the available resources. Coexistent processes can also be aware of each other, either because they were originally meant to cooperate directly to achieve their goals or because one process depends on the results of another. Due to the new situation coexistent processes create in a system, the operating system faces the following new control problems:

- **Mutual Exclusion** Resources that cannot be used by an arbitrary number of processes at the same time are commonly called *critical resources*, the section in the code of a program that accesses this resource is called a *critical section*. The problem of how to guarantee exclusive access to a resource is a new problem simply inexistent to systems where only one process is allowed to be active.

- **Deadlock** A *deadlock* is a very uncomfortable situation where at least one process cannot proceed its execution and is *stuck*. The processes in a deadlock are waiting for resources that are not just currently but indefinitely unavailable. It usually arises when process requests for critical resources are granted incrementally instead of granting all needed resources at once, avoiding the possibility of another process to snap up a still missing resource. But if the competitor never releases the resource (maybe because this process is also waiting for a resource before proceeding, or because of a failure) - we are in a deadlock. This problem is also possible if only one process can run at a time - but more processes may be active (→ multiprogramming): so one process can block all others and therefore the whole system.

- **Livelock** We speak of a *livelock* if mutual exclusion is guaranteed but the relative speed between processes can lead to sequences of actions that block each involved process. If processes are executed concurrently and are they are both releasing and obtaining a resource that each other wants, and this happens in a timely fashion that both of them cannot get the wanted resource each time they want it - then it is said that there is a livelock. A livelock may not go on forever, different relative speeds of execution can lead to a break of this sequence of actions. An easy example would be two processes that both set and reset their corresponding flags at the same time. Then the two processes also always test the others flag at the same time, always coming to the conclusion the opponent is in the critical section - and both resetting their flag without entering the critical section (s. [Sta01], section 5.2).

- **Starvation** In the case of *process-priorities* regarding resources it could happen that higher prioritized processes continuously access critical resources, preventing a process with a lower priority to get its turn. Also in the case that no priorities are present starvation might occur, due to unfairness, different access-frequencies, failures (e.g. *babbling idiot*), etc..

2.1.7 Mutual Exclusion on Single-Cores

One common method of enforcing mutual exclusion on critical resources is by *implicit synchronization* ([Kop97], ch. 10) of the processes: the execution of the active processes is organized such that no process is interrupted when accessing a critical resource (simplest achieved by disabling interrupts [Sta01]). This requires off-line analysis of the processes to generate a flawless schedule, then, since there is only one processor, there cannot occur any conflicts at all. This method has the obvious drawback of limited flexibility: the operating system cannot interrupt a currently executing process at any arbitrary time.

A dynamic alternative to off-line analysis would be *priority inheritance protocols* like the *priority ceiling protocol* for real-time systems. It avoids deadlocks despite incremental requests for resources by elevating the priority of processes that are in a critical section (s. [Kop97], ch. 11).

Another solution are *locks*: the interruption of processes is possible with the risk that a currently locked resource is unavailable until the continuation of its locker.

2.1.8 From Multiprogramming to Multiprocessing

A computing system that contains more than one simultaneously working processing unit is called a *multiprocessor* in case the processors all have access to a commonly shared memory (key property of a multiprocessor according to [TvS02]). In the case the processing units are even located on the same chip, today's term for this is a *multi-core processor* or *chip-level multiprocessor* (CMP). A processor with only one processing unit is called a *single-core processor*.

On the other hand, a *multicomputer* is a system that interconnects physically separated computer systems where each computer has its own private memory, leading to the field of *distributed systems* (see [TvS02] for details on remotely connected systems). The focus here as in this thesis lies on the current modern variant of multiprocessors, the multi-core processors.

With multiple processing units available it is possible to execute more than just one active process at a time, generating even more new problems, for instance:

- **Data coherence & consistency** If processes are not just competing for resources but also *share* them (like possibly memory), then the processes must rest relaxed that the shared resource is always in a correct state. Coherence is assured as long as there is only one processing unit present in a system, but with more processes concurrently accessing a shared resource that allows this it might happen that the shared resource is not consistent with the view of the accessing processes, that is, the shared resource is not in a coherent, a correct state.

2.1.9 Synchronization

With truly parallel execution on multiprocessors and multi-core processors, implicit ordering of the execution paths is not enough (s. next section). As in dynamically scheduled uniprocessor systems it is necessary to provide ([CS99], 1.3)

- **data synchronization** ⇒ mutual exclusion
- **event synchronization** ⇒ informing other processes that a certain point of execution is reached

In analogy, parallelism in problems themselves can be exploited using *temporal parallelism* or *data parallelism*. An example for the former would be *pipelining*, in the latter the data is split into independent parts. See [RM06] for more about how to solve problems in parallel.

There are three major components of a synchronization event:

1. Acquire synchronization method
2. Waiting algorithm for the synchronization to become available
3. Release synchronization method, enabling other processes to proceed pas a synchronization event

Waiting can be of type *busy-waiting* or *blocking*, whereas locking by busy-waiting is not a preferred locking technique. From [CS99], 5.5.1:

> Busy-waiting avoids the cost of suspension but consumes the processor and cache bandwidth while waiting. Blocking is strictly more powerful than busy-waiting because, if the process or thread that is being waited upon is not allowed to run, the busy-wait will never end. Busy-waiting is likely to be better when the waiting period is short, ...

According to [CS99] and [Her88], a shared data structure is said to be *lock-free* if the operations defined on it do not require mutual exclusion over multiple instructions. If the operations on the data structure guarantee that some process will complete its operation in a finite amount of time, even if other processes halt, the data structure is *nonblocking*. If it can be guaranteed that every process will complete its operation in a finite amount of time, the data structure is *wait-free*.

2.1.10 Mutual Exclusion using Locks

With multiple physical processors *explicit synchronization* is needed to enforce mutual exclusion when accessing critical resources. One common synchronization mechanism is a *lock*. Critical resources may be protected by their corresponding locks. A lock is *taken* by a process if that process successfully acquires it, giving it the exclusive access to work with the resource corresponding to this lock. The lock is said to be *owned* by that process. A lock is *free* if no process currently owns it. Processes possessing a lock must release this lock as soon as they leave their critical resource, otherwise other processes may starve or deadlocks might occur.

There are *software algorithms* to achieve mutual exclusion, mainly *Dekker's Algorithm* and *Peterson's Algorithm*. They both have in common the continuous looping until it is assured that the critical section can be entered exclusively. This *busy-waiting* named looping is not very efficient, even more the first algorithm is quite complex. Even worse, some prerequisites must be fulfilled by the underlying memory-subsystem to guarantee the correctness of the algorithms, that is: a sufficiently strong consistency model (s. 2.1.19). Also, the algorithms mentioned in the previous paragraph work with dedicated flags to achieve mutual exclusion. Those flags indicate whether one process has the right to enter its critical section or not. Hence a kind of locking is achieved by the algorithms, but there is no explicit single lock - all flags together represent the lock.

Hardware support for mutual exclusion is given primarily by special machine instructions that allow to manage a single memory cell to represent a single lock for a given critical resource. According to the hierarchy introduced in [Her88], atomic operations are ranked depending on their relative power (listed in ascending order):

1. atomic *load/store*
2. atomic *Test and Set*, *Fetch and Store*, *Fetch and Add*, *Exchange*

This means that *atomic* read/write registers that are *safe* (coherent), are less powerful and therefore less useful in the construction of lock-free data-structures than, for instance, the *Test and Set* instruction. Queues, lists etc. using primitives of lower power would be considered in this ranking to be even more "powerful".

2.1.11 Semaphores

A breakthrough in coping with concurrent processes was the introduction of semaphores with Dijkstra's treatise in 1965 [Dij65]. A semaphore is an *abstract structure* that contains an integer value. Processes can cooperate with each other by using a semaphore and issuing two kinds of signals to it:
a *test*- and an *increment*-signal. Both signals must be handled atomically. The test-signal corresponds to a request, the targeted semaphore decrements its value. The requesting process is blocked in case the semaphore's value is negative. Leaving a critical section a semaphore receives an increment-signal, resulting in an increment to be applied to the semaphore-value. The greater the initial value of a semaphore, the more processes can signal a test without being blocked. A semaphore that implements a fair first-in-first-out (FIFO) queue for handling the processes waiting for it is called a *strong semaphore* [Sta01].

The simplest type of semaphore is the *binary semaphore* - it corresponds with the lock described in the previous section. However, one should not misunderstand a semaphore for a lock - semaphores are not concerned with consideration of blocking or busy-waiting. Semaphores introduce an additional level of abstraction, in principle the signalling avoids busy-waiting and makes much more efficient scheduling mechanisms possible (blocked processes could be easily detected and put into a queue by the operating system). Nevertheless, in reality semaphores happen to be implemented using locks and hardware-primitives like the test-and-set operation. Of course this implementation is hidden from the applications using a semaphore.

2.1.12 Event Synchronization by Barriers

Event synchronization forces processes to come together at a certain point of execution. They *enter* the barrier, *wait* for the other processes and then all processes *leave* the barrier together. Barriers can be used to separate distinct phases of computation. However, according to [CS99], p.358 barriers are normally implemented without special hardware-support, using locks and shared memory instead.

2.1.13 The Parallel Random Access Machine (PRAM) Model

There exist some *abstract models* that can be used for designing parallel algorithms without the need to concern about the underlying implementation of a given system. Parallel algorithms are a major application for parallel systems. In accordance with [RM06] we look at the PRAM-model as an abstraction of a parallel system, consisting of N identical processors, a shared memory and a *memory-*

address unit (MAU) that allows all processors to access this shared memory. The following acronyms are stated here for further usage in this thesis:

> ... a problem might arise when more than one processor tries to access the same memory location at the same time. The PRAM model can be subdivided into ... categories based on the way simultaneous memory accesses are handled.
>
> Exclusive Read Exclusive Write (EREW) PRAM In this model, every access to a memory location (read or write) has to be exclusive. This model provides the least amount of memory concurrency and is therefore the weakest. Concurrent Read Exclusive Write (CREW) PRAM In this model, only write operations to a memory location are exclusive. Two or more processors can concurrently read from the same memory location. This is one of the most commonly used models.

Other models exist, but are irrelevant considering the practical nature of this thesis. It is pointed out here that the EREW model does not take any advantage of a multi-ported memory - like on a bus, all accesses are exclusive and thus serialized.

2.1.14 Real-Time Systems (RTS)

Referring to [Kop97], if the *correctness* of a system not only depends on the correct resulting values but also of their deliverance *in time*, then such a system is called a *real-time system*. The time when a result must be available is called a *deadline* (*completion deadlines*). The problem of ordering the execution of processes in a way that all deadlines are met is called the *scheduling problem* and is nontrivial. In order to give any guarantees about the correct function of a real-time system the execution time of processes must be known in advanced, or more precise: the longest execution time. This time is commonly referred to as the *worst-case execution time* (WCET). Computing this time becomes more complicated the more abstract the definition of the processes is. Speculative enhancements of designs (caches, speculative execution, ...) increase *unpredictability* and result in relatively rough estimates for the WCET. The WCET is an important area of research, an introduction in the subject as well as a broad overview is given in [Sto06].

Many systems that are *safety-critical* are real-time systems per definition (e.g. processes in nuclear plants, aeroplanes etc.). An automobile must also meet some stringent timing requirements in order to react to its user's input *in-time*, for instance the *break-* or *steer-by-wire* units.

2.1.15 Fault-Tolerant Systems (FTS)

Closely related to RTS are fault-tolerant systems, introducing or using present *redundancy* in resources to be able to handle errors (the results of faults) that might lead to failures of a system. The seriousness of errors that might be tolerated by a given FTS depends on the amount of redundancy and the mechanisms used.

Fault tolerance is not a focus in this thesis. It is mentioned here for completeness and to point out alternative usages of multiple processing cores in this field . Refer to [TvS02] for fault-tolerance in distributed systems and [Kop97] for fault-tolerance regarding real-time systems.

2.1.16 Architectures of Multiprocessor Systems

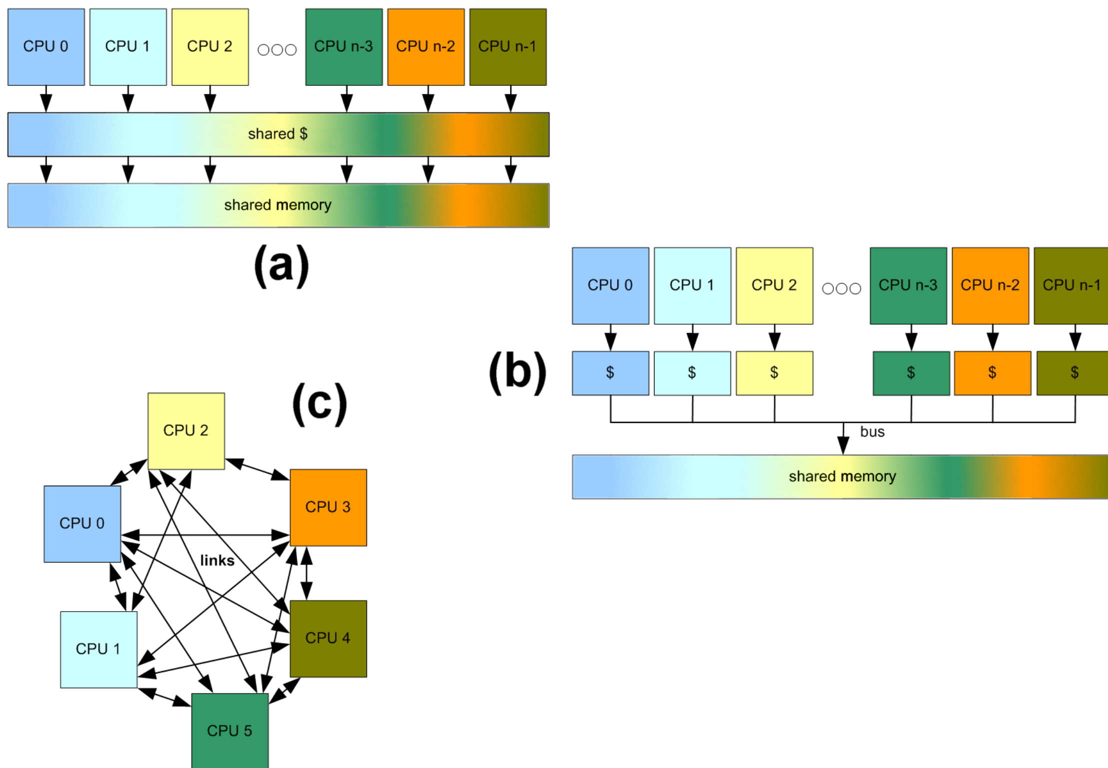


Figure 2.1: Architectures of multiprocessor systems, simplified

In accordance with [CS99] common types of small- to medium-scale multiprocessor system-architectures are presented in figure 2.1. These architectures provide a global physical address space, access to all of main memory is possible from any processor. Such a system is often called a *symmetric multiprocessor* (SMP).

Non-symmetric architectures are not considered here since they are irrelevant regarding this thesis.

Architecture of figure 2.1 (a) can make sense for a multi-core chip where the cores would *share* an *on-chip first-level cache*. Besides that this architecture was used mainly in the mid-1980s when it was typical to connect a low number of processors on one motherboard. More common today is to have local first-level caches and shared second-level caches.

Architecture of figure 2.1 (b) is the most common form for up to 20, 30 processors. A *bus* shared amongst the processors connects them to the shared memory. The number of processors to connect is limited by the bandwidth of the shared bus and the efficiency of the local caches that decrease the average-case load on the bus.

Simplicity is supported by using a bus, but for a massive parallel systems containing many processors a single shared bus might significantly limit performance. Hence, to efficiently connect a large number of processors the bus must be replaced by more efficient means of interconnections, typically a scalable point-to-point interconnection network. In figure 2.1 (c) such an interconnection network is given by links between each pair of processors. A *link* is a direct connection between any two processors. With the number of links necessary for full point-to-point connections in networks being a square function, pure linking can become too costly. A *switching network* allows to connect any two processors in a network, but it is possible that a connection cannot be established until other connections and hence resources are released. This depends on the amount of communication and the design of the switching network. Look into [CS99], chapter 10 for in-depth information on interconnection network design.

The discrepancy between using a bus and using links or switches is quite apparent. The results of this thesis make this difference also quite obvious.

2.1.17 Caches

Local caches reduce the *average load* on the bus, therefore increasing the number of processors that can be connected. Still the worst-case would be to assume all caches to be invalid, leading to a bus-load the interconnection network might be unable to handle. But caches not only add unpredictability (which makes them quite unattractive for real-time systems), they also introduce a new problem in parallel systems called *cache coherence*. Burdened with these disadvantages, caches become very uncomfortable in respect to real-time systems and are thereby often avoided as a whole. As an example ([CS99]) the second-level cache of all the 2048 alpha-processors of the CRAY T3D were deactivated simply to avoid the longer accesses to shared memory introduced by cache-misses. A very comfortable by-product: *no cache, no cache-coherence* to cope with.

2.1.18 Coherence

The term *coherence* was already mentioned briefly in section 2.1.8. Coherence in the uniprocessor case is a property that simply states that a read returns the *last* value written. In the parallel case this gets more complicated since there are parallel programs interacting such that a location in memory can be accessed concurrently by different processes on different processors.

From [CS99] we take the definition of a *coherent multiprocessor system*:

> ... we say that a multiprocessor system is coherent if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location ... that is consistent with the results of the execution and in which
>
> 1. operations issued by any particularly process occur in the order in which they were issued to the memory system by that process, and
> 2. the value returned by each read operation is the value written by the last write to that location in the serial order.

With this definition it is clear that all memory-accesses themselves must be *atomic*; if they were not, reading a currently written memory location can result in an arbitrary bit-value. Clearly, such a scrambled bit-value is not the result of any total ordering of the processes involved. It is interesting to add here that a bus simplifies this problem by its inherent property of *serialization*: only one action can be done on the bus at one time (assuming a one-channel bus).

It should be obvious by now how caches complicate things by introducing *cache-coherence*. Additional complex measures like *bus-snooping* (MESI, Dragon - s. [CS99]) are necessary to guarantee coherence for the more complex caching-methods as write-back caches. This additional overhead can be avoided by simply using no caches, it eases the handling of real-time systems greatly (→ 2.1.14).

2.1.19 Consistency

In essence, coherence says that a written value will eventually become visible to another processor core or other cores. But what is not defined is when this written value becomes visible. In parallel programming it is often desired to establish an order between reads and writes of a single and multiple programs or processes, in more detail taken from [CS99] again:

> ... Coherence says nothing about the order in which writes to different locations become visible. ... A *memory consistency model* for a shared address space specifies constraints on the order in which memory operations must appear to be performed (i.e., to become visible to the processors) with respect to one another. This includes operations to the same locations or to different locations and by the same process or different processes, so in this sense memory consistency subsumes coherence.

Strict consistency is the most strict kind of consistency ([TvS02]:

> The most stringent consistency model is called **strict consistency**. It is defined by the following condition:
> *Any read on a data item x returns a value corresponding to the result of the most recent write on x.*

Since time is unambiguous in uniprocessor systems strict consistency is normally present there. But in the multiprocessor case this consistency model might be far too restrictive, maybe even severely limiting average performance. But it may well be applicable for multi-core systems due to the common clocking on one chip-die.

As a form of weaker consistency *sequential consistency* is quite common for multiprocessor systems (also called *linearizability*). Weakening consistency even further we go toward distributed systems with *causal consistency*, *FIFO consistency*, *release consistency* etc., for details on those also called *data-centric* as well as also *client-centric consistency models* please look into chapter 6 of [TvS02].

Referring to the sufficient conditions for preserving sequential consistency in [TvS02] it is easy to see that using a bus for interconnecting our processor cores simplifies achieving this type of consistency by its inherent serialization. Still the consistency model has to be guaranteed by the bus protocol itself. One must be aware that it is *easy to violate such a strict consistency model.* For instance, priorities assigned to different processor cores may lead to older requests for access to be carried out after more recent accesses with a higher priority. So *starvation* is also a *clear violation of strict consistency.* Even worse, if a compiler rearranges the order of accesses to memory, consistency is violated before the hardware gets involved. Hence it might be necessary to prevent such optimizations, e.g. by using the keyword *volatile* when programming in ANSI-C.

2.1.20 Scalability

Since this thesis is concerned only with small- to medium-scale multiprocessor systems, large-scale multiprocessor systems may not seem important at all. Still, looking at scalability may uncover some of the bottlenecks also present in small-scale multiprocessor systems. For instance, looking at bus-based multiprocessor systems it is stated in [CS99] that extending a bus increases the latencies, resulting in lower usable bandwidth and thus lower maximum frequencies. More processors or nodes connecting to the bus also degrade the signal quality and hence performance. Clearly a bus-based system is not scalable over a certain technology-specific point.

A glimpse at the early attempts to scale bus-based multiprocessor systems is given in [ea93]:

> The performance of earlier bus-based multiprocessor machines had demonstrated performance degradation with more than four processors connected to the bus. Cache coherence traffic and bus contention made more processors counterproductive.

It becomes obvious by studying the results of this thesis (s. chapter 5), the limitations introduced by a bus are relevant also with only a few processor cores present, in particular when performance is time-critical.

2.1.21 Transactional Memory (TM)

Transactions are a very useful abstraction, grouping a series of actions and executing them with the following properties holding ([TvS02]):

> The all-or-nothing property of transactions is one of the four characteristic properties that transactions have. More specifically, transactions are:
>
> 1. Atomic: To the outside world, the transaction happens indivisibly.
> 2. Consistent: The transaction does not violate system invariants.
> 3. Isolated: Concurrent transactions do not interfere with each other.
> 4. Durable: Once a transaction commits, the changes are permanent.
>
> These properties are often referred to by their initial letters, **ACID**.

In addition we state from [TvS02]:

> A transaction that completes successfully *commits* and one that fails *aborts*. ... we will call this property *failure atomicity* to distinguish it from a more expansive notion of atomic execution, which encompasses elements of other ACID properties.

Transactions originate from database-design in the world of business where series' of operations must be executed as a whole or none at all (like transfers on accounts). The power of using transactions comes from the fact that if any action of the transaction fails, the whole transaction has no side effect.

Transactional memory (TM) offers a modern and comfortable solution to introduce more abstraction at the software designer's level. As is written in [LR07]:

> The basic idea is very simple. The ACI properties of transactions provide a convenient abstraction for coordinating concurrent reads and writes of shared data in a multi-threaded or multi-process system. Accesses to shared data originate in computations executing on concurrent threads that run on one or more processors. ... Today, this coordination is the responsibility of a programmer, who has only low-level mechanisms, such as locks, semaphores, mutexes, etc., to prevent two concurrent threads from interfering. ... Transactions provide an alternative approach to coordinating concurrent threads. A program can wrap a computation in a transaction. Failure atomicity ensures the computation completes successfully and commits its result in its entirety or it aborts. In addition, isolation ensures that the transaction produces the same result as it would if no other transaction were executing concurrently. ... If a programmer's goal is a correct program, then consistency is important, since transactions may execute in unpredictable orders. It would be difficult to write correct code without the assumption that a transaction starts executing in a consistent state. Failure atomicity is a key part of ensuring consistency.

Despite today's hype for transactional memory consider this ([TvS02]):

> Transactions are not a panacea. It is still (all too) easy to write an incorrect concurrent program, even with transactional memory.

The concept and idea of transactional memory and the related work done in this field (essentially hardware-acceleration, s. more in the succeeding section 2.2.3) heavily influenced the development of this thesis.

2.2 Related Work

A rough overview about the research and related work that is more or less related to this thesis is given in this section. It must be noted here that general design practice is mentioned along with references of concrete related work. An interesting fact to point out is that in parallel systems *performance is not always the main concern*. As can be seen in [ea04a], research is also concerned about how to *reduce* die area and thus *costs* in parallel systems by let adjacent processor cores share their resources (even caches, floating point units etc.) while trying to keep the performance degradation as low as possible.

2.2.1 Mutual Exclusion

The criteria against which to judge any locking mechanism are [CS99]:

- low latency
- low traffic
- scalability
- low storage cost
- fairness

If the underlying hardware has no support for mutual exclusion, locking must be done by software alone. Still, improving efficiency beyond the level pure **software approaches** can offer needs support by an atomic *test&set-, exchange-, swap-, fetch&op-* or *compare&swap-instruction* provided by the hardware. The atomicity condition for these instructions must be fulfilled regarding all processors in the system, not for one alone (chapter 3 shows what happens if this is violated).

According to [CS99] it becomes more common to implement a pair of **special instructions** to access a *synchronized variable* instead of a single read-write-modify instruction as mentioned in the previous paragraph. From [CS99]:

> The first instruction, commonly called *load-locked* or *load-linked* (LL), loads the synchronization variable into a register. It may be followed by arbitrary instructions that manipulate the value in the register - that is, the modify part of the read-modify-write. The last instruction of the sequence is the second special instruction, called a *store-conditional*. It tries to write the register back to the memory location (the synchronization variable) if and only if no other processor has written to that location (or cache block) since this processor completed its LL.

An example for a processor architecture that uses such a pair of special instructions is the Power PC 405 used in this thesis ([IBM05] and [Xil07g], appendix D).

Some first implementations of **locks in hardware** used a set of *lock lines* on the bus interconnecting the multiple processors. A processor that wanted to lock asserted a free locking line, a *priority circuit* elected the winner in case of multiple concurrent requestors [CS99]. Clearly such a hardware approach was inflexible since the number of locks and the waiting algorithm is fixed. Providing more locks could only be accomplished by the overlaying operating system that used these hardware mechanisms.

A similar implementation was realized in the **CRAY X-MP** [ea89]: a set of *lock registers* was shared by the processors and even allowed to be assigned to certain processes. But with only a small fixed number of shared registers this approach needed the operating system to make use of this underlying hardware to provide the software with a variable number of higher-level locks in memory.

About the **performance** of such software locking methods based on a special atomic instruction is said in [CS99]:

> Consider the atomic exchange or test&set lock. It is very low latency if the same processor acquires the lock repeatedly without any competition, since the number of instructions executed is very small and the lock variable will stay in that processor's cache. However, we have seen that it can generate a lot of bus traffic and scales poorly as the number of competing processors increase.

So, with increasing competition for a critical resource the above busy-waiting scheme for a lock might lead to overloading the bus and *thrashing* ([Kop97]) of the overall multiprocessor system. A way to prevent the overhead in communication to explode by increased contention is by using the advanced *ticket locking* algorithm as described in [CS99]:

> Every process wanting to acquire the lock takes a ticket number and then busy-waits on a global `now-serving` number ... until the `now-serving` number equals the ticket number it obtained. To release the lock, a process simply increments the `now-serving` number so that the next waiting process can acquire the lock.

Advanced lock algorithms like the Ticket-Lock described previously prevent all cores to rush for a lock when it is released and therefore avoid starvation with the one-way cycling of the ticket (like a token) between the processors. For further information on advanced locking algorithms see [CS99], 5.5.3.

[MS95] starts with a somewhat theoretical approach toward the question which of the atomic primitives *test&set*, ... to implement in hardware - and where: in the cache- or the memory-controller. It is also asked which coherence policy should be used for atomically accessed data and different answers to those general questions are examined, resulting in some experimental results. The paper can be recommended as an advanced reading on atomic primitives. In this context [CL97] is interesting in the aspect that it shows how cooperation between a compiler and a memory coherence protocol is able to improve the performance of Fortran programs running on distributed shared memory systems.

In [ea90] the architecture of a **RISC-based multiprocessor** is described. The goal was to bring many processors on one chip. In that perspective the paper is describing an approach for a multi-core chip, although the term is not used yet. Each processor has *12 channels* to send data to the other processors, 4 bytes can be sent on a channel without blocking the sending processor. In a multi-core chip these channels between the processors can be easily provided on-chip as well. With *compiler-support* the channels are used to coordinate the processes running on the different processors:

> The channels are used to execute parts of the program which are inherently sequential and only contain instruction level parallelism. The use of channels allows processors to drift in relation to each other ... The execution of operations on different processors is scheduled by the compiler. ... Compiler techniques for efficiently allocating a fixed number of channels have already been developed.

An important work with respect to this thesis is [NP00] where the frequent synchronization primitives locks, barriers and lock-free data structures are the focus of attention. The classical implementations of those primitives are compared against **hybrid synchronization primitives** that use hardware support and also the caches (implementation on a cache-coherent system) to improve efficiency and scalability of the primitives. To state a result from [NP00], the hybrid test&set locking is over twice as fast as the classical one without hardware support.

An example for a **specialized multi-core chip** is the HiBRID-SoC architecture from [ea02]. A DSP-, RISC- and VLIW-core are connecting to all their *common interfaces* by a 64-bit AMBA AHB bus. One of those interfaces connects them to the external SD-RAM. For *fast inter-core synchronization* each pair of the three cores share a block of *dual-ported memory on-chip*. Caching is not done for the on-chip but for the off-chip SD-RAM. This configuration shows some relevance regarding the hardware configurations used for this thesis (s. section 5.2.1).

2.2.2 Event Synchronization

In multiprocessor systems **interrupts** can be used *to signal events* between processors, maybe by using a dedicated shared memory region to convey data ([CS99]). **Barriers** are normally implemented using *locks*, *shared counters* or *flags*. A barrier for an arbitrary number of processors is called a *centralized barrier* when it uses only one counter, one lock and a single flag. When barriers are implemented using locks the barrier algorithms have the same problem as locking algorithms: $\Rightarrow$ all the processors that want to block at a barrier contend for the *same* lock.

One way to circumvent this bottleneck is to equip the system with **hardware barriers**. Barriers between an arbitrary number of processors can be realized using a *separate synchronization bus*. A simple wired-AND line is enough - all processors reaching a barrier assert their signal at this line. As soon as all processors are arrived the signal line yields 'high', releasing the waiting processors all at once.

[ea90] explains the drawback of the *idle time* of a processor waiting at a barrier and shows how compiler-support can help to improve this using **fuzzy barriers**:

> The waiting of processors at barriers is reduced by using compile-time techniques to find useful instructions such that can be executed by a processor after it is ready to synchronize. ... If processors reach the barrier at different times they are less likely to stall at a fuzzy barrier than at a fixed barrier. ... The fuzzy barrier makes the system tolerant of drift in the progress of individual instruction streams.

The code that is to be executed while waiting at a fuzzy barrier is called *barrier region* and is generated by code reordering and other program transformation techniques. Such transformations can make the programs quite large ([ea90]) and even more: a reordering of code may not be adequate for all applications.

In [CS93] an in-depth analysis of how to provide an efficient synchronization by barriers on a shared memory multiprocessor with a shared multi-access bus interconnection (like CSMA/CD) is given. Some applicable algorithms are presented together with their performance-results.

An innovative - if not unorthodox - **alternative to ordinary barriers** (hardware barriers or mapped onto locks) is given in [ea05b]: the waiting of a thread is forced by *continuous invalidation* of the respective I-line *of the instruction-cache*. Additional logic in the second-level instruction cache ensures that such artificial cache-misses are kept from the out-chip bus by ignoring it. As soon as all threads that are needed at this synchronization point have joined the waiting thread the caches resume normal operation. A big advantage here is the fast continuation since the threads resume operation instantly when the next instruction is given by the first-level cache (which got it by the second-level cache on barrier-release).

2.2.3 Transactional Memory (TM)

Hardware-TM (HTM) originated in the quest for programs that are not concerned about explicit locking. The research in the field of HTM shows some very interesting developments in adding hardware-support for transactions - mainly to simplify programming and improve performance. Many concepts are not inevitably tied to TM and may give inspiration for non-TM architectures as well.

The works presented in this section are a selection that seemed most related to the work done in this thesis, an extensive survey (50 sources) of transactional memory (as well as an attempt to categorize its implementations) can be found in [LR07].

The work of Jensen et al. ([ea87]) from 1987 describes how to use architectural support for writing **lock-free program code**, avoiding the performance degradation that is most severe *when locking only single instructions* (benchmarks supporting this statement are given in chapter 5). Support from the coherence protocol and the compiler are needed - the mostly complex details are left out here but can also be found in [LR07].

The paper [SS93] describes an extension of Jensen et al. such that not just a single but a *bounded number of memory locations can be locked by hardware support.* To accomplish this, new instructions operating on new reservation-registers were implemented. With this extension it is possible to access multiple memory locations using lock-free code. A similar approach is made in [HM93]. The goal is the same: to develop lock-free data structures. This time a **transactional cache** was used to monitor and buffer accesses regarding transactions. Still, the size of the transactional cache represented a bottleneck severely limiting the size of the transactions. This paper used and therefore coined the term transactional memory.

Referring to [LR07] the **first industrial implementation of transactional memory in hardware** is the IBM 801 storage manager system from 1988 [CM88]. Hardware support was added by *additional registers to keep track of ongoing transactions* and, most importantly, *associating a lock to each page of memory* (extension of page table entries and translation look-aside buffers). By collecting the locks of all data involved in a transaction the transaction could be executed, if some of the locks are already occupied hardware exceptions call some routines to resolve the matter (like waiting for the missing locks).

An interesting approach is done in [RG01]: the main idea here is that a processor does not need to get a lock but only needs to monitor it during executing a critical section, thus saving the need to set and release a lock. This is of course a kind of speculation and is called **speculative lock elision (SLE)**. Hardware support for the speculative execution of a critical section is used: the processor executes it as if the lock were not present - the lock elision is done by the hardware. In case of a conflict the hardware resets execution with restoring the original data.

After multiple failed speculations or with the critical section being too extensive to be supported in hardware, the processor does not speculate and acquires the lock explicitly. For all this to work a tight dependency between hard- and software is unavoidable (a downside of more complex hardware support).

The concept of transactions is taking over in [ea04b], defining a shared memory model where **all operations execute inside transactions**. In this transactional coherence and consistency model, *a transaction is a basic unit of work* which leads to a simplification of parallel programming in comparison with conventional synchronization. Transactions are executed speculatively, in case of conflicts only one transaction is allowed to commit system-wide (determined by a global token).

Interesting in [ea06] is that tracking data conflicts does not depend on caches and coherence or consistency protocols (like most other approaches for HTM). Instead, **address-information is sent when a transaction is ready to commit**, revealing address-conflicts. The implementation is called *Bulk*. Address-conflicts can be detected word-precise, the compressing of transaction-addresses into so-called *signatures* is done by the hardware. Such signatures represent a compressed superset of all the read- and write-addresses comprising a transaction (this can lead to *false conflicts* due to the compression).

Summing up this section, some main ideas influenced by HTM came up during this thesis as well. Still, there is a main difference to be aware of: transactions may not commit even if executed. In this thesis such an approach to redo a block of executions is out of the question, as *roll-backs* are in general considered of limited utility regarding safety-critical systems where time is not to be wasted and actions already done may be irrevocable (s. also [Kop97], chapter 1).

2.2.4 A Glance at Scalable Multiprocessors

Most of the more recent designs extending shared memory systems into large-scale multiprocessor systems use **distributed memory schemes**, leaving the classical shared memory architecture due to the difficulty implementing a shared memory for many processors. What is left is the *notion of a shared memory by* keeping a *shared address space*. Sharing a physical address space among processors of a large-scale multiprocessor system enables the usage of *simple load/store-instructions* to invoke network-operations. Without a shared physical address space it is necessary to use dedicated *message passing facilities* to communicate. The nCUBE/2 is an example for only private memories, the CRAY T3D on the other hand has a shared physical address space. Both systems are described in some more detail in the following.

The **nCUBE/2** is a rather old but still presentable representative for a multiprocessor system. The area of the 2,048 VAX-based processor nodes (hypercube configuration, build in 1991) that would have normally be used for the cache was occupied by diverse communication logic. Only a small code and an even smaller first-level data cache was available to each processor (128 to 64 bytes [CS99]). 14 channels are used as *unidirectional links* between the processors and can be used for synchronization. Each processor connects directly with *local SD-RAM*. This system has a shared address space, but no physically shared memory.

Yet another example, the **CRAY T3D** is a parallel system containing of 2,048 DEC Alpha 21065 microprocessors with up to 64 MB *private memory* each. Interestingly the second level cache of all processors (512kByte each) is deactivated to reduce the main memory's access time (cache-misses cost an additional clock cycle). The T3D has a special global AND- and global OR-network to support synchronization (primarily for barriers). A broad perspective can be gained in [CS99], 7.6. A compiler-perspective on how to work with the CRAY using the parallel extension of ANSI-C, Split-C is given in [ea95].

The highly parallel system **KSR1** from Kendell square research described in [ea93] has large local caches for each processor and achieves synchronization by hardware-support of the memory subsystem. This architectural technique is called *ALL-CACHE*. A shared physical address space mapped on devices is present, *locking is accomplished by putting memory pages into an atomic state.* A page in the atomic state is the only valid page throughout all local memories, the memory subsystem ensures invalidation of all other pages. A processor that successfully locks a memory page (make it atomic) automatically gets the page moved into its local memory where it can be used directly.

The **major bottleneck** for highly-parallel systems is the connection of the processors to the memory. In [SZ02] the insufficient performance of parallel systems built from commercial off-the-shelf components for programs with low locality is unveiled. So instead of using already available commercial modules as parts for a parallel system, further integration presents the possibility to *locate memory together with the executing cores on one chip.* This field of research focused on overcoming the interconnection bottleneck is known also as the area of **processor-in-Memory architectures (PIM)** and gives quite some insight into the new perceptions in this field. Like the results of this thesis the work of [SZ02] helps in gaining additional insight into how to judge a given multiprocessor architecture in respect to different types of parallel applications it is intended to host.

The **IBM BlueGene/L** team goes another way to weaken the bottleneck of the processor-memory interconnection, as well as other disadvantages in traditional massively parallel SMPs. [Tea02]:

> To scale the next level of parallelism, in which tens of thousands of processors are utilized, the traditional approach of clustering large, fast SMPs will be increasingly limited by power consumption and footprint constraints. ... In addition, due to the growing gap between the processor cycle times and memory access times, the fastest available processors will typically deliver a continuously decreasing fraction of their peak performance ... The approach taken in BlueGene/L (BG/L) is substantially different. ... The design point of BG/L utilizes IBM PowerPC embedded CMOS processors, embedded DRAM, and system-on-a-chip techniques that allow for integration of all system functions ... Because of a relatively modest processor cycle time, the memory is close, in terms of cycles, to the processor.

The BlueGene/L interconnects its highly-integrated nodes (having a large local memory) by a 3-dimensional torus network for point-to-point communication. One additional network is a global barrier- and interrupt network for event synchronization.

Since we are not concerned about large-scale multiprocessor systems we conclude this section here, but not without emphasizing that the hardware support and high-scale integration of such systems might be relatively advantageous for small- and medium-scale parallel systems as well.

For more information about large-scale systems and the approach to make them as easily programmable as a shared memory system see [TvS02], 1.4: *Distributed shared memory systems*. More details about synchronization approaches in massively parallel systems is given in [ea93, CS99]. For more insight into the different types of interconnection-networks possible for multi-core architectures see also [ea05a] (crossbars, shared bus fabrics etc.).

Chapter 3

Software Synchronization

A hardware-environment based on *two processor cores* is used in this thesis for the application and practical verification of the implemented concepts.

This chapter starts by briefly introducing the actual *Xilinx dual-core system* at hand. The features of the hardware are presented in order to get an overview. For the rest of the thesis an *abstract point of view* is maintained to favor *generality*. Plenty further in-depth information about the hardware can be found by consulting the respective references of the vendor Xilinx (esp. [Xil07e, Xil07h]).

The rest of this chapter is concerned with how to achieve *coherence* and *consistency* on the described hardware platform. Since there is no help from the hardware by default, this is accomplished just *by software means*. More to the point, the traditional approach of *spinning locks* to enforce mutual exclusion is followed.

The *digital flow* necessary for this thesis had to be developed by the author, the software tools used (at the time of this thesis) were unable to provide all of the functionality required. The complete flow is described in the appendix A.

3.1 The Hardware Platform

Here we present a fully-arranged dual-core system configuration as the executing hardware platform for our software. The main focus is on how to achieve *reliable communication* between the two PowerPC cores using the *on-chip shared memory*. To enable communication, synchronization between the cores regarding the access to the shared memory is necessary.

A Xilinx development board *ML410* with a mounted FPGA *Virtex-4 FX60* is used. The configuration of the dual-core system as it is used in this thesis is drafted in figure 3.2. It represents an **abstract architectural view** of our dual-core system and relieves us from irrelevant vendor-specific details (→ [Xil07e, Xil07h]).

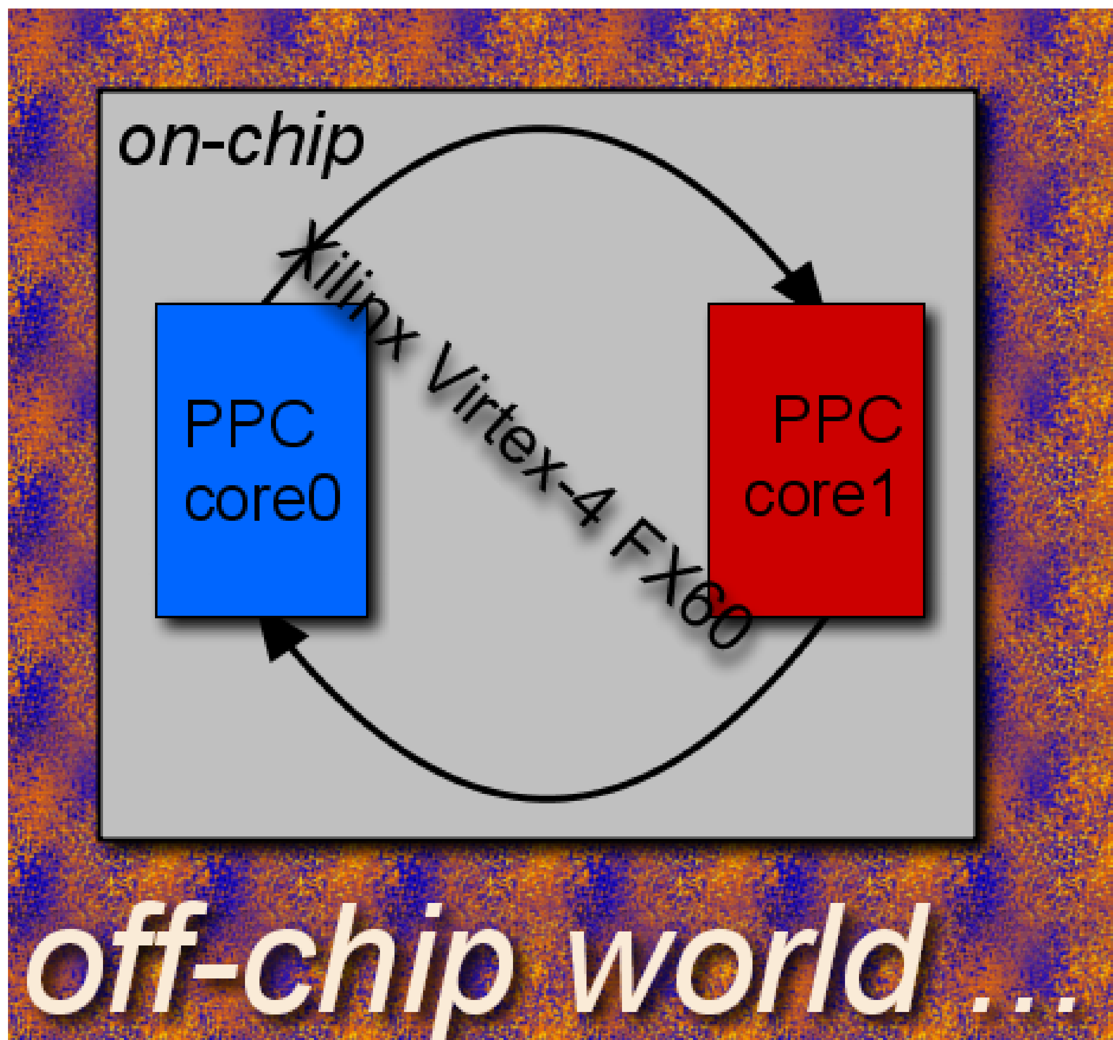


Figure 3.1: Ideal communication stays on-chip

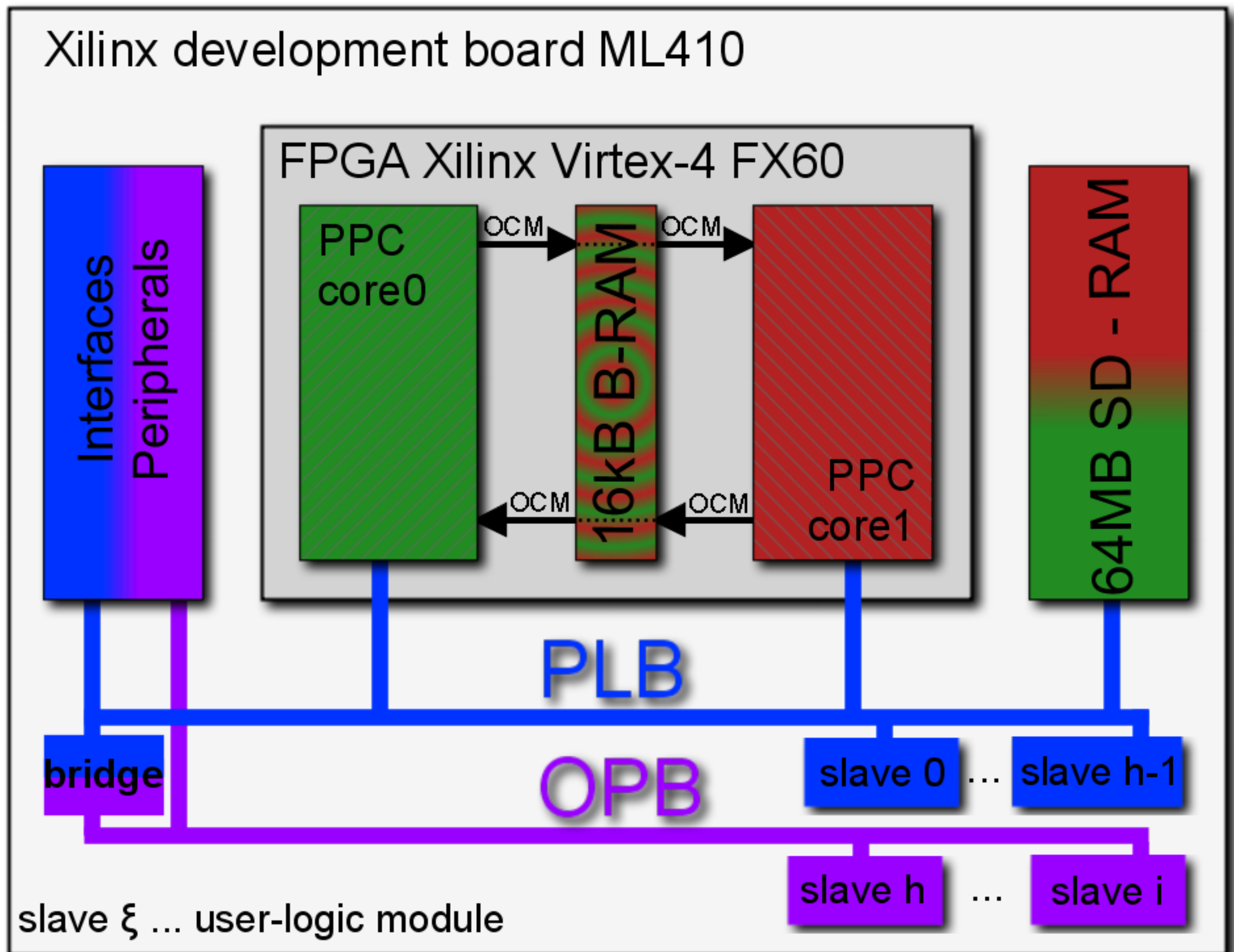


Figure 3.2: Architecture of present dual-core system

The *Block-RAM* is used to speed up the inter-core communication. The 'closer' a shared memory is to a processor the faster is the access. It would be optimal for all communication between the processor cores to stay *on-chip* as sketched in figure 3.1. However, the Xilinx reference design ([Xil07a]) uses only the significantly larger external SD-RAM as shared memory. It is one of the goals of this thesis to examine the impact on the system's performance such design decisions impose.

The two types of memory which are accessible by both cores are (s. fig. 3.1):

- 16kB (16,384 bytes) on-chip Block-RAM concurrently accessible by both cores' on-chip memory interface (OCM)
- 64MB (67,108,864 bytes) shared SD-RAM connected to the two cores through the IBM peripheral local bus (PLB)

We paid our architecture two different types of memory:

- *internal* memory
  The internal Block-RAM holds the **program-code** for each core, connected exclusively to its core by a dedicated instruction-side on-chip memory-controller (IOCM, left out in fig. 3.2). Hence there is no interference in our benchmarks by code-fetches.
  Another part of the Block-RAM serves as **shared memory** with each core accessing it directly via the data-side OCM (DOCM). This on-chip shared memory is solely for the interaction between the cores.

- *external* memory
  The external SD-RAM provides each core with a sufficiently large **private data**-section (heap, stack, read-only data etc.). There is also a small **shared memory** embedded, serving as counterpart to the internal shared memory for conducting the benchmarks in chapter 5.

Commonly the on-chip memory is limited much more than the external one (max. 522kByte Block-RAM for the Xilinx VFX60). For non-real time systems the 16kByte of shared on-chip memory may be far too less too hold big chunks of data, but for our purposes it is more than enough. In general a small but fast shared memory may still complement and disburden a larger external one to speed up on-chip synchronization - an advantage multi-core chips have over multiprocessor systems with more than just one chip. Evaluation of the performance between these different kinds of shared memory is also a major point of this thesis.

In our system the clocks relate to each other as the following proportion:

$$\boxed{f_{core0} : f_{core1} : f_{OCM} : f_{PLB} : f_{OPB} = 1 : 1 : 1 : 1 : 1}$$

To get this proportion and avoid wait-states using the busses we chose a global frequency of *100 MHz*. Increasing it would lead to wait-states (e.g. a core-PLB frequency division of 2:1) which would only proportionally worsen any results using a bus for our benchmarks (s. chapter 5). Considering the current state-of-the-art in the automotive environment this frequency constellation seems fully appropriate.

The detailed information about how to change the architecture by reconfiguration is sourced out to the appendix A.

3.2 Low-Level Coherence

As already described in the previous section our dual-core system has on-chip Block-RAM configured as shared memory. The access to this dual-ported RAM is done by the core-side on-chip memory-interfaces (OCM). Using OCM we have the same frequency for accessing the BRAM as we have for clocking the cores. Due to that fast access (*2* cycles per single store or load) there is *no caching* by default.

In the data sheet of the dual-ported Block-RAM is no information about whether it is allowed to concurrently read and write the same memory cell or not. Hence we must test the dual-ported shared memory for *data-coherence*. One processor core writes repeatedly to the same memory-word, thereby toggling all bits:

- First value: $2^{32} - 1$
- Second value: zero

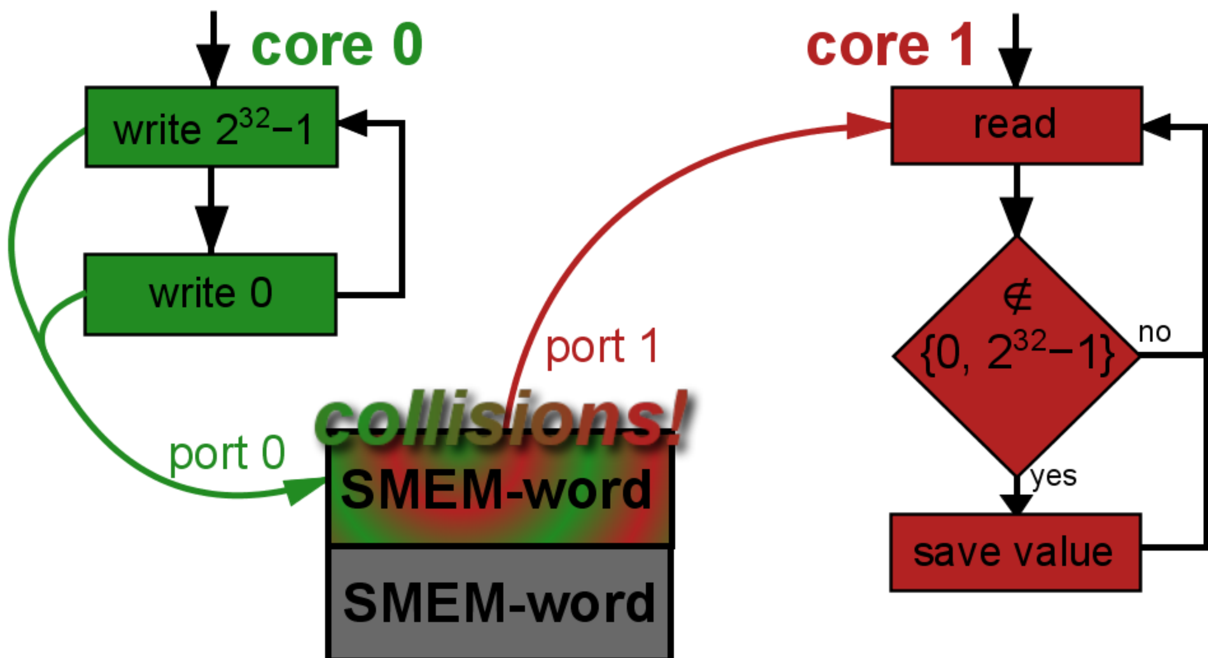


Figure 3.3: Dual-ported B-RAM without low-level coherence

A simple flow diagram is shown in figure 3.3. It indicates that the full concurrency introduced by the two ports causes problems in case one core reads when the other is currently writing. This structure is used in the automatic testing programs `prove_core0` and `prove_core1` which, due to size, are not printed in this thesis. The experimental evidence conducted shows that using the structure of figure 3.3 (implemented in ANSI-C) gives a statistic of over 60 percent corrupted reads.

With the help of assembler we can artificially generate the **worst-case**:
It is possible to program a series of writes and reads such that there are *no intermediate instructions* at all. If the overlapping is worst as demonstrated in figure 3.4, no read gives back a coherent state in case the 2-cycle long store-/read-instructions start at the same time. Then the read-instructions read out the bit patterns at exactly the same time they are written to, resulting in the almost *arbitrary values* shown in 3.4. Those values are experimental return-values and can be found in the log files and in the screenshot A.19.

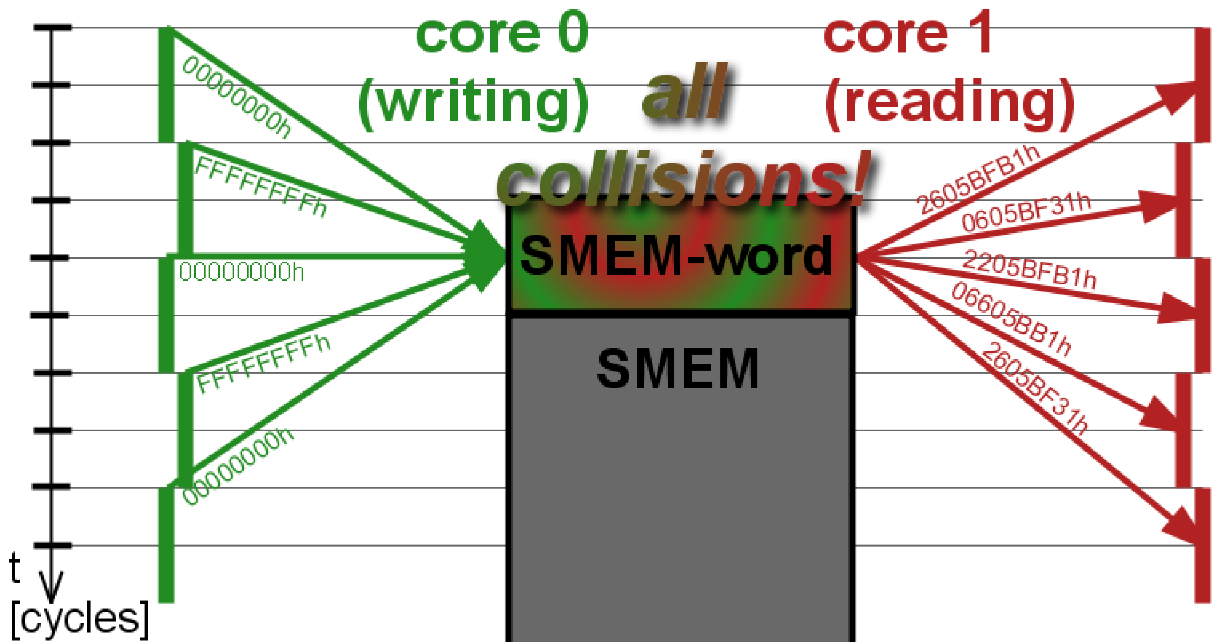


Figure 3.4: Worst-case non-coherent access: no single correct value read

Now it is also clear that our configuration of the system does not use the Block-RAM as it was intended to be used: the RAM-blocks have no additional logic that handles concurrent read-write accesses (like buffering and serializing accesses to the same memory cells). Summing up, we have no coherence using our dual-ported shared Block-RAM as it is. Whenever there is a writer involved the values read might be corrupt. Even worse, the *probability of corruption* depends on the frequency shared memory locations are accessed by all cores. It is impossible for a single core to tell if a value is valid or not - clearly this situation is unacceptable.

3.3 Coherence by Locking

Since the external SD-RAM is not concerned with coherence at all (accesses over an ordinary *bus* are *atomic* by nature) we will only look at the dual-ported on-chip Block-RAM here. With each core owning one port, the shared memory can be accessed completely concurrently. Hence if not stated explicitly otherwise the term *shared memory* always denotes the *on-chip Block-RAM* in the following.

The first idea how to achieve coherence in our on-chip shared memory originated from the PowerPC manual [Xil07g]: a suggestion on how to implement a *test&set-operation* that ensures atomicity by using a reserved load/store-pair. This special load-operation reserves the memory-location and if there is an intermediate access the subsequent conditional store-operation jumps back to do it all over again (more of *load-linked (LL)* in section 2.2.1). The flow diagram of the respective assembler routine named `smem_lock` is given in diagram 3.6. Unfortunately there is no explicit information in the manual if this works for the dual-core case as well.

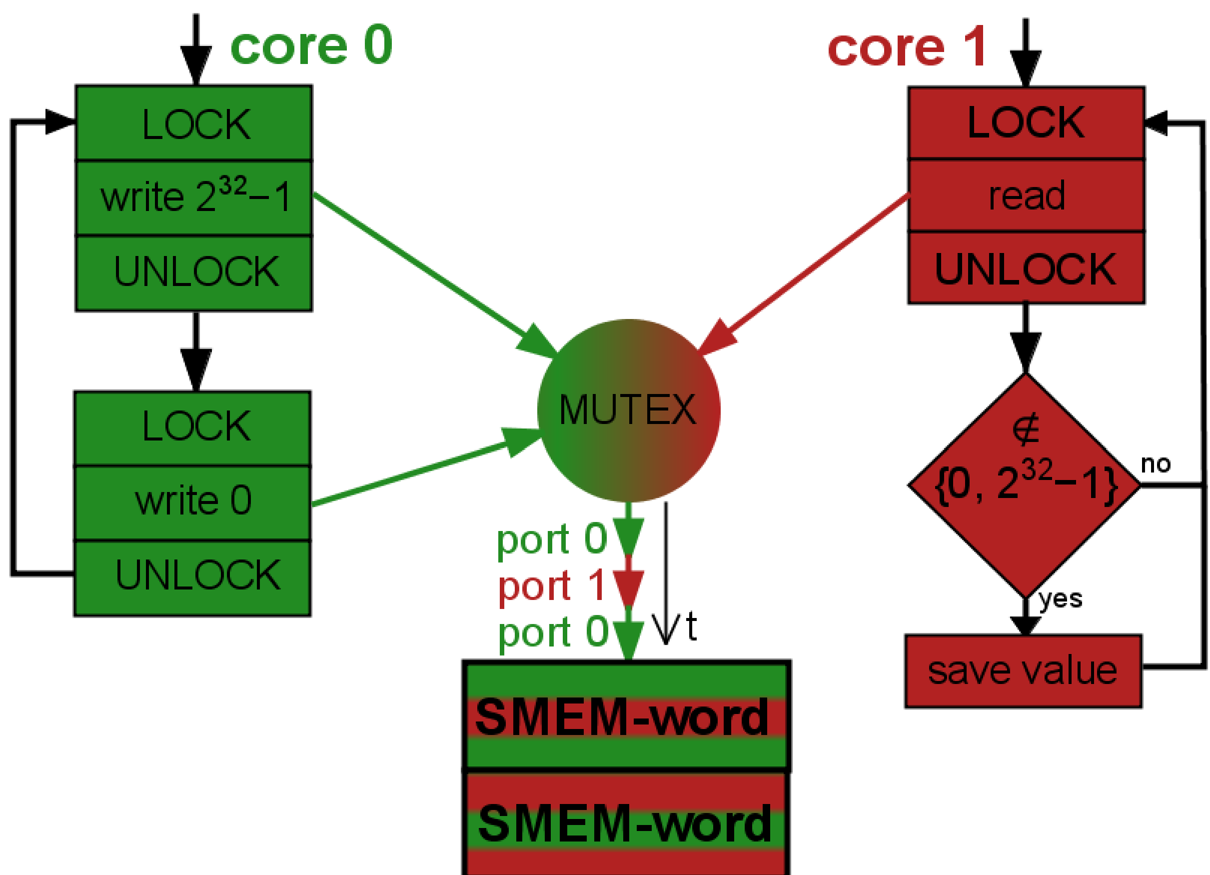


Figure 3.5: Atomic access by locking

The lock can be anywhere in the system as long as all cores can access it. Our lock is set at the end of the shared memory. It must be pointed out that the actual value of the lock depends on the core currently holding it - *each core must have a private lock-value* - all the lock-values must be disjunctive for this is to work (it is the only way we can validate which core won the lock with a memory that does not define the outcome of two concurrent writes to the same memory cell).

With the *test&set* as suggested in [Xil07g] we unveil a major flaw: it is only functional within the processes running on a single core. Closer inspection of the LL-commands shows that it does not work for both cores (the cores would have to exchange information about a memory-reservation, but apparently they do not).

By analysing the problem we can quickly identify its root: one core overwrites the lock already written by its opponent. But the core writing the lock last is not aware that the lock is already reserved since it loaded the value of the lock before the lock was written. This can happen only in a small time-window due to a **race-condition**: consider both cores are loading the value of the lock at the same time, then also testing in parallel. If the lock is free both cores set the lock and go on into their critical sections. This is a clear violation of the principle of mutual exclusion - only one core must be allowed to enter at any time. A core must enter the critical section if and only if it also owns the lock.

This analysis lead us to a revised extended assembler routine `smem_lock_2` with the flow given in figure 3.6. By an *additional test* before entering the critical section a core is now able to detect that it lost the race (of setting the lock) against the other core and retries getting the lock. On the other hand, if the lock is successfully taken, the last test confirms this and the critical section can be entered safely.

Billions of successful test runs exhaustively confirmed the functionality of the *test&set&test*-instruction that ensures atomic access to the shared memory. Presupposing a minimal distance of instructions between any two core's attempts to get the lock, *strict coherence* (temporal ordering of requests) is guaranteed as well. With just a slight time-difference in the locking-attempts the later core might win the lock over the prior one by overwriting it (→*race*, see previous paragraphs).

It must be pointed out that this is an software-synchronization *optimized* for the given system regarding execution time - changes like present caching may invalidate correctness. Since the traditional Dekker-algorithm (→ 2.1.10) is also independent from low-level mutual exclusion it could also be used here - at the cost of a greater software overhead. A formal proof of correctness can be given with arguments similar to those from Dekker, an outline of it is drawn here. In fact the *testset*-instruction alone corresponds to the flawed second step in developing Dekker's algorithm. *Mutual exclusion* can be guaranteed only by entering the critical section after an additional test. The result of this test is always correct due to the unbuffered direct 2-cycle access to the Block-RAM. A *deadlock* cannot occur since

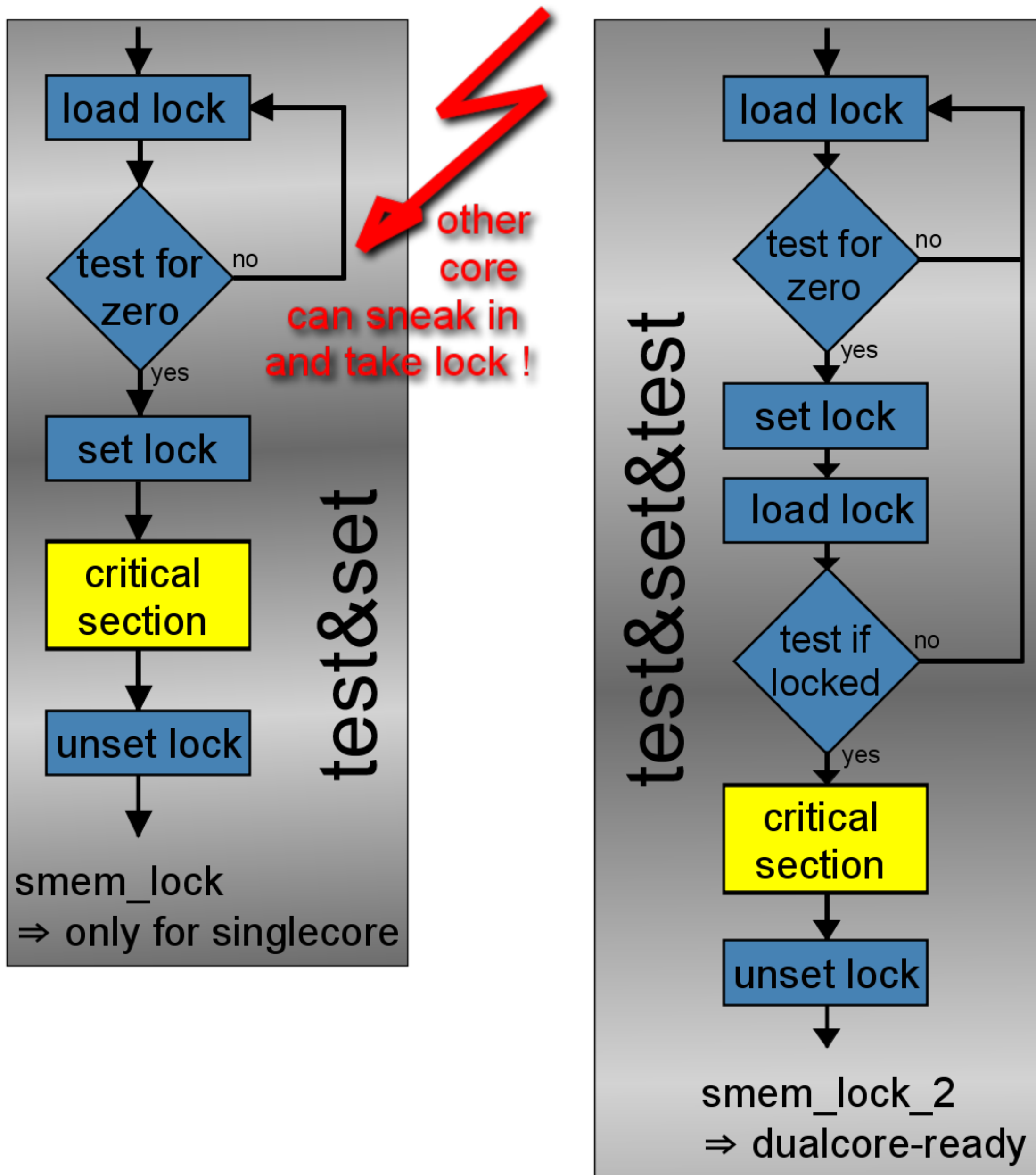


Figure 3.6: From non-coherent test&set to coherent test&set&test

the two cores never wait for each other forever during normal operation. A *livelock* cannot occur due to the impossibility that the cores delay their requests for the lock (together). Without flags we would have to unset the lock we try to set - but that never happens here, one core always proceeds. *Starvation* is prevented by cores dominating the lock implicitly (or better unwillingly) deferring to the later arriving other core, which overwrites the lock shortly after the dominating core.

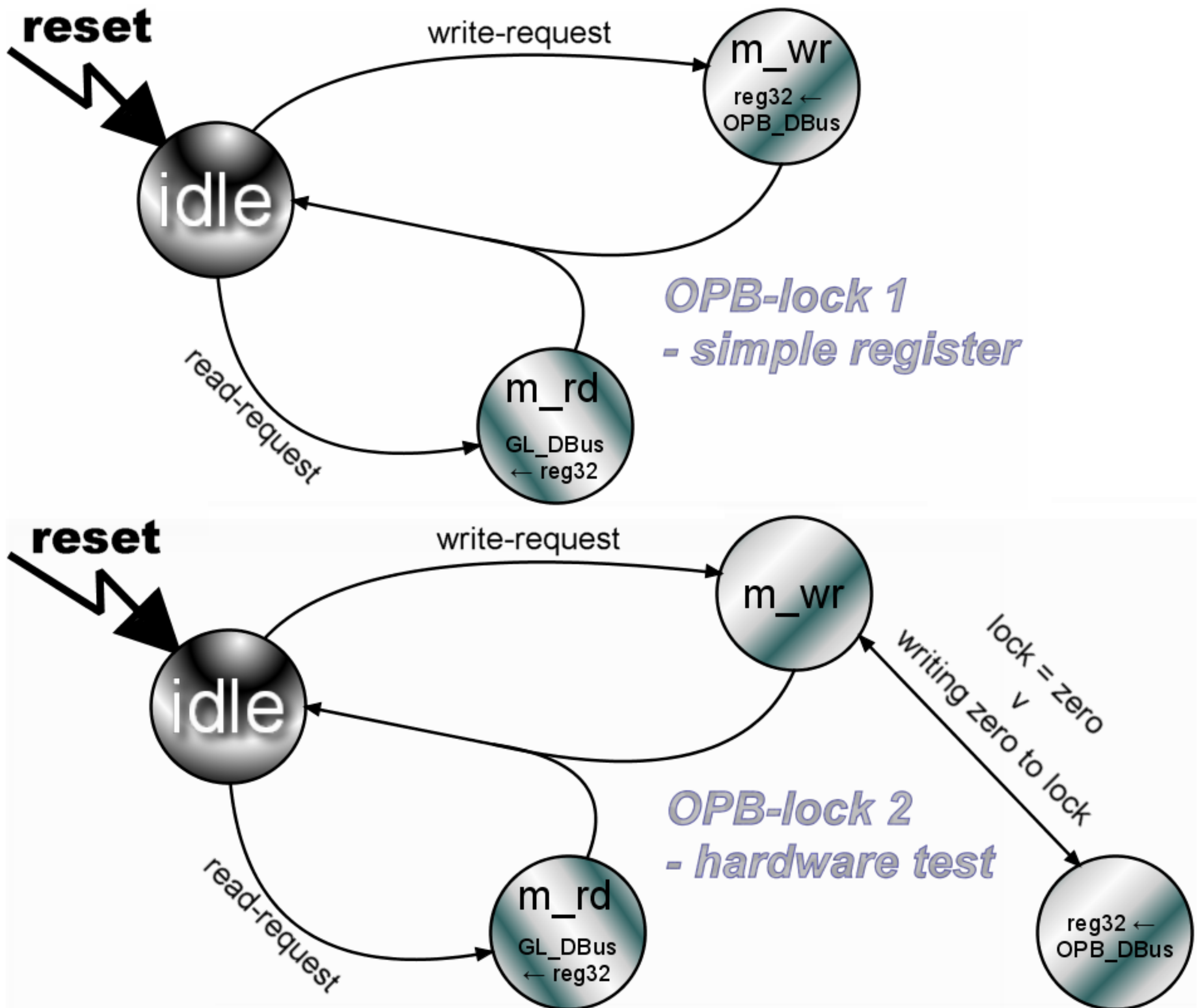


Figure 3.7: State machines of both OPB-locks 1 and 2

3.4 External Bus-Slaves as Locks

Up to now we simply used a word of the shared memory as lock, but this has the disadvantage that we cannot transfer any functionality of the locking-process from the assembler-routine into the hardware. With the two busses (*peripheral local bus (PLB)* and *on-chip peripheral bus (OPB)*) available we can add *user-logic modules* and make them *slaves* of one of those two busses (details on flow and busses in appendix A). So we source out our lock and look how to optimize it along the way.

At first we chose the OPB to work with: it is simpler and involves less overhead than the PLB. The OPB requires less compulsory ports in the VHDL-*entity* we must write for the Xilinx peripheral wizard to define our OPB-*slave*.

A simple *32-bit OPB-register* was implemented - it fits the size of a word in our PowerPC architecture and the size of the lock in shared memory. Hence with this OPB-register we can use the same assembler locking routine `smem_lock_2` by just changing the *destination address* for the lock to that of the OPB-slave. This address is used in our VHDL-entity to react to requests for access. The user-logic of the *OPB-lock* is quite simple, its functionality is drafted in figure 3.7.

Logically, the next step is to add hardware-support to improve performance. With our lock clinging to a bus it gets only one request at a time. Unfortunately this narrows down our possibilities to add hardware-support considerably. What we can do is to source out the first test of the *test&set&test*-routine into the hardware, making it obsolete in the software-routine. That results in a *set&test*-routine where the first action is to try to write the own lock-value to the lock. Our OPB-lock sets the lock not with every write (as a *dump* memory cell does), but only if the register - the lock - is actually equal to zero or there is zero on the data-bus (clearing the lock). With this improvement we establish *strict coherence* since the *first* arriving store irreversibly sets an empty OPB-lock until release (writing zero). This is always under the assumption the OPB is not reordering the requests for access to violate the temporal order they were issued. The state-machine of this more sophisticated *OPB-lock2* is shown in figure 3.7 as well.

The last test for the value of the lock could only be eliminated if setting the lock would **block** the caller until it has the lock, resulting in a simple *set* instead of the original *test&set&test*. Spinning could be avoided completely. Even though the OPB and the PLB have support for blocking the sender (but with time-limits), there is no way for a slave on the bus to block the bus-logic and, in parallel, serve another request. That would be necessary to release the lock while blocking the request to set it, clearly impossible with a serializing bus. Hence this one test in hardware seems to be the only support we can realize using slaves on a bus.

The two versions of the OBP-lock were translated to the PLB, requiring more compulsory ports and minor modifications due to the PLB-protocol (s. fig. 3.8).

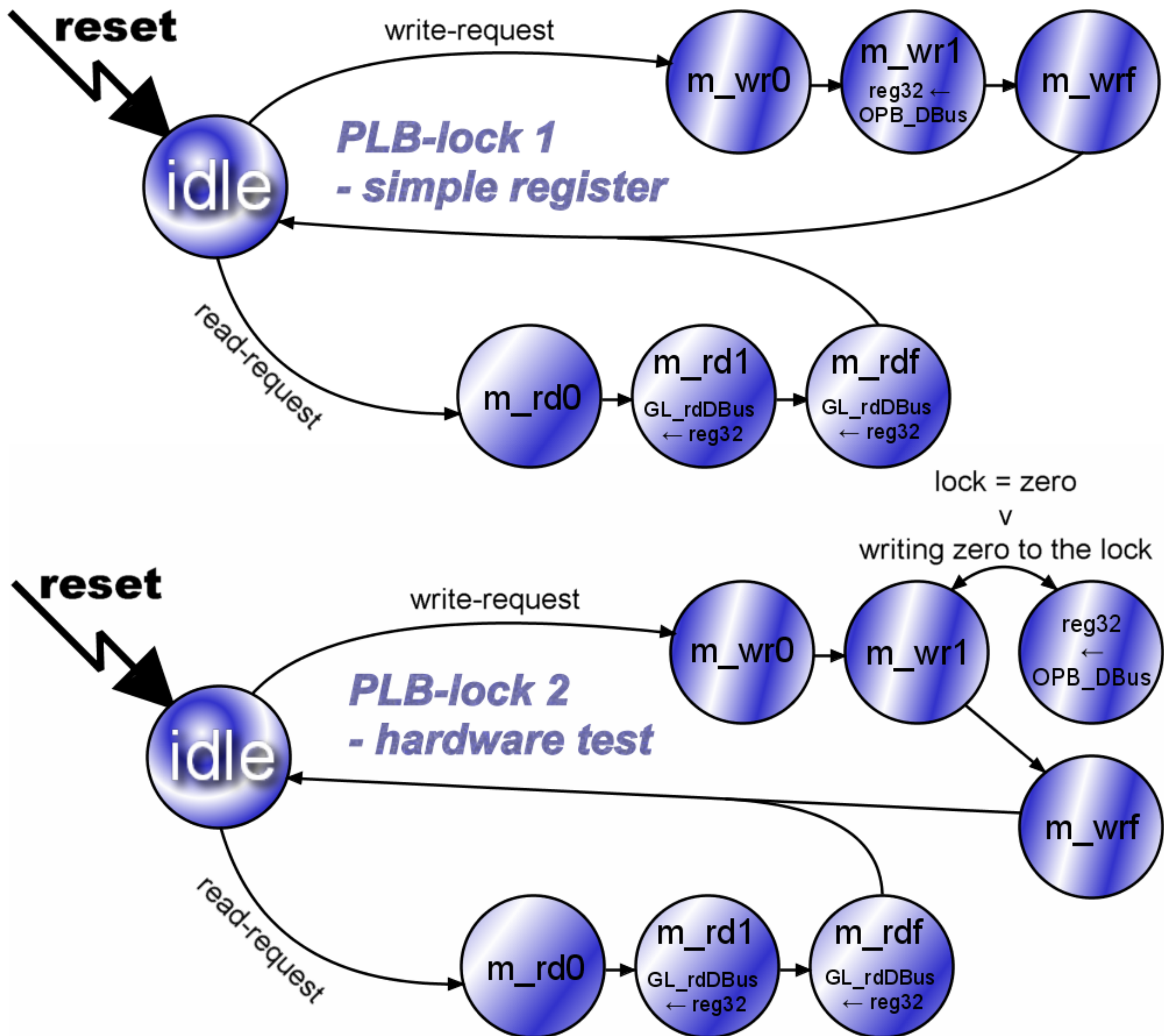


Figure 3.8: State machines of both PLB-locks 1 and 2

3.5 Event Synchronization

Also without hardware support there exists the need to coordinate the execution of intra- as well as inter-core processes. Concurrent processes can make use of a lock to meet at some point in their execution (spinning locks are common practice to achieve point-to-point event synchronization, see also section 2.1.12).

An *asymmetrical* form of a *primitive barrier* using *busy-wait* was used in the early stages of this thesis. One core must reach the point of synchronization prior to its opponent. Since this scheme is intolerant with violations of this fixed order of execution, a more sophisticated *symmetrical* form was developed and is used in the final programs wherever no hardware-support for synchronization can be used.

Both flows of synchronization are shown in figure 3.9.

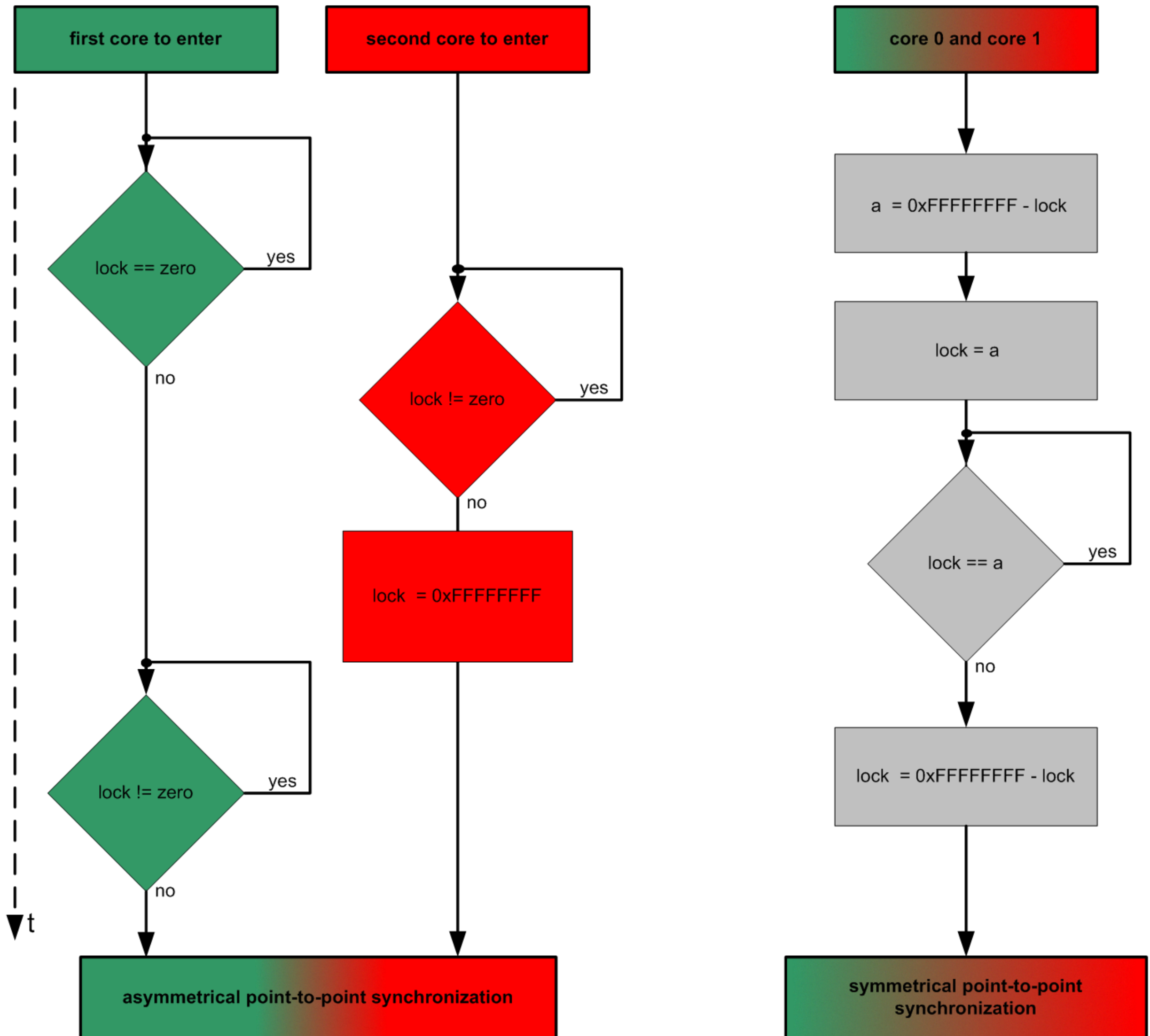


Figure 3.9: Dual-core event synchronization with busy-waiting

It must be added here that the *tightness of the synchronization* differs slightly in theory as well as in practice (depending on some factors like bus-latencies, contention etc.). Despite the fact experimental evidence always showed the synchronization to suffice in practice, some form of timing requirements may need guarantees (upper bounds) for the synchronicity. Software-synchronization like the one here using busy-waiting can never guarantee the tightness of synchronization that hardware-mechanisms can offer (s. chapter 4 for hardware-barriers).

3.6 Locking Performance

Five different locks were introduced for locking the access to the shared memory:

- a word of the shared memory serves as lock
- one simple OPB-lock, one OPB-lock2 with hardware-test
- one simple PLB-lock, one PLB-lock2 with hardware-test

The simple OPB- and PLB-locks are unlikely to bring an advantage in performance - on the contrary: it should worsen since they are connected to the busses instead of using the fast OCM. However, the hardware-enhanced versions bring one improvement: the routines for locking are shorter (*set&test* instead of *test&set&test*, s. section 3.4). Bringing it more to the point, performance of locking depends on:

- time to execute the locking routine without accessing the lock
- time to load and store the lock

The total execution time of the locking routine comprises both aspects.

3.6.1 Direct Access to the Lock

To examine how much impact the usage of the PLB- and OPB-locks has we measure their *closeness* to the processor cores by just accessing them: table 3.1 shows the number of cycles needed to execute only one store- or load-instruction.

	BRAM	PLB	OPB
load/store	2	≥7	≥12

Table 3.1: Time for direct access to different lock-locations [clock-cycles]

3.6.2 Locking, Best Case

Table 3.2 results from summing up the clock cycles for all the single operations in the assembler source code, confirming our suspicion about using external locks: locking by the PLB-locks is slightly worse, additionally crossing the PLB-OPB bridge accessing the OPB-locks halves the locking-performances - and we are still looking at the contention-free best-case where one core locks exclusively.

Lock-location	BRAM	PLB2	PLB1	OPB2	OPB1
Locking	33	35	46	45	74
Temporal locking-order		*		*	
Unlocking	12	13	13	20	20

Table 3.2: Theoretical *best case* (contention-free) locking results [clock cycles]

3.6.3 Locked Single Access, Best-, Worst- & Average-case

Here we eventually access our shared memory using the different types of locks at hand. We also look at the *worst-case (WC)*, using assembler-routines programmed exclusively for that purpose. The structure of the code executed in this worst-case is drafted in table 3.3.

core 0	1
POINT-TO-POINT SYNCHRONIZATION	
LOCK	LOCK
store to BRAM	load from BRAM
UNLOCK	UNLOCK
LOCK	LOCK
store to BRAM	load from BRAM
UNLOCK	UNLOCK
LOCK	LOCK
...	...

Table 3.3: WC single locked accesses - code structure

Of course in the contention-free case locking happens immanently. But in the worst-case the whole locking procedure is prolonged by unsuccessful locking attempts that results in jumps back to the first test in the *test&set&test* - this is called **spinning**. An additional counter counts up each time there is a jump back.

Spinning (busy-waiting) is the reason for the exploding overhead under bus-load - this can easily be observed experimentally in our scenario: one core writes, the other core reads repeatedly. Each single access is made atomic by a lock-unlock pair. The spinning for some cases is shown in table 3.4.

Ones more the best-case results confirm the delays introduced by using the busses OPB and PLB. Even worse in the worst-case our hardware-enhanced locks lead to *starvation* - indicated by the infinity-symbol in table 3.4. The first core to get the lock keeps it. The other core starves and can do its work only after the other core stops taking the lock as demonstrated in fig. 3.11, leading to unacceptable delays for the starving core. The other types of locks do not show that negative effect.

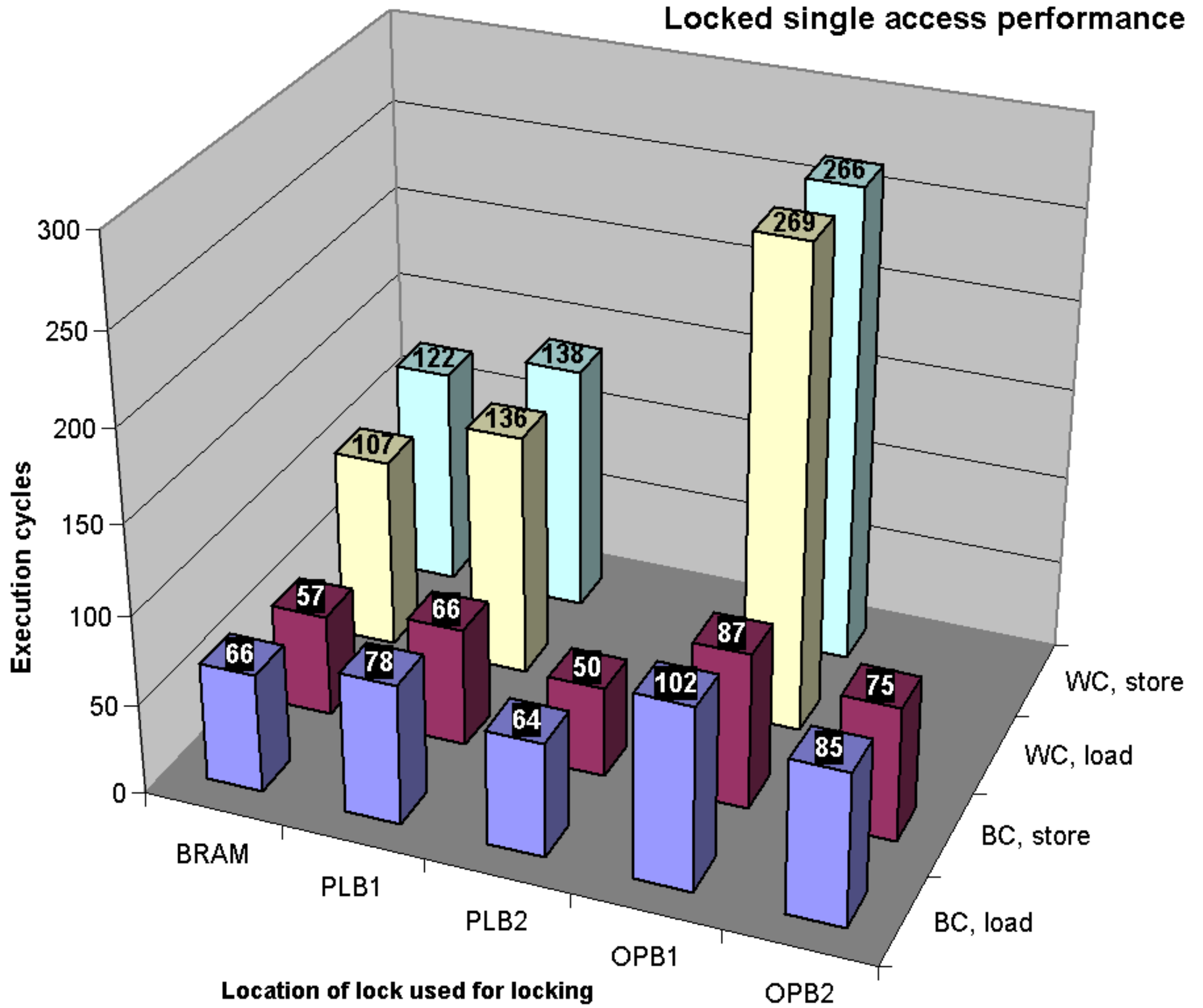


Figure 3.10: Single access performance using spinning locks (no values: starvation)

Lock-type	BRAM		PLB1		PLB2		OPB1		OPB2	
Locked load‖*store*	ld	st	ld	st	ld	st	ld	st	ld	st
BC, ticks	≤66	≤57	≤78	≤66	≤64	≤50	≤102	≤87	≤85	≤75
BC-spins	0	0								
WC	≥107	≥122	≥136	≥138	(65)	∞	≥269	≥266	(109)	∞
WC-spins	10	16								

Table 3.4: Spinning and locked single-access performance

The starvation is caused by the strict temporal ordering of locking that is not present with the locks using only a memory-cell or register, they allow an arriving core to overwrite the lock set just shortly before (*race condition*). With our hardware-enhancement this is not possible: in the WC (LOCK immediately following UNLOCK, s. table 3.3) the dominating core has two store-instructions succeeding in the code and sets the lock again before the other, currently spinning core checks that the lock is free and can inject its own store (shown in table 3.4). The probability of breaking that starvation corresponds to the likelihood of the SET in the *set&test*-routine of the waiting core to be applied exactly one cycle (issued one cycle earlier) after the UNLOCK of the core currently holding the lock (values taken from measurement and structure of assembler-routines):

$\mathbb{P}\{\textit{set\&test} \text{ of waiting starving core sneaks in}\} =$

$$= \mathbb{P}\{\text{store in } \textit{set\&test} \text{ at starving core}\} \cdot \mathbb{P}\{\text{UNSET at dominating core}\} =$$

$$= \frac{\text{store in } \textit{set\&test} \text{ routine}}{\text{whole routine}} \cdot \frac{\text{UNSET executing}}{\text{whole LOCK-read/write-UNLOCK}} = \frac{12}{45} \cdot \frac{11}{59} = \frac{132}{2655}$$

$\Longrightarrow \mathbb{P}\{\text{starving core can get the lock}\} \approx \underline{5\%}$

Hence - *statistically* - it holds that every *twentieth* access the starving core can steal the lock from its dominator. But since our probability calculus is based on uniform distributions that is just theoretic and too optimistic. Our tests revealed that in reality it is commonplace that in a row of 200 locked accesses the dominating core gets never interrupted - with the other core starving beyond all its deadlines.

Starvation can be easily *broken* by inserting (artificial) delays between successive locked critical sections. As a rule of thumb the relation of the time for a critical section (**LOCK** $\rightarrow$ **UNLOCK**) to the time between two critical sections (**UNLOCK** $\rightarrow$ **LOCK**) should be at worst

- 1 : 1 for the PLB-lock2
- 1 : 2 for the OPB-lock2

to avoid starvation. With this *relaxation of the WC* the usage of the hardware-enhanced locks seems safe enough for certain kinds of applications, but it must be pointed out that it is **not the worst-case** anymore. The greater the inserted delays the more we detach from the WC towards the *average case (AC)*.

Summing up, the WC-behaviour is certainly all else but ideal and kind of relativizes the performance improvement we achieved by hardware-support for the best and average case. For a non-real time system this may be acceptable, for safety-critical systems, in all probability it is not. How to attack the roots of this starvation- and fairness-problem and also better performance is shown in chapter 4.

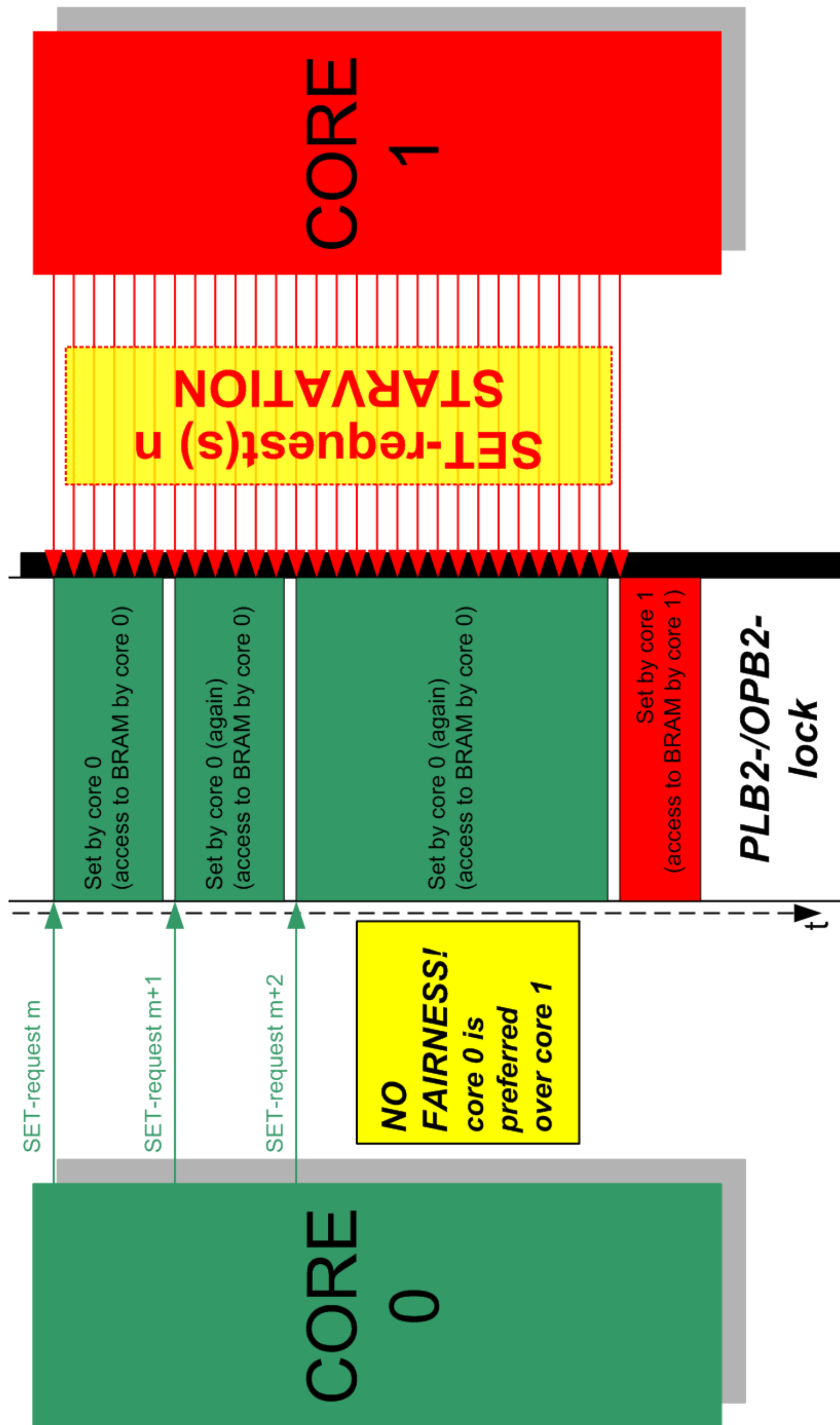


Figure 3.11: Unfairness in relaying requests for the lock causes starvation: stores to locked lock are ineffective

Chapter 4

Hardware Synchronization

This chapter starts with analysing the disadvantages of pure software synchronization (see chapter 3). After identifying the roots of the problems efficient hardware-solutions are developed to overcome them. Facilities for event-synchronization are implemented in hardware as well, allowing for tight point-to-point cooperation and hence the uncomplicated implementation of automatic testing.

The level of details provided in this chapter should suffice to get an extensive insight into the core subject of this thesis - which is the implementation of synchronization primitives in hardware. Please note that the appendix comprises no source-files - in the case of interest please contact the author or his instructors. However, the abstract descriptions in this chapter should provide any reader with the means to implement similar techniques and methods without the sources.

4.1 The Problems

Software synchronization is accompanied by the following severe problems:

- **no strict consistency**
  No temporal consistency complicates handling real-time systems.

- **unfairness**
  The multiple processor cores are not treated equally.

  - **starvation** (worst unfairness)
    A processor core may wait for a resource indefinitely.

Next we look deeper at the software locking mechanisms used in chapter 3.

4.1.1 PLB- and OPB-Lock, no Hardware Support

With a simple register on the bus the outcome of the race for this simple lock has a *random factor*. The factor is due to the possibility that a request for the lock may *overwrite* it shortly after the lock has already been set by another core. The previous core looses the race. A possibly present discrimination caused by the bus is weakened by this randomized race for the lock, resulting in a smoothed service ratio over time (average case). ⇒ POSITIVE: *no starvation* occurs due to this effect. ⇒ NEGATIVE: *no strict temporal order* in servicing the locking-requests.

4.1.2 BRAM-Lock, no Hardware-Support

With the lock located in the block-RAM we use no external busses (PLB, OPB) and the cores issue their requests for the lock (that is, the memory cell) directly through the two ports of the dual-ported memory. Still the *same race condition* as for the simple PLB- and OPB-locks without hardware-support prevents temporal consistency to be present. ⇒ NEGATIVE: *No temporal consistency.* Due to the better performance of the BRAM in comparison with the busses, fairness is *smoothed* even better than using the IBM-busses. ⇒ POSITIVE: *No starvation.*

4.1.3 PLB- and OPB-Lock, Hardware-Support

With the hardware-test for zero the lock ignores all requests for locking when it is set: the lock stays locked by its first locker until its release (s. figure 3.11). ⇒ POSITIVE: assuming the requests for the lock are relayed in temporal order, the lock is always set in *strict temporal order* too. ⇒ NEGATIVE: unfortunately *starvation* occurs in the worst-case (s. diagram 3.10), hence we have *no fairness.*

4.2 The Roots of the Problems

The reasons why software synchronization performs insufficiently are:

1. **Spinning** is inefficient (no process-switch or power-down possible)
2. **Lost Control** relaying the service-requests for the lock

When spinning, the processor spends far too much time for *application-remote computations* instead of accessing the shared memory as intended. To achieve worst-case fairness the two cores would have to keep and exchange *statistics* about their respective accesses to the lock. Since this cooperation itself needs synchronization it becomes clear that additional help for the software is desperately needed - software alone must spin to set a lock and ensure that it is set properly.

Using the external busses we face the problem of *uncertainty*: the bus takes the requests from the processor cores and relays them to its slaves (here the locks). What exactly happens with the store- and load-commands is up to the interna of the busses and is out of the designers immediate control. Figure 4.1 demonstrates the uncertainty added by handing over service requests to an external bus acting as a relay. For instance, there are *fixed- and dynamic-priority schemes* available for the IBM busses PLB and OPB. Depending on the priority scheme used completely different performance-results are possible - even starvation.

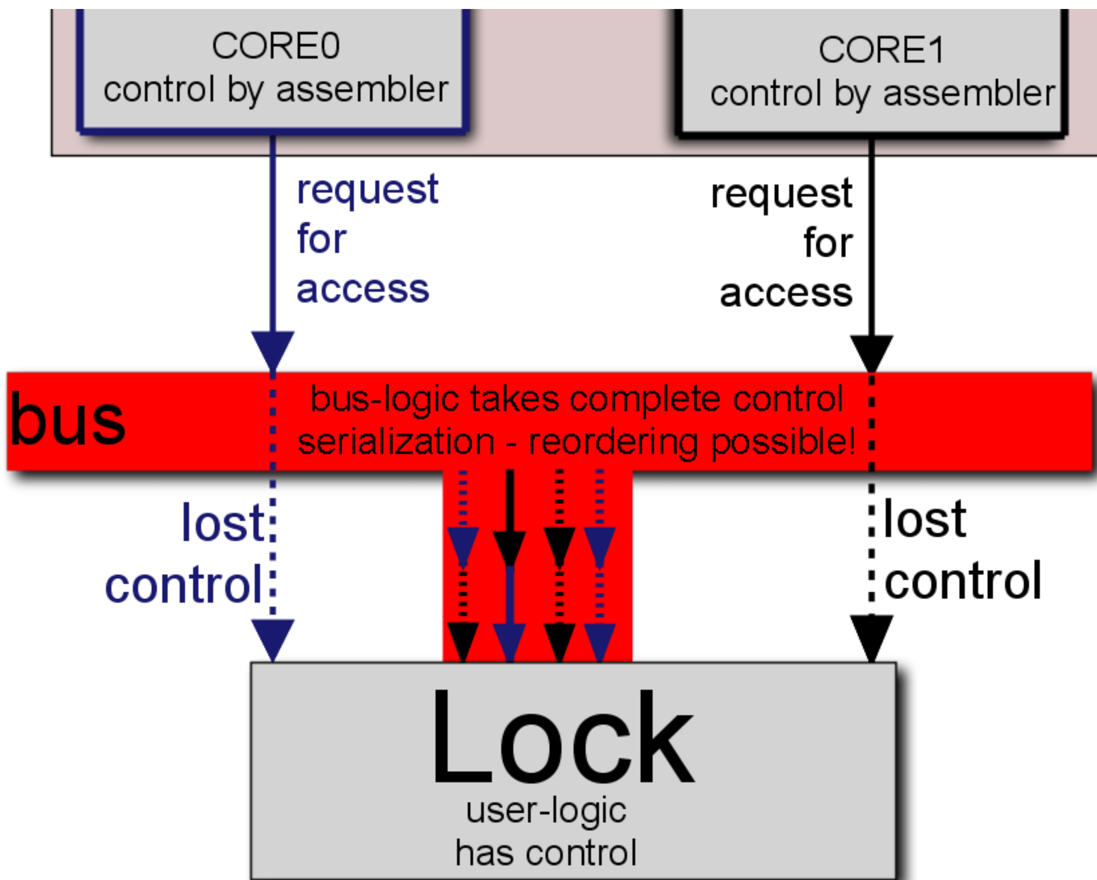


Figure 4.1: Broken chain of control over accessing bus-slaves

4.3 From SW- to HW-Synchronization

Clearly it is desirable to keep control over the path of requests for the lock from their being issued in the program until their service in hardware. To gain such a predictability of execution the hardware must be fully transparent. This is the reason why we abandon working with external busses in this chapter and concentrate on the on-chip block-RAM that enables full concurrency for two processor cores by its dual-ported architecture.

The two main problems of software synchronization are eliminated by

1. **Blocking** instead of spinning
2. **Pure user-logic** chain of control (request $\leftrightarrow$ service)

No bus is shared by the processor cores, therefore no serialization is forced on the service-requests of the cores $\Rightarrow$ *true concurrency is possible.*

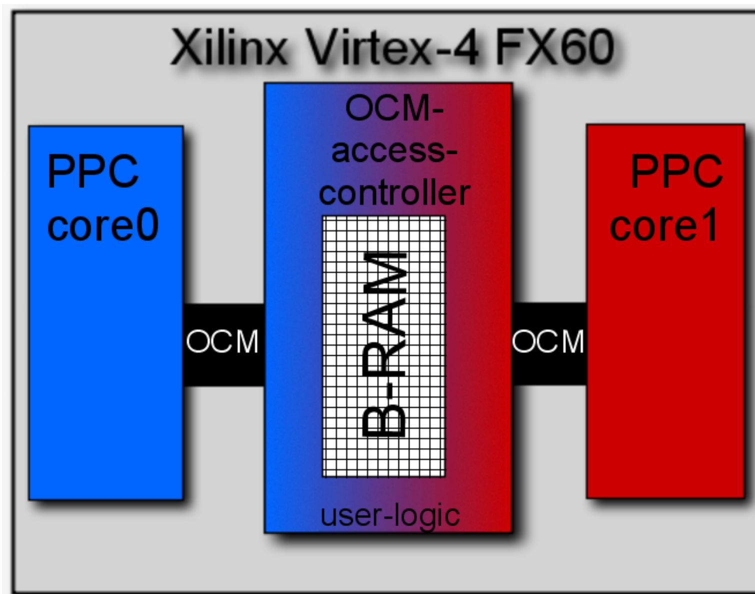


Figure 4.2: The OCM-access-controller takes control over the BRAM

User-logic is needed to surround the BRAM and take requests for the shared memory in order to coordinate their execution to ensure consistency. Using the digital flow of the appendix A user-logic could be inserted successfully by means of a new logic module called the *OCM-access-controller* (s. figure 4.2). The controller completely isolates the BRAM from the cores, takes all access-requests for the shared memory and handles them according to its programmed functionality.

Still there is the question on how to let a processor core wait that issues requests to the shared memory. Here the Xilinx Virtex-4 family of FPGAs offers a solution: blocking of a core is possible by delaying the acknowledge-signal for a core's OCM-controller (s. [Xil07f] for details on the OCM-protocol). Using this feature we can get rid of the busy-waiting and thus all the software-overhead for locking. One single store- or load-instruction is now enough to access the shared memory, consistency and synchronization as a whole are delegated to and handled by the new hardware unit controlling the access to the shared memory.

4.4 The OCM-Access-Controller

Figure 4.3 shows a rough draft of the OCM-access-controller. The controller takes requests for the shared memory (BRAM) and displaces them in time - if necessary. It must be noted that this delay is in the magnitude of the clock itself and is therefore incomparable with the big overhead involved using software synchronization.

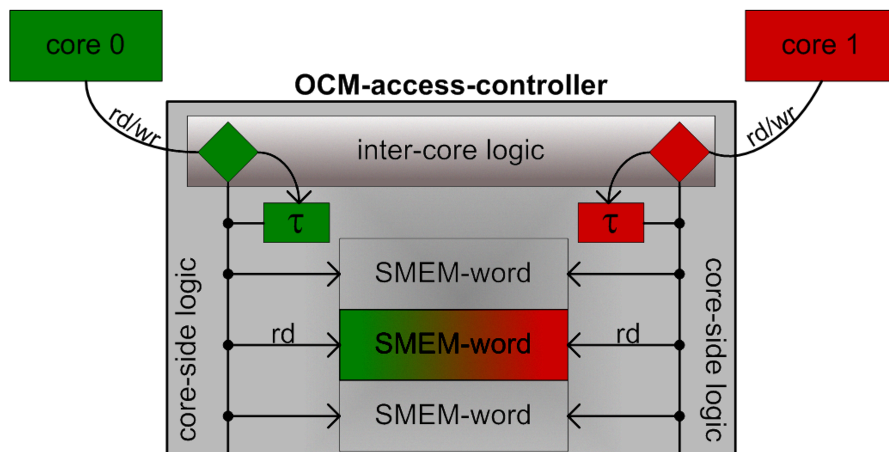


Figure 4.3: The OCM-access-controller delays requests for the BRAM if necessary

Also, as is indicated in figure 4.3, concurrent reads can be executed in parallel - if allowed and wanted. Not allowing concurrent accesses at all results in a *serialization*, obviously eliminating the advantage of multi-ported memory.

4.4.1 Features

The OCM-access-controller is finalized in two versions. The general framework, the version 4.0, offers a generic approach to the problem where the number of processor cores involved in competition is scalable. This was confirmed by simulations comprising test benches with 4 cores.

For the case of having two cores the theoretical testing (means simulation) was complemented by intensive practical testing, resulting in the specialized dual-core version 3.16.

The features of the OCM-access-controller in the finalized versions 4.0 and 3.16 are the following:

- Data Synchronization
  - **Implicit locking** of single accesses to the shared memory
    * Single load/stores do not need explicit locking
    * Reads are executed concurrently
  - **Global locking** of all of shared memory
    * by setting/resetting a global locking bit
    * by loading a counter with the number of accesses
  - **Address-sensitive locking**
    * regions in shared memory are lockable (WORD-granularity)
    * every processor core can lock one region at a time
    * other cores cannot access such regions until release
    * address-conflicts with already locked regions prevent/delay locking
    * address-comparisons may bring up timing issues
- Event Synchronization
  - **Simple Barrier**
    A minimum of two cores meet at a simple hardware barrier
  - **Extended simple barriers**
    (v4.0 only, limitation to 32 cores)
    A core can wait for some other cores in particular
  - **Complex barriers**
    (v4.0 only, no core-limitation)
    Each core specifies the ID of the barrier to be used and the number of other cores to wait for
- Set of special-purpose configuration registers
- Status-information for debugging the hardware
- Variable register-file for application-specific purposes (optional)

4.4.2 Description and Programming

Here the features presented in section 4.4.1 are explained in more detail, complemented by a rough description of how to use them concretely in practice.

Coherence Ensurance - Implicit Locking in Hardware

In this thesis the term *coherence ensurance* in a broader sense means all work done regarding the atomicity of **single accesses** (to the shared memory). As is explained in chapter 3, with no low-level coherence, explicit software locking is necessary (see figure 4.5). Hence the first goal was to eliminate this significant overhead and relocate the locking into the hardware.

With this feature the memory can be accessed by applications as if the glue logic was not present. The comfort in using this feature and the gain in means of less software-overhead is demonstrated in the code-comparison of figure 4.6.

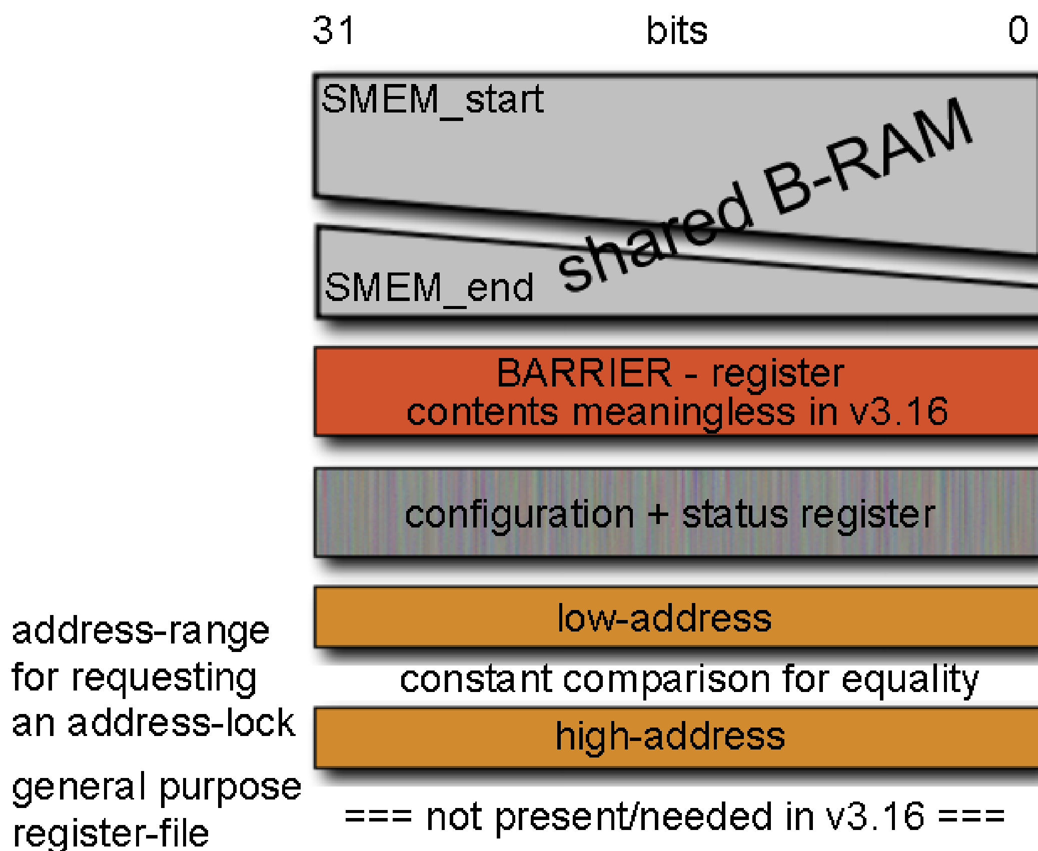


Figure 4.4: Core-side addressing of the OCM-access-controller

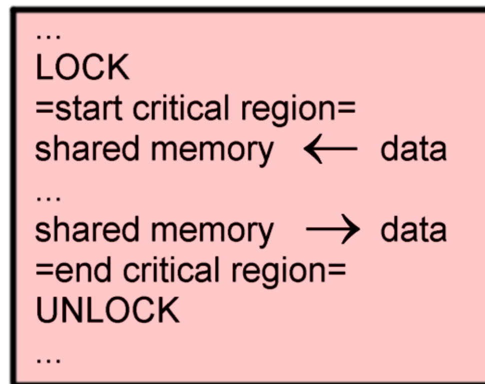


Figure 4.5: Explicit locking, principle

Consistency Ensurance - Explicit Global Locking in Hardware

With *consistency ensurance* all matters regarding the atomic execution of **groups of accesses** is concerned. To realize this it is necessary to realize a possibility to lock the shared memory globally - this must be done before the groups that are related are executed. Figure 4.7 shows global locking in principle (realizable easily by software locking) and more concrete for our OCM-access-controller.

The configuration register must be programmed to activate the global locking capability. That special-purpose (SP) register was added as an additional logic to the end of the address range of the shared memory. Figure 4.4 gives an overview over the registers that are accessible by the user. The hardware is generically adapting to varying sizes of the shared memory, hence the software can use the fixed offsets to the registers and there is no need to severely modify the sources.

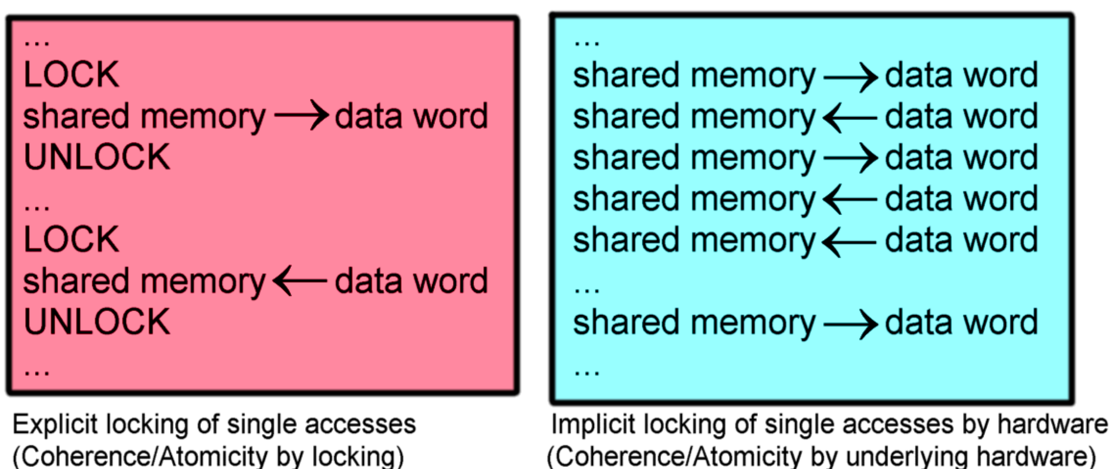


Figure 4.6: Pseudo-code of explicit software vs. implicit hardware locking

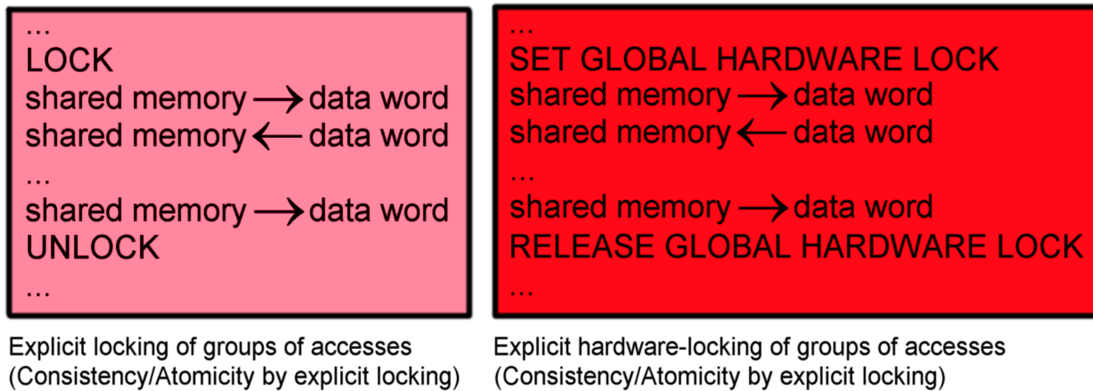


Figure 4.7: Pseudo-code for global locking

Requesting the global lock of the controller can be done by bit *1* or by writing the number of accesses wanted to lock globally into the global locking counter (bits 24 to 31 for the dual-core version 3.16). Those two methods are equivalent and trigger the same race for the global lock as described in section 4.4.5. As soon as the current core has obtained the global lock this is indicated by bit *2*.

Figure 4.8: Control- and Status-register of the OCM-access-controller

Address-Sensitive Locking of Regions in the Shared Memory

Previously it was described how accessing the shared medium is made *exclusive* in hardware - implicitly or explicitly. Despite the improved efficiency the processor cores are still waiting for each other - the inter-core access is *serialized*. Especially in the real-time sector blocks of data tend to stay relatively small in comparison with data-intensive applications, making it quite inefficient to lock the whole shared memory for each small block of data globally. That seems to be an overkill.

Here a new concept complementing the so-far exclusive techniques of access is introduced. Having costly multi-ported memory (dual-ported in our case) we would like to use this advantage, but allowing concurrent accesses through the different ports of the shared memory we once again face the danger of corrupting data written by multiple sources. Looking on it in more detail, the case of corruption only happens in case the destination addresses of concurrent accesses collide. With at least one of the accesses being a write such a case leads to scrambled bit patterns read or even stored in memory as described in chapter 3. Hence the addresses of accesses to be executed in parallel must be compared against each other.

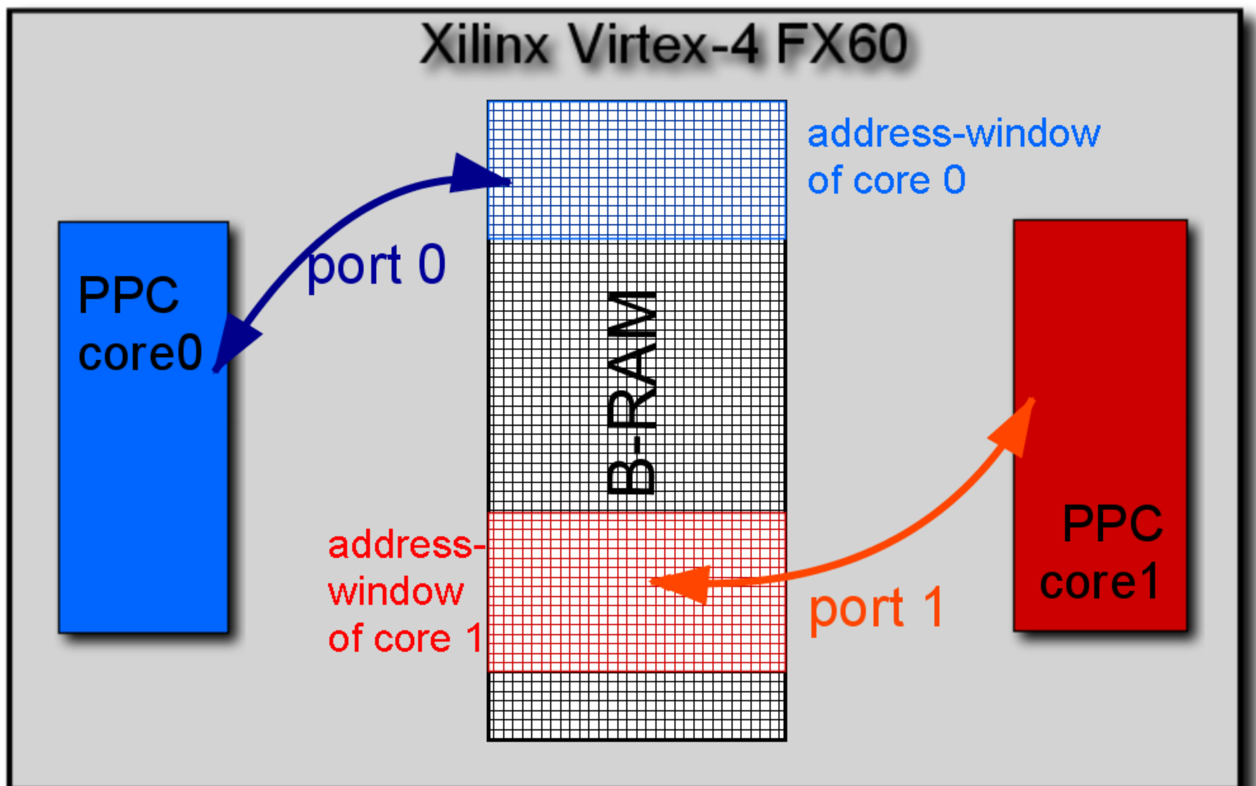


Figure 4.9: Concurrent access by disjunctive shared memory windows

These considerations lead to *address-sensitive* logic. Up to now the address played no major role besides the addressing of controller-internal registers.

The idea here is to partition the shared memory into topologically coherent chunks of data that are defined and described by their highest and lowest addresses. Such a window into the shared memory is subsequently called an *address-window* or an *address-region*. With the hardware supporting the inter-core protection of such address-regions the concurrent access to the shared memory becomes possible - illustrated in fig. 4.9. The hardware-protection makes sure that all cores own non-overlapping address-windows of the shared memory. To keep the functionality as straightforward as possible one core can reserve only **one window at a time**, but then it is the exclusive owner of that region of memory until releasing it.

Due to the definition of *atomicity* no other core can access a once reserved address-window of a core until it is set free again, also single accesses must wait. However, added address-sensitive signals enable single-accesses of other cores to *sneak in* into regions of memory not covered by locked address-windows. It is vital here to understand that global locking and the hereby described address-locking are totally *mutual exclusive* due to their very nature:

```
If any processor core holds the GLOBAL LOCK
all other cores must wait no matter what they request for,
and vice versa:
If at least one core owns a locked ADDRESS-WINDOW
no other core can acquire the GLOBAL LOCK
but
other cores may lock address-regions as long as they do not
overlap with already established regions.
```

In any case it holds that with the occurrence of a conflict the processor core that issued its request *later* has to wait and is *blocked* in the mean-time.

Contrary to the global lock that is shared among an arbitrary number of processor, now every core can hold a lock. Ideally all cores would hold such an *address-lock* for one address-region, and all regions would be distinct. Practically such a case will not always occur due to the need to **exchange data** between the cores.

Allowing the processor cores to reserve more than one address-region at a time would explode the constant effort needed for inter- and then also intra-core address-comparisons. More precisely, a naive approach needs to compare $\binom{n*m}{2}$ pairs of address-windows with at least 2 comparison each (n ... number of processor cores, m ... max. number of address-locks per core).

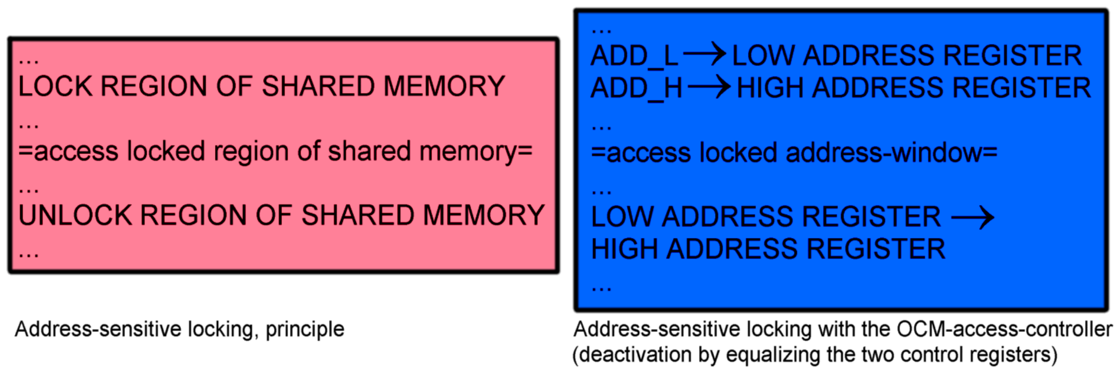


Figure 4.10: Pseudo-code for locking an address-window

Address-sensitive locking is deactivated as long as there is the same address-value in the two address-registers (s. fig. 4.4). Inserting two different values in the two registers corresponds to issuing a request to lock the memory-region between those two addresses (s. fig. 4.10). As soon as a request arises this is indicated by bit *4* in the *control- and status register* shown in figure 4.8. Bit *5* is set when the address-locking was successful, the respective core has its address-window defined by its two address-registers reserved and therefore has exclusive access to it.

Event-Synchronization by Hardware-Barriers

An efficient and comfortable point-to-point synchronization is realized by means of three different types of hardware barriers, making locking in this matter obsolete.

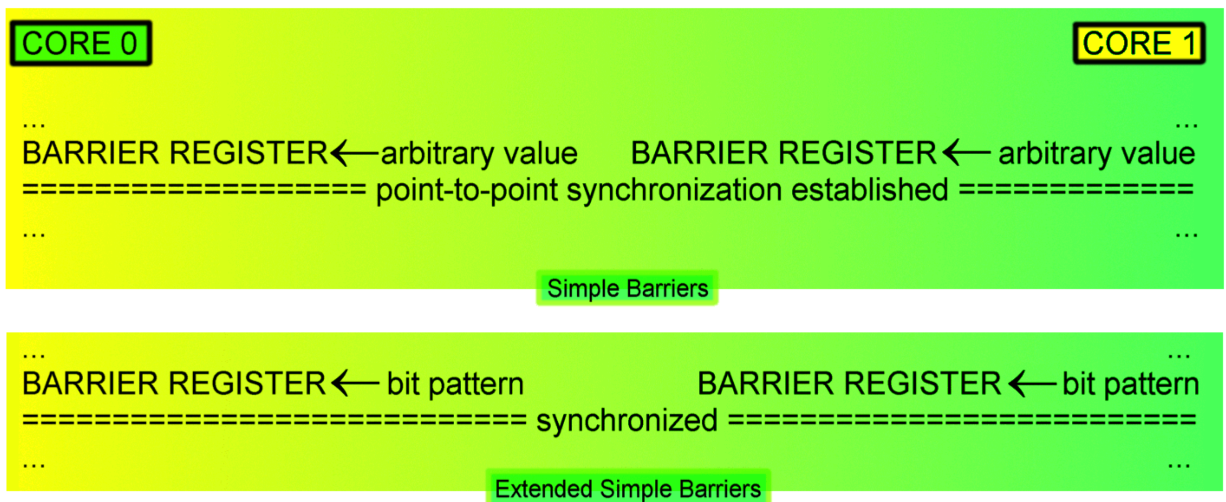


Figure 4.11: Event-synchronization by hardware-barriers

Simple barriers are the easiest method: each core that wants to meet at a given point of execution writes an arbitrary value to the barrier register shown in figure 4.4. Then the respective processor core is blocked until at least one other core writes to its corresponding counterpart-register. Please note that if the writes are buffered an additional read from the register/address is needed to lead to blocking of the processor core (or the read alone instead of the write).

With more than two processor cores in the system, *extended* simple barriers offer the possibility to define exactly for what other cores to wait for. Each bit in the register corresponds to the fixed number of a processor core in the system. The drawback is that the numbers of the cores must be known at compile-time, this may not be possible for dynamically executing systems.

Figure 4.11 demonstrates the practical usage of simple and extended simple barriers. As hardware locks the hardware barriers get rid of the time- and space-consuming overhead coming along with pure software solutions (s. chapter 3).

A more flexible replacement for the extended simple barriers are *complex barriers*. They allow a more hardware-remote level of programming: the numbers of the processor cores must not be known in advanced, only the number of other cores to wait for, making it unimportant on what processor core the program is eventually executed. Due to the additional level of complexity a more detailed description of complex barriers can be found in section 4.4.8.

Additional Status Information and Functionality

For debugging some additional bits were added to the register *0* shown in figure 4.8: these additional status bits can bring light into the question if there is another core currently holding the global lock (bit *3*), if address-sensitive locking is done by another core in the system (bit *6*) or if there is an address-conflict preventing the locking-request for an address-window to succeed immediately (bit *7*).

Finally there is the possibility to *bypass* the OCM-access-controller completely and access the shared memory directly by setting bit *0* of the control register. It is in the responsibility of the programmer to reactivate the controller before accessing its internal registers again, to avoid unpredictable behaviour.

4.4.3 Structure and State Machine

In this and the subsequent sections more in-depth information about the controller is presented. The main underlying methods realizing the features explained in section 4.4.2 are covered here in more detail. This section should be informative enough to understand and possibly copy or adapt the principles and techniques developed and used in this thesis. In the case that questions remain still unanswered after reading this section please feel free to contact the author.

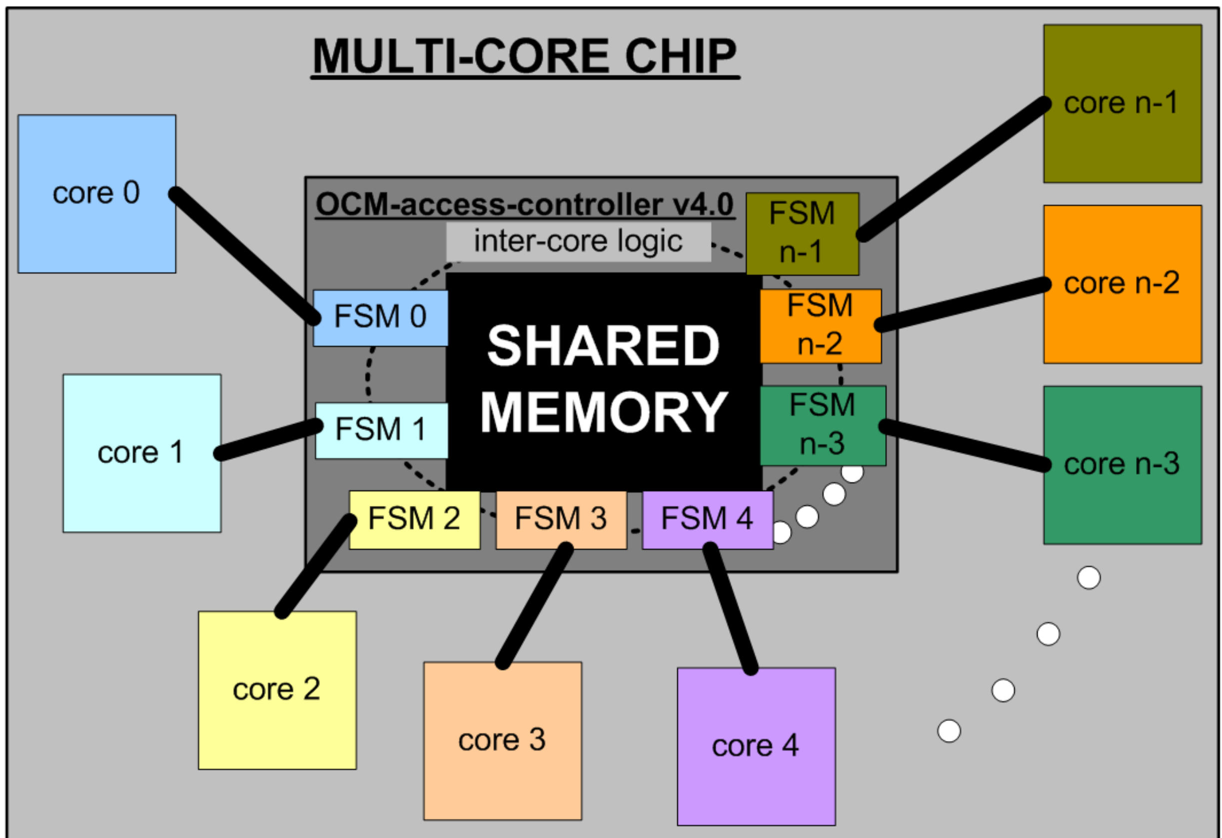


Figure 4.12: Abstracted schematic of OCM-access-controller v4.0 for multiple cores

The OCM-access-controller consists of *core-side* and *inter-core logic* as demonstrated in figure 4.12. The former handles the interfacing to the respective core and is essentially a state machine that made quite an evolution from being a *Mealy* to becoming a more stable mainly *Moore*-based automat having dedicated state - flip flops for all critical output signals. In this thesis only the final outcome is of importance, the previous development stages are not covered.

Figure 4.13 shows the state machine of the specialized dual-core version 3.16, figure 4.14 the generalized version 4.0 providing all functionality developed in this thesis.

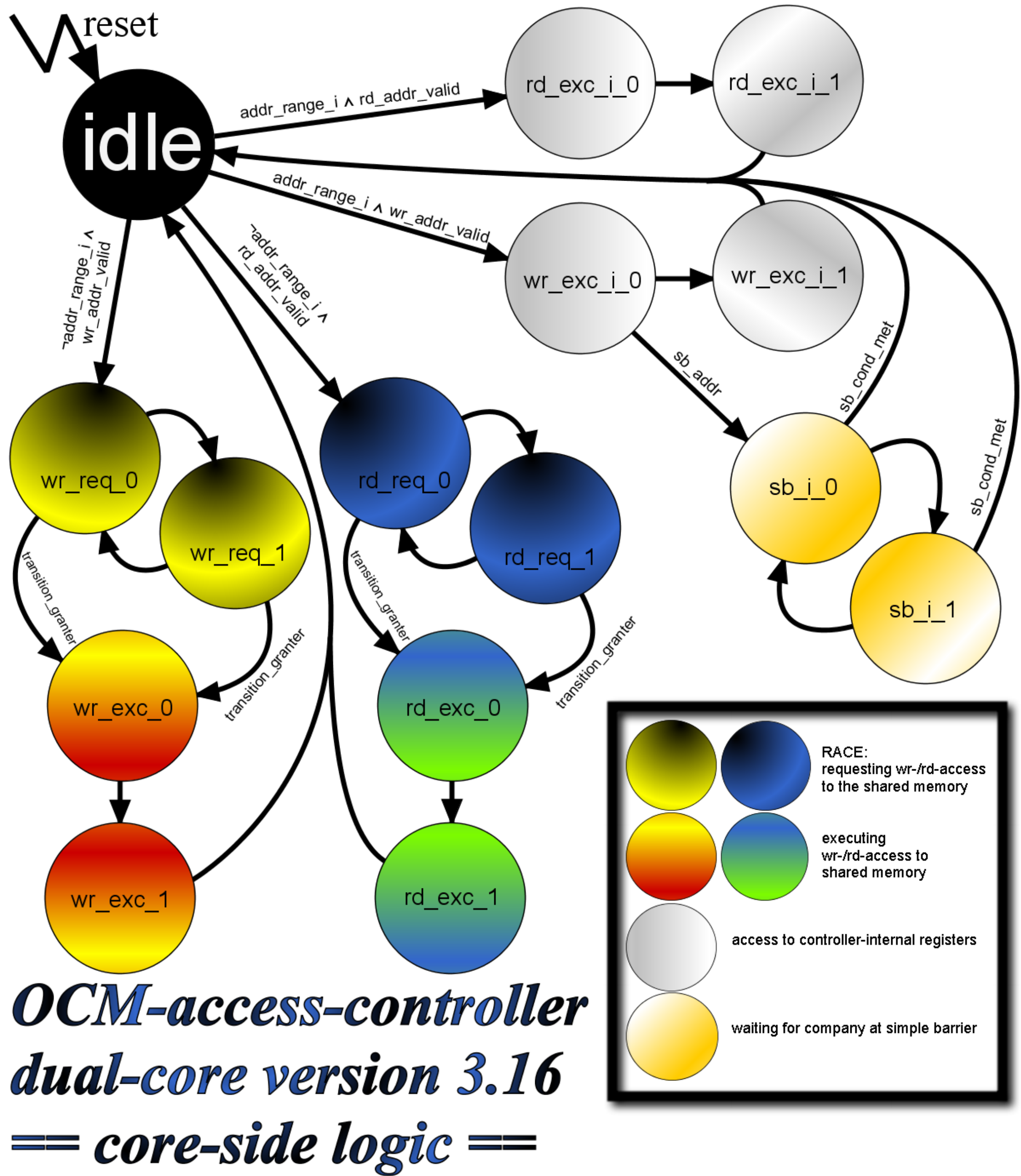
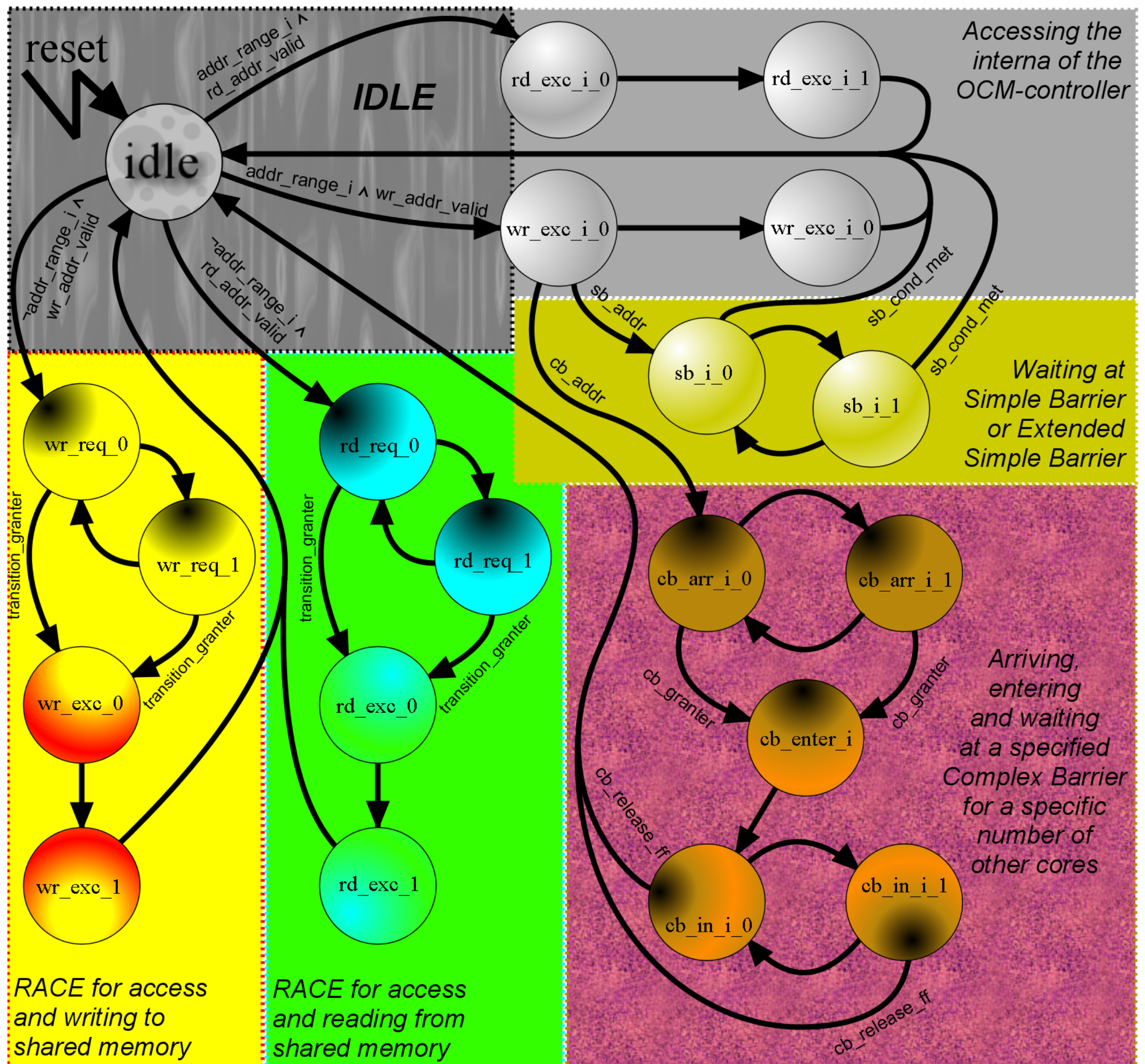


Figure 4.13: Core-side state machine of OCM-access controller v3.16

Details on the specific functional parts can be found in the subsequent sections.



OCM-access-controller
multi-core version 4.0
== core-side logic ==

Figure 4.14: Core-side state-machine for the beyond-dual-core case (v4.0)

4.4.4 Coherence Ensurance - Implicit Locking in Hardware

Single accesses to the shared memory that need not be grouped with other accesses suffer most from the need of *explicit locking*. Explicit locking costs a multiple of the actual access time to shared memory, hence dramatically worsening performance.

Existing examples of hardware descriptions handling an arbitrary number of participants in a race for a shared medium share a common factor: at least one *variable* that makes handling the generic number of competitors possible. It was the author's ambition to find a mechanism that allows an arbitrary number of participants to compete in a race against each other - but without variables which are somewhat notorious for the level of abstraction they add to hardware designs, easily leading to unsynthesizable or very badly synthesized hardware. (The recommendation to avoid variables is given, amongst others, in [Ska99]. The author himself made practical experiences with the problems variables in HDL-code can pose during an internship at a leading semiconductor company). Therefore, to avoid uncomfortable abstract variables in the underlying VHDL-code a *variable-free* but still unboundedly *scalable procedure* with respect to the number of cores has been established, organizing and controlling the *race* for the shared memory.

The first approach granting only one core to access the shared memory at a time was quite simple: a *ticket* was provided by a ring-register with only one bit set. This *ticketing*-scheme works quite well in the dual-core case, but it is as less scalable as it is **inefficient**: cores that do not want to access the shared memory but hold the ticket waste time that could be used by cores waiting for access. Hence this first approach (version *2.x* of the OCM-access-controller) was not pursued for long. In order to find an **efficient** mechanism to resolve concurrent requests for our critical resource the *requirements* for such a technique are the following:

- **Efficiency:**
  - only *competitors* (cores requiring access) attend the race and can become the *winner*
  - the race itself must not consume a big amount of *time*
    ⇒ ideally the race elects one winner per clock cycle which is the case with the mechanism presented here
- **Fairness:**
  - no competitor must wait *indefinitely* to get access
  - each competitor is served in a *finite* amount of time
    ⇒ ideally the worst-case waiting time (in clock cycles) is bounded only by the number of processor cores in the system (this holds with the mechanism presented here)

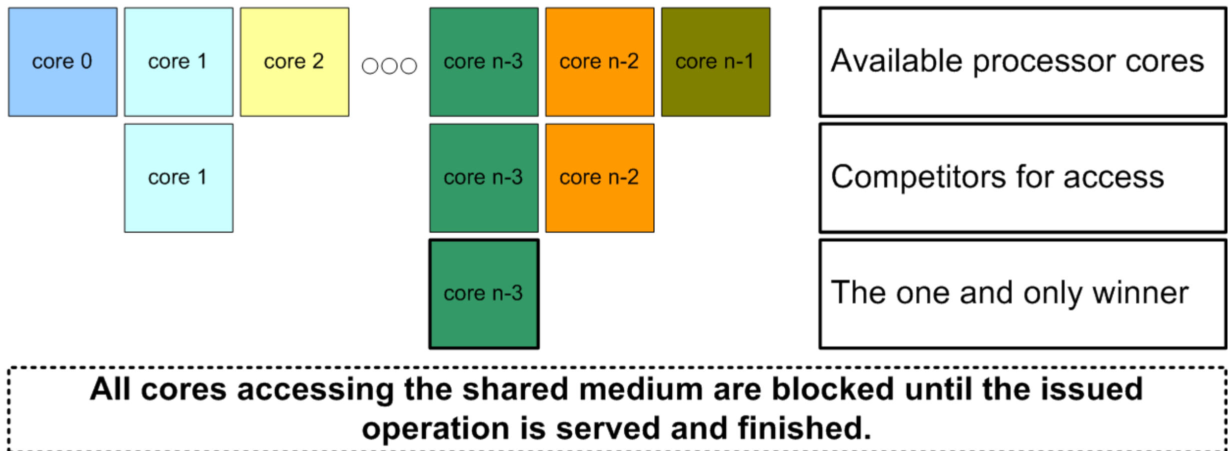


Figure 4.15: Principle of a fair and efficient race for access

Figure 4.15 shows the main *principle* behind the idea for an efficient and fair race: only one of the competing cores wins the race and may access. Now the question is how the race can resolve a winner of an unbounded *arbitrary number of competitors*, and ideally with the *time to resolve the race being a constant*, independent of the number of competitors. Looking at the worst case for such a race as shown in fig. 4.16 makes it more apparent: the election of the winner must not depend on the number of cores, or we have a practically non-scalable system that will destroy the advantage of having multiple processor cores at hand.

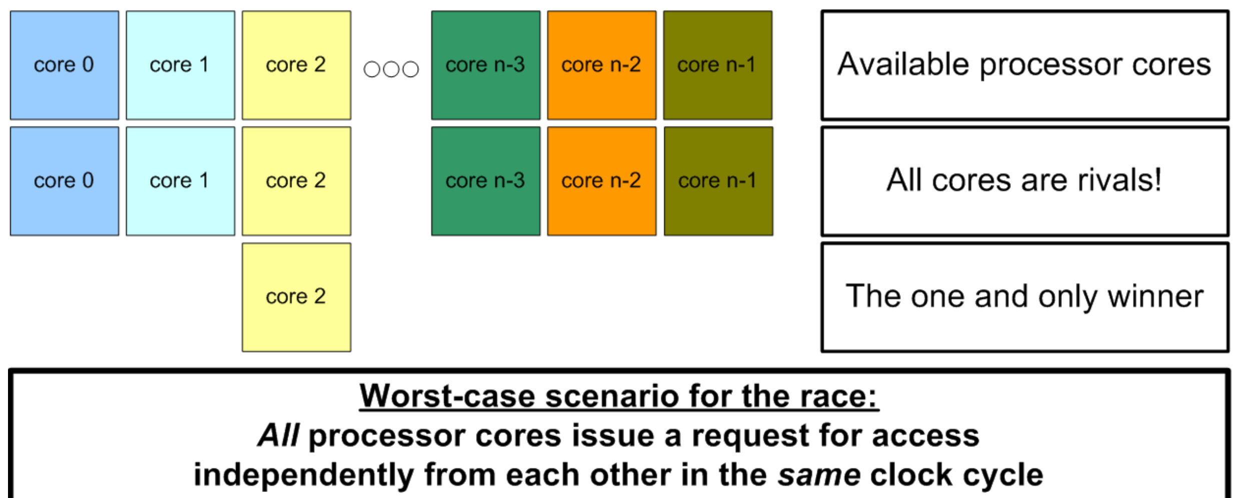


Figure 4.16: Worst-case race

A first idea would be to give each core a *unique priority*, numbering the cores depending to this priority. With this scheme it would be simple to get the winner of a race: the core with highest priority wins. But in the case the highest-priority core enters the race repeatedly shows that there exists no fairness: *starvation* of lower-priority cores can occur. Table 4.1 shows a possible unique prioritization of processor cores, with figure 4.17 indicating an efficient implementation.

bit-pattern	priority
1000..0000	highest priority
0100..0000	2nd-highest priority
0010..0000	3rd-highest priority
0001..0000	4th-highest priority
...	...
0000..1000	4th-lowest priority
0000..0100	3rd-lowest priority
0000..0010	2nd-lowest priority
0000..0001	lowest priority

Table 4.1: Core-Priorities determined by significance of single set bit

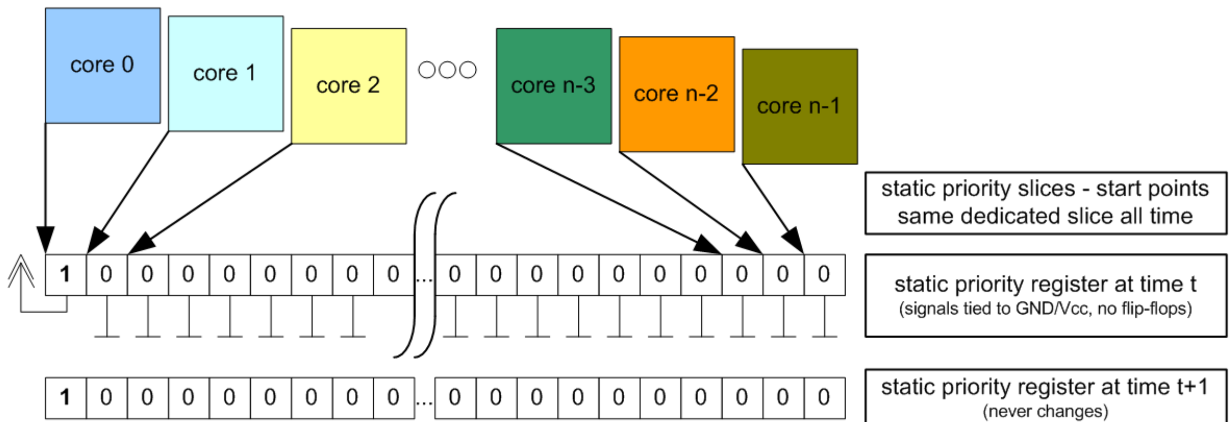


Figure 4.17: Fixed priority scheme: slices of an invariable register

To avoid starvation but keep the simple approach presented so far there is need for modifications. A fixed-priority scheme can bring problems in an SMP (symmetrical multiprocessing system) where probably no processor core is dedicated or special. In such a system there is no need for priorities at all. But using priorities solves our race efficiently. So the question is: what must be changed ?

The solution to this dilemma is to introduce *fairness by variability* into the unfair fixed priority-scheme. We simply *pass on* the *priorities* from one core to the next in an endless loop, with each one of all the cores holding the highest (and

lowest) priority for only one cycle per *period*. The length of this period is exactly the number of cores present in the system. Since the cycling of the priorities is triggered by the system clock itself no other logic can disrupt this scheme: with each core holding the highest priority once per period we have a worst-case delay equal to the period of the cycling priorities of when a core's request is served.

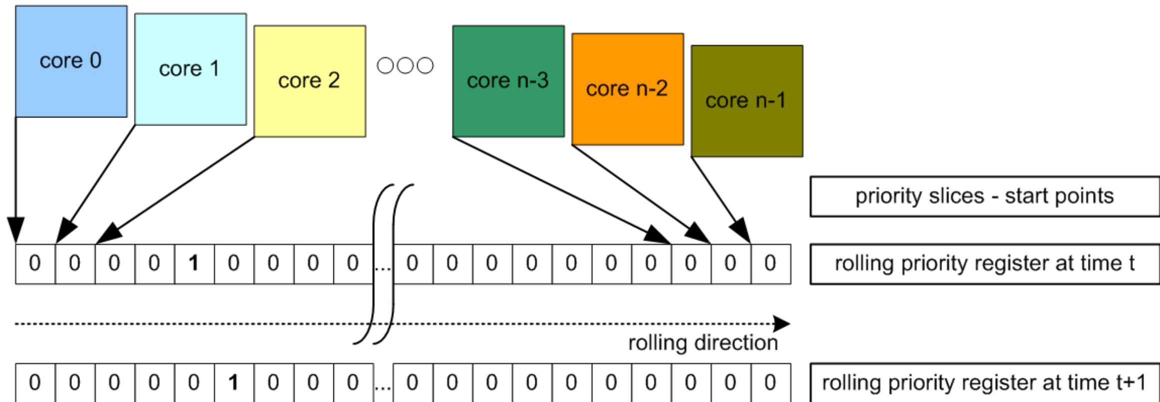


Figure 4.18: Variable Priority Scheme: Slices of a rolling register

Fig. 4.18 shows the implementation of cycling priorities using a *chain of flip-flops* equal in length to the number of cores present in the system. There is only one flip-flop set, all others are cleared. By assigning each core a different *slice* of this register we achieve a very efficient correspondence of priorities to the cores (means minimal hardware-overhead). By *rolling* this register (shifting with LSB/MSB lossless becoming MSB/LBS depending on direction) we can easily avoid problems with scalability: rolling is simply done by passing on the value hold by any flip-flop to the next in row. Hence rolling is done in one clock cycle independent of the number of flip-flops involved in the process, making this scheme truly *scalable*.

The actual race for the shared memory is given in figure 4.19 - slightly abstracted from the corresponding VHDL-code but with the original signal-names on its right column as a reference. The resulting signal-array called *granter_ff* holds transition signals for the finite state machines (s. fig. 4.13, 4.14), allowing only one of all cores to go from the request-stage to actually accessing the memory.

So, basically we implemented a *round-robin mechanism* in hardware: *'time slices'* are given only to cores that demand access to the shared memory. The algorithm always elects a winner if at least one core demands access, there is no idle time in such a case. The race for the memory is fast enough to be executed each clock cycle, yielding a high throughput of access to the shared memory. Scaling measures could involve parallelizing operations (e.g. the linear running OR in fig. 4.19).

With **fairness**, **efficiency**, thus **scalability** and even **predictability** given, our race seems fit for duty in parallel real-time applications.

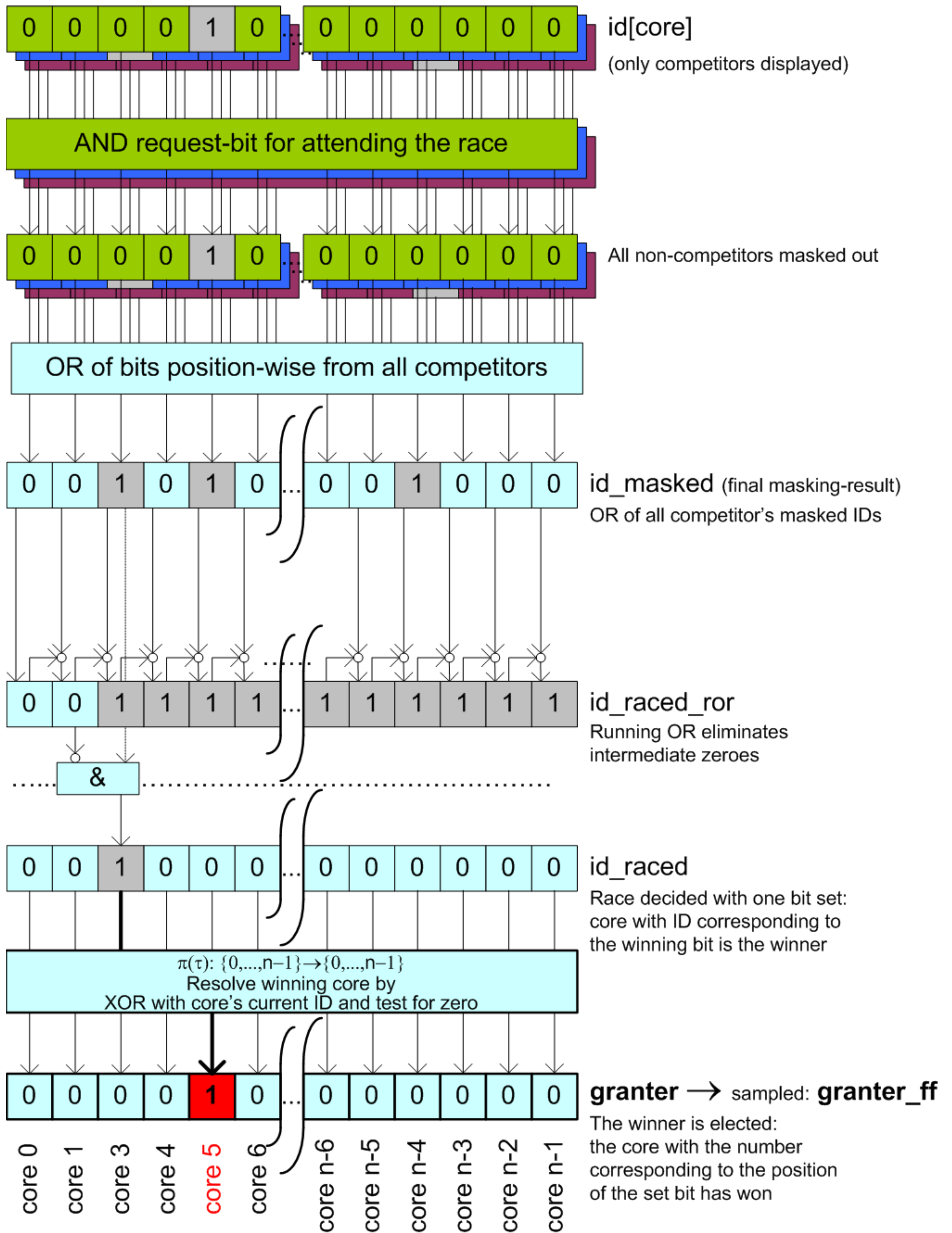


Figure 4.19: Detailed single-access race for the shared memory using core-priorities

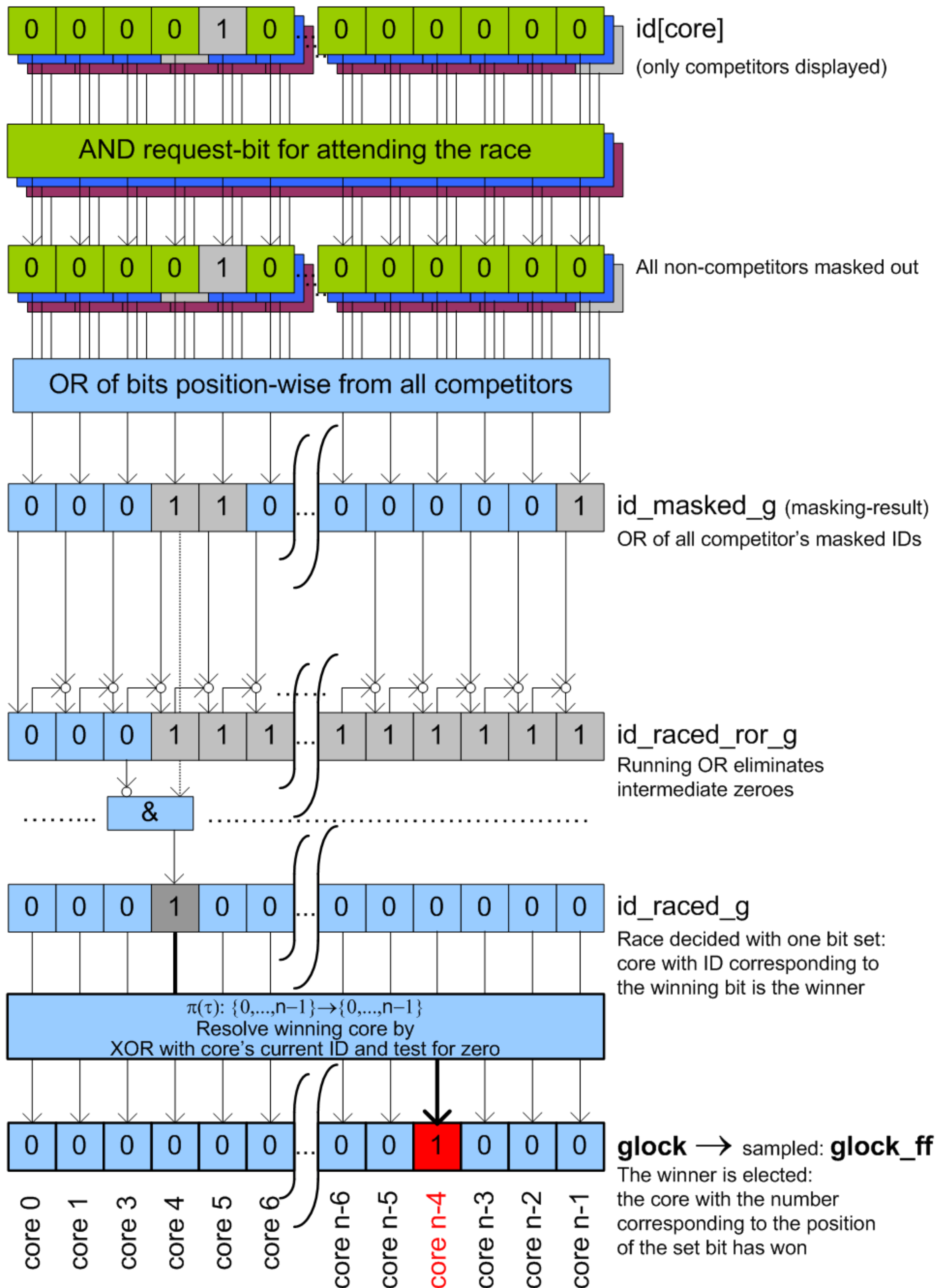


Figure 4.20: Race for the global lock of the OCM-access-controller

4.4.5 Consistency Ensurance - Explicit Spin-Free Global Locking by Hardware

In the previous section 4.4.4 it was explained how an arbitrary number of processor cores compete for the shared memory in hardware. Only single accesses were covered: all cores issue single accesses to the shared memory without any intra-core interdependence between those accesses.

In this section we look at larger amounts of data that must be transferred to or from the shared memory *as a whole*, preventing a violation of the consistency of **interdependent data**. Issuing sequences of single accesses to the memory is just not good enough in this case: the OCM-access-controller may interleave accesses of different cores since there is no information about the wished atomicity of execution. For instance, in the automotive sector blocks of data in the magnitude of a few hundred words must be transferred atomically.

Exclusive access to the shared memory must be demanded *explicitly*. The OCM-access-controller provides a control bit in its control register that must be set, resulting in all subsequent accesses to the shared memory to be executed exclusively without interference by other cores. Of course the shared memory must be available, otherwise the first access is delayed until the reservation of the shared memory is done successfully. Ones more a lock shared by all processor cores is subject to a race between all cores requesting exclusive access, only this time this so-called *global lock* is realized and also managed totally in hardware.

Due to the same advantages as the ones stated in section 4.4.4 the same race-mechanism as the one described there was used to handle the race for the *global lock* as well. By setting the request-bit described in section 4.4.2 and issuing a subsequent request for access a core joins the race for the global lock. Figure 4.19 gives an overview of the race single accesses participate in (winner accesses), the similar figure 4.20 shows the race for the global lock (winner owns global lock).

4.4.6 Event Synchronization - Simple Barrier

By writing to a special barrier-register (s. figure 4.4) the core-side state machine enters a cycle of waiting states (fig. 4.13, fig. 4.14). This cycling in the barrier-states is broken by the signal *sb_cond_met*, with it asserted the state machine gets back into the *idle*-state. Now a temporal synchronization is established between the two cores that just met at the barrier. The principle is shown in fig. 4.21. The signal *sb_cond_met* is acquired by an OR of all the other core's information about being in a barrier or not, figure 4.22 gives an idea of the generation. Hence the name *simple* barrier: the barrier reacts to *any* other core waiting at the barrier.

Simple barriers are easily implemented, thus being a cheap asset for any hardware

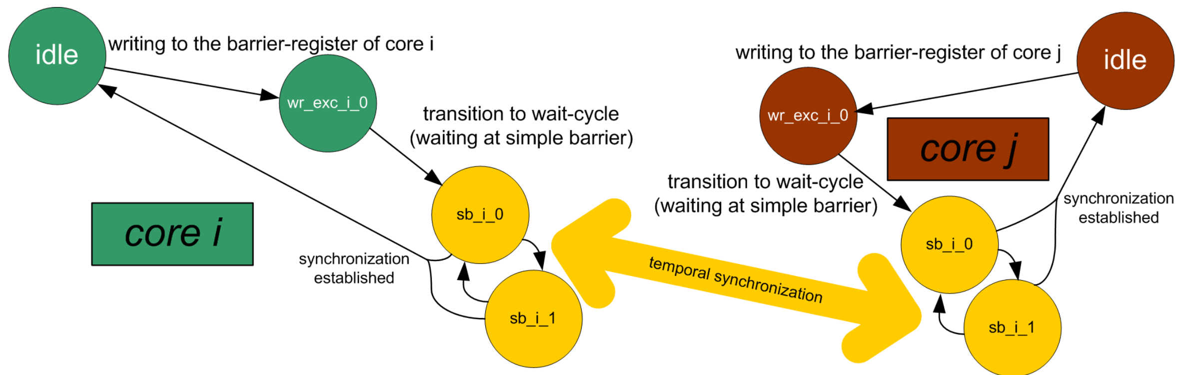


Figure 4.21: Simple Barrier, principle: two cores synchronize temporally

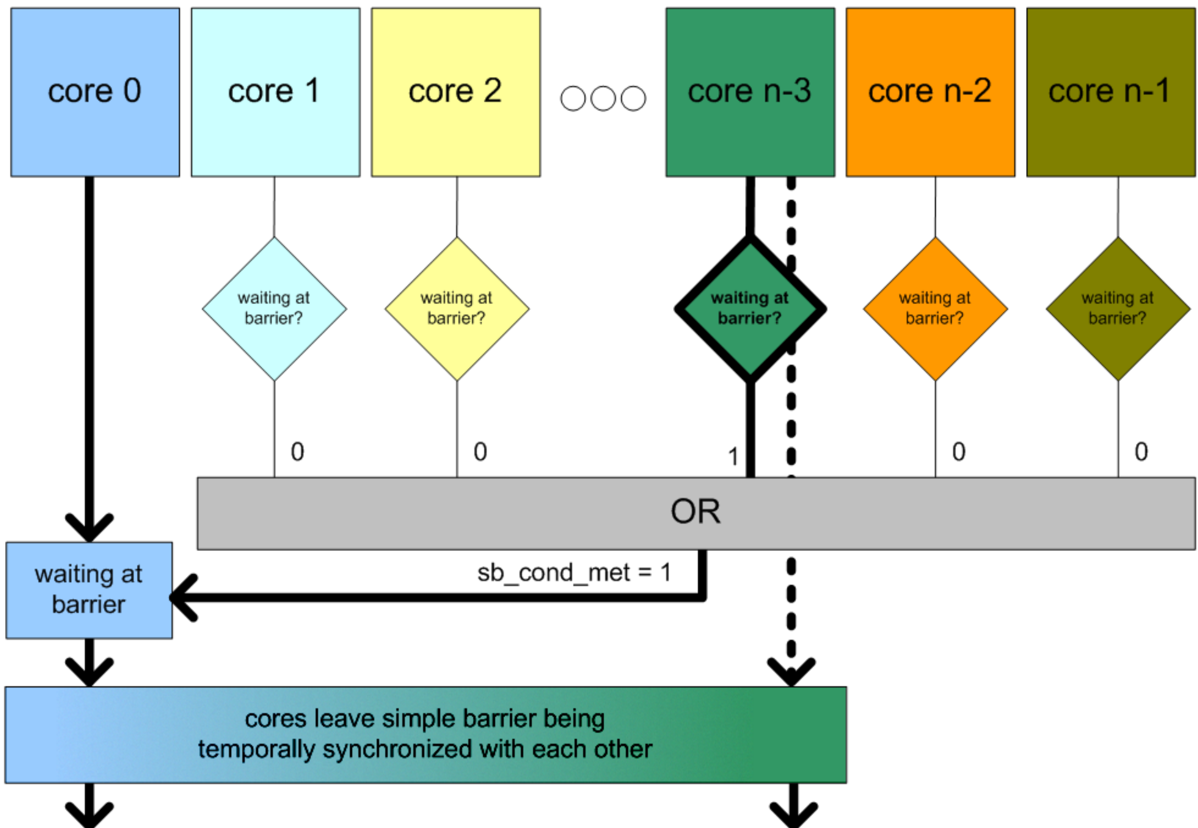


Figure 4.22: Generation of the transition signal *sb_cond_met* to leave simple barrier

synchronization unit. Since in a system with only two processor cores there is only one opponent for each core it makes no sense to have more complicated barriers. Hence such a simple barrier is the only event synchronization feature in the dual-core version *3.16* of the OCM-access-controller (s. state-machine 4.13).

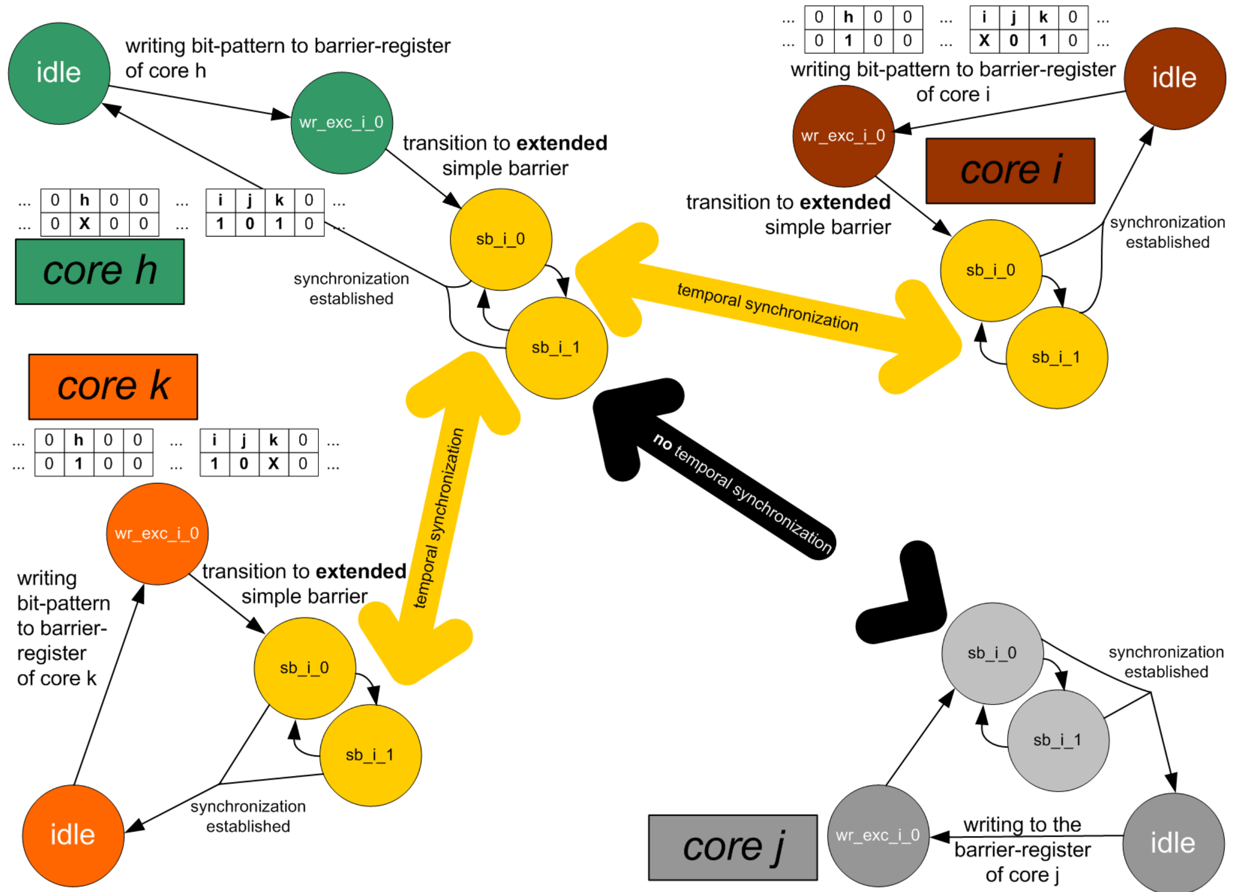


Figure 4.23: Multiple selective synchronization at extended simple barrier

4.4.7 Event Synchronization - Extended Simple Barriers

A downside of simple barriers is their non-scalability. With multiple cores in the system problems using such simple, core-insensitive barriers can occur: by the execution of multiple cores (temporally drifting away from each other) it might happen that the two false cores synchronize at a simple barrier, leading to failure of the subsequently depending operation - or even worse consequences.

For this reason the simple barrier was extended in functionality. Using the already present barrier-register additional information can be relayed to the OCM-access-controller, in particular *for what other cores the barrier should be sensitive for.* The idea was to keep it simple by invariably associating the bits of the register to the processor cores in the system. A set bit tells the controller to wait for that core, a zero tells otherwise. Writing a *zero* alone to the barrier-register results not in waiting for no other core but waiting at a *simple barrier* as described in the last section 4.4.6. So no additional register was necessary.

The structure of the newly established data format can be seen in diagram 4.24.

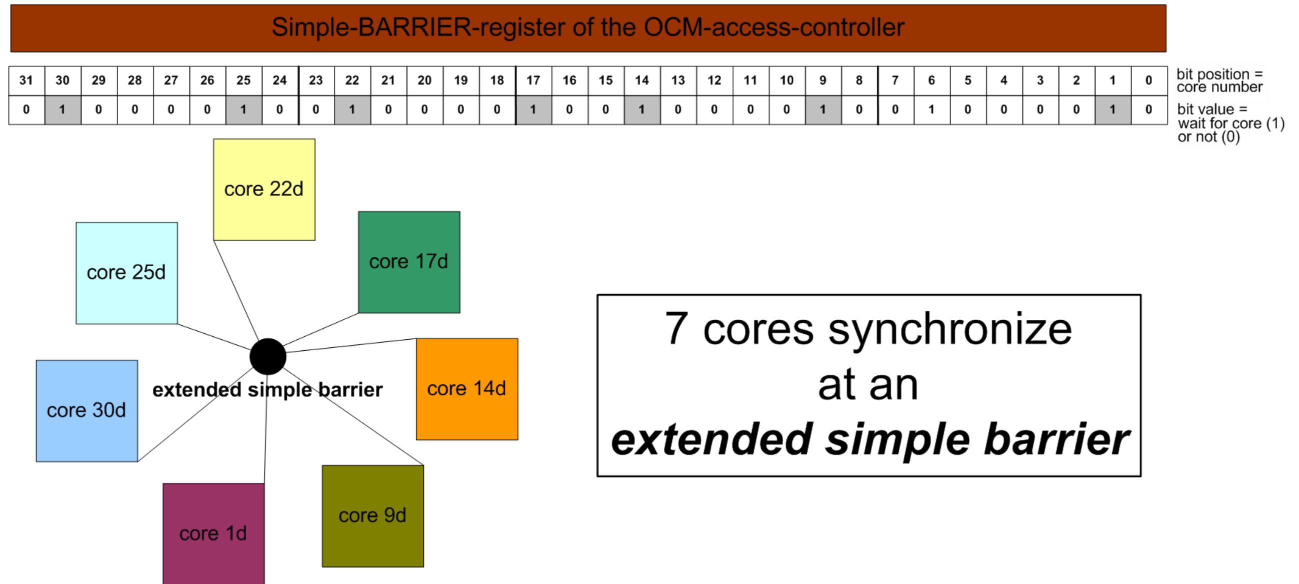


Figure 4.24: Multiple cores synchronize by extended simple barrier

How it works regarding the core-side state machines shown in figure 4.14 is demonstrated in the figure 4.23. Of course the logic to determine when a core can leave the barrier is not as simple as shown in figure 4.22 anymore. Still, by using the fixed correspondence of cores to bits as shown in figure 4.24 the underlying hardware is still quite simple - hence we can still speak of *simple* barriers.

The functionality of the extended simple barrier has been proven for *4* cores using a dedicated test bench. However, there is a limit to scalability in practice: a word is defined in the present PowerPC-architecture by *32* bits, limiting the maximum number of cores in the system to the same number.

4.4.8 Event Synchronization - Complex Barriers

Extended simple barriers have an obvious drawback: the programmer must be aware on what cores the software will execute exactly. For compile-time static scheduling where the software designer has absolute control over what part of the software will execute where and when, this might be good enough. In the perhaps more common case of SMP-systems where the final partitioning and mapping of the software on the available processors is uncertain, extended barriers are just not flexible enough.

To improve flexibility a leap in complexity is necessary to achieve more abstraction in handling the multiple processor cores in the system. Instead of telling exactly

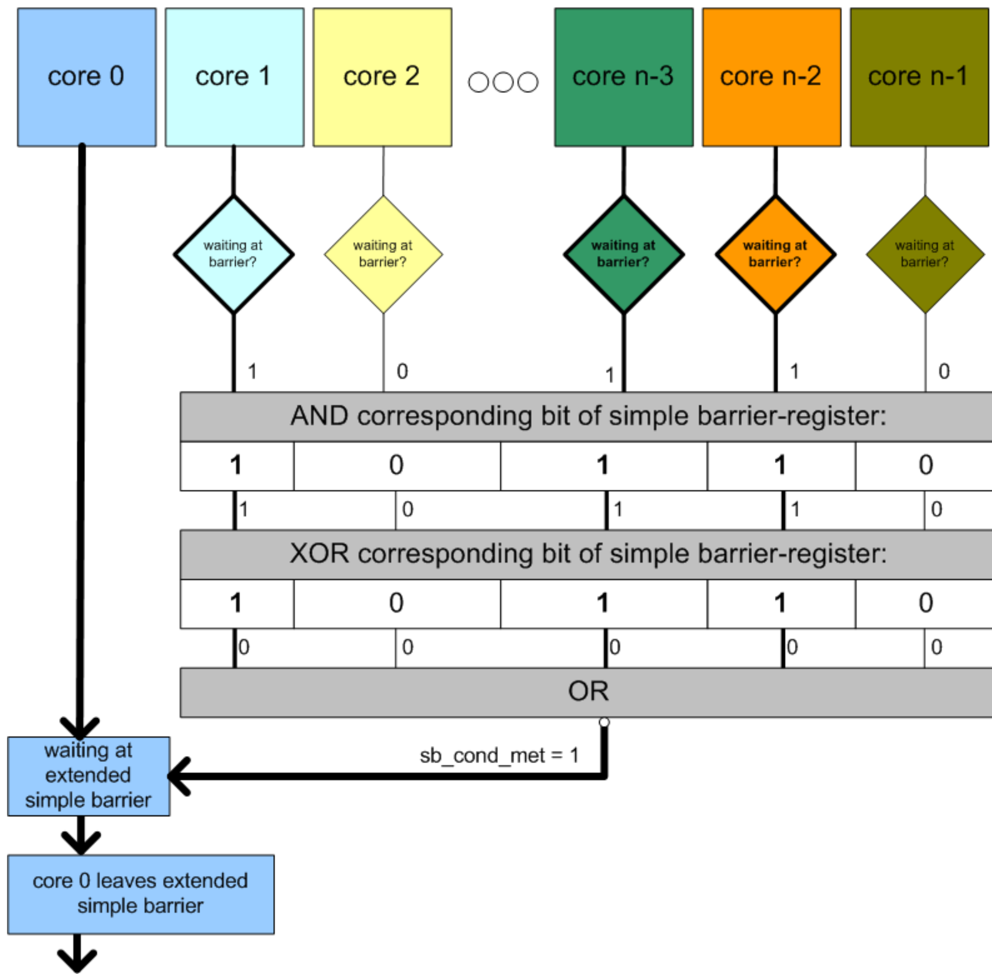


Figure 4.25: Generation of the release-signal for extended simple barriers

for what core to wait it is more optimal to specify only on *how many* cores to wait for. What cores will be executing the final software tasks is then irrelevant for the program code - no porting is necessary afterwards. With this new approach the problem of *how to distinguish between groups of cores* waiting for each other arises. For instance a group of three cores waiting for each other must not interfere with the meeting of a group of two other cores. But how to separate the groups ? The solution is to introduce different barriers: each group is defined by the ID of the barrier they are meeting at. Hence each group of cores waiting for each other has a *unique barrier* or better *ID* they do share, avoiding misunderstandings easily.

The newly introduced abstraction has severe consequences for the `complexity` of the underlying logic. Now a *global logic unit* must manage the interaction of the multiple core-side logics. This global logic is called the *complex barrier manager*. The original intent of the author of this thesis was to introduce the possibility to

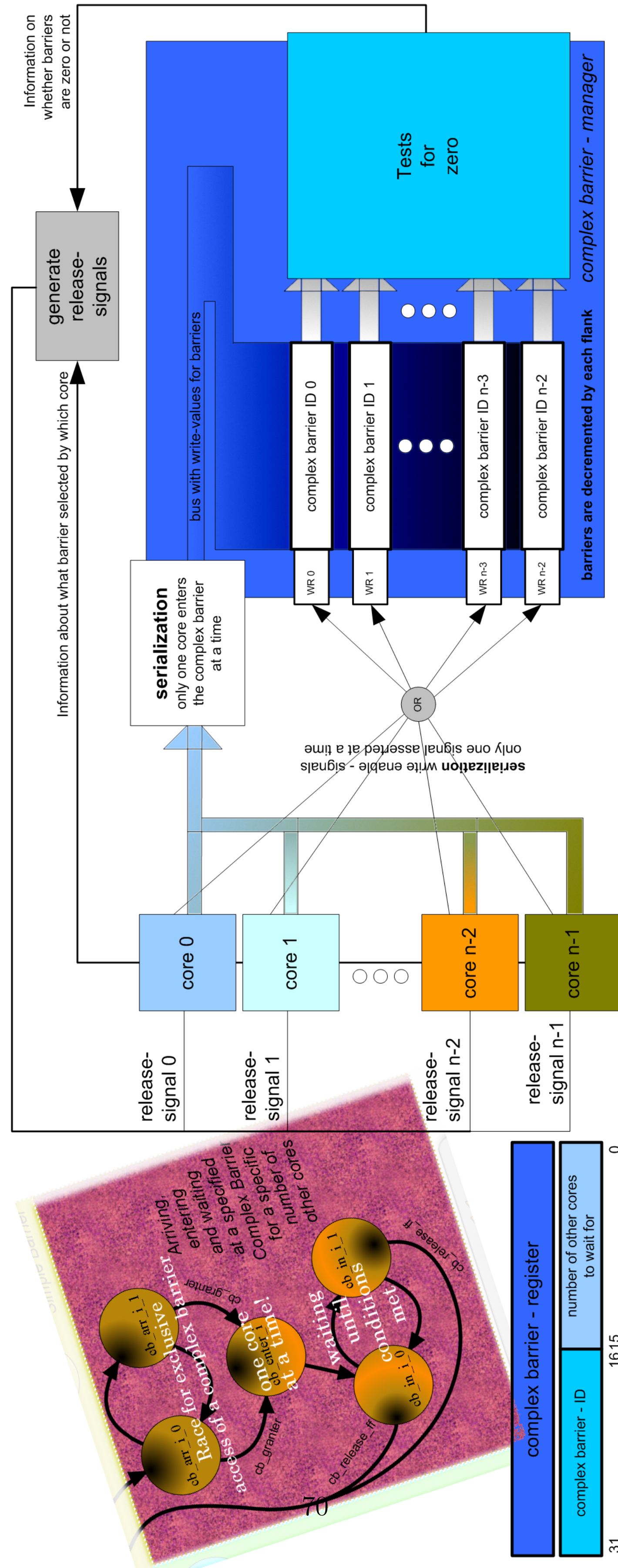


Figure 4.26: Complex Barrier Implementation, abstracted

establish barriers with arbitrary names, but this adds complexity unwarranted: in a system with n cores there can only be $n-1$ barriers active at any time - that implies a natural numbering of the barriers in that range. In the still not too abstracted schematic of the complex barrier mechanism shown in 4.26 one can see that $n-1$ barriers are implemented as registers. These registers are somewhat shared by all processor cores using a bus for writing them. It is assured that only one core uses the write-bus at a time by using a *serialization*: the race described in 4.4.4 was taken and slightly adapted to allow only one core at a time to transition into the state *cb_enter_i*. Therefore multiple cores may arrive at a complex barrier at the same time but only one core enters per clock cycle. The entering core puts the number of cores to wait for on the bus and the complex barrier register addressed by the specific *barrier ID* is written to. If the respective barrier register is not zero it is not written but decremented. All barrier registers are constantly checked for zero. If a complex barrier is zero and at least one core waits in it the *release-signal* for those cores is asserted - and the cores leave the complex barrier.

As a conclusion it is noted here that - as with extended simple barriers - the complex barriers were successfully tested for the *quad-core* case using a specialized test bench. The functionality for the dual-core case was successfully tested in practice as well. However, for the dual-core version *3.16* of the OCM-access-controller the complex barrier - handling was completely removed due to the overhead in logic and its irrelevance in the dual-core case.

4.4.9 Precision of Event Synchronization

The simple and extended simple barriers described previously provide applications with a *tight one-cycle synchronization* which is impossible to achieve by software means in the present context. Since not all involved processor cores are allowed to enter a complex barrier at once, at first glance the synchronization at a complex barrier seems to be dependent on the number of cores. Nevertheless, practically there are two factors that render such reasoning inert:

- *Different programs on different cores:*
  The points in time where also many cores issue the request to enter a complex barrier collide with a very low relative probability (except in cases where there are additional prior synchronization points).
- *Release from complex barrier does not depend on number of cores:*
  Only the meeting at a complex barrier is serialized. The hardware facilities allow for cycle-wise admittance of cores, making the serialization of simultaneously meeting cores still relatively fast considering the total waiting time *at* the barrier.

4.5 Worst-Case Performance

In this chapter we presented a full complement of hardware primitives to facilitate multi-core synchronization. In this section we rigorously analyse the work done and also mention the critical points to be aware of when using the present work.

4.5.1 Direct-Access Performance

Using hardware synchronization we do not have a lock located somewhere in memory and fortunately no spinning is possible anymore. Locking groups or series of accesses is now done by programming the respective registers of the OCM-access-controller. For single accesses there is no necessity to do even that: a single access is not locked explicitly and thus corresponds to a direct access to the BRAM, maximizing the boost in performance compared with software synchronization.

The increase in cycles for direct accesses to the shared BRAM via the controller compared to direct access without the controller is due to the overhead introduced by the OCM-access-controller's state machine. The requests are processed before forwarding them to the shared memory - thus ensuring exclusive access. Even when there is no competition this needs those additional cycles shown in table 4.2.

	load (BC-WC)	store (BC-WC)
BRAM direct	2	2
BRAM via OCM-access-controller	4-6	5-7
OCM-access-controller registers	3	3
PLB memory cell	$\geq$7	$\geq$7
OPB memory cell	$\geq$12	$\geq$12

Table 4.2: Access-performance for single accesses in the dual-core case

In the best case the access to the shared memory is temporally exclusive, in the worst-case the access is delayed by the opponent's access. The maximum delay is determined by the length of the actual execution of an access - which is two cycles as in direct access without the OCM-access-controller. The difference in performance is given by table 4.2: 4 to 6 and 5 to 7 cycles for reading and writing in the best and worst-case (no collision, collision). For details look at the state-machine 4.13 as a reminder: only one core-side state machine is allowed to be in the execution-states at a time - not considering concurrent read-access of course.

4.5.2 Prevention of Starvation in the Worst-Case

The worst-case is produced using long series of read- and/or write-accesses to the shared memory. A welcome effect is observed: the longer the series the more the average access-time approaches the best-case time - 4 cycles for loads, 5 for stores. This observation is positive because it shows that the actual delay of a single access (due to the opponent) happens only at the start of the access-series - then the two access-chains are kind of aligned with a fixed temporal offset. Overlapping in time like that, the two series of accesses do not collide when accessing the shared memory. Figure 4.27 roughly demonstrates this.

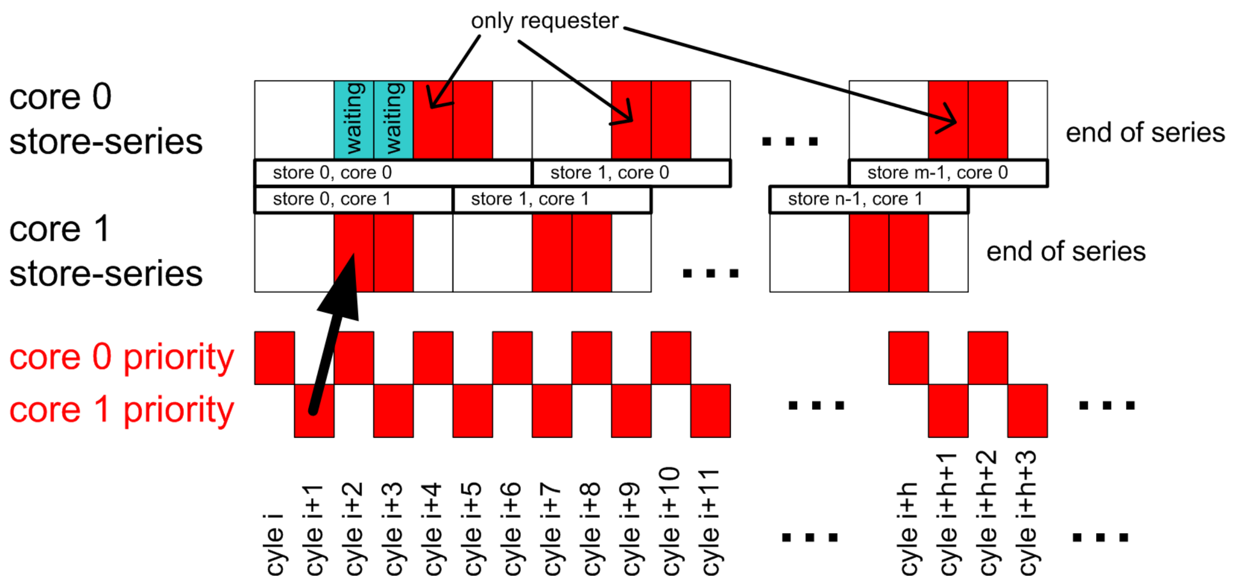


Figure 4.27: Temporal overlapping of worst-case store-accesses

Testing the worst-case with two series of loads uncovered that using hardware facilities does not protect from the possibility that *starvation* can occur. Indeed one of the cores starves while the other core executes its accesses in best-case time - uninterrupted. Obviously this must be related to the fact that the load-accesses need only four cycles to execute. Together with the uncomfortable fact that the variable core-priorities used here toggle between zero and one in our simple dual-core case this disadvantageous situation occurs: the number of cores (*2*) is a divisor of the number of cycles a load needs (*4*). At first glance it seems that a rule of thumb would be that our hardware architecture should favour commands with an odd number of cycles. Also, more than just two cores result in a much longer cycling-period of the core-priorities, breaking the exclusive possession of the shared medium, *smoothing out unfairness* over time.

Still this effect only occurring in our artificial worst-case is unbecoming. A more detailed analysis shows that a hardware architecture as the present OCM-access-controller can be built in a way to avoid such shortcomings even in such unlucky situation as our dual-core worst-case. In contrast to the simplified figure 4.27 figure 4.28 gives a more precise insight into what happens in the worst-case.

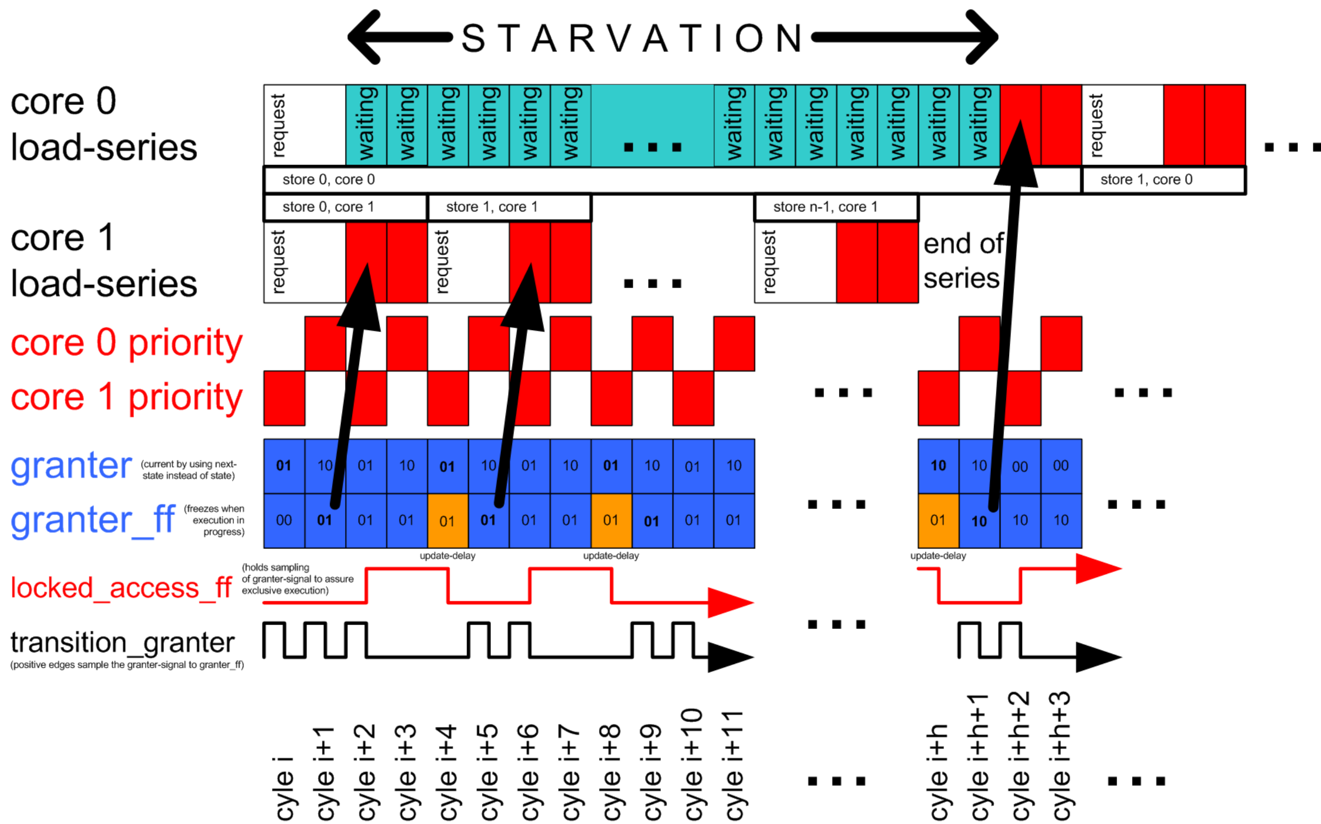


Figure 4.28: Analysis of starvation caused by dual-core worst-case

Figure 4.28 gives a detailed snapshot of what happens and shows clearly that our controller could be slightly more efficient: our granter-logic that grants cores accessing the shared memory could react faster, saving at least one cycle. The cycle which is kind of wasted is marked with a different color, demonstrating the slow reaction of the sampled granter-signal in comparison with the combinatorial granter-signal itself.

For the starvation to occur we pinpoint the following causes:

- **Short** cycle-**period** of **core-priorities** and **even-length access**
  When the granter-signal is sampled after an access, the core that had just finished this access already issued a new request. Due to the even length of the access the core has the highest priority *again*, resulting in the starvation of the other core. More cores or odd-length accesses would prevent this effect from occurring.
- **Slow** reaction of the **granting-logic**
  With a more rapid grant for the waiting other core the starvation would also be impossible.

The present implementation is based on considerations about signal-stability for our relatively high frequency. Using combinatorial signals directly is avoided to prevent combinatorial loops (which by the way occurred in the first version of an OCM-access-controller that was quite faster but not trustworthy).

Starvation in this worst-case can be removed by the following approaches:

- **Penalty** for cores that just finished an access
  - *naive approach*:
    memorize and penalize every access
  - more *intelligent* hence also more *complex* approach:
    penalize only when appropriate
- Insert **additional delay** before forwarding request to shared memory
- **Speed up** the granting **logic** to allow starving core to *sneak in*
  - *naive approach*:
    generate non-conservative sampling edges
  - more *intelligent* approach:
    use preparation signals to pre-signal ending of ongoing access
- **Accesses** instead of the system clock **rotate** the **core-priorities**
  (this method introduces additional levels of logic with longer signal-chains)

One simple approach to break the starvation would be to *delay new requests* by any core by one cycle. This would result in the re-request for the shared memory not to be considered by the granter-signal for another cycle, then the sampled signal would allow the starving core to enter. The disadvantage: all accesses would suffer an *additional one-cycle delay*, that is exactly what should be prevented by using the next-state signals instead of the state-bits itself for the granter-logic.

In the present thesis the second point was pursued. A simple **penalize-logic** was inserted as an effective starvation-prevention, however it added some additional clock-cycles necessary to process the new functionality. Our naive approach was to penalize any core that finished a single-access and to remove the penalty when another core accesses the shared memory. Of course this method is quite primitive - but also effective. A more efficient approach would try to implement more intelligence to inject penalties as really needed.

Accelerating the *granter-logic* would be a more desirable alternative to slowing it down as explained previously. Here there are also two possibilities. The first would be simpler but may require violations of the strict usage of only positive clock-edges. For instance, by using a negation of the system-clock the signal controlling the sampled granter-signal can easily be reshaped to allow sampling immediately after the end of an access is signalled by the *locked_access_ff* - signal. However, using negative clocks and negative edges is exactly what was tried to be avoided. Another approach here would be to make the granter more *anticipating* by additional *preparation-signals*. It seems apparent that one could use the information which is kept in the states to detect the soon ending of an execution (the *rd_exc_1*-state), to give just one example. Truly the last approach seems most promising.

4.5.3 Conclusion

With the exploding extent of the logic added to the OCM-access-controller during its development *signal propagation problems* occurred. As a consequence all unnecessary logic was removed in the later stages of development. For the non-worst case scenarios presented in the next chapter 5 the starvation-prevention was inactive, enabling benchmarking the performance without the additional delay introduced by a dump starvation-prevention-logic.

One additional cycle in between two consecutive accesses is enough to prevent the highly artificial starvation presented in this section. Surely, when looking at any compiler-generated assembler code the reader will agree that the worst-case analysed in this section cannot occur at all in the non-faulty daily routine.

However, in case the concepts of this thesis are used for further development a starvation-prevention should be in consideration from the beginning, to get rid of the possibility of starvation once and for all. Especially when ASIC-design is an issue, one of the more intelligent and hence more efficient solutions described in section 4.5.2 should be preferred (maybe considered together with even more complex methods that to discuss would exceed the scope of this thesis, e.g. a global ticket instead of primitive penalizing etc.).

Chapter 5

Benchmarking

This chapter provides performance results based on extensive benchmarks undertaken to examine the practicality of the synchronization mechanisms developed.

In the first part of this chapter we recapitulate the worst-case performance results and their implications, followed by the second larger part that covers all used non-worst-case test-scenarios. Different scenarios were developed and implemented in ANSI-C to allow the production of vivid results that give a good estimate of what to expect in the daily programming routine - the average case so to say.

5.1 Worst-Case Benchmarks

In this section we summarize the abstracted results gained in the previous chapters about soft- and hardware synchronization primitives. These results were obtained by worst-case test scenarios where the access is done by *assembler*-routines. The *best case* is equivalent with the case that only one processor core accesses the memory, the *worst-case* means all cores access the memory without any pauses.

The greatest differences in performance are present when executing single independent accesses: the OCM - access controller can score by making *locking instructions* (needed when using software locks) *obsolete*. Figure 5.1 shows the best and worst performances for all major synchronization methods implemented in this thesis (OPB- and PLB-locks in all versions considered together) - shown in relation to the *2*-cycle direct BRAM-accesses. It is apparent from fig. 5.1 that *hardware synchronization* clearly *outperforms* the spinning locks of software synchronization. The overhead introduced by hardware synchronization is very low in comparison.

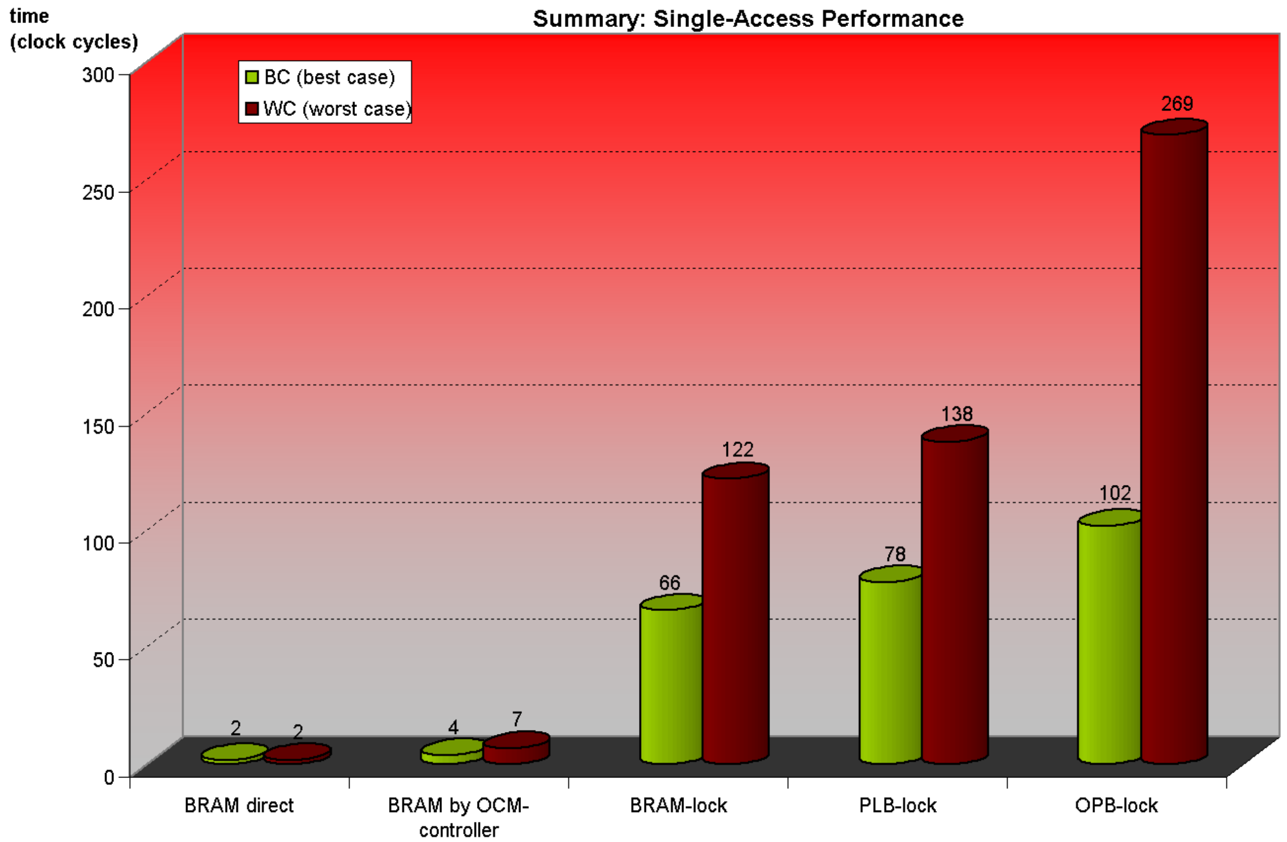


Figure 5.1: Performance of single accesses to the BRAM using all methods

In figure 5.1 one can see that all software-synchronized single-accesses need at least *13 times* the time of the hardware-synchronized ones, in the worst-case even longer. The question arises now how much of this maximum possible gain in performance can be observed in practice, with the applications not only accessing the memory but including non-memory operations as well. This is answered in section 5.2.

Since grouped access is related to single accesses and involves some *dead time* - waiting time for the lock - we do not provide similar diagrams like the ones for single accesses here: the performance difference will be less evident than in figures 5.1 depending on the group size. In difference to single operations the controller must be programmed explicitly in these cases. However, non-worst-case performance results of that kind are shown in the next section 5.2 as well.

5.2 Non-Worst-Case Benchmarking

First it must be emphasized here that the following test scenarios are *not* the worst-case. They were realized using a higher programming language for more flexibility in testing and parameterization. Nevertheless there was invested considerable effort to *tune* the testing as much as possible (registering loop-variables, loop-unrolling etc.) to decrease the computational overhead in favor of the communication between the processor cores. Doing so our test-scenarios achieve a high data-throughput via the shared memory. Although this is not the artificial worst-case producible only by assembler, this might as well represent the average-case in everyday's computing of embedded systems (a 'worst' average case, so to say).

For the 'real' worst-case please consult the previous chapters and section 5.1.

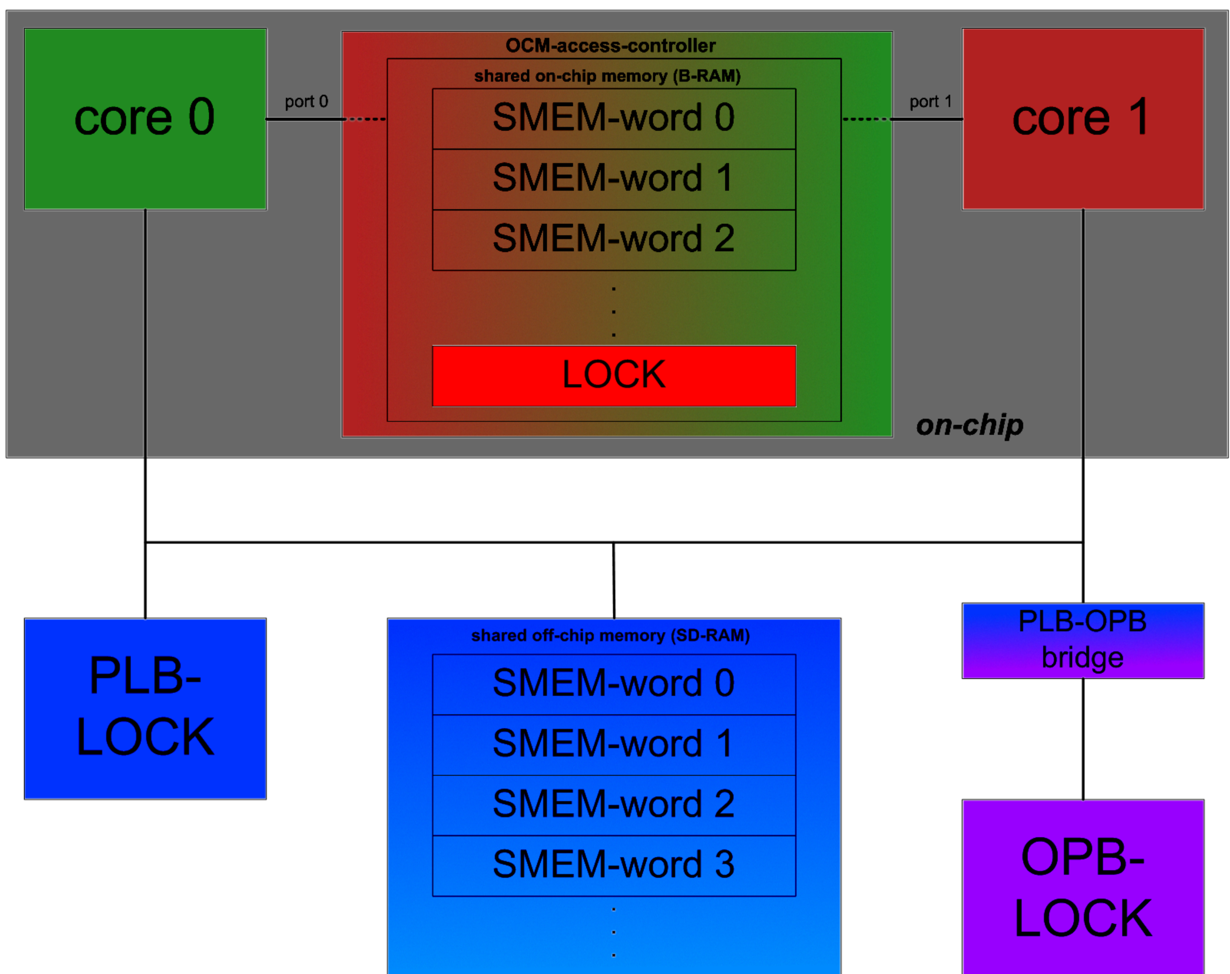


Figure 5.2: All modules available for system benchmark configuration

5.2.1 Benchmark System Configuration

It is common practice to use also the SD-RAM as shared memory, especially when large amounts of data are used. That is why we use SD-RAM in our real-world benchmarks here in addition to the BRAM used so far to increase the statistical data and broaden our perspective on what to expect in practice.

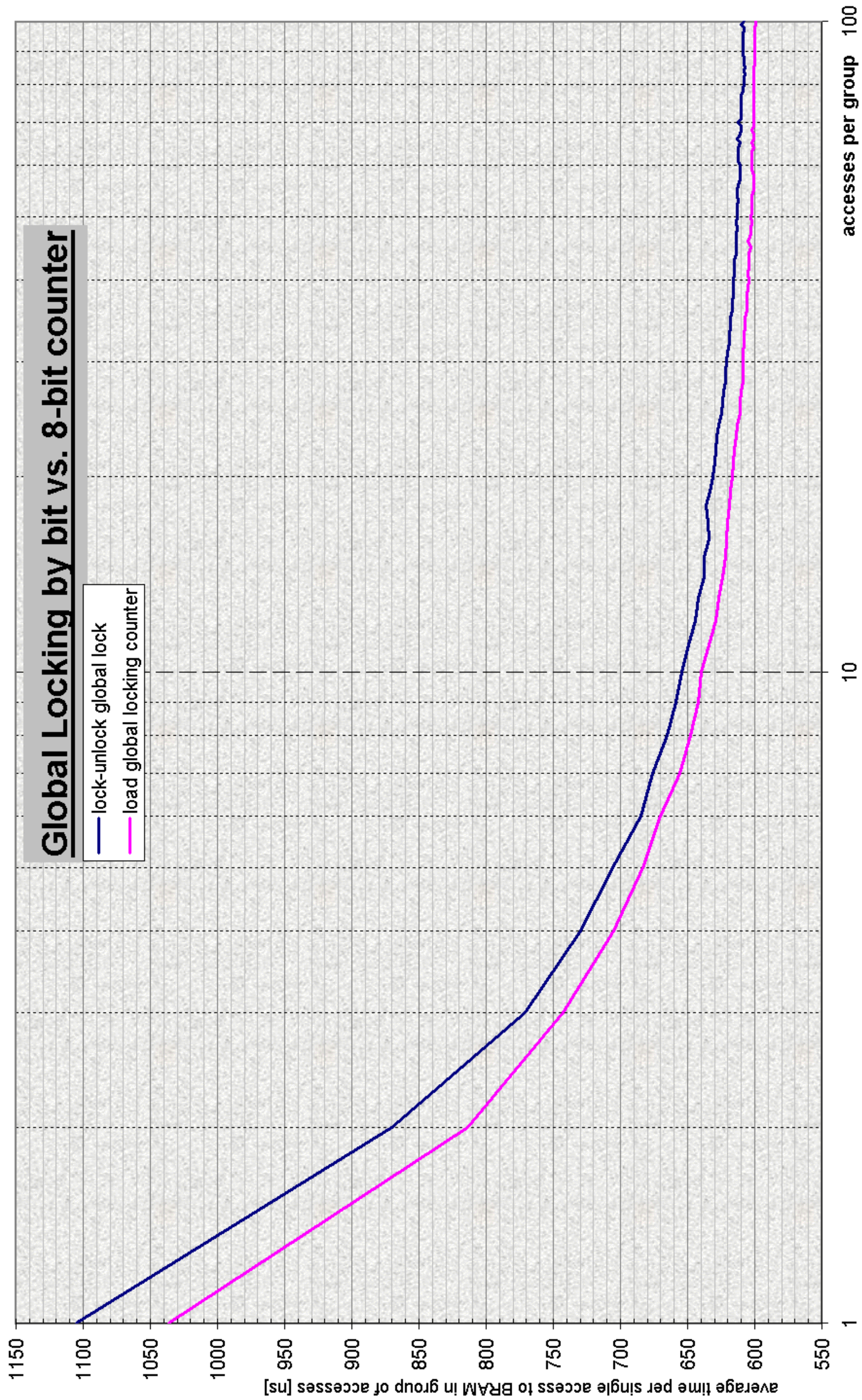
Figure 5.2 shows all system-modules available for a testing environment. Shared memory can be chosen to be *on-chip* with dual-ported fast access over the OCM or *off-chip* accessed by the PLB. Hardware-locking accessing the on-chip B-RAM can be done implicitly and explicitly by the OCM-access-controller. With deactivated OCM-access-controller software locking can be used: the lock can be located either on-chip in the B-RAM or off-chip in form of a PLB- or OPB-memory cell.

Configurations used for benchmarking either use the dual-ported on-chip BRAM or the single-ported off-chip SD-RAM as shared memory. In both cases all relevant pairings with locking methods of this thesis are examined to best cover the whole range of possible system configurations and their performances.

5.2.2 The Global Locking Counter

The OCM-access-controller has a *global locking counter* as an alternative to the global locking bit. The advantage of the counter is that **it resets itself** when all accesses are executed by the controller. Using the counter should hence give some performance gain. This improvement has been examined by using the *8*-bit global locking counter of the OCM-access-controller in version *3.16* on the dual-core PowerPC system.

The assumption that the usage of the counter brings the biggest advantage for small groups of accesses is testified by the results summarized in diagram 5.2.2. With small groups the advantage of *saving* one *unlocking-instruction* is relatively high whereas this improvement decreases with larger groups. Summing up, as a rule of thumb using the counter should be preferred if the group size is representable with by the word-size of the counter (*8* bits in *v3.16*) and unlocking is scheduled statically, otherwise the classical locking bit can be used.


Global Locking by bit vs. 8-bit counter
lock-unlock global lock
load global locking counter
average time per single access to BRAM in group of accesses [ns]
accesses per group
1150
1100
1050
1000
950
900
850
800
750
700
650
600
550
1
10
100

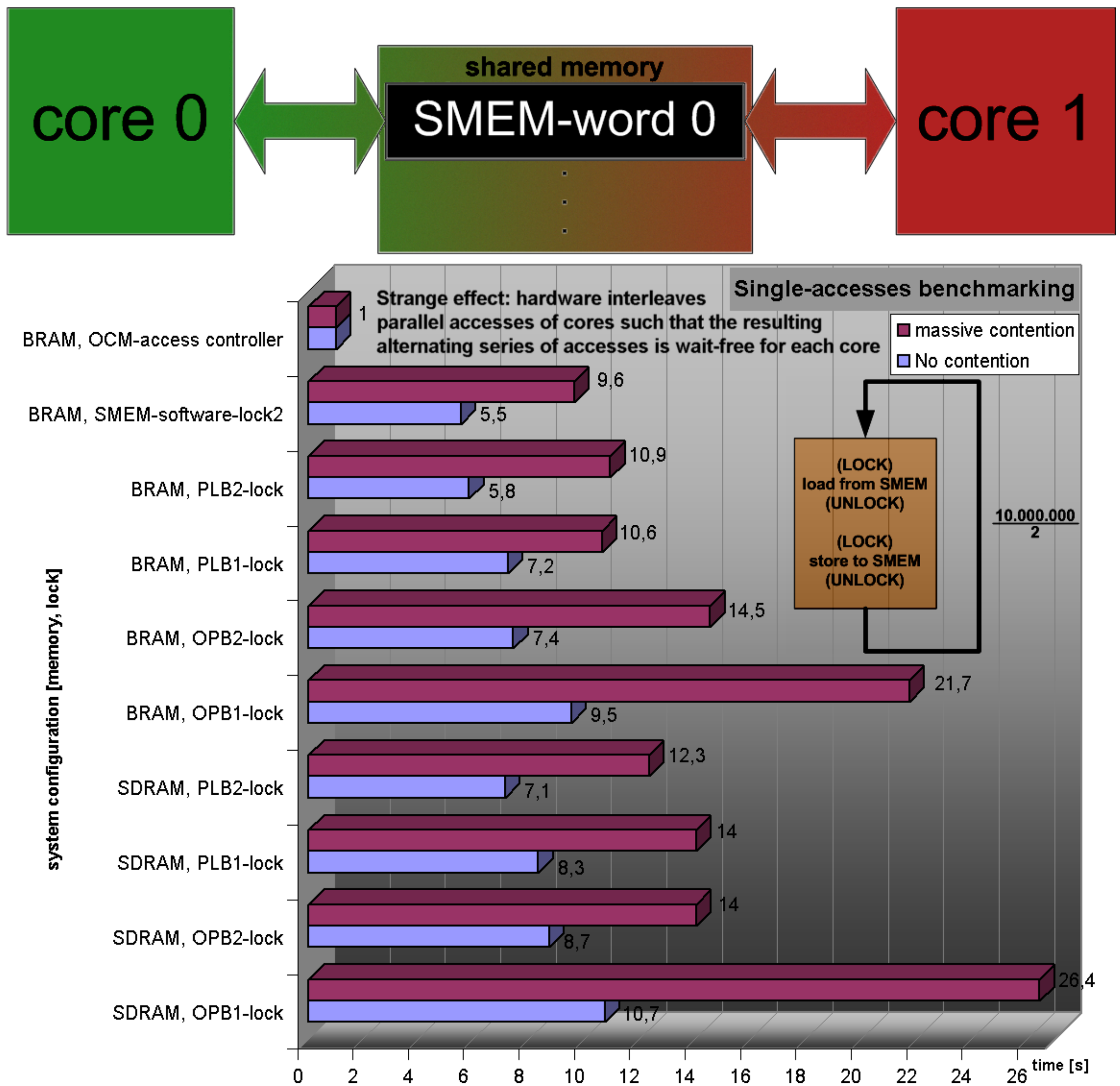


Figure 5.3: Benchmarking: single accesses (no contention: only one core active)

5.2.3 Heavy-Load Single Access Benchmark

Here we make a large-scale memory-access by both cores to the *same word* of the shared memory. The start of the scenario is synchronized temporally by a hardware barrier. A total of *10 million single accesses* are executed, resulting in **countless conflicts** between the two cores accessing the same word.

The results are shown in figure 5.3 and speak for themselves.

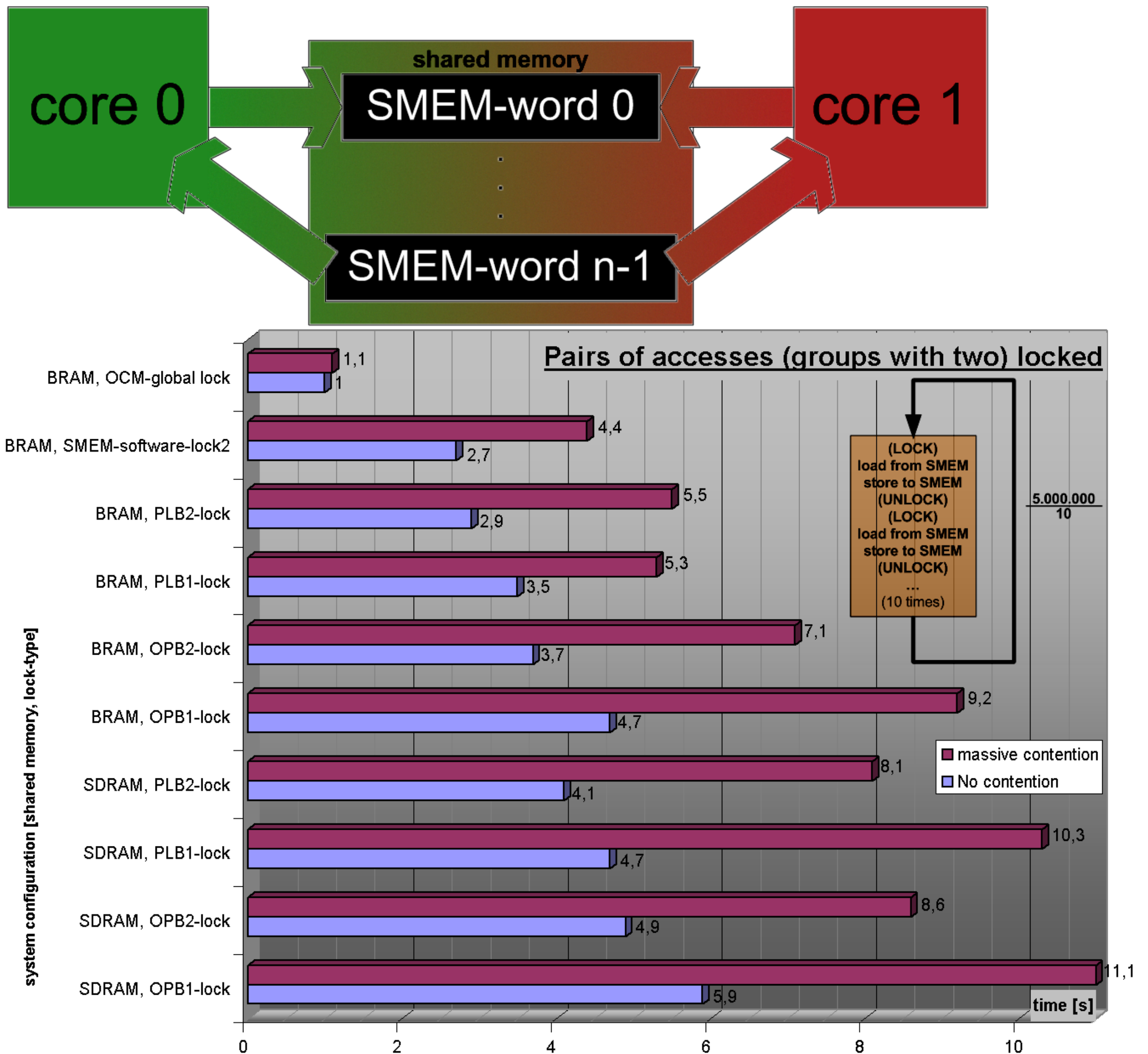


Figure 5.4: Benchmarking: paired accesses (no contention: only one core active)

5.2.4 Heavy-Load Paired Access Benchmark

Now we pair a *read* and a *write* from and to the shared memory with some basic *bit-operation* (to avoid compiler-optimizations eliminating the initial read). The combination of accesses is made *atomic by locking*. In this scenario the hardware synchronization cannot lock implicitly since there are two accesses grouped. Hence we **must lock explicitly** using the global locking bit of the OCM-access-controller. Therefore the results of global locking are not as superior over software locking as without grouping. Yet hardware locking outperforms the spinning locks.

Access-configuration (access-target, lock-location)	No contention (ms)	Massive contention (ms)
BRAM, OCM-access controller	1025	1025
BRAM, SMEM-software-lock2	5517	9619
BRAM, PLB2-lock	5810	10937
BRAM, PLB1-lock	7226	10613
BRAM, OPB2-lock	7421	14455
BRAM, OPB1-lock	9472	21731
SDRAM, PLB2-lock	7080	12288
SDRAM, PLB1-lock	8316	14031
SDRAM, OPB2-lock	8651	14010
SDRAM, OPB1-lock	10725	26448

Table 5.1: Benchmarking: single accesses, exact values

Access-configuration (access-target, lock-location)	No contention (ms)	Massive contention (ms)
BRAM, OCM-access controller	1030	1054
BRAM, SMEM-software-lock2	2734	4426
BRAM, PLB2-lock	2890	5512
BRAM, PLB1-lock	3520	5266
BRAM, OPB2-lock	3681	7071
BRAM, OPB1-lock	4726	9213
SDRAM, PLB2-lock	4058	8116
SDRAM, PLB1-lock	4701	10346
SDRAM, OPB2-lock	4874	8627
SDRAM, OPB1-lock	5908	11093

Table 5.2: Benchmarking: paired accesses, exact values

5.2.5 Heavy-Load Floating Windows Benchmark

In order to give a realistic assessment about the improvement of *address-locking* over *global locking*, an access-intensive benchmarking method has been developed.

The main idea is to assign each processor core an area of shared memory called an *address window* or *memory window*. These windows are accessed by their respective cores and then moved by an appropriate *offset*. The fact that the windows are moving in opposite directions and changing direction when colliding with the borders of the shared memory makes collisions between the core's windows inevitable, as is demonstrated in figure 5.5.

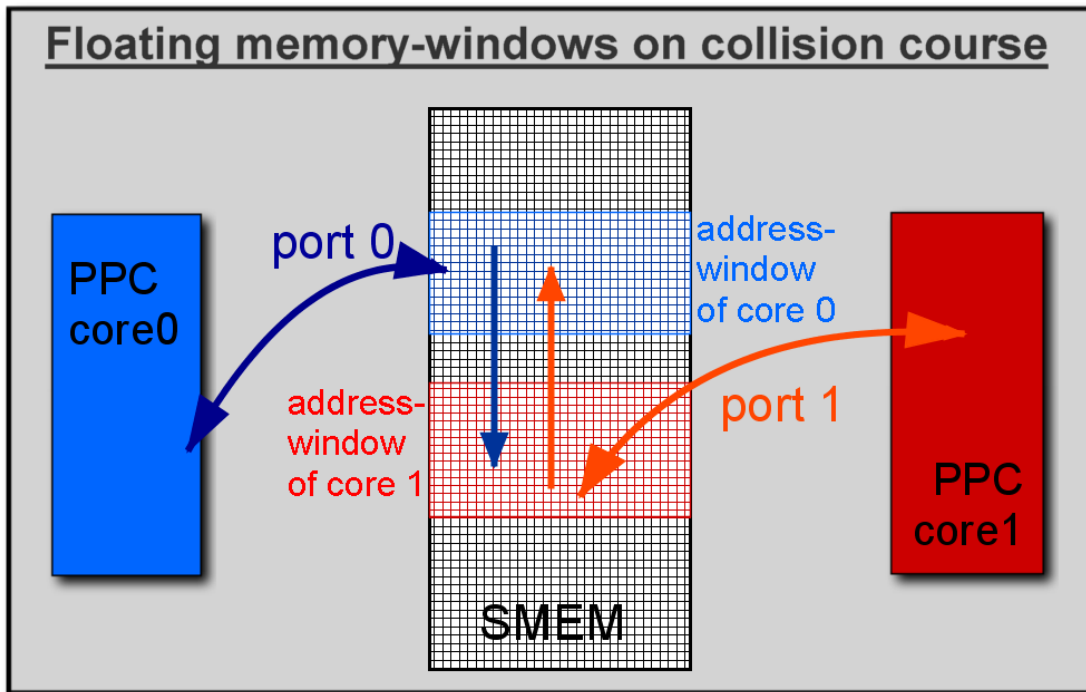


Figure 5.5: Benchmarking by moving memory-windows on a collision course

A fixed total number of times each core accesses its window and shifts it to a different location. Thus by varying the size of the memory window we get more and more accesses in total: the larger the window the more accesses to it per one of the *100 thousand times* a *whole address window* is *accessed*.

Strong *optimization efforts* were made to reduce the amount of memory-remote operations in order to simulate real applications with **heavy load** on the processor-shared memory system (or more like it, get an *upper bound* for such applications: in reality they will also compute something and not only access the memory).

The overall results are shown in diagram 5.6. In order to reduce the complexity all the different synchronization methods using SD-RAM as shared memory are compressed into a blue belt confined by the worst and best result curve.

Regarding the more interesting benchmarking configurations using shared B-RAM the following observations can be made by looking at figure 5.6 and numbering the most interesting result-curves from 1 to 4:

1. `B-RAM always outperforms SD-RAM as shared memory`
   The SD-RAM as a single-ported solution with the usage of spinning locks (software synchronization) can lead to *thrashing*, resulting in an (at best linearly) rapidly increasing time function. Also of course, SD-RAM has also a way longer access time (it is off-chip).

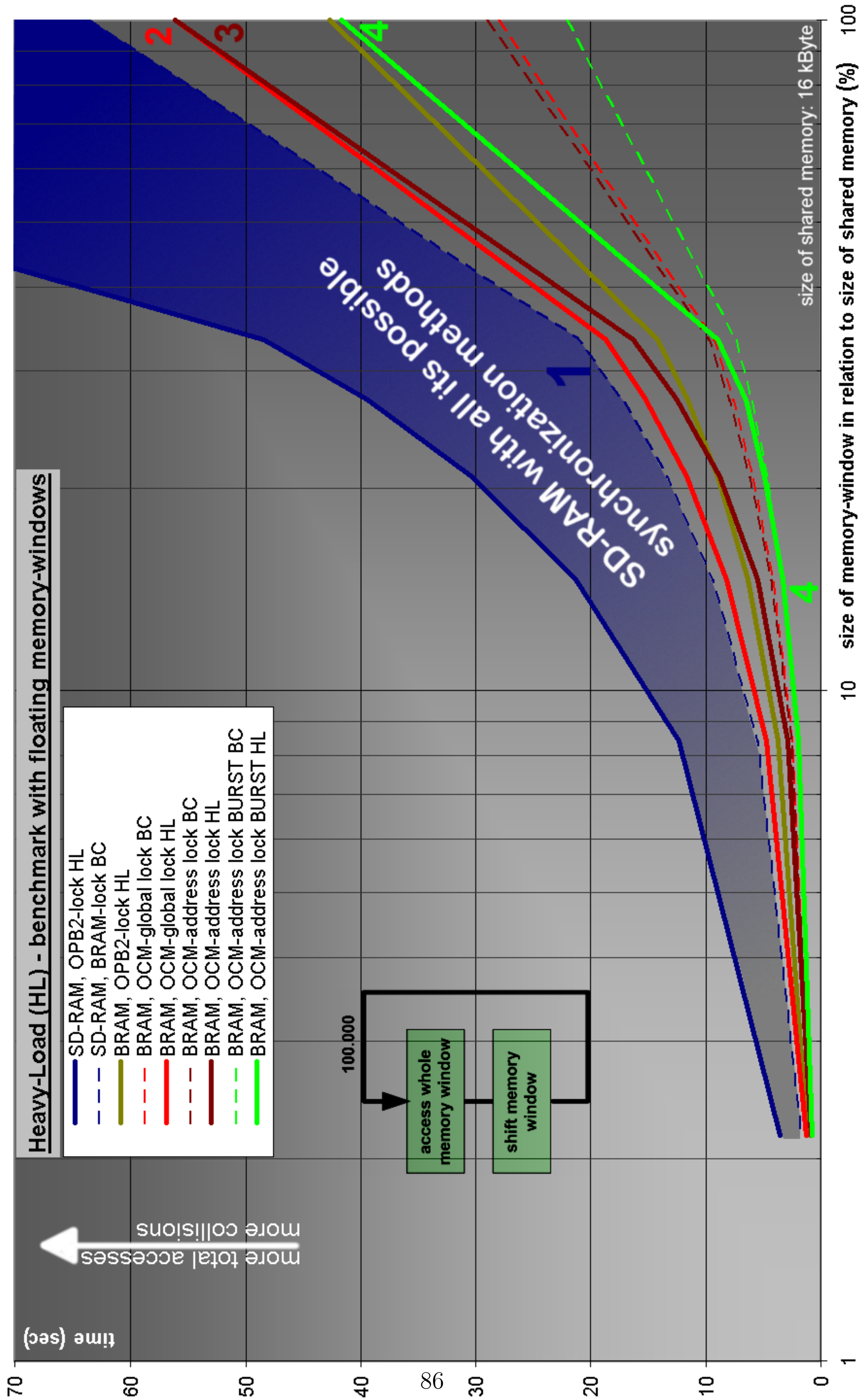


Figure 5.6: Results of benchmarking with floating windows

2. `Global locking with the B-RAM as shared memory performs worst of all B-RAM configurations`
   Global hardware locking is quite capable to handle the heaviest load without an unreasonable explosion in overhead and disadvantages. The relatively bad performance is due to the following two reasons:

   - Global locking prevents the efficient usage of both ports of the dual-ported memory: practically the B-RAM works as fast single-ported shared memory accessible over the OCM.
   - The slight processing-overhead for the locking logic decreases the relative performance when accessing large memory-areas.

3. `Address-locking behaves - in the case the windows are spanning the whole shared memory - like global locking.`
   No surprise here: in its worst-case address-locking brings no advantage to simpler global locking. With all cores requesting the whole shared memory to be address-locked potential concurrency is eliminated.

4. `Bursted address-locking outperforms all other methods`
   In the special *burst-mode* address-locking is used to reserve an address-window. Then the direct access to the shared memory is enabled, but with the OCM-access-controller keeping its locking *state*. After the whole *bursted* access the direct access to the B-RAM is deactivated again. At last the address-lock is released. The hardware guarantees that direct access is only possible with no address-conflicts present.

   This procedure achieves the best performances of all, going along with the contention-free best-cases until address-windows almost a third the size of the shared memory. Then the performance worsens due to multiple collisions but still outperforms all other heavy-load configurations.

The improvement by using *bursted vs. normal address-locking* becomes more significant with address-windows larger than *20* percent of the size of the whole shared memory. Then the overheads of the slower single accesses sum up to as much as *48* percent of the total time for the scenario. Hence the performance can almost be expected to be doubled at best in the average case by using bursted address-locking. Of course this improvement is less strongly developed with the decrease in total accesses over time.

In this scenario all was done to maximize the frequency of access to the shared memory. In more common applications the difference in performance will be respectively less pronounced, still in certain situations (especially those with a high volume in data-transfers and real-time issues) the superior hardware methods may be vital in reaching one's goals without investing dearly more into the underlying hardware (faster memory, faster busses, higher frequency etc.).

Mehr als die Vergangenheit
interessiert mich die Zukunft,
denn in ihr gedenke ich zu leben.

ALBERT EINSTEIN

Chapter 6

Conclusion

6.1 Summary

In this thesis a dual-core shared-memory system was set up using state-of-the-art equipment. In doing so a *digital flow* was developed that makes the usage of on-chip block-RAM as shared memory possible, in addition to the conventionally used SDRAM. In the end the system set up contains *two* processor cores, *two* different types of shared memory and offers *expandability in hardware* by means of the IBM core connect busses PLB and OPB. A possibility to add on-chip hardware modules was found by manually inserting the logic into the underlying tool-chain, thereby greatly extending the number of possibilities to explore in the system. Detailed information on the complete setup can be found in the appendix A.

With the system set up and ready to be programmed, the *non*-coherence of the dual-ported shared memory had to be corrected by using assembler-optimized *locking* of the shared memory as a software method to establish coherence. Despite some slight unpredictabilities in the underlying hardware this seems acceptable for the average case of *non*-real time systems, but the attempt to establish more control over the locking process by the help of the IBM busses failed and lead to *starvation*. Even worse, with these "quick fixes" of our design, the potential concurrency of the dual-ported shared memory lay dormant. It quickly became clear that with such software locking dual-ported memory is of no benefit.

By inserting an *on-chip* logic module as an *access-controller*, the concept of spinning locks in software was abandoned. The controller oversees the operation of all processor cores involved, comprising *inter-core* as well as the inevitable *core-side logic*. Multiple simultaneous demands to access the shared-memory are handled by allowing only one of the cores access. The method used here is inspired by the idea of *fairness*, being essentially a round-robin mechanism between all competitors. Since this decision is made during each hardware-cycle, a fast *throughput* to the shared memory is realized. Specifically in the case where there is no write-request the concurrent reads are executed truly in parallel by the new hardware.

In order to group accesses together that must be *atomic*, *global locking* is implemented in the hardware. Due to the usage of dual-ported memory, an additional locking-method has also been developed: each core can hold one *address-lock* at a time, corresponding to a coherent block of shared memory. As long as the core owns this lock it also has exclusive access to this memory-block. This *address-sensitive locking* exploits parallelism for data-structures and whole address-regions, not just for single accesses as explained in the last paragraph.

Also mechanisms to allow for *point-to-point synchronization* were realized in hardware. Simple and more sophisticated *barriers* covering the *dual*- and *multi*-core case were developed and tested successfully on our dual-core system - the *precision* of synchronization is clearly superior in comparison to software barriers.

To allow easy configuration and debugging, the controller implements *special-purpose registers* that are directly accessible by the software due to the memory-mapped architecture. A special bit enables to bypass the hardware unit and use the block-RAM as it is, unburdened by the newly added hardware.

During the evolution of this thesis program *code* was developed in the languages `VHDL`, `PowerPC assembler` and `ANSI C`, with increasing quantities in exactly that order. While a higher language as ANSI-C may be suffice to get average-case benchmarking results, assembler is a must to get meaningful and especially *worst-case* results. All of these kinds of results are presented in this thesis.

By delegating the responsibility of exclusive write-access to this hardware unit the *software-overhead* could be reduced significantly. Also the system at hand can be utilized much better, greatly improving the throughput to the shared memory and thus the overall performance of the system. Thus we can clearly say that the advantage of migrating delicate issues concerning consistency and mutual exclusiveness into hardware is (still) worth being considered in future multi-core hardware designs, in particular in the case of reliability being of major concern.

Complementing systems that lack hardware-support is likely to introduce much *overhead*, possibly leading to the *inadequacy* of the given hardware. With hardware-support the same and thus cheaper hardware resources might suffice.

The results also show that an intelligent hardware improves the performance with single-ported shared memory as well (examined by using the dual-ported memory only with global locking). Whether a dual-ported memory seems fit depends on the given application: the *coupling* between the processor cores is vital here. With a *tight coupling* we have a high frequency of communication, resulting in many collisions when accessing regions in the shared memory. In such a case dual-ported memory can not be used efficiently. But with a *less tight coupling* between the cores, the impact of a small dual-ported shared memory complementing a larger single-ported one may still be significant enough to warrant the additional costs.

6.2 Future Work

The comparison between SDRAM and on-chip BRAM serving as shared memory can be continued by *scaling* the *frequency* of the processor cores and hence the on-chip memory. However, this is likely to only sharpen the contrast between the *on-* and *off*-chip shared memory and hence will only emphasize the advantage of on-chip memory even more. Therefore it was of no primary concern in this thesis.

With the on-chip BRAM being clocked with the same frequency as the processor cores, the usage of caches was of no importance in this thesis. Nevertheless, with the off-chip SDRAM serving as shared memory, it might be interesting to examine the impact of *caching*. Of course this brings up new uncomfortable issues like cache-inconsistency. It is only logical that the whole testing procedures must also be reconsidered, with the potential outcome that the additional efforts do not warrant the usage of caches and the costly hardware they are accompanied by.

The underlying hardware mechanisms developed in this thesis can serve as a stable *base for more sophisticated hardware schemes*. For instance, the intelligence used when handling simultaneous requests for the shared memory can be improved by keeping statistics about the processor cores' accesses, thereby also preventing an artificially generated worst-case to lead to starvation (s. section 4.5). In the case the applications suggest it, it might be possible to allow *more address-regions* to be reserved *per core* at the same time (consider a "gap" in the middle of a large memory block which cannot be accessed if the whole block is locked).

Of course the concept of using dual-core shared memory cannot be extended for more than two cores in practice: despite all the simulations, we do not have *multi*-ported RAM beyond *dual*-"portedness" in reality. However, it might be possible to extend the hardware control in such a way that *multiple requests for* a *dual-ported shared memory* are handled, for example by always selecting two accesses per cycle that are executed. The hardware could be made intelligent enough to also pair two write-accesses to different memory locations, not only read-accesses. Still, if the outcome warrants the additional efforts, and hence costs, is to be shown.

What is quite fascinating is the idea to simulate the *"real" engine-control software* on the system developed in this thesis. This can happen by feeding the "real" software by *test-vectors* or even only simulate the software's *behaviour* in respect to data-transfers. Unfortunately the analysis of the immense engine control software raises some difficulties and was still unfinished with the conclusion of this thesis. Even with all data-dependencies fully uncovered, a reasonable partitioning of the software is still to be found, ideally minimizing the coupling of the processes executed at each core. Considering the extent of the engine control software this nontrivial optimization task requires significant help by *heuristic* search methods.

Closing this section, there is still a lot to be achieved in the step from a *uni*- to a *multi*-processor, both at the software- and at the hardware-side. A complete understanding of the software-processes involved would greatly facilitate the design of an appropriate dual-core multicontroller system for the automotive sector. Since this is likely not to happen in that order, this thesis might help in finding a good start for a reasonable hardware design.

6.3 Acknowledgement

I would like to express my gratitude to BMW for giving me the chance to work on a fascinating subject for my thesis.

I sincerely thank Dr. Benito Liccardi from BMW for his kind and invaluable support. Thank you so much for giving me the opportunity to complete a thesis in a current and exceptional field of research! It was a great challenge to do this project and I have expanded my knowledge tremendously in the process!

I thank all members of the department EA-400 at the BMW Group in Munich who welcomed the guest from Austria with open arms from the beginning. Thank you for the great time in Munich, and I am looking forward seeing all of you again!

My sincere thanks go to my instructor here at the Vienna University of Technology, Dr. Martin Schöberl. Without him, this thesis might never even have got off the ground. Thanks a lot for believing in me Martin!

I also thank Prof. Herbert Grünbacher and the Institute of Technical Computer Science of the Vienna University of Technology for supporting me and this project. I learned that writing a thesis in cooperation with the private sector can create many unexpected challenges, and I am grateful that you helped me overcome these.

Last but not least, I sincerely thank my family for their support. I know my curiosity often leads me to pursue goals that might not seem obvious to everybody else. Because of this, I value the unconditional support which I have received from you all even more. Thank you for backing me in all my decisions! Thanks for being there for me when I needed you the most! It is good to know that wherever life will take me, I will always be able to come home - you all are my safe haven.

I am also very grateful to all my friends, old and new,
- thank you all for sharing my enthusiasm about this undertaking!

If there exists one drug that is worth taking, clearly it is knowledge.

Christian Stoif

Vienna, the 1st of November 2008

Appendix A

Digital Flow

This appendix represents a brief guide for anyone who intends to construct a dual-core system with one of the current boards and software tools of Xilinx supporting dual processor cores. The digital flow presented here was established using a *Xilinx* development board *ML41x* containing a *Virtex-4 VFX60* FPGA device on it.

To be able to follow the hardware-part of the flow you need to install the *Xilinx Embedded Development Kit (EDK) 9.2i* or higher, the *Integrated Software Environment (ISE) 9.2i* or higher, and of course you need a Xilinx board containing one of the FPGAs with at least two hard PowerPC-cores in it.

Please be aware of the fact that the free ISE-version called `Webpack` is not supporting high-density FPGAs like the VFX-family that has more than just one PowerPC-core. Also, some *intellectual properties* (*IPs*) in ISE need *licences* to be functional, as does the main ISE application. However, the evaluation copy (of *ISE 9.2i*) allows unrestricted working for a period of *60* days.

A.1 The Xilinx Development Board ML410 and the Xilinx FPGA Virtex-4 FX60

Our platform to work with is a *Xilinx development board ML410* with the features given in the following block-diagram A.1:

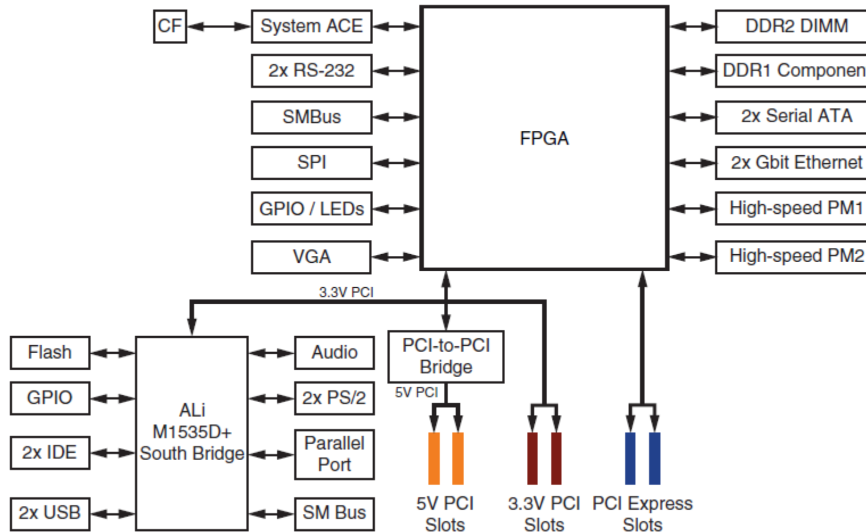


Figure A.1: High-level block diagram of the Xilinx ML410, taken from [Xil07e]

For this thesis we use one *USB*-port for programming the system using *JTAG* for the hardware and for the upload of *ELF*-applications. The *64*MB big *DDR1 Component* is used as external memory. For debugging one core uses the JTAG-connection over the USB-cable, the other core uses one RS-232 connection.

The FPGA itself is a black box in fig. A.1, the actual FPGA in our case is an FX60 from the Xilinx Virtex-4 family of FPGAs. The most important details are given in A.2, see [Xil07h] for more details. What concerns us is the *on-chip Block-RAM* which we need to implement a **high-speed on-chip shared memory**.

	Configurable Logic Blocks (CLBs)(1)					Block RAM				
Device	Array(3) Row x Col	Logic Cells	Slices	Max Distributed RAM (Kb)	XtremeDSP Slices(2)	18 Kb Blocks	Max Block RAM (Kb)	DCMs	PMCDs	PowerPC Processor Blocks
XC4VFX12	64 x 24	12,312	5,472	86	32	36	648	4	0	1
XC4VFX20	64 x 36	19,224	8,544	134	32	68	1,224	4	0	1
XC4VFX40	96 x 52	41,904	18,624	291	48	144	2,592	8	4	2
XC4VFX60	128 x 52	56,880	25,280	395	128	232	4,176	12	8	2
XC4VFX100	160 x 68	94,896	42,176	659	160	376	6,768	12	8	2
XC4VFX140	192 x 84	142,128	63,168	987	192	552	9,936	20	8	2

Figure A.2: Virtex-4 FX family, partial overview, taken from [Xil07h]

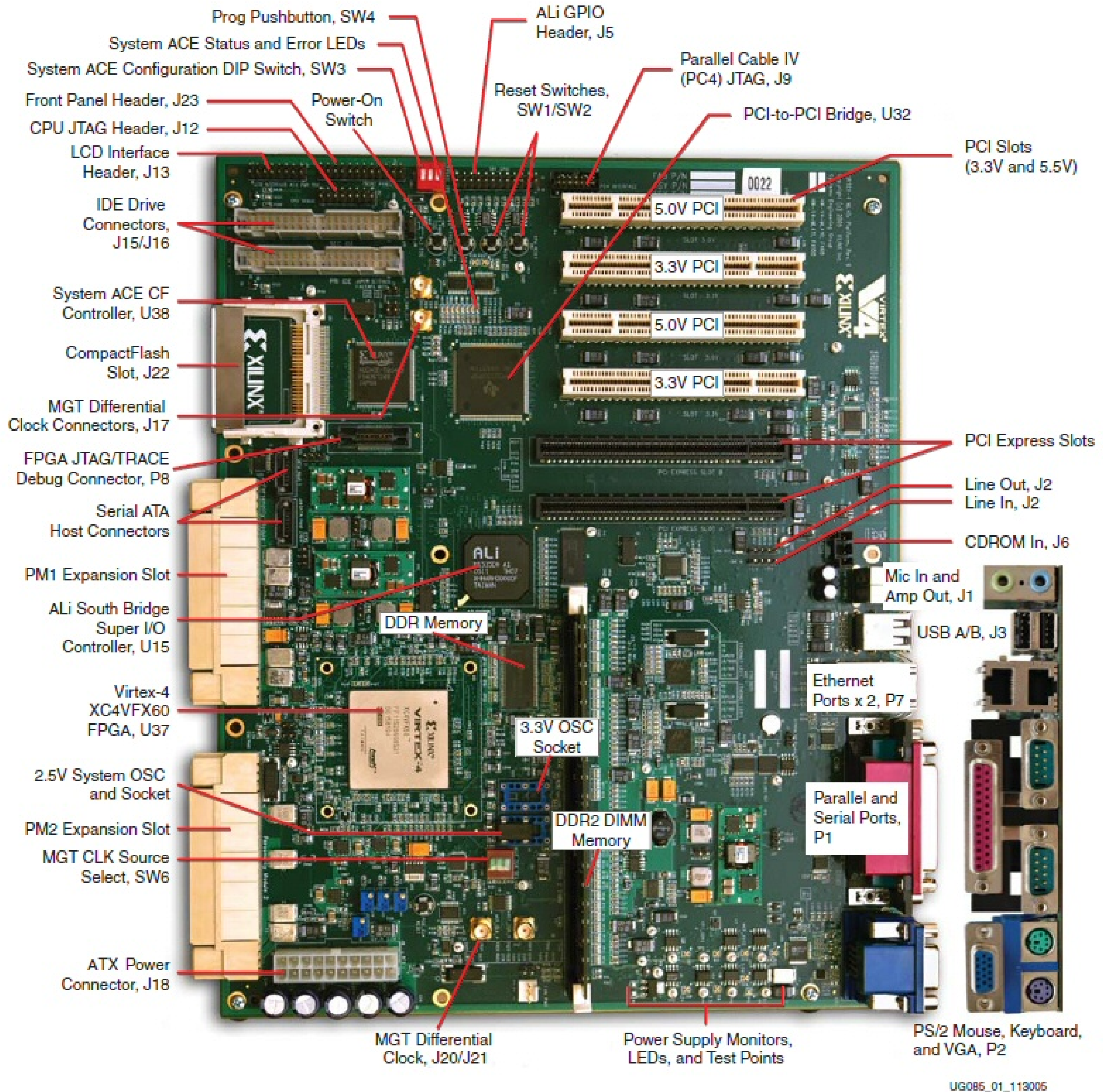


Figure A.3: Picture of the Xilinx ML410, from [Xil07e]

A.2 Hardware Flow

This section deals with the issue on how to configure the *Xilinx ML41x* board and its *VFX60* FPGA in order to access *both* cores. Since only one of the two PowerPC-cores is activated by default, the other core must be connected as well to get a fully functional *dual-core system* to work with.

A primary goal of the designer must be to minimize re-factorization iterations since the hardware side of the digital flow can easily become very costly in time. In optimal no rearrangement of the hardware configuration of the system should be necessary afterwards. That was also tried to accomplish with the initial hardware setup for this thesis. However, adding new logic modules can and could not be avoided considering the hardware nature of this thesis.

A.2.1 Setup of a Multi-core System with Xilinx EDK 9.2i

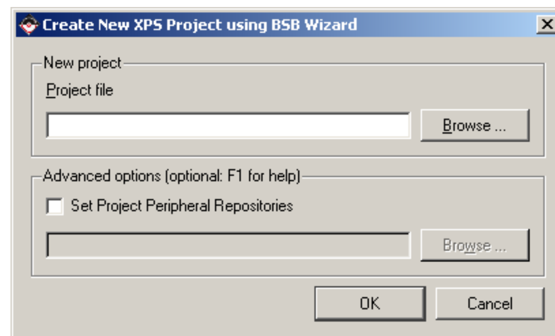


Figure A.4: Creating a new project in the EDK

The Xilinx *Embedded Development Kit (EDK)* is the most comfortable way to quickly configure your system to accommodate your wishes. Please refer to [Xil07b] about how to open and set up an initial project (done best with the *Base System Builder (BSB)* as shown in screenshot A.4). Make sure you do not forget to add the modules that are necessary to debug your hardware later by software (e.g. RS232_Uart, FPGA JTAG, etc.). Generation of some default test applications might also prove useful to find a quick entry into the software part (→ [Xil07b]).

Do not forget to enable the debugging-capability over the JTAG-cable for one of the two cores, otherwise the output of both may get mixed up completely - or even worse, the whole system may not function as soon as both cores try to use the same UART. To activate JTAG-debugging, go into the menu `DEBUG` → `DEBUG CONFIGURATION ...` as is done for the second core in screenshot A.5.

By default only one PowerPC-core is operational after setting up a new project. Therefore, the first step to do then is to make the second core operational too. After adding the second core make sure that all mandatory signals are connected - otherwise the core will not be sensible to any attempts to open a *GDB*-connection to it later on (it will simply not respond). One pitfall here is that in the `PORTS`-slide of EDK there are not all signals presented as default; to make sure you see all signals you must deactivate the filtering by choosing all signals as shown in screenshot A.7. Especially all *reset*- and *clock*-signals should be taken care of.

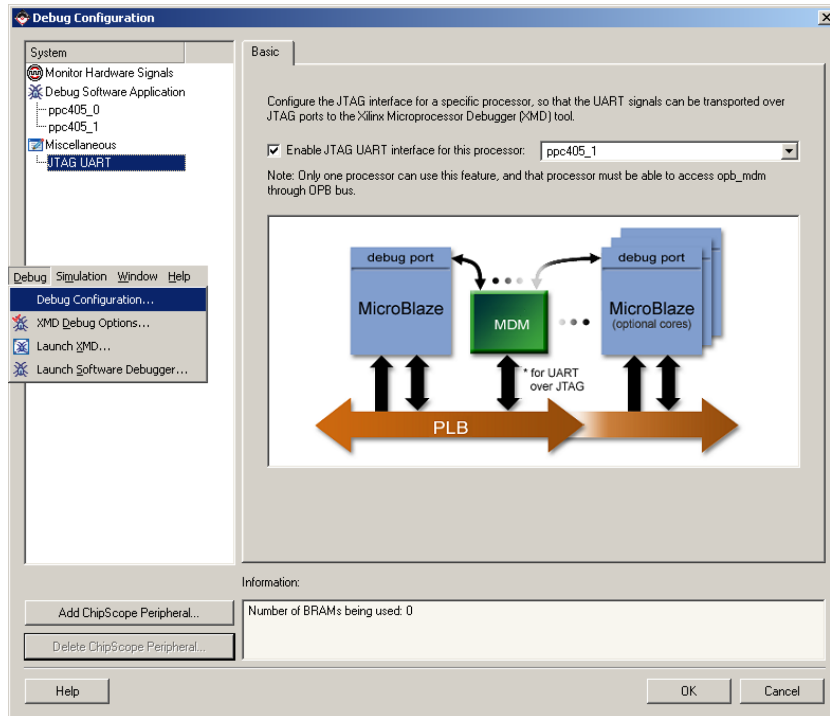


Figure A.5: Enabling debugging over JTAG for one core

The second core must get its own memory and other resources. In the following it is described how the configuration was established for the present thesis. First the basic system architecture was pinpointed, as is drafted in figure A.6. In difference to [Xil07a] where the shared memory is created using a shared region of the on-board SDRAM connected to the cores by the Intel *Peripheral Local Bus (PLB)*, we want to avoid the PLB here and use the *18*-kBit on-chip Block-RAMs to make our shared memory. The idea is to avoid using a bus to access our shared memory, since a bus, by definition, enforces *serialization* of the accesses to the shared memory.

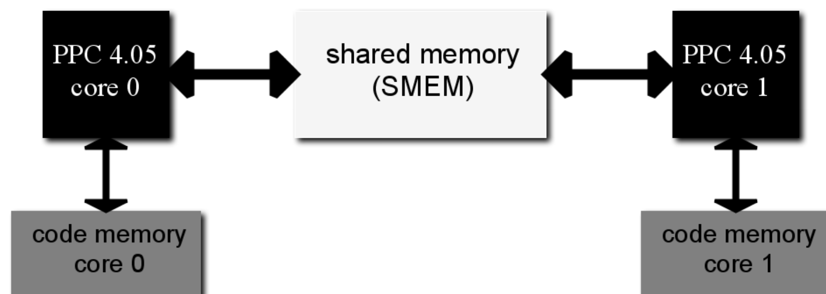


Figure A.6: draft of basic system-architecture

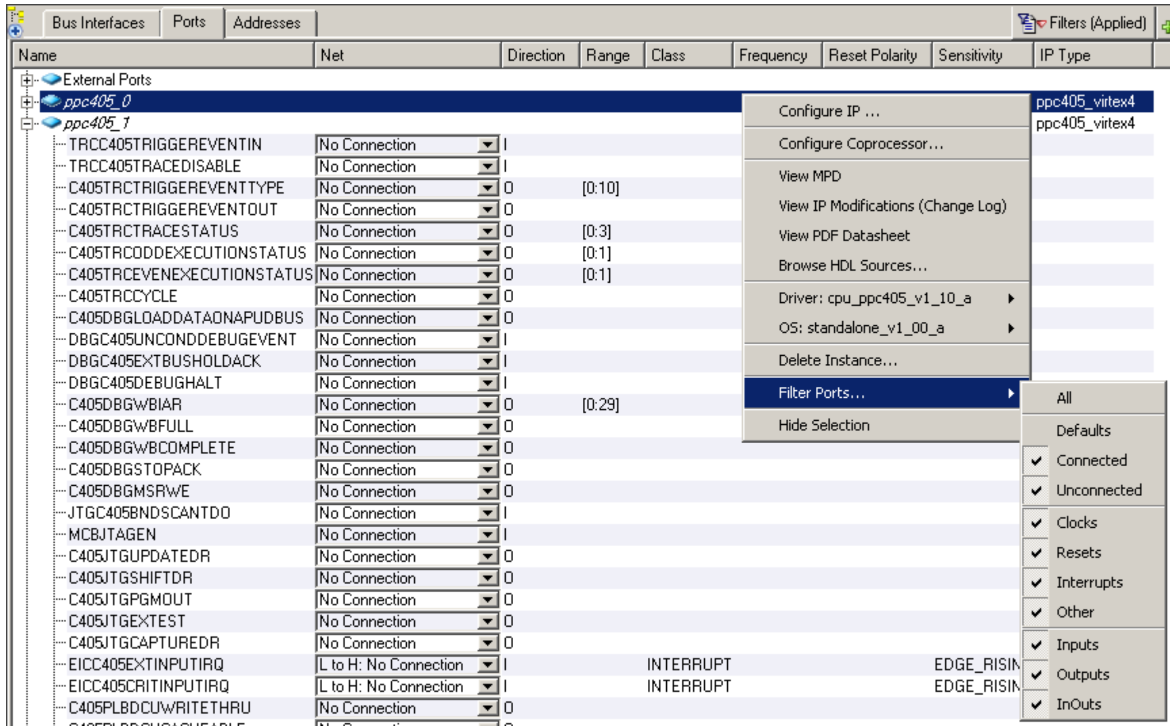


Figure A.7: deactivate port-filtering in the EDK

Each processor core can connect memory almost directly using the *on-chip memory controller (OCM)*. Fortunately there are *two* OCM-sides *per core*, the *data-side* and the *instruction-side* OCM. So it is also possible to create an instruction-memory private to each core, which avoids flooding the PLB by both core's instruction fetches targeting the SDRAM. The chain of modules (intellectual properties, IPs) necessary to add might not be clear at first, hence it is drafted in figure A.8.

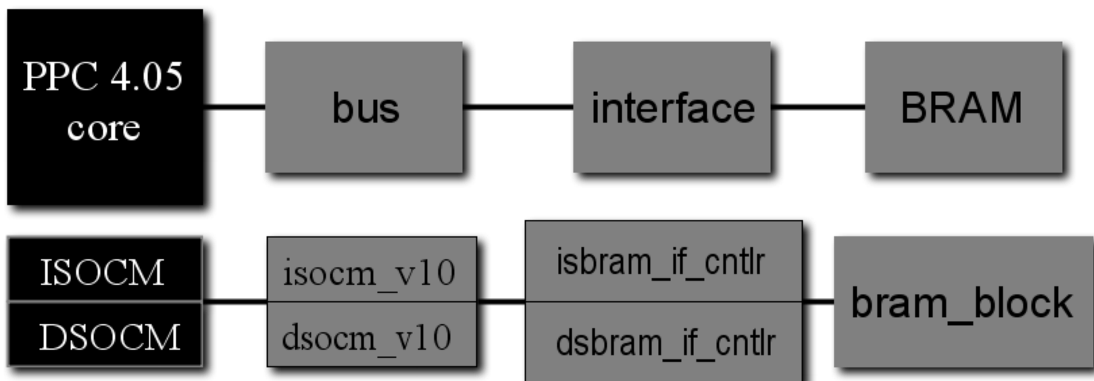


Figure A.8: OCM-chain of IPs

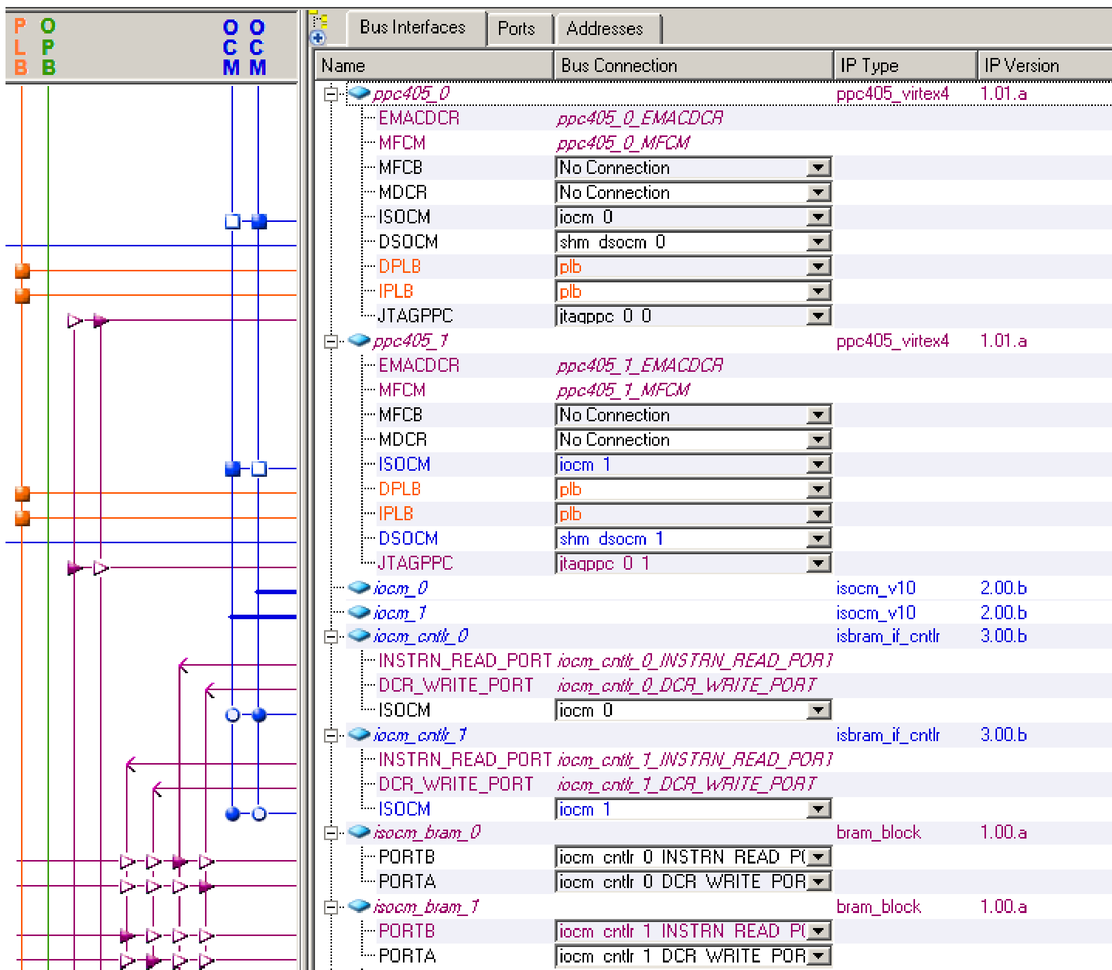


Figure A.9: ISOCM-chain, core-private

Making the connections for two instruction-side BRAM-blocks should give a result as shown in A.9. Details like the amount of memory used etc. should be determined considering the individual requirements and thus no default values are given here.

The connection of block-RAM with the two data-side OCM-controllers of the two PowerPC-cores may not be that obvious. The correct solution can be seen in A.10: each port of the dual-ported block-RAM is accessible by one of the two cores.

Block-RAM can be accessed relatively fast, but its size is limited. Even worse, the more blocks of RAM are organized together to form larger RAM-blocks, the slower the access might become. To avoid overloading the block-RAM it makes sense to use also SDRAM over the PLB, placing stack, heap and large arrays of data there. If no SDRAM is used as shared memory, the 64 MByte *SDRAM can be split up into two non-overlapping parts*, the first half dedicated to the first core, the second half used by the second one.

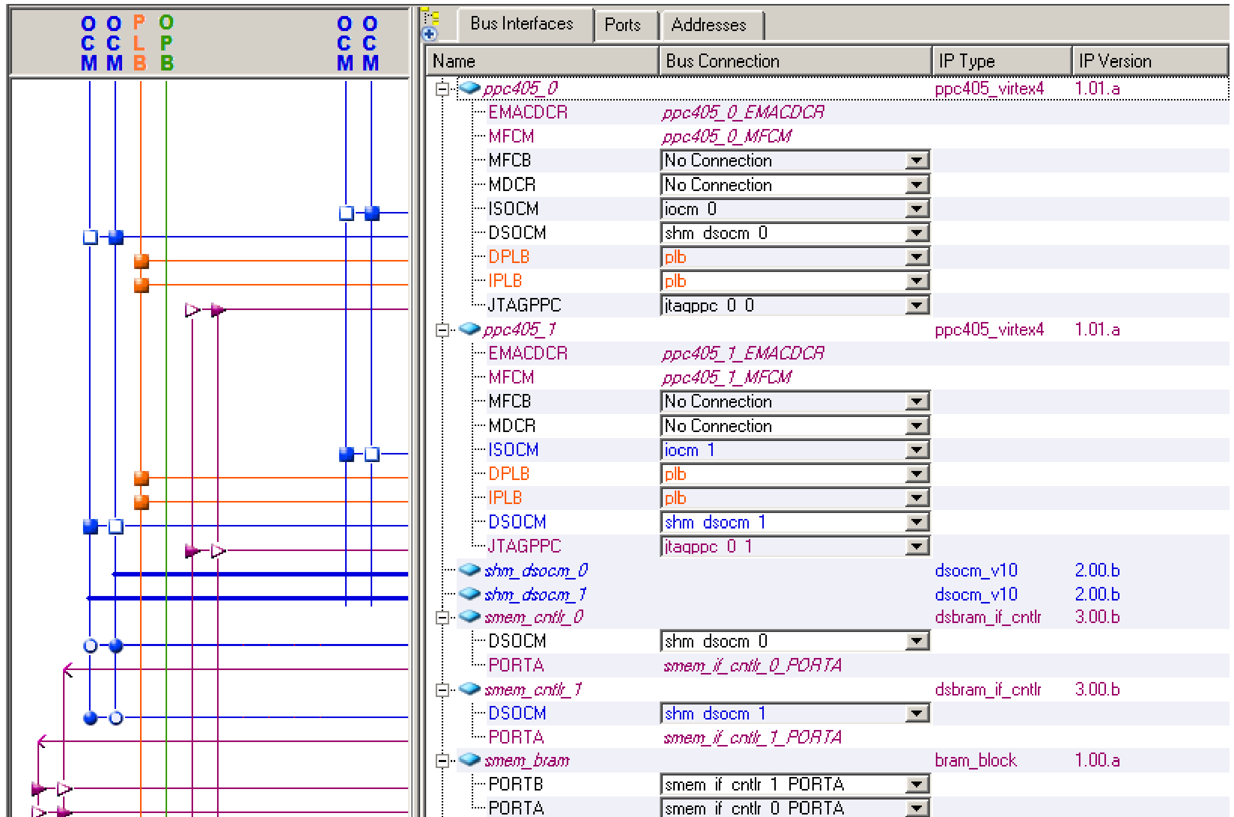


Figure A.10: DSOCM connections creating shared memory!

One should always be aware of the relatively low performance of SDRAM compared to the OCM-accessed BRAM. Hence SDRAM should be used with care, in particular when time-critical tasks are present in a system.

An overview of the hardware configuration of the whole system is given in A.11. This block diagram generated by the EDK can help in uncovering design-faults.

The last configuration-step after all connections are complete: *address-ranges* must be assigned to all IPs of the system. One must set, for instance, the address-ranges for both DSOCM-controllers to be the same, otherwise software-applications cannot be used on both cores without *re*-compilation after changing the shared-memory's base-address. For the example configuration of this thesis we give the address-ranges, as they were fixed in the EDK, in A.12. It must be pointed out that the automatic address-generation is not reliable in the case that both cores are about to be used. Manual editing may be required.

Told from experience, unfortunately it might happen that addresses of own IPs must be checked in the IP's own *MPD*-file as well, but the IPs from the Xilinx-libraries are generally all *generic*, so there should not be any inconsistencies. If in doubt, check the MPD-file of the respective IP. For more insight into working with Xilinx EDK we refer to [Xil07b] which works as a tutorial, as well as we recommend the EDK online documentation.

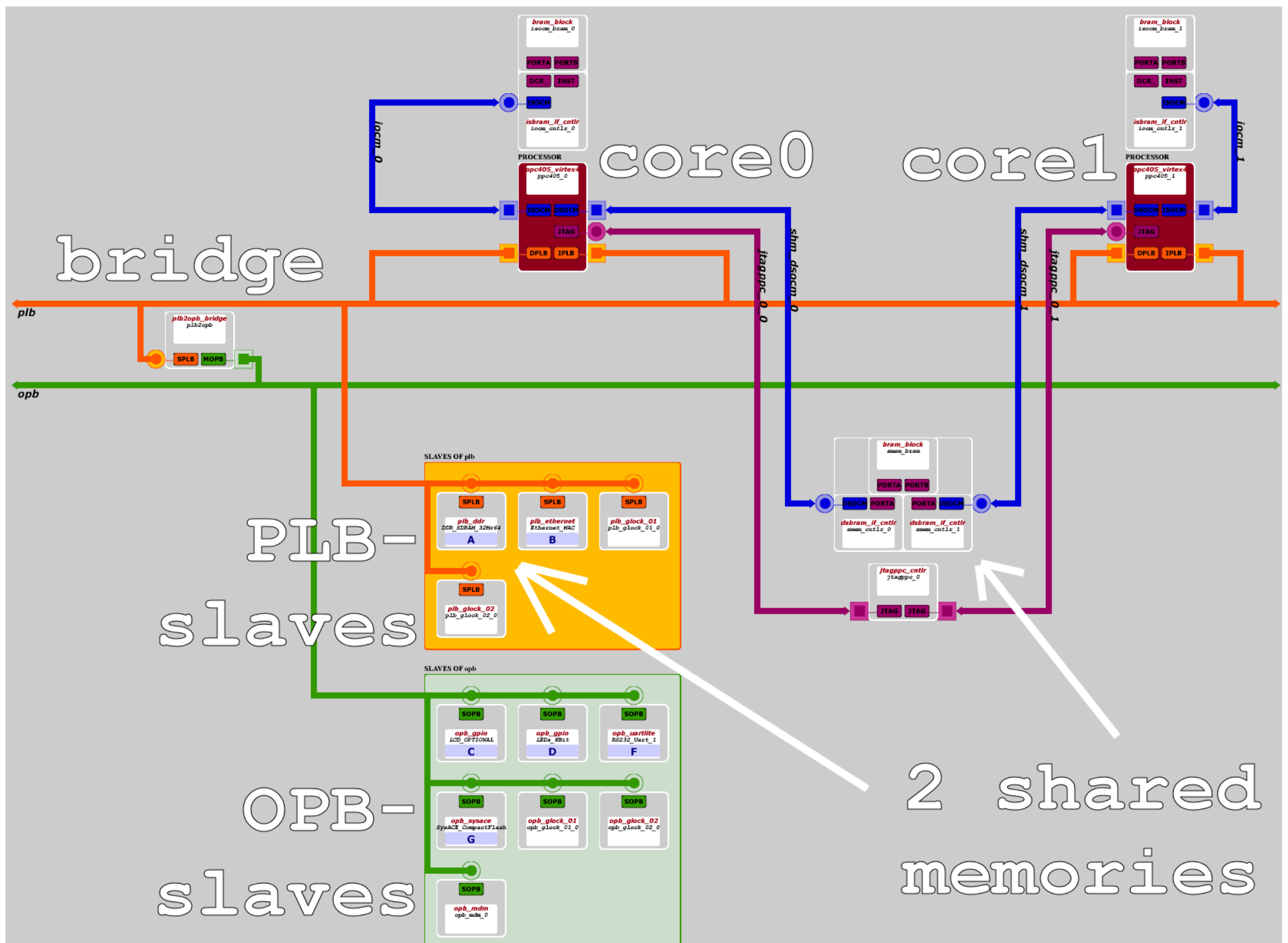


Figure A.11: Complete shared-memory system configuration

With all connections drawn and all address-ranges assigned, the configuration in the Xilinx EDK is finished. The EDK must be used to *generate netlists* for the Xilinx board based on the configured hardware design, it can also be used to complete the digital flow as a whole. However, the *transfer of the configuration into Xilinx ISE* is recommended for the experienced designer to get full control over the hardware-implementation. This is described in section A.2.3.

Bus Interfaces | Ports | Addresses

Instance	Name	Base Address	High Address	Size	Bus Interface(s)	Bus Connection	Lock	ICache	DCache	IP Type	IP Version
smem_cntlr_0	C_BASEADDR	*0xc0200000*	0xc0203fff	16K	DSOCM	shm_dsocm_0				dsbram_if_cntlr	3.00.b
smem_cntlr_1	C_BASEADDR	*0xc0200000*	0xc0203fff	16K	DSOCM	shm_dsocm_1				dsbram_if_cntlr	3.00.b
iocm_cntlr_0	C_BASEADDR	*0xffff0000*	0xffffffff	64K	ISOCM	iocm_0				isbram_if_cntlr	3.00.b
iocm_cntlr_1	C_BASEADDR	*0xffff0000*	0xffffffff	64K	ISOCM	iocm_1				isbram_if_cntlr	3.00.b
opb_glock_01_0	C_BASEADDR	*0x43000000*	0x43000000	1	SOPB	opb				opb_glock_01	
opb_glock_02_0	C_BASEADDR	*0x43100000*	0x43100000	1	SOPB	opb				opb_glock_02	
LEDs_8Bit	C_BASEADDR	*0x40000000*	0x4000ffff	64K	SOPB	opb				opb_gpio	3.01.b
LCD_OPTIONAL	C_BASEADDR	*0x40020000*	0x4002ffff	64K	SOPB	opb				opb_gpio	3.01.b
opb_mdm_0	C_BASEADDR	*0x41400000*	0x4140ffff	64K	SOPB	opb				opb_mdm	2.00.a
SysACE_CompactFlash	C_BASEADDR	*0x41800000*	0x4180ffff	64K	SOPB	opb				opb_sysace	1.00.c
RS232_Uart_1	C_BASEADDR	*0x40600000*	0x4060ffff	64K	SOPB	opb				opb_uartlite	1.00.b
Ethernet_MAC	C_BASEADDR	*0x80400000*	0x8040ffff	64K	SPLB	plb				plb_ethernet	1.01.a
plb_glock_01_0	C_BASEADDR	*0x04000000*	0x04000000	1	SPLB	plb				plb_glock_01	
plb_glock_02_0	C_BASEADDR	*0x04100000*	0x04100000	1	SPLB	plb				plb_glock_02	
plb2opb	C_DCR_BASEADDR			U	Not Connected					plb2opb_bridge	1.01.a
ppc405_0	C_IDCR_BASEADDR	*0b0100000000*	0b0100001111	16	Not Connected					ppc405_virtex4	1.01.a
ppc405_1	C_IDCR_BASEADDR	*0b0100000000*	0b0100001111	16	Not Connected					ppc405_virtex4	1.01.a
DDR_SDRAM_32Mx64	C_MEM0_BASEADDR	*0x00000000*	0x03ffffff	64M	SPLB	plb				plb_ddr	2.00.a
plb2opb	C_RNG0_BASEADDR	*0x40000000*	0x7fffffff	1G	SPLB	plb				plb2opb_bridge	1.01.a

Figure A.12: address-ranges of all IPs in design

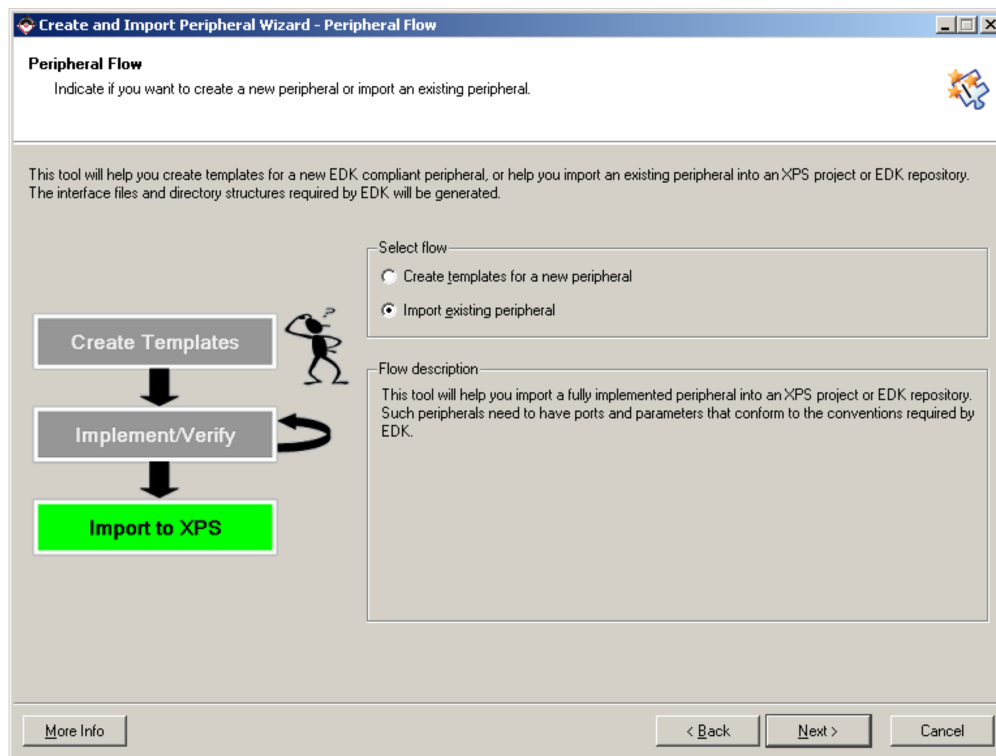


Figure A.13: The EDK peripheral wizard

A.2.2 Adding Individual IPs in Xilinx EDK

There are two possibilities to add own IPs into the hardware-configuration made with the Xilinx EDK. The easier way is to use the respective *wizard* in the EDK to do that. The way to *manually add* an IP goes along with porting the EDK-project to ISE. The latter approach is described in section A.2.3, the former here.

In chapter 5 of [Xil07b] is a detailed description on how an own IP is created using the wizard shown in A.13. But there is a slight inconvenience: the focus lies on using the Xilinx-specific interface called *IPIF (intellectual property interface)*. Using this interface destroys any chance that the created IP can be easily used on another hardware-platform. As much as IPIF may help a beginner in VHDL to avoid bus-specific code and to immediately create its own logic, it must be said that especially the low complexity of the OPB does not warrant abandoning genericity totally and turn to IPIF. Since it was one of the goals in this thesis to produce *generic* and *portable* code, the IPIF is not used to create own IPs here.

Instead of using IPIF we always import existing peripherals from our own VHDL source files. The steps through the wizard are quite simple to carry out, we do not go into too much detail here. What must be pointed out here are the common pitfalls that might occur in importing an own IP:

- **Incomplete interface**: The wizard helps in not allowing entities with incomplete port-list to be imported.

- **Erroneous interaction** with the bus: It is important for the stability of the whole system to be completely aware about the protocol of the bus the IP should connect to. For instance, if the bus demands a zero for all signals if the IP is inactive, but the IP switches to high instead, in the worst-case the whole system might as well hang up (e.g. OPB).

An imported IP can be easily added to the hardware-configuration. Some caution is recommended in configuring its attributes, it lies in the responsibility of the programmer of the IP to make the *logic configurable by generic parameters*. For instance, if the address decoding is done explicitly in hardware the IP might demand a *specific address*, and different addresses in EDK might lead to the software to access the IP using a false address, resulting in failure.

With a functional state-machine interfacing correctly to the respective bus and enabling data-transfers, the road is free to add any additional functionality into the VHDL-sources. The clear advantage is the possibility to use that code for any system where this bus is present, not just for Xilinx-systems.

A.2.3 Adding Individual IPs Manually: from EDK to ISE

The porting of an Xilinx EDK configuration to Xilinx ISE was done first, historically, due to an elementary problem in the Xilinx EDK 9.1i. The EDK 9.1i was unable to complete the flow for the configuration used for this thesis. In more detail, the error occurred in the *place&route* phase of the EDK (exact error message: ËRROR:Xflow in PAR: DeleteInterpProc called with active evals)̈. In the Xilinx online database was a hint on how to use the EDK-configuration in ISE and complete the flow without the EDK. With EDK 9.2i the problem was promised to be solved. Still, since the Xilinx ISE offers much more control over the design flow, the EDK was never used again for the whole flow. Thus, the completion of the hardware flow by ISE is a standard now, enabling also the insertion of user-defined logic where that would not be possible using solely the Xilinx EDK. In this section it is described in detail how **a configuration established in the Xilinx EDK is conveyed to the Xilinx ISE** - and what new opportunities appear by that.

Figure A.14: Xilinx ISE

There is a main drawback adding IPs using the peripheral wizard in Xilinx EDK 9.2i: an IP that is connected to the PowerPC-cores by the OCM is not supported and cannot be added to the configuration. This strikes one quite odd, even in the presentations of Xilinx the performance-gain by using OCM instead of the OPB or PLB is emphasized (s. [Xil07c]). Also, the use of the OPB and PLB are limiting the overall performance: OPB-IPs are connected via a PLB-to-OPB bridge (s. A.11), and both PLB and OPB are true busses, serializing and thus totally eliminating concurrency between the two PowerPC-cores. To add an IP that is not fed by *pre*-serialized data we have to leave those busses and go to the OCM. For instance, in A.11 the data-side OCMs of both cores connect to the same block-RAM serving as shared memory. An IP that connects to the core-side OCM-bus-controllers and the BRAM-controllers is the goal: in essence such an IP would work as a buffer for the incoming accesses to the shared memory - but an *intelligent buffer* that acts according to both core's requests.

It follows the complete list of steps that must be performed to add an IP into the design that was initially configured in Xilinx EDK 9.2i. The steps were developed by the author of this thesis by intensive analysis of the Xilinx-flow. Aliases used:

prj	...	name of project
submodule	...	name of own IP
EDK_prj	...	project-directory of the Xilinx EDK
ISE_prj	...	project-directory of the Xilinx ISE
/pcores	...	directory where all own IPs are stored subdirectory by subdirectory

1. **EDK**, make configuration as much as the EDK can help us to:
   - Add IPs
   - Configure IPs
   - Connect unconnected ports
   - Set address ranges

   With the configuration finished we must generate the netlists:

2. In the **Project Options**, check both boxes:
   - Implement design in ISE
   - Processor Design is a sub-module
     → This creates the internal sub-module ***prj_i***
   - Then activate the command `HARDWARE` → `GENERATE NETLIST`. This produces the `NGC`- and `VHD`-files in the directories *EDK_prj/implementation* and *EDK_prj/hdl*

With the netlists and the VHDL-files generated, the job of and in the Xilinx EDK is finished. As soon as the (hw-)configuration is fixed the EDK is only used if significant configuration-changes must be done.

```
=========================================================================
Timing constraint: Default path analysis
  Total number of paths / destination ports: 332 / 184
-------------------------------------------------------------------------
Delay:               3.511ns (Levels of Logic = 6)
  Source:            H0_DSOCM_WR_ADDR_VALID (PAD)
  Destination:       H0_S_BRAM_DSOCM_RDDBUS<22> (PAD)

  Data Path: H0_DSOCM_WR_ADDR_VALID to H0_S_BRAM_DSOCM_RDDBUS<22>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LUT2:I0->O            1   0.166   0.452  tr_reg_inp_0_or00001 (tr_reg_inp<0
>1>)
     LUT4:I3->O            4   0.166   0.605  nstate_0_mux0007<0>1 (nstate<0><0>
)
     LUT4:I1->O            2   0.166   0.608  granter_0_not0000_SW0 (N1506)
     LUT4:I1->O            1   0.166   0.000  H_S_BRAM_DSOCM_RDDBUS_0_mux0000<9>
21_G (N1776)
     MUXF5:I1->O           1   0.319   0.505  H_S_BRAM_DSOCM_RDDBUS_0_mux0000<9>
21 (H_S_BRAM_DSOCM_RDDBUS_0_mux0000<9>_map9)
     LUT3:I1->O            0   0.166   0.000  H_S_BRAM_DSOCM_RDDBUS_0_mux0000<9>
39 (H0_S_BRAM_DSOCM_RDDBUS<22>)
    ----------------------------------------
    Total                      3.511ns (1.341ns logic, 2.170ns route)
                                       (38.2% logic, 61.8% route)

=========================================================================
CPU : 34.13 / 363448.58 s | Elapsed : 34.00 / 363449.00 s

--> run -ifn ocm_access_cntrl.prj -ifmt mixed -top ocm_access_cntrl -ofn ocm_acc
ess_cntrl.ngc -ofmt NGC -opt_mode speed -opt_level 1 -p xc4vfx60-11 -iobuf NO_
```

Figure A.15: Synthesis of *IP ocm_access_cntrl* using XST

3. Create **Netlist of** your **own IP**
   A submodule is to be added *manually* into the netlists, therefore the configuration of the EDK (the VHDL- and NGC-files that were generated by it), must be *modified* to needlessly include this our own IP.

   - Write VHDL-code of submodule
   - *mkdir submodule* in directory */pcores*
   - Copy the HDL-files of the submodule to */pcores/submodule*
   - Create file */pcores/submodule/submodule.prj* with one such entry per source file:
     `vhdl work ⟨total path of HDL-source file⟩`
   - Generate the NGC-netlist of our own submodule using *Xilinx Synthesis Technology (XST)*:
     Go into your IP's project directory */pcores/submodule*, start XST (by the command of the same name) and synthesize your IP e.g. with the command at the bottom of A.15.

Please notice that the switch deactivating the buffering of the inputs, `-iobuf NO`, is absolutely necessary, otherwise buffers at input- and output-ports are inserted and the names of the nets do not match with the ones of the rest of the VHDL-hierarchy, resulting in errors and abortion of ISE when attempting to translate the whole design.

- Copy netlist *submodule.ngc* into *EDK_prj/implementation* - ISE will copy it from there along with the others.

The result of the last, the netlist */pcores/submodule/submodule.NGC* must be available for EDK/ISE when synthesizing the design!

4. **Modify HDL-sources** generated by Xilinx EDK - insert interface

Ideally, when the configuration is not changed furthermore in EDK, we must do the following only once:

- add own code to the VHDL-file */EDK_prj/hdl/prj.vhd*, it is the largest VHDL-file in that directory containing the code that interconnects all the board's IPs. Normally it will be necessary to define some new signals to reroute the signals meant for the BRAM-controller to the input-ports of our own IP! All what was done for this thesis was rerouting the BRAM-signals to our IP, and connecting the outputs of our IP with the BRAM-controllers.
Specifically:
- Add own component definitions
- Add additionally needed internal signals
- Set own component into logic
- Reroute signals to the inserted IP:
re-map ports in old component instantiations
- Save changes and make a backup-copy.

Caution!
In the case the command `GENERATE NETLIST` is executed again (in the EDK) all changes are lost - the VHDL-files are generated anew according to the EDK-configuration, overwriting any changes made afterwards!

5. **Transfer project to Xilinx ISE**

   - Open a new project
   - Copy *EDK_prj/hdl/prj_stub.vhd* into *ISE_prj* (contains top-module with, amongst others, sub-module *prj_i*)
   - Copy *EDK_prj/data/prj.ucf* to *ISE_prj*, then open the copy and edit the constraints as follows:

   ```
   NET "prj_i/C405RSTCORERESETREQ" TPTHRU = "RST_GRP";
   NET "prj_i/C405RSTCHIPRESETREQ" TPTHRU = "RST_GRP";
   NET "prj_i/C405RSTSYSRESETREQT̈PTHRU = "RST_GRP";
   ```

   - Add *ISE_prj/prj_stub.ucf* and *ISE_prj/prj.ucf* as existing sources to ISE-project
   - Add *EDK_prj/prj.xmp* as existing source. Now entering the EDK results in a notification about ISE handling the project.

6. Choose **Synthesize** in ISE to synthesize the design

   During Synthesis, ISE makes a new **top-netlist** using *prj_stub.vhd*. Luckily, because otherwise our changes to the VHDL-file would be ineffective. ISE analyses the VHDL-files like our modified *EDK_prj/hdl/prj.vhd*, but still do not need the netlists since we have our components defined literally as black boxes.

   Synthesis results mainly in the top-level netlist *prj_stub.NGC*, as long as the interface of our submodule and the VHDL-files of the rest of the design do not change we do not need to redo synthesis in ISE.

7. Choose **Implement Design** in ISE

   (a) **Translation** → ISE now copies all netlists from *EDK_prj/implementation*. With the NGC-file of our own IP present in this directory, there must not be any error during the Translation. If the netlist *submodule.ngc* was not copied into *EDK_prj/implementation* or the netlist is updated you can copy the netlist directly into *ISE_prj* (to keep consistency, also into *EDK_prj/implementation*), the ISE will indicate by question marks that there was a change and synthesis and the subsequent steps must be redone.

   The result of the translation is a **full logic description of the whole design** in the file *prj_stub.NGD*.

   (b) **Map** → The logical design of the NGD-file is now mapped onto the actual logic of the FPGA (be sure you chose the

right one). The result is an NCD-file *prj_stub.NCD* that can be used in the FPGA-editor. Other side-products of the mapping are PCF-(ASCII), NGM- and a MRP-(map report)-file.

(c) **PAR - place route** → The native circuit description (NCD-file from mapping), is complemented by detailed information about where the logic is placed on the FPGA. Other results are a PAR-, PAD-, CSV-, TXT- and GRE-file.

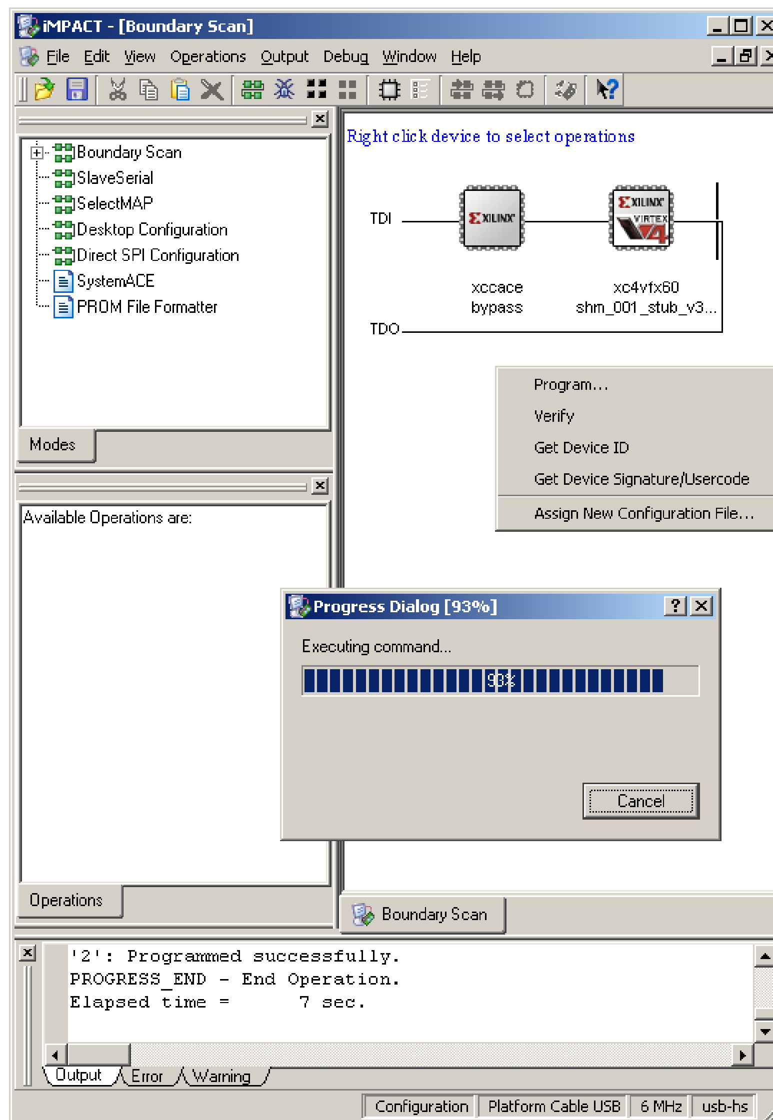


Figure A.16: Programming FPGA with iMPACT

8. Choose `GENERATE PROGRAMMING FILE` in ISE
   ISE takes the fully placed and routed NCD-design and produces a **configuration bit stream** as a binary file. The main result is the *ISE_prj/prj.BIT* - file that can be used by the *iMPACT*-program to load the FPGA with this design, shown in snapshot A.16.

9. Choose `CONFIGURE DEVICE` in ISE or load the iMPACT-program separately from the ISE-program folder `XILINX ISE 9.2i` → `ACCESSORIES` → `iMPACT`. In iMPACT, choose `EDIT` → `LAUNCH WIZARD`, select only the *BIT*-file as source and then load it with `PROGRAM` up to the FPGA as shown in snapshot A.16.

With all these previous steps carried out successfully, the FPGA-board is now ready to be programmed by software.

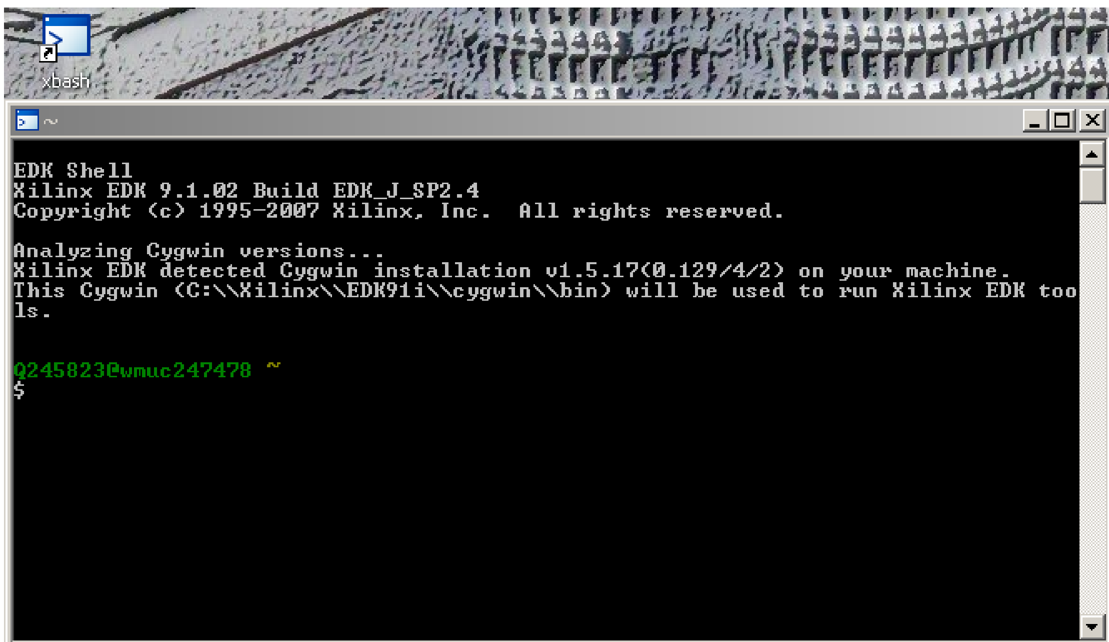


Figure A.17: *xbash*-prompt as universal tool to work with

A.3 Software Flow

Of main concern (with the hardware-design finished) is the correct connection to the two PowerPC-cores. There are some points that must be considered, otherwise the second or even both cores are unusable. The universal tool that helps us to do all necessary configuration is the *Cygwin*-prompt *xbash* shown in snapshot A.17.

A.3.1 Using the Xilinx Microprocessor Debugger (XMD)

We need one *xbash*-prompt for each core's application, and one prompt to prepare the two cores, means to make the uploading of any application possible at all. Why is that ? By default only one core is working. Loading up any application by its ELF-file normally resets the system or at least the PowerPC-core used by this application. Using both cores a *system-reset* always resets *both* of them, disrupting a possibly currently executing application on the other core. Therefore we have to configure our system such that uploading an application to any core does not affect the other core. We achieve this by using the *XMD* in the following way:

The first *xbash*-prompt is opened, then we execute the following script *./xs*:

```
cd EDK_prj
xmd -xmp prj.xmp
```

The first line moves us from our own network-directory to the local project directory of the Xilinx EDK (*/cygdrive/c/MCORE/Workspace/-main/MC_HW_Architecture/XILINX_PRJ/n2_ppc_shm001_OCM*). For our thesis the path is quite long, so it would be tedious to change the directory manually. The second line of the previous script actually starts the XMD with the argument of the EDK's project-file located in that directory. The following happens:

1. XMD searches for the file *xmd.ini* in the path XMD is launched, if it is found it is executed first.

2. XMD analyses the file *prj.xmp* to set up its configuration for connecting to the hardware-design, that is the PowerPC cores active in it. When we use both cores this is apparent in the project-file and XMD will give out information about both cores at start-up - shown in snapshot A.18.

Executing the initialization-file *xmd.ini* XMD ensures that no reset is done when uploading an ELF-file to one of the cores, done by the following commands:

```
connect ppc hw -debugdevice cpunr 1
debugconfig -reset_on_run disable
connect ppc hw -debugdevice cpunr 2
debugconfig -reset_on_run disable
terminal tcp
```

Thus, by executing the script *./xs* the hardware design is prepared to take programs for execution. Snapshot A.18 shows how starting XMD should look like for a dual-core system; using a single-core system results in output without the blocks of information regarding the second core.

For details on how to use XMD please see [Xil07d], chapter 12.

```
$ ./xs
Xilinx Microprocessor Debug (XMD) Engine
Xilinx EDK 9.2.01 Build EDK_Jm_SP1.2
Copyright (c) 1995-2007 Xilinx, Inc.  All rights reserved.

XMD%
Processor(s) in System ::

PowerPC405(1) : ppc405_0
Address Map for Processor ppc405_0
  (0b0100000000-0b0100001111) ppc405_0
  (0000000000-0x03ffffff) DDR_SDRAM_32Mx64       plb
  (0x04000000-0x04000000) plb_glock_01_0         plb
  (0x04100000-0x04100000) plb_glock_02_0         plb
  (0x40000000-0x4000ffff) LEDs_8Bit      plb->plb2opb->opb
  (0x40020000-0x4002ffff) LCD_OPTIONAL  plb->plb2opb->opb
  (0x40600000-0x4060ffff) RS232_Uart_1  plb->plb2opb->opb
  (0x41400000-0x4140ffff) opb_mdm_0      plb->plb2opb->opb
  (0x41800000-0x4180ffff) SysACE_CompactFlash   plb->plb2opb->opb
  (0x80400000-0x8040ffff) Ethernet_MAC  plb
  (0xc0200000-0xc0203fff) smem_cntlr_0  shm_dsocm_0
  (0xffff0000-0xffffffff) iocm_cntlr_0  iocm_0

PowerPC405(2) : ppc405_1
Address Map for Processor ppc405_1
  (0b0100000000-0b0100001111) ppc405_1
  (0000000000-0x03ffffff) DDR_SDRAM_32Mx64       plb
  (0x04000000-0x04000000) plb_glock_01_0         plb
  (0x04100000-0x04100000) plb_glock_02_0         plb
  (0x40000000-0x4000ffff) LEDs_8Bit      plb->plb2opb->opb
  (0x40020000-0x4002ffff) LCD_OPTIONAL  plb->plb2opb->opb
  (0x40600000-0x4060ffff) RS232_Uart_1  plb->plb2opb->opb
  (0x41400000-0x4140ffff) opb_mdm_0      plb->plb2opb->opb
  (0x41800000-0x4180ffff) SysACE_CompactFlash   plb->plb2opb->opb
  (0x80400000-0x8040ffff) Ethernet_MAC  plb
  (0xc0200000-0xc0203fff) smem_cntlr_1  shm_dsocm_1
  (0xffff0000-0xffffffff) iocm_cntlr_1  iocm_1

Info:AutoDetecting cable. Please wait.
Info:Connecting to cable (Usb Port - USB21).
Info:Checking cable driver.
Info: Driver version: 1027 (1027).
Info: Driver windrvr6.sys version = 8.1.1.0.Info: WinDriver v8.11 Jungo (c) 1997
 - 2006 Build Date: Oct 16 2006 X86 32bit SYS 12:35:07, version = 811.
PORT_INDEX = 0.
CB_CABLE_COUNTER = 0.
DeviceAttach: Cable found for usb21.
Calling setinterface num=0, alternate=0.
DeviceAttach: received and accepted attach for:
  vendor id 0x3fd, product id 0x8, device handle 0x2d00038
Info: Cable PID = 0008.
Info: Max current requested during enumeration is 280 mA.
Info:Type = 0x0605.
Info: Cable Type = 3, Revision = 0.
Info: Setting cable speed to 6 MHz.
Info:Cable connection established.
Info:Firmware version = 1027.
Info:File version of C:/Xilinx/ISE92i/data/xusbdfwu.hex = 1027.
Info:Firmware hex file version = 1027.
Info:CPLD file version = 0012h.
Info:CPLD version = 0012h.

JTAG chain configuration
--------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       0a001093           8        System_ACE
 2       21eb4093          14        XC4VFX60

PowerPC405 Processor Configuration
-------------------------------------
Version.............................0x20011470
User ID.............................0x00000000
No of PC Breakpoints................4
No of Read Addr/Data Watchpoints....1
No of Write Addr/Data Watchpoints...1
ISOCM...............................0xffff0000 - 0xffffffff
User Defined Address Map to access Special PowerPC Features using XMD:
        I-Cache (Data)........0x70000000 - 0x70003fff
        I-Cache (TAG).........0x70004000 - 0x70007fff
        D-Cache (Data)........0x78000000 - 0x78003fff
        D-Cache (TAG).........0x78004000 - 0x78007fff
        DCR...................0x78004000 - 0x78004fff
        TLB...................0x70004000 - 0x70007fff

Connected to MDM UART Target
Connected to "ppc" target. id = 0
Starting GDB server for "ppc" target (id = 0) at TCP port no 1234

PowerPC405 Processor Configuration
-------------------------------------
Version.............................0x20011470
User ID.............................0x00000000
No of PC Breakpoints................4
No of Read Addr/Data Watchpoints....1
No of Write Addr/Data Watchpoints...1
ISOCM...............................0xffff0000 - 0xffffffff
User Defined Address Map to access Special PowerPC Features using XMD:
        I-Cache (Data)........0x70000000 - 0x70003fff
        I-Cache (TAG).........0x70004000 - 0x70007fff
        D-Cache (Data)........0x78000000 - 0x78003fff
        D-Cache (TAG).........0x78004000 - 0x78007fff
        DCR...................0x78004000 - 0x78004fff
        TLB...................0x70004000 - 0x70007fff

Connected to "ppc" target. id = 2
Starting GDB server for "ppc" target (id = 2) at TCP port no 1236
JTAG-based Terminal Server.
(TCP Port no used is 4321)
XMD%
```


Figure A.18: Connect to dual-core design using XMD

A.3.2 Upload of Applications to the PowerPC Cores

With XMD started (like described in the previous section) we can upload programs to both cores. This is done by opening two new *xbash*-prompts. It is important not to use one prompt for both applications, strange effects like corrupt data-packages can occur in that case. One prompt per application, then there are no problems.

In a prompt we have to specify the ELF-file to use for the upload, that is the application we want to execute on a core. For the upload we use *GNU Debugger (GDB)* as explained in [Xil07d], chapter *11.* There are two possible commands:

1. `powerpc-eabi-gdb -nw prj.elf`
   Here the GDB-prompt is opened where commands must be typed in.
   The option *-nw* prevents using the GUI.

2. `powerpc-eabi-gdb prj.elf`
   Now the GUI is loaded. It simplifies stepping through the application's program by showing the code. But working with two cores the GUI is not always very comfortable at all (as one might notice trying it out).

As an example we show how to execute the two applications *prove_core0* and *prove_core1*. In the former, meant for the first PowerPC-core, the bits of a memory-word in the shared memory is toggled in an infinite loop. In the latter application the second core reads this memory-word and checks it for values others than the two allowed values, thus proving or disproving the coherence of the shared memory.

First XMD is initialized as described before, then we open one prompt per core, in each prompt using GDB to load the ELF-file wanted for the respective core. Then in an GDB-prompt we select the wanted core using

`target remote localhost:⟨port_number_of_core⟩`

The number of the port must be one of the two distinct port-numbers given for both cores by XMD, the numbers assigned by XMD at start-up can be obtained by looking at the output of XMD. Look at A.18 as an example. Using this example we establish a GDB-connection to core *0* via port *1234*, and core *1* via port *1236*.

After a connection was successfully established we can upload the ELF-application by the command `load`. Actual execution is started by **`c`** (for `continue`).

To display the output of the two applications the program *hyperterminal* can be used, as is shown in the example desktop given in snapshot A.19.

A.3.3 The Xilinx Software Development Kit (SDK)

Despite the fact the Xilinx EDK offers the possibility to generate some first useful testing applications for one's configuration - managing those applications in the

Figure A.19: Essential dual-core working environment

EDK is not very comfortable. Here the *Eclipse*-based *Xilinx Platform Studio SDK* is the right tool for serious development. Of course with the SDK you can create applications from scratch, but it is recommended to use the test-applications already generated by the Xilinx EDK, located in the directory *EDK_prj/SDK_projects*. Those applications generated by the Xilinx EDK can be used as a starting point for own programs. Still there are some major points one must be aware of when programming parallel programs with the SDK. The points are given in the following and can be taken for creating new dual-core applications. Assuming functional test-applications for the cores were created by the Xilinx EDK, we present a way to get two new applications by *duplicating* two already available applications. It holds for core *0* and core *1* respectively:

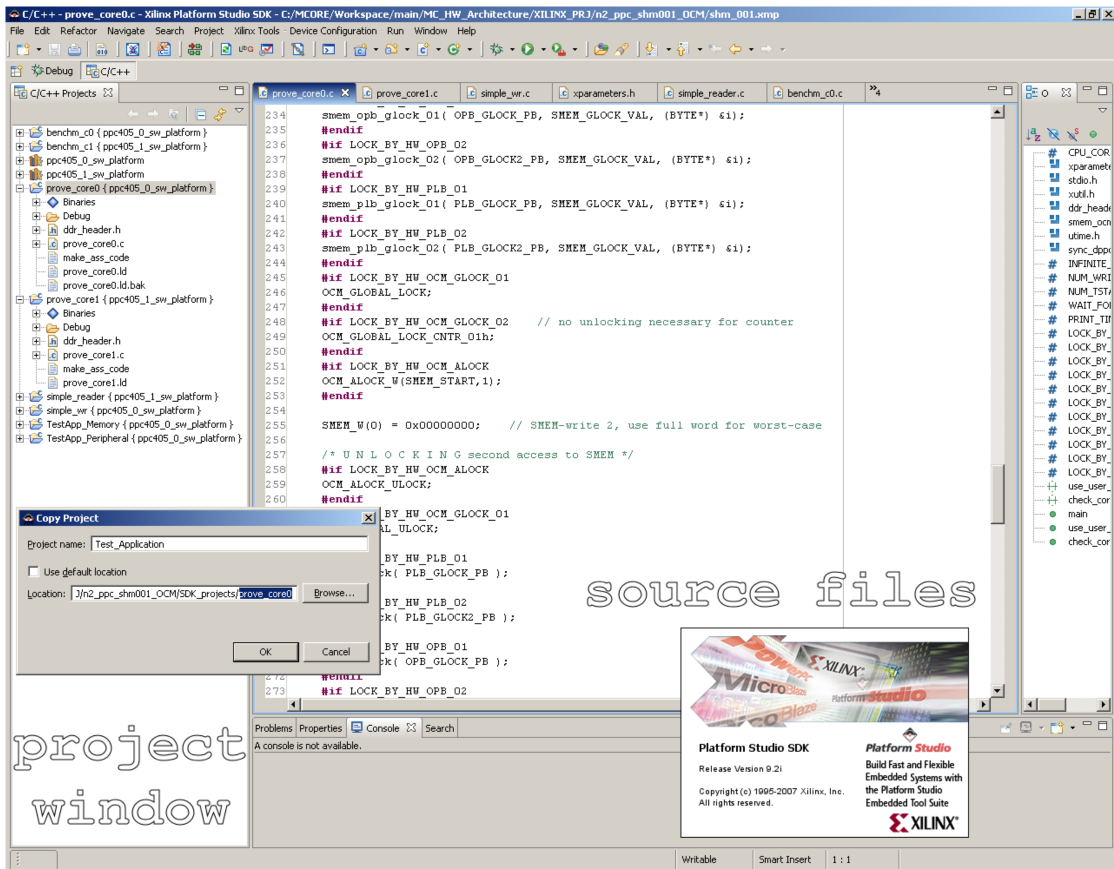


Figure A.20: Working environment of the Xilinx SDK

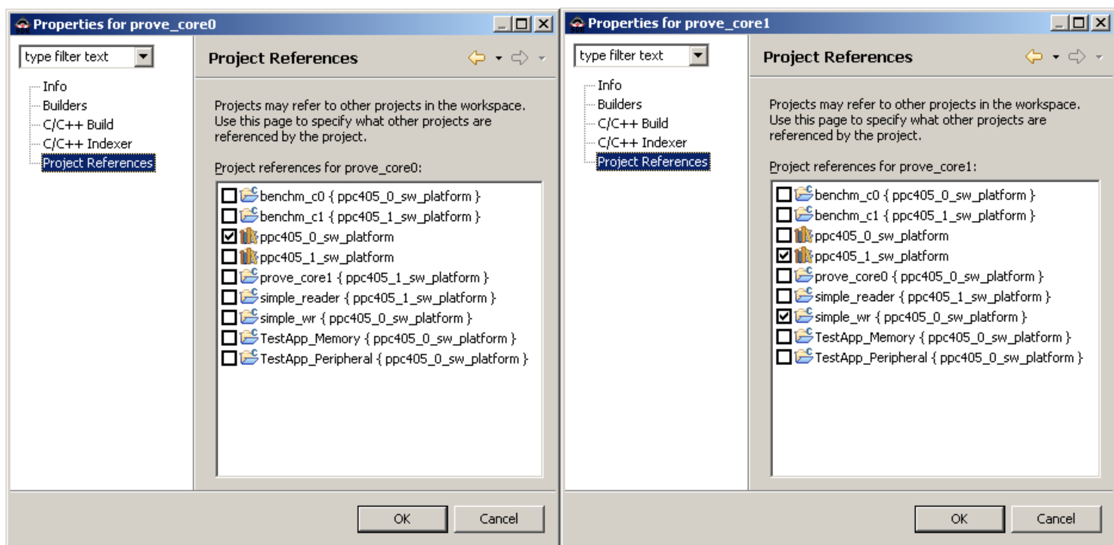


Figure A.21: References essential to core0- and core1- programs

Duplicating a core*0/1*-application:

- Copy an *existent* functional core-*0* or core-*1* project in the project window and paste it in with its new name (e.g. in snapshot A.20 *prove_core0* is about to be duplicated).

- Control the *dependencies* of the project in the `project properties` (right-click on project in project-window). In our case a core-*0* project has only a dependence on the PowerPC core-*0* - platform, a core-*1* on the other hand is also dependent on a core-*0* application to compile correctly (s. snapshot A.21 as a reference how to do that).

- Adapt the `project properties`. The most important points are:

  - Add the *path* where all your header-files are located as an *include path* (A.22 upper left).
  - Use *compiler optimization* with caution, maybe even deactivate it. For debugging *debug-symbols* must be added to the ELF-file (→ A.22 upper right).
  - When an *assembler-library* is present and used, it and its path must be added (→ A.22 lower left).
  - A *linker script* is essential to communicate the right *address-ranges* (→ A.22 lower right. It can be generated automatically (right-click on project, then snapshot A.23) and can then easily be modified for dual-core-specific purposes like changing the base-address in the SDRAM to divide it between the two cores. The effect can be observed when loading an ELF-file onto a core: all relevant address-areas are displayed in the GDB-prompt (see snapshot A.19).
    The linker script is of *vital importance*. Such as for some complex systems manual changes might always be necessary. Also, with an application expanding in nested function-calls, the size of the stack might become an issue.

- Inspect the linker script for **non-overlapping of all segments** regarding the applications for both core-0 and core-*1*. For instance, if the data in the SDRAM is located for both cores at the same base-address, the currently running core will be corrupted when loading the ELF-file to the second one (it might look like a reset of the other, but it is not). As an example snapshot A.24 shows the beginning of the two *asymmetrical* linker scripts of two applications used in this thesis.

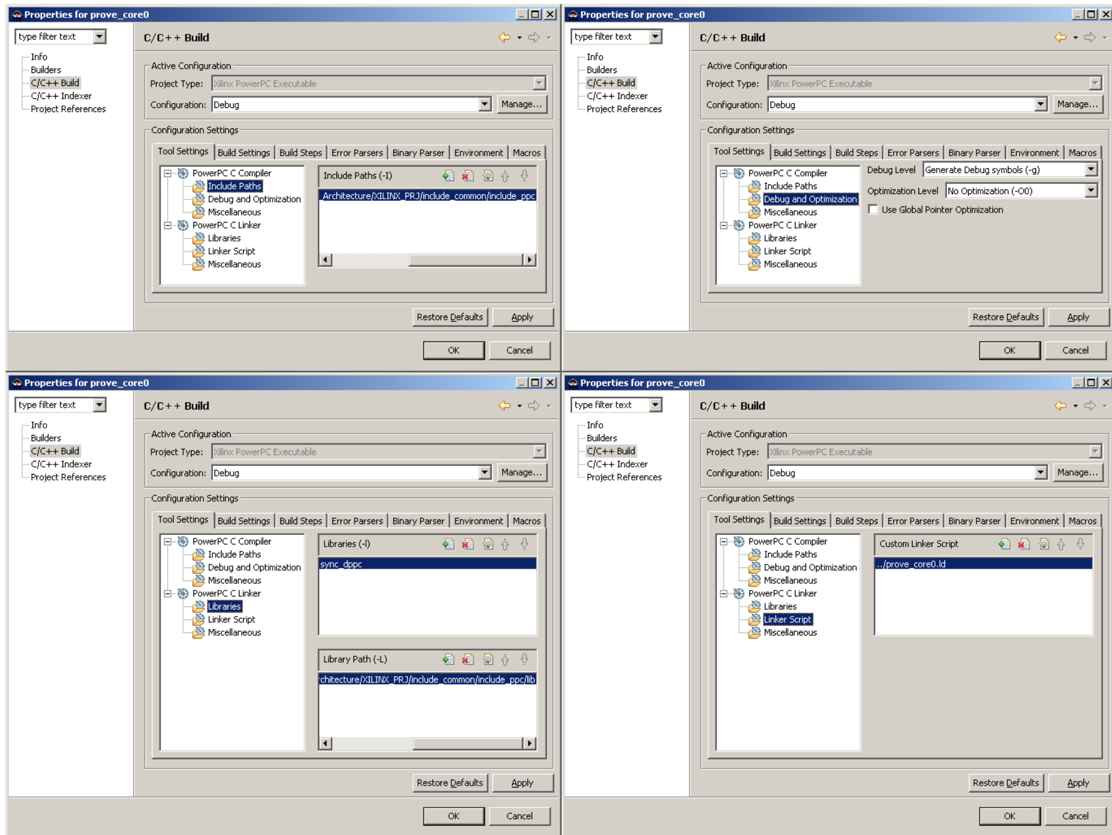


Figure A.22: Important project-properties for core0 and core1

- Rename all relevant, delete all irrelevant files, but the subdirectory *Binaries* and *Debug* can be left almost untouched - their contents (like Makefile etc.) are generated automatically when the rest is changed.

After the ELF-file of an application has been created feel free to use it to upload it on a proper hardware-design (s. previous section A.3.2).

A.3.4 Single-Core Application-Management by the SDK

With a second core inexistent (or left idle) the possibility to execute applications directly from the SDK is comfortable. To do so one must go to `RUN` → `RUN...` and make an entry for the application there (e.g. see snapshot A.25). Then XMD is executed automatically and even step-by-step debugging is possible in the SDK, making the development process as a whole much more efficient. However, using both cores at the same time unfortunately overburdens the capabilities of the current Xilinx Platform Studio SDK, resulting in the somewhat delicate procedure described in the previous sections.

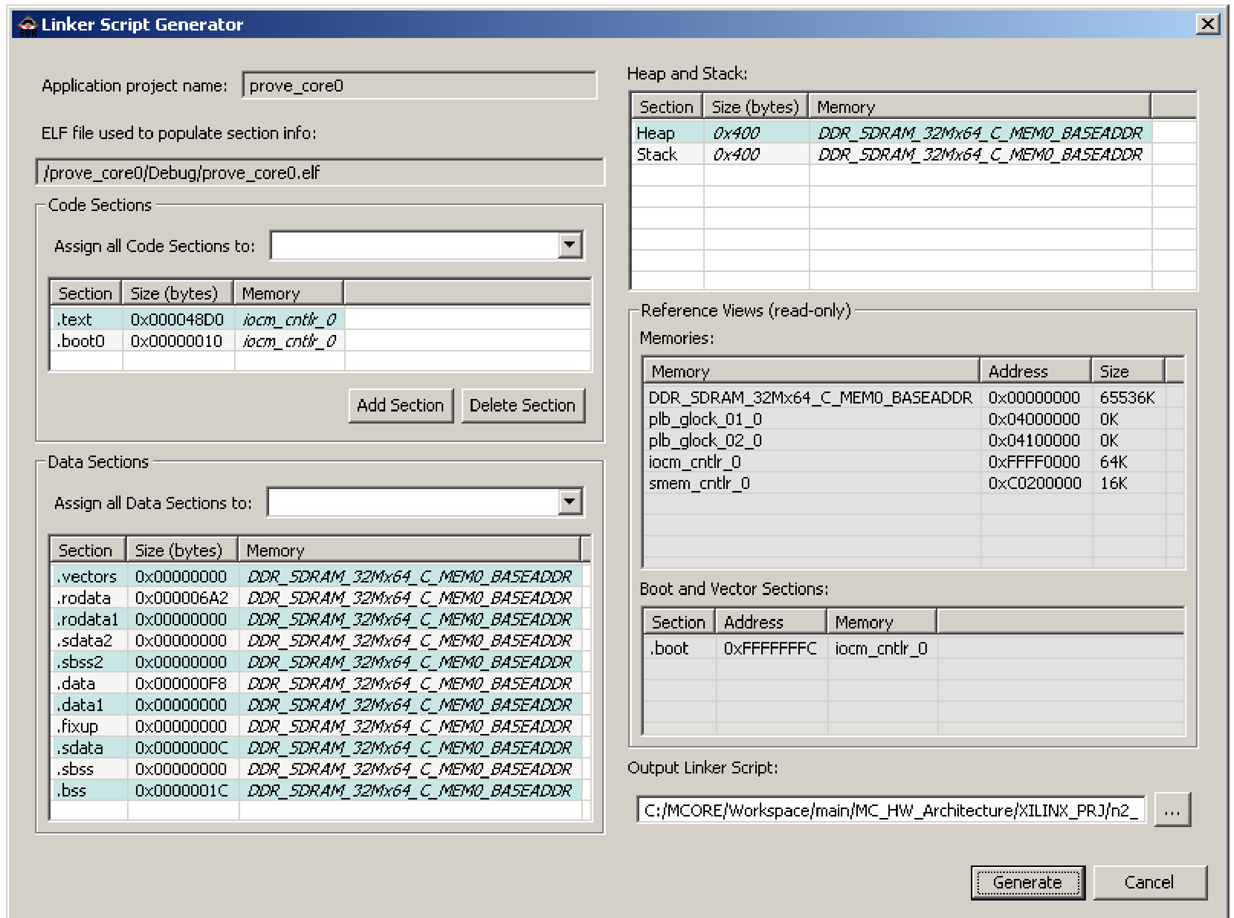


Figure A.23: Generation of a linker script in the SDK

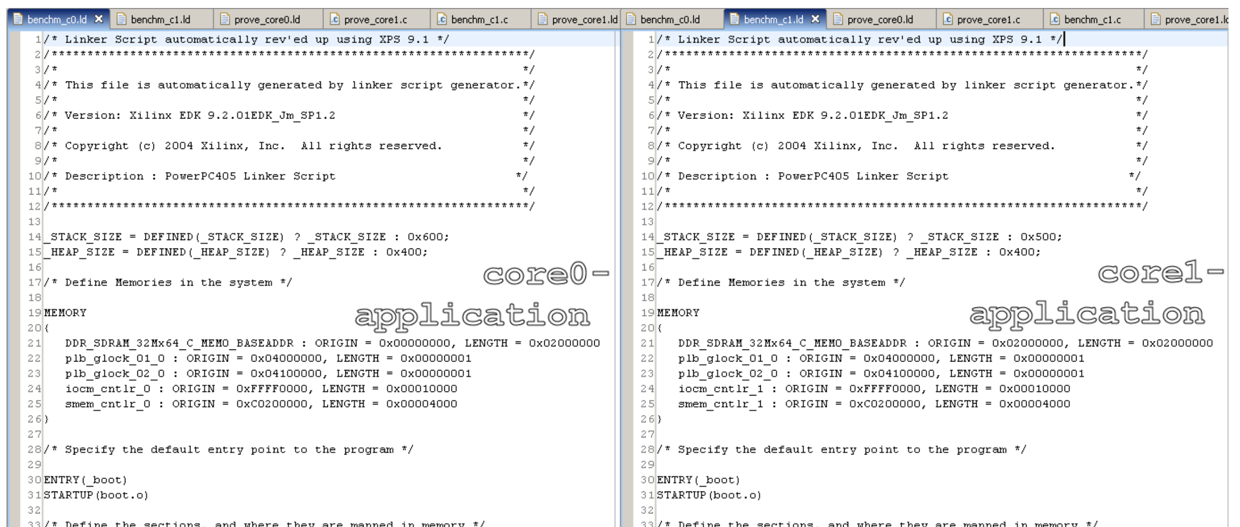


Figure A.24: Asymmetrical linker scripts for core 0 and core 1

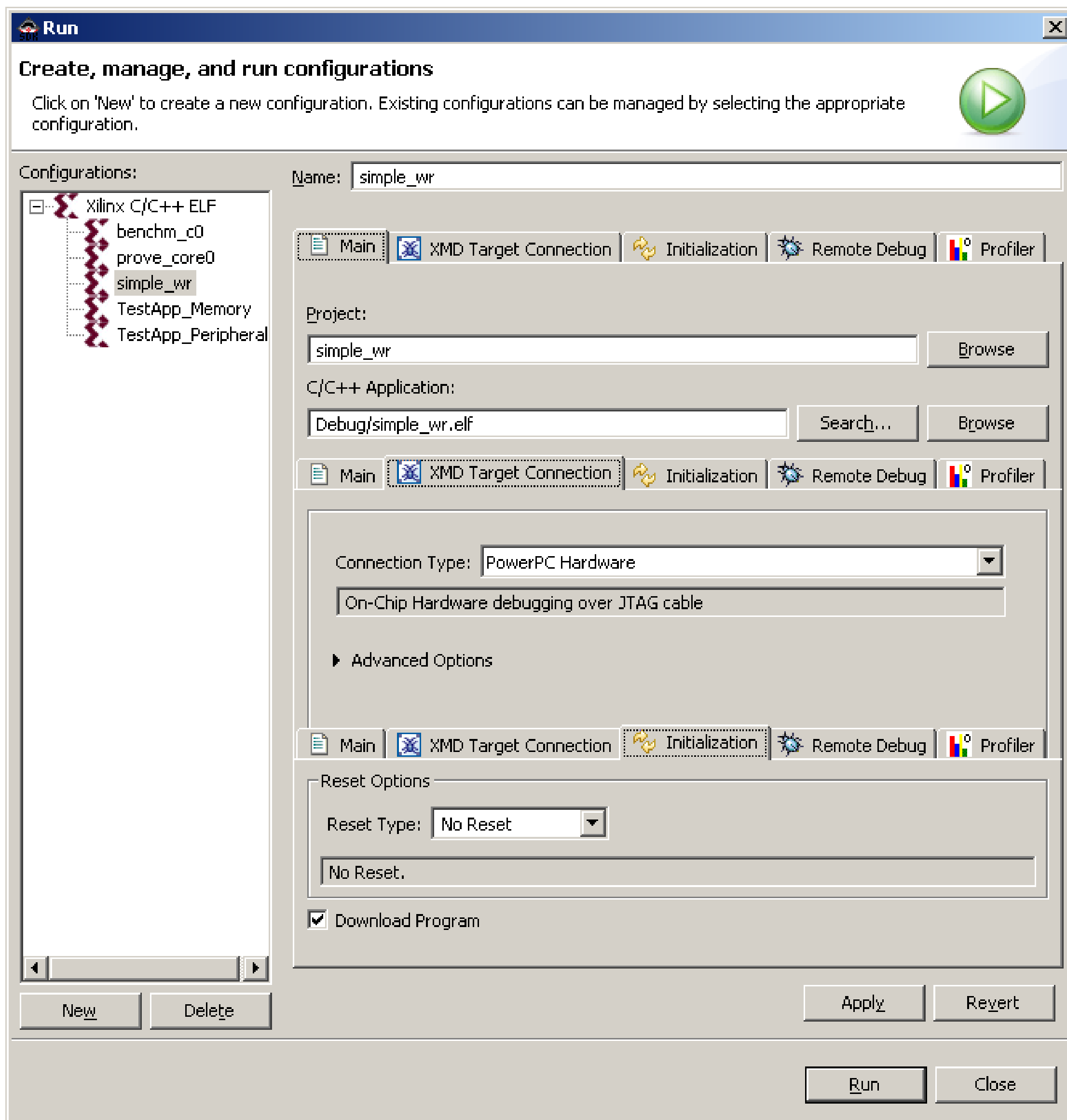


Figure A.25: Run-configuration of the SDK (only for single-core applications)

A.3.5 The Usage of Assembler

Early in making this thesis it became clear that the help of *assembler-subroutines* is necessary to achieve expressive results (for instance in the worst-case). Therefore a subdirectory in the include-directory was made, along with an assembler source file and a script to assemble it. The content of this script is:

```
powerpc-eabi-gcc -g -c -O0 sync_dppc.s
mv sync_dppc.o ../lib/libsync_dppc.a
```

The first line assembles the source file *sync_dppc.s* with no optimizations (`-O0`), without linking (`-c`) and with inserting platform-specific debugging information (`-g`). For details on how to handle the compiler see the *Options summary* - section in the manual of the GNU compilers ([GP05], chapter 3, section 1). In the first line the resulting *object*-file is moved to the library-directory where the SDK reads it from. Notably, a PowerPC-derivative of the *GNU compiler collection (GCC)* is used for all compiling, also to assemble our assembler-file into a library-file *libsync_dppc.a*. The directory where it is put to must be specified in the project's properties in Xilinx SDK. The file-extension `a` of the library-file is cut there - see snapshot A.22, lower left, as an example.

There is one more but less mentioned usage for assembler in this thesis: **assembling** the **ANSI-C sources** is done quite frequently to analyze the output of the GCC for the applications developed - and, e.g. when coping with the barrier-problem where a write-command alone didn't block the core at the wished point in the program - to find the *sources* of concurrent problems. Only the actual assembler-sources give a low-level point of view that covers all questions regarding the software-part of the digital flow! It might be unavoidable in a low-level development to use this help when needed.

During this thesis, in each project-directory of a software-application there was the script `make_ass_code` to quickly generate the assembler sources. The script executes the following command:

```
powerpc-eabi-gcc -c -I./../ppc405_0_sw_platform/ppc405_0/include
-I../../../include_common/include_ppc
-g -O0 -S -o prove_core0.o ./prove_core0.c
```

Two switches and their arguments give the paths of the Xilinx- and our own header-files. The GCC does not link (`-c`), makes debugging information (`-g`), does not optimize (`-O0`), stops after translating to assembler and before assembling (`-S`), thus the assembler-code is the final output. The name of this assembler output-file is explicitly given (`-o`). See [GP05], chapter 3, section 1, for detailed explanation.

A.3.6 The Directory-Structure

An overview of the final directory-structure grown over the duration of this thesis is given in figure A.26. The current state and contents of the directories are described there. For more details go into the top-level directory `MC_HW_Architecture` and open the text file `INFO_CONTENTS_DETAILED.TXT` for an even more detailed view on the directory-hierarchy.

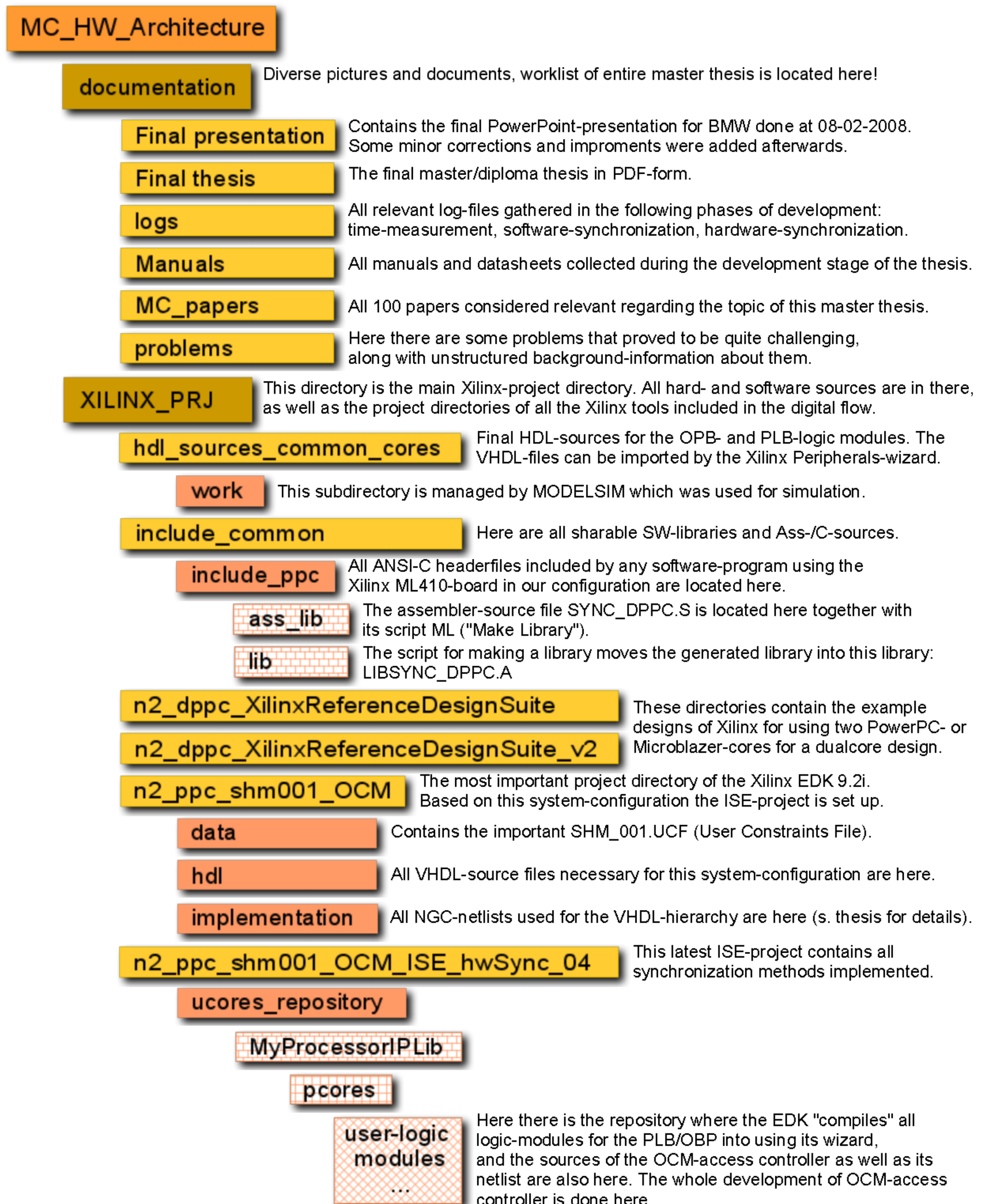


Figure A.26: The final directory structure evolved over this thesis

References

[CL97] Chandra and Larus. *Optimizing communication in HPF programs on fine-grain distributed shared memory.* 1997. University of Wisconsin, Madison.

[CM88] A. Chang and M. Mergen. *801 storage: Architecture and programming.* ACM Trans. Comput. Syst., 1988. Vol. 6(1), pp. 28-50.

[CS93] Shun Yan Cheung and Vaidy S. Sunderam. *Performance of Barrier Synchronization Methods in a Multi-Access Network.* 1993. Emory University, Atlanta, Georgia.

[CS99] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture, a hw/sw approach.* Morgan Kaufmann Publishers, Inc., Editorial and Sales Office, San Francisco, U. S. A., 1999. ISBN 1-55860-343-3.

[Dij65] E. Dijkstra. *Cooperating Sequential Processes.* 1965. Technological University, Eindhoven, The Netherlands.

[ea87] E. H. Jensen et al. *A new approach to exclusive data access in shared memory multiprocessors.* Lawrence Livermore National Laboratory, 1987. Technical Report UCRL-97663.

[ea89] August et al. *Cray X-MP: The Birth of a Supercomputer.* 1989. Cray Research.

[ea90] Rajiv Gupta et al. *The Design of a RISC based Multiprocessor Chip.* 1990. University of Pittsburgh, Philips Laboratories New York.

[ea93] Aboulenein et al. *Hardware support for synchronization in the scalable coherent interface (SCI).* 1993. University of Wisconsin, University of Oslo, Northwestern University.

[ea95] Arpaci et al. *Empirical Evaluation of the CRAY-T3D: a compiler perspective.* 1995. University of California, Berkely.

[ea02] Stolberg et al. *HiBRID-SoC: A multi-core System-on-Chip architecture for multimedia signal processing applications.* 2002. Universitt Hannover, Germany.

[ea04a] Kumar et al. *Conjoined-core chip multiprocessing.* 2004. University of California, San Diego, HP Labs, Palo Alto.

[ea04b] L. Hammond et al. *Transactional memory coherence and consistency.* Proc. 31st Annu. Int. Symp. on Computer Architecture, 2004. pp. 102-113.

[ea04c] Takayanagi et al. *A dual-core 64b UltraSPARC Microprocessor for Dense Server Applications.* 2004. Sun Microsystems, Sunnyvale, CA, USA.

[ea05a] Kumar et al. *Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling.* 2005. University of California, San Diego, IBM TJ Watson Research Center.

[ea05b] Sampson et al. *Fast synchronization for chip multiprocessors.* 2005. UCSD, UPC Barcelona, Paloa Alto, California.

[ea06] L. Ceze et al. *Bulk disambiguation of speculative threads in multiprocessors.* Proc. 33rd Annu. Int. Symp. on Computer Architecture, 2006. pp. 227-238.

[GP05] a division of the Free Software Foundation (FSF) GNU Press. *Using the GNU compiler collections.* Xilinx, Inc., 2005. For GCC version 4.2.1.

[Her88] M. P. Herlihy. *Impossibility and Universality Results for Wait-Free Synchronization.* 1988. Proceedings of the 7th Symposium on Principles of Distributed Computing, Toronto, Canada.

[HM93] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures.* Proc. 20th Annu. Int. Symp. on Computer Architecture, 1993. pp. 289-300.

[IBM05] IBM. *PPC405Fx Embedded Processor Core User's Manual.* IBM Corp., IBM Microelectronics Division, 2005.

[Kop97] Hermann Kopetz. *Real-Time Systems, Design principles for distributed embedded applications.* Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts, U. S. A., 1997. ISBN 0-7923-9894-7.

[LR07] Larus and Rajwar. *Transactional Memory.* Synthesis Lectures on Computer Science, Morgan&Claypool Publishers, 2007. ISBN 1-598291-24-6.

[MS95] Michael and Scott. *Implementation of atomic primitives on distributed shared memory multiprocessors.* 1995. University of Rochester, NY.

[NP00] Nikolopoulos and Papatheodorou. *Fast synchronization on scalable cache-coherent multiprocessors using hybrid primitives.* 2000. University of Patras, Greece.

[RG01] R. Rajwar and J. R. Goodman. *Speculative lock elision: enabling highly concurrent multithreading execution.* Proc. 34th Int. Symp. on Microarchitecture, 2001. pp. 294-305.

[RM06] Rajaraman and Murthy. *Parallel Computers, Architecture and Programming.* Prentice-Hall of India, New Delhi, 2006. ISBN 81-203-1621-5.

[Ska99] Kevin Skahill. *VHDL for programmable logic.* Addison-Wesley Longman, Inc., 1999. Variables: s. section 4.6.2.

[SS93] P. Sweazey and A. J. Smith. *Multiple reservations and the Oklahoma update.* IEEE Concurrency, 1993. Vol. 1(4), pp. 58-71.

[Sta01] William Stallings. *Operating Systems, Internals and Design Principles.* Prentice Hall International (UK) Limited, London, United Kingdom, 4th edition, 2001. ISBN 0-13-032986-6.

[Sto06] Christian Stoif. *A survey of the research on analysis of the worst-case execution-time (WCET).* Institute for technical computer science, silicon-on-chip group, Vienna University of Technology, 2006. http://www.soc.tuwien.ac.at/files/ss2006/seminar_WCETA.pdf.

[SZ02] Sterling and Zima. *Gilgamesh: a multithreaded processor-in-memory architecture for petaflops computing.* 2002. NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena.

[Tea02] The BlueGene/L Team. *An overview of the BlueGene/L supercomputer.* 2002. IBM, Lawrence Livermore National Laboratory.

[TvS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems, Principles and Paradigms.* Prentice Hall, Inc., Pearson Education, Upper Saddle River, New Jersey, international edition, 2002. ISBN 0-13-121786-0.

[Xil07a] Inc. Xilinx. *Dual Processor Reference Design Suite.* Xilinx, Inc., for edk and ise 9.2i edition, 2007. Application note XAPP996.

[Xil07b] Inc. Xilinx. *EDK Concepts, Tools, and Techniques.* Xilinx, Inc., 9.2i edition, 2007. PN0402644.

[Xil07c] Inc. Xilinx. *Embedded Solutions at Xilinx*. Xilinx, Inc., 2007. Slide-Show.

[Xil07d] Inc. Xilinx. *Embedded Systems Tools Reference Manual*. Xilinx, Inc., 2007. Xilinx EDK 9.1i reference.

[Xil07e] Inc. Xilinx. *ML410 Embedded Development Platform User Guide*. Xilinx, Inc., 2007. v1.6.1 UG085.

[Xil07f] Inc. Xilinx. *PowerPC 405 Processor Block Reference Guide*. Xilinx, Inc., 2007. version 2.0, UG018.

[Xil07g] Inc. Xilinx. *PowerPC Processor Reference Guide*. Xilinx, Inc., 2007. version 1.2, UG011.

[Xil07h] Inc. Xilinx. *Virtex-4 Family Overview*. Xilinx, Inc., 2007. v3.0 DS112.

Christian Stoif
Tibitsch 10
A-9210 Pörtschach am Wörther See

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe,
dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und
dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –,
die anderen Werken oder dem Internet
im Wortlaut oder dem Sinn nach entnommen sind,
auf jeden Fall unter Angabe der Quelle als Entlehnung
kenntlich gemacht habe."

Wien, 14. Nov. 2008

____
(Unterschrift Verfasser)