FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

SPEAR2C

Implementation of a Cache Controller for SPEAR2

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Michael Birner BSc.
Matrikelnummer 0526313

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr.tech. Andreas Steininger
Mitwirkung: Univ.-Ass. Dipl.-Ing. Jakob Lechner

Wien, 28.11.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Michael Birner BSc.
Griesgasse 28/3, 2340 Mödling

Danksagung

“Habe ich schon ’Danke’ gesagt?” - “Nein.” - “Kommt noch!” - Antonio Banderas alias El Mariachi in “Desperado”(1995).

Ich möchte mich hier an erster Stelle bei Andreas Steininger und Jakob Lechner für die unermesslich hilfreiche Unterstützung und für das gewonnene Gefühl unter bestmöglicher Betreuung gestanden zu haben bedanken. Zusätzlich danke ich dem gesamten ECS Institut für die wunderbaren letzten zwei Jahre in denen ich so unbeschreiblich viel gelernt habe, menschlich sowie beruflich. Die hier gewonnenen Erfahrungen haben mir für meinen Einstieg in das Berufsleben nach dem Studium enorm geholfen.
Ganz besonders möchte ich aber meinen Eltern danken, die mir meinen persönlichen Werdegang durch ihre Unterstützung während meiner gesamten Ausbildungszeit erst ermöglicht haben.

Abstract

Within the past 20 years the speed of processors has grown much faster than that of memories. To bridge the resulting performance gap, caching has been introduced. This thesis describes the completely redesigned memory layer for the *SPEAR2* core which serves as basis for the integration of various cache controllers.
SPEAR2 (**S**calable **P**rocessor for **E**mbedded **A**pplications in **R**eal-time Enviroments) is a RISC microprocessor architecture that was developed at the *Department of Computer Engineering* at the *Vienna University of Technology*. Thanks to its real time capability and its modular structure, this processor is especially suited to be used in embedded systems. In order to expand the application area of the processor and to use the full address space provided by the architecture, the *SPEAR2* core must be equipped with additional storage capacity, i.e., data and instruction memory need to be displaced into an external RAM module. The benefit of this approach comes at the cost of a performance degradation as the *SPEAR2* pipeline suffers from additional wait states caused by the increased latency of external memory accesses. The implementation of an internal cache memory therefore seems to be obligatory.
After providing a detailed theoretical background on the fundamental aspects of caching and some recent advancements in this field, this master thesis proceeds with simulations in order to find an appropriate caching strategy under the given constraints. The reader will be introduced to *SPEAR2SIM*, a novel simulation environment for the *SPEAR2* instruction set architecture that allows an efficient performance assessment of different caching strategies from which three candidates are selected. Their simulation results will be compared against the results of an experimental implementation in VHDL in order to be able to select the best adapted caching policy for a given application. Furthermore, practical problems and challenges of designing an efficient cache controller for the given architecture will be pointed out. As a result of this thesis, a novel prototype of the *SPEAR2* processor with a new memory layer that can be equipped with three different cache controllers (direct mapped, fully associative and two-way set associative) is available.

Kurzfassung

Innerhalb der letzten 20 Jahre stieg die Performance von Prozessoren um ein vielfaches schneller als jene von Speichermodulen. Um diesen enormen Leistungsunterschied zu überbrücken, begann man Prozessoren mit Caches auszustatten. Diese Diplomarbeit beschreibt die komplette Neugestaltung des Memory Layers der *SPEAR2* Architektur, welche als Basis für eine anschließende Implementierung verschiedener Cache Controller dient.
SPEAR2 (**S**calable **P**rocessor for **E**mbedded **A**pplications in **R**eal-time Enviroments) ist ein RISC-Mikroprozessor, der am *Institut für Technische Informatik* an der *Technischen Universität von Wien* entwickelt wurde. Dank seiner Echtzeitfähigkeit und seiner modularen Struktur ist *SPEAR2* besonders gut für den Einsatz in Embedded Systems geeignet. Um das Anwendungsgebiet jedoch zu erweitern, ist es notwendig die derzeit vorhandene Architektur mit mehr Speicherplatz auszustatten. Dies kann nur geschehen indem Daten- und Instruktions-Speicher aus dem FPGA in ein externes Speichermodul ausgelagert werden. Den Vorteil einer höheren Speicherkapazität bezahlt man jedoch mit einer enormen Leistungsverschlechterung, bedingt durch die wesentlich höheren Latenzzeiten beim Zugriff auf externe Speicher. Die Implementierung eines internen Cache Speichers ist demnach unausweichlich.
Nach einer Aufarbeitung der wichtigsten grundlegenden Aspekte bezüglich Caching sowie einiger fortgeschrittener Techniken, wird der Fokus dieser Diplomarbeit zuerst auf Simulationen von verschiedenen Caching Strategien liegen. In diesem Zu diesem Zwecke wird *SPEAR2SIM* vorgestellt, ein neuer *SPEAR2* ISA Simulator, der es auf eine effiziente Art und Weise ermöglicht für ein gegebenes Programm die am besten passende Caching Strategie zu finden. Diese Simulationsergebnisse werden anschließend mit den Ergebnissen einer einer experimentellen Umsetzung in Hardware verglichen, die darauf abzieht für vielversprechende Cache-Controller die entsprechenden Hardwareanforderungen zu bestimmen. Als Resultat dieser Diplomarbeit existiert ein neuer Prototyp des *SPEAR2* Prozessors, welcher mit drei verschiedenen Cache-Controllern (direct mapped, fully associative und two-way set associative) ausgestattet werden kann.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

”Schön, aber wozu ist das Ding gut?” - Ein Ingenieur der Forschungsabteilung Advanced Computing Systems Division (IBM) kommentierte 1968 den ersten Mikrochip.

Today, embedded systems are used in a broad variety of application areas. As the performance of their integrated processors is rapidly improving by reason of more and more aggressive clock rates combined with new sophisticated multicore designs, the area of operation of these systems is not longer only limited to simple controlling tasks. Mobile phones, e.g., are a very typical application field where advanced embedded microprocessors can be commonly found. While the first mobile phones implemented only simple stand-alone software solutions running on very simple processors,todays products like the *iPhone4S* from *Apple Inc.*®[1] include high performance system-on-chip architectures equipped with powerful multicore processors like the *ARM-A9*[2]. This steady development of the hardware allows us to execute advanced operating systems on our mobile phones that we only knew from being installed on our personal computers so far. Two recent well known examples are the *iOS* from *Apple Inc.*® used in the *iPhone* or *Android*, an open source operating system based on the Linux Kernel v2.6 which has been perfectly adapted to be used in embedded systems.

[1]http://www.apple.com/at/

[2]http://www.arm.com/products/processors/cortex-a/cortex-a9.php

However, the execution of such a complex software like an operating system on an embedded microprocessor requires a lot of random accessible memory to be available on the targeted hardware platform. This requirement directs this introduction to the key contribution of this thesis: As the performance difference between a processor and its main memory is huge for a reason that will be explained in short, every efficient modern computer architecture additionally needs to be equipped with fast cache memories in order to hide this discrepancy. The benefits that can be gained by executing a configurable operating system on a configurable hardware layer taking advantage of caching mechanisms motivated us to implement a completely redesigned memory layer together with an integrated cache controller for the *SPEAR2* processor.
The remaining part of *Chapter 1* will introduce the reader to a detailed problem description and to the history of the *SPEAR* processor family. The theoretical background concerning caching which is required for the description of the technical realization of this new memory layer will be completely covered by *Chapter 2*. *Chapter 3* will concern itself with the implementation of a novel simulation environment for the *SPEAR2* core and simulation results of several cache controllers whereas *Chapter 4* will provide the details of the implementation in hardware with a subsequent performance comparison with the simulated version of the processor.

1.1 Contribution and Problem Description

"The use is discussed of a fast core memory of, say, 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory" [26]. Back in the year 1965, M. V. Wilkes was probably one of the first computer engineers to write a scientific article about taking advantage of a small but fast buffer being placed between a processor and its main memory in order to gain a significant performance increase by the following modification of the data transfer paradigm for these two modules: Instead of requesting single data words, transfers are now performed from the main memory to this particular buffer in form of blocks. The reason why this approach clearly has a positive effect on computer systems equipped with a fast processor but a slow main memory will be the main topic of this section.
These very first proposed buffers were introduced as *slave buffers with dynamic storage allocation* which somehow sounded a little bit heavy handed. In order to get some shorter alias, computer architects decided to rename them into *caches*. This appellation originates from the French word "cacher", which means "to hide" [17]. Indeed, the first cache controllers were intended to operate hidden from the user and the processor. Even though the introductory definition from Wilkes dates back really far and the stated memory capacities seem to be ridiculously low today, it still can be applied when explaining the fundamental idea of caching.

However, this approach has not introduced a brand new concept since the idea of a block-wise data transfer between two memories of different speeds already existed before the first cache controllers appeared. In former days, an architecture consisting of a processor directly connected to its main memory that fetched data words from a slower magnetic tape in form of blocks was efficient enough as the performance of a CPU was equal or even smaller than the performance of its main memory module. Figure 1.1 illustrates a very simplified block diagram of the communication situation in that early period of computer architecture. As soon as the next block or page from a program had been received from the magnetic tape, the processor was able to operate on it without wasting any further wait cycles.

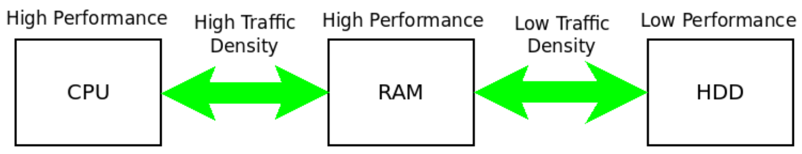


Figure 1.1: Communication situation in former days.

As it is visible in Figure 1.2, the situation began to change dramatically in the 80's resulting from the heavy performance increase of the processors while memories began to lag behind. This exponentially growing gap is the result from the following disparity [6]: While the main target for the processor industry is certainly the increase in computational performance achieved by higher frequencies and nowadays advanced multi core systems, memory manufacturers followed a different goal: The increase in capacity. According to [6], the improvement rate of the performance of processors, that is 60%, far outranks the improvement rate of access times of memories, which is calculated with less than 10% per year. Figure 1.2 shows this tremendous divergence.

Arrived in the year 2011, the performance gap has become huge. Today, a processor without at least one level of caching would suffer from an enormous performance degradation resulting from the waiting for a requested data word during every memory access. The actual communication situation is visible in Figure 1.3. Considering, e.g., a single core system (**non out-of-order execution**) clocked at 3,2 GHz connected to a DDR3 RAM module having an I/O clock of 800 MHz ($\approx 1,250$ ns) and a CL[3] of 10 ns, reading one single data word from the RAM block would take a time of approximately $11,250$ ns (10 ns $+\ 1,250$ ns). Consequently, the processor ($3,2$ GHz $\approx 0,31$ ns) would have to wait for about $\frac{11{,}250\text{ ns}}{0{,}31\text{ ns}} \approx 37$ cycles before the data transfer has completed. Accumulating this number over all memory accesses when executing a complete

[3]CL or *Column Address Strobe Latency* is the time interval between the request of the data and the instant of time at which the data becomes available.

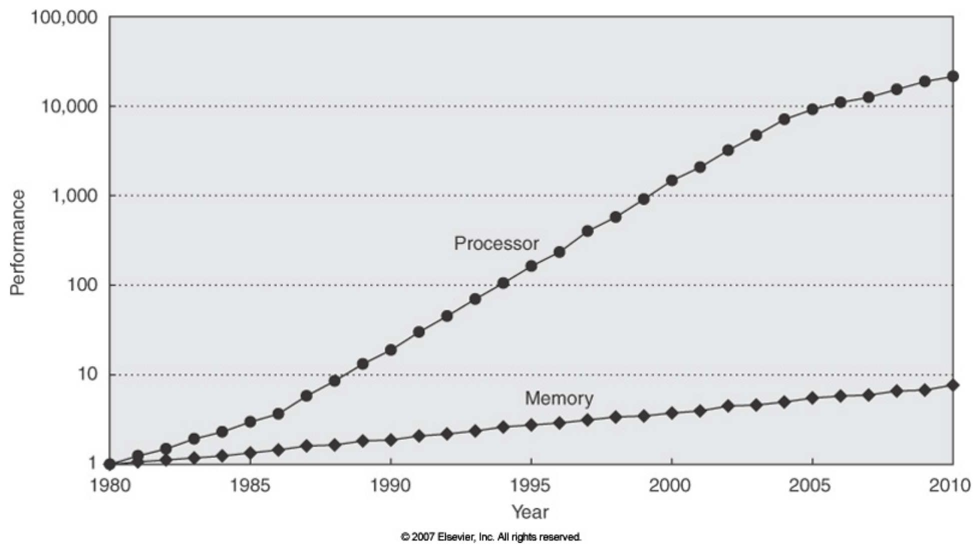


Figure 1.2: The processor memory performance gap [12].

program, it becomes immediately visible that the CPU would waste a lot of computing cycles resulting from this penalty just for doing nothing but waiting. As stated by [17], the fraction of all memory access instructions in a more advanced program can be up to 40%!

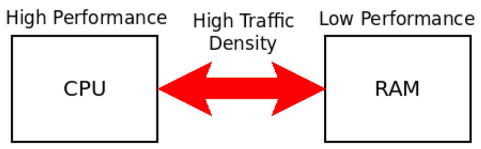


Figure 1.3: Communication situation in modern days without caching.

Programmers do not want to access the hard disk or the main memory. Their access times are far too slow compared to that of an internal register of a processor. As fast memory is expensive but desirable in an unlimited amount, a memory hierarchy clearly represents an efficient solution to this problem [12]. Figure 1.4 should be interpreted in the following way: The several levels of the illustrated ordering represent a division of all memories into different classes focusing on their speed and capacity. Memories with a high storage capacity but slow access time like a magnetic tape are placed at the bottom whereas fast but small memories like an internal processor register are placed at the top. The same holds for costs per byte. The higher a memory is positioned in the hierarchy, the more expensive per byte are its implementation costs.
Inspecting the common hierarchy in embedded, desktop or server computer systems, we can find

the cache being inserted at the level between the registers of a processor and the main memory. Although the cache memory is represented as one block, this level is usually split into three sublevels, L1, L2 and L3.

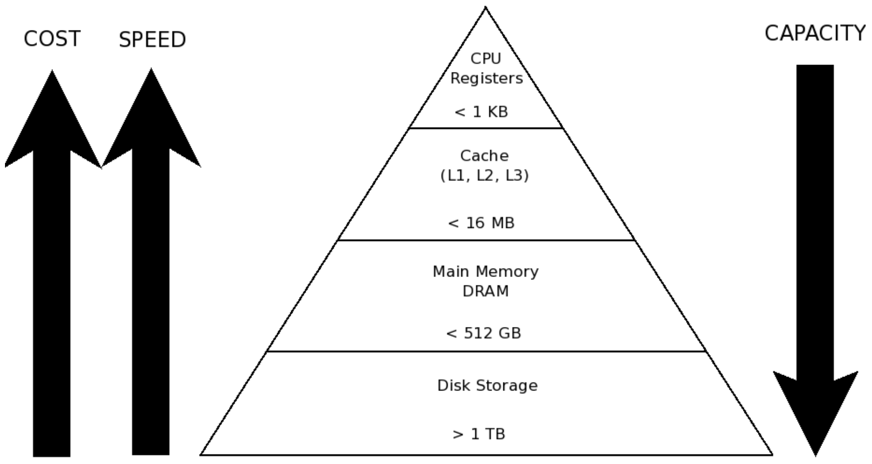


Figure 1.4: The memory hierarchy.

To finally present the benefits of this approach, two helpful properties of programs being executed by a processor need to be explained: These are (a) *the property of temporal locality* and (b) *the property of spatial locality* [20]. (a) states that data which will be referenced in the near future is likely to be already in use. On the other hand (b) means that if a data word is addressed by the processor, data words referenced by addresses near the actual referenced one tend to be accessed soon. A simple but descriptive example for both localities is an iterative access to an array in a loop where the temporal locality expresses itself through the iterative access and the property of spatial locality is provided by always accessing the same space of data, i.e., the repetitive access to the neighboring elements of the array. Beside profiting from the principle of locality, executing iterative access operations to the same memory block might enable the possibility to perform some kind of burst accesses which additionally decreases the latency.
Having these facts in mind it finally appears efficient to move data words from lower hierarchies to higher hierarchies in form of blocks in order to prevent the faster memories from accessing the slower memories too frequently, asking only for single data words. The key advantage of this approach can be demonstrated with the help of the following descriptive example.
In [8], Patterson and Hennessy compare the functionality of a memory hierarchy with the following abstract situation: A student is writing on some scientific report about any possible topic. He is sitting at his working place in a library, some books are already lying on his table, the rest

of the literature is ordered on a bookshelf which is positioned 50 foot beside him. After some time reading through the literature the student realizes that he needs more information about a special topic which is not included in the books that are already stacked on his desk. Hence he must walk to the bookshelf to find some work about the desired content and take it with him to his working place. Indeed it would be the much more efficient approach to grab more books from the shelf in order to protect the student from the situation in which he takes just one single book, goes back to his desk, sits down and recognizes that he needs more additional literature which results in standing up again, going to the bookshelf and taking only one book and so on. Grabbing a bundle of books will certainly increase the probability to keep sitting on his working place for a longer duration, spending more time on reading through the literature instead of always traveling between the bookshelf and the desk.
Finally a little note in the margin to clarify some misunderstandings might be given: Today, generally every buffer which is used to store commonly reused items is called cache [12]. Indeed, such a buffer does not necessarily need to reside physically inside a processor core. E.g., web caches from a web browser store recently used web pages to allow a faster access to them, or at least to a not up-to-date version of the web page.
This introduction will now proceed to the description of the targeted hardware platform in use, i.e., the *SPEAR2* architecture.

1.2 Target Description - The SPEAR Family

1.2.1 General Description

SPEAR (**S**caleable **P**rocessor for **E**mbedded **A**pplications in **R**eal-time Enviroments) was developed at the *Department of Computer Engineering* (ECS) at the *Vienna University of Technology* with the intention to design a softcore processor to be used in embedded systems that is able to fulfill real-time requirements. Therefore, two important features needed to be provided: (a) *Adaptability* and (b) *Real-Time Capability* [10]. Whereas requirement (a) is met by the usage of extension modules to extend the functionality of the processor, (b) is satisfied with properties like constant execution time, deterministic interrupt execution and hardware internal resolving of data and control hazards.

1.2.2 SPEAR

The fundamental features of the first *SPEAR* version are listed below:

- 16-bit RISC (load/store) architecture

- 3 stage pipeline (instruction fetch, instruction decode, execute/write back)
- 80 instructions (32 conditional instructions) with a constant execution time (CPI = 1)[4]
- 32 registers (26 general purpose registers & 6 special function registers)
- Separated data and instruction memory (Harvard Architecture) from which each can store up to 4 KB
- 1 KB of data memory reserved for the addressing of extension modules
- Data and control hazards resolved in hardware
- Worst case computable (no pipeline stalls)

The register bank of *SPEAR* includes six special registers: three frame pointers[5], two registers to store the return addresses when performing subroutine calls and one register to store the return address in the case of an exception. A simple memory architecture allows word-wise[6] accesses which doubles the amount of addressable memory in comparison to a byte addressing architecture. All instructions have a constant width of 16 bits and are executed in a constant time. Condition flags are used to determine if an operation is going to be performed or not. If an operation is skipped, a *NOP* instruction is inserted to keep the execution time of the program constant which helps to ease the calculations of the worst case execution time.
Extension modules are memory mapped and therefore accessed via normal load and store operations. Communication is performed by writing to and reading from special addresses (*Extension Module DATA fields*). Figure 1.6 illustrates the generic interface to the processor. An extension module may act as a functional extension (e.g., hardware multiplier) or as an I/O extension (e.g., RS232). The *Processor Control Module*, a special extension module which contains the status register and the interrupt handling logic must be included by default since it is essential for the processor to operate correctly. The generic extension interface will especially become important in the subsection that briefly explains the integration of the *AMBA* bus into the *SPEAR2* architecture. More detailed information about this very first processor of the *SPEAR* family can be found in [9] or [10]. Figure 1.5 shows the block diagram of the core.
Unfortunately, *SPEAR* suffered from several disadvantages like improper types of memories (the usage of asynchronous memories is not supported by new FPGAs), inefficient 16-bit memory accesses (wasted memory when storing only 8-bit values) and insufficient performance due to

[4]*CPI = Clock cycles per instruction*
[5]Frames are similar to stacks except that all data words can be accessed randomly without cleaning the stack.
[6]Since *SPEAR* is a 16-bit architecture, the width of a word is equal to 16 bits.

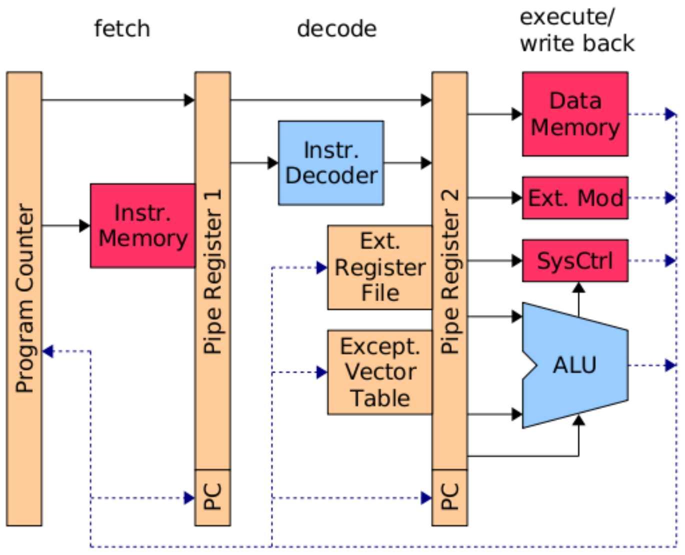


Figure 1.5: Block diagram of *SPEAR* [10].

16-bit operations. Consequently, a new processor based on the *SPEAR* architecture was developed: *SPEAR2*.

1.2.3 SPEAR2

Even though the source code has been completely redesigned, all the just introduced features from *SPEAR* have been adopted for the implementation of *SPEAR2* [10]. The following listing provides a brief overview on the most important facts of the new processor:

- 16-bit RISC (load/store) architecture with a customizable data path (16- or 32-bit)
- 4 stage pipeline (instruction fetch, instruction decode, execute, write-back)
- 122 instructions (40 conditional instructions) with a constant execution time (CPI = 1)
- 16 registers (14 general purpose registers & 2 special function registers)
- 4 stack pointers
- Separated data and instruction memory (Harvard Architecture)

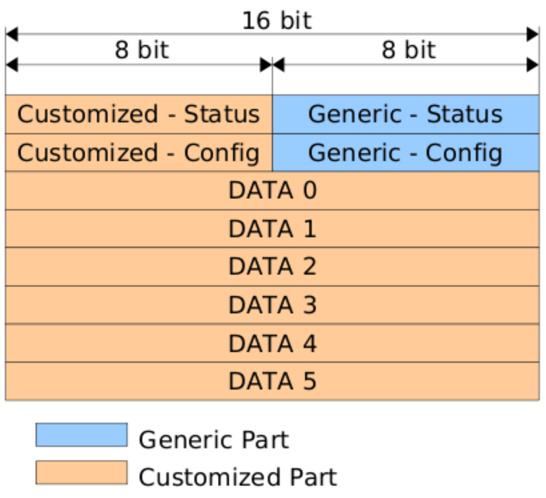


Figure 1.6: Generic interface to the extension modules [10].

- 32 KB of data memory reserved for the addressing of extension modules
- Data and control hazards resolved in hardware
- Worst case computable (no pipeline stalls)
- Sleep mode for power saving

The new key feature of *SPEAR2* is certainly its customizable data path which can be switched between a width of 16-bit or 32-bit. Clearly, the 32-bit data path version requires more resources but the increase in hardware requirements is not that big compared to the implementation of a complete 32-bit processor version. The main benefit is without a doubt the gained scalability. While using the 16-bit version might be the better approach for implementing simple embedded microcontroller tasks in primitive FPGAs with low storage capacity, the 32-bit version offers a noticeable performance improvement as the ALU can handle 32-bit values which is certainly suitable for more complex algorithms. Moreover, the broader datapath potentially allows to address a memory space of up to 4 GB. On this note it should be mentioned that the toolchain provides full compatibility between both versions. Hence the same C-code can either be compiled for the 16-bit or the 32-bit version of the processor.
In contrast to its predecessor, *SPEAR2* consists of four instead of three pipeline stages. The additional write back stage was introduced by reason of the registered output of a synchronous memory. Figure 1.7 shows the block diagram of *SPEAR2*.

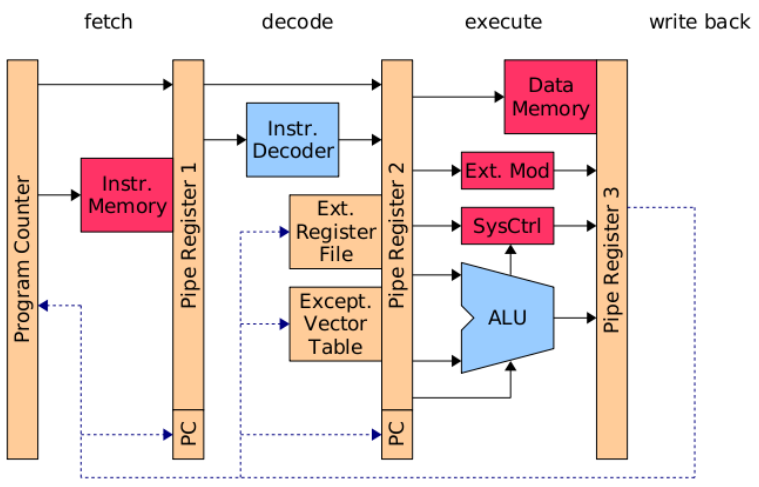


Figure 1.7: Block diagram of *SPEAR2* [10].

The following description will explain the duties of the several pipeline stages of *SPEAR2*. It might be skipped by the reader which already has experiences in computer architecture. Nevertheless, reading through this description might help to better understand the structure and the functionality of the new simulator toolchain that will be introduced in *Chapter 3*.

The *Fetch Stage* is responsible for fetching instruction words from the instruction memory and for managing the program counter which is incremented after every cycle or loaded with a new value if the ALU executes a jump operation.

The main function of the second pipeline stage, the *Decode Stage*, is the decoding of incoming instructions. This task includes the generation of the opcode for the ALU, the extraction of immediate values and the generation of addresses for the extended register file and the exception vector table respectively. When the ALU performs a jump operation or a subroutine call, the current instruction will be replaced by a *NOP* instruction. To be able to return to the previous program flow in situations where an exception arises or a subroutine is called, the program counter of the actual instruction is saved as return address.

All arithmetical and logical operations on incoming operands are performed by the *Execute Stage*. To prevent the pipeline from data hazards, the ALU is always fed with the newest values by a forwarding unit. Hence, no stalls are required to be inserted. Other possible operand sources are the extended register file, the program counter needed for the jump destination calculation, the output of the exception vector table when executing an interrupt or an immediate value

extracted from an instruction word. Beside arithmetical and logical operations, also memory and external module accesses are handled by this part of the pipeline.
The fourth and last stage finally merges the output from the ALU, the extension modules and the memory. For more detailed information on every pipeline stage and on their specific implementation details as well, the interested reader is referred to [10].
The *SPEAR2* ISA comprises 122 instructions which all have a size of 16 bits with a variable opcode size. 40 of them are conditionals. Just like its predecessor, *SPEAR2* executes every instruction within a well known constant execution time.
Data memory organization has changed from word access to byte access in order to achieve an easier portability to the GNU C compiler and to prevent the memory from fragmentation which occurs when 8-bit values are stored in a word accessed memory. Data access is limited by the width of the data path. Clearly, 32-bit accesses are only allowed with the 32-bit version of the *SPEAR2*[7] core. 16-bit accesses on the other hand are allowed with both versions. Byte ordering is little endian, memory accesses have to be aligned just like it is visible in Figure 1.8. To realize the favored byte access, four parallel directly accessible byte memories were used to build the data memory instead of implementing one single 32-bit wide memory block. As a result, only 30 from the 32 bits of the address lines are efficiently usable since the two least significant bits are needed to address single bytes (4 byte enable control lines). A more detailed description on the memory layer will be given in *Chapter 4*.

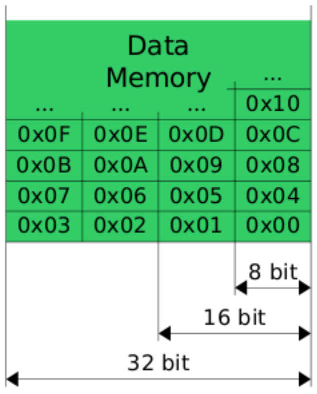


Figure 1.8: Memory organization of *SPEAR 2* [10].

[7]Remember that the 32-bit version of *SPEAR2* is not a complete 32-bit architecture. Only the data path is extended whereas the instruction set remains unchanged. The word size still equals 16 bits.

1.2.4 SPEAR2 AMBA Extension

The most important hardware extension for the *SPEAR2* architecture was certainly the implementation of an *AMBA* bus interface for the processor [16]. This allows an integration of *AMBA*-complient IP-cores, e.g., available in the *GRLIB*, which is an open-source library by *Gaisler Research*[8]. The *AMBA* modules in this library are based on the *AMBA 2.0* specification which was introduced by *ARM Ltd.* in the year 1999 [4].
The *AMBA 2.0* specification defines two data transfer protocols, one for high performance transfers and one that is especially suited to be used with low bandwidth modules. They are described in a complete technology independent manner and are used in the following three well-specified bus architectures:

Advanced High-Performance Bus (AHB) A two data path (read/write) bus for high performance communication. All data transfers are pipelined in one single stage, i.e., the tasks of sending the subsequent address and the actual data word are overlapped.

Advanced System Bus (ASB) Similar to the AHB bus except that the ASB uses only one data path for both, read and write operations.

Advanced Peripheral Bus (APB) A low performance bus for connecting simple modules like UARTs or timers.

The AHB (ASB) and the APB bus can be connected with a special bridging module. Figure 1.9 shows the block diagram of a typical *AMBA* system.
Additional components needed for successful bus operations are an arbiter and an address decoder. While the arbiter is needed for scheduling simultaneous bus requests from two ore more bus masters[9], the decoder is used to select the targeted slaves and to manage their responses.
It has to be taken into account that the *AMBA* bus is not implemented in an usual bus structure. Indeed it is has to be classified as a system of massive multiplexing operations. Figure 1.10 shows a simplified block diagram of the arbiter and the decoder handling incoming requests of three bus masters via this multiplexing mechanism.
The *SPEAR2* core is connected to the bus as master via its generic extension module interface. An appropriate extension module implementing the *AMBA* master state machine has been developed by [16]. Figure 1.11 shows the integration of the master into the *SPEAR2* architecture. A shared memory is used to decouple the processor from the bus master so that *SPEAR2* can continue with the execution of its program while the bus master is performing the data transfer.

[8]http://www.gaisler.com/

[9]*AMBA* is a multi master bus architecture.

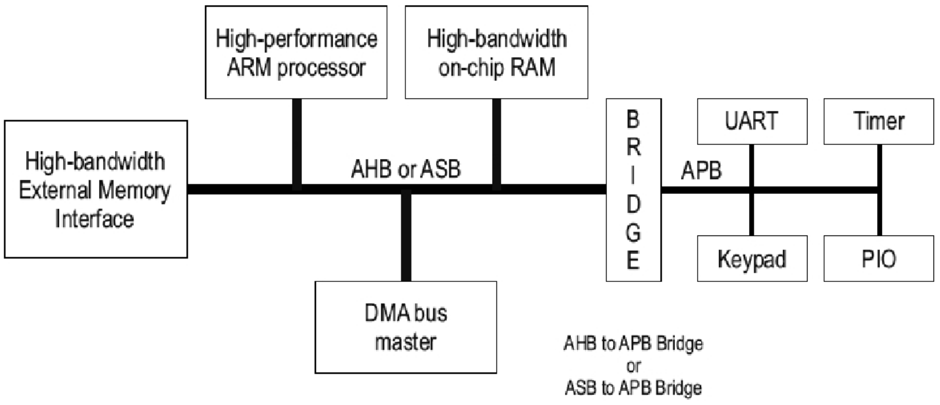

High-performance ARM processor
High-bandwidth on-chip RAM
High-bandwidth External Memory Interface
AHB or ASB
DMA bus master
BRIDGE
UART
Timer
APB
Keypad
PIO
AHB to APB Bridge or ASB to APB Bridge

Figure 1.9: A typical *AMBA* system [16].

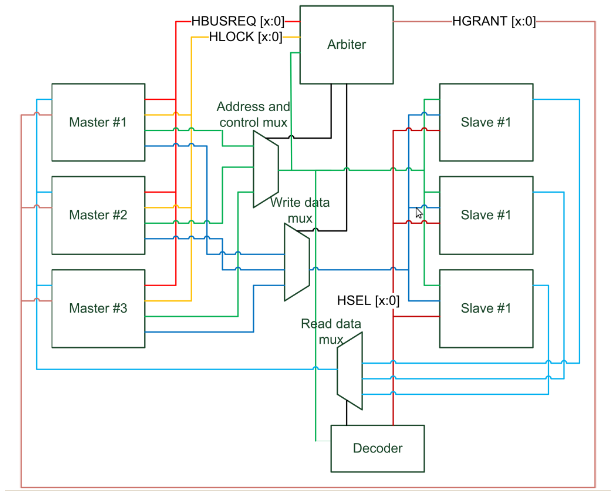


Figure 1.10: The *AMBA* multiplexer structure [16].

For more detailed information on the *AMBA* bus itself and its different kinds of operational modes, the interested reader is referred either to [16] or [4]. The integration of the new memory level for the *SPEAR2* processor with the help of the *AMBA* bus system will be the main contribution of *Chapter 4*.

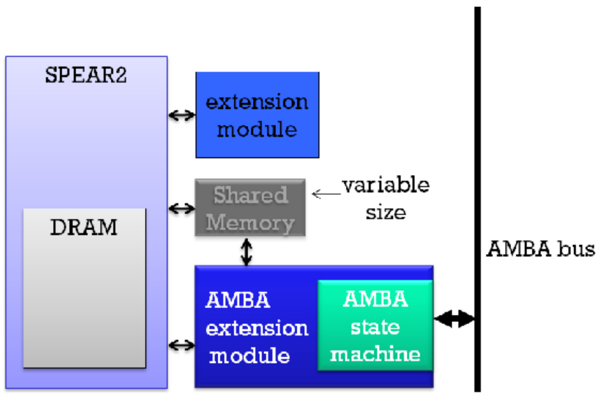


Figure 1.11: Integration of the *AMBA* master into the *SPEAR2* architecture [16].

CHAPTER 2

Caching

"Hits I Missed... And One I Didn't." - Johnny Cash.

To provide an adequate theoretical fundament for the following discussion on the search of an appropriate caching strategy for the *SPEAR2* processor, the next chapter summarizes the basic concepts of caching and describes some advanced techniques as well. Prior to examining different design aspects like block placement and block replacement algorithms in more detail, a number of elementary terms together with basic indicators for the estimation of cache performance are going to be introduced.

2.1 Basic Terms and Cache Performance

As introduced by [26], a cache controller is used to decouple a fast processor from its comparatively slow main memory module. In order to benefit from the memory hierarchy, data words are received and stored in form of *blocks*. A block consists of at least two or more data words. The number is typically chosen to be a power of two. Sometimes in literature also the term *line* is used. Common storage capacities reach from 4 KB up to 256 KB for processor internal L1 cache memories. As they are implemented on-chip they can be accessed at the speed of the processor which results in the convenience of no extra wait cycles that must pass for the operation

to complete.
When a requested data word can be found inside the cache, it is common to refer to such an event as a *cache hit*. No further information needs to be fetched from a lower memory hierarchy and therefore the system can fully profit from the performance advantage of the fast cache memory. The time needed to access the cache controller and to determine if a referenced data word is available in any block or not is identified as *hit time* T_{hit} [8]. Indeed, the duration of this operation is entirely independent from the result. On the contrary, a request yields a *cache miss* if the referenced data word can not be found in any of the blocks residing inside the cache. In this particular situation, the cache controller needs to fetch the desired information from the main memory. The time interval between the detection of the miss and the completion of the transfer of the block from the next lower level to the cache including a forwarding of the requested data word to the processor is denoted as *penalty time* $T_{penalty}$. An access of the main memory can be further characterized with the terms *latency* and *bandwidth*: Latency states how long it takes to fetch the first data word whereas bandwidth is used to determine how long it will take to transfer the rest of the block [12], i.e., the amount of information transported per time unit. Letting *BS* be the block size without the already received first word and *BW* the provided bandwidth of the next lower memory level from which to retrieve the block, the remaining transmission time $T_{transfer}$ can be defined as follows:

$$T_{transfer} = \frac{\text{BS}}{\text{BW}} \tag{2.1}$$

As a result, the penalty time can be stated as:

$$T_{penalty} = T_{latency} + T_{transfer} \tag{2.2}$$

The complete time lost due to a cache miss is calculated by summarizing the hit time and penalty time:

$$T_{miss} = T_{hit} + T_{penalty} \tag{2.3}$$

Generalizing this equation over the whole memory hierarchy yields an equation for the worst case cache miss, i.e., the request from the highest layer in the memory hierarchy falls through to the lowest layer:

$$T_{miss} = \sum_{i=0}^{N-1} (T^i_{hit} + T^i_{penalty}) \tag{2.4}$$

where N is the number of all memory layers in the hierarchy and T^i_{hit} and $T^i_{penalty}$ are the hit and the penalty time of the layer i respectively.

Unfortunately these timing parameters are somehow really hard to measure or to estimate. In addition, concerning the implementation in an FPGA they are strongly dependent on the place and route tool and the target technology. Thus some simpler and technology independent indicator for cache performance is certainly desirable. Such an indicator is, e.g., the *miss rate*, which is calculated by dividing the number of misses by the overall number of accesses to the cache [12] when executing a program.

$$\text{Miss Rate} = \frac{\text{Misses}}{\text{Accesses}} \tag{2.5}$$

Alternatively to the miss rate, it can be more informative to measure *misses per instruction* than measuring the misses per memory accesses as the miss rate does not include a clear relation to the total number of instructions.

$$\text{Misses/Instruction} = \frac{\text{Miss Rate} * \text{Memory Accesses}}{\text{Instruction Count}} \tag{2.6}$$

Although both introduced measurements provide important information on the cache access, they are only indirect measurements in the context of the overall system performance. Concerning an out-of-order execution or a speculative processor, a miss rate of, e.g., 40% does not need to be an evidence for a bad system performance since the CPU is capable of executing other instructions while waiting for the miss penalty to pass. But what exactly is the miss penalty of an out-of-order processor? One possible definition is stated by [12]: The penalty of such a processor starts in the cycle where the pipeline is not able to commit the maximum number of instructions. So if only four of five instructions are committed in a cycle because one instruction has to wait for the data to be received from the main memory, the out-of-order processor is said to be stalled. As a result the term stall must not be identified as a complete performance crash since the other four instructions still can be executed while waiting for the requested data to arrive. Thus, miss rate and misses per instructions should be interpreted as performance indicator for the cache controller itself and not for the complete system performance. Nevertheless, the negative impact on the overall performance resulting from inefficient memory access operations of an in-order-execution processor can be approximately calculated by putting the active cycles and the memory stall cycles into the following relation:

$$\text{Idle Ratio} = \frac{\text{Memory Stall Cycles}}{\text{CPU Cycles}} \tag{2.7}$$

The lower this value is, the more efficiently operates the cache controller. Sadly, also this equation might be very imprecise as memory stall cycles can also result from other I/O devices being accessed via memory load/store operations, e.g., the memory mapped extension modules of the *SPEAR2* architecture.

In this thesis, the miss rate and misses per instruction will be used as preferred performance indicators to estimate the efficiencies of all cache controllers under investigation. The overall system performance will be inspected by comparing the numbers of clock cycles needed for the executions of different benchmark programs. This indicator has been chosen since it can be extracted in a relatively efficient way from simulation runs. Moreover, as the *SPEAR2* processor belongs to the family of no in-order execution architectures, the just introduced drawbacks concerning the reliability of the miss rate can be ignored.
According to [17] it is important to acquire information about the reason of a cache miss to be able to improve future cache designs. Therefore, cache misses are categorized into the following three classes, also known as the three *Cs*: *Compulsory*, *Capacity* and *Conflict* misses. When starting the execution of a program, the very first access to a block will always lead to a miss. Since this is an almost unavoidable event, such a miss is called *compulsory miss*. Alternative appellations are *cold start miss* or *first reference miss*. Prefetching algorithms might help to improve the situation by bringing data into the cache based on a speculative decision before they are referenced by the processor. Such a prefetch might be performed statically at the start of the program or dynamically during execution. More detailed information on these algorithms can be found at the end of this chapter in the subsection *Prefetching*.
A *capacity miss* occurs when a cache can not store all blocks from the main memory which are actually needed for the execution of the current program. Hence, some blocks that might be used soon in the future must be discarded in order to yield free space for incoming data words.
If more than one block from the main memory can be placed at the same position in the cache, *conflict misses* occur when a data word is referenced by the processor which has been discarded by the controller because another block has been mapped to this particular position shortly before. The worst case concerning conflict misses occurs when two blocks from the main memory which are mapped to the same position in the cache are accessed in an alternating sequence. In this situation, which is referred to as *trashing* [17], the system completely loses the performance advantage gained by caching since the processor always needs to fetch the currently referenced data word from the memory, replacing the block inside the cache which will be needed in the next cycle. Especially the direct mapped block placement strategy is susceptible to trashing for a reason that will be explained in the subsequent section *Block Placement*.
Caused by the nature of every pipeline, instructions and data requests will occur simultaneously. As access timing parameters like the hit time are critical [20], it appears useful to divide the cache into an *instruction-* and a *data cache*, even though the main memory module is accessed via a Van Neumann architecture. Beside a decreased latency for every cache access, both, instruction and data cache return a cache hit in the best case and hence both words are immediately available in the next cycle. Anyway, attention must be paid in the case of both, instruction and data cache,

suffering from a miss and thus being forced to satisfy two main memory access requests at the same time on only one single access port. Furthermore, according to Smith [20], a crucial drawback of a split cache is to estimate an efficient partitioning of the available storage place as there is no general relationship between instructions and data words. One application might need 4 KBytes of instruction memory and only 200 Bytes of data memory while another one might occupy only 100 Bytes of instruction memory while claiming 2 MBytes of data memory. Another kind of subdivision of the cache memory can be applied to be able to profit from both, a fast hit time and a high storage capacity. This can be achieved by creating a multilevel structure of caches, thus adding an *L2* or even an *L3* cache to the architecture. The original cache controller is then referred to as *L1* cache. Anyway, the L2 and the L3 will not provide the same low latency like the L1 cache as their capacities are usually chosen to be a multiple of the L1 capacity which results in a significantly slower hit time. As a consequence, accesses to the higher levels of the cache controller can not be completed in one computational cycle. Nevertheless the penalty is far lower than accessing the main memory.

2.2 Block Placement

2.2.1 Basics

The way a cache is organized and its block placement policy are critical implementation decisions since they have a huge influence on the overall performance of the cache controller. Generally, a cache memory is not directly accessible and its storage capacity is rather small compared to those of memories from the lower levels of the memory hierarchy. Therefore, resulting from this many-to-one relation, some function is needed for an exact mapping of addresses referencing the main memory to their corresponding cache internal block addresses [20]. In other words, every block from the main memory must be assigned to one definite position inside the cache memory. Of course, this function completely depends on the placement strategy of the cache controller. Before explaining the three fundamental approaches for this address mapping, a generalized cache internal address format together with the most basic functional blocks of a simplified cache architecture are going to be introduced.
In its simplest form a cache design consists of a controller block implementing the placement algorithm, some memory for storing the data blocks which have been fetched from the main memory after they were requested by the processor, and a tag memory that stores well defined parts of the referenced memory addresses for all blocks that are residing inside the cache. These partial addresses are needed to identify the requested block inside the cache in a way that is going to be explained next. Figure 2.1 shows the block diagram of such a basic cache implementation.

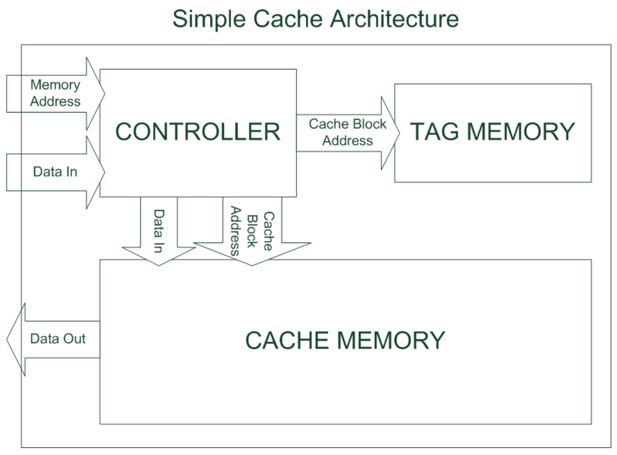


Figure 2.1: Block diagram of a simplified cache module.

Given some main memory address being referenced by the processor, the first part of the cache internal address, built by the least significant bits, is referred to as the *word address*, which is stored in the so-called *block field* (see Figure 2.2). The size of this field, which is totally independent from the placement strategy, determines the *block size*, i.e., the number of data words that fit into one block. E.g., for a given block size of 16, the corresponding number of bits is trivially calculated with $log_2(16) = 4$. In the case of a memory load operation sent by the processor the word address is used to extract the requested word from the targeted block. This action will be taken immediately in the case of a cache hit or otherwise in the case of a cache miss, after the complete block has been received from the main memory. The same holds for memory write operations where the word address again is used to address and update the corresponding data word in the targeted block.

Having separated the block field from the referenced address, the remaining part can be identified as *block address* [12] as it is used to address and identify single blocks inside the cache. Depending on the placement strategy in use, the block address can be further divided into two variable sub-fields: The *tag field* and the *index field*. While the index field determines the position of the targeted block inside the cache, the tag field is used for the comparison with the corresponding entry from the tag memory in order to check if the requested data word is residing inside the cache or not. The tag field is also known as *block frame address*.

In addition to the bits used for addressing, a valid bit needs to be added to indicate if the data block present at some position is valid or not [12]. The corresponding tag will only be checked against if this valid bit is set and ignored if it is cleared. In the latter case, the block is said to be *invalid*. Invalid blocks can appear when some other device that is sharing parts of its address

space with the processor is writing to a data word in the main memory which is also currently residing inside the cache memory or when the block is still empty at the beginning.
A typical address frame showing the just introduced sub fields is illustrated in Figure 2.2.

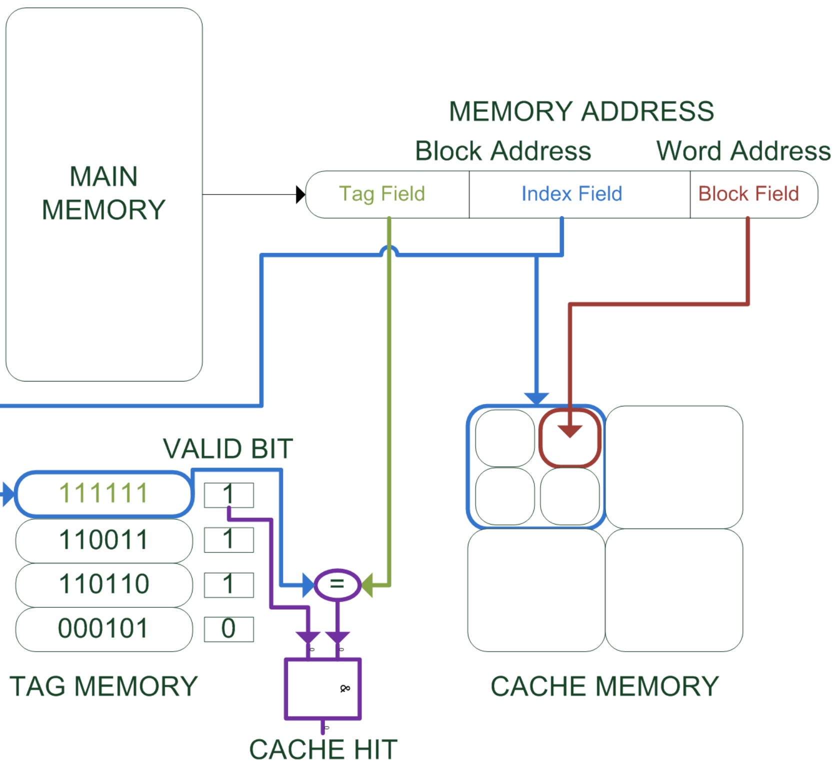


Figure 2.2: Cache internal addressing.

A simple cache controller might perform the following basic steps for every memory read access instruction:

1. Select cache block addressed by the *index field*.
2. Compare stored tag bits with bits from the *tag field* and check valid bit.
3. On a hit, select requested data word addressed by the *word address*. On a miss, forward referenced address to the main memory, fetch the complete block, store it in the cache and send the requested data word to the processor.
4. Go to 1.

A simplified write access might be executed in a similar way:

1. Select cache block addressed by the *index field*.
2. Compare stored tag bits with bits from the *tag field* and check valid bit.
3. On a hit, update selected data word addressed by the *word address*. On a miss, do nothing[1].
4. Go to 1.

As already stated, there exist three common basic placement strategies which will be explained in the following [12]:

Direct Mapped For every block from the main memory there exists only one feasible position inside the cache where it can be placed. A simple function to calculate this position is:

$$f_{map} := \text{(Block address) MOD (Number of blocks in cache)} \tag{2.8}$$

Figure 2.3 illustrates the placement of a block from the main memory address $0x0016$ at the cache internal position $0x0016$ MOD $0x0008 = 0x0006$.

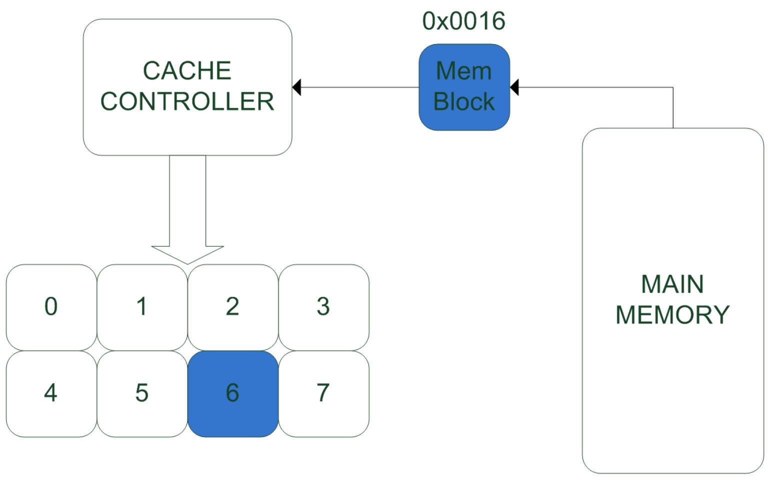


Figure 2.3: Direct mapped cache example.

The usage of the direct mapped placement policy is very popular for embedded processors for a reason that will be explained later on.

[1]The reason for this behavior will be justified in the subsection *Write Operations*.

Fully Associative There exists no restriction on the placement. Therefore a block from the main memory can be placed at any possible position. In theory, concerning lowering the miss rate due to conflict misses, this is definitely the best strategy as the cache controller can execute an efficient decision procedure to select a position which is currently not occupied by some other block or to select a block to be replaced based on the decision of a replacement algorithm which can be applied over the complete cache address space in order to unfold its full efficiency.

Regrettably, all these advantages go hand in hand with one tremendous drawback. In real terms, every block includes an address tag which has to be checked during every cache access. Indeed, this results in an enormous increase of the hardware requirements caused by many address compare operations which will definitely increase the hit time, thus slowing down the speed of the processor. As the critical path of most caches is defined to run through the comparators [17], increasing the number of compare operations will also increase the length of the critical path. In contrast to the direct mapped placement procedure, no index field is needed since every block can be placed at any possible position.

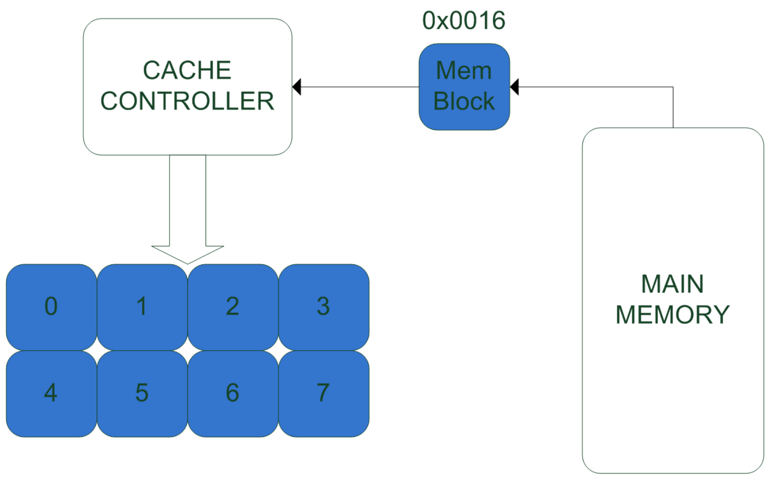


Figure 2.4: Fully associative cache example.

Set Associative The cache is divided into several *sets* of blocks. A block from the memory is mapped to one dedicated set in which it can be freely placed at any position. If every set contains n blocks, the cache is said to be *n-set associative*. A simple function proposed by [12] is:

$$f_{map} := \text{(Block address) MOD (Number of sets)} \tag{2.9}$$

The number of sets, which must be all equal in size, is calculated by $N = \frac{\text{Number of blocks}}{n}$. When accessing the set-associative cache, the bits of the index field are used to determine the targeted set. Figure 2.5 illustrates an example for a set count of $N = 4$ and an index field with the value "10". Consequently, set number two is selected.

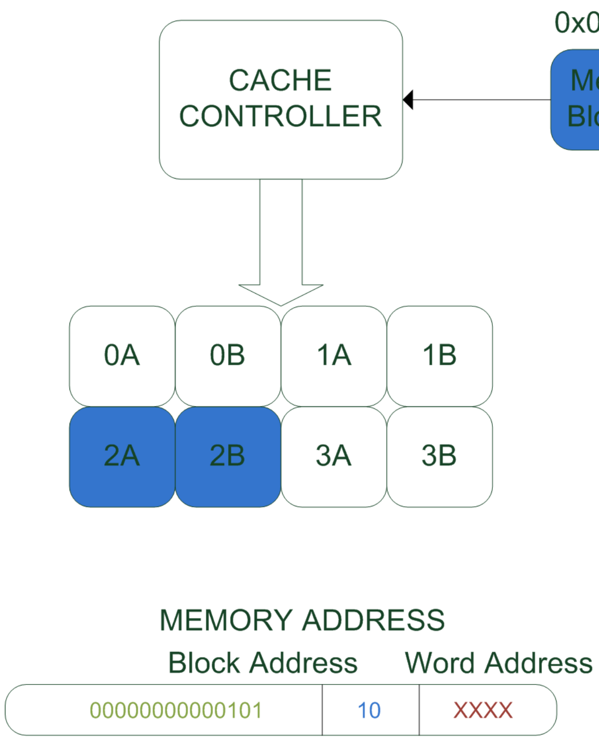


Figure 2.5: Set associative cache example.

Letting n be 1, the set associative organization changes to a direct mapped one. On the other hand, it is possible to derive a fully associative policy from a set associative placement strategy by letting n be equal to the overall block count of the cache. Examining the general address format for cache addressing, the increase of n until the fully associative strategy is reached can be illustrated by the following way: Letting the cache size remain constant, increasing the associativity results in a higher number of blocks per set, thus reducing the number of sets in the cache. Focusing on Figure 2.2 the index field becomes smaller and smaller, moving to the right until it disappears [12]. At the same time, by shrinking the index field, the tag field must consequently become larger, unfortunately increasing the hit time as there are more bits to be compared with.

As the mapping function represents a many-to-one relation, especially conflict misses seem to be

unavoidable since two or more different blocks from the main memory will be definitely mapped to the same position in the cache [20]. The following proposals concerning lowering conflict-, compulsory and capacity misses have been taken form [12].
Clearly, direct mapped cache controllers exhibit the highest miss rates as there is no room for alternative decisions concerning the final position of a block. Ignoring the just mentioned drawbacks of the fully associative placement policy, these conflict misses can be dramatically reduced by increasing the set size. Beside increasing the associativity, another approach might be to increase the overall size of the cache in order to allow more sets to reside inside the cache. In this case, the index field visualized in Figure 2.2 would increase, therefore moving to the left causing the tag field to shrink at the same time. This technique helps to lower both, the conflict and the capacity miss rate. Unfortunately, increasing the capacity also results in higher hardware costs, power consumption and higher hit times. Hence, larger caches are primarily used for the L2 or the L3 cache as the pressure on the L1 cache to execute very quickly is too big to allow a degradation of the access timing caused by a higher storage capacity.
The most simple way to decrease the miss rate, more precisely the compulsory and capacity miss rate is to increase the size of the blocks, i.e., to allow more data words to fit into one block. The decrease in the miss rate is a logical consequence from the property of the locality of space: As neighboring data words are often referenced in an iterative way, transferring bigger blocks of data from the main memory to the cache will boost the performance of the processor with a high probability. In addition, since fewer tags have to be stored in the tag memory, also the hit time will decrease [20]. On the other hand, if the cache is only small in size, a bigger block size might contrary increase capacity misses as fewer blocks can be stored in the cache. Exactly the same holds for conflict misses. Furthermore, in the case of a cache miss, more data words must be fetched from the main memory which leads to a rise in the penalty time if the processor is forced to stall until the transfer has completed. Thus, larger block size is only recommendable if the processor has some kind of fetch bypass mechanism or if the memory interface provides a high bandwidth and a low latency. All in all, the block size must be carefully chosen in order not to erroneously degrade the performance of the cache controller.
Focusing on the optimization of the hit time, the one and only realistic approach is to build simple caches. The aim should be to keep the hardware requirements as low as possible because smaller circuits can be clocked faster. From all the placement policies which have been introduced so far, the one which provides the lowest hit time is clearly the direct mapped strategy as only one tag entry has to be checked against. Furthermore, the retrieval of the data word can be performed in parallel with the tag check operation. This advantage and the fact that no replacement algorithm has to be executed make the direct mapped policy especially attractive to be used in embedded systems. Figure 2.6 presents access times of different cache controller

implementations depending on the overall size of the cache memory. Clearly visible is the dependency of the access time on the level of associativity and cache size. These values have been extracted from a CACTI[2] model that simulated a 90 nm feature size CMOS cache controller with 64-Byte blocks.

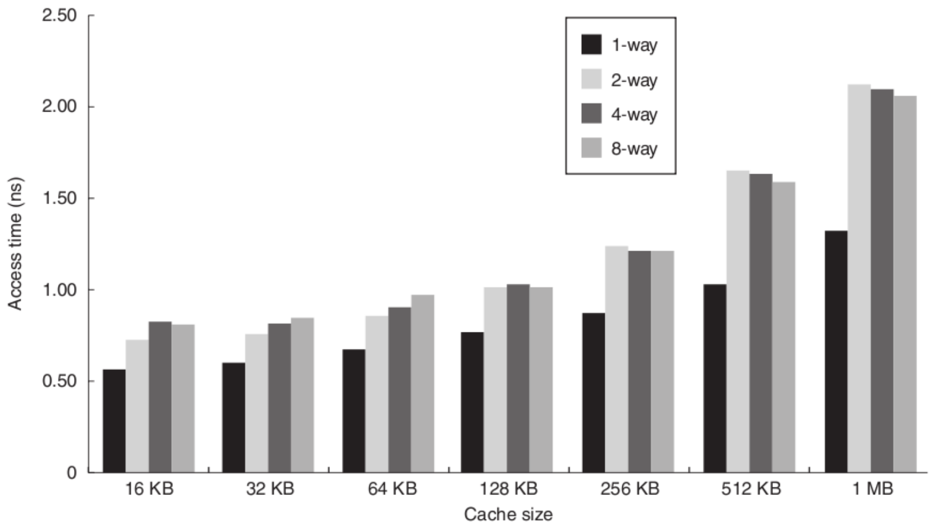


Figure 2.6: Hit times of different cache controllers [12].

While processors and caches provide more and more performance and rising storage capacities, also their power consumption increases. Especially on-chip caches, implemented by using arrays of densely packed SRAM cells consume a significant amount of the overall power of a chip [27]. Since cache sizes are increasing, this rising need for power has become a serious problem, in particular for embedded systems which are generally expected to be power saving. The following improvements of the basic placement policies show how to obtain cache implementations that consume less power and area or how to reduce the miss rate resulting from a more efficient placement of blocks inside the cache. While the first approach is implemented on the hardware layer, the second improvement presents one of many optimizations that can be applied on the software layer.

[2]CACTI is an integrated cache and memory access time, cycle time, area, leakage and power model - http://www.hpl.hp.com/research/cacti/.

2.2.2 Improvements - Paged Cache

In [27], Chang and Lai propose a paged cache implementation which achieves a noticeable decrease in the tag length and in the demand of chip area with a resulting decrease in power consumption and access time. The fundamental idea is to divide the cache into a number of areas which is equal to the count of pages being stored by the TLB[3]. These partitions are equal in size and upper bounded by the size of a page. The reduction in the length of the tag is achieved by allowing every partition to be mapped to only one particular page from the TLB. The reduction in power consumption and hit time on the other hand is a logical consequence from this restriction since for every cache access only one partition needs to be searched for the desired data instead of accessing the complete cache.
Figure 2.7 shows the different structural aspects of a conventional and the paged cache:

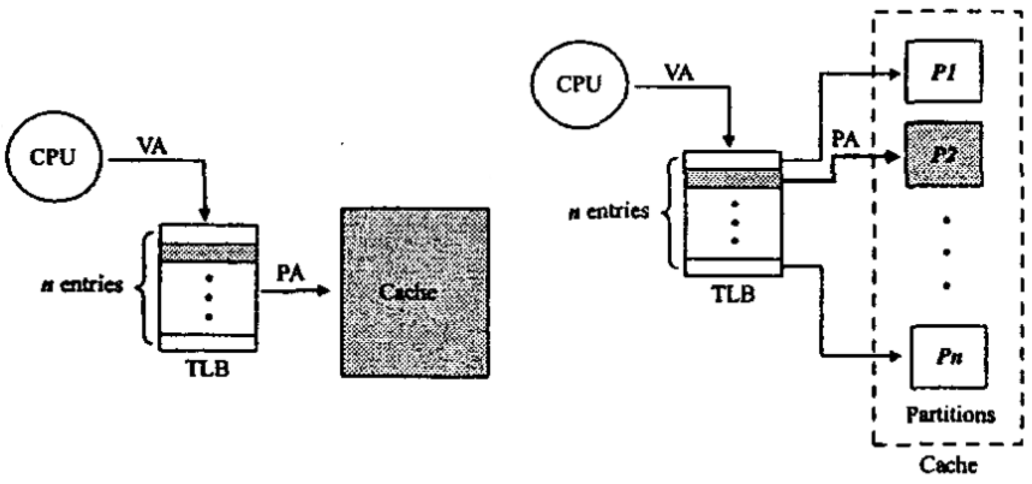


Figure 2.7: Conventional cache controller versus paged cache controller [27].

After the virtual address (VA) has been translated to a physical address (PA) in the TLB, a conventional cache passes the resulting address directly to the cache controller, thus performing an access on the complete cache logic in the worst case. In a paged cache architecture, only the partition related to the page entry will react to the fetch request. All other partitions do not need to be searched for. The new reduced tag length resulting from this partition can be calculated with:

[3]The TLB is a buffer which stores the recently used pairs of virtual addresses and their corresponding physical addresses.

$$\text{tag length} = log_2(\frac{\text{page size}}{\text{partition size}}) \tag{2.10}$$

For a 64 KB cache memory and a 32-entry TLB with a page size of 4 KBytes the corresponding partition size is 2 KBytes. Following the formula stated above, the resulting tag size is calculated to be only 1 bit. Figure 2.8 compares the tag length of a conventional cache implementation with the paged architecture. The conventional cache controller implements a one-way associative (direct mapped) 64 KBytes cache architecture with a tag length of 16 bit.

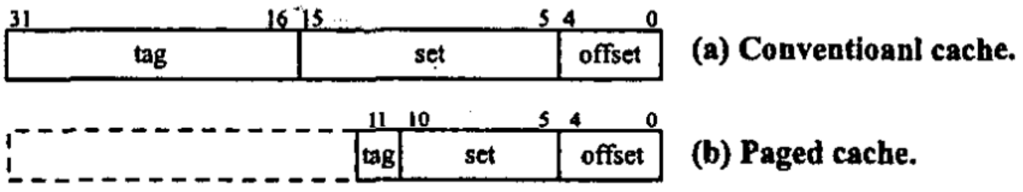
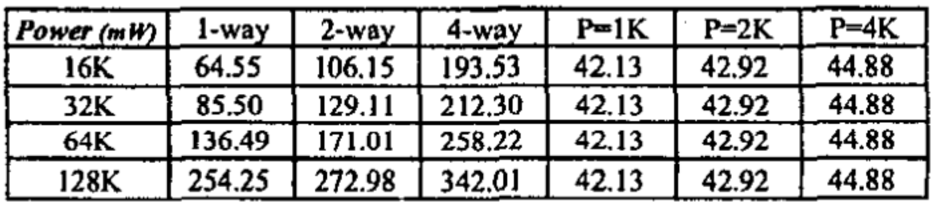


Figure 2.8: Tag length of a conventional (a) and paged cache (b) architecture [27].

As the TLB and the cache controller are set into a relationship concerning hits and misses, every hit by the cache controller implies a hit in the TLB and every miss in the TLB implies a miss in the cache controller. Based on this fact, [27] defines the following standard access sequence for every memory access instruction:

1. Send virtual address to TLB.
2. In the case of hit, the targeted page is used to determine the corresponding partition in the cache. In the case of a miss, the requested data from the page must be loaded from the memory, replacing the entry in the TLB and replacing the data in the cache partition. According to the access relationship, there is no further need to check the cache after a TLB miss.
3. In the case of a TLB hit, check the tag to determine if the cache access is a hit.
4. In the case of a cache hit, the requested data word is returned. In the case of cache miss the TLB address is used to fetch the particular part of the page from the main memory.

Concerning the evaluation of the decrease in power consumption, Chang and Lai performed *HSPICE* simulations based on an access model described in [19]. The results of this investigation show that the power consumption of a cache access is independent from the overall cache size but truly depends on the size of the partition. This behavior is contrary to conventional cache implementations where the need for power rises tightly correlated with the increase of the cache size and the associativity. A comparison of the power consumption of a conventional cache controller with that of a paged cache controller is visible in Figure 2.9.



Power (mW)	1-way	2-way	4-way	P=1K	P=2K	P=4K
16K	64.55	106.15	193.53	42.13	42.92	44.88
32K	85.50	129.11	212.30	42.13	42.92	44.88
64K	136.49	171.01	258.22	42.13	42.92	44.88
128K	254.25	272.98	342.01	42.13	42.92	44.88

Figure 2.9: Power Consumption per access of a conventional and paged cache [27].

The major drawback of this approach certainly is an increase in the conflict and capacity miss rate as the block size is noticeably bigger compared to that of a conventional cache implementation. Applying this approach on caches equipped with a small capacity can therefore be extremely inefficient as bigger blocks in a small cache result in the handicap to only store a few blocks inside the cache.

2.2.3 Improvements - Procedure Placement

Another optimization, although it is not directly applied on the hardware level, can be gained by implementing a procedure placement algorithm which achieves a lower miss rate for all accesses to the instruction cache by a clever reordering of the instructions in the instruction memory [18]. On a higher level of abstraction an application can be generally subdivided into to one main section calling several other procedures. Since these procedures are shared globally and might be called from any possible position in the main program, the alternating usage of two procedures which are mapped to the same position in the cache can easily lead to trashing [18]. The subsequent assembler code segment provides a simple example for such a worst case program being executed with an instruction cache with a block count of 2 and and a block size of 2.

```
// 2 blöcke zu 2 wörtern

.text
# Block 1
jmpi 4
nop
```

```
# Block 2
nop
nop

# Block 1
jmpi -4
nop
```

In this fundamental scenario, instruction at address 0 and instruction at address 5 will be executed alternately. For a direct mapped cache controller, both instructions will always be placed in block 0, thus leading to trashing. A fully associative cache controller will use both cache blocks instead, thus only suffering from two misses.
Before explaining the main concept of the algorithm, a few new terms need to be introduced: A *program line* is some portion of the instruction code which fits in into one block of the instruction cache. A procedure consists of at least one program line. Depending on the placement strategy, these program lines are mapped to distinct positions in the instruction cache. The *procedure placement problem* concerns itself with the finding of an adequate mapping of all procedures to the instruction memory in order to reduce the number of the cache misses.
The algorithm is divided into two phases. The effort of the first phase is to try to reduce the negative impact on the performance caused by the following problem: There is no guarantee that all instructions from a program line are going to be executed. In the worst case, only one instruction will be executed and this will be a branch operation jumping to another procedure that is stored far far away at the other end of the address space of the instruction memory. As a consequence, if not already present in the cache, this particular program line must be fetched and be brought into the instruction cache. Therefore, the first job to be done by the algorithm is to shift the start addresses of all procedures by an offset in order to gain some new ordering that minimizes the number of blocks being fetched by the cache controller. The best solution is found by trying all possible offsets and selecting the most efficient offset. Clearly, the offset must be smaller than the size of the block.
In the second phase, the procedure replacement is performed to achieve a cache wide distribution of all procedure calls in order to prevent two procedures from being mapped to the same cache block. The target is not to remove all these conflicts, which is without a doubt impossible due to the small memory space of an instruction cache, but to avoid those conflicts which appear frequently.
Before the algorithm can start, some input data needs to be extracted from a trace simulation of the original program, i.e., the miss distribution on all cache blocks and the number of misses for every procedure. Additional inputs are the memory space addressed by every subroutine and

important cache parameters, e.g., the block size. At first, the cache block with the highest number of conflicts is placed anywhere in the memory. All further blocks are positioned iteratively following the decision procedure of a heuristic cost algorithm which tries to estimate the lowest penalty cost of the actual procedure in the case of a conflict miss with the already positioned procedures caused by the usage of the same cache block position.
The positive effect of the algorithm due to distributing the memory references over the complete instruction cache and lowering the highest numbers of misses can be seen in Figure 2.10.

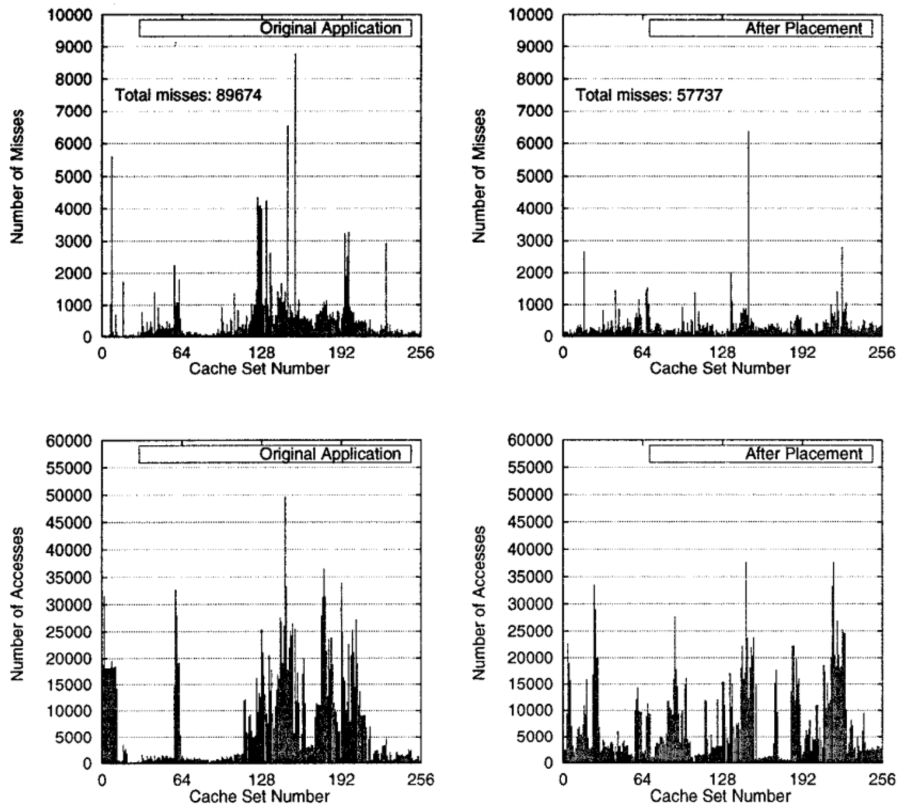


Figure 2.10: Positive effect of procedure placement on the miss rate [18].

The drawback on the other hand is easy to unveil: Every application needs to be pre-simulated in order to retrieve the essential information which is necessary for a efficient reordering of the instructions in the memory.

2.3 Block Replacement

2.3.1 Basics

This part of the thesis will focus on the question which block to replace when a cache miss occurs and there is no more free space left for storing new data words from the main memory. Using direct mapping as placement strategy, this question is very easy to answer: There is only one block to be chosen [12]. But as soon as there exists more than one possibility to place a block inside the cache and all these positions are already occupied, the cache controller has to discard at least one of the stored blocks based on a specific decision. Contrary to page replacement algorithms for the main memory which can be fully implemented in software, cache replacement algorithms must be completely realized in hardware to be able to execute very quickly without degrading the system performance too much [20]. The longer it takes for the procedure to select the block to be kicked out, the longer the miss penalty will become and the slower the cache access will get. Cache replacement algorithms can be classified into the following categories:

Usage-Based The usage of a block is taken into account.

Non-Usage-Based Some other decision than the usage is taken into account.

Fixed-Space The amount of allocated memory is fixed.

Variable-Space The algorithm varies the amount of memory allocated to a process.

The variable space replacement policy is clearly not suitable for a cache memory considering the fact that a cache is usually physically fixed in size and far too small to be divided into areas which are big enough to hold the complete working set of a program. Hence, no more further attention will be spent on this replacement category.
Before dealing with more complex replacement theories, the following listing will give an overview on four common basic strategies [12]:

Random A random block will be chosen and discarded from the cache. Apparently, this is a rather simple and probably inefficient kind of implementation since it takes no advantage of the principle of spatial and temporal locality at all. Data words which might be used soon by the processor can be discarded without restrictions of any kind. Considering, e.g, an iterative access to the elements of an array from which one of it is replaced by some arbitrary element, a cache miss will occur, thus resulting in an unnecessary decrease of performance as the cache controller can not take advantage of the locality of space. The same holds for the locality in time, e.g., instructions in a loop that are free to be randomly replaced by some other instructions. As the usage of a block is not taken into account when discarding a block from the cache, the Random algorithm has to be classified as a non-usage-based algorithm.

A pseudo random version of this replacement policy can be easily realized by using a modulo E counter for every set, where E reflects the maximum number of blocks being stored in this set [20]. The counter can be incremented with any event of interest, e.g., with every clock cycle, finally holding a value which is equal to the position of the block in the set which will be discarded next.

LRU As it can be guessed from the name, the *Least-Recently-Used* replacement algorithm is an usage-based algorithm. Taking advantage of the locality of time, different methods exist to discard primarily those blocks which have not been used recently in the past. Concerning implementation options, every block present inside the cache can be equipped with a counter to record the number of cycles without being accessed by the processor in the near past. During a miss, the block with the highest stored number is discarded. Truly, this is a very inefficient realization as there is a noticeable increase of hardware requirements when implementing these counters. To keep it more simple, a single status bit for indicating the recent usage of a block might be sufficient. Nevertheless, regardless of the used width of the counter, having some information about the usage of a block in the past, although this information might be very imprecise or expensive, does help to avoid discarding blocks from the cache, which are likely to be referenced by the processor soon.

Smith [20] proposes the following implementation strategies: For a set of two blocks, a hot/cold toggle bit is sufficient to decide which block to remove. A more general definition is the following: Having a set with a variable number of E elements, $E(E-1)/2$ status bits are required. This is exactly the number required to have a status bit for every pair of elements within the set. Thus the status bits define a strict partial order on the recentness of the element accesses. The replacement algorithm can be implemented in an efficient way by creating an upper left triangular matrix without the diagonal. Referencing a block i in this set will cause all values in the row i of the corresponding matrix to be replaced with a '1'. Afterwards, all values in column i are set to '0'. These two operations on the matrix are repeated with every reference to this set. The least recently used block is identified by the row in which all values are equal to '0' and its corresponding column in which all values are equal to '1'. Unfortunately, the number of required status bits increases with the square of the set size, resulting in the fact that this algorithm is only suitable for smaller set sizes. Choosing a LRU algorithm as replacement strategy is therefore only reasonable when the hardware costs are negligible and the gain in hit rate outweighs the additional penalty time due to the complex implementation.

Figure 2.11 illustrates some simple example for the LRU matrix algorithm showing the decision procedure for a set with a count of three blocks. After five successive accesses to this set, block #1 is identified as the least recently used block as its entries into the matrix exactly match with those of a least recently used block described by the algorithm above.

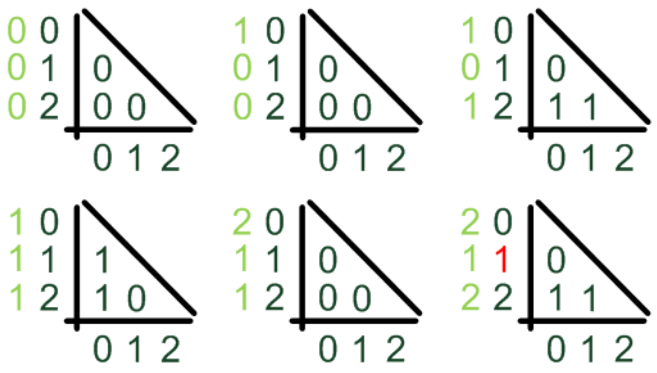


Figure 2.11: The matrix based LRU algorithm (the green column shows the access count).

LFU This replacement theorem is similar to the LRU policy except that the least frequently used block is discarded and not the least recently used one. Instead of counting the passed cycles without being referenced by the processor, every block counts the overall number of accesses over a defined period of time. The block with the lowest number of accesses is discarded.

FIFO To avoid the enormous hardware requirements caused by the LRU algorithm which becomes increasingly expensive with a rising number of blocks residing inside the cache, it can be much cheaper to discard the oldest block instead of the least recently used one [12]. According to [20], the FIFO algorithm has to be considered as non-usage-based one, due to the fact that the usage of a block does not improve its replacement status.

FIFO can be easily implemented by again equipping every set with a modulo E counter. This time, the counter is incremented after every replacement in this set.

Indeed, the replacement decision procedure can be extremely simplified. If invalid blocks are present in the cache, they should be the primary targets for removal. Beside the fact that their presence is only a waste of space, the execution of instructions by a processor on invalid blocks will definitely lead to an erroneous behavior of the system [17].

Although LRU algorithms are the most expensive strategies concerning hardware costs, they are also the most efficient ones concerning decreasing the miss rate. Nevertheless, the performance gap between the LRU algorithm in its most basic form and the optimal replacement strategy is huge [15]. The optimal replacement algorithm is defined as an algorithm that provides the best reachable minimal miss rate. In 1966 Belady stated that no real algorithm can ever reach the performance of an optimal replacement procedure since it needs to know about all future block references [5] when bringing new data to the cache. Gathering this important information would require a pre-run of the program with a following backwards assignment of blocks from the memory to their positions in the cache to construct a minimum replacement sequence. However, executing a program two times is not an option as it completely travesties the performance issues. As a result, such an approximated "optimal" algorithm, as described by Belady is solely applicable for analytic purposes, especially for the comparison of its optimal miss rates with those of feasible caching strategies.
According to [15] in highly associative caches such as L2 caches which use an LRU algorithm as replacement policy, the increase in cache misses can be up to 197% compared to the application's corresponding optimal algorithm. The reason for this degradation is that if a block inside the cache is used very frequently for a period of many cycles, its probability to be replaced shrinks with every access. If suddenly the processor starts to perform uniformly spread accesses on its cache, it will take some time for the block to become the least recently used one and to be finally discarded. In this particular situation, the former frequently accessed block is now referred to as *dead block*. The *dead time* is defined as the period of time between the event of the block becoming dead and the event of the block being kicked out of the cache. This dead time becomes worse with higher associativity due to a better distribution of accesses over the complete cache space. While the block placement performance improves by allowing more blocks to reside in a set, the replacement performance becomes worse. Dead blocks should be removed as soon as possible because they are unnecessarily occupying important cache space which is generally rare.

2.3.2 Improvements - Counter Based Algorithms

To reduce the negative impact of dead blocks on the cache performance, Kharbutli and Solihin [15] propose two special counter based algorithms taking advantage of the following fundamental idea: Every time a counter exceeds a certain threshold value, the block expires and is discarded to gain more free space, thus reducing the number of dead blocks and their resulting capacity and conflict misses. Algorithms using this technique, which is referred to as *dead line prediction technique*, store unique and dynamically learned threshold values for every block in

a small prediction table. Dead line prediction especially helps to keep frequently but not bursty accessed blocks inside the cache which are generally discarded by reason of their unsteady accesses.
The time duration in which a block is present in the cache is called *generation time*. It is divided into a *live time*, the time period in which the block is accessed by the processor and a *dead time*, the time from the last access to the deletion. The main intention of these algorithms is to keep the dead time as low as possible. An example *life cycle* of a block inside a cache memory is visualized in Figure 2.12. The time interval illustrated by $\triangle$ is referred to as the time between two accesses to the block. It is simply named *access interval*.

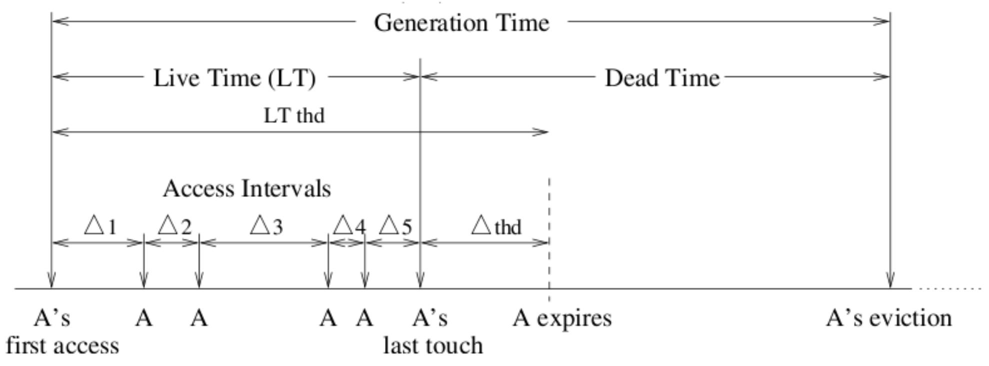


Figure 2.12: The life cycle of a cache block [15].

The *AIP (Access Interval Predictor)* algorithm works as follows: It records the number of accesses to the set where the block is stored. The counter is reset when accessing the block itself. If the counter reaches some threshold value $\triangle_{thd}$, the cache controller assumes that its life time has expired and as a result, the block is classified to be dead and can be discarded. The threshold value $\triangle_{thd}$ is learned from the past generations of the block and is typically chosen as the maximum from all previous accesses.
The *LvP (Live-time Predictor)* algorithm on the other hand estimates the live time of a block by counting the accesses to the block itself. Again, its threshold value LT_{thd} is dynamically learned by taking the maximum of all previous live times.
Figure 2.13 shows the block diagram of the proposed cache architecture. The hashed program counter (XOR function over all Bytes of the PC) of the instruction that misses on a cache block together with the 8-bit hashed address of this block is used to access the prediction table. The

field *C* includes the event counter which counts the events of interest. The past and the actual maximum value of this counter are stored in $maxC_{past}$ and $maxC_{present}$ respectively, whereas $maxC_{past}$ obtains its values from the prediction table. A block is free to be removed if the current counter value is both bigger than the present maximum and the past maximum. The reason why the local counter value must also be compared with the past maximum is because the actual generation may not have reached its maximum. Consequently the present maximum might not include the real maximum value. As the counter value in the LvP algorithm is not reset during the generation, the $maxC_{present}$ field can be ignored since it holds a value equal to the local counter field. On discarding a block from the cache, the actual counter value is used to update the $maxC_{stored}$ stored in the prediction table. Detailed information about the algorithms can be found in [15].

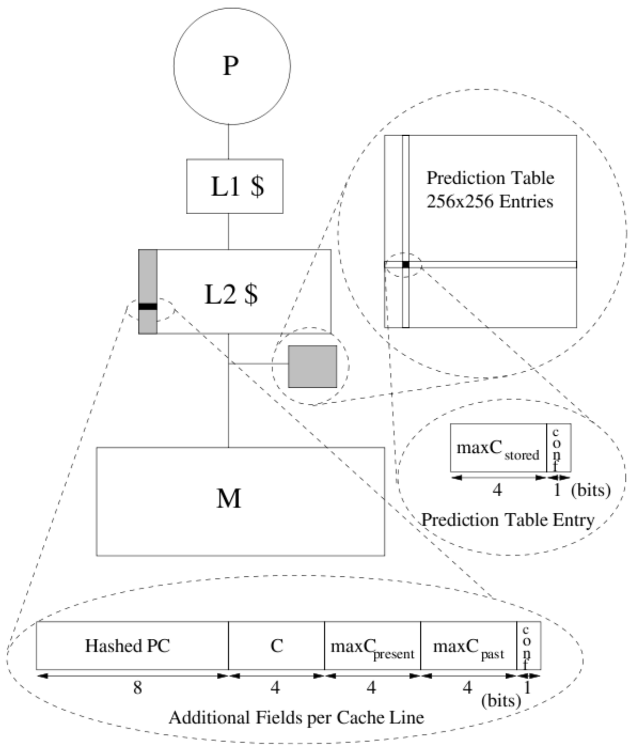


Figure 2.13: Block diagram of the proposed counter cache architecture [15].

AIP and LvP have been benchmarked with the result of speeding up 10 out of 21 Spec2000 applications by up to 40% and 11% on average. As every block is expanded with additional 21 bits, the hardware overhead is stated with 4.1% storage overhead for a 64 Bytes block plus an extra 40 KBytes tagless direct mapped prediction table. Although the overhead appears to be huge, this need for extra resources is considerably smaller than the overhead caused by the next presented improvement.

2.3.3 Improvements - Sequence Based Algorithms

To gain the ability to identify dead blocks, *Sequence Based Algorithms*, which are similar to the just presented counter based algorithms, store encoded traces or *sequences* which are n subsequent memory access events that lead to the removal of the block. In [3], the authors present a *Dead-Block Predictor (DBP)* that uses a so called *dead-block table* as storage place for these signatures. A *history table* stores a recent trace for every tag from L1 cache. Every change of such a trace forces the DBP to compare the actual history of a tag with its corresponding entry in the dead-block table. On a match, the block becomes a candidate for removal. The predictor learns new sequences by simply storing the corresponding trace of the history table in the dead-block table when a block is kicked out of the cache. Accuracy can be increased by additionally equipping every trace with two bit saturating counters to determine the confidence of the prediction. Regrettably, for every block there can be many more than just a few such sequences. Recording all these events will result in huge prediction tables that can be several MBytes in size [15].
Figure 2.14 illustrates the idea of the proposed cache architecture. Truncated addition is used to encode every trace to gain a compact representation. Referencing block A2 in the cache triggers the controller to compare the corresponding trace from the history table with the entry in the dead-block table. Since the trace PCi, PCj and PCk results in a match, the block A2 is a candidate for removal.
The DBP can easily be extended with a prefetch mechanism by additionally storing some address in the dead-block table which is assumed to be following the block that is predicted to be dead. Higher accuracy can be achieved by also storing prior accessed addresses. Nevertheless, the more information is used for the prefetching of the subsequent addresses, the more storage place is needed which might slow down the performance of the cache controller. Algorithms following this approach are named *Dead-Block Correlating Address Predictors*. Figure 2.15 shows an example where again block A2 is referenced. The trace in the history table includes some prior reference to the memory address A1 which is mapped to the same block as A2. A look-up in the dead-block table reveals that the block is dead and can be discarded. Address A3 is recommended to be prefetched.
Compared with counter based algorithms, sequence based algorithms have some little timing advantage as they mark blocks to be dead immediately whereas counter based algorithms have

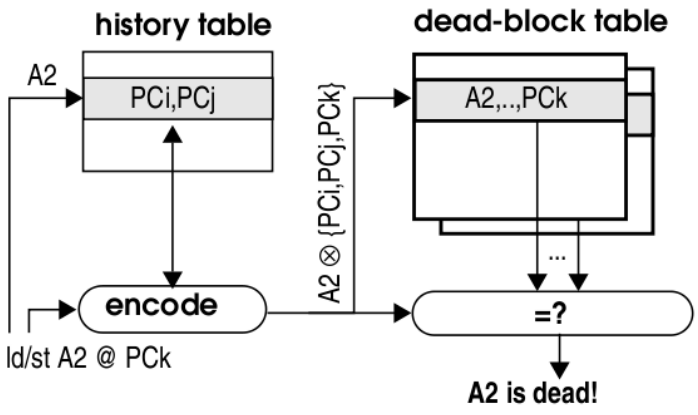


Figure 2.14: Architecture of a sequence based replacement cache controller [3].

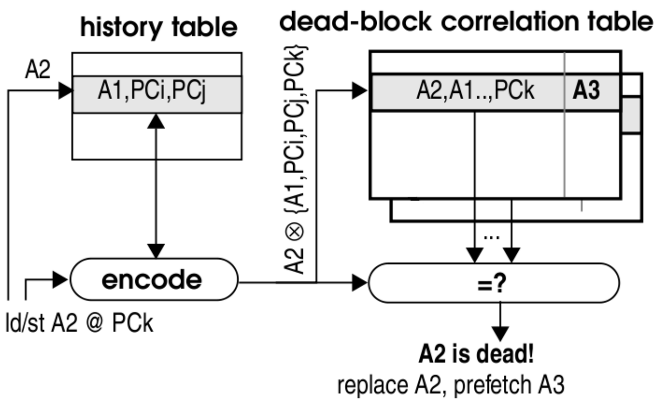


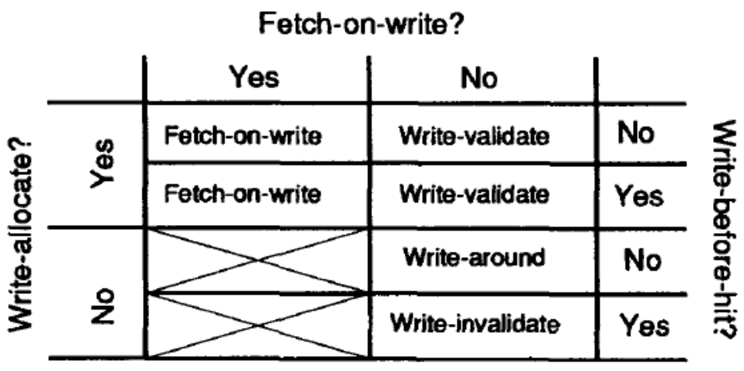
Figure 2.15: Architecture of a sequence based replacement cache controller with address prediction [3].

to wait for the threshold value to be reached. On the other hand, as already stated, sequence based algorithms request a huge amount of memory for the storage of the sequences for all cache blocks. In addition, if some sequence is a subset of another longer sequence, it might be the case that the cache controller wrongly marks the block as dead after having passed the shorter sequence although it might be needed by the processor in the next cycles. This miss prediction can not happen in a counter based cache controller as the threshold values are conservatively chosen as the maximum of previous life times.

2.4 Write Operations

All performance issues and cache internal operations so far have been related to cache read accesses. Now it is time to focus on the contrary access operation, the write request. In contrast to the read operation were it is desirable to achieve a latency that is as low as possible, the main aim of every efficient cache write policy is definitely to save bandwidth and to reduce the frequency of the write traffic to the main memory [13]. In comparison with the read access, there exists one fundamental drawback. Before any write operation can be started, the processor must determine the type of access. Usually, memory access operations can be performed byte-wise, word-wise and double word-wise. As a result, data and addresses must be appropriately adapted by some special combinatorial logic for every of these access types [12]. A practical example for this particular type of problem will be given in *Chapter 4*. A read operation on the other hand can always safely return the complete data word.
Again, the tag check operation results in a *write hit* if the corresponding data word can be found in the cache and in a *write miss* in the other case. There exist different strategies for both situations.
In the case of a write hit, two different approaches can be applied. On the one hand, the information can be immediately written into both, cache memory and main memory. This strategy is referred to as *write through* policy. On the other hand, the information might at first only be written into the cache. The corresponding block in the main memory is updated when it is replaced in the cache by some other block. In this case, the cache is said to implement a *write back* strategy. In order to identify an altered block, dirty bits must be added that mark all blocks that have been modified via write operations since they have been fetched from the main memory. Every block that is replaced and which has a cleared dirty bit does not need to be written back to the main memory, thus saving memory bandwidth. In addition, multiple writes to the same block in the cache require only one write operation on the memory, thus profiting from the locality of space and time. This reduction in the required bandwidth certainly saves power which makes the *write back* strategy attractive to be used in embedded systems. Nevertheless, it is more complicated to be implemented. According to [20], another issue which has to be taken into account when deciding either for the *write back* or the *write through* strategy is the maintenance of data coherency. Concerning a multiprocessor system or multilevel caches, *write through* will certainly be the better decision concerning an easy management of the data coherency. The main memory can be easily used as shared storage place. Otherwise the cache memory must be shared or some sophisticated coherency protocol must be implemented. Nevertheless, even when using the *write through* policy, care must be taken when data is altered by some other device in the main memory that is also residing in the cache memory of the processor.

The performance of both write strategies can be significantly improved by using a write back buffer. In fact, such a buffer is obligatory for the *write back* in order to provide the possibility of simultanoues fetch and write back operations. In the case of *write through*, several write requests might be stored to allow the processor to continue with its execution. Nevertheless, an additional forward logic will be needed as successive instructions might request the data words from the main memory that are yet residing in the buffer, waiting to be written into the main memory. If write requests are not buffered, the processor is said to suffer from *write stalls*.
When using a direct mapped placement policy, the performance can be increased by implementing a so called *write before hit* policy which always starts to write to the cache memory although the compare operation with the tag memory has not been finished yet. Indeed, in the worst case the written block has to be declared as invalid if the tag check operation yields a write miss. Care must be taken in order to make sure that the destroyed block has already been written back to the main memory before it has been altered.
Concerning write misses, it is usual to distinguish between *no fetch on write* and *fetch on write* operations. In the first case, the only action to be taken on a miss is to update the data word in the main memory. On the other hand, a cache controller implements a *fetch on write* strategy if it additionally fetches the complete block with the updated data word from the memory and places it in the cache. The property that enables the cache controller to even do a lookup in the cache to check if a requested data word is available is referred to as the *write allocate* property. When configured as *no write allocate*, there will never be a lookup in the cache memory. Indeed, *no write allocate* combined with *fetch on write* makes no sense at all since the cache is always bypassed and can therefore not be written. When the cache controller is configured as *write allocate* with *no fetch on write*, the updated data word will be written into the cache block without fetching the corresponding block from the main memory. To maintain data coherency, all valid bits without the one for the actually written data word in this block must be turned off. This policy is called *write validate*. When using *no write allocate* and *no fetch on write*, two cases have to be distinguished depending on the usage of the *write before hit* strategy. First, when writing to the memory before having checked the entries from the tag memory, all write operations triggered by a miss must be declared invalid since *no write allocate* is used. This policy is named *write invalidate*. In the other case, a *write around* strategy is used as no write operation ever reaches the cache.
Using *write before hit* has no influence on the validity of the blocks stored in a cache that implements a *fetch on write* strategy as the block will always be reloaded from the memory. Figure 2.16 illustrates all write strategies in a tabularly form.



		Fetch-on-write?		
Write-allocate?		Yes	No	Write-before-hit?
	Yes	Fetch-on-write	Write-validate	No
	Yes	Fetch-on-write	Write-validate	Yes
	No		Write-around	No
	No		Write-invalidate	Yes

Figure 2.16: All write strategies on a miss [13].

2.5 Prefetching

Cache fetch algorithms are usually requesting data words from the main memory on demand from an external source like the processor. These kinds of accesses are referred to as *actual requests*. One approach to lower the compulsory miss rate and to reduce the from a miss resulting number of penalty cycles can be achieved by the usage of *prefetch* algorithms [20]. A cache controller taking advantage of prefetching mechanisms tries to predict which data or instructions words tend to be referenced in the next upcoming cycles and if not present in the cache, attempts to preload them before they are requested by the processor. This second type of cache accesses has been given the appellation *prefetch lookup*. In the best case, the complete number of penalty cycles when performing some memory access operation can be hidden if the next block to be requested can be prefetched while operating on the currently referenced one. To achieve this best case situation it is necessary to benefit from unused memory bandwidth as much as possible without colliding with demand fetches [12]. A major drawback of the proposed strategy is that this performance increase can easily turn into a performance degradation when not designing the predictor carefully enough. Prefetching data blocks which will not be used in the next cycles might replace blocks in the cache which tend to be referenced soon, forcing the cache controller to reload them from the main memory when they are requested. In this particular situation, the cache memory is said to be *polluted*.

The ratio of the number of blocks transfered by prefetching to the total number of data memory accesses is denoted as *prefetch ratio*. The *transfer ratio* is defined as the the sum of the prefetch ratio and the cache miss ratio. Letting D be the number of penalty cycles caused by a demand miss, P the number of cycles needed for a prefetch operation and A the penalty costs resulting from an interfering prefetch access with a demand fetch, a prefetch algorithm will be effective if and only if the following equation holds [20]:

$$D * mr_{no\_prefetch} > [D * mr_{prefetch} + P * pr + A * (ar - 1)] \tag{2.11}$$

where mr_x stands for the miss ratio of the corresponding access mode, pr for the prefetch ratio and ar for the access ratio on the main memory.

The efficiency of a prefetching algorithm depends on several concerns. At first, the probably most important factor on the degree of memory pollution is the block size. The higher the block size, the more useless data words will be loaded into the cache in the case of a miss-predicted prefetch. On the other hand, when prefetching only a small number of data words, the overhead of the prefetching algorithm might be higher than the performance won. Other important implementation decisions are when to trigger some prefetch, which data blocks to prefetch and what replacement status to be given some prefetched block. Concerning decreasing the grade of the pollution of the memory, the cache controller might assign the highest possible replacement priority to every prefetched block so that the block is quickly removed if it was brought into the cache on a faulty prediction. As the cache controller should be designed in an efficient way in order to execute very fast, the simplest decision on the question which block to prefetch is certainly the next sequential block. This technique is referred to as *one block lookahead* (OBL). A more sophisticated decision procedure is going to be explained in the following. The start of the prefetch operation can be triggered by many events, e.g., by some cache miss on the instruction cache or when processing the last data word of some actual block being resident in the cache memory. Although the data transfer might not be finished when the next data word is requested by the processor, at least the advantage of a reduced number of penalty cycles being stalled is yielded.

Especially instruction caches can profit from the OBL prefetching approach by reason of the generally iterative execution of a program. This task is usually performed outside of the cache and implemented in the following way [12]: On a miss, two instead of one blocks are fetched from the memory. While the requested block that caused the miss is directly placed in the cache, the second block is stored outside the cache in some extra buffer that is called *instruction stream buffer*. If the next subsequent request results in a cache miss, the fetching from the main memory can be left out if the data block current in the instruction stream buffer is the one which is needed. While operating on the actual data block, the next prefetch can be started, filling the buffer with the next data block from the memory which is supposed to be requested in the near future. The same technique can be applied to prefetch data from the data memory. The Pentium 4, e.g., uses eight such stream buffers to prefetch data into its second level cache. From a processor with two four-way set associative 64 KByte caches, up to 70% of all misses can be captured with 8 stream

buffers, buffering either data or instructions.
However, prefetching instructions in a sequential fashion fails when executing branches in a program. The situation even gets worse in superscalar architectures that fetch more than one instruction per cycle. In this case, a more sophisticated mechanism is needed to successfully predict which instruction words to prefetch [11]. Simple branch prediction techniques that are usually executed in the decode stage will not hide the latency of an instruction cache miss as they are performed too late. Thus, [11] proposes to take branch prediction actions on the base of blocks instead of single instructions. Such a fixed length sequence of instructions is referred to as *flow block*. The size is upper bounded by the block size of the instruction cache in use and should be chosen in a way so that there is at least one branch instruction per flow block. On the other side, the more branch instructions are residing in a flow block, the worse the accuracy of the branch prediction will become. As soon as the first instruction of a flow block is executed by the processor, a prediction can be performed. Information needed for this task is usually stored in a flow block prediction table which includes the following entries for every flow block:

Flow Block Address Tag This address is used to locate the corresponding flow block table entry for every flow block.

Last Branch Target Address Identifies the last predicted address that has been selected after the branch.

History Counts the number of non-sequential exits in the past.

Indirect Branch Bit Indicates if there is an indirect branch instruction present in the flow block. Indirect branches are not taken as their target address is usually not predictable.

LRU is used as replacement policy. Accuracy can be increased by storing more than one target address if more than one branch instruction is present in a flow block. The algorithm can be simply described as follows: Whenever there is change from flow block *A* to flow block *B*, *B* is added as the last target address to *A*'s entry in the flow block table. The sequential bit is set according to the relationship between those two blocks, i.e., it is set if *B* is indeed the sequential neighbor of *A* and cleared in the other case. Afterwards the flow block table is searched through for the entry of flow block *B* in order to locate the next block to be prefetched into the instruction cache. These tasks are repeated for every switch between two flow blocks. Cache pollution by the way can be avoided by using a highly associative prefetch buffer instead of bringing prefetched blocks directly into the instruction cache.
So far, the focus in this subsection has been on *dynamically* prefetching algorithms that are executed directly in hardware. An alternative might be to prefetch data blocks before starting the

execution of a program. This approach, which has to be to categorized as *statically* prefetching algorithm, can help to lower the compulsory miss rate that dominates the cache misses when starting the execution of a new program. An example for statical optimization is the prefetching via a compiler [12], i.e., the insertion of data fetch instructions into the source code so that data words are loaded before they are actually needed. Anyway, inserting these extra instructions makes only sense if they can be executed overlapped with the original instructions of the program. Prefetched data can either be loaded into the cache or directly into a register. Again, care must be taken in order not to destroy the gained benefit by the overhead introduced by additional instructions. Loop unrolling is very popular concerning scheduling prefetch operations with the execution. Please refer to [12] for more options.

2.6 Conclusion

This chapter has introduced the reader into the fundamental aspects of caching and all its potential difficulties concerning finding the best adapted caching strategy for a given architecture. Beside choosing a placement and replacement policy, also parameters like block size or block count do have an enormous influence on the overall performance. Applying an optimization on one aspect will often lead to a degradation of another one. Even extensions like prefetching mechanisms which are supposed to increase the performance of a cache controller can decrease their efficiency if they are not implemented carefully enough. The behavior of the *SPEAR2* processor on some of these variable parameters, especially on different placement and replacement policies, is going to be investigated in the following chapter.

CHAPTER 3

Simulation

"What happens if a big asteroid hits Earth? Judging from realistic simulations involving a sledge hammer and a common laboratory frog, we can assume it will be pretty bad." - Dave Berry (American Writer and Humorist).

The following chapter is subdivided into two parts: First of all, a novel simulator toolchain for the *SPEAR2* ISA is presented. The reader is going to be introduced into the structure and the functionality of the simulator *SPEAR2SIM* and its generic cache interface that allows an easy connection of the processor with different cache controllers. Simulation results of programs executed on the simulated core taking advantage of these cache modules and a subsequent discussion on their efficiency are the key contribution of the second part.

3.1 SPEAR2SIM Concepts

To be able to decide for the optimal caching strategy for an application running on the *SPEAR2* processor, an investigation of different implemented cache controllers operating under changing parameters, e.g., different block sizes and block counts, has to be carried out. As the main focus lies on the estimation of hit and miss rates and timing and hardware requirements are not

Section	Contents
.text	Executable instructions.
.rodata	Constant data, read-only data.
.data	Initialized global and static variables.
.bss	Uninitialized data.

Table 3.1: Standard ELF sections and contents.

crucial for this particular part of the decision finding[1], off-the-target simulations of programs being executed on an emulated version of the core are absolutely sufficient. *SPEAR2SIM* is a cycle accurate simulator of the 32-bit data path version of the *SPEAR2* processor programmed in the language C, implementing the complete *SPEAR2* ISA, thus providing the ability to execute programs written in the architecture specific assembly language. This approach is far more efficient than simulating programs via the VHDL testbenches with *Modelsim* as a simulation in software certainly features easier and faster ways to extract the needed information about the cache usage (hit rates, miss rates etc.) by the processor in order to determine the theoretical possible increase in performance.

3.1.1 The ELF Storage Allocation

Before explaining the functionality of the simulator toolchain in detail, a short overview of the *ELF* [7] binary file format is given. As different architectures feature different ways to store data and instructions, the ELF file format uses the concept of sections to group each and every type of program data. Typically it is reasonable to choose a grouping according to the final storage destination. E.g., instruction words might be grouped in a section that will be stored in a flash memory or some other kind of read-only[2] memory. On the other hand it seems to make sense to place global variables together in a read/write memory like a standard SRAM which offers efficient read and write access operations.
Object files differ in type: *Relocatable object files*, e.g., include instructions and data words being in an optimized format for easy composition with other modules whereas *executable object files* can be directly executed on a processor. This format is usually generated by a *linker* which translates one or more relocatable object ELF files into one single executable ELF file. Table 3.1 shows the standard ELF sections together with their usual contents.

[1]Timing and Hardware requirements will be examined in the next chapter which focuses on the implementation of promising cache controllers in hardware (VHDL).

[2]At least some memory which is really difficult to write to while executing a program.

Figure 3.1 illustrates the standard ELF file format from the linking view[3]. The program header describes zero or more segments which include important information needed for the execution of the program [22]. This header is only optional for relocatable object files as they are not directly executable on their own. Information for the process of linking all included sections is stored in the section header which is on the other hand only optional for executable files. The ELF header serves as index for all contents of the complete object file.
This thesis will not go into this topic in detail. For those who are interested in the complete specification of the ELF format, further information can be found in [7]. Implementation details on the complete toolchain (gcc, linker etc...) of the *SPEAR2* architecture are the main topic of [21].

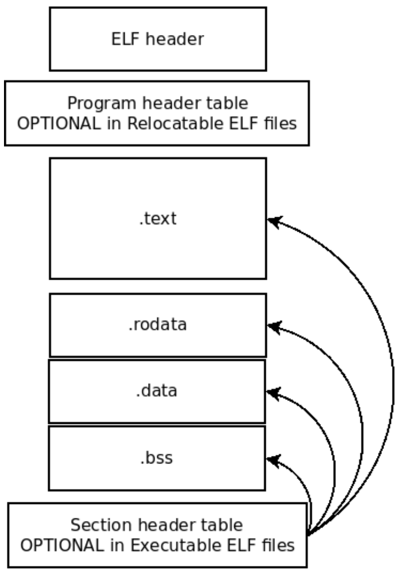


Figure 3.1: Standard ELF file format from the linking view.

3.1.2 Preprocessing of ELF Binaries

In order to perform efficient simulations with *SPEAR2SIM* it must be possible to execute programs which have been built with the original *SPEAR2* gcc-toolchain. For this reason, the executable ELF file is converted into a format that can be interpreted by the simulator. The *GNU Binutils* program *spear32-objdump* is very useful since it extracts the needed information, more precisely the assembly code and the memory contents listed in the corresponding sections, from

[3]The view on the building of a programm.

Location	$.../spear2/toolchain/spear2sim/parser$
Syntax	$./parser\ \$progname\$.txt$

Table 3.2: Location and syntax of the parser.

the executable ELF file. The output of this process is a file named $\$progname\$.txt$ that serves as input file for a parser which is responsible for separating the instructions from the memory contents.

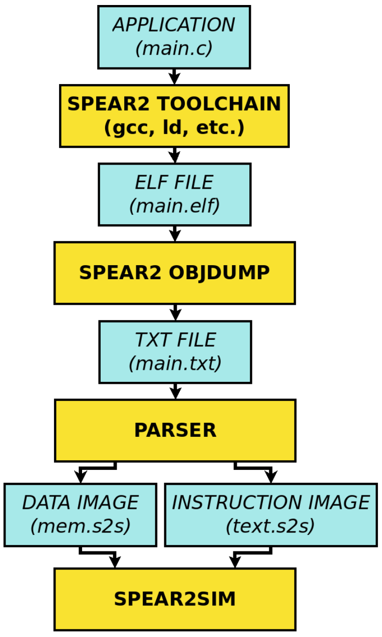


Figure 3.2: *The SPEAR2SIM* toolchain.

Processing the $.txt$ file yields the following two images: *text.s2s* and *mem.s2s* (see Figure 3.2) where *text.s2s* includes the instructions from the .text section and *mem.s2s* holds all data words that will be stored inside the simulated RAM. Indeed, the simulator does not differ between data from the .rodata, .data or .bss section. As the complete memory content will be stored into only one single simulated memory, the parser merges all information together into one image. The format of this memory image can be seen below[4]:

[4]Remember that the byte ordering of *SPEAR2* is little endian. The least significant byte is stored at the lowest address.

double word$_0$	word address$_1$	byte3	byte2
	word address$_0$	byte1	byte0
double word$_1$	word address$_3$	byte3	byte2
	word address$_2$	byte1	byte0
double word$_{...}$	...	...	...
	...	...	...
double word$_n$	word address$_{2n+1}$	byte3	byte2
	word address$_{2n}$	byte1	byte0

text.s2s is formatted in the following way:

opcode$_0$	[operand1]	[operand2]
opcode$_1$	[operand1]	[operand2]
...	...	...
...	...	...
opcode$_n$	[operand1]	[operand2]

After having generated these two images, the parsers final duty is to add an abortion term to the instructions listed in *text.s2s*. As the two last instructions of every program are a *NOP* directly followed by a *JMPI -1*, the *NOP* instruction needs to be replaced by a new halt-command in order to bring the simulation to an end, thus preventing the simulator from looping forever. In this sense, the parser scans through all instructions for this last command sequence and replaces this particular *NOP* instruction with an *END* instruction which will force the ALU to return an erroneous value, causing the simulator to stop the execution of the program. After having finished all preparation tasks, the images are copied automatically to the *SPEAR2SIM* main directory and the simulation of the program can finally be started.

3.2 SPEAR2SIM Implementation

SPEAR2SIM is a cycle accurate simulator of the *SPEAR2* ISA that has been implemented in a modular structure similar to that of the processor in order to reproduce its pipeline like program execution. The key ambition of this approach is to achieve a better understanding of the processor internal actions based on an accurate visualization of the internal stages. Figure 3.3 shows the block diagram of *SPEAR2SIM*. The main module *spear2sim.c* of the simulator can be compared with the top-level entity of the VHDL implementation of the processor core as it implements the complete pipeline framework with all pipeline registers and calls the stage corresponding functions of the simulated pipeline units, i.e., *alu.c*, *inst_mem.c*, *data_mem.c*, *reg_bank.c* and the cache modules.

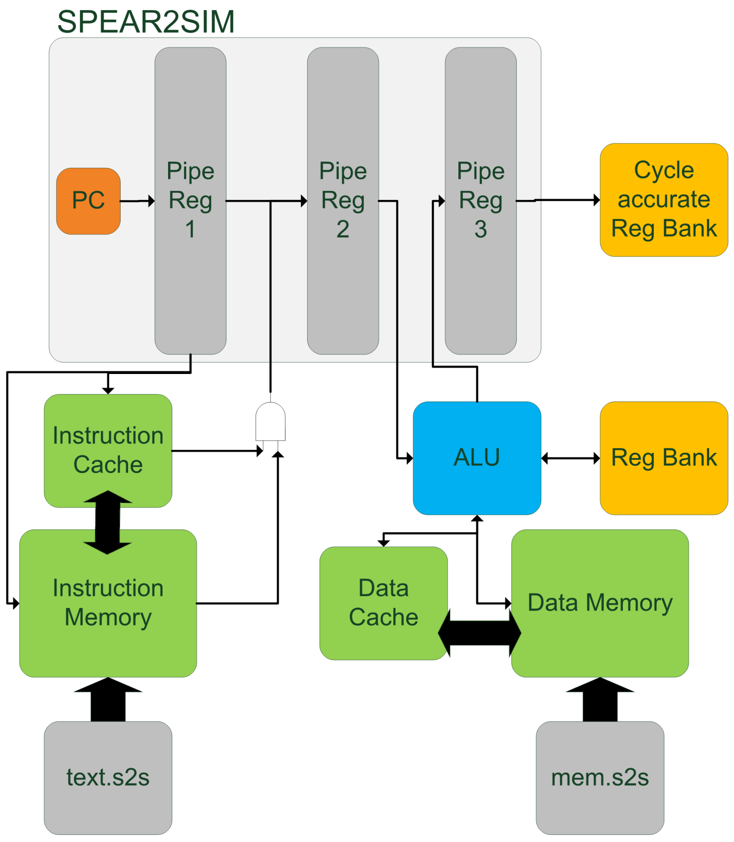


Figure 3.3: *SPEAR2SIM* block diagram.

Like in the VHDL implementation of the processor, data is transfered from one pipeline register to the next one with the difference that it is manipulated by the functional units in between in a serial way and not in parallel. This virtualization of the hardware layer offers noticeable simplifications concerning the execution of an application by the simulated *SPEAR2* pipeline. When performing memory access instructions, the simulated core does not have to wait for several penalty wait cycles to pass as data words are immediately available or stored with every load/store operation. As the same advantage also holds for the register bank, a forwarding unit does not need to be implemented to prevent the ALU from suffering from data hazards. If the ALU calculates some result, it can be directly stored into the register bank without running through the WB stage. Nevertheless, in order to provide a cycle accurate simulation, these penalty cycles are added to the cycle counter by the simulator for every memory access.
In addition, the first pipeline register (IF - ID) does not need to address and enable the instruction memory in order to prepare the next instruction word to be executed in the next cycle since the

decoding stage can access the instruction memory on its own without any delay. The reader might start to argue that there is no real need for the implementation of any pipeline register by reason of the simplified memory access. However, since it is the key intention of the simulator to offer a pipeline accurate illustration, these buffer registers are necessary to inform the user about the recent status of every pipeline stage in every clock cycle or to trace the stream of instructions running through the processor. The second pipeline register (ID - EX), e.g., includes the operands and opcode to be operated on in the next cycle whereas the last pipeline register (EX - WB) holds the result from the execute stage, the write back address and the opcode from the last instruction as well.
As all results from the ALU are immediately stored into the register bank, a cycle accurate storage of the write-back results is needed. This is achieved by creating a second register bank which behaves like the one implemented in the VHDL design of the core, thus storing values only after they have passed the WB stage. A cycle accurate visualization of the data memory is inefficient and useless due to the confusing output caused by the huge number of data words to be printed.
Another simplification is the following: *SPEAR2SIM* uses the assembler code and not the byte code of an application for its execution. Since instructions are not represented as 16-bit words like in the hardware implementation, no real decoding task is necessary but also no visualization of this task is possible. The ID stages is therefore only responsible for the forwarding of the opcode and the operands via the second pipeline register to the EX stage.
Before any simulation can be started, the instruction and data images provided by the parser have to be read in. This task is performed by special routines implemented in the instruction and data memory module. To this purpose, the generated text.s2s and data.s2s images must be placed in the same directory like the *SPEAR2SIM* executable. After the instruction memory and the data memory have been filled, the simulator will execute the program like the real processor in several clock cycles, moving operands and opcodes from one pipeline stage to the next one. The following listing shows the most important steps taken by the simulated pipeline during one clock cycle.

1. Increment the program counter.

```
pc.value = pc.value + 1;
```

2. Read opcode, operand1 and operand2 from instruction memory addressed by program counter in pipe_reg1 (IF stage) and store it in temporary buffer.

```
temp_opcode = inst_mem_read_opcode(pipe_reg1.pc.value);
temp_operand1 = inst_mem_read_operand1(pipe_reg1.pc.value);
temp_operand2 = inst_mem_read_operand2(pipe_reg1.pc.value);
```

3. Execute opcode on operand1 and operand2 stored in pipe_reg2 (EX stage).

```
execute();
```

4. Store program counter from pipe_reg1 (ID stage) in pipe_reg2 (EX stage) and store next program counter value in pipe_reg1 (ID stage). This updated program counter value in reg_bank1 will be used to fetch the next instruction.

```
pipe_reg2.pc.value = pipe_reg1.pc.value;
pipe_reg1.pc.value = pc.value;
```

5. Store opcode from pipe_reg2 (EX stage) in pipe_reg3 (WB stage).

```
strncpy(pipe_reg3.opcode), pipe_reg2.opcode, OPCODE_MAX);
```

6. Store temporary opcode, temporary operand1 and temporary operand2 in pipe_reg2 (EX stage). This will be the next instruction to be executed.

```
strncpy(pipe_reg2.opcode), temp_opcode, OPCODE_MAX);
strncpy(pipe_reg2.operand1), temp_operand1, OPCODE_MAX);
strncpy(pipe_reg2.operand2), temp_operand2, OPCODE_MAX);
```

7. Return to 1

Every simulation in *SPEAR2SIM* can be either executed in a *single step mode* or finished without any break. The second mode is referred to as *one click mode*. While the *single step mode* is especially suited for debugging purposes, the *one click mode* offers a quick extraction of important statistics, e.g., cache miss and hit rates, from the program execution. The two modes can be enabled/disabled by setting the following define before compilation visible in Table 3.3. When executing the simulator in the *single step mode*, the console interface will output important information about all passed memory accesses, cache accesses and the recent values of the

Define	**Semantic**
SINGLE_STEP	Run *SPEAR2SIM* in *single step mode*, else in *one click mode*.

Table 3.3: SPEAR2SIM execution mode defines.

Define	**Semantic**
FINAL_REPORT_MEM_ACCESS	Memory load and store access counter values.
FINAL_REPORT_DATA_CACHE_ACCESS	Data cache read and update counter values.
FINAL_REPORT_DATA_CACHE_STAT	Data cache hit, miss and conflict counter values.
FINAL_REPORT_INST_CACHE_ACCESS	Instruction cache read and update counter values.
FINAL_REPORT_INST_CACHE_STAT	Instruction cache hit, miss and conflict counter values.
FINAL_REPORT_REG_BANK_CONSOLE	Print register bank values to console.
FINAL_REPORT_REG_BANK_FILE	Print register bank values to file.
FINAL_REPORT_DATA_MEM_FILE	Print data memory values to file.

Table 3.4: SPEAR2SIM final report defines.

register bank and pipeline registers for every cycle. A final report at the end of every execution lists all these values together with some additional dumping information if desired. This final output of the simulator is fully configurable by altering the defines from Table 3.4 in the *spear2sim.h* header file.

As already stated, *SPEAR2SIM* is able to simulate the *SPEAR2* ISA taking advantage of an integrated cache memory. In this first version, several different cache controllers have been implemented. They can be activated by defining the following macros listed in Table 3.5. *USE_INST_CACHE* and *USE_DATA_CACHE* must be activated for the others defines to have an effect.

Replacement strategies for the fully associative and set associative cache controllers can be altered by the following defines from Table 3.6. For the set associative placement policy, only the FIFO replacement strategy is available. Implementing a LRU policy does not seem to be that reasonable for a 2-way set associative cache controller since a data word can only be placed at two positions.

Different cache parameters like block count, block size and so on can be found and configured

Define	**Semantic**
USE_INST_CACHE	Use instruction cache.
USE_DATA_CACHE	Use data cache.
USE_DIRECT_MAPPED_INST_CACHE	Use a direct mapped instruction cache controller.
USE_DIRECT_MAPPED_DATA_CACHE	Use a direct mapped data cache controller.
USE_FULLY_ASSOCIATIVE_INST_CACHE	Use a fully associative instruction cache controller.
USE_FULLY_ASSOCIATIVE_DATA_CACHE	Use a fully associative data cache controller.
USE_SET_ASSOCIATIVE_INST_CACHE	Use a 2-way set associative instruction cache controller.
USE_SET_ASSOCIATIVE_DATA_CACHE	Use a 2-way set associative data cache controller.

Table 3.5: SPEAR2SIM cache controller mode defines.

Define	**Semantic**
**_CACHE_REPLACEMENT_LRU*	Use least recently used replacement strategy.
**_CACHE_REPLACEMENT_FIFO*	Use first in first out replacement strategy.

Table 3.6: SPEAR2SIM cache replacement mode defines.

Define	**Semantic**
**_MEM_ADDRESS_WIDTH*	Width of the addressed memory.
**_CACHE_INDEX_WIDTH*	Width of index field.
**_CACHE_BLOCK_WIDTH*	Width of block field.

Table 3.7: SPEAR2SIM cache parameter defines.

in the corresponding header files of the cache controller in use (see Table 3.7).
By defining the macro *SPEAR2_NORMAL_MODE*, *SPEAR2SIM* will simulate the core in its standard version with integrated data and instruction memory overriding all given cache parameters. Defining neither *SPEAR2_NORMAL_MODE* nor any cache option, a *SPEAR2* processor accessing an external memory without taking advantage of a cache controller will be simulated.

3.2.1 Module Description

The next section is going to provide an elementary overview on all modules that are included in *SPEAR2SIM* with a following description of the generic cache interface. The reader which is especially interested in developing extensions for the simulator toolchain will find important information in this part of the thesis, which mainly covers design considerations related to the programming language used for the implementation. Nevertheless, details should be principally looked up in the source code or in the source code documentation.

reg.c

Simple registers of the *SPEAR2* architecture that are used as basic blocks for the implementation of the register bank and the pipeline registers are simulated by the *reg.c* module. An extract from the *reg.h* header file showing the declaration of the standard register data type and the contents of the three different pipeline registers are visualized in the following code segment:

```
/* standard register */
typedef struct reg
{
    int32_t value;
} REG;

/* pipeline register 1 */
typedef struct pipe_reg1
{
    REG pc;
} PIPE_REG1;

/* pipeline register 2 */
typedef struct pipe_reg2
{
    REG pc;
    char opcode[OPCODE_MAX];
    char operand1[OPCODE_MAX];
    char operand2[OPCODE_MAX];
} PIPE_REG2;

/* pipeline register 3 */
typedef struct pipe_reg3
{
    REG pc;
    REG alu_result;
    REG alu_write_back_address;
    char opcode[OPCODE_MAX];
} PIPE_REG3;
```

The first pipeline register only needs to store the last incremented program counter value which will be used in the next cycle in order to fetch the next instruction from the instruction memory. Just like in the hardware version of the processor, the storage of the decoded instructions is performed by the second pipeline register and allows the execute stage to obtain the information needed for the actual computational cycle. While the write back address and the result from the ALU in the last register pipeline are important for the storage of the write back results into the cycle accurate register bank, the opcode is used to show the user which has been the last instruction that completed and left the pipeline.

reg_bank.c

Defines concerning the naming of all included registers are specified in the *reg_bank.h* header file. *REG_BANK_SIZE* determines the total number of registers residing inside the register bank.

```
/* Size of register bank */
#define REG_BANK_SIZE       16

/* register name defines */
#define R0                  0
#define R1                  1
#define R2                  2
#define R3                  3
#define R4                  4
#define R5                  5
#define R6                  6
#define R7                  7
#define R8                  8
#define R9                  9
#define R10                 10
#define R11                 11
#define R12                 12
#define R13                 13
#define RTS                 14
#define RTE                 15
```

Table 3.8 provides a brief overview of all functions implemented by this module. To call the corresponding subroutine necessary for the cycle accurate output, an “*r_*“ must be attached as prefix to every function name.

inst_mem.c

In contrast to the instruction memory implemented in VHDL which stores instructions as 16-bit words, the simulated version of the memory already contains separated and decoded opcodes and

Function	**Semantic**
int8_t reg_bank_write(int32_t value, uint16_t rX)	Write value to register rX.
*int32_t reg_bank_read(int16_t rX, int16_t *errv)*	Read value from register rX.
void reg_bank_reset(void)	Reset all registers in register bank.
void reg_bank_dump(void)	Dump register bank to console.
int8_t freg_bank_dump(void)	Dump register bank to file.

Table 3.8: Functions provided by reg_bank.c.

operands that have been pre-formatted by the *SPEAR2SIM* parser. The most important defines and declarations of this module can be found in the subsequent code segment.

```
/* size defines */
#define INST_MEM_SIZE                    65530

/* penalty defines */
#define INST_MEM_PENALTY                 9

typedef struct inst_mem
{
    char opcode[OPCODE_MAX];
    char operand1[OPCODE_MAX];
    char operand2[OPCODE_MAX];
} INST_MEM;
```

An entry in the instruction memory consists of three char arrays storing the opcode and two optional operands. The complete instruction memory is implemented by creating an array of this struct. The storage capacity can be configured by altering the value of *INST_MEM_SIZE*. When simulating the *SPEAR2* core interfacing an external memory, the number of penalty cycles needed to wait for every access to complete is defined by *INST_MEM_PENALTY*.

data_mem.c

The module *data_mem.c* provides byte, word and double word accesses to the simulated data memory. All addresses must be passed as byte addresses since all modifications depending on the access type are performed internally. Anyway, memory double word and word operations have to be aligned, i.e., the address must be evenly divisible by 4 and 2 respectively.

```
/* size defines */
#define DATA_MEM_BASE            0x00000000
#define DATA_MEM_TOP             0x0007FDFF
#define DATA_MEM_SIZE            (DATA_MEM_TOP - DATA_MEM_BASE)
```

Function	**Semantic**
int8_t inst_mem_read_in(void)	Read in instructions from text.s2s image located in *SPEAR2SIM* main directory.
char inst_mem_read_opcode(uint16_t address)*	Read opcode from instruction memory.
char inst_mem_read_operand1(uint16_t address)*	Read operand1 from instruction memory.
char inst_mem_read_operand2(uint16_t address)*	Read operand2 from instruction memory.
void inst_mem_dump(void)	Dump instruction memory to console.
int8_t finst_mem_dump(void)	Dump instruction memory to file.

Table 3.9: Functions provided by inst_mem.c.

Function	**Semantic**
int8_t data_mem_read_in(void)	Read in data from data.s2s image located in *SPEAR2SIM* main directory.
int8_t data_mem_write32bit(int32_t value, uint32_t address)	Write 32 bit value to data memory.
*int32_t data_mem_read32bit(uint32_t address, int16_t *errv*	Read 32 bit value from data memory.
*int32_t data_mem_read16bit(uint32_t address, int16_t *errv)*	Read 16 bit value from data memory.
int8_t data_mem_write8bit(int32_t value, uint32_t address, uint16_t pos)	Write 8 bit value to data memory.
*int32_t data_mem_read8bit(uint32_t address, int16_t *errv)*	Read 8 bit value from data memory.
int8_t fdata_mem_dump(void)	Dump data memory to file.

Table 3.10: Functions provided by data_mem.c.

```
/* penalty defines */
#define DATA_MEM_PENALTY        9
```

The size of the data memory is defined through the address space between *DATA_MEM_BASE* and *DATA_MEM_TOP*. The number of penalty cycles needed to wait for every memory access to complete when simulating the *SPEAR2* core operating on an external memory module is determined by the value of *DATA_MEM_PENALTY*.

Function	**Semantic**
int execute(void)	Execute instruction on operands stored in pipeline register 2.
*int32_t convert_operand_reg(char *operand)*	Convert a register number from string to numerical.
*int32_t convert_operand_int(char *operand)*	Convert an operand from string to numerical.

Table 3.11: Functions provided by alu_op.c.

alu_op.c

The *alu_op.c* module is responsible for two important tasks: As all operands stored in the instruction memory are represented as character arrays, they must be converted into numerical data types in order to be able to execute logical or arithmetical operations on them. Furthermore, the register numbers must be determined from given register names. The second task is to call the correct subroutine from *alu.c* depending on the actual opcode stored in the second pipeline register. In the case of a requested *END* instruction, the simulation is stopped triggered by a corresponding error-return value of the *execute()* function.

alu.c

This central functional unit of the simulator implements the complete *SPEAR2* ISA, i.e., each instruction from the processor is simulated by exactly one subroutine from *alu.c*. For a more detailed reference on all these functions and instructions, e.g., information on the behavior of the processor internal flags during different operations, the interested reader is referred to the appendix of [21] or to the source code documentation.

3.2.2 Generic Cache Interface

To allow an easy development and integration of new cache controllers, *SPEAR2SIM* defines a generic cache interface to the simulated core. Its obligatory declarations and functions together with simple examples are going to be explained on the following pages. These code examples primary illustrate the routines of a direct mapped data cache. The routines for the instruction cache implementing the same placement strategy are similar expect for the data types that are operated on.
The generic data type declarations for the cache memory and the cache blocks are defined in the shared main cache header file *cache_interface.h*. As it is visible in the following code segment, an entry of the data cache memory is implemented as a structure including a two dimensional

data field, one dimension for the blocks and one for the data words respectively. Another one dimensional array serves as the tag memory. Extra fields like an usage counter for every cache block which are used by more sophisticated block replacement algorithms like the LRU must be implemented locally in the corresponding cache controller. These generic data type declarations serve as a minimal interface to all feasible cache policies and should therefore not be extended.

```
typedef struct data_cache
{
    uint32_t address[DATA_CACHE_SIZE];
    int32_t mem[DATA_CACHE_SIZE][DATA_CACHE_BLOCK_SIZE];
} DATA_CACHE;

typedef struct data_cache_block
{
    int32_t data[DATA_CACHE_BLOCK_SIZE];
} DATA_CACHE_BLOCK;
```

Table 3.12 gives a brief overview on all compulsory functions which are required by a new cache controller to be implemented. As all function calls by the pipeline are hard coded it is important for every new cache controller to implement these necessary subroutines exactly as explained in the following to guarantee the correct functionality of the processor!

Function	Semantic
void init_data_cache(void)	Initialize data cache tags with default values.
void data_cache_write_block(DATA_CACHE_BLOCK *block, uint32_t mem_block_address)	Write a block of data to data cache.
uint32_t data_cache_search_block(uint32_t mem_address, int16_t *errv)	Check data cache for the requested data word.
int32_t data_cache_fetch_data(uint32_t mem_address, int16_t *errv)	Fetch data word from data cache. Update the block in case of a miss.
void data_cache_update(int32_t value, uint32_t block_address, uint32_t word_address)	Update a word in a block.
void data_cache_dump(void)	Print all data cache entries on the console.
int8_t fdata_cache_dump(void)	Print all data cache entries to a file.

Table 3.12: Cache interface overview.

void init_data_cache(void)

The initialization function is needed to fill all tags with default values that must not be mixed up with the values that will be stored during the execution of a program. Letting all tags e.g. being initialized with zeros, accessing block 0 from the main memory would instantly lead to cache hit although the corresponding data has never been brought into the cache. The following execution on erroneous data words would lead to a mail-functional behavior of the processor. Consequently it is certainly the better choice to choose an initialization value which is outside the range of the tag address space. An example for a reasonable initialization can be seen below:

```
/*!
 * \brief Init data cache tags with default values
 */
void init_data_cache(void)
{
    int32_t i = 0;

    for(i = 0; i < DATA_CACHE_SIZE; i++)
    {
        data_cache.address[i] = 0xFFFFFFFF;
    }
}
```

As the width of the tag will always be smaller than 32 bits and the initialization value is chosen to be 0xFFFFFFFF, there will never be some mix up with the tag addresses referenced by the processor. By initializing the cache memory in this way, there is no need for an additional valid bit. Furthermore, this function might also include the initialization procedure for extra counters like the LRU fields needed in more sophisticated replacement policies.

data_cache_write_block(DATA_CACHE_BLOCK *block, uint32_t mem_block_address)

This function should be called whenever some block of data must be brought into the cache after a memory load operation that resulted in a cache miss. In other words, this function implements the placement and replacement strategy of the cache controller. Address translation from the memory block address to the corresponding cache block address must be executed following the global placement policy. The next code segment shows the write procedure and replacement decision of a direct mapped cache controller.

```
/*!
 * \brief Write block of data to data cache
 * \param *block Block of data
 * \param mem_block_address Memory block address
 */
```

```
static void data_cache_write_block(DATA_CACHE_BLOCK *block, uint32_t mem_block_address)
{
    int32_t i = 0;
    uint32_t data_cache_block_address = 0;

    /* address translation - get cache_block_address */
    data_cache_block_address = mem_block_address % DATA_CACHE_SIZE;

    /* if address tag is not default value - CONFLICT */
    if(data_cache.address[data_cache_block_address] != 0xFFFFFFFF)
        data_cache_conflict_counter++;

    /* write new address tag */
    data_cache.address[data_cache_block_address] =
    (mem_block_address >> DATA_CACHE_INDEX_WIDTH);

    /* write block into cache */
    for(i = 0; i < (DATA_CACHE_BLOCK_SIZE); i++)
    {
        data_cache.mem[data_cache_block_address][i] = block->data[i];
    }
}
```

As it is usual for direct mapped cache controllers, the one and only possible cache internal position is calculated by taking the memory block address modulo the cache size, i.e., the overall number of blocks that can be stored inside the cache memory. A conflict occurs when the already stored entry in the tag memory is not equal to the initialization value. The new tag address is calculated by a right shift of the memory block address by the number of bits used for the index field.

uint32_t data_cache_search_block(uint32_t mem_address, int16_t *errv)

The task of this subroutine should be the localization of a requested data word in the cache, i.e., the identification of the corresponding cache position and the following comparison of the stored tag with the tag field of the referenced memory address. For a direct mapped cache controller, such a procedure might be implemented in the following way:

```
/*!
 * \brief Check the data cache for data
 * \param mem_address Memory address (bytes)
 * \return DATA_CACHE_MISS if data is not found (check errv), number of block else
 */
uint32_t data_cache_search_block(uint32_t mem_address, int16_t *errv)
{
    uint32_t mem_block_address = 0;
    uint32_t data_cache_block_address = 0;
    *errv = DATA_CACHE_HIT;

    /* get mem block address - clear bits for block-internal addressing */
    mem_block_address = ((mem_address) >> (DATA_CACHE_BLOCK_WIDTH + 2));
    /* get chache block address */
    data_cache_block_address = mem_block_address % DATA_CACHE_SIZE;

    /* look for block in cache - direct mapped - only one tag needs to be checked */
    if((mem_block_address >> DATA_CACHE_INDEX_WIDTH) ==
    data_cache.address[data_cache_block_address])
        return (data_cache_block_address);

    /* if not found */
    *errv = DATA_CACHE_MISS;

    return DATA_CACHE_MISS;
}
```

The selection of the targeted block is again performed by the same modulo operation presented in the write procedure. The same holds for the extraction of the tag address from the memory block address.

int32_t data_cache_fetch_data(uint32_t mem_address, int16_t *errv)

As can be concluded from the name, this subroutine is responsible for the fetching of the requested data word from the cache memory. This task includes the calling of the corresponding search routine and the retrieving of the complete block from the main memory in the case of a cache miss as well with a subsequent write access on the cache. If the search yields a cache hit, the requested data word is extracted from the block and returned to the calling function.

```
/*!
 * \brief Fetch data from data cache. Update block in case of a miss
 * \param mem_address Memory address (bytes)
 * \param *errv Error variable
 * \return DATA_CACHE_MISS in case of a miss (check errv), data word else
 */
int32_t data_cache_fetch_data(uint32_t mem_address, int16_t *errv)
{
    uint32_t data_cache_block_address = 0;
    uint32_t word_address = 0;
    uint32_t mem_block_address = 0;
    uint32_t mem_read_address = 0;
    DATA_CACHE_BLOCK temp_block;
    int32_t i = 0;

    int16_t lerrv = 0;
    *errv = DATA_CACHE_HIT;

    /* look for block in cache */
    data_cache_block_address = data_cache_search_block(mem_address, &lerrv);

    /* if not found return a MISS */
    if(lerrv == DATA_CACHE_MISS)
    {
        *errv = DATA_CACHE_MISS;

        /* get mem block address */
        mem_block_address = ((mem_address) >> (DATA_CACHE_BLOCK_WIDTH + 2));
        mem_read_address = mem_address & (0xFFFFFFFF -
        ((int32_t) pow(2, DATA_CACHE_BLOCK_WIDTH + 2) - 1));

        /* read data from memory */
        for(i = 0; i < DATA_CACHE_BLOCK_SIZE; i++)
        {
            temp_block.data[i] = data_mem_read32bit(mem_read_address +
            (4 * i), &lerrv);
        }

        /* write data to cache */
        data_cache_write_block(&temp_block, mem_block_address);

        return DATA_CACHE_MISS;
    }

    /* get word_address - kill all bits except the ones needed for block
    internal addressing */
    word_address = (mem_address / 4) & ((DATA_CACHE_BLOCK_SIZE) - 1);

    /* return desired data word */
    return data_cache.mem[data_cache_block_address][word_address];
}
```

void data_cache_update(int32_t value, uint32_t block_address, uint32_t word_address)

This subroutine simply implements the cache memory write update procedure in the case of a write hit.

```
/*!
 * \brief Update word in block
 * \param value New value
 * \param block_address Address of block
 * \param word_address Address of word inside block
 */
void data_cache_update(int32_t value, uint32_t block_address, uint32_t word_address)
{
    /* update word in block */
    data_cache.mem[block_address][word_address] = value;
}
```

void data_cache_dump(void)

The obligatory dump function can be used to print the currently stored contents of the cache on the console. Indeed this function is only applicable on small cache implementations, e.g., for debugging reasons at the very start of the implementation of a new design. For larger capacities, *fdata_cache_dump* might be the better choice.

```
/*!
 * \brief Print all data cache entries on console
 */
void data_cache_dump(void)
{
    int i = 0;
    int j = 0;

    for(i = 0; i < DATA_CACHE_SIZE; i++)
    {
        printf("BLOCK TAG : %x\n", data_cache.address[i]);

        for(j = 0; j < (DATA_CACHE_BLOCK_SIZE); j++)
        {
            printf("BLOCK %d - DATA %d : %x\n", i, j, data_cache.mem[i][j]);
        }
        printf("\n");
    }
}
```

int8_t fdata_cache_dump(void)

Dumps the cache contents to a dump file instead on the console. As already explained, this function should be used for simulations using a cache memory with a high storage capacity.

```
/*!
 * \brief Print all data cache entries in a dump file
 * \return DATA_CACHE_MISS in case of an error, DATA_CACHE_HIT else
 */
int8_t fdata_cache_dump(void)
{
    FILE *data_cache_dump;
    int i = 0;
    int j = 0;

    /* open dump file */
    if((data_cache_dump = fopen("dump/data_cache_dump", "w")) == NULL)
```

```
        return DATA_CACHE_MISS;

    for(i = 0; i < DATA_CACHE_SIZE; i++)
    {
        fprintf(data_cache_dump, "BLOCK TAG : %x\n", data_cache.address[i]);

        for(j = 0; j < (DATA_CACHE_BLOCK_SIZE); j++)
        {
            fprintf(data_cache_dump, "BLOCK %d - DATA %d : %x\n", i,
            j, data_cache.mem[i][j]);
        }
        printf("\n");
    }

    /* close dump file */
    fclose(data_cache_dump);

    return DATA_CACHE_HIT;
}
```

Interfacing the Core - Example

The next code segment from the ALU shows the application of the cache interface, i.e., the hard coded call of the cache fetch function by the core.

```
#ifdef USE_DATA_CACHE
        errv = DATA_CACHE_HIT;

        /* search for data in cache */
        data_cache_return = data_cache_fetch_data(valY, &errv);

        /* if not found in the cache */
        if(errv == DATA_CACHE_MISS)
        {
            errv = DATA_MEM_SUCCESS;
            data_cache_miss_counter++;

            /* read value from mem */
            valX = data_mem_read32bit(valY, &errv);

            if(errv == DATA_MEM_FAILED)
                return DATA_MEM_FAILED;

            /* mem access penalty - pipeline is frozen at that time */
            cycle_counter = cycle_counter + DATA_MEM_PENALTY + DATA_CACHE_BLOCK_SIZE;
        }
        else
        {
            /* if found, get it from the cache */
            valX = data_cache_return;
            data_cache_hit_counter++;
        }

        data_cache_read_access_counter++;

#endif
```

This example code has been extracted from a memory load operation which firstly tries to fetch the requested data word from the cache. In the case of a miss, the cache controller will update the corresponding cache memory location internally. The final access to the external data memory forwards the needed data word to the processor. Please note that this is the very first beta version of *SPEAR2SIM*, i.e., many updates and modifications are likely to be applied in the future. For an up-to-date version of the simulator tool chain, please check the *SPEAR2* homepage.

3.3 Simulation Results

This subsection serves as a turtorial for the procedure of finding the best caching strategy for a given application. As different programs offer different kinds of temporal and spatial localities, these benchmarks should not be interpreted as general performance indicators for the presented cache controllers. The next pages reveal the key benefit of *SPEAR2SIM*, i.e., the possibility to efficiently find an appropriate cache controller for a program that is going to be executed on the *SPEAR2* processor. Nevertheless, excessive benchmarks that are concerned with the general influence of different cache sizes, block sizes etc. on the overall performance and on the different kinds of miss rates can be found in [12].

3.3.1 A Little Motivation

Before comparing the hit and miss rates of two benchmark applications running on different cache controllers tested in several simulation runs, Figure 3.4 demonstrates to the reader the positive effect of caching on the overall system performance. The bars drawn in the Figure represent the number of cycles needed by three different versions of the simulated processor for their executions of the benchmark program *cache_benchmark1.c* (see appendix). The core simulated in the *INT_MEM* mode corresponds to the original version of the *SPEAR2* architecture equipped with internal synthesized data and instruction memory. The total number of cycles is considerably less compared to those of the other configurations as all data words and instructions are immediately available in the subsequent cycle after the memory access request. Letting the data and instruction memory reside in an external RAM like it is simulated in the *EXT_MEM* mode, the performance begins to degrade to a level that is unacceptable for most reasonable applications. This situation resulting from the higher latency for every memory access becomes even worse with the data and instruction memory residing in the same physical memory and sharing the same interface to the pipeline[5]. The disparity of needed cycles between the *EXT_MEM* and the *INT_MEM* configuration can approximately stated as $1:6$. To the rescue, even two simple cache controllers for the data and instruction memory that implement a direct mapped placement policy with a block count of 64 and a block size of 4 (512 Bytes) can help to significantly improve the performance again. The cycle count in the *CACHE_MEM* configuration considerably decreases to a value near the one from the *INT_MEM* configuration.

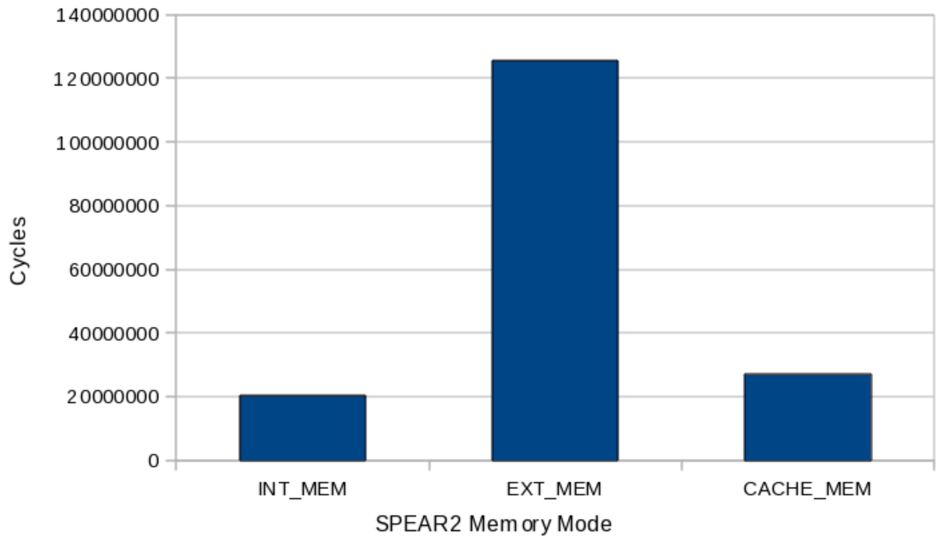


Figure 3.4: Performance boost gained by caching.

The upcoming measurements are going to investigate the performance issues of programs running on the *SPEAR2* processor taking advantage of the following three cache controllers:

- Direct Mapped
- 2-Way Set Associative
- Fully Associative

[5]A detailed description of the transformation from a Harvard to a Van Neumann architecture will be given in the next chapter.

3.3.2 Cache Benchmark 1

The first measurements focus on the performance of the three cache controllers acting as instruction caches. The benchmark program in use, i.e., *cache_benchmark1.c*, represents a simplified mix of excessive loops accessing a huge area of the data memory address space mixed with a number of sequential statements that have been implemented as simple *NOP* chains since the meaning and the result of the program are not that interesting. The main duty of the sequential statements beside a simulation of general program tasks is to interfere with the loop instructions in order to reduce the advantage gained by the temporal locality resulting from executing instructions in a loop. The following code segment shows two example functions that execute this excessive memory access over the data memory address space, intercepted by sequential *NOP* chains that are executed by calling the subroutines *foo5()* and *foo6()*. The included *NOP_CHAIN* macros in *sum_array* will not have any negative influence on the temporal locality as they are themself part of the instructions in the loop. The rest of the program is build by similar subroutines. For more information on the benchmark programs, the interested reader is recommended to have a closer look on the appendix.

```
for(j = 0; j < LOOP_COUNT; j++)
{
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        foo5();
        sum = sum + a1[i];
        foo6();
        sum = sum + a4[ARRAY_SIZE - i - 1];
        foo5();
    }
    NOP_CHAIN
    foo5();
}

for(i = 0; i < ARRAY_SIZE; i++)
{
    foo5();
    a3[i % ARRAY_SIZE] = a2[(i + 20) % ARRAY_SIZE];
}
```

Since *cache_benchmark.c* holds only approximately 1754 instructions, the instruction cache memory must not have a capacity higher than this number. As no commercial or bigger benchmark programs have been available for testing the performance of the processor equipped with the new memory layer, all simulations have been performed using only comparatively small instruction caches. This is no real disadvantage at all since the values extracted from the simulations will scale with the size of the program and the usage of bigger caches. Anyway, the needed

Function	Instruction Memory Address	Instruction Cache Address
sum_array()	0xE8 $\rightarrow$ 298	E8 $\rightarrow$ 98
foo5()	0x93A $\rightarrow$ A3E	3A $\rightarrow$ 3E

Table 3.13: Instruction Memory occupied by *sum_array* and *foo5*.

stressing of the cache by frequent memory accesses is achieved by a clever implementation of the program that will lead to an overlapping of instructions in the cache memory.
The following execution of the benchmark program now presents the danger of a performance degradation when using a direct mapped placement strategy caused by an overlapping cache memory address space of the two code segments occupied by the functions *foo5* and *sum_array*. The cache internal addresses resulting from the direct mapped mapping function in a 256 word instruction cache are visible in Table 3.14. As *foo5* is excessively called in *sum_array*, using a direct mapped placement strategy will lead to a noticeable higher number of conflict misses. The difference in miss counts between a fully associative and a 2-way set associative cache controller executing a FIFO replacement policy together with a direct mapped cache controller are visible in Figure 3.5. All cache controllers have been implemented with a block size of 4 and a block count of 64 (512 Bytes). Table 3.14 lists the corresponding miss rates.

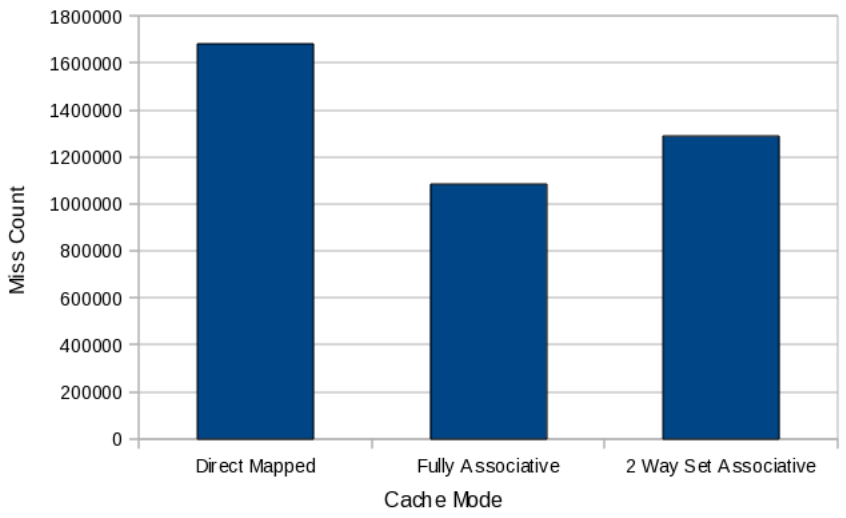


Figure 3.5: Direct Mapped vs. Fully Associative vs. 2-Way Set Associative (INST.).

Indeed, the different heights of the bars imply a clear defeat for the direct mapped cache controller but the difference is not that huge as it could be in the worst case where two instructions

Cache Mode	**Miss Count**	**Miss Rate**
Direct Mapped	1682891	0.083
Fully Associative	1083729	0.053
2 Way Set Associative	1287472	0.063

Table 3.14: Direct Mapped vs. Fully Associative vs. 2-Way Set Associative (INST.).

that are mapped exactly to the same cache block are called in an alternating sequence (*trashing*). Usually, any cache controller seems to be perfectly adapted to be used for instruction memories as instructions are typically fetched in an iterative way. Regrettably, due to the missing ability to decide for alternative storage blocks, the direct mapped cache controller will suffer from significantly higher miss rates in situations just like the one in the presented example. Concentrating on the miss count, the fully associative placement strategy clearly would be the best choice. Nevertheless, focusing on the miss rate, the benefit does not seem to be that enormous like it appears to be on the first look. The question is: Are the additional needed hardware resources justifiable when implementing an fully associative instead of a direct mapped cache controller? Indeed, an answer can only be given when knowing the exact hardware overhead. Nevertheless when examining the miss rate of the 2-way set associative cache controller, it is clearly visible that even a small improvement, i.e., allowing an instruction to be placed on two positions instead of one significantly improves the situation. In addition, implementing a cache controller with such a low grade of associativity is not that complex compared with a fully associative cache controller. The 2-way set associative placement policy can easily be deduced from the direct mapped by just splitting the cache memory into two direct mapped memories. Now the cache controller can decide in which of these memories to place an incoming instruction, which explains the evident performance increase. A justification concerning the additional cost of the hardware requirements will be given in the next chapter. To sum up, the decision of which of these cache controllers to be used for this specific application is moved to the hardware layer.
Contrary to the just mentioned problem concerning too small programs, a bigger data cache can easily be tested in an excessive way. E.g., four arrays $a_n[ARRAY\_SIZE]$ with $n \in [1,4]$ and $ARRAY\_SIZE = 8000$ cover a big area of the data memory space of the *SPEAR2* architecture. Although the external memory would offer far more memory space than the internal synthesized one, the size is left unchanged in all different modes in order to provide equal system conditions for all simulation runs. Anyway, it is not that important to cover the complete data memory space as all data words are positioned modulo the cache size. Accessing an address space that is a multiple of the cache size is perfectly sufficient to investigate all performance issues. Figure 3.6 compares the different numbers of misses. The corresponding miss rates are

stated in Table 3.15.

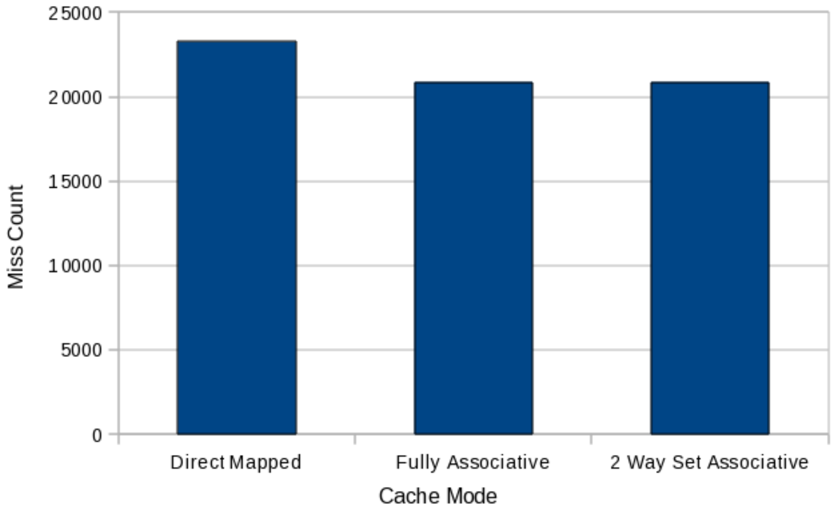


Figure 3.6: Direct Mapped vs. Fully Associative vs. 2-Way Set Associative (DATA).

Focusing on the results, the difference in the miss rates for all three cache controllers is minimal. Therefore, it is not necessary to perform further investigations concerning the hardware requirements as it is reasonable to decide for the direct mapped caching strategy in this case.

3.3.3 Cache Benchmark 2

Concerning the investigation of the data cache performance, the above used benchmark program provides a somehow unrealistic scenario as programs tend to access the data memory in a distributed manner and not iteratively. A more efficient approach, e.g., is therefore to use a recursive implementation of the *Mergesort* algorithm (*cache_benchmark2.c*) equipped with some modifications to guarantee a distributed access over the complete data memory space. Mergesort is a sorting algorithm following the divide and conquer principle. An array of numbers is sorted

Cache Mode	**Miss Count**	**Miss Rate**
Direct Mapped	23254	0.050
Fully Associative	20870	0.048
2 Way Set Associative	20869	0.048

Table 3.15: Direct Mapped vs. Fully Associative vs. 2-Way Set Associative (DATA).

by iteratively and recursively dividing the array into sub-arrays that are small enough for the subsequent reordering of the included numbers when they are again merged together. The here stated algorithm implements an *out-of-place execution*, i.e., new arrays must be generated in order to provide the data memory space for the sub arrays which enhances the desired distributed memory access. The locality of time and space is again disturbed by executing distributed memory access in between the actual algorithm and by again executing chains of *NOP* instructions. For more details concerning the source code, please refer to the appendix.
Figure 3.7 introduces the miss rates of the three cache controllers executing *cache_benchmark2.c*. Both, the fully associative and the 2-way set associative cache implement a FIFO replacement policy. An improved version of the fully associative cache controller using LRU as replacement policy has been simulated in order to illustrate the benefits of using a more sophisticated replacement strategy than FIFO. The block count has been chosen to be 64, the block size is 4 (512 Bytes). Just as expected, the direct mapped cache controller exhibits a far more higher miss rate when performing distributed memory accesses. In addition, it seems to be surprising that the 2-way set associative cache performs almost equal good as the fully associative for this particular application. Indeed it can be concluded that the fully associative cache controller can only exhibit its full efficiency when using LRU as replacement policy. As the fully associative cache controller is known for its high hardware requirements (we will justify this statement in the next chapter), but its performance advantage is minimal for this application when implementing a FIFO replacement strategy, the next benchmarks will only concentrate on the comparison of the direct mapped with the 2-way set associative caching strategy. The corresponding miss rates are presented in table 3.16.
Figure 3.8 compares the performance issues of the direct mapped caching strategy versus the 2-way set associative one focusing on different cache sizes. The results obtained from this simulation shows that is also important to consider the overall cache size beside the used caching strategy when deciding for an appropriate cache controller. Indeed, the miss count of the direct mapped cache controller approaches that of the 2-way set associative controller with a rising cache capacity. Ergo, if only small cache memories are available, deciding for the 2-way set associative cache controller would be the better approach. The higher the capacity gets, the more reasonable becomes the usage of the direct mapped cache controller. Again, the finding of the hopefully right decision is moved to the hardware layer. At least we can safely exclude the fully associative cache controller as its performance advantage is almost negligible compared to that of the 2-way set associative controller.

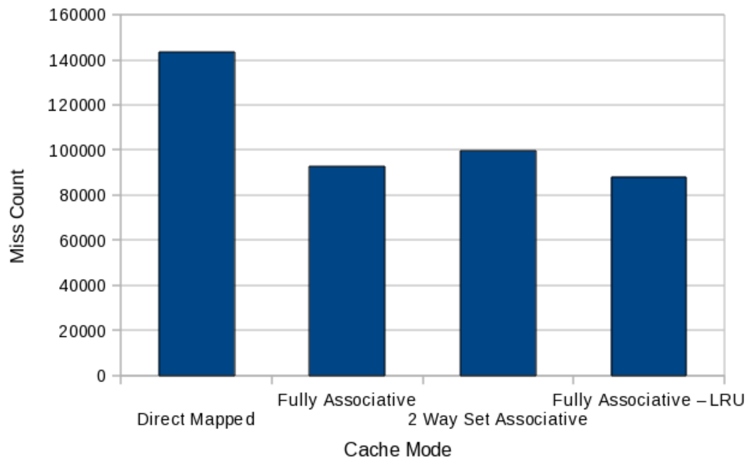


Figure 3.7: Direct Mapped vs. Fully Associative vs. 2-Way Set Associative (DATA).

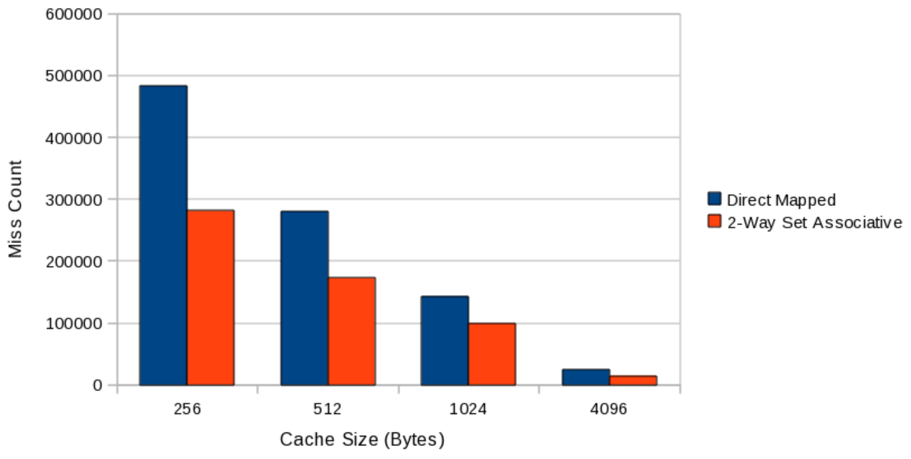


Figure 3.8: Direct Mapped vs. 2-Way Set Associative (Cache Size).

3.4 Conclusion

To sum up, as the optimal caching strategy for a given application is strongly dependent on the localities of time and space exhibited by the program, it is important to perform simulations similar like the way it has been explained in this chapter, i.e, testing all possible cache controllers acting as instruction and data caches with a subsequent investigation of their behavior

Cache Mode	**Miss Count**	**Miss Rate**
Direct Mapped	143514	0.021
Fully Associative	92688	0.013
2 Way Set Associative	99592	0.014
Fully Associative (LRU)	88153	0.012

Table 3.16: Direct Mapped vs. Fully Associative vs. 2-Way Set Associative (DATA).

for different memory capacities. After having completed this task the next step is to compare the hardware relevant aspects of the cache controllers.

CHAPTER 4

Implementation

"A good idea is about ten percent and implementation and hard work and luck is 90 percent." - Guy Kawasaki (Former Apple Marketing Expert).

This final chapter is going to cover the most important design details related to the implementation of the new configurable memory architecture for the *SPEAR2* softcore taking advantage of three different selectable cache controllers. Before explaining the necessary adaptations on the original design and the available operational modes for this novel memory layer, the first section is going to provide a brief overview on cache controller implementations from two well known processor architectures.

4.1 Related Work

4.1.1 The 603e PowerPC^{TM}

PowerPC^{TM}(**P**erformance **O**ptimization **W**ith **E**nhanced **R**ISC – **P**erformance **C**omputing) is a RISC architecture developed by *Apple*, *IBM* and *Motorola* (also known as the AIM consortium) in the year 1991 [23] [1]. The 603e, a second generation PowerPC^{TM}architecture, is a 32-bit

high performance superscalar[1] design.
The L1 data and instruction cache both provide a storage capacity of 16 Kbyte and are organized in a four-way set associative way, being built of 128 sets of four blocks, each of them including one address tag and maximal 32 bytes of data. In order to be able to manage cache coherency, every block in the instruction cache is also equipped with a valid bit whereas every block in the data cache additionally includes two state bits that implement a three state MEI (modified, exclusive, invalid) protocol. These two state bits in combination with a special on-chip snooping logic guarantee that the data cache is always in a coherent state. The extra snooping mechanism scans the bus interface and the memory system to which the cache is connected for referenced addresses during the program execution and compares them with addresses stored inside the cache tag memory. On a hit, corresponding actions are taken by the cache controller to maintain the data coherency. As the instruction cache is not snooped by this special logic, coherency must be maintained by software taking usage of the valid bit that is invalidated by the hardware if the corresponding block is altered.
The mapping from memory addresses to cache internal positions is realized by so called *chunks*, that are also known as *cache pages*. A chunk has the same size as one way of the cache, that is 128 blocks $*$ 32 bytes $=$ 4 Kbyte. Since the 603e has an address space of 4 Gbyte, the memory needs to be subdivided into 1 million such chunks. The internal organization of data and instruction cache is illustrated in Figure 4.1.

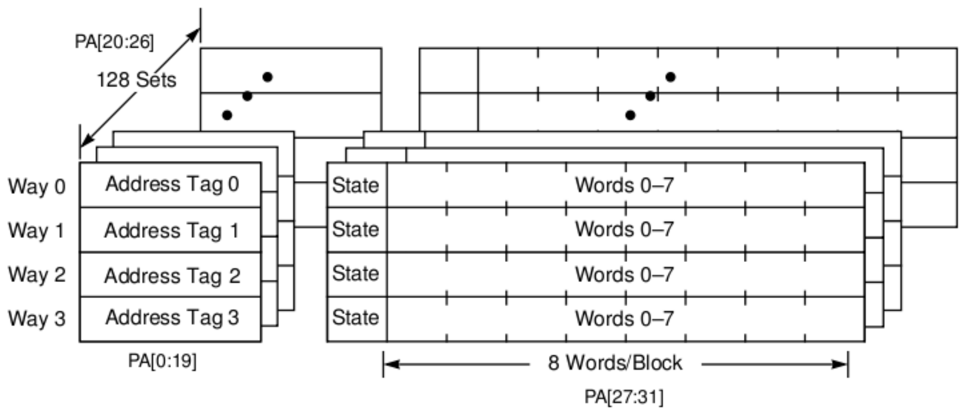


Figure 4.1: Internal cache organization of the 603e [1].

An index represents the relative position of a block inside a chunk and is determined by bits

[1]A superscalar processor is an architecture (single core) with redundant functional units that is able to execute two or more instructions simultaneously (instruction level parallelism) [25].

PA[20:26] of the memory address. As it can be seen in Figure 4.2, every block from the memory with the same index is assigned to a fixed corresponding set in which it can be placed in any of the four available ways. The appropriate chunk is identified by the vector PA[0:19] (tag address). The remaining five bits in PA[27:31] are used to select a word from the block.
In the case of a cache miss, data words are loaded from the main memory in 8 transfers of 32-bit or 4 transfers of 64-bit respectively. LRU is used as replacement policy. While the data cache is blocked until the transfer has completed due to a single ported interface, the instruction cache allows sequential fetching from other blocks while storing data at the same time after the load of the critical first double word of the previous load instruction has completed. This technique is referred to as *critical word first*. On the contrary, a cache controller implementing *early restart* will fetch the data words from the main memory in the normal order but will also restart the execution of the processor as soon as the requested data word has arrived.
Cache locking is available for both caches. Locked blocks will be updated once on a miss and will remain in the cache memory until the lock bit is cleared. Cacheability, write back policy and memory coherency can be configured at the block and page level by manipulating the so called *WIMG* attributes. More precisely, *WIMG* is an acronym for the following four properties which can be enabled/disabled by altering the corresponding bits in a special register:

Write-Through (W) Setting this bit to '1', the modification of data in the cache will also trigger an update of the original data in the main memory. Otherwise, the cache will be configured to use a write-back strategy, forcing the controller only to update the modified data in the main memory when it is required (e.g., in case of a cache replacement operation).

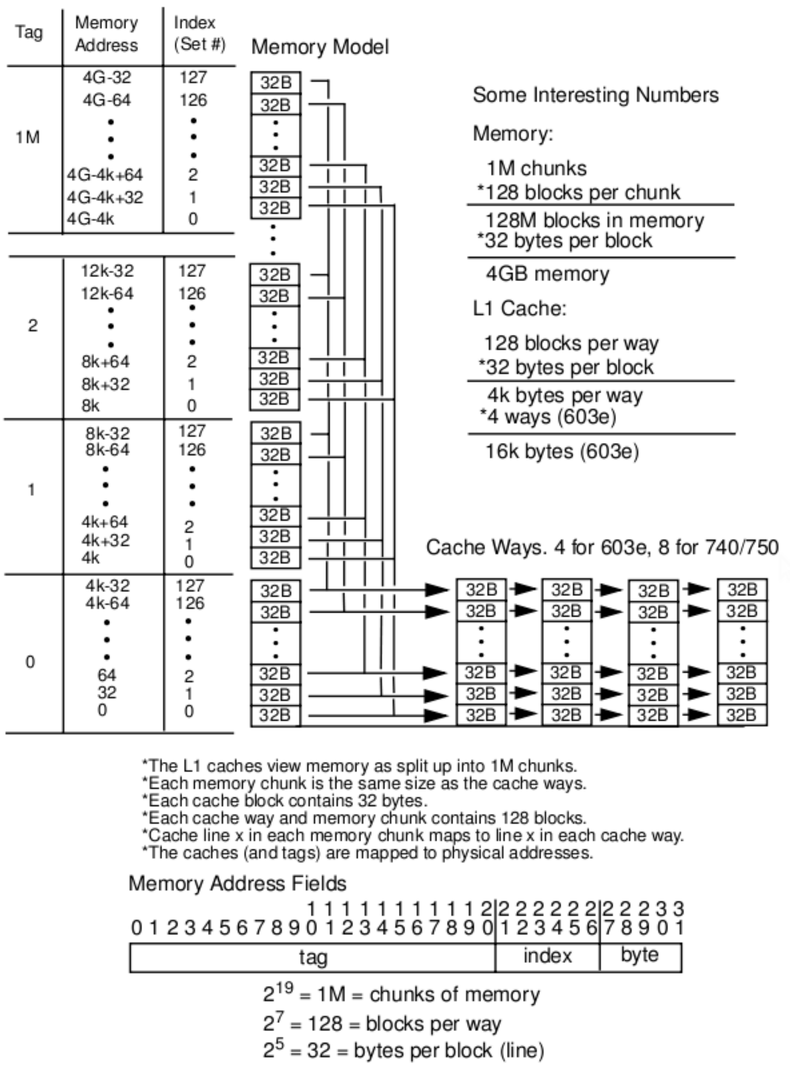
Caching-Inhibited (I) By enabling this attribute, all data words will be directly referenced in the main memory therefore bypassing the cache.

Memory Coherency (M) If **M** is equal to '1', every access to this position will be considered to be global, thus enforcing data coherency by triggering other snooping devices holding the same but altered data to update the corresponding memory location.

Guarded Memory (G) Protects the system from undesired out-of-order or prefetch memory operations.

While the **G** attribute prevents the system from machine exceptions[2], the **M** attribute forces all copies of an addressed memory location to be in a coherent state. **W** and **I** configure the processor how to use its cache.

[2]Prefetch operations to memory locations which are not occupied by the program might lead to exceptions whereas an out-of-order access to a peripheral might return undesired results when being accessed in this way.



*The L1 caches view memory as split up into 1M chunks.
*Each memory chunk is the same size as the cache ways.
*Each cache block contains 32 bytes.
*Each cache way and memory chunk contains 128 blocks.
*Cache line x in each memory chunk maps to line x in each cache way.
*The caches (and tags) are mapped to physical addresses.

2^{19} = 1M = chunks of memory
2^{7} = 128 = blocks per way
2^{5} = 32 = bytes per block (line)

Figure 4.2: Memory to cache mapping of the 603e [1].

4.1.2 The LEON2 Processor

The LEON2 processor is a 32-bit SPARC V8 RISC architecture that has been designed to be especially suited for embedded applications [2]. SPARC stands for **S**calable **P**rocessor **A**rchitecture and was developed by Sun Microsystems in the year 1987 [24]. LEON2 is either

available as synthesizable VHDL softcore design that provides just like the *SPEAR2* core an easy composition with additional extension modules through its *AMBA* AHB/APB bus interface or alternatively as radiation hard ASIC chip[3].
Instruction and data cache are implemented in a Harvard architecture, thus providing simultaneous memory accesses via separated data and address buses. Both cache controllers can be either configured to execute a direct mapped or a set associative placement strategy. The grade of associativity can be switched between 2 and 4. When using the set associative placement policy it is possible to choose from the following three different replacement strategies: least recently used (LRU), least recently replaced (LRR) and pseudo random (RAND). Beside the placement and replacement policy, the user can also configure the set- and the block size, i.e. for the set size it is possible to select a value from the range of 1 to 64 Kbyte whereas the block size can be chosen from an interval of 8 up to 32 bytes of data.
On a miss on the instruction cache, the requested instruction is loaded together with the complete block from the main memory and stored into the corresponding cache location. When activating the *instruction burst fetch mode* in a special cache configuration register, all instructions from the loaded cache block are forwarded to the integer unit if possible[4], thus reducing the number of penalty cycles. Figure 4.3 shows the address format for the instruction cache controller.

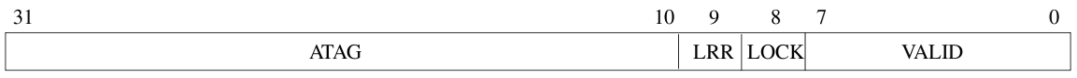


Figure 4.3: LEON2 instruction cache address format [2].

The *ATAG* field includes the tag address which is compared with the corresponding entry in the tag memory in order to determine if the requested data word is present in the cache or not. Cache locking is configurable by setting the bit in the *LOCK* field. While the *LRR* bit stores the replacement history of the targeted cache block, the *VALID* field determines for all Bytes residing in the block if they are valid or invalid where one bit of the field represents the status of one sub block that includes four Bytes.
The address format of the data cache controller exactly equals to the address format of the instruction cache. Quite the opposite, the data cache controller always fetches only 4 bytes of data from the main memory in the case of a cache miss. The rest of the original block is invalidated in order to maintain the cache coherency. Write through with no allocate on a write miss is used as write back strategy.

[3]http://atmel.com/dyn/products/product_card.asp?part_id=3178
[4] This is only possible if there are no branches to be executed.

4.2 Adaptations for The New Memory Layer

Although many of the advanced features (e.g., the configurable write-back/through policy on the block and page level) provided by the PowerPC^{TM}architecture seem to be out of reach for the *SPEAR2* processor at the moment, it is certainly desirable to own an architecture that implements a configurable memory layer equipped with a generic cache interface in order to be able to efficiently design more sophisticated cache controllers in the future. Therefore, the aim has been to implement a cache design that can at least be compared with the cache controller of the *LEON2* architecture concerning aspects like a configurable placement policy and a variable number of blocks and sets. The following description of the novel memory layer for the *SPEAR2* processor implemented in VHDL shows that this aim has in fact been perfectly reached.
Unfortunately, certainly one of the weak points of the original architecture is that it has not been implemented in a clean modular way. As a result, the *SPEAR2* processor did not include an explicit memory layer. Instruction and data memory were directly instanced beside all other modules in the *spear.vhd* top layer. Figure 4.4 represents the simplified block diagram of the original top entity *spear.vhd*.

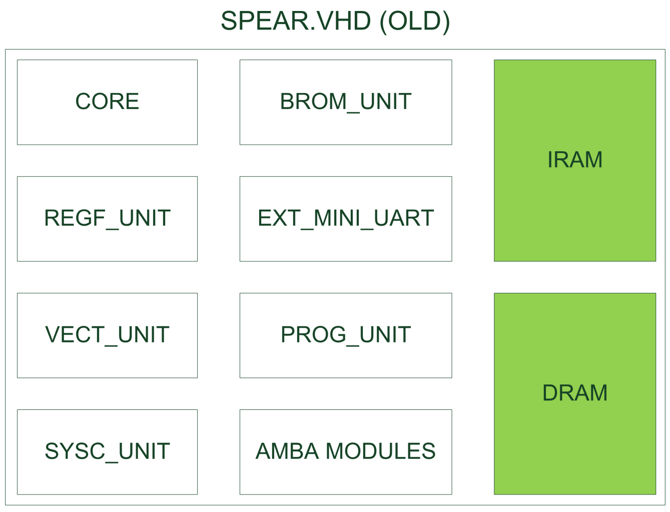


Figure 4.4: Old *spear.vhd* block diagram.

VHDL modules participating in the assembling of the memory were *iram.vhd* for the instruction memory and *byteram.vhd* and *dram.vhd* for the data memory. The data memory was constructed by generating four instances of the *byteram.vhd* module in order to yield the preferred byte access for write operations.

4.2.1 Adaptation 1 - Implementation of an Explicit Memory Layer

Therefore, in order to provide an easy configurable memory architecture allowing the user to switch between the different available versions using some specific VHDL generics of the top level entity, the first optimization to be applied has been the introduction of an explicit memory layer. The block diagram of the *spear.vhd* with the added memory top level replacing the original dram and iram unit is visualized in Figure 4.5. Its exact functionality, ports and internal modules as well are going to be explained in detail in the subsequent subsection.

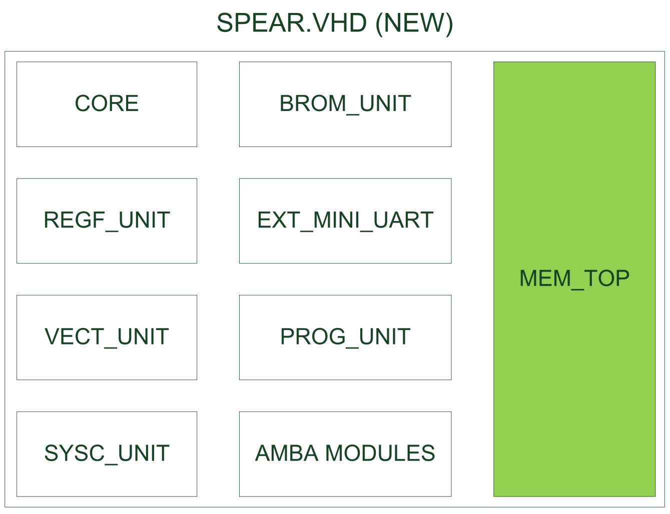


Figure 4.5: New *spear.vhd* block diagram.

4.2.2 Adaptation 2 - Implementation of an Explicit Instruction and Data Read Signal

One of the most essential optimizations has certainly been the implementation of an explicit read signal. Its introduction has been important for the following reason: In the old *SPEAR2* memory design, the memory read access always remained active but the data words on the other hand only were taken over in the case of an active read request in the pipeline. As all data words are immediately available in the following cycle when accessing an internal memory, this permanent read enable strategy did not affect the performance of the processor in a negative way. On the other hand, performing read operations on an external memory which suffers from a latency of more than just one cycle, the processor will waste most of its computational cycles waiting for data words that will not be taken over if there has not been an explicit read request from the pipeline. This scenario would especially be a problem at the boot phase of the program execution where the source code is received from the serial interface and stored inside the instruction memory. Since this task is performed by the boot loader located in the boot memory, addressed by the upper address space of the instruction memory, the programmer would never be able to write to the external memory since the instruction memory controller would always send read requests to the external RAM. This dilemma made the introduction of a read signals unavoidable. As a result, the first step to be taken beside the extension of the *core.vhd* interface (see Table 4.1) was to define the point in time where to enable the instruction memory by sending the appropriate read signal. Since the address space of the programmer is mapped to the upper end of the instruction memory address space, the condition in the original implementation to choose between the data words from the programmer port or the incoming data from the instruction memory port was based on the value of the most significant bits of the program counter. For a value not equal to 0, the instructions were taken from the programmer input. For a value equal to 0, the code from the instruction memory was forwarded to the decoding stage. In order to activate the instruction memory at the right time, it has been necessary to introduce the following additional check of the program counter value into *core.vhd*.

Port	**Type**	**Direction**	**Semantics**
irami_en	std_ulogic	OUT	Instruction memory read enable.

Table 4.1: Port extension of the *core* component.

```
-- mbirner : added read enable trigger - if highest bits (boot rom address space)
-- are 0 -> enable instruction memory !
-----------------------------------------------------------------
-- check if instructions are taken from bootrom or instruction-RAM --
-----------------------------------------------------------------
if v.f.pcnt(WORD_W-1 downto CONF.bootrom_base_address) = INST_ADDR_NULL then
  irami_ren <= '1';
end if;
```

Exactly the same approach was necessary to be applied on the data ram as the *dramsel* signal was only used as write enable signal in combination with *write_en* and *byte_enable*. Read accesses again were permanently active. Enabling the read access only when requested by the processor is achieved by splitting the *dramsel* in the *spear.vhd* architecture into one *dram_write_en* signal and one *dram_read_en* signal.

```
-- mbirner : added explicit read and write signal
-- dramsel <= coreo_extwr or coreo_memen; -> old ram select signal
dram_write_en <= coreo_extwr;
dram_read_en <= coreo_memen;
```

Beside the introduction of these two enable signals, also the data ram address signal was needed to be adapted. As the *SPEAR2* data memory is addressed byte wise, the least two significant bits from the memory address were already truncated in the top level *spear.vhd*. This task has been moved into the memory layer. Therefore the address vector has been extended to its original size.

```
-- drami_addr <= coreo_extaddr(CONF.data_ram_size-1 downto 2); -> old addressing
-- mbirner : added dram signals
dram_addr <= coreo_extaddr(CONF.data_ram_size-1 downto 0);
```

Table 4.2 lists the added signals in *spear.vhd*.

Signal	**Type**	**Semantics**
dram_read_en	std_ulogic	Data memory read enable.
dram_write_en	std_ulogic	Data memory write enable.
dram_addr	std_logic_vector[CONF.word_size-1:0]	Data memory address.

Table 4.2: Signal extension of the *spear* component.

4.2.3 Adaption 3 - Extension of Generics Declaration of the *SPEAR2* Entity (*spear.vhd*)

As the new memory layer is fully configurable by selecting the memory mode, the caching strategy and the block count and block width of both data and instruction cache, it must be possible to set all these parameters when generating an instance of the *SPEAR2* processor. In this sense, the *CONF* type in the generics declaration of the top entity has been extended by the generics listed in Table 4.3.
The user can choose from three different memory modes to be used within the *SPEAR2* core. By setting *mem_mode* to *SPEAR2_INT_MEM*, the original design storing data and instructions in a FPGA-internal RAM will be synthesized. All other additional generics will be ignored in this case. Interfacing the external memory without a cache controller is enabled by *SPEAR2_EXT_MEM*. Taking advantage of caching mechanisms becomes available with setting the generic value to *SPEAR2_CACHE_MEM*. The caching strategy is selected by the entry in the *cache_mode* field. The three available policies are listed in Table 4.4. Block width and block count can be freely chosen with the only restriction that the block count must be equal to a power of 2. The *reg_mode* generic determines if an additional register is included into the data and address path resulting in a higher feasible storage capacity for a reason that is going to be explained soon.

Generic	Semantics
mem_mode	Selects the kind of memory to be used.
data_cache_mode	Selects the caching strategy for the data cache.
instruction_cache_mode	Selects the caching strategy for the instruction cache.
reg_mode	Determines if some extra register should be included into the data and address path of both caches.
data_cache_block_width	Determines the block width of the data cache (number of bits in block field).
data_cache_block_count	Determines the number of blocks residing inside the data cache.
instruction_cache_block_width	Determines the block width of the instruction cache (number of bits in block field).
instruction_cache_block_count	Determines the number of blocks residing inside the instruction cache.

Table 4.3: Extended generics in the *spear* component.

Cache Mode	Semantics
SPEAR2_DIRECT_MAPPED	Direct mapped cache.
SPEAR2_FULLY_ASSOCIATIVE	Fully associative cache controller.
SPEAR2_SET_ASSOCIATIVE	2-way set associative cache controller.

Table 4.4: Available caching strategies for *SPEAR2*.

4.2.4 Adaption 4 - Extension of the Port Declaration of the *SPEAR2* Entity (*spear.vhd*)

In order to equip the core with an interface to the external RAM, the *SPEAR2* entity description has been extended with two new records: *ext_ram_in_type* and *ext_ram_out_type*. Table 4.5 and Table 4.6 introduce the reader to the added ports. Please note that these two ports provide a generic interface which allows for connecting the *SPEAR2* processor to any type of external memory.

The *finished* signal informs the memory layer that the last operation on the external memory has completed. In the case of a read access, the requested information can then be read in from the *data* port.

On the other hand, the *wr_rd_en* signal switches between read and write requests. A logical ’0’

Port	Type	Semantics
data	std_logic_vector[31:0]	External RAM data in.
finished	std_ulogic	External RAM finished signal.

Table 4.5: Input port description of the *ext_ram_in_type* record.

Port	Type	Semantics
data	std_logic_vector[31:0]	External RAM data out.
addr	std_logic_vector[31:0]	External RAM address.
wr_rd_en	std_ulogic	External RAM read or write select.
start	std_ulogic	External RAM transfer start.
trans_size	std_logic_vector[2:0]	External RAM transfer size.
burst	std_ulogic	External RAM burst mode enable.

Table 4.6: Output port description of the *ext_ram_out_type* record.

triggers a read operation while a logical '1' requests a write access. The selected operation is started by setting the output port *start* to '1'. Burst operations save computation cycles when performing iterative read accesses to the external memory. This mode can be activated by the writing of a '1' to *burst*. The length of the burst transfer is determined by the block size of the requesting cache controller. The width of the transfer is set by *trans_size*. Table 4.7 lists all possible values. During an operation on the external RAM, all output signals will remain active until the finished transmission is acknowledged with the *finished* signal. Figure 4.6 illustrates a typical access example where a burst read operation of the length 4 is performed on an external memory.

4.3 Implementation Details of the New Memory Layer

The previous subsection has introduced the interface of the memory layer to the processor without taking care of the underlying communication mechanism with the SDRAM module. This particular important information will be given in the following, before providing a detailed description of the internal structure of the novel memory layer.
The SDRAM modules[5] are controlled by an *AMBA* SDRAM controller provided by the *GRLIB* that has already been included into the *SPEAR2* design by [14]. As it can be concluded from the name of the VHDL design, this controller is equipped with an *AMBA* AHB bus interface, acting on the bus as a slave. In this sense, the external RAM interface of the *SPEAR2* core introduced in the last subsection must be routed to an *AMBA* AHB bus master in order to successfully perform

[5]Two Samsung SODIMM M464S3254ETS-L7A (CL3 PC133 256MB) chips

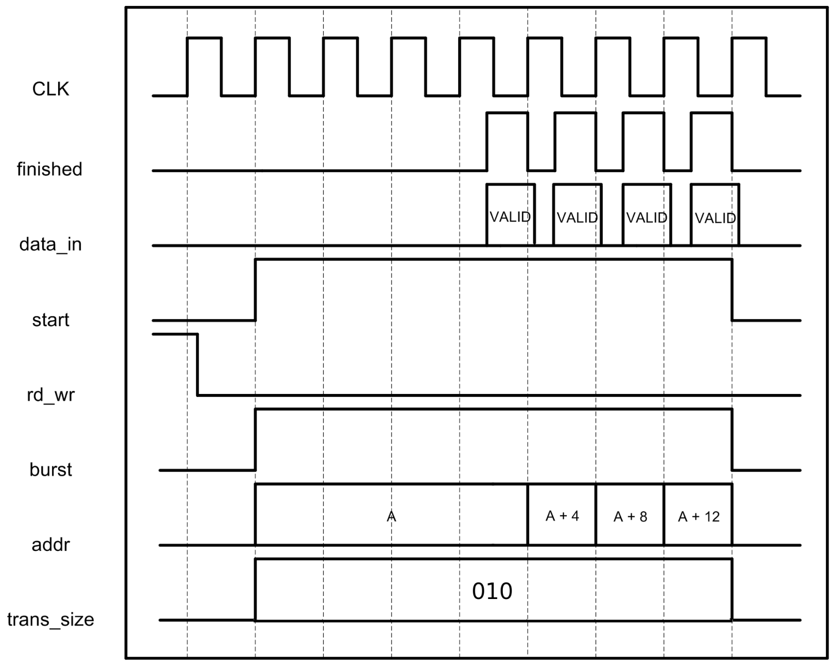


Figure 4.6: Burst access example (read).

Value	**Semantic**
“000“	Byte access.
”001“	16-bit access (word).
”010“	32-bit access (double word).

Table 4.7: External memory transfer width.

read and write operations on the external memory through the SDRAM controller. Such an AHB master module is also already provided by the *GRLIB*. Figure 4.7 shows a simplified block diagram of the processor interfacing the external memory in the just described way. For more information on the *AMBA* AHB bus and on the actions that are necessary to be taken by a master and a slave for a successful bus operation, the interested reader is referred to [4]. This thesis will continue with a description on the internal activities of the memory layer for every memory access operation.
As already stated, the new memory top layer for the *SPEAR2* architecture is fully configurable by selecting three different modes of memories to be used. Depending on the actual selected value for the generic, the internal structure of the memory layer will change, resulting in a different behavior during read and write operations. Before focusing on the single modes of

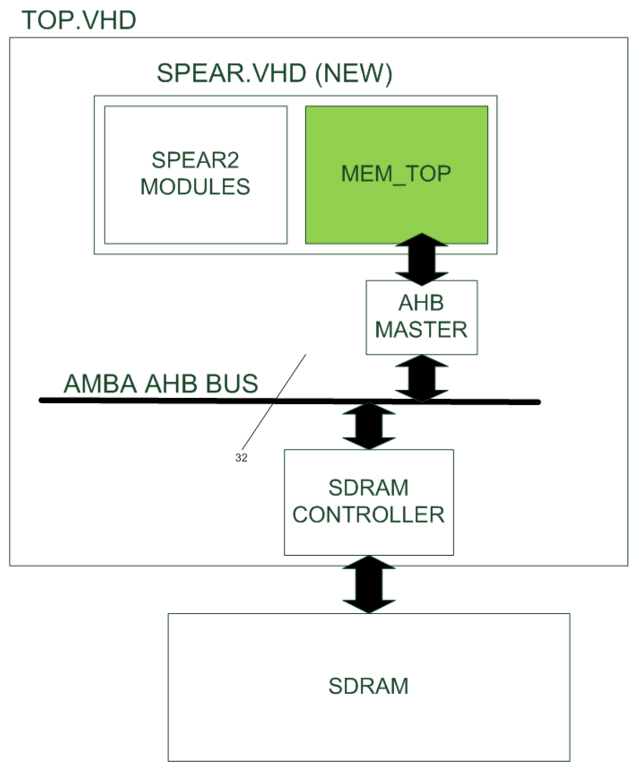


Figure 4.7: Connection of the memory layer with the SDRAM module.

the memory unit, Table 4.8 list the port description. The ports of the external memory interface have been left out since they are the same as in *spear.vhd* and have already been explained in Table 4.5 and Table 4.6. Figure 4.8 illustrates the embedding of the new memory layer into the *SPEAR2* processor. Although the description of the three memory modes will start with the internal memory configuration, this first block diagram shows a memory layer equipped with cache controllers accessing the external memory.
Since there exists two address ports interfacing the instruction memory, the memory top entity must decide which address to forward depending on the selected operation.

```
------------------------------------------
-- select between read or write address --
------------------------------------------
if instruction_ram_wr_en = '1' then
  instruction_ram_addr_sig <= instruction_ram_waddr_sig;
elsif instruction_ram_rd_en_sig = '1' then
  instruction_ram_addr_sig <= instruction_ram_raddr_sig;
else
  instruction_ram_addr_sig <= (others => '0');
```

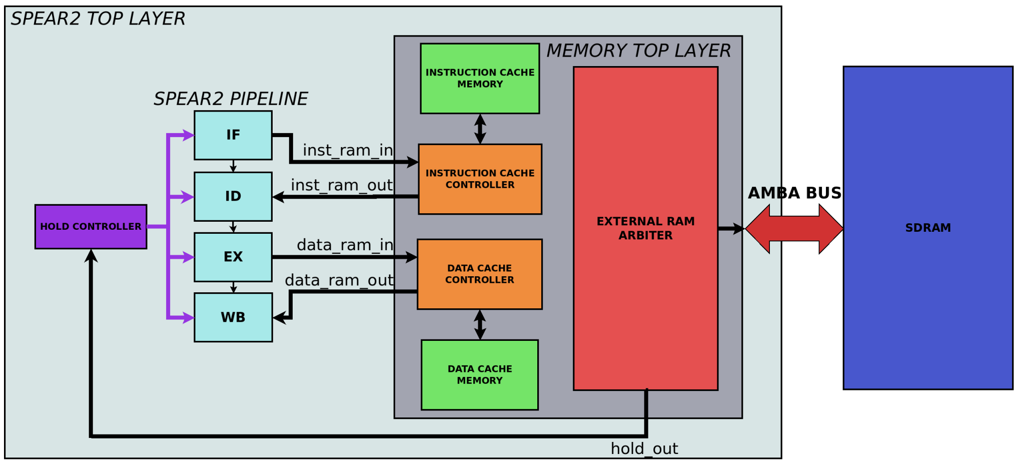


Figure 4.8: Embedding of the novel memory layer into the *SPEAR2* entity.

```
end if;
```

All write and read accesses to the memory are controlled via the *hold* port. By setting it to *HOLD_ACTIVE*, all requests will be blocked until releasing the pipeline again.

```
if hold_in = HOLD_ACTIVE then
  ------------------------
  -- block all requests --
  ------------------------
  instruction_ram_rd_en_sig <= '0';
  instruction_ram_wr_en_sig <= '0';
  data_ram_rd_en_sig <= '0';
  data_ram_wr_en_sig <= '0';
else
  ------------------------------------
  -- else execute incoming requests --
  ------------------------------------
  instruction_ram_rd_en_sig <= instruction_ram_rd_en;
  instruction_ram_wr_en_sig <= prog_prupdate_sig;
  data_ram_rd_en_sig <= data_ram_rd_en;
  data_ram_wr_en_sig <= data_ram_wr_en;
end if;
```

4.3.1 Internal Memory

The original *SPEAR2* memory architecture can be built by setting the *mem_mode* generic of the *SPEAR2* component to *SPEAR2_INT_MEM*. The corresponding block diagram of the entity *mem_top* resulting from this configuration can be seen in Figure 4.9.

Port	**Type**	**Direction**	**Semantics**
clk	std_ulogic	IN	Clock input.
reset	std_ulogic	IN	Reset input.
hold_in	std_ulogic	IN	Hold input.
data_ram_data_in	std_logic_vector[31:0]	IN	Data memory data in.
data_ram_data_out	std_logic_vector[31:0]	OUT	Data memory data out.
data_ram_addr	std_logic_vector[data_size-1:0]	IN	Data memory address.
data_ram_rd_en	std_ulogic	IN	Data memory read enable.
data_ram_wr_en	std_ulogic	IN	Data memory write enable.
data_ram_byte_en	std_logic_vector[3:0]	OUT	Data memory byte enable.
instruction_ram_data_out	std_logic_vector[15:0]	OUT	Instruction memory data out.
instruction_ram_data_in	std_logic_vector[15:0]	IN	Instruction memory data in.
instruction_ram_raddr	std_logic_vector[instr_size-1:0]	IN	Instruction memory read address.
instruction_ram_waddr	std_logic_vector[instr_size-1:0]	IN	Instruction memory write address.
instruction_ram_rd_en	std_ulogic	IN	Instruction memory read enable.
instruction_ram_wr_en	std_ulogic	IN	Instruction memory write enable.
hold_out	std_ulogic	OUT	Hold output.

Table 4.8: Port description of the *mem_top* component.

int_mem.vhd

Although the basic structure did not change with the new memory layer, the source code has been completely rewritten in order to perform the following optimizations: The module *int_mem* serves now as new basic memory block for the FPGA-internal instruction and data memory. In contrast to the original design, there is no further need for an explicit separation of the instruction memory module *iram* from the data memory module *byteram* as the new design merges the functionality of both memories into one single generic entity. In this sense, some essential modifications have been applied to allow the combination of both entities into the *int_mem* module.
As it is visible in the port description in Table 4.10, the read and write clock signals have both been replaced by one general clock signal. The same holds for the read and the write address ports. Maybe there was some intention to provide simultaneous read and write accesses to the data memory but this case can be safely excluded resulting from the following fact: As the *SPEAR2* processor is a non-superscalar load/store architecture it is only possible to perform one single read or write access on the data memory but not both actions at the same time. The generics of this new module are listed in Table 4.9.

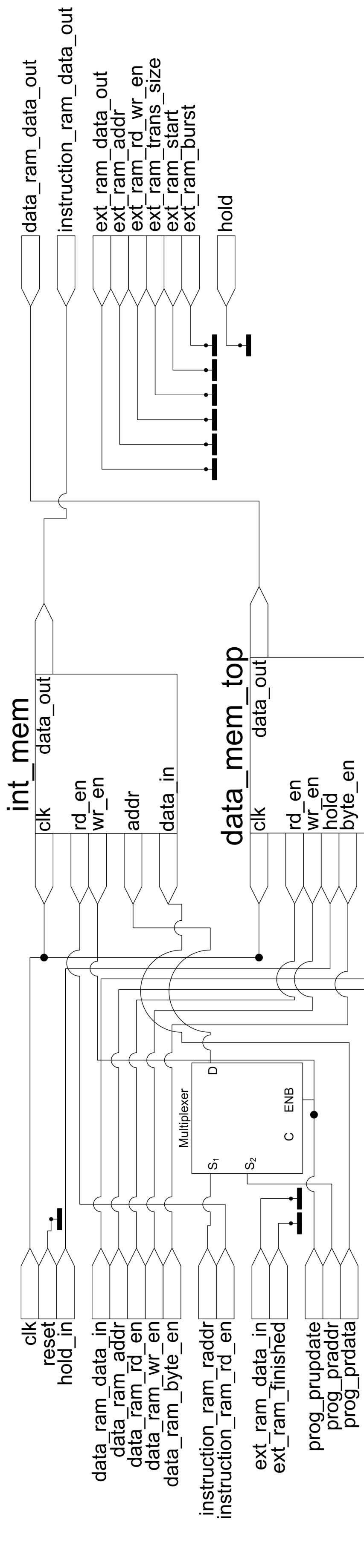


Figure 4.9: *SPEAR2* internal memory configuration.

Generic	**Semantics**
DATA_WIDTH	Defines width of data bus.
ADDRESS_WIDTH	Defines width of address bus. Determines size of memory.

Table 4.9: Generics of the *int_mem* component.

Port	**Type**	**Direction**	**Semantics**
clk	std_ulogic	IN	Clock input.
rd_en	std_ulogic	IN	Memory read enable.
wr_en	std_ulogic	IN	Memory write enable.
addr	std_logic_vector[adddr_width-1:0]	IN	Memory Address.
data_in	std_logic_vector[data_width-1:0]	IN	Data in.
data_out	std_logic_vector[data_width-1:0]	OUT	Data out.

Table 4.10: Port description of the *int_mem* component.

Furthermore, since the handling of the hold mechanism has been transferred to the top level of the memory layer, the hold (*iram.vhd*) and the enable signal (*byteram.vhd*) of the original design became useless and have therefore been removed from the port description of the new design. By the way, these two signals had a completely identical functionality but were unfortunately named in two different ways, i.e, the enable signal in the *iram.vhd* unit was generated internally by the inverted hold signal whereas the *byteram.vhd* unit directly used an incoming enable signal which again was generated on layer above (*dram.vhd*) by inverting the hold signal. This port is now replaced by an explicit read enable signal which was missing in the original design.

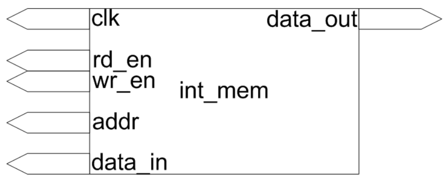


Figure 4.10: Internal Memory block diagram.

data_mem_top.vhd

Similar to the original *SPEAR2* design, *data_mem_top* implements the data memory by generating four single memory blocks, each of them providing the ability to store bytes instead of instancing one single memory module that can store double words. Otherwise it would not be possible to offer the required byte access for write operations. The selection of the bytes to be written is performed by the incoming byte enable signal *byte_en*. Read accesses on the other hand always return the complete 32-bit data word. The extraction of the requested information from the complete double word is done by a special logic in the *SPEAR2* entity and is therefore not considered by the memory layer. The hold signal from the processor masks incoming read and write requests in the case of a stalled pipeline.
As the memory access in this module is performed byte-wise, the two least significant bits from the incoming memory address need to be truncated. The following listing shows the part of the VHDL code segment which is responsible for this address adaption and the partition of the incoming 32-bit data signal into four bytes that are transmitted to their corresponding byte memories.

```
---------------------------
-- assign incoming bytes --
---------------------------
mem0_data_in_sig <= data_in(7 downto 0);
mem1_data_in_sig <= data_in(15 downto 8);
mem2_data_in_sig <= data_in(23 downto 16);
mem3_data_in_sig <= data_in(31 downto 24);

-------------------------------------------------------------------------------
-- assign incoming address - kick 2 least significant bits - byte addressing --
-------------------------------------------------------------------------------
mem_addr_sig <= addr(ADDRESS_WIDTH - 1 downto 2);
```

The outgoing data signal must be connected in the same way in which the input signal has been partitioned.

```
------------------------
-- assign output data --
------------------------
data_out(7 downto 0)   <= mem0_data_out_sig;
data_out(15 downto 8)  <= mem1_data_out_sig;
data_out(23 downto 16) <= mem2_data_out_sig;
data_out(31 downto 24) <= mem3_data_out_sig;
```

The generics list is equal to that in Table 4.9. The same holds for the port description listed in Table 4.10. Additional ports are listed in Table 4.11.
Figure 4.11 shows the data memory top level implemented with four byte memory instances.

Port	Type	Direction	Semantics
hold	std_ulogic	IN	Hold in.
byte_en	std_logic_vector[3:0]	IN	Byte enable.

Table 4.11: Additional ports of the *data_mem_top* component.

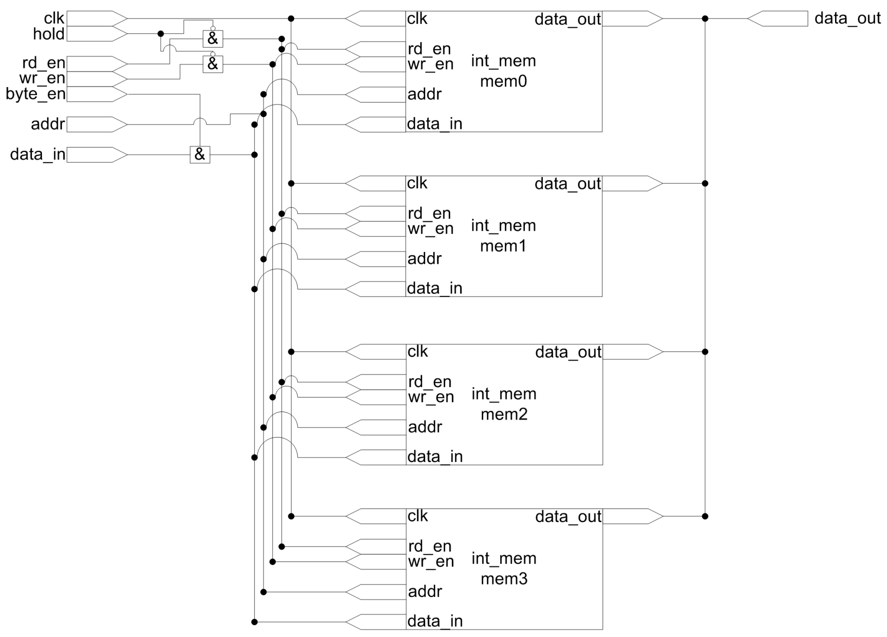


Figure 4.11: Data memory top level block diagram.

4.3.2 External Memory

Synthesizing the *SPEAR2* core accessing the external memory without taking advantage of caching mechanisms, the memory architecture will change from *Harvard* to *Van Neumann* due to the single ported interface to the SDRAM controller. Figure 4.12 introduces a simplified block diagram that illustrates this change. Due to the common link for accessing data and instructions and the high latencies of the external memory, the modified architecture will suffer from a dramatic performance decrease. However, the configuration presented in the next section will equip the memory layer with a cache controller, thus lowering the degradation.
In order to schedule simultaneous instruction and data accesses that appear whenever some operation on the data memory is requested by the processor additionally to the instruction read request[6], extra functional units are needed to handle these situations and to guarantee an ordered

[6]Of course, the instruction read requests are permanently active during the complete execution of a program.

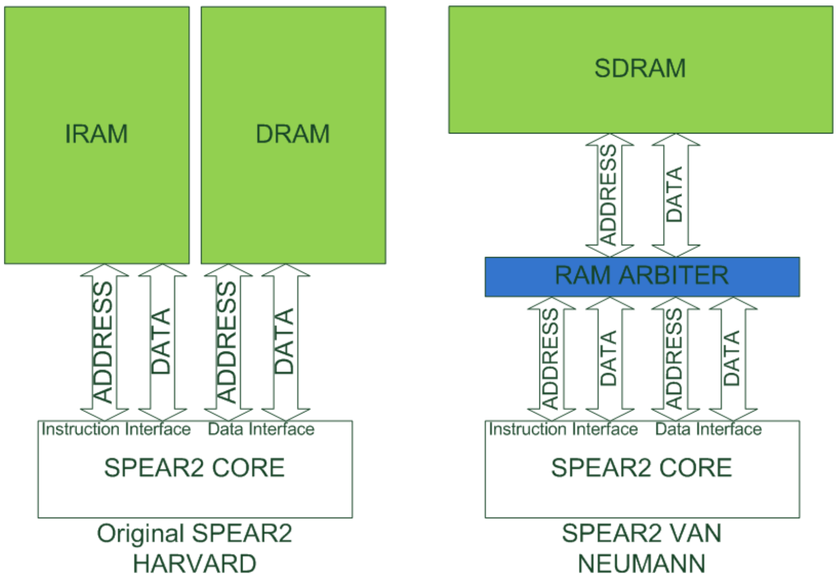


Figure 4.12: The change from Harvard to Van Neumann.

access to this bottleneck. This particular task is performed by the *RAM arbiter* which is responsible for the ordering of memory access requests from the *instruction memory controller* and the *data memory controller*. Both memory controllers have been implemented by one generic controller that is configurable to act either as data memory controller or as instruction memory controller. Figure 4.13 shows the block diagram of the *mem_top* component. Clearly visible, the two internal memory blocks have been replaced replaced by the two memory controllers and the RAM arbiter. Their detailed functionality will be explained in the following. Prior to this, the reader with less experience on the *SPEAR2* architecture must be introduced into the behavior of the pipeline during an operation on the external memory.

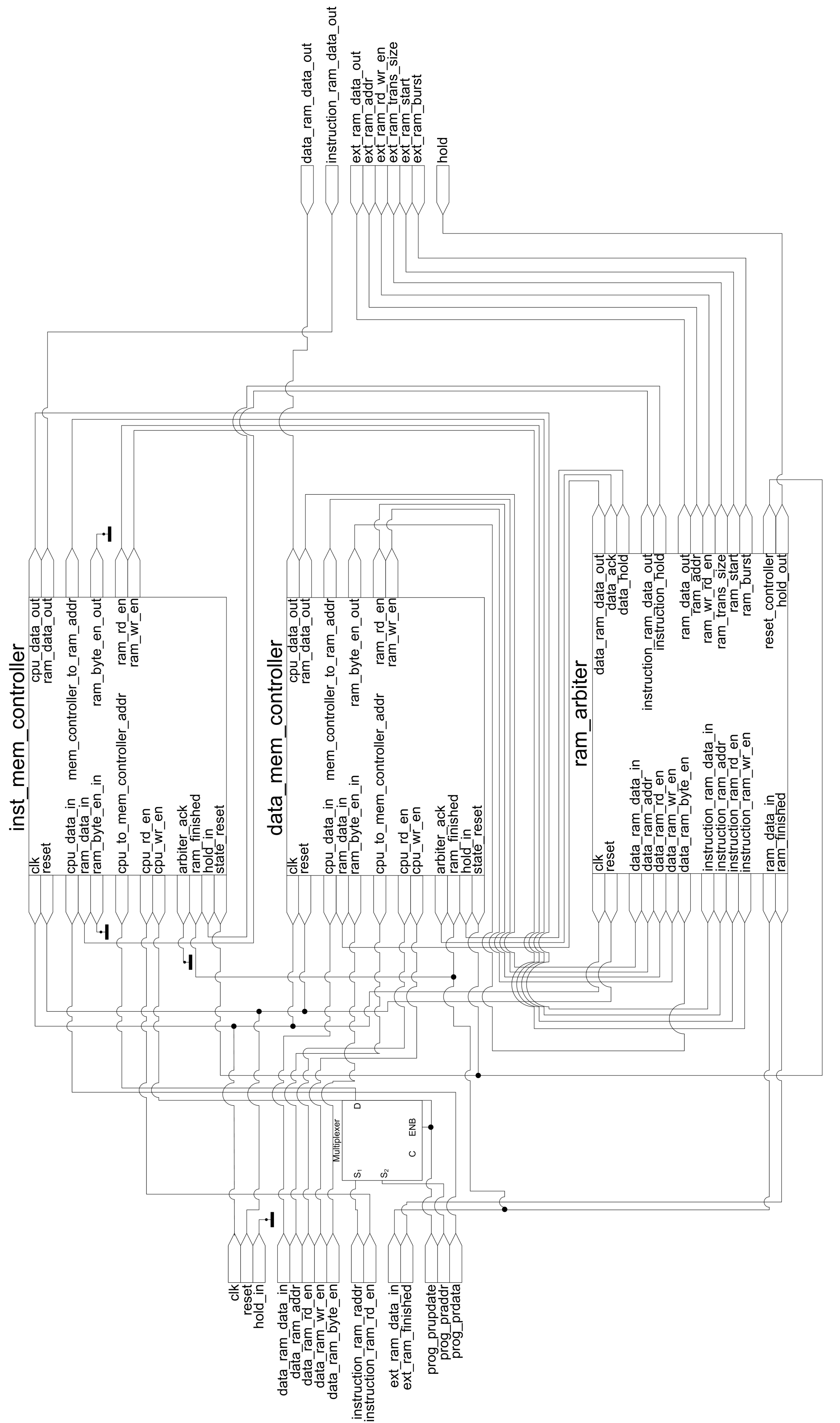


Figure 4.13: *SPEAR2* external memory configuration.

In the case of external memory accesses, the pipeline must be halted during every read or write access in order not to cause an erroneous program execution resulting from the ALU trying to operate on data words which have not been received yet. More precisely, the pipeline needs to be stopped exactly in the subsequent cycle after a request has been sent to the external memory and not immediately. This behavior can be explained with the help of the following simple example: When requesting a read access through a *LDW* instruction, the processor might need the received data word for the next instruction, thus operating on it through its forwarding mechanism. Let this subsequent instruction be an *ADD* operation. In the sense of a correct program execution, the cycle with the *LDW* instruction needs to finish without activating the hold signal so that the pipeline can fill all registers of the execute stage with the correct instruction code for the following *ADD* operation. Activating the hold signal immediately would result in the pipeline to still remain in the same state after all operations on the memory have finished and the hold signal is released. The write back stage will therefore not be able to forward the received data to the *ADD* operation as the pipeline has not switched to the next state yet. The following delayed switch of the pipeline will result in a loss of the just received data as the next operation will already be operating on the external memory. The same problem holds for the retrieve of the next instruction from the instruction memory. Activating and releasing of the hold mechanism in the same cycle of the read request will result in a pipeline that will never switch to the next task as the program counter will never be incremented. Although this is some kind of very straightforward behavior, this little trap has somehow been overseen by the author for quite a long time, resulting in a faulty behavior when connecting the new memory layer with the *SPEAR2* pipeline. Figure 4.14 illustrates the described behavior with a graphical example.

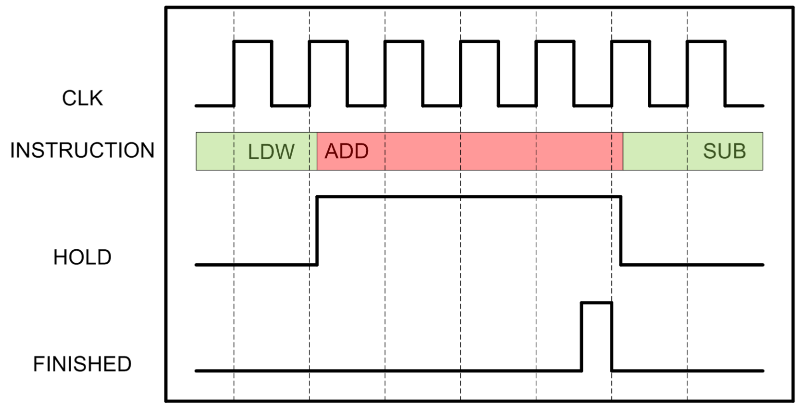


Figure 4.14: The behavior of the *SPEAR2* pipeline.

Memory Controller

The memory controllers act as the direct interfaces to the pipeline. Their main duty is to receive read and write requests from the data and instruction ports and to send them immediately to the RAM arbiter. After having transmitted an incoming request to the RAM arbiter, the memory controller will remain in a blocking state waiting for the actual operation to complete, thus ignoring all transitions on the pipeline interface and remaining idle until the RAM arbiter signals that all operations on the extern memory have finished. Instruction memory accesses are given a higher priority than the data memory accesses. In the case of simultaneous transmitted requests to the RAM arbiter, the data memory controller is forced to wait until the operation requested by the instruction memory controller has finished. On the other hand, the instruction memory controller must consequently buffer a received instruction word until the data memory operation has completed.
Table 4.12 lists all ports of the entity declaration. Table 4.13 shows the generics that can be configured when generating an instance of the memory controller.

Port	**Type**	**Direction**	**Semantics**
clk	std_ulogic	IN	Clock input.
reset	std_ulogic	IN	Reset input.
cpu_data_in	std_logic_vector[DATA_WIDTH-1:0]	IN	CPU Data in.
cpu_data_out	std_logic_vector[DATA_WIDTH-1:0]	OUT	CPU Data out.
ram_data_in	std_logic_vector[31:0]	IN	RAM data in.
ram_data_out	std_logic_vector[DATA_WIDTH-1:0]	OUT	RAM data out.
ram_byte_en_in	std_logic_vector[3:0]	IN	RAM byte enable in.
ram_byte_en_out	std_logic_vector[3:0]	OUT	RAM byte enable out.
cpu_to_mem_controller_addr	std_logic_vector[ADDRESS_WIDTH-1:0]	IN	CPU to memory controller address.
mem_controller_to_ram_addr	std_logic_vector[ADDRESS_WIDTH-1:0]	OUT	Memory controller to RAM arbiter address.
cpu_rd_en	std_ulogic	IN	CPU read request.
cpu_wr_en	std_ulogic	IN	CPU write request.
ram_rd_en	std_ulogic	OUT	External RAM read request.
ram_wr_en	std_ulogic	OUT	External RAM write request.
arbiter_ack	std_ulogic	IN	RAM arbiter acknowledge signal.
ram_finished	std_ulogic	IN	External RAM operation finished signal.
hold_in	std_ulogic	IN	Hold in.
state_reset	std_ulogic	IN	State reset signa.

Table 4.12: Port description of the *memory_controller* component.

Generic	Semantics
DATA_WIDTH	Determines the width of data words.
ADDRESS_WIDTH	Determines the width of the address.
MODE	Selects the mode to be used.

Table 4.13: Generics of the *memory_controller* component.

By setting *MODE* to *RAM_ARBITER_MASTER_MODE*, the memory controller will be configured as instruction memory controller, gaining the permission to be the first to perform its read or write operation on the external memory. Care must be taken when receiving data from the RAM arbiter as all read operations will return 32-bit words, thus reading two instructions per read access. In order to return the correct data word to the processor, the requested 16-bit instruction must be extracted from the received double word based on the value of the least significant bit of the address.

```
-------------------------------------------
-- switch bytes in case of 16 bit access --
-- if necessary                          --
-------------------------------------------
if MODE = RAM_ARBITER_MASTER_MODE then
  if address_reg(0) = '1' then
    output_reg_next <= ram_data_in(31 downto 16);
  else
    output_reg_next <= ram_data_in(15 downto 0);
  end if;
else
  -------------------
  -- 32 bit output --
  -------------------
  output_reg_next <= ram_data_in;
end if;
```

When being configured as slave via the generic *RAM_ARBITER_SLAVE_MODE*, no adaptations need to be applied on the returned value.
The functionality of the incoming and outgoing data ports like *cpu_data_in* and etc. is straightforward and does not deserve any further explanation. Finished operations on the external RAM are signaled by the *ram_finished* signal received from the RAM arbiter. In the case of two simultaneous requests, the first transition of this signal will trigger the master to switch into a wait state and the slave to start waiting for its operation to complete. The first '1' on the *ram_finished* signal will therefore inform the slave that the operation of the master has finished and that its operation is now going to be handled by the RAM arbiter. In the special case of no request from the master, the slave must be informed that its request will immediately be executed. This is realized by the *arbiter_ack* signal which is set to '1' by the arbiter in this particular situation. On

the other hand, the controller without any requests must be informed that the other one is going to access the memory, if this is really the case. As the pipeline will switch to the next instruction which might be another memory operation, the unemployed controller must be brought into the *HOLD* state in order to block all new incoming requests from the pipeline. This is achieved by the arbiter sending the *hold_in* signal. After all requests have been satisfied, both memory controllers will be waiting in their *HOLD* states. The last task to be done by the arbiter is to initiate a reset in both controllers and to release the *hold_out* signal in order to reactivate the pipeline. The reset is performed by pulling *reset_state* to '1'. Figure 4.15 shows a state diagram of the data memory controller. The one for the instruction memory controller is similar except the missing *SLAVE_WAIT* states.

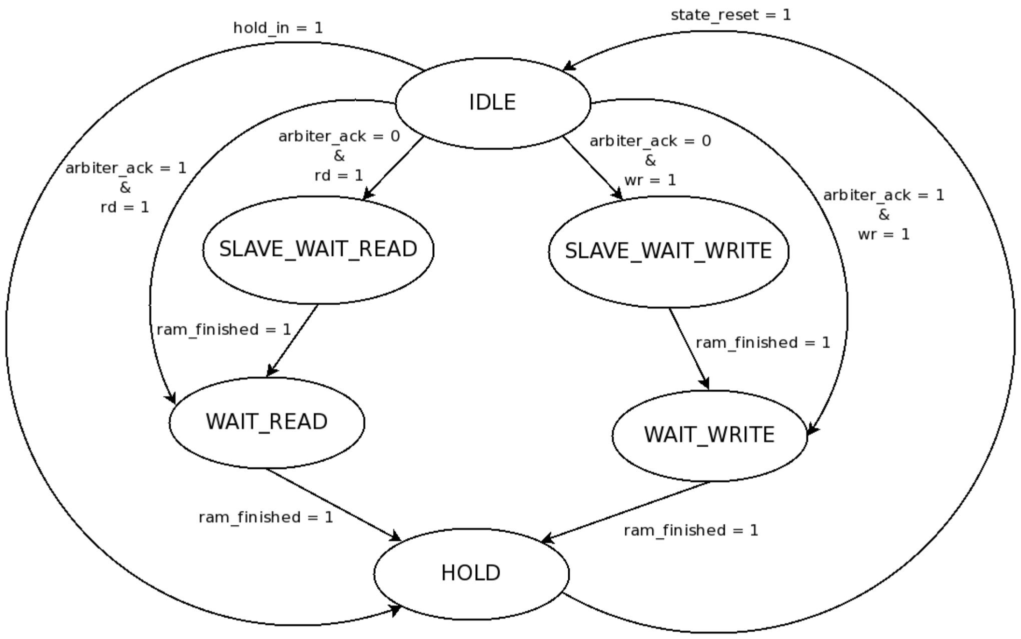


Figure 4.15: State diagram of the *memory_controller* component.

RAM Arbiter

The *RAM arbiter* is responsible for the ordered access to the single ported RAM interface, the generation of the appropriate signals for the *AMBA* master that will initiate the bus transfer and the correct handling of the *hold_out* signal in order to control the pipeline. The generics and the additional ports of the RAM arbiter are listed in Table 4.14 and Table 4.15 respectively. The remaining ports are equal to those listed in Table 4.8.

Generic	Semantics
MODE	Determines memory mode to be used.
REG_MODE	Determines if some extra register should be included into the data and address path.
EXT_RAM_ADDRESS_WIDTH	Determines the width of the external memory address bus.
DATA_RAM_ADDRESS_WIDTH	Determines the address space and the size the of data memory.
INSTRUCTION_RAM_ADDRESS_WIDTH	Determines the address space and the size of the instruction memory.
DATA_CACHE_BLOCK_WIDTH	Determines the block width of the data cache.
INSTRUCTION_CACHE_BLOCK_WIDTH	Determines the block width of the instruction cache.

Table 4.14: Generics of the *ram_arbiter* component.

Port	Type	Direction	Semantics
data_ack	std_logic	OUT	Data memory controller acknowledge signal.
data_hold	std_logic	OUT	Set data memory controller to *HOLD*.
instruction_hold	std_logic	OUT	Set instruction memory controller to *HOLD*.
reset_controller	std_ulogic	OUT	Reset signal for both memory controllers.

Table 4.15: Port description of the *ram_arbiter* component.

The value of the *MODE* field determines if the RAM arbiter is configured as interface to a cache controller or to the memory controller. The main difference between these two modes is the number of sequential read operations per read access and will be explained in detail within the *SPEAR2_CACHE_MEM* configuration of the memory layer.
As already introduced, the *data_ack* signal is used to inform the data memory controller that the instruction memory controller has no active request on the external memory and that its request is going to be executed next. The *data_hold* and *instruction_hold* signal are needed to inform the corresponding memory controller that it must switch to its *HOLD* state in situations where it has not sent any request to the RAM arbiter but the opposite controller did. The *reset_controller*

signal is used to reset both controllers into their *IDLE* state again, thus allowing them to receive the next requests from the pipeline which are again immediately send to the RAM arbiter.

4.3.3 Cache Memory

First of all, the arbitration mechanism of the single ported external RAM interface in this configuration is exactly the same as in the *SPEAR2_EXT_MEM* mode except that the memory controllers are replaced by *cache controllers*. The block diagram of the *mem_top* component is visible in Figure 4.16. The cache memories are implemented by the original memory layer of *SPEAR2_INT_MEM*, i.e., by generating one instance of the *int_mem* component for the instruction cache memory and a *data_mem_top* instance for the data cache memory. Just like the memory controllers, these cache controllers serve as direct interfaces to the pipeline, thus receiving requests from the processor and immediately transmitting them to the RAM arbiter but this time only if the requested data word is not already residing inside the cache memory. That means, before any signal is sent to the RAM arbiter, the cache controller will at first look for the referenced tag entry in its tag memory. On a miss, the request is sent to the RAM arbiter, on a hit, the referenced data word is immediately returned from the cache memory to the pipeline. Again, care must be taken if one controller exhibits a cache hit while the other one suffers from a miss, thus accessing the external RAM. Consequently, the controller with the cache hit must again be brought into an *HOLD* state via a transition on the *hold_in* port. In the case of both cache controllers returning a hit, the requested data and instruction word will be both available in the subsequent cycle. As a result, the pipeline will not be stalled as no request will be sent to the RAM arbiter.

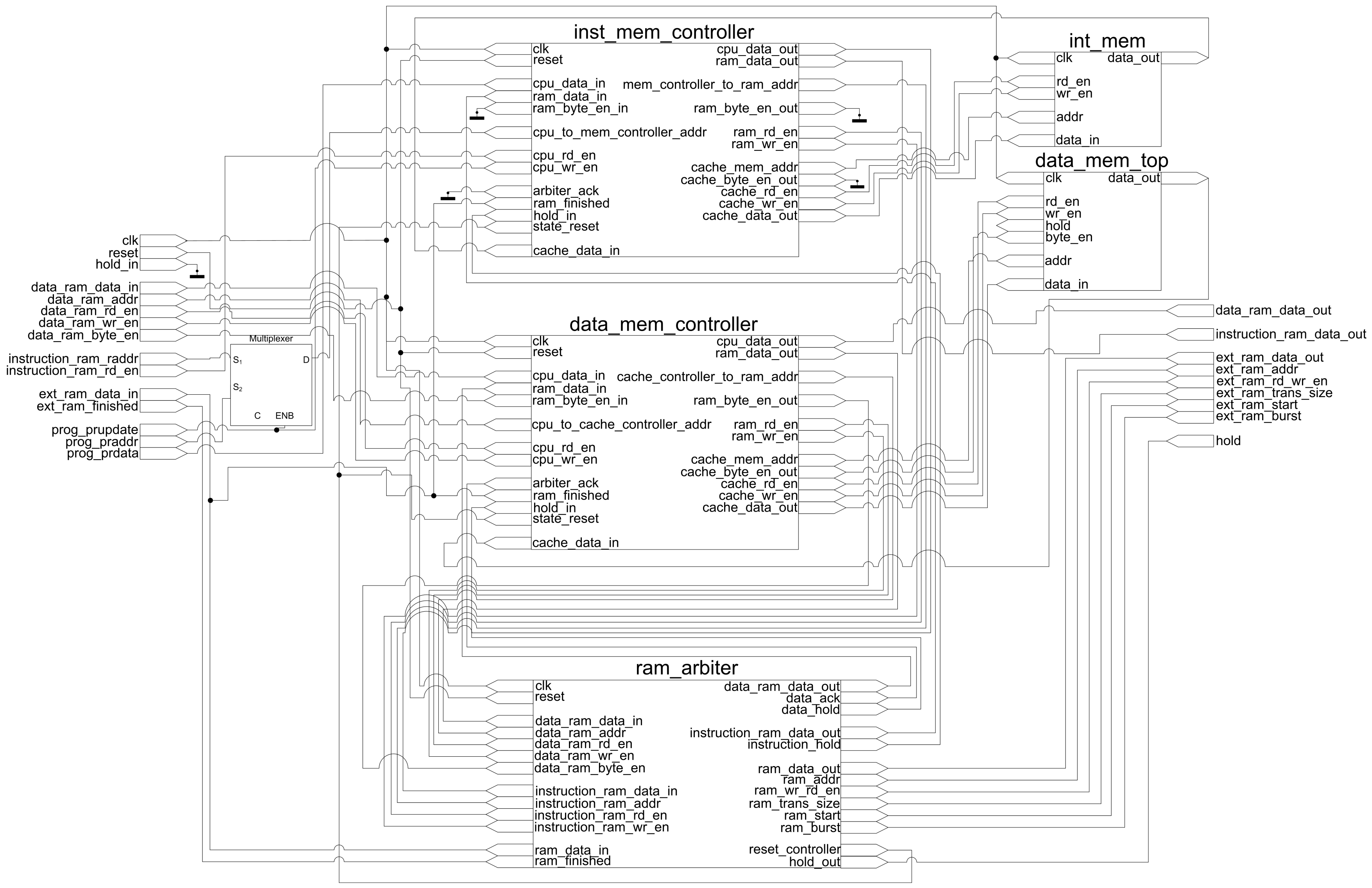


Figure 4.16: *SPEAR2* cache memory configuration.

As the write strategy has been chosen to be write-through, every write on the cache will also trigger a write operation on the external RAM. Write operations on the external memory are handled in the same way like in the *SPEAR2_EXT_MEM* configuration. This is certainly not true for read operations. As a cache controller always fetches the complete block on a miss, the RAM arbiter executes several sequential reads on the external memory for both, instruction and data cache. The number of read accesses equals the block size, i.e., the number of data words residing inside a block. Consequently, the slave controller has to wait for a number of positive transitions on the *ram_finished* port that is equal to the block size of the master controller and vice versa. As the width of an instruction word is 16-bit but every read access returns a 32-bit data word, the number of read operations is consequently only half the block size for an instruction cache controller. Furthermore, the instruction memory cache also has a data width of 32-bit which in addition halves the needed address space. Figure 4.17 shows the state diagram of the data cache controller. The one for the instruction cache controller is again similar except the missing *SLAVE_WAIT* states.

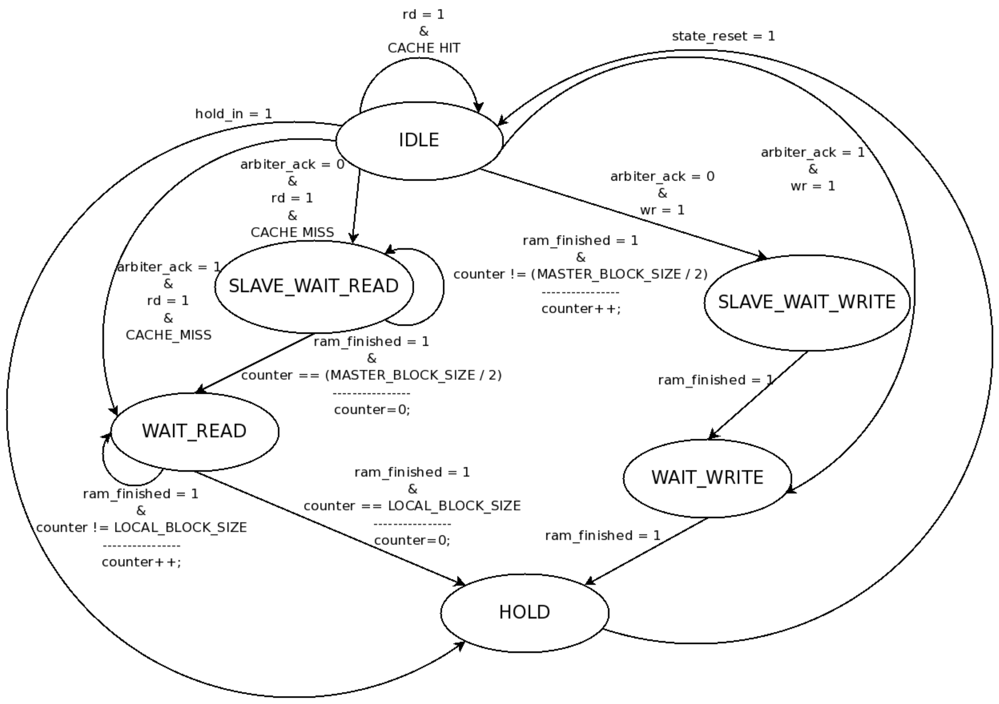

rd = 1 & CACHE HIT
state_reset = 1
hold_in = 1
IDLE
arbiter_ack = 0 & rd = 1 & CACHE_MISS
arbiter_ack = 0 & wr = 1
arbiter_ack = 1 & wr = 1
ram_finished = 1 & counter != (MASTER_BLOCK_SIZE / 2) ---------------- counter++;
arbiter_ack = 1 & rd = 1 & CACHE_MISS
SLAVE_WAIT_READ
SLAVE_WAIT_WRITE
ram_finished = 1 & counter == (MASTER_BLOCK_SIZE / 2) ---------------- counter=0;
ram_finished = 1
WAIT_READ
WAIT_WRITE
ram_finished = 1 & counter != LOCAL_BLOCK_SIZE ---------------- counter++;
ram_finished = 1 & counter == LOCAL_BLOCK_SIZE ---------------- counter=0;
ram_finished = 1
HOLD

Figure 4.17: State diagram of the *cache_controller* component.

Above all, the RAM arbiter is now able to use the *burst* mechanism due to the iterative accesses on the memory. For details concerning the VHDL implementation, please refer to the source

code and to the source code documentation. Especially *ram_arbiter_arc.vhd* implements important procedures concerning the address translation which is needed to correctly address the *AMBA* master in order to provide byte, word and double word accesses.
The new *SPEAR2* memory layer can be equipped with cache controllers implementing the following three caching policies:

- Direct mapped
- Fully associative
- 2-way set associative

The generic lists and port descriptions are the same for all caching strategies. See Table 4.17 and Table 4.16 for more information. Table 4.16 only lists additional ports compared to those that are already listed in Table 4.12. The *MASTER_BLOCK_WIDTH* generic value is needed by the slave controller in order to know the block size of the master controller which defines the number of incoming positive transitions on the *ram_finished* port that the slave must wait before its request is going to be executed by the arbiter.

Port	**Type**	**Direction**	**Semantics**
cache_data_in	std_logic_vector[31:0]	IN	Cache data in.
cache_data_out	std_logic_vector[31:0]	OUT	Cache data out.
cache_byte_en_out	std_logic_vector[3:0]	OUT	Cache memory byte enable.
cache_controller_to_ram_addr	std_logic_vector[ADDRESS_WIDTH-1:0]	OUT	Cache controller to RAM arbiter address.
cache_controller_to_cache_mem_addr	std_logic_vector[CACHE_ADDRESS_WIDTH-1:0]	OUT	Cache controller to cache memory address.
cache_rd_en	std_ulogic	OUT	Cache memory read enable.
cache_wr_en	std_ulogic	OUT	Cache memory write enable.

Table 4.16: Additional ports for all cache controllers components.

Generic	**Semantics**
DATA_WIDTH	Determines the data width.
ADDRESS_WIDTH	Determines the address width.
MASTER_BLOCK_WIDTH	Determines the block width of the master controller.
BLOCK_WIDTH	Determines the local block width.
BLOCK_COUNT	Determines the local block count.
MODE	Determines if the controller is acting as master (INSTRUCTIONS) or slave (DATA)

Table 4.17: Generics of all cache controllers components.

Thanks to a generic interface and a modular structure, the one and only noticeable difference between these three cache controllers is the way they are searching for a requested data word in their tag memories. The following three code segments will show the tag search procedures of the three presented cache controllers, beginning with the direct mapped placement strategy that simple inspects on only one definite position, that is the block defined by the index field[7].

```
if (tag_mem(v_cache_block_address_int)(TAG_WIDTH - 1 downto 0) = v_mem_tag_address)
    and (tag_mem(v_cache_block_address_int)(TAG_WIDTH) = '1') then
  cache_rd_en <= '1';
  ...
  ...
  ...
end if;
```

Searching for a requested data word indeed becomes more complicated with the fully associative placement policy.

```
for i in 0 to BLOCK_COUNT - 1 loop
  if (tag_mem(i)(TAG_WIDTH - 1 downto 0) = v_mem_tag_address)
      and (tag_mem(i)(TAG_WIDTH) = '1') then
    cache_rd_en <= '1';
    v_cache_block_address := std_logic_vector(to_unsigned(i, INDEX_WIDTH));
  end if;
end loop;
...
...
```

[7]Remember *Chapter 2* - Block Placement.

As it is visible in the VHDL code segment, the search procedure of this cache controller will cover the complete tag memory. The tremendous negative effect of the loop statement on the hardware requirements will be examined in the following.
Last but not least, the next code segment presents the tag search procedure of the 2-way set associative cache controller. Indeed, this placement strategy can be easily deduced from the direct mapped one by just splitting the cache memory into two parts that are equal in size. As we have seen in *Chapter 2*, the tag field will consequently become on bit larger, thus moving to the right, causing the index field to shrink.

```
if (tag_mem(v_cache_block_addressA_int)(TAG_WIDTH - 1 downto 0) = v_mem_tag_address)
    and (tag_mem(v_cache_block_addressA_int)(TAG_WIDTH) = '1') then
  cache_rd_en <= '1';
  ...
  ...
  ...
elsif (tag_mem(v_cache_block_addressB_int)(TAG_WIDTH - 1 downto 0) = v_mem_tag_address)
       and (tag_mem(v_cache_block_addressB_int)(TAG_WIDTH) = '1') then
  cache_rd_en <= '1';
  ...
  ...
  ...
```

The reader which is especially interested in technical details concerning the implementation in VHDL, e.g., address translation, state transitions and etc., is referred to the source code and the source code documentation.
The evaluation and practical experiments yielded many good but also some bad results. By synthesizing our direct mapped cache controller on a Cyclone IV FPGA, we found out that the maximal cache memory capacity of the instruction cache is upper bounded with 64 KBytes whereas the data cache memory can only be equipped with 16 KBytes. The question now certainly is why is maximum storage of the data cache that noticeable smaller than the one of the instruction cache? Focusing on the *SPEAR2* architecture, the instruction memory has a tremendous timing advantage over the data memory as the only signal to run through the combinatorial path between the processor and the memory level is the value of the program counter. In contrast to the signals of the instruction memory, the data path signals connected with the data cache are much longer: In the worst case, a read instruction yields a cache hit and the requested data word will be immediately available. If the subsequent instruction is also going to operate on the just received data word, the data from the cache will be routed through the forwarding unit, entering the huge memory access address-multiplexer logic in the processor top level entity[8] and finally

[8]Remember that all extension modules of the *SPEAR2* architecture are memory mapped, thus they are accessed via normal load/store instructions.

reaching the memory layer again where its tag field must be compared with the values stored in the tag memory. Above all, there is another huge multiplexer logic positioned shortly before the external RAM interface in the RAM arbiter which needs to adapt incoming addresses to be able to correctly drive the signals which are routed to the *AMBA* bus master. As the searching for a given address in the tag memory is a complex task that costs a lot of cycle time, every of the implemented cache controllers can be equipped with an additional register, thus breaking up the long combinatorial path. This register can help to meet the timing requirements even for larger data cache sizes. in order to be able to increase the capacity of the tag memory. This mode is enabled by setting the *reg_mode* generic in the top-level to *REG_TRUE*. The difference between a cache controller and its registered version is that when including this extra register, enable signals, data and addresses will be transmitted to the RAM arbiter in the subsequent cycle and not immedeately in the case of a cache miss. The usage of this additional register clearly depends on the used hardware technology. We found out that for a direct mapped cache controller on a Cyclone IV the additional register must be included to achieve a block count of more than 64. Otherwise the timing analyzer of the used place and route (*QUARTUS*) tool raised critical warnings concerning timing violations.
The now following investigation on the required combinatorial functions and register is going to justify the estimated hardware overheads from *Chapter 2* for every of the three implemented cache controllers. Clearly visible at first sight is the tremendous increase of hardware costs when switching from a direct mapped caching strategy to a fully associative one beyond a cache size of 1024 KBytes (Figure 4.18). The reason for this cost explosion has already been explained: As the complete tag memory needs to be searched during every cache access, the *loop* statement shown in the source code example above will result in the synthesis of a huge multiplexer and comparator logic, therefore increasing the combinatorial path which will additionally result in an increased runtime of the address and data signals. Figure 4.19 and Figure 4.20 illustrate the resource requirements for block sizes of 8 and 16. We can conclude some interesting facts from this figures. Firstly, the ratio of needed hardware resources between the direct mapped and the fully associative cache controller does not change with a variable block size and can be constantly approximated with 1:2 for a block count higher than 64. For a block count lower than 64, the difference is minimal, thus it is reasonable to implement the fully associative cache controller instead of the direct mapped one. To the rescue, as the 2-way set associative caching policy has shown to be very efficient in the simulations of our two benchmark programs, this cache controller might be used instead of the direct mapped caching strategy in order to approximately keep the performance of the fully associative one for block counts higher than 64.
Furthermore, the need for combinatorial functions and registers decreases with an increasing block size. Remembering the address format introduced in *Chapter 2*, increasing the block size

will result in a bigger block field, thus shrinking the tag field or the index field at the same time. Since less sets and blocks can reside inside the cache, less tags have to be compared which explains the decreasing hardware costs. Nevertheless, the number of needed memory bits will increase.

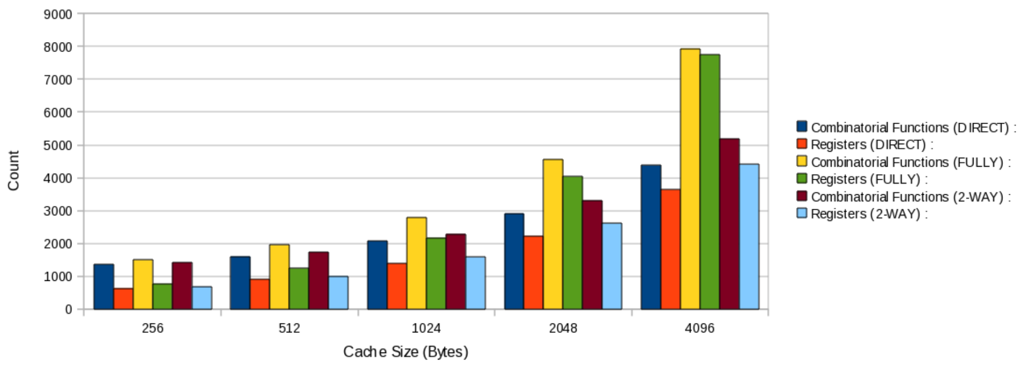


Figure 4.18: Hardware requirements (BLOCK SIZE = 4).

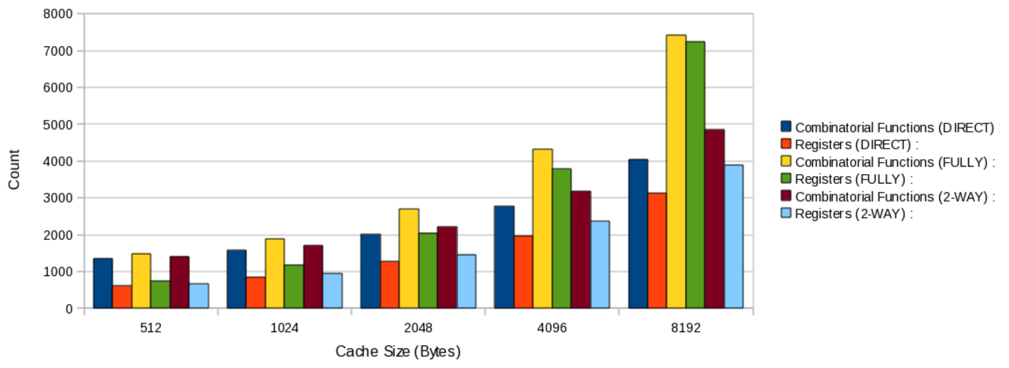


Figure 4.19: Hardware requirements (BLOCK SIZE = 8).

Last but least it is time to prove the performance increase gained by caching with the execution of the *cache_benchmark2.c* program on the hardware version of the *SPEAR2* core. The exact values are listed in Table 4.18. As expected, the use of caches can significantly reduce the performance hit due to the slow external memory. However, it seems somewhat counter-intuitive that the fully associative caching strategy performs worse that the direct mapped and the 2-way set associative caches.

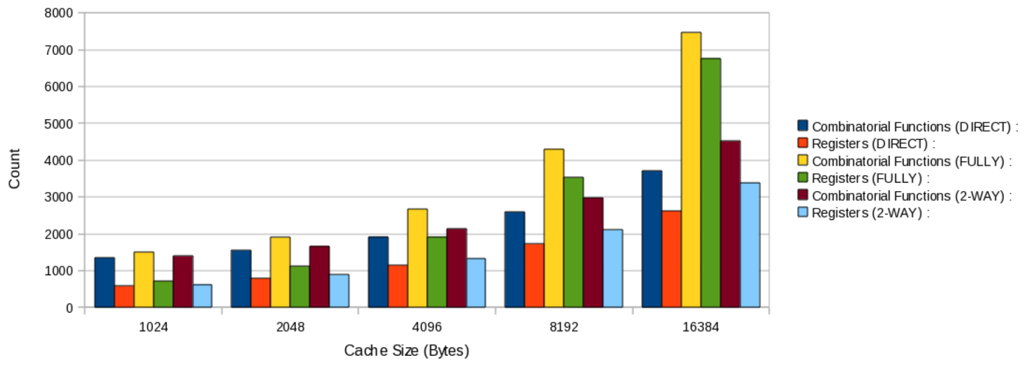


Figure 4.20: Hardware requirements (BLOCK SIZE = 16).

Mode	**Cycles**
SPEAR2_INT_MEM	47073111
SPEAR2_EXT_MEM	460293819
SPEAR2_DIRECT_MAPPED	59550053
SPEAR2_SET_ASSOCIATIVE	58518027
SPEAR2_FULLY_ASSOCIATIVE	61150181

Table 4.18: Performance evaluation in hardware (*cache_benchmark2.c*).

To the rescue, *SPEAR2SIM* enables us to reveal the reason for this: First of all, we already have shown that the fully associative cache controller is only useful when implementing a LRU replacement policy. For the implemented fully associative cache a FIFO policy has been used. Comparing the miss counts on the data cache and the instruction cache extracted from a simulation run with *SPEAR2SIM* shows the following (Table 4.19): Although the data cache miss count is considerably higher for the direct mapped cache controller, the fully associative one implementing FIFO as replacement strategy exhibits a horrible behavior on fetching instructions which finally explains the higher number of needed cycles. Using an instruction cache implementing a fully associative caching strategy is therefore not recommendable for this specific application.

Mode	**DATA misses**	**INSTRUCTION misses**
SPEAR2_DIRECT_MAPPED	40915	442848
SPEAR2_FULLY_ASSOCIATIVE	13284	544769

Table 4.19: Simulation results.

4.4 Conclusion

”Don’t look back in anger, at least not today ...“ Noel Gallagher (Former Oasis member).

We have pointed out that in the last 20 years the performance gap between processors and memory modules has become huge. Caching has been introduced to mitigate this discrepancy. We have shown that the efficiency of a cache controller depends on various aspects like the used placement and replacement strategy, the block size, the block count etc. Moreover, applying an optimization on one aspect might lead to a degradation of another one, e.g., storage capacity can be increased with the sacrifice of a larger hit time. Even advanced techniques like prefetching, that are supposed to speed up a cache controller can result in a performance degradation when they are not implemented carefully enough.

In order to efficiently find an appropriate caching strategy for a given application, a novel simulator toolchain, i.e., *SPEAR2SIM* has been introduced. This *SPEAR2* ISA simulator is capable of executing programs that have been built with the original *SPEAR2* toolchain. With this environment it has become possible to extract useful information about the cache usage when running an application on the emulated core. This particular information, i.e., the miss rates and miss counts of the data and instruction cache serve as a basis for the decision finding.

The subsequent redesign of the original memory architecture of *SPEAR2* with a following implementation of three promising cache controllers in hardware (VHDL) has shown that finding the best adapted caching strategy is not only about tuning its efficiency (i.e., miss rate). Several aspects such as hardware requirements and timing issues do have a huge influence on an adequate overall decision.

As a result of this thesis, a new memory layer that can be equipped with three different cache controllers has been designed. The different modes and caching strategies can be easily selected by specific generics in the *SPEAR2* entity. The external memory interface can be connected with any possible kind of memory or memory controller.

This short resume finally closes this master thesis with the hope that the implementation of the new memory architecture together with the cache controllers has provided a solid basis for future projects like the integration of a memory management unit, the implementation of an embedded operating system and many more.

APPENDIX A

Benchmark Programs

```
/*!
 * \file    cache_benchmark1.c
 * \author  Michael Birner
 * \date    27.06.2011
 * \version 1.0
 * \brief   SPEAR2 cache benchmark program
 *
 * Copyright 2011 Michael Birner - michael.birner@gmx.at
 *
 * This file is part of SPEAR2SIM.
 *
 * SPEAR2SIM is free software: you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation, either version 3
 * of the License, or (at your option) any later version.
 *
 * SPEAR2SIM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with SPEAR2SIM. If not, see http://www.gnu.org/licenses/.
 */

/*----------------------------------------------------------------------------*/
/*                              INCLUDES                                      */
/*----------------------------------------------------------------------------*/

#include "cache_benchmark1.h"

/*----------------------------------------------------------------------------*/
```

```
/*                              DECLARATIONS                                  */
/*----------------------------------------------------------------------------*/

int a1[ARRAY_SIZE];
int a2[ARRAY_SIZE];
int a3[ARRAY_SIZE];
int a4[ARRAY_SIZE];

void foo1(void);
void foo2(void);
void foo3(void);
void foo4(void);
void foo5(void);
void foo6(void);

/*----------------------------------------------------------------------------*/
/*                              SUBROUTINES                                   */
/*----------------------------------------------------------------------------*/

/*!
 * \brief Init function for array
 */
void init_array(void)
{
    int i = 0;

    /* fill all arrays with default values */
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        a1[i] = 1;
        a2[i] = -1;
        a3[i] = 0;
        a4[i] = 0;
    }
}

/*!
 * \brief Sum function over array entries
 * \return Sum over array entries
 */
int sum_array(void)
{
    int i = 0;
    int j = 0;
    int sum = 0;

#ifdef SPEAR2
#ifdef DEBUG
    printf("\n\rEntering sum_array()");
#endif
#endif
```

```
    for(j = 0; j < LOOP_COUNT; j++)
    {
        for(i = 0; i < ARRAY_SIZE; i++)
        {
            foo5();
            sum = sum + a1[i];
            foo6();
            sum = sum + a4[ARRAY_SIZE - i - 1];
            foo5();
        }
        NOP_CHAIN
        foo5();
    }

    return sum;
}

/*!
 * \brief Some memory access function
 */
void foo1(void)
{
    int i = 0;

#ifdef SPEAR2
#ifdef DEBUG
    printf("\n\rEntering foo1()");
#endif
#endif
    NOP_CHAIN
    NOP_CHAIN
    NOP_CHAIN
    NOP_CHAIN

    for(i = 0; i < ARRAY_SIZE; i++)
    {
        foo5();
        a4[i] = a3[ARRAY_SIZE - i - 1];
        NOP_CHAIN
    }
}

/*!
 * \brief Some memory access function
 */
void foo2(void)
{
    int i = 0;
    int j = 0;
    int val1 = 0;
    int val2 = 0;

#ifdef SPEAR2
#ifdef DEBUG
    printf("\n\rEntering foo2()");
#endif
#endif

    NOP_CHAIN

    j = ARRAY_SIZE - 1;

    for(i = 0; i < ARRAY_SIZE; i++)
    {
        NOP_CHAIN
        val1 = a4[i];
        val2 = a2[j];
        j = j - 1;
        NOP_CHAIN
    }
}

/*!
 * \brief Some memory access function
 */
void foo3(void)
{
    int i = 0;

#ifdef SPEAR2
#ifdef DEBUG
    printf("\n\rEntering foo3()");
#endif
#endif

    NOP_CHAIN
    NOP_CHAIN
    NOP_CHAIN

    for(i = 0; i < ARRAY_SIZE; i++)
    {
        foo5();
        a3[i % ARRAY_SIZE] = a2[(i + 20) % ARRAY_SIZE];
    }

    NOP_CHAIN
    NOP_CHAIN
}

/*!
 * \brief Some memory access function
 */
void foo4(void)
{
    int i = 0;

#ifdef SPEAR2
#ifdef DEBUG
    printf("\n\rEntering foo4()");
#endif
#endif

    for(i = 0; i < ARRAY_SIZE; i++)
    {
        a1[i] = a2[ARRAY_SIZE - 1 - i];
        NOP_CHAIN
    }
}

/*!
 * \brief Some NOP function
 */
void foo5(void)
{
#ifdef SPEAR2
#ifdef DEBUG
    printf("\n\rEntering foo5()");
#endif
#endif

    /* a chain of nops */
    NOP_CHAIN
    NOP_CHAIN
    NOP_CHAIN
    NOP_CHAIN
}

/*!
 * \brief Some NOP function
 */
void foo6(void)
{
#ifdef SPEAR2
#ifdef DEBUG
    printf("\n\rEntering foo6()");
#endif
#endif

    /* a chain of nops */
    NOP_CHAIN
    NOP_CHAIN
    NOP_CHAIN
    NOP_CHAIN
}

/*-------------------------------------------------------------------------*/
/*                                MAIN                                     */
/*-------------------------------------------------------------------------*/

/*!
 * \brief main
 */
int main(void)
{
    int sum = 0;

#ifdef SPEAR2
    uint32_t cycles = 0;

    /* module handlers */
    module_handle_t counterHandle;

    /* init counter */
    counter_initHandle(&counterHandle, ((uint32_t)-320));
    /* set counter prescaler */
    counter_setPrescaler(&counterHandle, ((uint8_t)255));

    printf("\n\rStarting Benchmark !!!\n\r");

    /* reset and start counter */
    counter_reset(&counterHandle);
    counter_start(&counterHandle);
#endif

    /* init test arrays */
    init_array();

    /* call first subroutine */
    sum = sum_array();

    /* do some stuff */
    for(sum = 0; sum < LOOP_COUNT; sum++)
    {
        foo2();
        foo5();
        foo1();
        foo3();
        foo4();
    }

#ifdef SPEAR2
    /* stop counter and output value */
    counter_stop(&counterHandle);
    cycles = counter_getValue(&counterHandle);

    printf("\n\rResult : %d", sum);
    printf("\n\rCycles : %d\n\r", cycles);
#endif

    return sum;
}

/* EOF */


/*!
 * \file    cache_benchmark1.h
 * \author  Michael Birner
 * \date    10.10.2011
 * \version 1.0
 * \brief   Header file for cache_benchmark1.c
 *
 * Copyright 2011 Michael Birner - michael.birner@gmx.at
 *
 * This file is part of SPEAR2SIM.
 *
 * SPEAR2SIM is free software: you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation, either version 3
 * of the License, or (at your option) any later version.
 *
 * SPEAR2SIM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with SPEAR2SIM. If not, see http://www.gnu.org/licenses/.
 */

#ifndef __CACHE_BENCHMARK1_H__
#define __CACHE_BENCHMARK1_H__

/*---------------------------------------------------------------------------*/
/*                               INCLUDES                                    */
/*---------------------------------------------------------------------------*/

#include <stdio.h>

/* SPEAR2 lib */
#include <drivers/counter.h>

/*---------------------------------------------------------------------------*/
/*                               DEFINES                                     */
/*---------------------------------------------------------------------------*/
```

```
#define NOP_CHAIN asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop");

#define ARRAY_SIZE  8000
#define LOOP_COUNT  10

/*-------------------------------------------------------------------------*/
/*                               DECLARATIONS                              */
/*-------------------------------------------------------------------------*/

#endif /* __CACHE_BENCHMARK1_H__ */

/* EOF */


/*!
 * \file    cache_benchmark2.c
 * \author  Michael Birner
 * \date    27.06.2011
 * \version 1.0
 * \brief   SPEAR2 cache benchmark program
 *
 * Copyright 2011 Michael Birner - michael.birner@gmx.at
 *
 * This file is part of SPEAR2SIM.
 *
 * SPEAR2SIM is free software: you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation, either version 3
 * of the License, or (at your option) any later version.
 *
 * SPEAR2SIM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with SPEAR2SIM. If not, see http://www.gnu.org/licenses/.
 */

/*----------------------------------------------------------------------------*/
/*                                INCLUDES                                    */
/*----------------------------------------------------------------------------*/

#include "cache_benchmark2.h"

/*----------------------------------------------------------------------------*/
/*                              DECLARATIONS                                  */
/*----------------------------------------------------------------------------*/

int a[ARRAY_SIZE];
int b[LOOP_COUNT];

void foo1(void);
void foo2(void);

/*----------------------------------------------------------------------------*/
/*                              SUBROUTINES                                   */
/*----------------------------------------------------------------------------*/

/*!
 * \brief Init function for array
 */
void init_array(void)
{
    int i = 0;

    /* fill all arrays with default values */
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        a[i] = ARRAY_SIZE - i - 1;
    }

    for(i = 0; i < LOOP_COUNT; i++)
    {
        b[i] = 1;
    }
}

/*!
 * \brief Merge sort algorithm
 */
void merge_sort(int list[], int size)
{
    int i = 0;
    int j = 0;
    int k = 0;
    int val = 0;

    if(size > 1)
    {
        int temp1[size / 2];
        int temp2[(size + 1) / 2];

        NOP_CHAIN

        for(i = 0; i < size / 2; ++i)
        {
            temp1[i] = list[i];
        }
        for(i = (size / 2); i < size; ++i)
        {
            temp2[i - (size / 2)] = list[i];
        }

        for(j = 0; j < LOOP_COUNT; j++);
        {
            NOP_CHAIN
            val = b[j];
            for(k = 0; k < LOOP_COUNT; k++)
            {
                val = a[(k + size) % ARRAY_SIZE];
                val = a[ARRAY_SIZE - 1 - k];
            }
            NOP_CHAIN
        }

        foo1();
        foo2();

        merge_sort(temp1, (size / 2));
        merge_sort(temp2, ((size + 1) / 2));

        foo1();
        foo2();
        for(j = 0; j < LOOP_COUNT; j++)
        {
            NOP_CHAIN
            val = b[j];
            for(k = 0; k < LOOP_COUNT; k++)
            {
                val = a[(k + size) % ARRAY_SIZE];
                val = a[ARRAY_SIZE - 1 - k];
            }
            NOP_CHAIN
        }

        int *pos1 = &temp1[0];
        int *pos2 = &temp2[0];

        NOP_CHAIN

        for(i = 0; i < size; ++i)
        {
            if(*pos1 <= *pos2)
            {
                list[i] = *pos1;
                if(*pos1 == temp1[(size / 2) - 1])
                {
                    pos1 = &temp2[(size + 1) / 2 - 1];
                }
                else
                {
                    ++pos1;
                }
            }
            else
            {
                list[i] = *pos2;
                if(*pos2 == temp2[(size + 1) / 2 -1])
                {
                    pos2 = &temp1[size / 2 - 1];
                }
                else
                {
                    ++pos2;
                }
            }
        }
    }
}

/*!
 * \brief Some NOP function
 */
```

```
void foo1(void)
{
    NOP_CHAIN
    NOP_CHAIN
}

/*!
 * \brief Some NOP function
 */
void foo2(void)
{
    NOP_CHAIN
    NOP_CHAIN
}

/*--------------------------------------------------------------------------------*/
/*                                MAIN                                            */
/*--------------------------------------------------------------------------------*/

/*!
 * \brief main
 */
int main(void)
{
    int sum = 0;

#ifdef SPEAR2
    uint32_t cycles = 0;

    /* module handlers */
    module_handle_t counterHandle;

    /* init counter */
    counter_initHandle(&counterHandle, ((uint32_t)-320));

    printf("\n\rStarting Benchmark !!!\n\r");

    /* reset and start counter */
    counter_reset(&counterHandle);
    counter_start(&counterHandle);
#endif

    /* init test arrays */
    init_array();

    /* call the merge sort algorithm */
    merge_sort(a, ARRAY_SIZE);

#ifdef SPEAR2
    /* stop counter and output value */
    counter_stop(&counterHandle);
    cycles = counter_getValue(&counterHandle);

    printf("\n\rResult : %d", sum);
    printf("\n\rCycles : %d\n\r", cycles);
#endif

    return sum;
}

/* EOF */


/*!
 * \file    cache_benchmark2.h
 * \author  Michael Birner
 * \date    10.10.2011
 * \version 1.0
 * \brief   Header file for cache_benchmark2.c
 *
 * Copyright 2011 Michael Birner - michael.birner@gmx.at
 *
 * This file is part of SPEAR2SIM.
 *
 * SPEAR2SIM is free software: you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation, either version 3
 * of the License, or (at your option) any later version.
 *
 * SPEAR2SIM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with SPEAR2SIM. If not, see http://www.gnu.org/licenses/.
 */

#ifndef __CACHE_BENCHMARK2_H__
#define __CACHE_BENCHMARK2_H__

/*----------------------------------------------------------------------------*/
/*                              INCLUDES                                      */
/*----------------------------------------------------------------------------*/

#include <stdio.h>

/* SPEAR2 lib */
#include <drivers/counter.h>

/*----------------------------------------------------------------------------*/
/*                              DEFINES                                       */
/*----------------------------------------------------------------------------*/
```

```
#define NOP_CHAIN asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop"); \
                  asm("nop");

#define ARRAY_SIZE  15000
#define LOOP_COUNT  2

/*--------------------------------------------------------------------------------*/
/*                              DECLARATIONS                                      */
/*--------------------------------------------------------------------------------*/

#endif /* __CACHE_BENCHMARK2_H__ */

/* EOF */
```

Bibliography

[1] *PowerPC 603e and EM603e - RISC Microprocessor Family User's Manual.* IBM, https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/, 1.1 edition, 1998.

[2] *LEON2 Processor User's Manual.* GAISLER RESEARCH, http://www.gaisler.com/cms/, 1.0.23 edition, 2004.

[3] Babak Falsafi An-Chow Lai, Cem Fide. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *ISCA 01 Proceedings of the 28th annual international symposium on Computer architecture*, pages 144 – 154. ACM, 2001.

[4] ARM. *AMBA Specification.* ARM, 2 edition, 1999.

[5] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 1966.

[6] Carlos Carvalho. The Gap between Processor and Memory Speeds. In *ICCA 2002*, pages 27 – 34. ICCA, 2002.

[7] TIS Committee. *Executable and Linkable Format (ELF).* Number 1.2. Tool Interface Standards (TIS), 1995.

[8] John L. Hennessy David A. Patterson. *Rechnerorganisation und Rechnerentwurf.* Oldenbourg Verlag, 2009.

[9] Martin Delvai. *Handbuch für SPEAR.* Embedded Computing Systems Group Technische Universität Wien, 2002.

[10] Martin Fletzer. *SPEAR2 - An Improved Version of SPEAR.* Embedded Computing Systems Group Technische Universität Wien, 2007.

[11] Eugene J. Shekita Honestey C. Young. An Intelligent I-Cache Prefetch Mechanism. In *Computer Design: VLSI in Computers and Processors. ICCD '93.*, pages 44 – 49. IEEE, 1993.

[12] David A. Patterson John L. Hennessy. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2006.

[13] Norman P. Jouppi. Cache Write Policies And Performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191 – 201. IEEE, 1993.

[14] Martin Luipersbeck. *Integration eines AMBA SDRAM-Controllers fuer SPEAR2*. Embedded Computing Systems Group Technische Universität Wien, 2011.

[15] Yan Solihin Mazen Kharbutli. Counter-Based Cache Replacement Algorithms. In *IEEE Transactions On Electronic Computers*, volume 57, pages 433 – 447. IEEE, 2005.

[16] Josef Mosser. *AMBA4SPEAR2: An AMBA Extension Module for the SPEAR2 Processor Core*. Embedded Computing Systems Group Technische Universität Wien, 2008.

[17] Tariq Jamil Richard Stacpoole. Cache memories - Bridging the performance gap. *IEEE Potentials*, pages 24 – 29, April/May 2000.

[18] Cosimo Antonio Prete Saverio Lorenzini, Gabriele Luculli. A Fast Procedure Placement Algorithm For Optimal Cache Use. In *9th Mediterranean Electrotechnical Conference, 1998. MELECON 98.*, pages 1279 – 1283. IEEE, 1998.

[19] N. Jouppi S.E. Wilton. An Enhanced Access And Cycle Time Model For On-Chips Caches. In *EC WRL Research Report 9315*. DIGITAL - Western Research Laboratory, 1994.

[20] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14:473–530, 1982.

[21] Martin Walter. *The SPEAR2 Hardware/Software Interface*. Embedded Computing Systems Group Technische Universität Wien, 2011.

[22] Wikipedia. Executable and linkable format — wikipedia, the free encyclopedia, 2011. [Online; accessed 28-November-2011].

[23] Wikipedia. Powerpc — wikipedia, the free encyclopedia, 2011. [Online; accessed 28-November-2011].

[24] Wikipedia. Sparc — wikipedia, the free encyclopedia, 2011. [Online; accessed 28-November-2011].

[25] Wikipedia. Superscalar — wikipedia, the free encyclopedia, 2011. [Online; accessed 28-November-2011].

[26] M. V. Wilkes. Slave Memories and Dynamic Storage Allocation. In *IEEE Transactions On Electronic Computers*, volume EC-14, page 270. IEEE, 1965.

[27] Feipei Lai Yen-Jen Chang. Paged Cache: An Efficient Partition Architecture For Reducing Power, Area And Access Time. In *2002 Asia-Pacific Conference on Circuits and Systems, 2002. APCCAS '02.*, pages 473 – 478. IEEE, 2002.