UNIVERSITÄT DES SAARLANDES

Naturwissenschaftlich-Technische Fakultät I

Fachrichtung Informatik

Integration of Various Memory Modes into a Multi-Core MIPS Machine Construction

Masterarbeit im Fach Informatik
Masterthesis in Computer Science

von / by

Jenny Hotzkow

angefertigt unter der Leitung von / supervised by

Prof. Dr. Wolfgang J. Paul

begutachtet von / reviewed by

Prof. Dr. Wolfgang J. Paul
Asst. Prof. Dr. Peter-Michael Seidel

vorgelegt am / submitted

August 26, 2014

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ________________ ______________________

(Datum/Date) (Unterschrift/Signature)

Abstract

Modern processors use caches and multi-core architectures to maximize their performance. Nevertheless, the formal specification and verification of the complete design of such a processor including a cache based shared memory system are hardly studied in literature. To a large extent this is due to the complexity of the constructions involved. However, the problem is dealt with at the chair of computer architecture and parallel computing at Saarbrücken University with a multi-core MIPS design [8].

When dealing with caches, it is crucial to ensure cache consistency. But another matter arises when dealing with I/O devices. In particular it has to be ensured that no stale data is read by the I/O devices. This means for a cache coherence protocol that cache lines corresponding to memory addresses of the I/O ports have to be *clean*. Common solutions involve a procedure, where the CPU forces the cached data to be written into the main memory. This can be done with a *flush*, but then the cache lines are invalidated and have to be re-fetched for further usage. Therefore uncached or write through writes are preferred, since they avoid this performance issue. But the current MIPS design of [8] only supports the write back policy.

This shortcoming is addressed in the course of this thesis. The existing model is extended to support both policies write back and write through. In particular the shared memory consistency for this modified design is proven. This results in the capability to ensure memory consistence for any access by simply enabling the write through policy.

Acknowledgments

In this chapter I would like to express my gratitude to some people, whose support helped me to get this work done.

First and foremost, I want to thank my supervisor Prof. Paul without whose support, ideas and advice this thesis would not have been created.
The greatest thanks goes to my partner, who not only endured my bad mood, whenever something did not progress as wished, but also was a great help for the editing work. He struggled through the whole thesis repeatedly, even though it is not his favorite field of science.
To my family who tolerated the miss of some phone calls and never put me under pressure.
And finally thanks to my colleagues who helped, if some formalisms tried to flee my thoughts.

Contents

1 Introduction

Multi-core processors and caches are an integral part of each modern computer - sold at any electronics store. Such computers consist of multiple components, e.g. multiprocessors, different levels of caches, main memory and hard disks.

From the processors point of view they seem to access a single shared memory. But in reality the data may be physically distributed among multiple cache and memory layers and multiple data copies of the same main memory location may exist. These copies can be shared among the various caches. Furthermore each processor has a small, fast, but expensive local cache only accessed by itself. The processor simply accesses the data of its own cache for reads and writes. If the cache does not contain the requested data, it will fetch the respective data from other caches or the main memory. For this, the caches and main memory need to communicate with each other over a set of shared communication wires called *bus*. This behavior is usually organized by a *cache coherence protocol* [6]. Such a protocol provides the necessary mechanisms and ensures certain properties, i.e. the safety condition of data consistency.

Now if the cache data is modified by a processor, these changes must eventually be reflected in the main memory. There exist different policies, when memory is updated with the recent data. In the so called *write back* mode, the modified data entry is only forwarded to the main memory when it is evicted from the cache. This results in high performance, as the main memory is accessed as little as possible. But it has the drawback that the main memory cannot be used as a consistent storage place. Moreover, it is common practice in system architecture to use the main memory as buffer between the I/O devices and the processors. In the favor of memory consistency, the *write through* policy has been invented. This mode issues a memory update, whenever the cache data is modified. Without support of the write through policy for inputs and outputs one can encounter the stale data problem [3, 4]. For this reason, usually a combination of write through and write back caches is used for recent computer architectures [7].

It is desirable that the correctness for such cache models is verified. In other words it has to be verified that the cache protocol indeed satisfies certain safety conditions such as data consistency. It is the responsibility of the hardware or software to enforce data consistency. But existing automated verification techniques for cache coherence protocols only consider sequential atomic memory accesses. This means even local accesses on different caches are assumed to be performed in an arbitrary sequential order. Protocols, modeling this behavior, are referred as *atomic protocol*s. A survey of verification methods for such atomic protocols is presented by [11]. Obviously, an atomic protocol should not be implement literally as it is, namely in a sequential way. Because then, the memory system would be a sequential bottleneck. But after all, the purpose of multiprocessors is to gain speed by maximized parallelism. Therefore we study the shared memory hardware implementing atomic protocols in a parallel way. Such a shared memory hardware supports parallel memory accesses, as long as they are not in conflict, i.e. if all accesses are reads or the memory addresses are different. To the best of our knowledge only two such designs were published in open literature [16, 8].

One of these designs is the (undocumented) open-source design of a 64-bit (chip multi-threaded) microprocessor, published by OpenSPARC™[16]. Another multi-core MIPS design was studied by the chair of computer architecture and parallel computing at Saarbrücken University. The study [8] features a gate level implementation of pipelined processors with a shared memory system implementing an atomic MOESI protocol. This approach is enhanced by the specification of a MIPS-86 architecture, resembling models of modern x86 architectures, by [12].

But still, the presented concept [8] only supports the write back memory mode. The goal of this thesis is to extend this design to provide the feature of not only the write back but additionally the write through policy. For the verification of this modified design a simulation relation to an atomic protocol is established. For this purpose it is necessary to adapt the gate level design of the shared memory hardware, as well as the atomic protocol specification.

With this additional feature memory can become a consistent storage place by enabling the write through mode. This provides the possibility to prevent the mentioned stale data problem with minimum interfering. Moreover, it models the behavior of recent computer architectures, which use combinations of write through and write back.

The thesis is structured as follows. At first in Chapter 2, an overview of the mechanisms used in this thesis is provided. The atomic MOESI protocol [15] is introduced in Chapter 3. In particular, the necessary modifications to recognize the write through policy are presented. Moreover the impact of the write through mode on the realization of the different operations (read, CAS = compare and swap, write) is specified. Later on in Chapter 5, this atomic protocol is used to establish a simulation relation to the shared memory design, in order to provide a correctness proof for the hardware.

Once the atomic approach is introduced, the gate level design of the shared memory hardware is presented in Chapter 4. In particular a multi-core design with two layers of memory is considered - one level of caches and the main memory. Each processor has a data and an instruction cache. For any memory request the processor addresses the respective local cache. A processor request may require communication with other caches or the main memory. This behavior is implemented in [8] with the help of two control automata. These automata model the interaction between the caches and main memory. One automaton is used to model the behavior as *master*, whose processor request is processed. The other automaton models the behavior as *slave*, who reacts to the request of another processor. These automata need to be extended by additional states to implement the behavior for the write through policy. Furthermore, the model is extended to inform the other caches, when the write policy changes. That way, the write mode is consistent in all caches.

Last but not least, a 'paper-and-pencil' correctness proof is provided in Chapter 5. This proof provides the building plan for a formal verification, i.e. machine readable proofs. In particular the proof states all necessary properties to show correct behavior with respect to the atomic protocol. Furthermore *sequential consistency* for the shared memory system is demonstrated. This means the answers of reads in the presented gate level design are the same as if the operations of all processors were executed in some sequential order. For each processor, this order is consistent with the local order, which is specified by the executed program.

Then in Chapter 6, the achievements of this thesis are summarized and finally possible improvements and open research questions are presented in Chapter 7.

2 Background

This thesis treats the model of a multi-core MIPS machine (MIPS-86) as introduced in [12, 13] and [8]. It contains several hardware models, which will be used in the course of this thesis. In particular, in [8] a multi-core processor model of a pipelined machine with caches and shared memory is presented.

That book is the base for this thesis, as it introduces all necessary concepts like hardware and cache models. Moreover, it serves as the main reference of this whole study. In the following sections, a short overview of these basic elements is given. This will provide the reader with the necessary background to grasp the modifications of the cache model introduced in the later chapters.

2.1 Caches

Almost any computer design uses a memory hierarchy to gain the best possible ratio between performance and cost. There are different levels of memory between the on-chip register files - having high costs but elevated performance - and the off-chip main memory - with poor performance but lower costs. The levels close to the processor are called *caches*.

When working with a multi-core design, caches can increase the degree of parallelization. They allow the processors to read the same data in parallel or to work locally on the own cache independently, as long as the entry is not used by any other processor.

Each such cache has three memories (RAMs) *i* a *state* memory, a *tag* memory and a *data* memory component. A single entry of the cache - containing each of these components - is called *cache line*. The data part contains the memory lines - the byte values of certain addresses which were transfered from the main memory into the cache. If these data bytes in the cache are modified and the modification is not yet passed to the main memory, the respective cache line is called *dirty*.

The state indicates whether a cache line is valid. In the next Section 2.1.1, we will see that the state may contain further information as well. In particular for the presented design, the state component gives some indication whether a cache line is modified, shared among other caches or consistent with the main memory.

A tag is necessary to identify the corresponding memory address for each cache line. For each memory address a set of possible cache locations is specified. Whenever memory data is copied to the cache, one of these locations is used. There exist different possibilities to associate a set of cache lines with a memory address. These mapping strategies can basically be grouped as

- *direct mapped* cache. The cache location is determined by means of computing the memory address modulo the cache size. This mapping concludes in exactly one possible cache location per memory address.

- *k-way associative* cache. For each referenced cache address there are k many direct mapped caches (k ways), which can be used for a certain memory address resulting in k possible cache locations pooled as one set. Now, if ℓ bits of the line address are used as cache line address, this will result in 2^ℓ sets of cache locations. In order to decide if the requested data is held in a certain way, a hit signal is computed out of the state and tag of the respective cache line.

- *fully associative* cache. A special case of a k-way associative cache, i.e. a line address can be stored at any cache location. To be exact, the set of possible cache locations exactly consists of all existent cache entries. In other words, each cache line is held in a separate way.

2.1.1 Abstract Caches

In [8] when modeling a cache coherence protocol the authors try to avoid the restriction to a specific mapping strategy and instead use abstract caches. As the name suggests, each of the previously listed cache types can be abstracted to an abstract cache model *aca*. How that can be done is shown in detail in [8, Section 8.1]. The set of all possible abstract cache configurations is denoted by K_{aca}. The abstract cache configuration *aca* encapsulates for each cache line two components.

- the data part $aca.data : \mathbb{B}^{29} \to \mathbb{B}^{64}$, the value which was written/read from the memory.

- the cache state $aca.s : \mathbb{B}^{29} \to S$, stating if the cache line is invalid, modified or distributed among multiple caches.

S is a set of the binary encoded possible states for a cache line as presented in Table 2.1. The subsequent chapters use the 5 states of the MOESI protocol from [15] to express the state of a cache line.

In this MOESI protocol, the state I indicates that the data within the cache line is *invalid* or meaningless. Meanwhile for cache states different from I the data is considered *valid*. Whenever a cache line is modified and not forwarded to the memory, the cache line becomes dirty. The resulting state is modified (M), if the cache line is exclusively contained in this cache and owned (O) otherwise. If the cache state $aca.s(a)$ is exclusive or shared but was not modified (i.e. there is no cache with owned state for a), the cache line is *clean* meaning the cache and main memory data are consistent for this line address.

s	synonym	name
10000	M	modified
01000	O	owned
00100	E	exclusive
00010	S	shared
00001	I	invalid

Table 2.1: A listing of the MOESI cache states $s \in S$

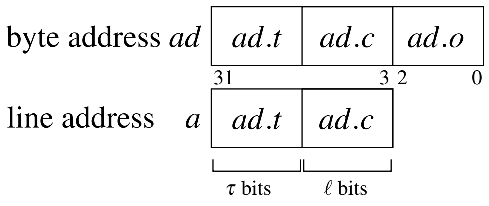


Figure 2.2: The decomposition of a 32 bit byte address *ad* and a 29 bit line address *a*.

A byte address *ad* with a length of 32 bits is composed of 29 bits for the line address and 3 bits offset, as each cache line contains 8 bytes of data. Furthermore the line address *a* can be decomposed in τ bits for the tag and a cache line address of ℓ bits to address the RAMs constituting the cache. This decomposition is depicted in Figure 2.2.

2.1.2 Allocation Policies

An *allocation policy* specifies when data is brought into the cache and when it remains in the main memory only. The allocation policy distinguishes between read and write accesses. Anyway, data is copied from main memory to the cache, if the referenced data is in none of the caches (*cache miss* in all caches). In that case the whole memory line is written to the cache. For a *read miss* (cache miss for a read access) the data is always brought into the cache. Meanwhile for *write misses* different policies exist.

- *Write Invalidate:* The data RAM of the cache is never updated by a write access but only by read misses. Moreover in case of a cache hit the write operation invalidates the cache line.
- *Read Allocate:* The cache is only touched in case of a read miss or a *write hit*. In other words if the accessed line is already cached, the data RAM is updated. But otherwise the write is only performed on the main memory and the line is not transfered into the cache.
- *Write Allocate:* The data RAM is always updated by a write access or in

case of a read miss. If a cache miss occurs, the line is copied to the cache and updated by the write access afterwards. That way, whenever any cache miss happens, the referenced line is transfered to the cache.

The underlying design [8] uses the write allocate policy. Therefore in the course of this thesis the same policy is used.

2.1.3 Memory Write Modes

All levels of the memory hierarchy need to comply with a *memory consistency model*, which defines the rules for the order and visibility of memory updates. To make a memory update visible (i.e. to external devices), an access has to update all instances of the accessed data. In particular, the main memory has to be updated.

The main memory is quite slow compared to a cache. An access to the main memory results in a performance loss, since the processor has to wait for the memory even longer, whenever any data is modified. This problem increases if multiple processors are used, as the memory can only be accessed by one processor at a time. Consequently, other processors have to wait for the completion of other main memory accesses, before they can proceed with their own (write) access. Inspired by this subject matter, different *write policies* (also referred as write modes) have been invented [14]. In particular, the issues regarding performance, reliability and complexity have to be taken into account, whenever a write policy is chosen in the system design phase.

The following write policies exist:

Write Through
Whenever a write operation is applied to the cache, the main memory is updated as well. Accordingly, the cache and memory data are always consistent. As a disadvantage, the performance suffers seriously. Nevertheless, this policy may be necessary, e.g. to write data to devices, for logging mechanisms to restore a consistent state in databases or simply to prevent data loss in case of a power outage or failure.

Write Back
In contrast to the previous policy, the main memory is only updated at certain occasions. Such an occasion is the *eviction* (when a cache location is currently occupied for another memory address) of a dirty cache line or a *flush* access (invalidation of a cache line). A possible implementation is that on every write hit the cache data is updated and a so called dirty flag is set. This flag indicates that the main memory and the cache data memory are not consistent any more for this address. Therefore when a flush access is issued, the main memory has to be updated with the cache data. In the MOESI protocol this is the case, if a modified (M) or owned (O) cache line is flushed.

The cache model from [8] is currently working in write back mode only. In the course of this thesis, we will see the necessary modifications to support both write modes. Thus the resulting memory model features the capability to make a cache line consistent to the main memory by use of the write through policy.

2.2 Memory Systems

Memory is visible for the user as a line addressable RAM. The configurations are mappings

$$m : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$$

Furthermore, K_m denotes the set of all possible memory configurations.

In fact, much more complicated architectures may be used in the underlying implementation, e.g. caches, TLB's or store buffers. These components are hidden from the user by the abstract memory view m. This thesis uses a memory system consisting of two components - the main memory and a number of abstract caches. Formally the memory system configuration ms contains

- $ms.mm : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$, line addressable memory
- $ms.aca : [0 : P-1] \rightarrow K_{aca}$, a sequence of abstract cache configurations

The set of all possible configurations is denoted by K_{ms}. Moreover the data parts of the various caches are always kept consistent. Therefore they maintain the following invariant:

$$ms.aca(i).s(a) \neq I \wedge ms.aca(j).s(a) \neq I \rightarrow ms.aca(i).data(a) = ms.aca(j).data(a)$$

The value of the abstracted memory $m(ms)$ for any address is defined to be the data of a cache aca in case of a cache hit and the content of the main memory mm otherwise.

$$m(ms)(a) = \begin{cases} ms.aca(i).data(a) & : ms.aca(i).s(a) \neq I \\ ms.mm(a) & : \text{otherwise} \end{cases}$$

memory system slices

A *memory system slice* $\Pi(ms,a)$ summarizes all entries for an address a. In particular it contains the main memory line $ms.mm(a)$ and the cache contents $ms.aca(i).X(a)$ for $X \in \{data, s\}$.

$$\begin{aligned} \Pi(ms,a) = \; & (ms.aca(0).data(a), ms.aca(0).s(a), \\ & \ldots, \\ & ms.aca(P-1).data(a), ms.aca(P-1).s(a), \\ & mm(a)) \end{aligned}$$

accesses and access sequences

A memory is accessed sequentially. Thus, only one processor can access the same

component	description
$acc.a[31:3]$	the line address accessed by the processor
$acc.data[63:0]$	the input data to update a memory for a write or CAS operation
$acc.cdata[31:0]$	the comparison data for a CAS access
$acc.bw[7:0]$	the byte write signals for a write or CAS access
$acc.w$	write enable signal, has value one in case of a write access
$acc.r$	the read signal, enabled in case of a read access
$acc.cas$	the CAS signal, is true in case of a CAS access
$acc.f$	the flush request signal, if enabled the cache line is written back to the main memory and invalidated afterwards

Table 2.3: A listing of the access components

memory at the same time. For the caches this is no problem, since each processor i has its own cache $aca(i)$. This cache is not accessed by any other processor $j \neq i$. As a result, multiple processors may access their cache in parallel, if the accesses are not in conflict with each other, i.e. the cache line is accessed for a read operation or it is exclusive in this cache. But for the main memory or if accesses to the caches are in conflict, this condition has to be enforced by the cache communication protocol. Therefore, memory systems can be accessed sequentially or in parallel.

The components of an access acc of a set of accesses $acc \in K_{acc}$ are listed in Table 2.3. For CAS accesses acc the predicate $test(acc,d)$ denotes if the comparison data $acc.cdata$ is equal to the upper or lower half of the data $d \in \mathbb{B}^{64}$ with respect to the byte write signal $acc.bw[0]$.

$$test(acc,d) \equiv acc.cdata = \begin{cases} d[63:32] & : \neg acc.bw[0] \\ d[31:0] & : acc.bw[0] \end{cases}$$

In the remainder of this thesis an additional component is necessary to determine the write mode of an access , namely $acc.m$. It stores the mode bit from the special purpose register of the accessing processor.

Depending on whether the memory system is accessed sequentially or in parallel, there are three kinds of access sequences and all are denoted by acc:

sequential access sequences

infinite case

$$acc : \mathbb{N} \rightarrow K_{acc}$$

finite case for n many accesses

$$acc : [0 : n-1] \rightarrow K_{acc}$$

multi-port access sequences

$$acc : [0 : P-1] \times \mathbb{N} \rightarrow K_{acc}$$

here P is the number of processors and $acc(i,k)$ denotes the access k to cache port i.

memory transition for a single access
The memory update function δ_M specifies the semantics of a single access acc applied to a memory m.

$$\delta_M : K_m \times K_{acc} \ \rightarrow \ K_m$$

The output function $dataout$ determines the answers for read or CAS accesses.

$$dataout(m,acc) \in \mathbb{B}^{64}$$

It is defined as

$$acc.r \vee acc.cas \rightarrow dataout(m,acc) = m(acc.a)$$

The next configuration m' for an access acc on a memory m is determined by the memory update function

$$m' = \delta_M(m,acc)$$

A multi-bank memory consists of multiple banks, which can be accessed individually. Each bank contains one byte of data. The function $byte(i,x)$ extracts the i-th byte from a bit-string $x \in \mathbb{B}^{8k}$. Let $i \in [0 : k-1]$:

$$byte(i,x) = x[8 \cdot i + 7 : 8 \cdot i]$$

The byte-wise modification of a multi-bank memory is specified as

$$byte(i, modify(x,y,bw)) = \begin{cases} byte(i,y) & : bw[i] = 1 \\ byte(i,x) & : bw[i] = 0 \end{cases}$$

With help of these definitions the update function of a memory m for any address a is defined as

$$m'(a) = \begin{cases} modify(m(a), acc.data, acc.bw) & : acc.a = a \wedge (acc.w \\ & \quad \vee acc.cas \wedge test(acc, m(acc.a))) \\ m(a) & : \text{otherwise} \end{cases}$$

Now, let us consider sequential access sequences acc of accesses. Then, the change of the memory state and the corresponding outputs $dataout[i]$ are defined as

$$\Delta_M^0(m,acc) = m$$
$$\Delta_M^{i+1}(m,acc) = \delta_M(\Delta_M^i(m,acc), acc[i])$$
$$dataout(m,acc)[i] = dataout(\Delta_M^i(m,acc), acc[i])$$

iterated memory transitions for the atomic protocol
In [8] a similar notation as for δ_m is introduced for the iterated transitions of the atomic protocol. This notation is used in the correctness proof in order to relate the effects of accesses to the constructed shared memory hardware to the effects in the atomic protocol.

Let us consider a sequential access sequence acc' with sequences is of ports of memory systems ms. Then for the step numbers n, the effect of n steps of the atomic protocol is defined by

$$\begin{aligned}\Delta_1^0(ms, acc', is) &= ms\\ \Delta_1^{n+1}(ms, acc', is) &= \delta_1(\Delta_1^n(ms, acc', is), acc'(n), is(n))\\ dataout1(ms, acc', is, n) &= dataout1(\Delta_1^n(ms, acc', is), acc'(n), is(n))\end{aligned}$$

δ_1 is the transition function and $dataout1$ the output function of the atomic MOESI protocol, which are defined in Chapter 3.

Lemma 2.1 (decomposition of 1 step accesses [8, Lemma 8.8])
Let $ms' = \Delta_1^x(ms, acc', is)$. Then

$$\Delta_1^{x+y}(ms, acc', is) = \Delta_1^y(ms', acc'[x : x+y-1], is[x : x+y-1])$$

Lemma 2.2 (memory abstraction 1 step iterative [8, Lemma 8.11])

$$m(\Delta_1^y(ms, acc', is')) = \Delta_M^y(m(ms), acc'))\ .$$

This notation becomes useful in the later correctness proof in Section 5.8.

2.3 Hardware Model

When constructing hardware the *digital model* is considered traditionally. A digital model is an abstraction from the reality. In particular, the signals are binary-valued and time is counted in hardware cycles.

But in reality, registers and gates need a certain time to change their value (called *propagation delay*). This delay can vary between the different hardware components. The minimal possible delay until the respective value may have changed, is called *minimal propagation delay*. After the *maximal propagation delay* passed, the output has definitively settled to its aimed value.

Signals may have either a digital value or an undefined value Ω. Such values can arise due to any kind of violation of the stability conditions (*metastability*). Another reason might be that the register value is currently changed due to regular clocking. Accordingly, a circuit signal y is a function

$$y : \mathbb{R} \to \{0, 1, \Omega\}$$

These reality based considerations are treated in the *detailed hardware model*. It was shown in [8] that under certain conditions the digital model is an abstraction of the detailed model. In particular, this is the case if certain timing constrains are maintained. *Tristate drivers* (Section 2.4.2) and the main memory can only be modeled in the detailed hardware model. This is explained by the fact that the absence of *glitches* (unintended transition before the signal settles to its intended value) and contention have to be proven. The issue of glitches was dealt with in [8] by introducing a register stage to the control drivers. In other words, the enable signal for a tristate driver is determined by the output of a *set-clear flip-flop*, as described in the later Section 2.4.2.

2.4 Drivers and Multi-Core Communication

In order to analyze the communication between caches of multi-core systems, the respective buses and drivers have to be investigated. The driver determines if the respective output signals to a bus are enabled or disabled.

There exist different bus types for various purposes. For example all caches need to be connected to the main memory to allow write or read operations. As mentioned in Section 2.2, only one access can be applied to the memory at once. Tristate drivers (see Section 2.4.2) are used to implement this behavior.

Meanwhile, caches need to communicate among each other as well. In particular cache communication is necessary in case of a cache miss to decide if *data intervention* (fetching the line from other caches) is possible or the main memory has to be accessed. This is accomplished by *open collector drivers* (see Section 2.4.1), which allow multiple enabled drivers.

2.4.1 Open Collector Driver (OC-drivers)

The inputs of an open collector driver determine, whether it has to be enabled or disabled. In particular, if the input is 0 the driver puts the signal 0 on the bus. Otherwise if the signal is 1, the driver is disabled and produces an output value between 0 and 1, which is called the *high impedance state* (Z). Then, drivers y can compute the digital value 0, a value Ω (only for the detailed model) or the high impedance state Z.

$$y : \mathbb{R} \rightarrow \{0, \Omega, Z\}$$

Let us assume the timing constrains for correct behavior are maintained. In particular, let us consider time intervals - at least the size of the maximal propagation delay - during which the input signals are binary stable values. Then the possible outputs of a driver y are

$$y(t) = \begin{cases} 0 & : yin(t) = 0 \\ Z & : yin(t) = 1 \end{cases}$$

Multiple open collector drivers y_i with inputs $y_i in$ are connected to an open collector bus b. In the physical design a *pull-up resistor* is used to drive the digital value 1 on the bus, if all the connected drivers are disabled. The value on the bus in the digital model is computed as:

$$b(t) = \begin{cases} 0 & : \exists i \;\; y_i(t) = 0 \\ 1 & : \forall i \;\; y_i(t) = Z \end{cases}$$

In other words, the AND function of the driver inputs is computed, if the digital model is considered.

$$b(t) = \bigwedge y_i in(t)$$

Using de Morgan's law an open collector bus can be used to compute the negated OR $\neg b$ of signals u_i - connected by an OC driver to the bus b.

$$b(t) = \bigwedge \neg u_i = \neg \left(\bigvee u_i \right)$$

Later on in Chapter 4, this is used to transmit the cache protocol control signals on the bus, e.g. the protocol signals for a cache hit or data intervention.

2.4.2 Tristate Drivers

In contrast to open collector drivers, the tristate drivers are controlled by an output enable signal yoe, which determines whether the driver is enabled or disabled. If the output enable signal is active, the input signal yin is propagated to the output y. Thus the output signal of the tristate driver is defined by

$$y(yin, yoe) = \begin{cases} yin & : yoe = 1 \\ Z & : yoe = 0 \end{cases}$$

Just like open collector drivers, tristate drivers can be connected to a *tristate bus*. A tristate bus allows only one driver to output a signal different from Z. Maintaining this invariant, the bus value $b(t)$ is derived as

$$b(t) = \begin{cases} y_i(t) & : \exists i \;\; y_i(t) \neq Z \\ Z & : \text{otherwise} \end{cases}$$

control of a tristate bus
It is possible to construct self destructing hardware due to bus contention [8, Section 3.5.5] even if contention is absent in the digital model. In order to avoid this issue the output enable signal $y_i oe$ is generated by a *set-clear flip-flop*, which is controlled by the signals $y_i oeset$ and $y_i oeclr$ (displayed in Figure 2.4) and when a driver is turned off in cycle t another driver is only turned on in cycle $t+2$.

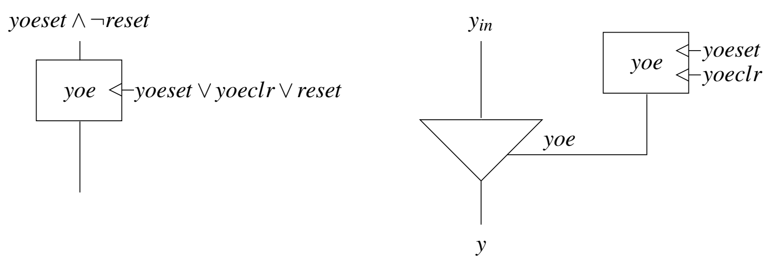


Figure 2.4: Implementation of the output enable signal *yoe* for a tristate driver y

The register y_ioe - to enable the driver y_i - has to be set one cycle before the output of y_i is actually on the bus and cleared in the last cycle during which the driver is sending on the bus. In other words, if one aims for an enabled driver during the cycles $t \in [a : b]$, the signal y_ioeset is activated at cycle $a-1$ and disabled with y_ioeclr in cycle b.

This principle will become important in the later cache protocol specification (Chapter 4) to determine which control signals of the bus have to be raised in which cycle. Because tristate drivers are used for the memory bus and the data broadcast of caches.

2.4.3 Main Memory Bus

As already mentioned the main memory bus is driven by tristate drivers. This bus consists of the following components (*wires*):

$b.d \in \mathbb{B}^{64}$ the bus data. For a write operation some cache puts a cache line on the bus to store it in the main memory. In case of a read operation the main memory provides the requested line.

$b.ad \in \mathbb{B}^{29}$ the line address, accessed due to a main memory operation.

$b.mmreq \in \mathbb{B}$. The request signal is raised to imply the execution of a main memory operation.

$b.mmw \in \mathbb{B}$ the main memory write signal, enabled for a write access.

$b.ack \in \mathbb{B}$. The main memory acknowledgment signal is activated in the last cycle of a main memory operation. In particular it indicates the completion of an access.

A system - designed to guarantee k units access to the main memory - allows each unit to control the main memory bus components. Therefore each unit j has an input register Q_j to read data from the memory and a set of output registers a_j, d_j, mmw_j and $mmreq_j$.

These registers are connected with the tristate bus components. Note that both registers Q_j and d_j are connected to the data component of the bus, because *b.d* can be controlled from both, the main memory or the unit. In contrast to that, the components a, *mmw* and *mmreq* are mere inputs to the main memory.

2.5 Control Automata

Generally, control automata can be used to model the behavior of a program or in this particular case to control the communication over the bus.

The advantage of them is the capability to express behavior for varying time. Moreover, the resulting model is independent from the scale of the modeled system, e.g. the number of participants in a communication protocol.

In the *gate level design* (the implementation of the cache model) certain control signals are activated with respect to the specified cache coherence protocol from Chapter 3. These are derived with the help of two control automata - one to act as master in the protocol, whose request is processed by the cache and a slave automaton to serve the request of other processors. This section aims to give a short overview of control automata (also referred to as *finite state transducers*). For further detail the formal specification is presented by [8, Section 3.6].

For any control automaton, a set of states, a transition function and an output function is specified. Dependent on the current state and the input, the transition function determines the next state of the automaton. At each step, the automaton produces an output as specified by the output function.

A control automaton can be represented as a directed graph, as we will see in the later Chapter 4. The edge labels specify the conditions (input for the transition function) to enter the respective state. If the edge is not labeled, the automaton will proceed into the next state in the next hardware cycle independent of the input.

The control logic of the gate level design is realized with two control automata. The remaining hardware is referred to as *data paths*. The inputs for the control automata are determined by the data paths. The modeled design has to implement certain invariants. Among other things it has to ensure the absence of bus contention for multiple tristate drivers on the same bus.

2.6 Special Purpose Register

In program execution, there are various events (so called *exceptions*) requiring immediate attention. They are treated as 'interrupts', meaning that the typical control flow of the program is interrupted. These interrupts may range from internal interrupts like the detection of a malformed instruction right up to external interrupts issued by input/output devices.

When such an event signal is activated, an interrupt handler is called. This is a routine specifying how the respective exception is handled. Dependent on the

i	synonym	explanation
0	sr	status register (contains masks to enable/ disable maskable interrupts)
1	esr	exception status register
2	eca	exception cause register
3	epc	exception pc (address to return to after interrupt handling)
4	edata	exception data (contains effective address on pfls)
5	mode	mode register $\in \{0^{31}1, 0^{32}\}$

Table 2.5: Special Purpose Registers

return type the processor execution may repeat the interrupted instruction, abort it or continue with the next instruction after returning from the exception handler. Such interrupt service routines are treated in the literature [13, 2, 12, 10] and are thus not discussed in further detail here.

To implement this mechanism additional information needs to be stored (e.g. the return address or exception cause). This data is stored in the *Special Purpose Register* (SPR). The respective registers for the underlying MIPS machine are listed in Table 2.5. In the course of this thesis only the *mode* register is interesting, as later on it is used to store the current write policy for a memory operation. The remaining registers are not considered at all in this thesis. The SPR may contain additional registers for more complex machines like the MIPS-86 which further implements store buffers and memory management units (MMUs). As this sophisticated design is beyond the scope of this thesis, only the simplified basic machine in [8] is investigated.

3 Atomic MOESI Protocol with Write Through Policy

In this chapter an overview of the effect of accesses is given. In particular, the necessary specifications for the memory write mode write trough are presented in detail in Section 3.1. Following this, you see the behavior for a write policy change in Section 3.2. With this knowledge it is possible to modify the protocol tables of the atomic MOESI protocol (Section 3.3) and the respective switching functions, which specify the main steps of the protocol (Section 3.4). There are certain invariants, which have to be satisfied by the protocol. These are presented in Section 3.5. These sections are the base to formalize the algebraic specification of any memory operation in Section 3.6.

3.1 Memory Write Mode

The atomic MOESI protocol as presented in [8, Section 8.3] is currently only working with the memory write mode *write back*. Auxiliary definitions are necessary to additionally permit the write through policy. Moreover it is necessary to identify the write mode, which is applied to the currently processed access. Additionally, the place to store the corresponding write mode bit has to be specified.

Definition 3.1
The *write mode wm* of the current processor configuration c has value 1 in case of write policy write back (wb) and 0 for write through (wt).

$$\begin{aligned} wb(c) &\equiv wm(c) = 1 \\ wt(c) &\equiv wm(c) = 0 \\ wt(c) &\Leftrightarrow \neg wb(c) \end{aligned}$$

We have seen in Section 2.6 that the special purpose register *mode* has 31 unused bits. More precisely the last bit is used to determine, if the processor is currently in system or user mode. Therefore it is feasible to use the bit $mode[1]$ to store the value for the current write mode $wm(c)$.

$$c.spr(mode)[1] = wm(c)$$

Whenever the memory system is accessed by any access *acc*, this value is used to determine the write policy. The protocol will ensure that the memory is updated, if the cache line is dirty and $wt(c)$ is true. For this reason the new component $acc.m$ of an access contains the current write mode.

$$acc.m = spr(mode)[1] = wm(c)$$

The atomic MOESI protocol is a sequential protocol operating on a multiport memory system *ms*. The following two functions define the semantics:

1. the *transition function*

   $$\delta_1 : K_{ms} \times K_{acc} \times [0 : P-1] \rightarrow K_{ms}$$

   with $ms' = \delta_1(ms, acc, i)$. It computes the memory system configuration, after one access *acc* is applied to a cache port $i \in [0 : P-1]$ of the memory system *ms*.

2. the *output function*

   $$dataout1 : K_{ms} \times K_{acc} \times [0 : P-1] \rightarrow \mathbb{B}^{64}$$

   with $d = dataout1(ms, acc, i)$. This function specifies the output *d* for a read or CAS access *acc* at cache *i* of the memory system *ms*.

To ease readability, in the subsequent sections the following abbreviations are used:

$$mm' = ms'.mm$$
$$aca' = ms'.aca$$

The traditional MOESI states from Section 2.1.1 are extended by the state bit $wm = aca.s[5] \in \mathbb{B}$. This bit denotes the write mode of the last access to the respective cache line.

In particular, the cache line will be clean for $wm = 0$, because the last access was applied using the write through policy. On the other hand for $wm = 1$, the cache line is in write back mode and may be dirty. Accordingly, the new set of states for $wm \in \{0, 1\}$ is

$$S = \{wm00001, wm00010, wm00100, wm01000, wm1000\}$$

The write mode bit serves as an input to the dirty bit computation, since in write through mode the write access is applied to the main memory as well. In particular the value $wm = 1$ implies that a modified value is only in the cache and not updated in the memory. The remaining state bits $aca.s[4 : 0]$ are pictured in Table 3.1.

In the following sections an acronym for the MOESI cache state $s[4 : 0]$ is used. Let $X \in \{M, O, E, S, I\}$ be a MOESI state, then the notation $s = X$ is the abbreviation for $s[4 : 0] = X$. Similar for a set $Y \subseteq \{M, O, E, S, I\}$ the term $s \in Y$ is the abbreviation for $s[4 : 0] \in Y$.

$s[4:0]$	synonym	state name
10000	M	modified
01000	O	owned
00100	E	exclusive
00010	S	shared
00001	I	invalid

Table 3.1: The cache state bits $s[4:0]$ according to the MOESI protocol, with their synonyms and names

Definition 3.2
The next write mode $aca'(i).s(a).wm$ of cache port i at address $a = acc(i,k).a$ is defined in accordance to the current access $acc(i,k)$ to the respective cache line.

$$\begin{aligned} aca^0(i).s.wm &= 0 \\ aca'(i).s(a).wm &= aca'(i).s(a)[5] = \begin{cases} acc(i,k).m & : a = acc(i,k).a \\ aca(i).s(a).wm & : \text{otherwise} \end{cases} \end{aligned}$$

Initially all cache lines are invalid, meaning that the write mode will be determined on the first access. But, the initial state has to be defined to determine the value after reset. In particular, the write mode is initially set to value 0 (write through).

From now on, the following abbreviations are used with respect to the currently considered access *acc*:

$$\begin{aligned} wb &\equiv acc.m = 1 \\ wt &\equiv acc.m = 0 \\ wm &\equiv aca.s(acc.a).wm = 1 \\ wm(i)(a) &\equiv aca(i).s(a).wm \end{aligned}$$

If the considered access is clear from the context, the predicate *wm* is used as an abbreviation to indicate the write mode of a cache line. Otherwise, the arguments are specified explicitly using the predicate $wm(i)(a)$.

3.2 Memory Mode Switching

While an access in write through mode implies a clean cache state, any access in write back mode may yield a dirty state. For this reason, a memory update has to be ensured when the write policy for a potentially dirty line is changed to write through. That way any cache line in write through mode is clean after the cache access is completed.

The state transitions are specified in a way that the next transition will lead into a clean state after setting the mode to write through. In particular, the algebraic specification is modified, such that a cache write will always update the memory, if the write through policy is enabled. In fact, even for read accesses of modified or shared lines it is mandatory to update the memory with the cache line content in case of a write mode switch to write through.

Additionally, the mode bit in the caches has to be flipped, when changing the write policy. That is easily done in case of an exclusive or modified cache line. But in case of a shared or owned line it is necessary to change the write mode in all caches holding this line. Therefore, it is essential to extend the cache coherence protocol in order to ensure that the (slave) caches switch their write mode for the accessed cache line.

For global write and CAS accesses, an interactive protocol between the master and slaves is executed. In Section 3.4 we will see the necessary modifications to broadcast the current write mode with help of the *new mode signal* $nm = acc.m$. This protocol has three phases, because the slave response is necessary to process the access. But read accesses to shared lines are served locally in the underlying model, meaning no slave response is required. Therefore an additional two phase protocol is introduced for this case. The *local write mode switch* predicate $lwms$ is computed to determine if the write mode of a cache line is changing. Then the *write mode switch* signal wms is raised to inform the slaves, if they have to change their write mode. This protocol signal is only used for read hits of shared or owned cache lines.

Definition 3.3
The predicate $lwms$ indicates a write mode switch. In particular, the write mode of the current access differs from the one for the addressed cache line.

$$lwms \;\equiv\; acc.m \neq wm$$

If the protocol signal wms is 1, all slave caches with a cache hit will change their write mode for this line to the new value, i.e. the inverted current write mode of the cache line. As a matter of fact, all slaves j with valid data for this line ($aca(j).s(a) \neq I$) invert their write mode.

$$aca'(j).s(a)[5] \;=\; \neg wm(j)(a) = acc.m$$

This gives us an additional case for the slave transitions, as pictured in Table 3.2, when only the master protocol signal wms is active, because a read access to a shared cache line is executed. The MOESI state in the slave caches stays unchanged. But if the signal wms is raised, the cache will flip the current write mode for the cache line address on the bus. This behavior is modeled in the algebraic specification in the later Section 3.6. As this issue only occurs for shared and owned lines, the slave cannot be in exclusive or modified state for this cache line. Hence, the slave state transition is not defined for these states (marked by table entry '-').

There are two possibilities for the next state of an owned cache line. We know that the line is clean after the read access. Because either the write policy changes to write through (hence the memory is updated) or it was in write through mode before (and therefore clean) and the read access did not modify the cache data. Therefore it would be possible to set the next state to shared instead of owned. For shared cache lines no additional memory access is necessary, when the line is flushed. But this approach has the disadvantage that the line is not fetched from the owner cache, but from the main memory in case of a cache miss. This results in a poor performance for this case. For this reason the presented design keeps the owned state even if the cache line is clean.

slave state	$\overline{Ca}, \overline{im}, \overline{bc}, \overline{nm}, wms$ read hit and memory write mode switch
wm M	-
wm O	$\neg wm$ O
wm E	-
wm S	$\neg wm$ S
0 I	0 I

Table 3.2: The newly introduced slave state transition case considering a change of the memory write policy

3.3 Protocol Tables

Table 3.3 describes the effect of the different accesses taking the modifications of the previous section into account. Thereby, a CAS hit is denoted by *CAS+* and a miss by *CAS-*.

Table 3.3a shows the cache states when acting as master in the protocol. In particular, it specifies the new local state $aca'(i).s(a)[4:0]$ of the cache line at address $a = acc(i,k).a$, dependent on the *type* of the access $acc(i,k)$ (the columns of the table). The respective next state and protocol signals for a slave are listed in Table 3.3b.

The master protocol signals Ca, im and bc may be raised by the master, with respect to the local state of the cache line and the access type. The master signal *wms* indicates that the write policy of this access differs from the previous one. The signal *nm* is used to broadcast the new write mode in case of a global access.

As mentioned in the previous section, the cache line will always be clean if it is in write through mode. For this reason, the cache state is not modified (M) but exclusive (E) in that case. Compared to the original protocol [8], this constraint issues an additional condition in the master state transition. Especially the write mode of the current access (signal *wb*) is taken into account.

To summarize the changes of the protocol tables in [8]:

- The new master protocol signals *wms* broadcasts a write mode change. If

master	read	write	flush	CAS-	CAS+
M	wb?M:E	wb?M:E	I	wb?M:E	wb?M:E
O	wms O	Ca, im, bc, nm ch?O:(wb?M:E)	 I	wms O	Ca, im, bc, nm ch?O:(wb?M:E)
E	E	wb?M:E	I	E	wb?M:E
S	wms S	Ca, im, bc, nm ch?O:(wb?M:E)	 I	wms S	Ca, im, bc, nm ch?O:(wb?M:E)
I	Ca, nm ch?S:E	Ca, im, nm wb?M:E	 -	Ca, im, nm wb?M:E	

(a) Master state transitions

slave	$\overline{Ca}, \overline{im}, \overline{bc}, \overline{nm}, wms$ read hit & mode switch	$Ca, \overline{im}, \overline{bc}, nm, \overline{wms}$ read miss	$Ca, im, \overline{bc}, nm, \overline{wms}$ write or CAS miss	$Ca, im, bc, nm, \overline{wms}$ write or CAS hit
M	-	ch, di O	di I	-
O	 O	ch,di O	I di	ch S
E	-	ch,di S	di I	-
S	 S	ch S	 I	ch S
I	I	I	I	I

(b) Slave state transitions

Table 3.3: Protocol state transitions

this signal is activated the current write mode of all slaves with valid data is inverted.

- The new master protocol signal *nm* broadcasts the new write mode for a *global* access. Any slave *j* with valid data for the accessed cache line will update its write mode with this value $ca(j).s[5] = nm$.
- The additional case distinction (wb?M:E) sets the MOESI state to exclusive clean (E) if a write through access was applied and to modified (M) otherwise.

The protocol signals have the following meaning:

Ca, the master has the intention to cache the line $acc.a$ after processing the access.

im signals the intention of the master to modify the line (i.e. for a write or CAS+ access).

bc, the line will be broadcast by the master after the modification. This signal is activated for shared data after a write or CAS+ access.

wms, the write mode of the processed cache line is changing. This signal is only used for read hits.

nm, the new write mode for a global access.

$ch(j)$, the slave cache $aca(j)$ contains valid data for the requested line $acc.a$.

$di(j)$, the slave cache $aca(j)$ has the requested cache line and will put it on the bus.

The master protocol signals Ca, im, bc, nm, wms of cache i will be broadcast to all other caches (slaves) $j \neq i$. The OR function of all slave signals is computed and the result is made accessible to the master via an open collector bus.

$$ch = \bigvee_{j \neq i} ch(j),\ di = \bigvee_{j \neq i} di(j)$$

Dependent on these slave protocol signals the new state of the master can be determined. The notation $s?X : Y$ means, if the signal s is true, the new state is X and Y otherwise.

$$s?X : Y \ = \begin{cases} X & s \\ Y & \bar{s} \end{cases}$$

3.4 Switching Functions

As already mentioned, the master and slaves need to communicate over the bus on certain occasions. This behavior is modeled with the help of a protocol, which is composed of certain phases. These phases are specified with the switching functions presented in this section.

The protocol for cache misses, write and CAS accesses of shared lines is split into 3 phases (C1, C2, C3). Meanwhile, the protocol for a read hit with write mode switch consists of 2 phases (CS1 and CS2). Therefore, five sets of switching functions are extracted from the Tables 3.3.

The functions CS1 and CS2 are necessary due to the support of multiple write modes. Therefore they were not present in the original design of [8]. For the functions C1 and C2, only minor changes of the original protocol with write back mode are necessary. Only the additional status bit for the write mode has to be considered. In particular the function C1 additionally computes the new mode signal nm. This signal is used as input for the function C2 and determines the new write mode bit for the slave cache. But the third switching function (C3), requires more effort to be adapted to the new policy write through. Specifically, the additional case distinction (wb?M:E) has to be integrated.

To ease readability, some abbreviations regarding the type are used:

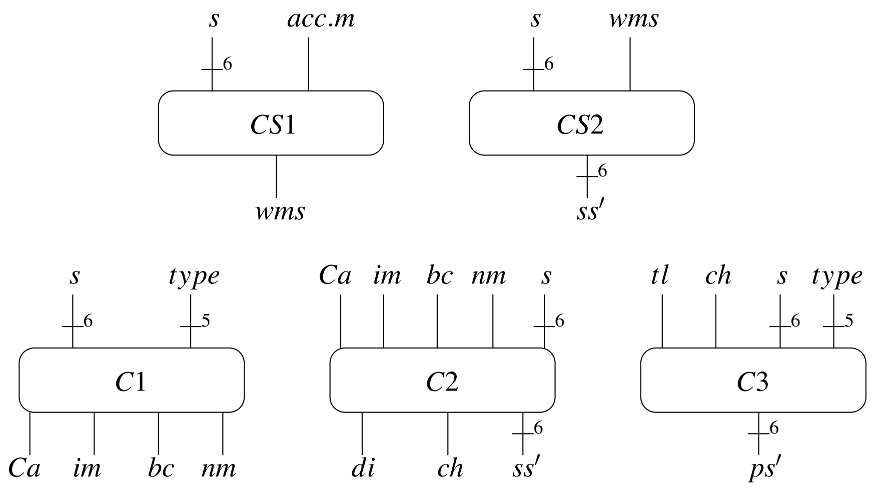


Figure 3.4: Symbols of the circuits CS1, CS2, C1, C2 and C3

w	the master is currently processing a write access
r	the master is currently processing a read access
cas	the master is currently processing a compare and swap access, meanwhile cas^+ denotes a hit and cas^- a miss respectively

$CS1$: This function computes the mode broadcast signal wms for a write mode switch.

$$CS1 : S \times \mathbb{B} \rightarrow \mathbb{B}$$

We have already seen the condition determining when the mode switch is broadcast to the slaves in Section 3.2. The master state $s \in S$ and the write mode of the access $acc.m$ are necessary to compute the predicate. The signal $CS1.wms$ can be computed with

$$CS1(s, acc.m).wms = 1 \;:\Longleftrightarrow\; s \in \{S, O\} \wedge lwms$$

Where, $lwms$ was defined with $lwms \equiv acc.m \neq s[5]$.

$C1$: This function computes the master protocol signals $C1.Ca, C1.im, C1.bc$ and $C1.nm$ dependent of the master state $s \in S$ and the type of the access $acc.type \in \mathbb{B}^5$ with

$$acc.type \;=\; (acc.r, acc.w, acc.cas, acc.f, acc.m)$$

$$C1 : S \times \mathbb{B}^5 \;\rightarrow\; \mathbb{B}^4$$

The component functions $C1.X, X \in \{Ca, im, bc\}$ are defined by looking up the corresponding entry in the master protocol Table 3.3a.

$$\forall X \in \{Ca, im, bc\} :\; C1(s, type).X \;=\; 1$$
$$:\Longleftrightarrow\; \text{master table entry } (s, type) \text{ contains } X$$

$$C1(s, type).nm = acc.m$$

In contrast to CS1, which broadcasts a write mode switch via signal *wms*, the function C1 transmits the write mode of the current access. Then the slaves set their cache state to this write mode *nm*.

*C*2: The slaves use this function in order to compute the response signals $C2.ch$, $C2.di$ and the next slave state (denoted with $C2.ss'$). The input arguments are the slave cache state $s \in S$ and the master protocol signals Ca, im, bc, nm.

$$C2 : S \times \mathbb{B}^4 \rightarrow \mathbb{B}^2 \times S$$

Again, the values for the component functions $C2.X$ with $X \in \{ch, di\}$ and the next state $C2.ss'$ are looked up in the protocol Table 3.3b.

$$\begin{aligned} \forall X \in \{ch, di\} : C2(s, Ca, im, bc, nm).X &= 1 \\ &:\Longleftrightarrow \text{ slave table entry } (s, Ca, im, bc, nm) \text{ contains } X \end{aligned}$$

$$\begin{aligned} C2(s, Ca, im, bc, nm).ss'[4:0] &= s' \\ &:\Longleftrightarrow \text{ slave table entry } (s, Ca, im, bc, nm) \text{ contains } s' \end{aligned}$$

In addition to that, the slave sets its write mode to the value transmitted by the master with signal *nm*.

$$C2(s, Ca, im, bc, nm).ss'[5] = nm$$

*CS*2: When using this function the master will read the value from its own cache entry, such that the slaves have no need to compute the signals *ch* and *di*. But it is still mandatory to update the write mode of the slave caches. The signal *wms* of the master protocol and the slave cache state *s* are necessary to compute the new cache state bits.

$$CS2 : S \times \mathbb{B} \rightarrow S$$

The next slave state $CS2.ss'$ is defined by

$$CS2(s, wms).ss' = \begin{cases} \neg s[5] \circ s[4:0] & : wms \\ s & : \neg wms \end{cases}$$

For $a, b \in \mathbb{B}^*$, the term $a \circ b$ denotes the concatenation of the bits of a with the bits of b. In Section 3.6, we will see that the function $CS2$ is only used if *wms* was raised.

$C3$: This function depends on the master state $s \in S$, the slave response ch, the result of the local CAS test $tl = test(acc, aca(i).data(acc.a))$ and the type of the access $acc.type$. Note that $acc.type$ contains the current memory write mode and $wb = acc.m$ holds. The function C3 computes the next state of the master (denoted with $C3.ps'$).

$$C3 : S \times \mathbb{B}^5 \times \mathbb{B} \times \mathbb{B} \rightarrow S$$

The function is defined by translation of the master protocol table. In particular, either the table entry yields a single MOESI state or a case distinction - depending on the slave response signal ch and the write mode wb - is done.

$$\begin{aligned}
& C3(s, type, ch, tl).ps' = acc.m \circ s' \\
& \quad :\Longleftrightarrow \text{ master table entry } (s, type, ch, tl) \text{ contains } s' \vee \exists s', s'', s''' : \\
& \qquad ch \wedge wb \wedge \text{master table entry } (s, type, ch, tl) \text{ contains} \\
& \qquad\qquad ch?s' : (wb?s'' : s''') \\
& \qquad \vee\, ch \wedge \overline{wb} \wedge \text{master table entry } (s, type, ch, tl) \text{ contains} \\
& \qquad\qquad ch?s' : (wb?s'' : s''') \\
& \qquad \vee\, \overline{ch} \wedge wb \wedge \text{master table entry } (s, type, ch, tl) \text{ contains} \\
& \qquad\qquad ch?s'' : (wb?s' : s''') \\
& \qquad \vee\, \overline{ch} \wedge \overline{wb} \wedge \text{master table entry } (s, type, ch, tl) \text{ contains} \\
& \qquad\qquad ch?s''' : (wb?s'' : s')
\end{aligned}$$

For the case of a CAS hit ($acc.cas = 1$ and $s \neq I$) the column CAS+ is chosen if $tl = 1$ and CAS- otherwise.

3.5 State Invariants

A cache line is called clean if the memory and cache content are the same. Accordingly, the predicate *clean* is defined.

Definition 3.4 (clean)
The predicate *clean* is true if the cache line content is the same as the memory content for the line.

$$clean(a) \equiv aca(i).data(a) = mm(a)$$

A number of invariants is necessary to ensure correct behavior of the cache coherence protocol:

Invariant 3.5 (single exclusive state)
The states E and M are exclusive, i.e. only one cache has meaningful data for this line and in other caches the line is invalid.

$$aca(i).s(a) \in \{E, M\} \wedge j \neq i \implies aca(j).s(a) = I$$

Invariant 3.6 (clean state)
An exclusive cache line (state E) is always clean. In other words the memory and cache content are identical for this line.

$$aca(i).s(a) = E \implies clean(a)$$

Invariant 3.7 (shared cache lines)
Shared lines, i.e. lines in state S are clean or they have an owner.

$$aca(i).s(a) = S \implies (clean(a) \vee \exists j \neq i : aca(j).s(a) = O)$$

Invariant 3.8 (same data for shared lines)
The data corresponding to nonexclusive cache lines is identical.

$$aca(i).s(a) = S \wedge aca(j).s(a) \in \{O,S\} \implies aca(i).data(a) = aca(j).data(a)$$

Invariant 3.9 (unique owner)
When a cache line is shared, in particular in state S or O, the line has to be invalid or non-exclusive in all other caches. Moreover at most one cache can have state O for the same line.

$$\begin{aligned} aca(i).s(a) = S \wedge j \neq i &\implies aca(j).s(a) \in \{I,O,S\} \\ aca(i).s(a) = O \wedge j \neq i &\implies aca(j).s(a) \in \{I,S\} \end{aligned}$$

These 5 invariants were used in [8]. Nevertheless, they are also applicable to the adapted protocol, which supports the write trough policy.

It is possible to prove the validity of these invariants with a 'paper-and-pencil' proof resulting in many case distinctions. Therefore one tends to use an automated verification tool for this task. For cache protocols various verification techniques exist [11, 5]. Respectively, there are numerous different tools [1, 9] and respective property languages one could use for this task.

For the Invariants 3.5 (single exclusive state) and 3.9 (unique owner) it is sufficient to apply a simple *reachability analysis*. In this analysis the states of the cache protocol are modeled with finite state machines and the state space is explored. The states $s_i \in \{M,O,E,S,I\}$ of each component (participating caches) are subsumed in a global state $(s_1,\dots,s_n)$ for all n caches. The analysis shows that global states $s_i = s_j = E$ or $s_i = s_j = O$ are not reachable for different caches $i \neq j$. Furthermore only combinations of invalid (I), shared (S) and owned (O) states are reachable for any state s_j with respect to a cache state $s_i \in \{S,O\}$.

For the remaining invariants, a more complex analysis is necessary. The protocol behavior has to be modeled. Then one can check that the properties for the invariants is indeed satisfied. For this purpose simulation tools like Merφ [1] or LTSA [9] are suitable. These tools work with the *enumeration method* as presented in [11]. The protocol components are abstracted by globally accessible state variables. This results in a set of cache states $(c_1,\dots,c_2)$ and a set of variable values $(v_1,\dots,v_q)$.

The model is implemented by translating the next-state transition rules of the protocol in the respective property language of the tool. It is important to take into account that at most one cache may have the control for the memory bus. Therefore, a locking mechanism is used for shared variables. The behavior of the arbiter is modeled accordingly. Now the invariant properties can be modeled in the tool (LTSA) and their validity can be verified.

Due to the introduced modifications regarding the write through policy one can conclude further invariants for the protocol:

Invariant 3.10 (clean write through)
After a memory update the state is clean. This is the case for valid cache lines after any access in write through mode. For all addresses a:

$$wm(i) = 0 \wedge aca(i).s(a) \neq I \implies clean(a)$$

Invariant 3.11 (same write policy)
The broadcasting protocol - issued for a write mode switch as introduced in Section 3.2 - ensures that all caches have an identical write mode for the same cache line.

$$aca(i).s(a) \in \{S, O\} \wedge aca(j).s(a) \neq I \implies wm(i)(a) = wm(j)(a)$$

The notation $sinv(ms)(a)$ is used to indicate the validity of the state invariants for a cache line address a and the memory system ms. The notation $SINV(t)$ is used to denote that the invariants hold up until a certain cycle t, while the hardware abstracted memory system $ms(h)$ is regarded. Then for all cycles $t' \in [0:t]$ the following properties hold:

$$\begin{aligned} sinv(ms) \; &:\Longleftrightarrow \; \forall a : sinv(ms)(a) \\ SINV(t) \; &:\Longleftrightarrow \; \forall t' \in [0:t] : sinv(ms(h^{t'})) \end{aligned}$$

Lemma 3.12 (invariants for invalid state [8, Lemma 8.7])
The invariants hold for a line which is invalid in all caches. This is particularly the case after a reset in cycle -1.

$$\forall a, i : aca(i).s(a) = 0I \implies sinv(ms)$$

3.6 Algebraic Specification

In the following section, the impact of an access acc at cache port i is investigated. This is done by considering the functions $ms' = \delta_1(ms, acc, i)$ and $d = dataout1(ms, acc, i)$. Only components that change due to the impact of the access acc are specified below. In that context the validity of the invariants from the previous section is assumed, before the access acc is processed, i.e. $sinv(ms)$ is assumed.

In the favor of readability the function arguments $ms.aca$ and $ms.m$ are only stated explicitly, if they are not apparent from the context. Additionally the line address $acc.a$ is denoted by a in the following sections, if applicable. Furthermore all components x of the access acc are abbreviated by $x = acc.x$.

A hit for an address a in an atomic abstract cache $aca(i)$ is defined as

$$hit(aca,a,i) \equiv aca(i).s(a) \neq I$$

Over the course of this section multiple definitions are used to discuss the different access types and cases:

$$\begin{aligned}
lread(aca,acc,i) &\equiv hit(aca,a,i) \wedge (r \vee cas \wedge \neg test(acc,aca(i).data(a))) \\
lwrite(aca,acc,i) &\equiv hit(aca,a,i) \wedge (w \vee cas \wedge test(acc,aca(i).data(a))) \\
&\quad \wedge aca(i).s(a) \in \{E,M\} \\
broadcastms(aca,acc,i) &\equiv lread \wedge lwms \wedge aca(i).s(a) \in \{O,S\} \\
rglobal'(aca,acc,i) &\equiv lread \wedge aca(i).s(a) = M \wedge lwms \wedge wt \\
wglobal'(aca.acc,i) &\equiv lwrite \wedge wt \\
rlocal(aca,acc,i) &\equiv lread \wedge \neg broadcastms(aca,acc,i) \\
&\quad \wedge \neg rglobal'(aca,acc,i) \\
wlocal(aca.acc,i) &\equiv lwrite \wedge wb
\end{aligned}$$

Here $lread$ and $lwrite$ are the accesses, which were treated locally in the original protocol. But due to the modifications for the write through policy additional cases occur:

- $broadcastms$: The write policy of a shared cache line is changed. In order to maintain Invariant 3.11 (same write policy) the write mode has to be changed in all caches holding this line. This issues an interactive protocol even for a read access, which could be handled locally before.
- $rglobal'$: Similar to the previous case, the read access cannot be served locally by the cache, if the write policy is changed from write back to write through. This is because the main memory needs to be updated for this case.
- $wglobal'$: Again the memory needs to be written. This time due to a write access with write through policy.

These access types can then be summarized as

- $local$: the access can be served by the cache locally and no communication with the remaining memory system is required.

- *global′*: summarizes the newly introduced access types which need either interactions with other caches due to a mode broadcast or an access to the main memory is required.

- *global*: any access which cannot be handled locally due to a cache miss or since the line - shared among multiple caches - is modified.

$$\begin{aligned}
local(aca,acc,i) &\equiv rlocal(aca,acc,i) \vee wlocal(aca,acc,i) \\
global'(aca,acc,i) &\equiv rglobal'(aca,acc,i) \vee broadcastms(aca,acc,i) \\
&\quad \vee wglobal'(aca,acc,i) \\
global(aca,acc,i) &\equiv \neg local(aca,acc,i) \wedge \neg global'(aca,acc,i) \wedge \neg f
\end{aligned}$$

The specification of the consecutive paragraphs has to fulfill certain criteria:

- maintaining the state invariants $sinv(ms')$ from Section 3.5.

- in particular maintaining Invariant 3.10 (clean write through) by issuing a memory update, if a modified cache line is set to write through.

- modeling the same behavior for the memory abstraction $m(ms)$ as the ordinary memory semantics, when an access acc is applied.

$$m(ms') = \delta_M(m(ms), acc)$$

- obtaining correct response signals d from the memory abstraction $m(ms)$ for read accesses.

$$dataout1(ms,acc,i) = dataout(m(ms),acc) = m(ms)(acc.a)$$

Flush

A flush access will invalidate the addressed cache line. Therefore it is necessary to ensure that the line is written to the main memory if it was modified. This is only the case, if the cache line is in write back mode ($wm = 1$) and in modified or owned state. Otherwise the cache line is already clean according to the invariants of the previous section. The only difference to [8] is the condition wm to indicate the write back mode.

$$\begin{aligned}
f \;\rightarrow\; & aca'(i).s(a) = I \wedge \\
& (wm \wedge aca(i).s \in \{M,O\} \;\rightarrow\; mm'(a) = aca(i).data(a))
\end{aligned}$$

Local Write Accesses

The local cache line - addressed by a - is updated and the state is set to M or E corresponding to the memory write mode.

$$\begin{aligned}
lwrite(aca,acc,i) \;\rightarrow&\\
aca'(i).data(a) =&\; modify(aca(i).data(a),data,bw)\wedge\\
aca'(i).s(a) =&\; C3(aca(i).s(a),acc.type,sprot.ch,\\
&\quad test(acc,aca(i).data(a))).ps')\\
=&\begin{cases}1M & : wlocal(aca,acc,i)\\ 0E & : wglobal'(aca,acc,i)\end{cases}\\
wglobal'(aca,acc,i) \;\rightarrow&\; mm'(a) = modify(aca(i).data(a),data,bw)
\end{aligned}$$

Thereby both access types $wlocal$ and $wglobal'$ require the predicate $lwrite$ to be true in their definitions and will issue the first set of transitions. Compared to [8], the case $0E$ for the MOESI state and the memory update transition due to a $wglobal'$ access are new.

Global Accesses

Any $global$ access is handled by an interactive cache protocol, which consists of three phases. First the master protocol signals $mprot$ are computed and put on the bus by the master. Next the slaves fetch the master protocol signals from the bus and compute the slave respond and the new slave state. These slave response signals are ORed together. The resulting slave protocol signals $sprot$ are then put on the bus. In the last step the master receives the slave protocol signals and computes the new master state. The respective equations for this protocol process are similar to [8]. The only alternation are the new master protocol signals wms and nm.

$$\begin{aligned}
global(aca,acc,i) \;&\rightarrow\\
mprot.(Ca,im,bc,nm) \;&=\; C1(aca(i).s(a),acc.type)\\
mprot.wms \;&=\; 0\\
\forall j: sprot(j) \;&=\; C2(aca(i).s(a),mprot.(Ca,im,bc,nm)).(ch,di)\\
sprot \;&=\; \bigvee_{j\neq i} sprot(j)\\
\forall j: aca'(j).s(a) \;&=\; \begin{cases}C3(aca(j).s(a),acc.type,sprot.ch,0).ps' & : i=j\\ C2(aca(j).s(a),mprot).ss' & : i\neq j\end{cases}
\end{aligned}$$

If the broadcast signal $mprot(i).bc$ is active for a write or CAS access, the modified data ($modify(aca(i).data(a),data,bw)$) is broadcast to the other caches via the bus.

On the other hand in case of a cache miss a slave puts the requested data on the bus if the data intervention signal di was raised. By the protocol tables and the state invariants follows that there can be at most one intervening slave j. If there is no data intervention, the data is fetched from the main memory. As presented in [8], this results in the following cases for the bus data *bdata*:

$$global(aca,acc,i) \rightarrow$$
$$bdata = \begin{cases} modify(aca(i).data(a),data,bw) & : mprot(i).bc \\ aca(j).data(a) & : sprot(j).di \\ mm(a) & : \text{otherwise} \end{cases}$$

All slave caches j holding valid data for the accessed address will signal a cache hit $sprot(j).ch$. These caches store the broadcast result, if signal $mprot(i).bc$ was activated by the master.

$$\forall j \neq i : global(aca,acc,i) \wedge mprot(i).bc \wedge sprot(j).ch$$
$$\rightarrow aca'(j).data(a) = bdata$$

We recall that a CAS access only modifies the data of a cache line, if the comparison data matches the value at the accessed address. This is indicated by a positive result for the function $test(acc,tdata)$. Hereby, *tdata* is the old value stored in cache i in case of a hit or the bus value otherwise. For a CAS hit the signal $mprot(i).bc$ is active, such that the test data is defined by

$$global(aca,acc,i) \wedge cas \rightarrow$$
$$tdata = \begin{cases} aca(i).data(a) & : mprot(i).bc \\ bdata & : \text{otherwise} \end{cases}$$

CAS misses and read accesses with a cache miss are referred as global reads.

$$rglobal(aca,acc,i) \equiv global(aca,acc,i) \wedge (r \vee cas \wedge \neg test(acc,tdata))$$

Whereas a global write is a CAS hit or a write access of a shared cache line.

$$wglobal(aca,acc,i) \equiv global(aca,acc,i) \wedge (w \vee cas \wedge test(acc,tdata))$$

For a global read access the data of the missing cache line will be directly copied from the bus. Until here the specification for global accesses was the same as in [8]. Additionally to these specifications, the main memory will be updated, if the current access conforms to the write through policy, but the cache line is dirty and was previously in write back mode. This case is implemented by the second equation.

$$rglobal(aca,acc,i) \rightarrow aca'(i).data(a) = bdata$$

$$\begin{array}{l} rglobal(aca,acc,i) \wedge wt \\ \wedge sprot(j).di \wedge wm(j)(a) \end{array} \rightarrow mm'(a) = aca'(i).data(a) = bdata$$

Note that the memory is updated by the slave cache j - which provides the data on a data intervention - because the master has no information of the last write mode *wm*. If no cache provides the data for the read miss, there exists no owner for the line. In this case the data is directly fetched from the main memory. This means the cache line is already clean.

For a global write the master modifies either the own cache data in case of a hit or the bus data otherwise. In the later case the broadcast signal $mprot(i).bc$ is off. In addition to [8], the modified value is written to the memory if the write through policy is enabled.

$$\begin{aligned} wglobal(aca,acc,i) &\rightarrow \\ aca'(i).data(a) &= \begin{cases} modify(aca(i).data(a),data,bw) & : mprot(i).bc \\ modify(bdata,data,bw) & : \text{otherwise} \end{cases} \\ wglobal(aca,acc,i) \wedge wt &\rightarrow mm'(a) = aca'(i).data(a) \end{aligned}$$

Read Accesses or CAS misses

We have to consider the specification for the previously mentioned special cases of the access types *broadcastms* and *rglobal′*. For this the different cases of a write policy change are distinguished.

Switching to Write Through:
Every cache line has to be clean, the next cycle after this change of the write policy. Therefore it is necessary to update the main memory with the cache data of the accessed cache line even in case of a read access.

This case gives us $wt \wedge wm = 1$. In other words the cache line is currently in write back mode and the access will issue a mode switch to write through. Then for read accesses or CAS misses applies

$$rglobal'(aca,acc,i) \vee broadcastms(aca,acc,i) \wedge wt \rightarrow mm'(a) = aca(i).data(a)$$

Switching Between any Write Mode:
Generally on a read access or CAS miss when changing the write policy, the bit for the write mode is inverted meanwhile the remaining state bits stay unchanged.

Now there are two possibilities, either the accessed cache line is exclusive to the master cache (state E or M) or it is shared (state S or O). In the first case only the local cache state is updated

$$\begin{aligned} &lread(aca,acc,i) \wedge lwms \wedge aca(i).s(a) \in \{M,E\} \\ &\quad \rightarrow aca'(i).s(a) = acc.m \circ E \end{aligned}$$

The next state will be exclusive (E) as either the line was in write through mode before, which implies the predicate *clean* and therefore conforms to state E. Or the previous write policy was write back, but a memory update was issued due to the mode switch to write through, as specified in the previous paragraph. Thus, we have always clean exclusive for the cache line after processing this access.

For a shared cache line, all caches - holding this line - have to update their write mode. Considering that, communication with the other caches is necessary. Therefore the broadcast protocol was introduced with the switching functions $CS1$ and $CS2$.

First the master i determines if the write policy was changed by computing the signal $mprot.wms$. Then the slaves j fetch the master protocol signals $mprot$ from the bus and compute their next states. Additionally the master state is modified as mentioned above.

$$
\begin{aligned}
&broadcastms(aca, acc, i) \rightarrow \\
&\quad mprot.(Ca, im, bc, nm) = 0 \\
&\qquad mprot.wms = CS1(aca(i).s(a), acc.m).wms \\
&\qquad \forall j : aca'(j).s(a) = \begin{cases} CS2(aca(j).s(a), wms) & : j \neq i \\ acc.m \circ aca(i).s(a)[4:0] & : j = i \end{cases}
\end{aligned}
$$

In Absence of a Mode Switch:
For this case, simply a local read can be applied, such that the cache state stays unchanged.

$$rlocal(aca, acc, i) \rightarrow aca'(i).s(a) = aca(i).s(a)$$

Answers of Reads or CAS Requests

For a read request or a CAS request the local cache data is returned in case of a hit or the bus data otherwise. This is the same formalism as in [8]:

$$r \vee cas \rightarrow dataout1(ms, acc, i) = \begin{cases} aca(i).data(a) & : hit(aca, a, i) \\ bdata & : \text{otherwise} \end{cases}$$

4 Gate Level Design

In this chapter the implementation of a shared memory design is introduced, which simulates the atomic protocol from the previous chapter. In this context we are going to see the data paths and respective control automata, which compute the control signals. For the realization of the specified MOESI protocol, multiple components like memory, processors and caches need to communicate with each other. For this reason, interfaces between these components are necessary.

4.1 Interfaces

The interfaces are only slightly different to the reference [8]. In particular the only differences are the additional signal *pmode* to indicate the write mode of the processor and two additional memory bus lines for the master protocol signals *wms* and *nm*. Nevertheless, the overall gate level design is provided for a better understanding of the topic.

There are two interfaces to consider

1. between processors p and their caches. This interface needs only signals for the interaction.
2. caches $ca(i)$ and the bus b. For this interface dedicated registers are used.

In the consecutive sections, for interface signals X and a cache i the abbreviation $X(i)$ is used for the cache component ca(i).X.

4.1.1 Processor and Cache Interface

Processor Signals to the Cache ($p \rightarrow ca$)

- $ca(i).preq$ - processor request signal,
- $ca(i).pdin$ - processor data to write in the cache,
- $ca(i).pcdin$ - compare data used for CAS requests,
- $ca(i).pa$ - line address to access,

- $ca(i).ptype = (ca(i).pr, ca(i).pw, ca(i).pcas)$ - the type of a cache request (read, write or CAS). For every request one of the type bits has to be set.
- $ca(i).bw[7:0]$ - byte write signals. In case of a read requests the byte write signals are disabled. In [8] CAS requests were defined on words, therefore only the first or last half of the byte write signals is active:

$$ca(i).preq \wedge ca(i).pr \rightarrow ca(i).bw = 0^8$$
$$ca(i).preq \wedge ca(i).pcas \rightarrow ca(i).bw \in \{0^4 1^4, 1^4 0^4\}.$$

- $ca(i).pmode$ - write mode of a request. The value 1 denotes write back policy $pmode = 1 \Leftrightarrow wb$ and the value 0 corresponds to write through $pmode = 0 \Leftrightarrow wt$.

Cache Signals to the Processor ($ca \rightarrow p$)

- $ca(i).mbusy$ - a control signal generated by the control automaton signaling that the memory system is busy,
- $ca(i).pdout$ - data output for a requested line.

4.1.2 Bus Interface of Caches

Dedicated Registers between the Bus and Caches ($ca \rightarrow b$)

- $ca(i).bdout$: cache data output - put on the bus by cache *i*,
- $ca(i).bdin$: cache data input - read from the bus,
- $ca(i).badout$: (master) cache address output - put on the bus by cache *i*,
- $ca(i).badin$: (slave) cache address input - read from the bus,
- $ca(i).mprotout$, $ca(i).sprotout$: master and slave protocol signals - put on the bus by cache *i*,
- $ca(i).mprotin$, $ca(i).sprotin$: slave and master protocol signals - read from the bus.

Memory bus

As described in Section 2.4 the memory bus consists of multiple tristate lines. The respective outputs to the bus are controlled via tristate drivers. These are the first three items in the listing of bus lines below. Therefore these are tristate lines, whereas the last component ($b.prot$) is implemented as an open collector bus. As a consequence only one processor can use the tristate lines (used for main memory

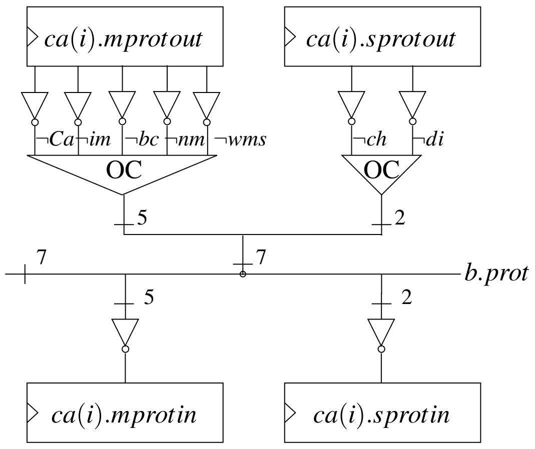


Figure 4.1: Open collector bus *b.prot* for the cache protocol signals

accesses). Therefore the control automata of Section 4.3 will ensure that only one master can access the main memory at the same time. Meanwhile multiple caches can communicate over the open collector bus at the same time.

- *b.d*: the data part, which is used to transmit a line to a cache or the main memory.
- *b.ad*: the memory line address,
- *b.mmreq*, *b.mmw*, *b.ack*: these lines are used to transmit the memory protocol signals.
- $b.prot[6:0] = b.mprot[4:0] \circ b.sprot[1:0]$: used for the transmission of the cache protocol signals.

The following synonyms for the cache protocol signals are used:

$$\begin{aligned} b.Ca &= b.mprot[4] = b.prot[6] \\ b.im &= b.mprot[3] = b.prot[5] \\ b.bc &= b.mprot[2] = b.prot[4] \\ b.nm &= b.mprot[1] = b.prot[3] \\ b.wms &= b.mprot[0] = b.prot[2] \\ b.ch &= b.sprot[1] = b.prot[1] \\ b.di &= b.sprot[0] = b.prot[0] \end{aligned}$$

The bus lines *b.wms* and *b.nm* could be combined to one line due to the fact that they are never used simultaneously in the specified protocol. But their function is more apparent if they are separate.

Protocol for Cache and Processor Communication

For the interaction between a processor *p* and its cache *ca* the previous listed processor signals are used. At the start of a request the processor activates the signal *preq*. This signal *preq* stays enabled until the request is completed. Thus it will not be disabled until the next cycle. The cache acknowledges the processor request by activating *mbusy*. This signal is then lowered in the last cycle of a request.

Following these rules we have for the first cycle *t* of an access:

$$\neg mbusy^{t-1} \wedge preq^{t}$$

and the last cycle $t' \geq t$ of an access:

$$\neg mbusy^{t'} \wedge preq^{t'}$$

In case of a local read hit it is possible to handle the request in one cycle. Therefore the signal *mbusy* is not raised at all, but only the processor request signal *preq* is activated and a new request can start immediately at the next cycle.

While the request is processed, the processor signals have to stay stable. In particular from activating *preq* until lowering *mbusy* the cache input signals have to be stable.

The input signals of cache $ca(i)$ at cycle *t* are defined as

$$cain(i,t) = \{pa, type, pbw, preq, pmode\} \cup \begin{cases} \{pdin\} & ca(i).pw^{t} \\ \{pdin, pcdin\} & ca(i).pcas^{t} \\ \emptyset & \text{otherwise} \end{cases}$$

Then the condition of stable inputs is formalized by

$$ca(i).preq^{t} \wedge ca(i).mbusy^{t} \wedge X \in cain(i,t) \;\rightarrow\; ca(i).X^{t+1} = ca(i).X^{t}$$

4.2 Data Paths of the Cache RAMs

The data paths for the tag, data and state RAMs are presented in the Figures 4.2, 4.3 and 4.4. The necessary control signals are generated by the control automata (presented in Section 4.3).

The cache components can be addressed in two possible ways, namely either for a protocol interaction with other caches via the bus or from the local processor. Then, an access from the processor side is indicated by signals ending with *a* and signals from the bus side are ending with a *b*.

The auxiliary registers $tagouta'$, $dataouta'$, $souta'$ and $soutb'$ are used to avoid a read and write to the same port in a single cycle. That way the design is also usable for pulse-triggered RAMs. Though edge-triggered RAMs are used in this thesis. The outputs *tagoutb* and *dataoutb* are never used in the same cycle when being written, such that no auxiliary registers are necessary for these outputs. Lemma 8.24 from [8] shows that the values within the auxiliary registers are the same as the outputs of the respective edge triggered RAMs. Therefore the pre-fetching is not necessary and these registers can be just replaced by wires.

Due to the changes of the protocol the registers are further used in additionally cycles like in state *emupdate*. Thats why we formulate with Lemma 4.1 a modified version of Lemma [8, L. 8.24]. The proof is almost identical to the original one in the reference and therefore not explicitly listed.

Lemma 4.1 (auxiliary registers [8, Lemma 8.24])
The output values of the registers *tagouta′*, *dataouta′*, *souta′*, and *soutb′* are the same as the corresponding output signals from the RAMs - in cycles when they are used.

1. $w(i)^t \vee localw(i)^t \vee wglobal'(ca(i)^t) \wedge emupdate(i)^t$
   $\vee\, global(ca(i)^t) \wedge mupdate(i)^t \quad \rightarrow\; dataouta'(i)^t = dataouta(i)^t$
2. $w(i)^t \vee global(ca(i)^t) \wedge mupdate(i)^t \;\rightarrow\; tagouta'(i)^t = tagouta(i)^t$
3. $w(i)^t \vee localw(i)^t \vee mlr1(i)^t \vee emupdate(i)^t \vee mupdate(i)^t$
   $\rightarrow\; souta'(i)^t = souta(i)^t$
4. $sw(i)^t \vee slr1(i)^t \vee supdate(i)^t \;\rightarrow\; soutb'(i)^t = soutb(i)^t$

4.2.1 Tag-RAM

The data paths of the tag RAM are only modified for the selection signal of a multiplexor (mux), which determines if *tagouta* or *tagouta′* is used.

The tag of a cache line is used to determine if the cache contains a line, addressed by the processor or bus and the respective signals *phit* and *bhit* are computed. In case of a flush, the tag of the memory address and cache line entry differ. For this reason a mux will select the tag of the cache line entry instead of the tag of the current processor address. In particular the currently stored cache entry is put on the bus in order to update the main memory.

4.2.2 Data-RAM

The data paths of the data RAM differ only in two selection signals for multiplexor from the original one: the auxiliary register *dataouta′* is used for the cycles described in Lemma 4.1 and the unmodified data is put on the bus via *b.data* for a memory update due to a read in *emupdate* or a $mlr0 \rightarrow mupdate$ transition.

4.2.3 State-RAM

The data paths of the state RAM contain the cache protocol communication. In particular *b.mprot* and *b.sprot* are computed to determine the new master or slave state of the cache.

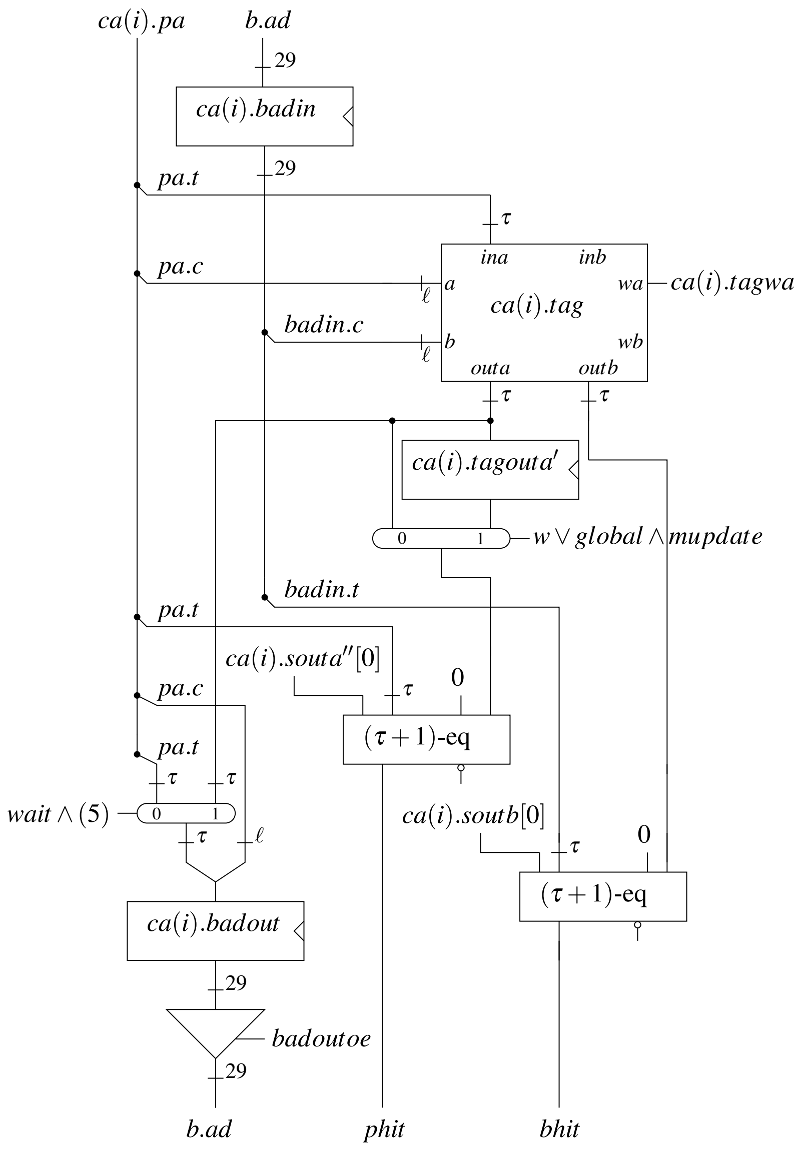


Figure 4.2: Data paths of the Tag-RAM of a cache

In the course of this thesis, the cache state was extended with an additional bit for the write mode and a second interactive protocol using the circuits $CS1$ and $CS2$ was introduced. For this reason, the data paths are adapted at multiple places to include these changes.

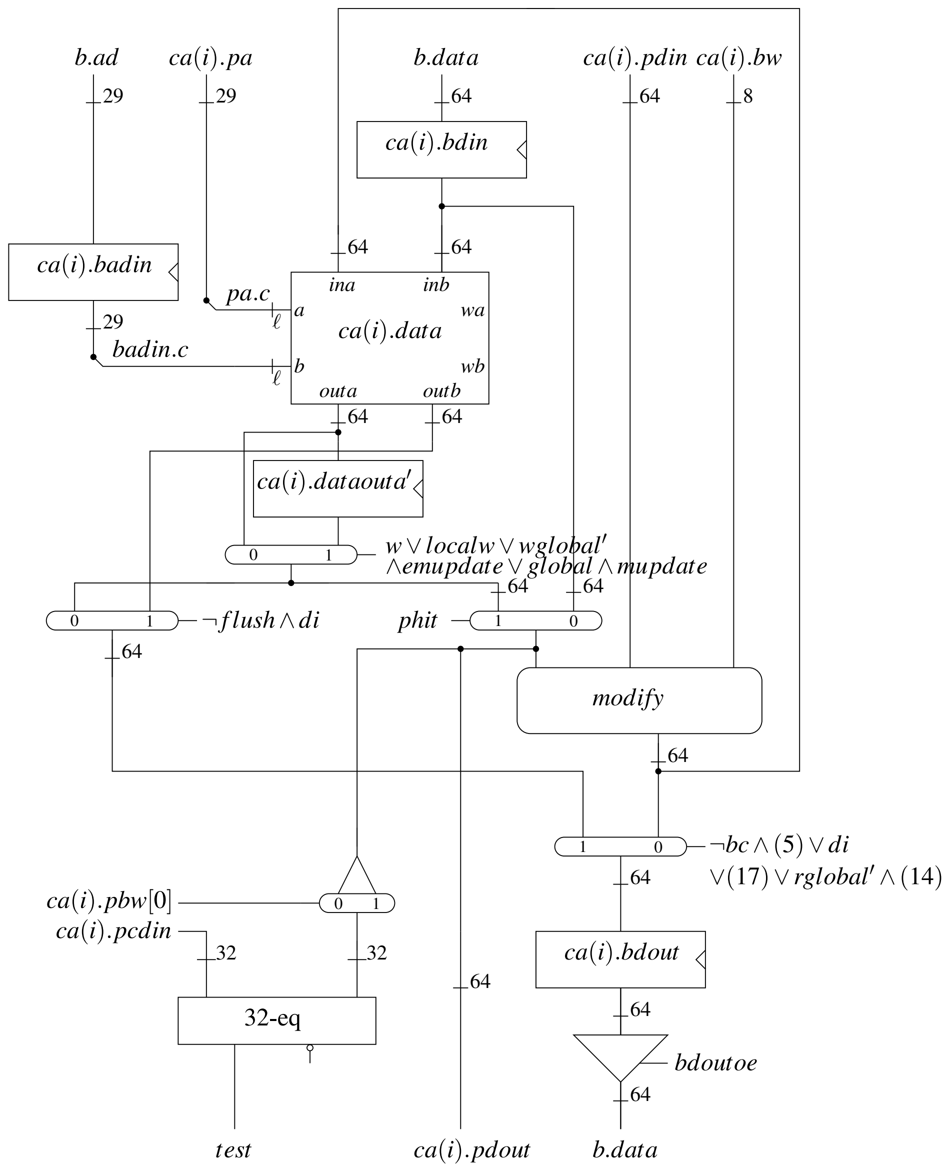


Figure 4.3: Data paths of the Data-RAM of a cache

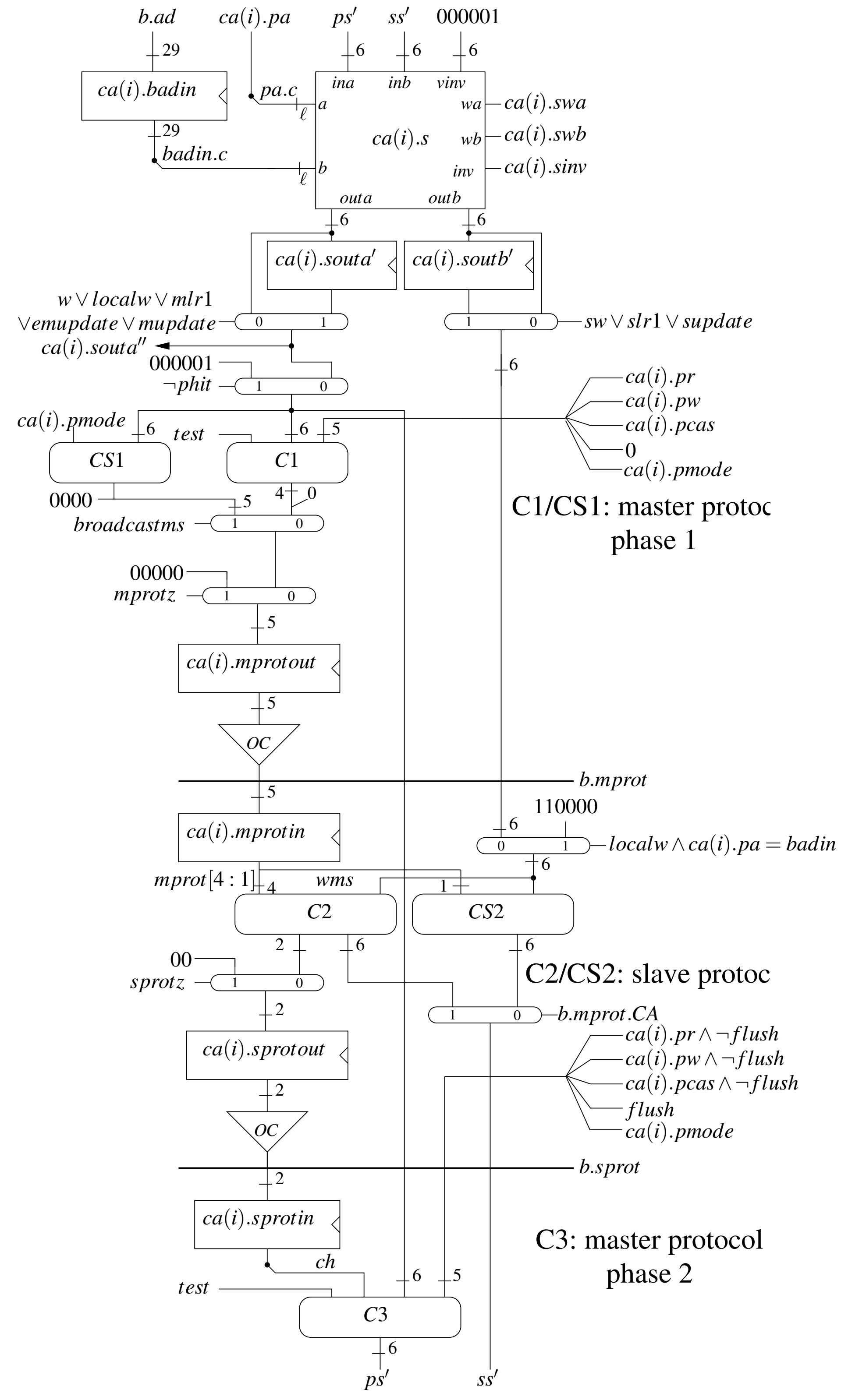


Figure 4.4: Datapaths of the State-RAM of a cache

4.3 Control Automata

In this section we will see the control automata, which implement the activation of the previously presented signals like the cache protocol signals or memory control signals. The specified invariants from Section 3.5 need to be maintained and a control mechanism for the newly introduced $global'$ accesses is necessary. Therefore it is essential to introduce new states and transitions to the master and slave control automata of [8]. Moreover the modifications of the accesses have to be applied to the control mechanism. Not only the changes but the overall automata construction is presented.

The mentioned automata are presented in Figure 4.5 as a directed graph labeled by the transitions - explained within this section. The additional states and transitions are represented by orange and the modified ones with green. The meaning of the different states is listed in Table 4.6. The first part of the table is still the same as in the reference [8] and the new states are appended.

4.3.1 Sets of Automata States

The respective states are grouped in certain sets in order to simplify the formalism for the later correctness proof of the cache protocol. Moreover the sets are used for case distinction, because exclusive control of the bus is required for $global$ or $global'$ accesses (in the following just referred as global). In particular the tristate bus was specified to be only used by one processor at the same time.

This is ensured by use of an arbiter, which generates grant signals $grant[i]$ for the master automaton i to acknowledge exclusive control over the bus. The respective cache $ca(i)$ participates as sole master in the protocol to serve its processor request. In the following sections the same arbiter construction as presented in [8] is used, which was proven to ensure liveness and fairness.

Master M

$$\begin{aligned} M = \{&idle,\ localw,\ wait,\ flush,\ m0,\ m1,\ m2,\ m3,\ mdata,\ w,\\ &mupdate,\ wait',\ emupdate,\ mlr0,\ mlr1\}\end{aligned}$$

Slave S $= \{sidle,\ sidle',\ s1,\ s2,\ s3,\ sdata,\ sw,\ supdate,\ slr1\}$

Local L $= \{idle,\ localw\}$

Update U $= \{wait',\ emupdate,\ mlr0,\ mlr1,\ mupdate\}$

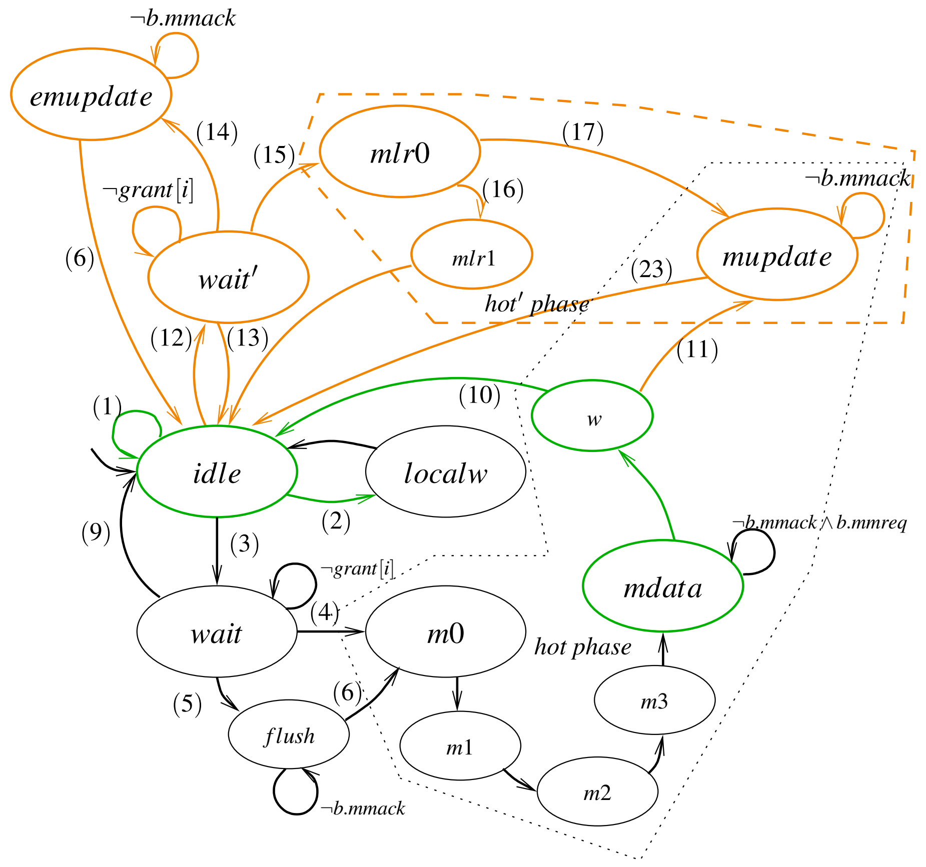
Global G $= \{\{mupdate\} \cup M \setminus (L \cup U)\}$

Warm W $= G \setminus \{wait\}$

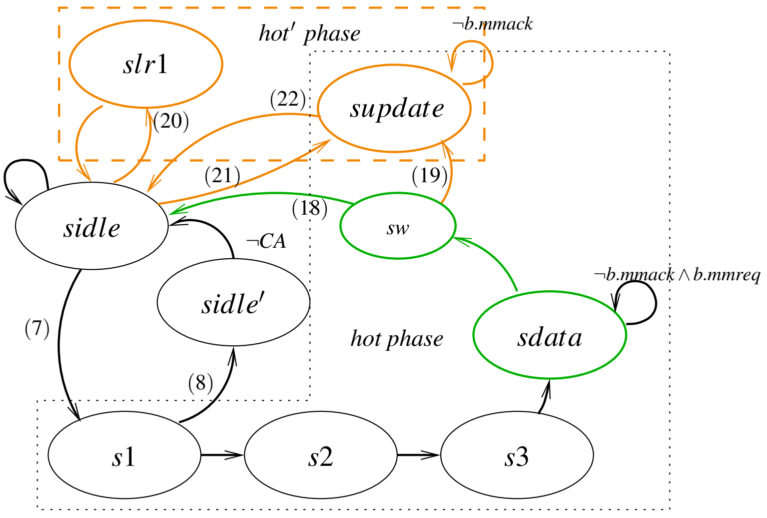
Warm$'$ W' $= U \setminus \{wait'\}$

Hot H $= W \setminus \{flush\}$

Hot$'$ H' $= \{mlr0,\ mlr1,\ mupdate\}$



(a) Master Automaton



(b) Slave Automaton

Figure 4.5: Cache Control Automata: the additional states and transitions are represented by orange and the modified ones in green

#	**master state**	**implemented behavior**	**slave state**	**implemented behavior**
0	*idle*	local read accesses (unless conflicting with a global transaction on the bus)	*sidle*	snooping on bus until an interactive protocol starts
1	*localw*	local write accesses (unless in conflict)	*sidle*	
2	*wait*	wait until exclusive bus access gets granted, compute *Ca*, *im*, *bc*, *nm*, *wms*	*sidle*	
3	*flush*	write back a dirty cache line to *mm*, compute *Ca*, *im*, *bc*, *nm*, *wms*	*sidle*	
4	$m0$	transmit *Ca*, *im*, *bc*, *nm*, *wms* via *b.mprot*	*sidle*	
5	$m1$	wait for the slave response	$s1$	check for a hit, compute the slave response signals *ch*, *di*
$6'$			*sidle*$'$	wait until *Ca* is lowered on the bus (in *sidle* new transaction would start)
6	$m2$	wait for slave response	$s2$	transmit response signals *ch*, *di* on bus line *b.sprot*
7	$m3$	analyze slave signals, prepare memory access or data broadcast (if necessary)	$s3$	prepare data for data intervention (if necessary)
8	*mdata*	wait for memory response (if necessary), read data from the bus (if necessary)	*sdata*	transmit data on the bus or read data from the bus (if necessary)
9	*w*	write *data*, *tag*, *s*	*sw*	write *data*, *s* (if necessary)
10	*mupdate*	write the data to the cache RAMs and the *mm* (used only in *wt* mode)	*supdate*	in case of a cache hit, set the write mode bit for the line address on the bus to *wt*, write the new MOESI state (if necessary)
11	*wait*$'$	wait for bus grant, in case of a mode broadcast compute *Ca*, *im*, *bc*, *nm*, *wms*	*sidle*	
12	*emupdate*	write the data to the cache RAMs and the *mm* (used only for exclusive line access in *wt* mode)	*sidle*	
13	$mlr0$	broadcast the change of the write policy by raising *wms*	*sidle*	
14	$mlr1$	wait while the slaves change their write mode	$slr1$	in case of a cache hit, flip the write mode bit for the line address on the bus

Table 4.6: An overview of the master and slave automata states

The master states in the *hot* or *hot′* phase have to run synchronized with the respective slave states of the *hot* and *hot′* phase. The *hot* phase implements the following protocol steps:

1. the master got granted exclusive bus control and computes the master protocol signals Ca, im, bc, nm, wms and puts them on the bus
2. slave: compute the slave responds ch, di and the new slave state
3. master: compute the new master state

Meanwhile the *hot′* phase implements the write mode broadcast

1. master: compute and broadcast with signal *wms*, if a write mode switch is effecting other caches; here the signals Ca, im, bc, nm are 0
2. slave: in case of a cache hit, flip the respective write mode bit if *wms* was raised

Other accesses - like local reads or writes in write back mode - may be served in parallel, meaning multiple caches can act as master at the same time for this case. Then, no interaction with other components of the memory system is necessary such that the cache request is directly applied and the new master state is computed.

In the subsequent sections the notation from [8] is used for automata states. In particular, let $z(i), zs(i)$ be the state of a master or a slave automaton i, then $x \in M : x(i)^t$ implies that during cycle t the master automaton i is in state x and analogue $x \in S : x(i)^t$ for a slave automaton i. This is abbreviated by $x(i)^t$ with

$$x(i)^t \equiv \begin{cases} z(i)^t = x & : x \in M \\ zs(i)^t = x & : x \in S \end{cases}$$

Then the following notation is used for any set of states A
with $A \in \{M, S, L, U, G, W, W', H, H'\}$:

$$A(i)^t \equiv z(i)^t \in A$$
$$A(i)^{[t:t']} \equiv \forall q \in [t : t'] : A(i)^q$$

Statements without any index t are implicitly quantified for all cycles. The automaton index i is omitted, whenever it is clear from the context, which cache is meant or the statement is quantified for all cache indexes. For any transition numbered with (n) the term $(n)(i)^t$ denotes that the condition is valid for the automaton of cache i in cycle t.

4.3.2 Automaton Transitions and Control Signals

Now we are going to look at the implementation of the different processor requests. Regarding the specifications of Chapter 3, it is reasonable to distinguish between the different access types. Therefore the following predicates are defined as a function of the hardware signals of the interfaces from Section 4.1. The definitions for $phit$ and $snoopconflict$ are the same as in [8]. The predicates $lread$ and $lwrite$ were defined as local read or writes in the reference. Now, they are further distinguished as $local$ or $global'$ accesses, because $global'$ transactions cannot be processed locally any more in the modified design. Each access, which is neither $local$ nor $global'$ is referred as $global$.

$$\begin{aligned}
lwms &\equiv s(pa.c)[5] \neq pmode \\
write &:= phit \wedge (pw \vee pcas \wedge test) \\
lwrite &:= write \wedge s(pa.c) \in \{E, M\} \\
lread &:= phit \wedge (pr \vee pcas \wedge \neg test) \\
broadcastms &:= lread \wedge lwms \wedge s(pa.c) \in \{O, S\} \\
rglobal' &:= lread \wedge s(pa.c) = M \wedge lwms \wedge wt \\
wglobal' &:= lwrite \wedge wt \\
rlocal &:= lread \wedge \neg broadcastms \wedge \neg rglobal' \\
wlocal &:= lwrite \wedge wb
\end{aligned}$$

Thereby $s(a) \in \mathbb{B}^6$ is the cache state for a line address $a \in \mathbb{B}^{29}$.

$$\begin{aligned}
local &:= rlocal \vee wlocal \\
global' &:= rglobal' \vee broadcastms \vee wglobal' \\
global &:= \neg local \wedge \neg global' \\
wglobal &:= global \wedge (pw \vee pcas \wedge test) \\
rglobal &:= global \wedge (pr \vee pcas \wedge \neg test) \\
phit &:= pa.tag = tag(pa.c) \wedge s(pa.c) \neq I \\
snoopconflict &:= \neg sidle \wedge pa = badin
\end{aligned}$$

A *snoopconflict* arises, whenever a processor has a local request for a line address, which is currently involved in a global transaction. In this case the data of the respective line may change due to the global transaction, such that the processor can not progress with the request until the global one is completed.

The set of $global'$ transactions subsumes the request types, which were newly established due to the additional write through policy. These correspond to one of the cases:

- $wglobal'$: a write access with write through policy
- $rglobal'$: a local read request changing the write mode to wt. If the cache line was modified (M), it is necessary to update the memory and to set the state to clean exclusive (E).

- *broadcastms*: the write policy of a shared line (O or S) is changing. This issues an update of the write mode bit in all caches with valid data for the accessed line.

Next we will consider each of the automata states and show their implementation together with the possible transactions. For this the generated signals are noted in form of $signalname := condition$. This means the signal is only raised, if the condition is fulfilled.

Many of the below specifications are already established in [8]. Nevertheless they are noted here for a better understanding of the overall concept. The additional or modified states and transitions are highlighted in Figure 4.5.

Master State idle

As we have seen, a processor request ends, whenever the mealy signal *mbusy* is lowered. This is done in state *idle* in case of a local read or if no processor request exists.

$$\neg mbusy \;=\; \neg preq \vee rlocal \wedge \neg snoopconflict$$

Additionally the processor waiting for $\neg mbusy$ may expect the answer for a read request, such that the content of $ca(i).data(ca(i).pa.c)$ is transmitted to the processor via signal $ca(i).pdout$. Now there are four possible transitions to take from state idle:

$(1): idle \rightarrow idle$ This is the case in absence of a processor request, whenever a local read is processed or the cache is currently participating as slave in a global transaction for the requested line. The later case is indicated by a snoopconflict, such that we have for this transition

$$(1) \;\equiv\; \neg preq \vee snoopconflict \vee rlocal$$

If a local read changes the write policy from write through to write back, the write mode in the state RAM has to be updated.

$$ca(i).swa := preq \wedge \neg snoopconflict \wedge rlocal \wedge wms \wedge wb$$

$(2): idle \rightarrow localw$

$$(2) \;\equiv\; preq \wedge \neg snoopconflict \wedge wlocal$$

In order to process the local write the registers $ca(i).dataouta'$ and $ca(i).souta'$ are clocked, as they are used in the next state for the outputs of the data and state RAMs. This is shown in the respective data paths in Section 4.2.

$$\begin{aligned} ca(i).dataouta'ce &:= (2) \\ ca(i).souta'ce &:= (2) \end{aligned}$$

$(3) : idle \to wait$

$$(3) \;\equiv\; preq \wedge \neg snoopconflict \wedge global$$

The processor has a *global* request, which can be only served with control over the bus. Therefore the automaton is going to state *wait* and remaining there until the arbiter grants the exclusive bus control.

$(12) : idle \to wait'$

$$(12) \;\equiv\; preq \wedge \neg snoopconflict \wedge global'$$

A $global'$ request is processed, such that the automaton has to remain in state $wait'$ until the bus access is granted.

For both transitions leading to the states *wait* or $wait'$ the signal $ca(i).reqset$ is raised to signal the bus request to the arbiter.

$$ca(i).reqset \;:=\; (3) \vee (12)$$

Master State wait′

In order to process the $global'$ request, interaction with other memory system components than cache i is necessary. Therefore the bus is used, such that the automaton has to wait until its bus request is granted with the signal $grant[i]$. Then the following next transitions are possible:

$wait' \to wait'$ The automaton remains in state $wait'$, while no bus access is granted.

$$\neg grant[i]$$

$(13) : wait' \to idle$ When the automaton entered state $wait'$ the predicate $global'$ was true for the processor request. But at the time when the bus access is granted, this is no longer the case. This behavior is similar to a CAS access, whose condition became false (transition (9)).

$$(13) \;\equiv\; grant[i] \wedge \neg global'$$

This scenario occurs for example if the criterion to broadcast a change of the write policy is fulfilled for a master cache i. Hence the automaton goes for a $global'$ transaction from *idle* to state $wait'$. At the same time another automaton j is already in state $mlr0$ in order to broadcast the write mode change.

Therefore cache i will get granted the bus after automaton j completes its processor request. By this point the condition *broadcastms* is no longer fulfilled for automaton i, such that it will go for a delayed local read instead.

Similarly the state of the requested line may change due to a global transaction, while the automaton is waiting for the bus access. Hence the predicates $wglobal'$ and $rglobal'$ may no longer hold.

In state $idle$ no access to the bus is necessary and thus the request of cache i is 0. Moreover, a local read can be served within one cycle, such that the processor can directly read the cache data. Then by the rules for processor and cache communication, the $mbusy$ signal has to be lowered for this case.

$$\begin{aligned} ca(i).reqclear &:= (13) \\ \neg ca(i).mbusy &:= (13) \wedge rlocal \end{aligned}$$

$(14) : wait' \rightarrow emupdate$ The automaton progresses into state $emupdate$ to process a read or write of an exclusive cache line (M or E) with main memory update. In particular this is the case for a write in write through mode or a read with write mode switch from wb to wt.

$$(14) \equiv grant[i] \wedge (wglobal' \vee rglobal')$$

The state of the cache line is set to $0E$ in the next automaton state $emupdate$. Therefore the auxiliary register $souta'$ is clocked. In case of a write access the cache line data is modified, such that additionally $dataouta'$ has to be clocked.

$$\begin{aligned} ca(i).dataouta'ce &:= (14) \wedge wglobal' \\ ca(i).souta'ce &:= (14) \end{aligned}$$

The tag RAM stays unchanged, as it is only modified if the requested line is not yet cached. This would contradict with the premise of a cache hit for a $global'$ transaction.

In order to put the data on the bus ($ca(i).bdout$) and to update the main memory in the next state $emupdate$, the following signals need to be activated:

$$\begin{array}{l} ca(i).mmreqset \\ ca(i).mmreqoeset \\ ca(i).mmwset \\ ca(i).mmwoeset \\ ca(i).bdoutce \\ ca(i).bdoutoeset \end{array} \tag{4.2}$$

$$\begin{array}{l} ca(i).badoutce \\ ca(i).badoutoeset \end{array} \tag{4.3}$$

$(15) : wait' \rightarrow mlr0$

$$(15) \equiv grant[i] \wedge broadcastms$$

The master automaton i has to signal the other caches - holding the requested cache line - that the write policy has changed. For this reason the register for the master protocol signals is clocked. But in order to determine if the slaves have a cache hit, they need to know the accessed address. Therefore the affected address is put on the bus by enabling the driver with *badoutoeset* and clocking the address $pa(i)$ into the bus address output register by enabling *badoutce*.

$$\begin{aligned} mprotoutce &:= (15) \\ badoutoeset &:= (15) \\ badoutce &:= (15) \end{aligned}$$

Master State emupdate

In this state the main memory is updated with the cache line on the bus. This results in two possible transitions:

emupdate $\rightarrow$ *emupdate*

The automaton stays in *emupdate*, while the memory is busy (the completion of the access is not yet acknowledged $\neg$*b.mmack*).

emupdate $\rightarrow$ *idle*

The active signal *b.mmack* indicates that the memory access is finished. Therefore the bus signals can be cleared and the automaton goes back to *idle* to process the next processor request.

$$(6) \equiv b.mmack$$

During the last cycle in this state (when *b.mmack* is 1) the cache state is changed to $ps' = 0E$ accordingly to the specification. Therefore the state RAM has to be clocked. Additionally, in case of a write or CAS hit ($wglobal'$) the data RAM is updated.

$$\begin{aligned} ca(i).swa &:= (6) \\ ca(i).datawa &:= (6) \wedge wglobal' \end{aligned}$$

$$\begin{aligned} ca(i).bdoutoeclear &:= (6) \\ ca(i).badoutoeclear &:= (6) \\ ca(i).mmreqoeclear &:= (6) \\ ca(i).mmreqclear &:= (6) \\ ca(i).mmwoeclear &:= (6) \\ ca(i).mmwclear &:= (6) \end{aligned}$$

Moreover, the signals *req* and *mbusy* can be lowered now, as it is the last cycle of the request and the bus is not needed any longer.

$$\begin{aligned} \neg ca(i).mbusy &:= (6) \\ ca(i).reqclear &:= (6) \end{aligned}$$

Master State mlr0 and Slave State sidle

On the master automaton transition from state $wait'$ to $mlr0$ the master protocol signals are committed via the protocol bus. In particular the signal $mprot.wms$ is raised.

Now, there are two possibilities for the master automaton to progress. Either the write policy changes from write through to write back or the other way around. In case of the write through policy ($wt \Leftrightarrow pmode = 0$), the main memory needs to be updated. For this reason the automaton proceeds into state $mupdate$. Otherwise, only the cache state of the master and slave caches have to be updated. This case is handled in state $mlr1$.

$$\begin{aligned}(16) &\equiv wb\\(17) &\equiv wt\end{aligned}$$

In both cases the master cache state is updated in the next automaton state. Therefore $souta'$ is clocked.

$$ca(i).souta'ce$$

For the case of a memory update (transition (17)), the cache data is put on the bus by activating the signals from Equation (4.2). The address was already put on the bus during state $wait'$, such that the signals from Equation (4.3) are not necessary. For a condition Y and an equation (X) (a set of signals) the notation $(X) := Y$ means that all signals, listed in (X), are activated if condition Y is satisfied.

$$(4.2) := (17)$$

Every slave automaton j will progress into state $slr1$ or $supdate$, in order to change the write mode bit in the own cache state RAM, if it has a hit for the accessed line. In particular the next slave state is $slr1$, if a write mode change is broadcast, but no memory update is issued. This is the case when the write mode changes from write through to write back ($mprot.wms \wedge ca(j).s[5](b.ad) = 0$). Otherweise ($mprot.wms \wedge ca(j).s[5](b.ad) = 1$), the slave proceeds into state $supdate$. Note that even slaves with a cache miss will proceed in one of these states. But in this case they will do nothing, but return into state $sidle$ in the next cycle.

$$\begin{aligned}(20) &\equiv \neg grant[j] \wedge b.mprot.wms \wedge \neg ca(j).s[5](b.ad)\\(21) &\equiv \neg grant[j] \wedge b.mprot.wms \wedge ca(j).s[5](b.ad)\end{aligned}$$

In order to compute the new slave cache state with function $CS2$ in the next automaton state, the master protocol signals have to be clocked into register $mprotin$. Furthermore the slave cache state for address $badin$ will be updated. Therefore the auxiliary register $soutb'$ and register $badin$ are clocked.

$$\begin{aligned}ca(j).soutb'ce &:= (20) \vee (21)\\ca(j).mprotince &:= (20) \vee (21)\\ca(j).badince &:= (20) \vee (21)\end{aligned}$$

Master State mlr1 and Slave State slr1

The master protocol output signals are no longer needed, such that they are cleared by raising the signals

$$ca(i).mprotz$$
$$ca(i).mprotoutce$$

Next the master cache *i* updates the own write mode bit of the state RAM.

$$ca(i).swa$$

At the same time all slave automata *j* - with a hit for the requested address (signaled by *bhit*) - update their write mode bit. Anyway all slave automata go back into state $sidle$.

$$ca(j).swb \;:=\; bhit$$

Now, the request can be completed by lowering the *mbusy* signal and going back to state idle, where the cache data output is transmitted to the processor.

$$\neg ca(i).mbusy$$
$$ca(i).reqclear$$
$$ca(i).badoutoeclear$$

Master State mupdate and Slave State supdate

Any slave *j* without cache hit goes back to state $sidle$. Otherwise the slaves remain in state $supdate$ until the memory access is acknowledged.

$$(22) \equiv b.mmack \vee \neg b.mmreq \vee \neg bhit$$

In this state the cache line on the bus is written to the main memory. This results in two possible transitions for the master:

$mupdate \rightarrow mupdate$

The automaton stays in $mupdate$, while the memory is busy (the completion of the access is not yet acknowledged $\neg b.mmack$).

$mupdate \rightarrow idle$

The active signal *b.mmack* indicates that the memory access is finished. Therefore the bus signals can be cleared and the automaton goes back to $idle$ for the next processor request.

$$(23) \equiv b.mmack \vee \neg b.mmreq$$

$$\begin{aligned}
ca(i).bdoutoeclear &:= (23)\\
ca(i).badoutoeclear &:= (23)\\
ca(i).mmreqoeclear &:= b.mmack \wedge \neg rglobal\\
ca(i).mmreqclear &:= b.mmack \wedge \neg rglobal\\
ca(i).mmwoeclear &:= b.mmack \wedge \neg rglobal\\
ca(i).mmwclear &:= b.mmack \wedge \neg rglobal
\end{aligned}$$

If the memory was requested by a slave due to a read miss, the intervening slave disables the output drivers after the main memory access is completed.

$$\begin{aligned}
ca(j).mmreqoeclear &:= b.mmack \wedge \neg ca(j).mprotin.im \wedge ca(j)sprotout.di\\
ca(j).mmreqclear &:= b.mmack \wedge \neg ca(j).mprotin.im \wedge ca(j)sprotout.di\\
ca(j).mmwoeclear &:= b.mmack \wedge \neg ca(j).mprotin.im \wedge ca(j)sprotout.di\\
ca(j).mmwclear &:= b.mmack \wedge \neg ca(j).mprotin.im \wedge ca(j)sprotout.di
\end{aligned}$$

Now the protocol output signals are no longer needed, such that they are cleared by raising the signals

$$\begin{aligned}
ca(i).mprotz &:= (23)\\
ca(i).mprotoutce &:= (23)\\
ca(j).sprotz &:= (22)\\
ca(j).sprotoutce &:= (22)
\end{aligned}$$

Similar to state *emupdate*, the cache RAMs are written during the last cycle in state *mupdate*. There are two possibilities to reach the master state *mupdate*. The first one is a read access for a shared cache line with write policy switch to write through. Therefore only the cache state for the master *i* and slave caches *j* need to be written. The second possibility is a *global* access in write through mode. In this case the data and tag RAM of the master are updated as well. Moreover, the data RAM of the slaves is updated for a broadcast ($mprot.bc = 1$).

$$\begin{aligned}
ca(i).swa &:= (23)\\
ca(j).swb &:= (22) \wedge bhit(j)\\
ca(i).datawa &:= (23) \wedge global\\
ca(i).tagwa &:= (23) \wedge global\\
ca(j).datawb &:= (22) \wedge bhit(j) \wedge ca(j).mprotin.bc
\end{aligned}$$

Additionally, the signals *req* and *mbusy* can be lowered now, as it is the last cycle of the request and the bus is not needed any longer.

$$\begin{aligned}
\neg ca(i).mbusy &:= (23)\\
ca(i).reqclear &:= (23)
\end{aligned}$$

Master State localw

The new master state $ps' = 1M$ is written to the state RAM by enabling signal *swa*. In addition to that the write enable of the data RAM is set with *datawa* to update the RAM with the modified data. This transaction is only one cycle long, such that the automaton lowers its *mbusy* signal and goes directly back to state *idle* in the next cycle.

$$\begin{aligned}
&ca(i).swa\\
&ca(i).datawa\\
&\neg ca(i).mbusy
\end{aligned}$$

Master State wait

The cache remains in state *wait* until the bus is granted, which is acknowledged by $grant[i]$. Then there are three ways to progress:

(9): $wait \rightarrow idle$

When entering state *wait* the condition of a global CAS hit was fulfilled. But while waiting in this state, another cache completed an access to the same memory line. This can result in a CAS miss, when the bus access is finally granted. In that case, the cache will perform a delayed local read access and go back to state $idle$.

$$(9) : grant[i] \wedge phit \wedge pcas \wedge \neg test$$

Then, the bus access is no longer needed and the read can be answered within one cycle such that the bus request and *mbusy* signal can be lowered.

$$\begin{aligned} ca(i).reqclear &:= (9) \\ \neg ca(i).mbusy &:= (9) \end{aligned}$$

(4): $wait \rightarrow m0$

The bus access is granted and there is no need to evict the cache line with a flush access. Then if the condition for a global access is still fulfilled (i.e. (9) is not satisfied), the cache can start the global protocol to process its processor request.

$$(4) \equiv grant[i] \wedge \neg(5) \wedge \neg(9)$$

(5): $wait \rightarrow flush$

If the request is granted but the corresponding cache line is occupied for another memory address and it is dirty, transition (5) will issue a flush access. Any line in write through mode or Exclusive state is clean, such that there is no need to access the main memory. For a shared state (S), there are still other caches holding the respective cache line and hence no memory update is necessary as well. In any other case a memory access is necessary to write the current cache line data to the main memory.

$$(5) \equiv grant[i] \wedge \neg phit \wedge s(pa.c)[5] \wedge s(pa.c) \in \{O, M\}$$

Then the following signals are set for a $global$ transaction:

$$\begin{aligned} ca(i).bdoutce &:= (5) \\ ca(i).badoutce &:= (5) \text{ or } (4) \\ ca(i).mmreqset &:= (5) \\ ca(i).mmwset &:= (5) \\ ca(i).bdoutoeset &:= (5) \\ ca(i).badoutoeset &:= (5) \text{ or } (4) \\ ca(i).mmreqoeset &:= (5) \\ ca(i).mmwoeset &:= (5) \\ ca(i).mprotoutce &:= (4) \end{aligned}$$

Master State flush

During this state the main memory is updated with the cache line data. The automaton has to wait until this access is finished. This is acknowledged by the memory with the raised signal *mmack*. The automaton will write the invalid state (I) into the cache state RAM and progress to state $m0$ after receiving *mmack*.

Accordingly two transitions are possible:

$flush \rightarrow flush$

The automaton is remaining in state $flush$, until the cache line is written to the main memory ($\neg b.mmack$).

(6): $flush \rightarrow m0$

$$(6) \equiv b.mmack$$

The memory access is completed and the automaton can advance into state $m0$ to handle the processor request. For this, all the bus signals are cleared, the cache state is invalidated and the requested address together with the master protocol signals are put on the bus.

$$\begin{aligned} ca(i).bdoutoeclear &:= (6) \\ ca(i).mmreqoeclear &:= (6) \\ ca(i).mmreqclear &:= (6) \\ ca(i).mmwoeclear &:= (6) \\ ca(i).mmwclear &:= (6) \\ ca(i).badoutce &:= (6) \\ ca(i).mprotoutce &:= (6) \\ ca(i).swa &:= (6) \end{aligned}$$

Master State m0 and Slave State sidle

This phase is exactly one cycle long. This means the master automaton always advances into state $m1$ in the next cycle.

For the slave automaton, there are two possibilities, either it stays in $sidle$ or it proceeds to state $s0$.

$sidle \rightarrow sidle$

the signal Ca is not active on the bus or the master automaton of this cache is the one in control of the bus. Therefore the slave just stays in $sidle$.

(7): $sidle \rightarrow s1$

The slave automaton for cache j proceeds into state $s1$, if the signal Ca was activated on the bus by some master i in phase $m0$ and the control automata do not belong to the same cache $i \neq j$.

$$(7) \equiv b.mprot.Ca \wedge \neg grant[j]$$

In the last cycle the master clocked its protocol signals into the output register, such that they are available during this phase. Now the slave is clocking these master protocol signals and the requested address into its input registers by enabling *mprotince* and *badince*.

$$\begin{aligned} ca(j).mprotince &:= (7) \\ ca(j).badince &:= (7) \end{aligned}$$

Master State m1 and Slave State s1

The master is advancing into state $m2$, meanwhile the slave takes one of two possible transitions:

- (8): $s1 \rightarrow sidle'$

  $$(8) \equiv \neg bhit$$

  If the slave has no valid data for the requested line ($\neg bhit$), it goes to state $sidle'$ and remains there until the request is completed. In particular the slave will not leave state $sidle'$ until the master protocol signal Ca is lowered again.

- $s1 \rightarrow s2$ All slave automata j with valid data for the requested cache line go to state *s2* and participate in the global protocol.

  $$ca(j).sprotoutce := bhit$$

Slave State sidle′

The slave automaton is waiting in this state, until the protocol signal Ca is lowered by the master.

Master State m2 and Slave State s2

During this phase the slave protocol signals are transmitted on the bus, such that the master i clocks them into its input register.

$$ca(i).sprotince$$

Then the master and slave automata advance into state $m3$ and $s3$ respectively.

Master State m3 and Slave State s3

Now there are three possible scenarios to transfer data over the bus:

1. a hit for a write or CAS request is broadcast to the slaves. Then the protocol signal *bc* is activated and the master transmits the modified data on the bus.

2. a cache miss and there is no owner for the cache line. The master requests the data from the main memory.
3. the requested data is received from a slave cache j, which indicated data intervention by raising $sprot(j).di$.

Accordingly, the master has to set a memory request, if neither a broadcast nor a data intervention takes place.

$$\begin{aligned} ca(i).mmreqset &:= \neg ca(i).mprotout.bc \wedge \neg ca(i).sprotin.di \\ ca(i).mmreqoeset &:= \neg ca(i).mprotout.bc \wedge \neg ca(i).sprotin.di \end{aligned}$$

If the master intends to broadcast modified data, it has to clock the data output and enable the bus driver respectively.

$$\begin{aligned} ca(i).bdoutce &:= ca(i).mprotout.bc \\ ca(i).bdoutoeset &:= ca(i).mprotout.bc \end{aligned}$$

For data intervention the (owner) slave will put the line on the bus.

$$\begin{aligned} ca(j).bdoutce &:= ca(j).sprotout.di \\ ca(j).bdoutoeset &:= ca(j).sprotout.di \end{aligned}$$

Master State mdata and Slave State sdata

According to the previous scenarios, the master reads the data from the bus if necessary. The respective control signals are cleared when the automaton advances into the next state. This is the case in the next cycle, if no memory request was started or when the memory access is completed (acknowledged by *mmack*).

In the next state the cache RAMs are updated, if no main memory update is necessary. Therefore the registers $dataouta', souta'$ and $tagouta'$ are clocked, as their value is used for the cache update (see the data paths in Section 4.2).

$$\begin{aligned} ca(i).mmreqclear &:= b.mmreq \wedge b.mmack \\ ca(i).mmreqoeclear &:= b.mmreq \wedge b.mmack \end{aligned}$$

$$\begin{aligned} ca(i).bdince &:= b.mmreq \wedge b.mmack \vee ca(i).sprotin.di \\ ca(i).bdoutoeclear &:= \neg b.mmreq \vee b.mmack \\ ca(i).badoutoeclear &:= \neg b.mmreq \vee b.mmack \end{aligned}$$

$$\begin{aligned} ca(i).dataouta'ce &:= \neg b.mmreq \vee b.mmack \\ ca(i).souta'ce &:= \neg b.mmreq \vee b.mmack \\ ca(i).tagouta'ce &:= \neg b.mmreq \vee b.mmack \end{aligned}$$

If the master broadcasts any data, the slave clocks its bus data input register and writes the new value into its RAM. Additionally the register $soutb'$ is clocked in order to update the slave state RAM during state sw or $supdate$.

If the slave j transmitted data on the bus for data intervention, it disables the output drivers again.

$$\begin{aligned} ca(j).bdince &:= ca(j).mprotin.bc \\ ca(j).soutb'ce &:= b.mmack \vee \neg b.mmreq \\ ca(j).bdoutoeclear &:= ca(j).sprotout.di \end{aligned}$$

Master State w and Slave State sw

Two transitions are possible for the master and slave automata. If the processed request is a write or CAS hit in write through policy, the memory needs to be updated with the modified data (in state *mupdate*). Otherwise only the cache RAMs need to be updated to complete the request.

(10): $w \rightarrow idle$ and (18): $sw \rightarrow sidle$

$$(10) \equiv ca(i).pmode \vee (\neg ca(i).mprotout.im \wedge \neg ca(i).sprotin.di)$$

If the main memory was accessed for a read miss ($\neg ca(i).mprotout.im \wedge \neg ca(i).sprotin.di$), the cache line is already clean. Therefore in this case or if the access is applied in write back mode, no memory update is necessary. Hence only the cache RAMs need to be written to complete the processor request and the protocol signals can be cleared.

$$(18) \equiv ca(j).mprotin.nm \vee (\neg ca(j).mprotin.im \wedge \neg ca(j).sprotout.di)$$

$$\begin{aligned} ca(i).mprotoutce &:= (10) \\ ca(i).mprotz &:= (10) \\ ca(j).sprotoutce &:= (18) \\ ca(j).sprotz &:= (18) \end{aligned}$$

Specifically, the master and slave automata write the results from the previous interaction into their RAMs. Therefore, the master enables the respective write signals.

$$\begin{aligned} ca(i).datawa &:= (10) \\ ca(i).tagwa &:= (10) \\ ca(i).swa &:= (10) \end{aligned}$$

Meanwhile the slave automata raise the following signals to update their data and state RAMs.

$$\begin{aligned} ca(j).datawb &:= ca(j).mprotin.bc \wedge (18) \\ ca(j).swb &:= (18) \end{aligned}$$

The request is completed, such that the bus request and *mbusy* signals are disabled.

$$\begin{aligned} ca(i).reqclear &:= (10) \\ \neg ca(i).mbusy &:= (10) \end{aligned}$$

The master can go to state *idle* and the slaves progress into state *sidle*.

(11): $w \rightarrow mupdate$ and (19): $sw \rightarrow supdate$

Note that it is not possible to go directly from *mdata* to *mupdate*, since in case of a write miss the necessary data in $ca(i).bdin$ is not available in the previous cycle.

$$\begin{aligned}(11) &\equiv \neg ca(i).pmode \wedge (ca(i).mprotout.im \vee ca(i).sprotin.di)\\(19) &\equiv \neg ca(j).mprotin.nm \wedge (ca(j).mprotin.im \vee ca(j).sprotout.di)\end{aligned}$$

For a global write or CAS access in write through mode, the modified data needs to be written to the main memory. Due to this reason, the master proceeds with state *mupdate* and the slaves with state *supdate*. Therefore the respective signals to put the address and modified data on the bus are raised.

Alternatively one could just keep the address on the bus for this case and only disable it for transition (10). But in any case, it is necessary to put the modified data on the bus. In case of a write miss, this value is only computable once the register value *bdin* was updated in the previous state.

$$\begin{aligned}(4.3) &:= (11)\\ca(i).mmreqset &:= (11) \wedge ca(i).mprotout.im\\ca(i).mmreqoeset &:= (11) \wedge ca(i).mprotout.im\\ca(i).mmwset &:= (11) \wedge ca(i).mprotout.im\\ca(i).mmwoeset &:= (11) \wedge ca(i).mprotout.im\\ca(i).bdoutce &:= (11)\\ca(i).bdoutoeset &:= (11)\end{aligned}$$

In case of a read miss the master does not know, if the accessed cache line is clean or dirty. Therefore the intervening slave signals the memory request if necessary. In particular the memory is updated if the cache line is dirty (state M or O in write back mode).

$$\begin{aligned}ca(j).mmreqset &:= (19) \wedge ca(j).s(badin) \in \{110000, 101000\}\\&\quad \wedge \neg ca(j).mprotin.im \wedge ca(j)sprotout.di\\ca(j).mmreqoeset &:= (19) \wedge ca(j).s(badin) \in \{110000, 101000\}\\&\quad \wedge \neg ca(j).mprotin.im \wedge ca(j)sprotout.di\\ca(j).mmwset &:= (19) \wedge ca(j).s(badin) \in \{110000, 101000\}\\&\quad \wedge \neg ca(j).mprotin.im \wedge ca(j)sprotout.di\\ca(j).mmwoeset &:= (19) \wedge ca(j).s(badin) \in \{110000, 101000\}\\&\quad \wedge \neg ca(j).mprotin.im \wedge ca(j)sprotout.di\end{aligned}$$

5 Correctness

In this chapter the correctness of the presented design is proven. The overall concept follows the lines of the proof from [8], such that the respective lemmas can be either directly reused or need only minor changes. These lemmas are noted with the original reference number of the source. When the proof of the adapted lemmas is very similar to the original one, it is not explicitly listed. Due to the changes of the protocol, additional lemmas are necessary in order to prove the validity of the invariants and bus communication rules. The respective lemmas and proofs are provided in the subsequent sections.

First we are going to see that the principles for the bus and driver specifications are satisfied. In particular it is shown that at most one cache is acting as master in the interactive cache coherency protocol, that the signals committed over the bus are correctly computed, enabled and disabled and that the gate level design matches the previous specifications.

5.1 Only a Single Master on the Bus

The specified design allows that multiple local processor requests are served in parallel. Then each of the corresponding caches will act as master cache and locally process the access to the local cache. But as soon as a processor request requires interaction with other components of the memory system, only one of these requests can be processed at the same time. Such requests were specified as $global$ or $global'$ transactions and are consolidated as global if applicable.

For $global$ requests the necessary properties are already proven in the reference book, such that the respective lemmas are only listed, if they are used for the later introduced lemmas. With respect to $global'$ requests we are going to see that due to the use of an arbiter, indeed at most one cache is acting as master in the global transaction.

Each cache which aims for a $global'$ transaction needs access to the bus to process the request. For this reason it will request the bus by raising signal *req* and not lower the request until the global phase of the protocol is completed.

Lemma 5.1 (request at global′)
Any master automaton in the phase $global'$ - more precisely in one of the states of the set U - has an active request for the bus.

$$U(i) \;\rightarrow\; req(i)$$

PROOF. By induction over the cycle t.

Induction Base:
For $t = 0$ the statement is true, because initially the automata are in state idle ($idle(i)^0$) and therefore they cannot be in the $global'$ phase ($\neg U(i)$).

Induction Hypothesis:
Let us assume $U(i)^{t-1} \to req(i)^{t-1}$ to be true.

Induction Step:
We aim to show the Lemma for cycle t. In case of $\neg U(i)^t$, there is nothing to show at all. That is why, only automata states in the $global'$ phase are considered, i.e $U(i)^t$ is assumed. Now, there are two possibilities, either automaton i was in phase *update* before ($U(i)^{t-1}$). Or it just entered the state $wait'$ and consequently $\neg U(i)^{t-1}$ holds.

$U(i)^{t-1}$**:** Given the (IH) one can conclude that the request was enabled in the previous cycle ($req(i)^{t-1}$) and by the automata construction (A) follows for the state

$$\neg(mupdate(i)^{t-1} \wedge (23)(i)^{t-1} \vee emupdate(i)^{t-1} \wedge (6)(i)^{t-1}) \\ \wedge \neg mlr1(i)^{t-1} \wedge \neg(wait'(i)^{t-1} \wedge (13)(i)^{t-1})$$

In other words the automaton was not in the last cycle of the $global'$ phase, as it is still in the update phase during cycle t and the automaton would have returned into state idle otherwise. Additionally, we know that the signal *req* is only cleared in this last cycle for a $global'$ transaction. Respectively follows $\neg reqclear(i)^{t-1}$. Altogether, this gives us:

$$\begin{aligned} req(i)^t &= req(i)^{t-1} && (HW) \\ &= 1 && (IH) \end{aligned}$$

$\neg U(i)^{t-1}$**:** In this case the automaton just entered the $global'$ phase and is therefore in state $wait'$ at cycle t. We have by (A):

$$idle(i)^{t-1} \wedge (12)(i)^{t-1} \wedge reqset(i)^{t-1} \wedge wait'(i)^t$$

From the hardware construction (HW) of registers (set/clear flipflops as presented in [8]) it is given that the request signal will stay stable, until it is lowered due to activating signal $reqclear(i)$. It follows

$$req(i)^t = 1$$

■

Once the bus access is granted, the master automaton will stay in control of the bus until the request is disabled due to the end of the global phase.

Lemma 5.2 (grant stable [8, Lemma 8.15])
During an active request a grant is not taken away:

$$grant[i]^t \wedge req(i)^t \;\rightarrow\; grant[i]^{t+1}$$

After setting its request, a cache automaton i can only proceed into the *warm* or *warm′* phase, if the arbiter grants the bus access, which is indicated with $grant[i] = 1$.

Lemma 5.3 (grant at warm′)
A master automaton i can only be in the *warm′* phase W', if the access to the bus was granted by the arbiter.

$$W'(i) \;\rightarrow\; grant[i]$$

PROOF. by induction over the cycle t.

Induction Base:
For cycle $t = 0$, all master automata are in state idle, such that there is nothing to show.

Induction Hypothesis:
Let us assume for cycle $t-1$ that the statement $W'(i) \rightarrow grant[i]$ is true.

Induction Step:
Now, we consider two cases for $t-1$:

$\neg W'(i)^{t-1}$**:** From (A) and the definition of W' it is common knowledge that the automaton was in state *wait′* and proceeded to the *warm′* phase in the last cycle.

$$\neg wait'(i)^t \wedge wait'(i)^{t-1}$$

Lemma 5.1 (request at global′) gives us an active request signal for this cycle ($req(i)^{t-1}$), since the automaton is in state $wait' \in U$. Furthermore, the automaton i can only proceed into the *warm′* phase, if it has the grant from the arbiter according to (HW) of the control automaton. Therefore $grant[i]^{t-1}$ has to be true. Now, with application of Lemma 5.2 (grant stable) one can conclude $grant[i]^t$.

$W'(i)^{t-1}$**:** As before follows from Lemma 5.1 (request at *global′*), that the request signal is active ($req(i)^{t-1}$). At the same time the automaton has $grant[i]^{t-1}$ for the bus by (IH). As the automaton is still in a *warm′* state, the request was not lowered in the previous cycle. This results in $req(i)^t = 1$, hence Lemma 5.2 (grant stable) ensures $grant[i]^t$. ■

Similarly there is a lemma for the *warm* phase of a *global* transaction.

Lemma 5.4 (grant at warm [8, Lemma 8.17])
A master can only be in the *warm* phase W if the access to the bus is granted by the arbiter:

$$W(i) \;\rightarrow\; grant[i]$$

Additionally, the construction of the arbiter ensures that only one cache automaton has *grant* at the same time.

Lemma 5.5 (grant unique [8, Lemma 8.14])

$$grant[i] \wedge grant[j] \;\rightarrow\; i = j$$

These lemmas allow to prove the property that only one cache can act as master in the critical protocol phases *warm* and *warm′*. From that we can conclude correct behavior of the control automata and properly use of the tristate buses.

Lemma 5.6 (warm unique [8, Lemma 8.18])
At most one control automaton can be in a *warm* or *warm′* phase at the same time.

$$(W(i) \vee W'(i)) \wedge (W(j) \vee W'(j)) \implies i = j$$

PROOF. When the statement $W(i) \vee W'(i)$ is true, there is an active grant signal for cache i ($grant[i]$) according to the Lemmas 5.4 and 5.3 (grant at warm or warm′). The same lemmas apply for cache j, such that we have $grant[j]$ for the satisfied condition $W(j) \vee W'(j)$.

Now follows from Lemma 5.5 (grant unique) that both indexes i and j have to belong to the same cache control automaton $i = j$. ■

5.2 Open Collector Bus Signals

The different caches communicate with each other through the protocol signals via an open collector bus. These are grouped as master (*mprot*) or slave protocol signals (*sprot*). As described in the Background Chapter 2, the open collector bus computes the OR over the input signals of the connected open collector drivers.

It has to be ensured that only the master cache is putting *mprot* signals different from zero on the bus. Additionally all slave automata - not participating in the current protocol - have to stay *silent*, in means of putting 0 on the open collector bus.

Lemma 5.7 (silent slaves [8, Lemma 8.19])
When a slave is not participating in the protocol, it transmits the values 00 as slave response on the control bus.

$$zs(i) \in \{sidle, sidle', s1, slr1, supdate\} \implies sprotout(i) = 00$$

The master participates in a protocol during the hot phases H, H'. During these phases the master and slave automata are in sync, as we are going to see in the next section. But outside these phases, the master is not transmitting any protocol data. This behavior is mirrored in the following lemma.

Lemma 5.8 (silent master [8, Lemma 8.20])
When a control automaton is not participating as master in the protocol, it puts all zeros for *mprot* on the protocol bus.

$$\neg H(i) \wedge \neg H'(i) \implies mprotout(i) = 00000$$

PROOF. by induction over the cycle t.

Induction Base:
At cycle $t = 0$ all automata are in state $idle^0$ and the signal *mprotz* is active. By (HW) the signal *mprotz* causes the register *mprotout* to be cleared, see Figure 4.4. This results in $mprotout(i)^0 = 00000$ and the lemma is valid for cycle $t = 0$.

Induction Hypothesis:
Let us assume the statement $\neg H(i)^{t-1} \wedge \neg H'(i)^{t-1} \Rightarrow mprotout(i)^{t-1} = 00000$ holds.

Induction Step:
For $t > 0$ the case $z(i)^t \notin H \wedge z(i)^t \notin H'$ is assumed, because otherwise there is nothing to show. Then there are three different cases to consider.

$H(i)^{t-1}$**:** This case implies the master just left state $z(i)^{t-1} = w \vee mupdate$. During the last cycle in these states the register *mprotout* is clocked and the signal *mprotz* is activated. This results in $z(i)^t = idle \Rightarrow mprotout(i)^{t-1} = 00000$. The register is not clocked again until one of the automata transitions (4), (15) or $flush \wedge (6)$, which would lead into the hot phase in the consequent cycle.

$$mprotoutce^{t-1} \wedge mprotz^{t-1} \implies mprotout(i)^t = 00000$$

$H'(i)^{t-1}$**:** From this case follows that the master automaton just left state $mlr1$ or $mupdate$. According to the specification, the signals *mprotoutce* and *mprotz* are activated in state $mlr1$ and the last cycle in state $mupdate$ $(t-1)$. This gives us

$$mprotoutce^{t-1} \wedge mprotz^{t-1} \wedge \implies mprotout(i)^t = 00000$$

$\neg H(i)^{t-1} \wedge \neg H'(i)^{t-1}$**:** Regarding the automata construction follows

$$z(i)^{t-1} \notin \{flush,\ wait,\ wait'\} \wedge z(i)^t \notin \{m0,\ mlr0\}$$

Therefore the register is not clocked in cycle $t-1$ by (HW) and we can conclude

$$\begin{aligned} mprotout(i)^t &= mprotout(i)^{t-1} && (HW) \\ &= 00000 && (IH) \end{aligned}$$

■

5.3 Automata Synchronization

During the hot phase of the protocol, the master and slave automata will exchange the protocol signals over the bus. The respective interactive protocol assumes a certain order. In particular for a global request, first the master computes and broadcasts the *mprot* signals and the slaves answer with the slave response *sprot*. These signals are then used to determine the next master state.

Obviously this interaction can only work out, if the master and slave are in sync during this critical phase. In this section we are going to see that this condition is indeed satisfied by the presented design.

If no cache is in the hot phase, all caches are locally serving their processor requests. Accordingly the corresponding slave automata remain in state *sidle* until the critical phase starts. The following lemmas are proven together and therefore form a single induction hypothesis.

Lemma 5.9 (idle slaves [8, Lemma 8.21])
If no automaton is in a hot phase, then all slaves are idle.

$$(\forall i : \neg H(i) \wedge \neg H'(i)) \implies \forall j : sidle(j)$$

PROOF. by induction over cycle t.

Induction Base:
Any automaton i is initially in state $idle$. This state is not in the set of the *hot* or *hot'* phase ($idle(i)^0 \notin H(i) \wedge \notin H'(i)$), hence there is nothing to show.

Induction Step:
It is necessary to argue about all states, which will require the subsequent lemmas. Thats why the proof of the induction step is moved to the end of the section. ■

Now let us consider the automata synchronization between the master and slave during the hot phase.

Lemma 5.10 (sync [8, Lemma 8.22])
Consider a *hot* phase of the master i - lasting from cycles t to t', i.e we have

$$\neg H(i)^{t-1} \wedge H(i)^{[t:t']} \wedge \neg H(i)^{t'+1}$$

Then,

1. For the master i we have

$$m0(i)^t \wedge m1(i)^{t+1} \wedge m2(i)^{t+2} \wedge m3(i)^{t+3} \wedge mdata(i)^{[t+4:t''-1]}$$
$$\wedge\, w(i)^{t''} \wedge \begin{cases} idle(i)^{t'+1} & : (10)(i)^{t''} \rightarrow t'' = t' \\ mupdate(i)^{[t''+1:t']} \wedge idle(i)^{t'+1} & : (11)(i)^{t''} \rightarrow t'' < t' \end{cases}$$

2. The slave automaton of cache i does not leave state *sidle*.

$$sidle(i)^{[t:t'+1]}$$

3. Not affected slaves, i.e. slaves j with $j \neq i$ and $\neg bhit(j)^{t+1}$, will go from state $s1$ to $sidle'$ and remain there until the end of the hot phase.

$$sidle(j)^{t} \wedge s1(j)^{t+1} \wedge sidle'(j)^{[t+2:t']} \wedge sidle(j)^{t'+1}$$

4. The affected slaves, i.e the slaves j with $j \neq i$ and $bhit(i)^{t+1}$, run in sync with the master of the transaction.

$$\begin{aligned} &sidle(j)^{t} \wedge s1(j)^{t+1} \wedge s2(j)^{t+2} \wedge s3(j)^{t+3} \wedge \ sdata(j)^{[t+4:t''-1]} \\ &\wedge sw(j)^{t''} \wedge \begin{cases} sidle(j)^{t'+1} & : (18)(j)^{t''} \rightarrow t'' = t' \\ supdate(j)^{[t''+1:t']} \wedge sidle(j)^{t'+1} & : (19)(j)^{t''} \rightarrow t'' < t' \end{cases} \end{aligned}$$

Lemma 5.11 (sync for H$'$)
Analogue to the previous lemma, let us consider the *hot*$'$ phase of master i - lasting from cycles t to t', i.e. exactly the cycles meanwhile the master is in the states $mlr0$ and $mlr1$ or *muptdate*.

$$\neg H'(i)^{t-1} \wedge H'(i)^{[t:t']} \wedge \neg H'(i)^{t'+1}$$

Then,

1. For the master i we obtain

$$\begin{aligned} &mlr0(i)^{t} \wedge \begin{cases} mlr1(i)^{t+1} & : (16)(i)^{t} \rightarrow t' = t+1 \\ mupdate(i)^{[t+1:t']} & : (17)(i)^{t} \rightarrow t' \geq t+1 \end{cases} \\ &\wedge idle(i)^{t'+1} \end{aligned}$$

2. The slave automaton corresponding to the current master does not leave state *sidle*.

$$sidle(i)^{[t:t'+1]}$$

3. All slaves $j \neq i$ with a cache hit are running in sync with the master

$$\begin{aligned} &sidle(j)^{t} \wedge \begin{cases} slr1(j)^{t+1} & : (20)(j)^{t} \rightarrow t' = t+1 \\ supdate(j)^{[t+1:t']} & : (21)(j)^{t} \rightarrow t' \geq t+1 \end{cases} \\ &\wedge sidle(j)^{t'+1} \end{aligned}$$

   Note that even slaves with a cache miss go into state $slr1$ or *mupdate*, because $ca(j).badin$ is clocked in cycle t. Hence the hit signal *bhit* is not available until cycle $t+1$. In such a case ($\neg bhit$) the slave simply does nothing in the respective state and goes back to *sidle* in the next cycle. Therefore, the slave state RAM is only written in case of a hit.

PROOF. Part (1) follows directly from the master automaton construction. The parts (2) and (3) together with Lemma 5.9 (idle slaves) are proven by induction over t.

Induction Hypothesis:
Let for all cycles $q \leq t$ the following two equations hold:

- $\forall q < t : (\forall i : \neg H(i)^q \wedge \neg H'(i)^q) \implies \forall j : sidle(j)^q$
- $\forall q < t : \forall q' : \neg H'(i)^{q-1} \wedge H'(i)^{[q:q']} \wedge \neg H'(i)^{q'+1} \implies part(2) \wedge part(3)$

Induction Step:
First the induction step of Lemma 5.11 (sync for H') is considered, in particular $\neg H'(i)^{t-1} \wedge H'(i)^t$. Let us assume idle slaves at cycle $t-1$ by Lemma 5.9 (idle slaves). The automata construction implies

$$wait'(i)^{t-1} \wedge grant[i]^{t-1}$$

According to Lemma 5.5 (grant unique) only one cache can have grant during this cycle $\forall j \neq i : \neg grant[j]^{t-1}$. Moreover from Lemmas 5.4 and 5.3 (grant at warm and warm') follows that these caches without grant cannot be in the warm phase and therefore not in the hot phase.

$$(\neg W(j)^{t-1} \rightarrow \neg H(j)^{t-1}) \wedge (\neg W'(j)^{t-1} \rightarrow \neg H'(j)^{t-1})$$

By Lemma 5.9 (idle slaves) all slave automata are in state *sidle*, when no master is in a warm phase ($wait' \notin W' \cup W$).

$$\forall j : sidle(j)^{t-1}$$

Now with use of Lemma 5.8 (silent master), part (1) of Lemma (sync for H') and the construction of the automata and hardware one can conclude for $q \in [t-1 : t']$

$$b.mprot^q = \bigvee_j mprotout^q = \begin{cases} 00001 & : q = [t : t'] \\ 00000 & : q = t-1 \end{cases}$$

From this follow parts (2) and (3) of Lemma (sync for H') by construction of the slave automaton as presented in Section 4.3.

In particular part (2) is fulfilled, because the transitions to leave *sidle* contain the condition $\neg grant[j]$, which is not fulfilled for the master.

Part (3) is satisfied as the slave automaton has to choose transition (20) or (21) due to the value of *b.mprot*. These transitions will lead into state *slr*1 or *supdate*. Moreover the slave cannot go to state *s*1 because of signal $Ca = 0$. But the automaton will progress into state *slr*1, if the condition $b.mprot.wms \wedge \neg grant[j] \wedge \neg ca(j).s[5](b.ad)$ is fulfilled and into state *supdate* in case of $b.mprot.wms \wedge \neg grant[j] \wedge ca(j).s[5](b.ad)$. The slaves go back into *sidle* in case of a cache miss or the next cycle after $slr1(j)^{t+1}$ or when the memory access is completed in $supdate(j)^{t'}$.

Now only the induction step of Lemma (idle slaves) is left to show. For this purpose cycle t is considered, during which no cache is in a hot phase $\forall i : \neg H(i)^t \wedge \neg H'(i)^t$. There are two cases to consider depending on the state of the previous cycle $t-1$.

$\forall i : \neg H(i)^{t-1} \wedge \neg H'(i)^{t-1}$ From the induction hypothesis it follows that all slave automata are in state *sidle* during cycle $t-1$, as no master is in a hot phase.

$$\forall j : sidle(j)^{t-1}$$

Furthermore this implies that the master protocol signals are all zeros respective to Lemma 5.8 (silent master).

$$b.mprot^{t-1} = 00000$$

Then by construction of the slave automaton follows Lemma (idle slaves).

$\exists i : H(i)^{t-1} \vee H'(i)^{t-1}$ From Lemma 5.6 (warm unique) it follows that this i is unique. According to the master automaton construction, this master i has to be in the last cycle $t-1$ of a hot phase H or H'. Now, regarding the slave states one can easily apply the Lemmas (sync and sync for H′) and conclude by part (3)

$$\forall j : sidle(j)^t$$

■

In addition to that, at most one slave is going to take partly control of the bus. In particular in case of data intervention, one slave uses the *bdata* tristate wire. Then, the master will not broadcast any data, but only receive the slave data through its input register *bdin*.

Lemma 5.12 (di unique [8, Lemma 8.23])
Only one slave will provide data for data intervention during a *global* request.

$$SINV(t) \wedge sprotout(i).di^t \wedge sprotout(j).di^t \rightarrow i = j$$

5.4 Tristate Drivers

In this section, we will consider for each register $X(i)$ - connected to a memory bus component $b.Y$ through a tristate driver - the cycles t during which the signals are put on the bus, i.e. when

$$X(i)^t = b.Y^t$$

The function Cy is defined to contain the set of cycles, when $X(i)$ is put on the bus.

$$Cy(X,i) = \{t | Xoe(i)^t\}$$

When using tristate drivers, it is mandatory to show absence of bus contention to gain correct behavior. In other words it is required that at most one driver is enabled in the same cycle in the detailed hardware model.

The tristate bus is only used during the warm phases of the master states. In particular the signals are controlled by the current master or the intervening slave automaton in case of data intervention. The signal *bdout* is additionally treated in the end of the next section in Lemma 5.24. For now let us consider the remaining signals of the warm phases.

Lemma 5.13 (no contention [8, Lemma 8.25])
Assume signal X is activated by the master i during its *warm* or *warm*$'$ phase.

$$\forall i : Cy(X,i) \rightarrow W(i) \vee W'(i)$$

Then,

$$i \neq j \rightarrow Cy(X,i) \cap Cy(X,j) = \emptyset$$

PROOF. by contradiction.
Let us assume two tristate drivers $i \neq j$ for the same signal X are activated at the same cycle t.

$$\exists t : t \in Cy(X,i) \cap Cy(X,j)$$

By application of the hypothesis that would mean

$$t \in (W(i)^t \cup W'(i)^t) \cap (W(j)^t \cup W'(j)^t)$$

But according to Lemma 5.6 (warm unique) there can be only one automaton in a warm phase at the same time. This implies $i = j$. Therefore we have a contradiction to the assumption $i \neq j$. ■

Now we will consider when exactly the drivers are enabled and the signals put on the bus. For this purpose a lemma for each signal X - connected via a tristate driver to the bus - is provided.

Regarding the automata from Figure 4.5 it becomes obvious that certain states may last for several cycles. In these cases it is not sufficient to use the previous cycle $t-1$ to argue about values of the previous state. Instead the function ez will denote the cycle before the master entered the current state $z(i)^t$.

$$ez(t,i) = \max\{t' \mid t' < t \wedge z(i)^{t'} \neq z(i)^t\}$$

Lemma 5.14 (mmw [8, Lemma 8.28])
The main memory is only written by the master in states *flush*, *mupdate* and *emupdate*. The slave may issue the memory update in state *supdate* on a read miss due to a write mode switch to write through. In other words it is in state modified $M = 10000$ or owned $O = 01000$.

$$\begin{aligned} t \in Cy(mmw,i) \iff & flush(i)^t \vee emupdate^t \\ & \vee mupdate(i)^t \wedge (mprotout(i).im^{ez(t,i)} \vee mlr0(i)^{t-1}) \\ & \vee supdate(i) \wedge (\neg mprotin(i).im^{ez(t,i)} \wedge sprotout(i).di^{ez(t,i)} \\ & \quad \wedge ca(i).s(badin^{ez(t,i)})^{ez(t,i)} \in \{110000, 101000\}) \end{aligned}$$

PROOF. Both implication directions of the equivalence are shown independently. First let us consider the more general statement

$$t \in Cy(mmw,i) \implies flush(i)^t \vee emupdate(i)^t \vee mupdate(i)^t \vee supdate(i)^t$$

Let $[t : t'] \subset Cy(mmw,i)$ be a maximal interval with

$$\neg mmwoe(i)^{t-1} \wedge \forall q \in [t : t'] : mmwoe(i)^q \wedge \neg mmwoe(i)^{t'+1}$$

Then the statement $flush(i)^q \vee emupdate(i)^t \vee mupdate^q \vee supdate(i)^q$ is proven via induction for cycles $q \in [t : t']$.

Induction Base:
We have for the base case $q = t$. Furthermore it follows from the hardware construction $mmwoeset(i)^{t-1}$ as the register - containing the enable signal *mmwoe* for the tristate driver - has to be clocked with value 1, the cycle before the driver is active. The automata construction implies the following constraint for the interval $[t : t']$:

$$\begin{aligned} & wait(i)^{t-1} \wedge (5)(i)^{t-1} \wedge flush(i)^t \;\vee\; wait'(i)^{t-1} \wedge (14)(i)^{t-1} \\ & \vee (mlr0(i)^{t-1} \wedge (17)(i)^{t-1} \vee w(i)^{t-1} \wedge (11)(i)^{t-1}) \wedge mupdate(i)^t \\ & \vee sw(i)^{t-1} \wedge (19)(i)^{t-1} \wedge supdate(i)^t \end{aligned}$$

Induction Hypothesis:

Let us assume that for $q-1$ the following statement is true:

$$\begin{aligned} & q-1 \in Cy(mmw,i) \\ & \quad \implies flush(i)^{q-1} \vee emupdate(i)^{q-1} \vee mupdate(i)^{q-1} \vee supdate(i)^{q-1} \end{aligned}$$

Induction Step:

For $q > t$, we have for the previous cycle either $flush(i)^{q-1}$, $emupdate(i)^{q-1}$, $mupdate(i)^{q-1}$ or $supdate(i)^{q-1}$ by application of the induction hypothesis and $mmwoe(i)^q$ from the definition of the function Cy. Now the following cases result from the automata construction:

$flush(i)^{q-1}$: The memory access is not yet completed and hence not acknowledged by the main memory. In this case the master remains in state flush.

$$\neg b.mmack^{q-1} \wedge flush(i)^q$$

If *b.mmack* is true, the signal *mmwclear* is activated, which will disable the driver output to the bus. Such that we can conclude

$$flush(i)^{t'} \wedge m0(i)^{t'+1}$$

$emupdate(i)^{q-1} \vee mupdate(i)^{q-1} \vee supdate(i)^{q-1}$:
Similar to the previous case, the master remains in state $emupdate$ or $mupdate$ until the memory access is completed. Similar the intervening slave does not leave state $supdate$ until the access is completed.

$$\neg b.mmack^{q-1} \wedge emupdate(i)^q$$
$$\neg b.mmack^{q-1} \wedge b.mmreq^{q-1} \wedge mupdate(i)^q$$
$$\neg b.mmack^{q-1} \wedge b.mmreq^{q-1} \wedge supdate(i)^q$$

The master or slave clears the write signal with *mmwclear* during the last cycle in state $emupdate$, $mupdate$ and $supdate$ (signaled by $b.mmack^{t'}$). This leads to the conclusion

$$(emupdate(i)^{t'} \vee mupdate(i)^{t'} \vee supdate(i)^{t'}) \wedge idle(i)^{t'+1}$$

From this follows the implication

$$t \in Cy(mmw,i) \implies flush(i)^t \vee emupdate(i)^t \vee mupdate(i)^t \vee supdate(i)^t$$

For the reverse direction the following implication has to hold.

$$\left.\begin{array}{r} flush(i)^t \vee emupdate^t \\ \vee mupdate(i)^t \wedge (mprotout(i).im^{ez(t,i)} \vee mlr0(i)^{t-1}) \\ \vee supdate(i) \wedge (\neg mprotin(i).im^{ez(t,i)} \wedge sprotout(i).di^{ez(t,i)} \\ \wedge ca(i).s(badin^{ez(t,i)})^{ez(t,i)} \in \{110000, 101000\}) \end{array}\right. \implies t \in Cy(mmw,i)$$

The truth of this statement can be easily shown by consideration of the automata construction. In particular the driver for the signal *mmw* is enabled by setting *mmwset* when entering one of the four automata states $flush$, $emupdate$, $mupdate$ or $supdate$.

Moreover, there is no data contention by Lemma 5.18 (no contention), as for the first three states *mmw* is activated by the master during its warm phase. For the latter case only the intervening slave activates *mmwoe*, which is unique by Lemma 5.12 (di unique). The conditions $mprotout(i).im$ and $\neg mprotin(j).im$ exclude each other such that $Cy(mmw,i) \cap Cy(mmw,j) = \emptyset$ for the master i and slave j. ∎

The proofs of the lemmas for the other signals follow very much the same pattern. One is just arguing about the automata construction to prove during which cycles the drivers are activated or not. Therefore (for reasons of clarity and comprehensibility), only the lemmas are stated without proof for the remaining signals.

There are only five reasons to request access to the main memory. The first one is a read access of a memory line, which is not cached by the master and no slave provides it for data intervention.

Another possibility is that data intervention for a read takes place, but the write policy of the respective line changes from write back to write through. For this case it was specified that the memory needs to be updated if the cache line is dirty.

The remaining cases are a flush access, a write in write through mode or a local read access, which changes the write policy of a dirty cache line from write back to write through.

Lemma 5.15 (mmreq [8, Lemma 8.29])
The five states during which the *mmreq* signal can possibly be active are either $mdata$, $sdata$, $flush$, $emupdate$ or $mupdate$.

$$\begin{aligned} t \in Cy(mmreq,i) \iff & \\ & mdata(i)^t \wedge \neg(mprotout(i).bc^{ez(t,i)} \vee sprotin(i).di^{ez(t,i)}) \\ & \vee\ mupdate(i)^t \wedge (mprotout(i).im^{ez(t,i)} \vee mlr0(i)^{t-1}) \\ & \vee flush(i)^t\ \vee\ emupdate(i)^t \\ & \vee supdate(i)^t \wedge (\neg mprotin(i).im^{ez(t,i)} \wedge sprotout(i).di^{ez(t,i)} \\ & \quad \wedge ca(i).s(badin^{ez(t,i)})^{ez(t,i)} \in \{110000, 101000\}) \end{aligned}$$

Lemma 5.16 (badout [8, Lemma 8.30])
The signal *badout* comes always from the master. In the original protocol it is active during the warm phase except for states w and $wait$. For the new protocol part, we need it additionally during the hot' phase $H'(i)$ and in the states $mupdate$ and $emupdate$. But state $mupdate$ is already in the set of warm and hot' states. This results in

$$t \in Cy(badout,i) \iff (W(i)^t \wedge \neg w(i)^t \wedge \neg wait(i)^t) \vee H'(i)^t \vee emupdate(i)^t$$

Moreover, the enable signal *badoutoe* will stay 1, for all the above specified states. But the content of the address register *badout* changes after flush. Because when the master entered state $flush$, the address of the line, which had to be written to the main memory was on the bus, whereas after flush the address of the current processor request is available through $b.ad$.

Both, masters and slaves are able to activate the signal *bdout*, namely for broadcast or data intervention purposes.

Lemma 5.17 (bdout [8, Lemma 8.31])
On one hand, the master may use *bdout* to broadcast its modified data to the slave caches or to update the main memory. On the other hand in case of data intervention, the intervening slave will provide the data by clocking it in the output register *bdout* and enabling the respective driver.

$$\begin{aligned} t \in Cy(bdout, i) \iff & mdata(i)^t \wedge mprotout(i).bc^{ez(t,i)} \\ & \vee\ sdata(i)^t \wedge sprotout(i).di^{ez(t,i)} \\ & \vee\ flush(i)^t \\ & \vee\ emupdate(i)^t \\ & \vee\ mupdate(i)^t \end{aligned}$$

The states $mdata$ and $sdata$ belong to the same cycles t. Therefore it is necessary to prove that only one of both cases applies at the same time. This is done with Lemma 5.24 in the next Section.

In Lemma 5.13 (no contention), it was proven that bus contention is absent for any signal X enabled by the master during its warm phase. Now one can formulate a more general statement if the data signal *bdout* is treated separately. From the automata construction, it becomes obvious that the drivers for these signals X are never activated simultaneously by multiple automata. This results in the following lemma.

Lemma 5.18 (no contention 2 [8, Lemma 8.32])
There is no contention on the tristate bus for any component connected to signal $X \in \{mmw, mmreq, badout\}$.

$$i \neq j \wedge X \neq bdout \implies Cy(X,i) \cap Cy(X,j) = \emptyset$$

The absence of contention for the signal *bdout* is shown separately in the later Lemma 5.24.

In the specification of main memory accesses the detailed hardware model is involved. For this it is important to have stable signal values during the cycles when they are used. That means in order to ensure clean memory operations it is necessary to prevent spikes. In [8] the respective lemmas were proven, which show the absence of spikes during enabled and disabled drivers. Such that we can conclude correct behavior for the employed tristate bus.

5.5 Protocol Data Transmission

Next we will consider the transmission and processing of protocol data. Protocol transmission happens during states $z \in \{m0, m1, m2, m3, mlr0, mlr1, mupdate\}$. For the first four of these states, the correctness is already proven in the reference [8, Section 8.5.5]. Such that only states $mlr0$, $mlr1$ and $mupdate$ are left to show.

Lemma 5.19 (before mlr0)
In the cycle before entering $mlr0$, the processor address and the output of circuit $CS1$ is clocked into the registers $badout$ and $mprotout$. Let $mlr0(i)^t \wedge \neg mlr0(i)^{t-1}$, then

$$\begin{aligned} badout(i)^t &= pa(i)^{t-1} \\ mprotout(i)^t &= 0000\,CS1(aca(i).s(pa(i)^{t-1})^{t-1}, ca(i).pmode^{t-1}) \end{aligned}$$

PROOF. From the automata construction the signal $phit$ is true in cycle $t-1$ and therefore

$$ca(i).s(pa(i)^{t-1})^{t-1} = aca(i).s(pa(i)^{t-1})^{t-1}$$

The output of the state RAM is used as input for circuit $CS1$. The processor inputs are required to be stable, such that the lemma simply follows by automata and hardware construction. ■

Lemma 5.20 (mlr0)
During $mlr0$, the protocol data and the bus address of the master are broadcast. Meanwhile the master register $mprotout(i)$ stays unchanged. Let $mlr0(i)^t$ hold, then for all $j \neq i$:

$$\begin{aligned} mprotout(i)^{t+1} &= mprotout(i)^t \\ mprotin(j)^{t+1} &= mprotout(i)^t \\ badin(j)^{t+1} &= badout(i)^t \end{aligned}$$

PROOF. The register $mprotout$ is not clocked again in state $mlr0$ such that the first equation follows directly from the automata construction.

For $mprotin(j)^{t+1}$ the master protocol signal from the bus during cycle t is used as input. According to the hardware construction, this master protocol signal is computed as OR over all signals $mprotout(k)$. There is only one automaton i, which commits $mprot \neq 00000$ by Lemmas 5.6 (warm unique) and 5.8 (silent master).

$$mprotin(j)^{t+1} = b.mprot^t = \bigvee_k mprotout(k)^t = mprotout(i)^t$$

Now the same Lemma (warm unique) together with Lemma 5.16 (badout) and 5.18 (no contention 2) is applied to determine the value of $badin$.

$$badin(j)^{t+1} = b.ad^t = badout(i)^t$$ ■

Lemma 5.21 (mlr1)
The register $mprotout(i)$ is set to zero during state $mlr1$. In contrast to a global transaction, no slave response is required. The slaves only test for a hit and update

the write mode with help of circuit $CS2$. Let $mlr1(i)^t$, then for all slaves j with $slr1(j)^t \wedge bhit(j)^t$:

$$\begin{aligned} mprotout(i)^{t+1} &= 00000 \\ aca(j).s^{t+1}(badin(j)^t) &= CS2(aca(j).s(badin(j)^t)^t, mprotin(j)^t[0]) \end{aligned}$$

PROOF. The predicate $bhit(j)$ implies that cache j has valid data for the bus address $badin$ and therefore

$$ca(j).s^t(badin(j)^t) = aca(j).s^t(badin(j)^t)$$

Then again the lemma follows from the hardware and automata construction. ■

Lemma 5.22 (mupdate)
The master protocol signals $mprotout(i)$ are cleared when leaving $mupdate$. The new slave state is written during the last cycle in state $supdate$. In particular, let $mupdate(i)^{[t:t']} \wedge b.mmack^{t'}$, then all slaves j with $bhit(j)$ are in state $supdate(j)^{[t:t']}$.

$$\begin{aligned} mprotout(i)^{t'+1} &= 00000 \\ mprotin(i)^{[t+1:t']} &= mprotin(i)^t \\ badin(i)^{[t+1:t']} &= badin(i)^t \\ aca(j).s^{t'+1}(badin(j)^t) &= CS2(aca(j).s(badin(j)^t)^t, mprotin(j)^t[0]) \end{aligned}$$

PROOF. Again $bhit(j)$ implies that cache j has valid data for the bus address $badin$ and therefore

$$ca(j).s^t(badin(j)^t) = aca(j).s^t(badin(j)^t)$$

Then again the lemma follows from the hardware and automata construction. ■

During the states $mdata$ and $sdata$ the master or a slave uses wire $bdout$ to commit its cache data. But it has to be ensured that only either the master broadcasts its data or the slave is intervening data.

Lemma 5.23 (no di after bc [8, Lemma 8.37])
The broadcast signal $mprotout(i).bc$ is only raised for a write hit of a shared cache line, whereas a data intervention is only necessary for a miss of the requested line. Let $m2(i)^t$, then for all j:

$$mprotout(i).bc^t \implies \neg sprotout(j).di^t$$

Remember Section 3.5, where $SINV(t-1)$ was defined, i.e. $SINV(t-1)$ implies that the state invariants hold until cycle $t-1$. Just as in the reference [8], a series of lemmas (x,t) for cycle t is necessary, which uses $SINV(t-1)$ as hypothesis. These lemmas are then used to show that the invariants hold for cycle t as well.

$$SINV(t-1) \wedge \bigwedge_x lemma(x,t) \rightarrow SINV(t)$$

The respective main induction hypothesis is

$$SINV(t) \wedge \bigwedge_x lemma(x,t)$$

Next, the absence of contention for *bdout* is shown. Due to Lemma 5.18 (no contention) it is ensured that no contention occurs for any signals driven by the master. But for the signal *bdout* we have seen that not only the master but also a slave cache or the main memory may activate its driver as response to a read request. Therefore we have to ensure that it is impossible to have multiple drivers enabled via *badoutoe* at the same cycle.

Lemma 5.24 (bdout contention [8, Lemma 8.38])
Assume $SINV(t-1)$. Then there is no contention on the data bus $b.d$ until cycle t:

$$\forall q \leq t : \forall j \neq i : q \in Cy(bdout, i) \implies q \notin Cy(bdout, j)$$

PROOF. induction over q, with $0 < q \leq t : q \in Cy(bdout, i)$.

Induction Base:
For $q = 1$ the states *idle*, *localw*, *wait* or *wait'* are possible according to the automata construction. This means the driver for *bdout* is not enabled $q \notin Cy(bdout, i)$, hence there is nothing to show.

Induction Hypothesis:

$$q-1, \forall j \neq i : q-1 \in Cy(bdout, i) \implies q \notin Cy(bdout, j)$$

Induction Step:
There are two cases regarding cycle $q-1$:

$q-1 \in Cy(bdout, i)$: there are only five states in which *bdoutoe* can be active, namely $flush, sdata, mdata, emupdate$ and $mupdate$. For all of these, the automaton i stays in the same state as for cycle $q-1$, if the memory access is not yet finished.

If *bdoutoe* was activated due to a read miss with data intervention - while the automaton was in state *mdata* - the automata will proceed directly into the next state in the next cycle.

For a read miss without data intervention, the main memory will provide the requested line Therefore neither the master, nor the slave will activate *bdoutoe* (no broadcast and no data intervention).

The master activates *bdoutoe* to broadcast the modified data. This is only done in case of a cache hit. Therefore the master advances into state w in the next cycle.

We conclude that an enabled *bdout* driver by the master in cycles $q-1 \in Cy(bdout, i)$ and $q \in Cy(bdout, i)$ is only possible for the states $flush$, *emupdate* and *mupdate*.

$$q-1 \in Cy(bdout,i) \wedge q \in Cy(bdout,i)$$
$$\implies flush(i)^q \vee emupdate(i)^q \vee mupdate(i)^q$$

If we assume $bdoutoe(j)^q$, i.e. $q \in Cy(bdout,j)$ for a different cache $j \neq i$, this implies

$$\begin{aligned} flush(j)^q \vee sdata(j)^q \vee mdata(j)^q \\ \vee emupdate(j)^q \vee mupdate(j)^q \end{aligned} \tag{5.25}$$

Lemma 5.12 (di unique) and $SINV(t-1)$ ensure that only one slave cache can have an active signal $ca(j).sprotout.di$, which implies an enabled *bdout* driver. The driver is disabled when leaving *sdata*, such that we have no contention $Cy(bdout,i) \cap Cy(bdout,j) = \emptyset$. By Lemma 5.6 (warm unique) we know any combinations with $flush(j) \vee mdata(j) \vee emupdate(j) \vee mupdate(j)$ are impossible for $i \neq j$. Due to this reason there is no $j \neq i$ with $q \in Cy(bdout,j)$.

$q-1 \notin Cy(bdout,i)$: In addition to the previous scenario, we have to consider the case of one cycle long states *sdata* and *mdata* with data intervention or data broadcast. Then, the main memory is not accessed at all and the automata directly proceed to the next state in the next cycle. Therefore the automaton i can be in one of the following states:

$$flush(i)^q \vee sdata(i)^q \vee mdata(i)^q \vee emupdate(i)^q \vee mupdate(i)^q$$

Again the lemma would be invalid, if for any automaton $j \neq i$ Equation (5.25) is true. Now, the same argumentation as before applies. The only combinations which are not captured yet, cover the cases of $mdata(i)^q$ and $sdata(i)^q$. For these, the states $flush(j)^q$, $mdata(j)^q$, $emupdate(i)^q$ and $mupdate(i)^q$ are impossible by Lemma 5.6 (warm unique). Meanwhile, the previous Lemmas 5.23 (no di after bc) and 5.12 (di unique) exclude possible combinations of $sdata(i)^q$ with $sdata(j)^q \vee mdata(j)^q$.
■

5.6 Data Transmission

In this Section multiple lemmas are listed, which concern the transfered data. These ensure that the specifications for the main memory system are satisfied. In particular they reveal, in which states and cycles the memory is updated or the registers are written. For all of the lemmas in this section $SINV(t-1)$ is assumed.

Lemma 5.26 (flush transfer [8, Lemma 8.39])
Let us consider a maximal time interval $[s:t]$, during which a master i is in state flush.

$$\neg flush(i)^{s-1} \wedge \forall q \in [s:t] : flush(i)^q \wedge \neg flush(i)^{t+1}$$

Then the data of $bdout(i)^s$ is written at address $badout(i)^s$ to the main memory.

$$mm(badout(i)^s)^{t+1} = bdout(i)^s$$

Lemma 5.27 (mupdate transfer)
Again, let $[s:t]$ be a maximal time interval in master state $mupdate$.

$$\neg mupdate(i)^{s-1} \wedge \forall q \in [s:t] : mupdate(i)^q \wedge \neg mupdate(i)^{t+1}$$

Then the memory at address $badout(i)^s$ contains the new value $bdout(i)^s$ after cycle t.

$$b.mmw^{[s:t]} \implies mm(badout(i)^s)^{t+1} = bdout(i)^s$$

PROOF. This lemma follows the lines of the proof for flush as presented in [8, L. 8.39]. ■

Lemma 5.28 (mdata write hit [8, Lemma 8.40])
In this state the master transfers data to the slave caches in case of a broadcast.

$$mprotout(i).bc^{t-1} \wedge mdata(i)^t$$

In particular the value $bdout(i)^t$ is broadcast to all slave caches, which have a hit and are therefore in state $sdata$.

$$\forall j : sdata(j)^t \implies bdin(j)^{t+1} = bdout(i)^t$$

Lemma 5.29 (mdata data intervention [8, Lemma 8.41])
If the master has a cache miss for a cache line, which is owned by another cache, the owner will provide the respective data.

$$mdata(i)^t \wedge sprotout(j).di^{t-1}$$

Then the data $ca(j).bdout^t$ of the intervening slave j is transferred to the master i.

$$bdin(i)^{t+1} = bdout(j)^t$$

Lemma 5.30 (mdata miss no intervention [8, Lemma 8.42])
Assume the master automaton i is in state $mdata$ for a maximal interval of cycles $[s:t]$. Furthermore there is no hit and no data intervention for the requested cache line in cycle $s-1$.

$$\neg mprotout(i).bc^{s-1} \wedge \neg sprotin(i).di^{s-1}$$

Then, the memory is accessed by the master to fetch the line $mm^s(badout(i)^s)$.

$$bdin(i)^{t+1} = mm^s(badout(i)^s)$$

Classification of accesses		
by hardware execution	**by atomic execution**	**by operation type**
local read	local read	read, negative CAS
delayed local read	local read	negative CAS
local read with memory update	global′ read	(read , negative CAS) & write through
local read with wms broadcast	msbroadcast	read, negative CAS
local write	local write	(write, positive CAS) & write back
local write with memory update	global′ write	(write, positive CAS) & write through
global access	global access	write, read, CAS
memory update	global or global′ access	(write, read, CAS) & write through
flush access	flush access	flush access

Table 5.1: Classification of accesses

5.7 Hardware Computation

This section outlines the relation between the hardware computation from Chapter 4 and the atomic specification of Chapter 3. In particular a series of accesses $acc(i,k)$ is constructed from the respective hardware computation. These accesses are then used to prove necessary properties dependent of their type.

Previously there were two classifications for accesses. They can be grouped by the type of their operation (write, read or CAS) or depending on the way they are treated in the atomic protocol in Section 3.6. In this section a third classification of accesses is presented, namely depending on the way they are treated by the hardware. These three classifications are shown in Table 5.1.

In order to show sequential consistency we have to argue that certain accesses cannot overlap and some sequential order of accesses is observable. For this purpose start and end cycles of a hardware access $acc(i,k)$ have to be introduced.

In general the end of a read, write or CAS access is defined as the cycle t, when the busy signal $mbusy(i)^t$ is off and the processor request signal $preq(i)^t$ is still active. As in [8], a flush access starts and ends by entering and leaving the state flush.

Formally, the predicate $somend(i,t)$ is true, if such access $acc(i,k)$ ends in cycle t.

$$someend(i,t) \equiv preq(i)^t \wedge \neg mbusy(i)^t \vee flush(i)^t \wedge \neg flush(i)^{t+1}$$

Then, the end cycle $e(i,k)$ for some cache i of the k-th access - belonging to a series of accesses - is defined by

$$e(i,k) \equiv \begin{cases} min\{t \mid someend(i,t)\} & k=0 \\ min\{t \mid t > e(i,k-1) \wedge someend(i,t)\} & k>0 \end{cases}$$

According to the automata construction an access can end in one of the following states:

$$idle(i)^{e(i,k)} \vee localw(i)^{e(i,k)} \vee flush(i)^{e(i,k)} \vee w(i)^{e(i,k)} \vee wait(i)^{e(i,k)} \\ \vee wait'(i)^{e(i,k)} \vee emupdate(i)^{e(i,k)} \vee mlr1(i)^{e(i,k)} \vee mupdate(i)^{e(i,k)}$$

Taking these states into consideration, there are different groups depending on how an access is treated by the hardware.

- An access (i,k) is a local read if it ends in state *idle*.

$$idle(i)^{e(i,k)}$$

- An access (i,k) is a delayed local read if it ends in state *wait* or *wait'*.

$$wait(i)^{e(i,k)} \vee wait'(i)^{e(i,k)}$$

- An access is a local read of an exclusive or modified cache line with consequent memory update, if it ends in state *emupdate*.

$$rglobal'(i)^{e(i,k)} \wedge emupdate(i)^{e(i,k)}$$

- An access is a local read of a shared line with a write mode switch to write back, if it ends in state *mlr*1.

$$mlr1(i)^{e(i,k)}$$

- An access (i,k) is a local write, if it ends in state *localw*.

$$localw(i)^{e(i,k)}$$

- An access (i,k) is a local write in write through mode, if it ends in state *emupdate*.

$$wglobal'(i)^{e(i,k)} \wedge emupdate(i)^{e(i,k)}$$

- An access (i,k) is a global access if it ends in state *w*.

$$w(i)^{e(i,k)}$$

- The main memory update for a *global* or *global'* access (i,k) in write through mode ends in state *mupdate*.

$$mupdate(i)^{e(i,k)}$$

- An access (i,k) is a flush, if it ends in state *flush*.

$$flush(i)^{e(i,k)}$$

The accesses belonging to the states *emupdate* or *mlr*1 are summarized as *global'* accesses. Then again, local reads, delayed local reads and local writes in write back mode are categorized as *local* accesses.

In the sequel the following abbreviations are used:

$$\begin{aligned} global(i,k,aca) &= global(aca,acc(i,k),i) \\ global'(i,k,aca) &= global'(aca,acc(i,k),i) \\ local(i,k,aca) &= local(aca,acc(i,k),i) \end{aligned}$$

Furthermore the arguments *k* and *aca* are omitted if they are clear from the context.

Now the start cycles $s(i,k)$ of accesses are considered. Local reads are starting and ending in the same cycle. The same holds for delayed local reads. In contrast to that, *global'* accesses start, when their *warm'* phase begins. Accesses ending in *mlr*1 start 1 cycle before they end. A flush starts when the master enters state *flush*. Last but not least, *global* accesses start in the cycle, when their *hot* phase begins - namely when entering state *m*0. Let $t = e(i,k)$, then

$$s(i,k) \equiv \begin{cases} t & : idle(i)^t \vee wait(i)^t \vee wait'(i)^t \\ t-1 & : localw(i)^t \vee mlr1(i)^t \\ max\{q | q < t \wedge wait'(i)^q\} + 1 & : emupdate \vee mupdate(i)^t \wedge global'(i)^t \\ max\{q | q < t \wedge wait(i)^q\} + 1 & : flush(i)^t \\ max\{q | q < t \wedge m0(i)^q\} & : w(i)^t \vee mupdate(i)^t \wedge global(i)^t \end{cases}$$

Accesses start only, when no snoop conflict exists. Now, from the construction these starting states follow:

$$\begin{aligned} &idle(i)^{s(i,k)} \vee wait(i)^{s(i,k)} \vee wait'(i)^{s(i,k)} \\ &\qquad \vee flush(i)^{s(i,k)} \vee m0(i)^{s(i,k)} \vee emupdate(i)^{s(i,k)} \vee mlr0(i)^{s(i,k)} \end{aligned}$$

By construction one easily gets the following lemma - regarding the start and end cycle of accesses.

Lemma 5.31 (local order [8, Lemma 8.43])

$$\forall k : s(i,k) \leq e(i,k) < s(i,k+1)$$

The parameters for an access $acc(i,k)$ of the sequential computation are defined with the help of the end cycles. The two cases from [8] are considered.
First for a flush access $flush(i)^t$ with $t = e(i,k)$ the access components are determined with

$$\begin{aligned} acc(i,k).a &= badout(i)^t \\ acc(i,k).r &= acc(i,k).w \;=\; acc(i,k).cas \;=\; 0 \\ acc(i,k).f &= 1 \\ acc(i,k).m &= pmode(i)^t \end{aligned}$$

For all other cache accesses, $acc(i,k)$ is constructed from the processor input at the end of the hardware access. Note that these inputs do not change until the signal $preq$ is lowered again.

$$\begin{aligned} acc(i,k).a &= pa(i)^t \\ acc(i,k).data &= pdin(i)^t \\ acc(i,k).cdata &= pcdin(i)^t \\ acc(i,k).bw &= pbw(i)^t \\ acc(i,k).w &= pw(i)^t \\ acc(i,k).r &= pr(i)^t \\ acc(i,k).cas &= pcas(i)^t \\ acc(i,k).f &= 0 \\ acc(i,k).m &= pmode(i)^t \end{aligned}$$

For accesses different from flush the last cycle $d(i,k)$ - when the decision was made to treat a request local or global - is necessary for the consequent lemmas.

$$d(i,k) = \begin{cases} \max\{q \mid q \leq s(i,k) \wedge wait(i)^q\} & : acc(i,k).cas \wedge (mupdate(i)^{e(i,k)} \\ & \quad \wedge global(i)^{e(i,k)} \vee w(i)^{e(i,k)} \vee wait(i)^{e(i,k)}) \\ \max\{q \mid q \leq s(i,k) \wedge wait'(i)^q\} & : wait'(i)^{e(i,k)} \vee global'(i)^{e(i,k)} \\ \max\{q \mid q \leq s(i,k) \wedge idle(i)^q\} & : \text{otherwise} \end{cases}$$

Lemma 5.32 (global$'$ end cycle)
Let $acc(i,k)$ be an access, then the following three statements hold.

1. If the predicate $global'$ is true in cycle $d(i,k)$, then the access ends in one of the states $mlr1$, $emupdate$ or $mupdate$.

$$\begin{aligned} & SINV(e(i,k)) \wedge global'(i,k,aca^{d(i,k)}) \\ & \qquad \Longrightarrow broadcastms(aca^{d(i,k)}, acc(i,k), i) \wedge mlr1(i)^{e(i,k)} \\ & \qquad\qquad \vee\; rglobal'(aca^{d(i,k)}, acc(i,k), i) \wedge emupdate(i)^{e(i,k)} \\ & \qquad\qquad \vee\; wglobal'(aca^{d(i,k)}, acc(i,k), i) \wedge emupdate(i)^{e(i,k)} \\ & \qquad\qquad \vee\; \neg acc(i,k).m \wedge mupdate(i)^{e(i,k)} \end{aligned}$$

2. An analogue statement applies for the reverse direction. For any access ending in one of the states $mlr1$ or $emupdate$, the test for $global'$ had to succeed in cycle $d(i,k)$.

$$mlr1(i)^{e(i,k)} \vee emupdate(i)^{e(i,k)} \implies global'(i,k,aca^{d(i,k)})$$

3. For the state $mupdate$ one can only conclude that either a $global$ or a $global'$ access is processed.

$$\begin{aligned} & mupdate(i)^{e(i,k)} \wedge SINV(e(i,k)) \\ & \quad \implies \neg acc(i,k).m \wedge (global(i,k,aca^{d(i,k)}) \vee global'(i,k,aca^{d(i,k)})) \end{aligned}$$

PROOF. *proof of statement (1):*
This statement is proven by contradiction. For this purpose we assume that a $global'$ access is not ending in the states as indicated by the lemma. In particular it is assumed that for $SINV(e(i,k)) \wedge global'(i,k,aca^{d(i,k)})$ the access $acc(i,k)$ is ending in state:

$$wait(i)^{e(i,k)} \vee wait'(i)^{e(i,k)} \vee idle(i)^{e(i,k)} \vee localw(i)^{e(i,k)} \vee w(i)^{e(i,k)}$$

Let us consider the different accesses which can end in these states.

- delayed local read ($wait(i)^{e(i,k)} \vee wait'(i)^{e(i,k)}$):
  According to the definitions the access starts and ends at the same cycle $s(i,k) = e(i,k)$ and the decision for a delayed local read is done in cycle $d(i,k) = s(i,k)$. By the automata construction this requires one of the transitions $(9)(i)^{d(i,k)} \vee (13)(i)^{d(i,k)}$. If transition (9) is taken the definition of $rlocal(i)^{d(i,k)}$ is fulfilled, which contradicts $global'(i)^{d(i,k)}$. The second transition directly requires $\neg global'(i)^{d(i,k)}$ which again contradicts to the assumption.

- local read ($idle(i)^{e(i,k)}$):
  A local read is starting and ending in the same cycle $s(i,k) = e(i,k)$ with $d(i,k) = s(i,k)$. This means that none of the transitions (2) (3) or (12) is taken by automaton i at cycle $d(i,k)$.

  $$\neg(2)(i)^{d(i,k)} \wedge \neg(3)(i)^{d(i,k)} \wedge \neg(12)(i)^{d(i,k)}$$

  The condition for (12) includes $global'(i)$. Such that from $\neg(12)(i)^{d(i,k)}$ follows $\neg global'(i)^{d(i,k)}$, which is in conflict with the assumption.

- local write ($localw(i)^{e(i,k)}$):
  This access is starting at cycle $s(i,k) = e(i,k) - 1$ with $d(i,k) = s(i,k)$. According to the automata construction, transition (2) is taken at cycle $d(i,k)$, which requires $pmode \wedge wlocal$. But an access - fulfilling the definition for $wlocal$ - cannot be $global'$. Therefore we have again the contradiction with $\neg global'(i)^{d(i,k)}$.

- global access ($w(i)^{e(i,k)}$):
  This case implies transition (4) or (6) was taken in cycle $s(i,k)-1$. But that means the automaton is going for a *global* transaction, i.e. $global(i)^{d(i,k)}$ holds. From the definitions of *global* and *global'* follows directly

$$global(i)^t \implies \neg global'(i)^t$$

  Therefore the end state $w(i)^{e(i,k)}$ would contradict with the assumption of $global'(i)^{d(i,k)}$.

proof of statement (2):
The definition gives us for the start cycle $s(i,k) = max\{q|q < e(i,k) \wedge wait'(i)^q\} + 1$ for $emupdate(i)^{e(i,k)}$. For $mlr1(i)^{e(i,k)}$ the access started one cycle before it ends $s(i,k) = e(i,k) - 1$. In addition to that, the decision cycle for these access types is defined with $d(i,k) = \max\{q \mid q \leq s(i,k) \wedge wait'(i)^q\}$.

Now let us assume $\neg global'(i)^{d(i,k)}$ in order to reveal a contradiction. But then the automaton i would not take transition (12) according to the automata construction. Furthermore this means the states $mlr1$ and $emupdate$ are not visited for access $acc(i,k)$.

$$\neg(mlr1(i)^{e(i,k)} \vee emupdate(i)^{e(i,k)})$$

But this would contradict the assumption and hence $global'(i)^{d(i,k)}$ has to hold.

$$mlr1(i)^{e(i,k)} \vee emupdate(i)^{e(i,k)} \implies global'(i,k,aca^{d(i,k)})$$

proof of statement (3):
Again the statement is proven by contradiction. Therefore

$$acc(i,k).m \vee \neg global(i)^{d(i,k)} \wedge \neg global'(i)^{d(i,k)}$$

is assumed for an access ending in state *mupdate*. The first condition $acc(i,k).m = 1$ is impossible by the automata construction, because only accesses in write through mode can lead to $mupdate(i)^{e(i,k)}$.

During the decision cycle the master is either in state *idle*, *wait'* or *wait*. The first case directly implies $global(i)^{d(i,k)}$, because otherwise condition $(3)(i)^{d(i,k)}$ is not satisfied. Furthermore the second case implies $global'(i)^{d(i,k)}$, because otherwise transition (13) would be chosen during cycle $d(i,k)$.
If we assume $\neg global(i)^{d(i,k)}$ for the last case, the master would go for transition $(9)(i)^{d(i,k)}$ and the access ends in state *wait* instead of *mupdate*.

Therefore we can conclude

$$\begin{aligned} &mupdate(i)^{e(i,k)} \wedge SINV(e(i,k)) \\ &\qquad \implies \neg acc(i,k).m \wedge (global(i,k,aca^{d(i,k)}) \vee global'(i,k,aca^{d(i,k)})) \quad \blacksquare \end{aligned}$$

Lemma 5.33 (global end cycle [8, Lemma 8.44])
Let $acc(i,k)$ be the considered access. Then

1. If the global test succeeds in cycle $d(i,k)$, the access ends in state w or $mupdate$:

$$global(i,k,aca^{d(i,k)}) \implies w(i)^{e(i,k)} \vee \neg acc(i,k).m \wedge mupdate(i)^{e(i,k)}$$

2. Similarly for the reverse direction, if the access ends in state w, the global test succeeded in cycle $d(i,k)$:

$$w(i)^{e(i,k)} \wedge SINV(e(i,k)) \implies global(i,k,aca^{d(i,k)})$$

PROOF. The statement is proven along the lines of the previous lemma (global′ end cycle). ■

Lemma 5.34 (local end cycle [8, Lemma 8.45])
Local accesses always end in state *localw*, *idle*, *wait* or *wait′*.

$$\begin{aligned} SINV(e(i,k)) \wedge local(i,k,aca^{d(i,k)}) & \\ \implies & rlocal(aca^{d(i,k)},acc(i,k),i) \\ & \quad \wedge (idle(i)^{e(i,k)} \vee wait(i)^{e(i,k)} \vee wait'(i)^{e(i,k)}) \\ & \vee wlocal(aca^{d(i,k)},acc(i,k),i) \wedge localw(i)^{e(i,k)} \end{aligned}$$

$$\begin{matrix} localw(i)^{e(i,k)} \vee idle(i)^{e(i,k)} \\ wait(i)^{e(i,k)} \vee wait'(i)^{e(i,k)} \end{matrix} \implies local(i,k,aca^{d(i,k)})$$

Lemma 5.35 (slave write at hot [8, Lemma 8.46])
The cache RAMs $X \in \{data, tag, s\}$ for caches $j \neq i$ - acting as slave in the protocol - are only written during the *hot* or *hot′* phase of a master i. In particular the write enable $Xwb(j)$ of slave j is not enabled outside of the *hot* or *hot′* phase.

$$grant[i]^t \wedge \neg H(i)^t \wedge \neg H'(i)^t \rightarrow \forall j : Xwb(j)^t = 0$$

PROOF. The signal Xwb is only enabled during state sw, $slr1$ or $supdate$. We have seen in Lemma 5.10 (sync) and 5.11 (sync for H′) that these slave states are only possible, if a mater is in state w, $mlr1$ or $mupdate$. This implies that some automaton r has to be in $H(r)$ or $H'(r)$.

But according to the Lemmas 5.4, 5.3 (grant at warm or warm′) and 5.5 (grant unique), only the automaton with *grant* can be in a warm phase. The hot phase is a subset of the warm phase, such that

$$\forall r \neq i : \neg grant[r]^t \implies \neg H(r) \wedge \neg H'(r)$$

That means that all slave automata have to be in state *sidle* by Lemma 5.9 (idle slaves).

$$\forall j : sidle(j)^t$$

Therefore, $Xwb(j)^t = 0$ is a consequence of the automata construction, which concludes the lemma. ∎

Lemma 5.36 (stable master [8, Lemma 8.47])
Assuming that $acc(i,k)$ is a flush, *global* or *global′* access:

$$acc(i,k).f \vee global(i,k,aca^{d(i,k)}) \vee global'(i,k,aca^{d(i,k)})$$

The cache is not written until the end cycle, i.e. for a memory update the cache RAM is clocked in the last cycle of state *emupdate* or *mupdate*. Therefore it does not change during the entire access.

$$\forall q \in [s(i,k) : e(i,k)] : aca(i)^q = aca(i)^{s(i,k)}$$

PROOF. In case of a flush, the cache RAMs are only written when the master automaton for cache i is leaving the state *flush*. This is the last cycle $e(i,k)$ of the access. The changed RAM values are then visible in the next cycle $e(i,k)+1$.

For a global access the clock signals - for the RAMs of the master cache - - are only enabled in one of the states $z(i)^{e(i,k)} \in \{w, emupdate, mlr1, mupdate\}$. In particular the cache RAMs are only written during the last cycle of the access. Then again, the change of the cache content is only visible in the next cycle after the access ended.

Last but not least, one has to show that the cache is not updated by the slave automaton. This is easy, because Lemma 5.35 (slave write at hot) ensures that the slave automaton only issues cache updates during the hot phase. In particular the cache is only updated during the slave states *sw*, *slr*1 or *supdate*. But the slave automaton for cache i is not leaving state *sidle* according to Lemmas 5.10 and 5.11 (sync and sync at H′). ∎

Lemma 5.37 (last cycle of wait [8, Lemma 8.48])
The cache is not modified in the cycle when the master enters its warm phase.

$$wait(i)^q \wedge \neg wait(i)^{q+1} \implies aca(i)^q = aca(i)^{q+1}$$

Lemma 5.38 (last cycle of wait′)
Cache i is not written during the last cycle q in state *wait′*.

$$wait'(i)^q \wedge \neg wait'(i)^{q+1} \implies aca(i)^q = aca(i)^{q+1}$$

PROOF. According to the automata construction, the master automaton does not update the cache in state *wait′*. For this reason only the slave could possibly write to the cache during cycle q. For this the slave automaton needs to be in state $z \neq sidle$

as stated in Lemma 5.35 (no slave at hot). But this means there has to be another master $j \neq i$ with $grant[j]$ due to Lemmas 5.4, 5.3 (grant at warm and warm′). This is impossible, as only one cache has grant at the same cycle by Lemma 5.5 (grant unique). ■

Lemma 5.39 (last cycle of flush [8, Lemma 8.49])
The automaton for cache i enters $flush$, if the cache line for address pa is currently occupied for another address. This means that the abstract cache state for pa is invalid. This is the same state which is clocked in the state RAM when leaving $flush$ during cycle t.

$$flush(i)^t \wedge \neg flush(i)^{t+1} \implies aca(i).s(ca(i).pa^t)^t = aca(i).s(ca(i).pa^t)^{t+1}$$

Lemma 5.40 (overlapping accesses with flush [8, Lemma 8.50])
Assume $acc(i,k)$ is a flush with address $a = acc(i,k).a$ ending at cycle $e(i,k) = t$. And let $acc(r,s)$ be any access - except a local read - to the same address a. Then the time intervals of these two accesses are disjoint. Thus, only local reads can overlap with flushes:

$$\begin{aligned}
&SINV(t) \wedge (i,k) \neq (r,s) \wedge acc(i,k).f \wedge e(i,k) = t \\
&\wedge\ \neg(rlocal(r,s,aca^{d(r,s)}) \wedge d(r,s) = s(r,s)) \\
&\wedge\ acc(r,s).a = acc(i,k).a \rightarrow [s(i,k) : e(i,k)] \cap [s(r,s) : e(r,s)] = \emptyset
\end{aligned}$$

In particular even delayed local reads are not possible, since the respective transitions $(9)(r)$ and $(13)(r)$ require $grant[r]$.

Lemma 5.41 (overlapping accesses with global [8, Lemma 8.51])
Assume $SINV(t)$. Let $acc(i,k)$ be a global access with address $a = acc(i,k).a$ ending at cycle $e(i,k) = t$. Let $acc(r,s)$ be an access with address a overlapping with $acc(i,k)$. Then $acc(r,s)$ is a local read and the overlap is in cycle $s(i,k)$ or $acc(r,s)$ is a local write and the overlap is in cycle $u \in \{s(i,k), s(i,k)+1\}$.

$$\begin{aligned}
&SINV(t) \wedge global(i,k,aca^{d(i,k)}) \wedge (i,k) \neq (r,s) \wedge \neg acc(r,s).f \\
&\wedge acc(i,k).a = acc(r,s).a \wedge u \in [s(i,k) : e(i,k)] \cap [s(r,s) : e(r,s)] \\
\implies\ &wlocal(aca^{d(r,s)}, acc(r,s), r) \wedge u \in \{s(i,k), s(i,k)+1\} \\
&\vee rlocal(aca^{d(r,s)}, acc(r,s), r) \wedge u = s(i,k) = s(r,s) = d(r,s)
\end{aligned}$$

Lemma 5.42 (stable local [8, Lemma 8.52])
Assume $acc(i,k)$ is a local access, which ends in cycle $e(i,k) = t$. Then the content of cache i is not changing during the access.

$$SINV(t) \wedge local(i,k,aca^{d(i,k)}) \implies aca(i)^{s(i,k)} = aca(i)^t$$

Lemma 5.43 (overlapping accesses with local write [8, Lemma 8.53])
Assume $acc(i,k)$ is a local write access at address $a = acc(i,k).a$, which ends at cycle $e(i,k) = t$. Then there cannot be another local access $acc(r,s)$ to the same

address, while the local write is processed.

$$\begin{array}{rl} & SINV(t) \wedge (i,k) \neq (r,s) \wedge e(i,k) = t \\ \wedge & local(i,k,aca^{d(i,k)}) \wedge localw(i)^t \\ \wedge & acc(i,k).a = acc(r,s).a \wedge local(r,s,aca^{d(r,s)}) \\ \rightarrow & [s(i,k) : e(i,k)] \cap [s(r,s) : e(r,s)] = \emptyset \end{array}$$

Lemma 5.44 (overlapping accesses with rglobal′ or wglobal′)
Let $acc(i,k)$ be a $rglobal'$ or $wglobal'$ access ending at cycle $t = e(i,k)$. Then, the accessed cache line is exclusive to cache i, such that there is no other access overlapping with $acc(i,k)$.

$$\begin{array}{l} SINV(t) \wedge (i,k) \neq (r,s) \wedge acc(i,k).a = acc(r,s).a \\ \wedge (rglobal'(aca^{d(i,k)}, acc(i,k), i) \vee wglobal'(aca^{d(i,k)}, acc(i,k), i)) \\ \implies [s(i,k) : e(i,k)] \cap [s(r,s) : e(r,s)] = \emptyset \end{array}$$

PROOF. by contradiction.

Let us assume the intervals overlap in cycle $q \in [s(i,k) : e(i,k)] \cap [s(r,s) : e(r,s)]$. The accesses have to concern different caches $i \neq r$ as only one access can be applied to a cache at the same time, see Lemma 5.31 (local order).

Furthermore $s(i,k)$ is defined to be the first cycle in the $warm'$ phase. With application of Lemmas 5.3, 5.4, 5.5 (grant at warm′, grant at warm, grant unique) one can conclude, that the automaton for cache r cannot be in a warm phase.

$$W'(i)^q \wedge \neg(W(r)^q \vee W'(r)^q)$$

Therefore, $acc(r,s)$ cannot be a $global$ or $global'$ access.

As a result, $acc(r,s)$ has to be a local access. The definitions of $rglobal'$ and $wglobal'$, requires that the state of the cache line with address $a = acc(i,k).a$ has to be exclusive to cache i.

$$aca(i).s(a)^{d(i,k)} \in \{M, E\}$$

The decision cycle for a $global'$ access is $d(i,k) = s(i,k) - 1$. Therefore

$$aca(i).s(a)^{s(i,k)-1} \in \{M, E\}$$

This state is not changing during access $acc(i,k)$ as implied by the Lemmas 5.38 and 5.36 (last cycle of wait′, stable master). In particular during cycle q, only cache i has valid data for address a.

$$aca(i).s(a)^q = aca(i).s(a)^{d(i,k)}$$

But this contradicts to the existence of another cache $r \neq i$ holding this line, as only one cache can have an exclusive cache line according to invariant 3.5 (single exclusive state). Therefore, $acc(r,s)$ cannot be a local access either. ■

Lemma 5.45 (overlapping accesses with broadcastms)
Only local reads can overlap with a mode broadcast, because no local write is possible for a shared line and any delayed read or global access needs to be granted. In particular, the overlap is in cycle $s(i,k)$.

$$\begin{aligned}
&SINV(t) \wedge (i,k) \neq (r,s)\\
&\wedge\, broadcastms(aca^{d(i,k)}, acc(i,k), i) \wedge acc(i,k).a = acc(r,s).a\\
&\quad : [s(i,k) : e(i,k)] \cap [s(r,s) : e(r,s)] = q\\
&\qquad \implies (q = \emptyset \,\vee\, q = \{s(i,k)\} \wedge rlocal(aca^{d(r,s)}, acc(r,s), i))
\end{aligned}$$

PROOF. by contradiction. Let us assume

$$q \neq \emptyset \wedge (\neg rlocal(aca^{d(r,s)}, acc(r,s), i) \vee q \neq \{s(i,k)\})$$

With the same argumentation as in the previous lemma, only local accesses are possible for $acc(r,s)$.

$$W'(i)^q \wedge \neg(W(r)^q \vee W'(r)^q)$$

According to the definition for *broadcastms*, $acc(i,k)$ is a read access of a shared cache line.

$$aca(i).s(a)^{d(i,k)} \in \{S,O\}$$

As before, the state stays unchanged during access $acc(i,k)$ due to Lemmas 5.38 and 5.36 (last cycle of wait$'$, stable master).

$$aca(i).s(a)^q = aca(i).s(a)^{d(i,k)} \implies aca(r).s(a)^q \in \{S,O\}$$

By Invariant 3.9 (unique owner) no cache can have an exclusive state for the respective cache line.. As a consequence, $acc(r,s)$ cannot be a local write.

Now, local reads are one cycle accesses and from Lemma 5.11 (sync for H$'$) follows that all slaves are in *sidle*, *slr*1 or *supdate* while the master is in state *mlr*1 or *mupdate*.

$$sidle(r)^{e(i,k)} \vee slr1(r)^{e(i,k)} \vee supdate(r)^{e(i,k)}$$

By Lemma 5.11 (sync for H$'$) all slaves with a cache hit are in state *slr*1, while the master is in state *mlr*1. In case of a memory update $mupdate(i)$, the slaves with *bhit* will remain in state *supdate* until $e(i,k)$. But this would raise a *snoopconflict* for the master automaton of processor r, such that the read access cannot start in cycles q with $s(i,k) < q \leq e(i,k)$. This concludes the lemma with

$$q = \emptyset \vee rlocal(aca^{d(r,s)}, acc(r,s), i) \wedge q = \{s(i,k)\} \qquad \blacksquare$$

The last lemmas can be summarized as follows. The only possible overlaps between accesses to the same cache address a are:

1. a flush with local reads,

2. a *global* access with local reads or local writes. In this case a local access ends at most one cycle after the start of the *global* access.

3. a *global'* read access with broadcast of a write mode switch can overlap with local reads. In this case the local read has to occur meanwhile the master automaton is in state $mlr0$.

Additionally, a local read can overlap with other local reads or a delayed local read.

The set of accesses $E(a,t)$ with address a ending in cycle t is defined with

$$E(a,t) = \{(i,k) \mid e(i,k) = t \wedge acc(i,k).a = a\}$$

For the second and third case of overlapping accesses, only one access can end in the end cycle t of the granted master automaton. In the other cases this set can contain more than one access. The next Lemma formalizes this.

Lemma 5.46 (simultaneously ending accesses [8, Lemma 8.54])
For any address a and end cycle t one of the following cases applies. Either set $E(a,t)$ contains a single element, all accesses in $E(a,t)$ are local reads and at most one access is a delayed local read or one access in $E(a,t)$ is a flush and all other accesses are local reads.

$$\begin{aligned} SINV(t) \quad \Longrightarrow \quad & \#E(a,t) = 1 \\ \vee \quad & (\forall (i,k) \in E(a,t) : idle(i)^{e(i,k)} \wedge \overset{\leq 1}{\exists} (r,s) \in E(a,t) : \\ & \quad (wait(r)^{e(r,s)} \vee wait'(r)^{e(r,s)})) \\ \vee \quad & (\overset{1}{\exists} (i,k) \in E(a,t) : acc(i,k).f \wedge \forall (r,s) \in E(a,t) : \\ & \quad (r,s) \neq (i,k) \rightarrow idle(r)^{e(r,s)}) \end{aligned}$$

PROOF. by application of the Lemmas for end cycles 5.32 5.33 5.34 (global', global and local end cycle) and the previous overlapping lemmas. ■

Let predicate $P(i,k,a,t)$ be true if access (i,k) to address a ends at cycle t and $acc(i,k)$ is not a local read.

$$P(i,k,a,t) \equiv e(i,k) = t \wedge acc(i,k).a = a \wedge \neg rlocal(aca^{d(i,k)}, acc(i,k), i)$$

Then, there is an relation between predicate $P(i,k,a,t)$ and the set $E(a,t)$.

$$P(i,k,a,t) = 1 \iff (i,k) \in E(a,t) \wedge \neg rlocal(aca^{d(i,k)}, acc(i,k), i)$$

With this the following lemma - regarding the affected memory system slices - is concluded.

Lemma 5.47 (unchanged memory slices [8, Lemma 8.55])

With the assumption of $SINV(t)$ the following three statements hold.

1. The memory system slice for address a is not changed at cycle t if the predicate $P(i,k,a,t)$ is not fulfilled for any access (i,k).

$$(\forall(i,k):\neg P(i,k,a,t)) \implies \Pi(ms(h^{t+1}),a) = \Pi(ms(h^t),a)\ .$$

2. In particular an access $acc(i,k)$ - ending at cycle t - does not change the memory system slice for address a, if $P(i,k,a,t)$ does not hold. In other words neither the cache RAMs nor the main memory is modified due to access $acc(i,k)$.

$$t = e(i,k) \wedge \neg P(i,k,a,t) \implies \Pi(\delta_1(ms(h^t),acc(i,k),i),a) = \Pi(ms(h^t),a)\ .$$

3. At most one of the accesses, which end in cycle t can change slice a. Only for this access $acc(i,k)$ is $P(i,k,t,a)$ true.

$$P(i,k,t,a) \wedge P(r,s,t,a) \implies (i,k) = (r,s)$$

The definition for predicate $P(i,k,a,t)$ argues over the decision to go for a local read at cycle $d(i,k)$. This means, it is necessary to ensure that this decision does not change until cycle t. For this reason the following lemmas are introduced.

Lemma 5.48 (stable local decision [8, Lemma 8.56])
The conditions - leading to a local access - are still satisfied at the end cycle $t = e(i,k)$.

$$local(i,k,aca^{d(i,k)}) \implies local(i,k,aca^t)\ .$$

Lemma 5.49 (stable global decision [8, Lemma 8.57])
The conditions - leading to a global access - are still satisfied at the end cycle $t = e(i,k)$.

$$SINV(t) \wedge global(i,k,aca^{d(i,k)}) \implies global(i,k,aca^t)$$

Lemma 5.50 (stable global′ decision)
The conditions - leading to a $global'$ access $acc(i,k)$ - are still satisfied at the end cycle $t = e(i,k)$.

$$SINV(t) \wedge global'(i,k,aca^{d(i,k)}) \implies global'(i,k,aca^t)$$

PROOF. by contradiction. Let us assume that the condition of a $global'$ access is not satisfied any more at the end cycle t.

$$SINV(t) \wedge global'(i,k,aca^{d(i,k)}) \wedge \neg global'(i,k,aca^t)$$

The predicate $global'$ depends on the state of cache i for address $a = acc(i,k).a$ and the access $acc(i,k)$ itself. The access signals do not change while it is processed, hence only the cache state can change and in this way issue the $global'$ test to fail at cycle t. The cache RAMs do not change while the access is processed, by Lemma 5.36 (stable master).

$$aca(i)^{s(i,k)} = aca(i)^{e(i,k)}$$

Neither do they change in the cycle before the access starts, according to Lemma 5.38 (last cycle of wait').

$$aca(i)^{s(i,k)-1} = aca(i)^{s(i,k)} = aca(i)^{e(i,k)}$$

The decision cycle of a $global'$ access is $d(i,k) = s(i,k) - 1$.
Therefore $\neg global'(i,k,aca^{d(i,k)})$ is required to get $\neg global'(i,k,aca^{t})$. But this contradicts to the assumption. ■

With these lemmas the predicate $P(i,k,a,t)$ can be reformulated.

Lemma 5.51 (reformulation of $\mathbf{P(i,k,a,t)}$ [8, Lemma 8.58])
Let $SINV(t)$ hold. Then,

$$P(i,k,a,t) \equiv e(i,k) = t \wedge acc(i,k).a = a \wedge \neg rlocal(aca^{t}, acc(i,k), i)\ .$$

5.8 Relating the Hardware Computation with the Atomic Protocol

The presented hardware computation is simulating the sequential atomic protocol. In order to prove this, a simulation relation has to be established. In particular, the effect of a hardware access $acc(i,k)$ ending in cycle t is the same as the effect of the same access $acc(i,k)$ applied to port i of the memory system $ms(h^{t})$ of the atomic protocol.

Lemma 5.52 (1 step [8, Lemma 5.59])
Assuming that $SINV(t)$ is true, the following two statements apply.

1. The memory system slice for address a is changing if predicate $P(i,k,a,t)$ is true for an access $acc(i,k)$. In addition, the new value for the hardware access is the same as in the atomic protocol.

$$\Pi(ms(h^{t+1}),a) = \begin{cases} \Pi(\delta_1(ms(h^{t}),acc(i,k),i),a) & \exists(i,k) : P(i,k,a,t) \\ \Pi(ms(h^{t}),a) & \text{otherwise} \end{cases}$$

2. The output for a read or CAS access $acc(i,k)$ of the hardware computation ending in cycle $e(i,k) = t$ is the same as for the atomic protocol.

$$acc(i,k).r \vee acc(i,k).cas \implies pdout(i)^{t} = pdout1(ms(h^{t}),acc(i,k),i)$$

PROOF. The lemma is already proven for local and *global* accesses in [8]. As a result it is left to show the case for a $global'$ or a *global* access with memory update. Then, the hardware access $acc(i,k)$ is ending in one of the following three states according to the definition of $e(i,k)$ and the hardware construction.

$$emupdate(i)^t \vee mlr1(i)^t \vee mupdate(i)^t$$

For all these cases the predicate $P(i,k,a,t)$ is fulfilled. Hence for the case $\neg\exists(i,k) : P(i,k,a,t)$ there is nothing to show by Lemma 5.47 (unchanged memory slice). Moreover, exactly one access can satisfy $P(i,k,a,t)$ by the last part of Lemma 5.47. This is the $global'$ or memory update access $acc(i,k)$, which updates the memory system slice for address $a = acc(i,k).a$. Let us consider the possible master automaton states at cycle $t = e(i,k)$:

- $acc(i,k)$ ends in state *emupdate*:
  The master automaton was in state $wait'$ in the cycle before the access started $s(i,k)-1$. Then it follows by Lemmas 5.38 (last cycle of wait$'$) and 5.36 (stable master) that the cache did not change during cycles $q \in [s(i,k)-1 : t]$.

  $$aca(i)^q = aca(i)^t$$

  In addition, no other access to address a overlaps with $acc(i,k)$, i.e. only $acc(i,k)$ ends in cycle t by Lemmas 5.44 (overlapping accesses with rglobal$'$ or wglobal$'$) and 5.46 (simultaneously ending accesses). Therefore the main memory for address a is not changing by Lemma 5.47 (unchanged memory slice).

  $$mm^q(a) = mm^t(a)$$

  This together with the automata construction, the data paths of the hardware and the definitions of $rglobal'$ and $wglobal'$ implies the statement.

  $$\Pi(ms(h^{t+1})) \;=\; \Pi(\delta_1(ms(h^t), acc(i,k), i), a)$$

  Furthermore by automata and hardware construction, the outputs for read or CAS accesses are given with

  $$pdout(i)^t = aca(i).data(a)^t = pdout1(ms(h^t), acc(i,k), i)$$

- $acc(i,k)$ ends in state $mlr1$ or *mupdate*:
  The $global'$ or *global* test succeeds at cycle t, according to Lemmas 5.32, 5.33 (global$'$, global end cycle) and 5.50, 5.49 (stable global$'$, global decision).

  $$global'(aca^t, acc(i,k), i) \vee global(aca^t, acc(i,k), i)$$

  With this premise, cache i does not change during the access by Lemma 5.36 (stable master).

  $$\forall q \in [s(i,k) : t] : aca(i)^q = aca(i)^t$$

Furthermore, cache i is the same as in the cycle before the access starts due to Lemmas (last cycle of wait′) and 5.37 (last cycle of wait).

$$aca(i).s(a)^{s(i,k)-1} = aca(i).s(a)^t$$

Lemma 5.45 (overlapping accesses with msbroadcast) states that only local reads are possible and only during cycle $s(i,k)$. This implies for all cache RAMs X:

$$global'(aca^t, acc(i,k), i) \implies \forall j : aca(j).X(a)^{[s(i,k)-1:t]} = aca(j).X(a)^t$$

Similarly by Lemma 5.41 (overlapping accesses with global) we get

$$global(aca^t, acc(i,k), i) \implies \forall j : aca(j).X(a)^{[s(i,k)+2:t]} = aca(j).X(a)^t$$

Furthermore the main memory content for address a is unchanged as only local reads or local writes can overlap, which do not modify the main memory.

$$mm^{[s(i,k)-1:t]}(a) = mm^t(a)$$

The protocol data transfer lemmas, together with the stability of processor inputs entail for all slaves $j \neq i$ for a $global'$ access

$$\begin{aligned} mprotin(j)^{s(i,k)+1} &= mprotout(i)^{s(i,k)} \\ &= 0000CS1(aca(i).s(a)^t, ptype(i)^t) \end{aligned}$$

In case of a $global$ access we have

$$\begin{aligned} mprotin(j)^{s(i,k)+1} =& mprotout(i)^{s(i,k)} \\ =& C1(aca(i).s(a)^{s(i,k)-1}, ptype(i)^{s(i,k)-1})0 \\ =& C1(aca(i).s(a)^t, ptype(i)^t)0 \\ sprotout(j)^{s(i,k)+2} =& C2(aca(j).s^{s(i,k)+2}(a), mprotin(j)^{s(i,k)+1}).(ch, di) \\ =& C2(aca(j).s^t(a), mprotout(i)^t).(ch, di) \\ sprotin(i)^{s(i,k)+3} =& \bigvee_j sprotout(j)^{s(i,k)+2} \\ =& \bigvee_j C2(aca(j).s^t(a), mprotout(i)^t).(ch, di) \end{aligned}$$

Now one can conclude the statement from the transfer lemmas.

$$\Pi(ms(h^{t+1})) \ = \ \Pi(\delta_1(ms(h^t), acc(i,k), i), a)$$

In particular the data transfer lemmas implicate for the answer of reads or CAS accesses:

$$pdout(i)^t = pdout1(ms(h^t), acc(i,k), i)$$

■

Now, all required properties are introduced, which are necessary to conclude the simulation lemmas below. Moreover, the remaining definitions and lemmas are the same as in [8].

The accesses are ordered by their end time. For this reason the set $E(t)$ is defined to contain all accesses with end time t.

$$E(t) = \{(i,k) | e(i,k) = t\} = \bigcup_a E(a,t)$$

The number of accesses - ending at cycle t - is denoted by $\#E(t)$. Then, $NE(t)$ denotes the number of accesses, which have ended before cycle t.

$$\begin{aligned} NE(0) &= 0 \\ NE(t+1) &= NE(t) + \#E(t) \end{aligned}$$

This function is used to number the accesses $acc(i,k)$ according to their end cycles.

$$\begin{aligned} seq(E(0)) &= [0 : NE(1) - 1] \\ seq(E(t)) &= [NE(t) : NE(t+1) - 1] \end{aligned}$$

If there is a flush access $acc(i,k)$ - ending in the same cycle as one or more local reads - $acc(i,k)$ is ordered last. Then, the sequentialized access sequence acc results from these definitions with

$$acc'(seq(i,k)) = acc(i,k)$$

The function $is(n)$ determines the cache port index i for the n-th access of the sequence.

$$is(n) = \in \{i | seq(i,k) = n\}$$

Now, it is possible to relate the hardware computation with the atomic protocol.

Lemma 5.53 (relating hardware with the atomic protocol [8, Lemma 8.60])

1. The first t cycles of hardware computation and the first $NE(t)$ sequential atomic accesses lead to the same abstract memory system configuration ms.

$$ms(h^t) = \Delta_1^{NE(t)}(ms(h^0), acc'[0 : NE(t) - 1], is[0 : NE(t) - 1])$$

2. The state invariants $SINV$ are valid until cycle t.

$$SINV(t)$$

3. The memory abstraction after the first t cycles and the memory abstraction after $NE(t)$ sequential atomic memory accesses are equal.

$$m(h^t) = \Delta_M^{NE(t)}(m(h^0), acc'[0 : NE(t) - 1])$$

Finally, the Lemma for sequential consistency from [8] is applicable.

Lemma 5.54 (sequential consistency [8, Lemma 8.62])
The hardware memory is sequentially consistent. Let access $acc(i,k)$ ending at cycle $e(i,k)$ be a read or CAS access $acc(i,k).r \vee acc(i,k).cas$. Then

$$pdout(i_x)^t \;=\; \Delta_M^{n_x}(m(h^0), acc')(acc(i,k).a)$$

Where for $x \in [1 : \#E(t)]$, n_x is set to $n_x = NE(t) + x - 1$ and i_x is defined with $i_x = is(n_x)$ for $acc(i_x,k_x) = acc'(n_x)$.

6 Conclusion

At the beginning of this thesis the lack of complete designs *at the gate level* and their verification for multi-core architectures with caches in the literature has been pointed out. In particular the data coherence is an issue when dealing with caches. This theme is addressed by the multi-core MIPS model and its implementation from [8]. But whenever I/O devices are used, it is mandatory to ensure that no stale data is read by the device. For this purpose the CPU forces the cached data to be written into the main memory respective to the I/O ports. Therefore, writes in write through mode become useful to solve this issue. This mode avoids the performance issues, which would occur if a flush is used instead. The underlying MIPS design in [8] did not support this write policy.

Motivated by this, the goal of this thesis was to adapt the existing cache model to additionally support the write through policy. For this purpose, the gate level design of the cache model was modified. In particular the control automata were extended by additional states and several transitions were modified. These efforts resulted in a fully synthesizeable gate level design. The correctness, i.e. sequential data consistency for this approach was proven by simulation to an atomic MOESI model. For this purpose the atomic specification had to be revised as well in order to support the write through policy.

In the presented design the write policy for shared cache lines is consistent in all caches. This results in a very strong consistency model. However no additional mechanisms are necessary to implement a weaker but more performant consistency model, whereby the write mode does not need to be broadcast (see Chapter 7 for further detail).

7 Future Work

The presented design hopefully represents a fully working approach, supporting a very strong consistency model. Still some extensions and optimizations are possible.

The proof presented here is hopefully a solid base for a formal verification. Nevertheless, it is just a building plan and it remains to be translated to a machine readable proof and to be prototyped e.g. on a field-programmable gate array (FGPA).

A potential optimization might be to use a write buffer to improve the performance of write operations. Another possibility is to limit the memory updates - caused by a write mode switch to *wt* - for shared lines, such that they are only issued if an owner exists. Therefore, the protocol needs to be elaborated in such a way that the slave response is extended to signal the owner state.

From the cache model presented here, it is only a small step to introduce the non cache-able mode. One could even design a (more fine grained) mechanism by restricting the write modes to certain address regions. Similar to the standard partition into read only and read write memory, it is possible to group certain addresses as write back, write through or none cache-able respective to the usage.

The design presented here provides all necessary instruments to implement two alternative consistency models. In the current one, even a read access issues a write to the main memory for dirty lines. Alternatively one can gain better performance for reads if this memory update is omitted. This way, consistency of the cache and main memory content is no longer ensured by reads.

If a weak consistency model is sufficient, one can gain even better performance. The approach is to allow as much caches as possible to work in write back mode to minimize the number of main memory accesses. Then the caches are allowed to have different write modes for the same memory address. That way the protocol can be simplified. In particular it is no longer necessary to broadcast a change of the write policy.

But then, a cache line in write through mode is not necessarily clean any more, i.e. the following scenario is possible: an access of cache i in write through policy updates the main memory and afterwards another cache $j \neq i$ in write back mode writes to the same cache line. Cache j broadcasts the modified data, but not the write policy. Therefore cache i will update its cache content with the broadcast data, which is not clean, but the write policy is still write through.

Because of this this, the Invariants 3.10 (clean write through) and 3.11 (same write policy) are no longer satisfied. As a matter of fact, there is no way to reason about the clean status of a cache line with an owner, except to compare it with the main memory value. The write through policy only ensures that the current access is immediately forwarded to the main memory.

Bibliography

[1] In R. Alur and T. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*. 1996. ISBN 978-3-540-61474-6.

[2] C. Baumann, W. J. Paul, and S. Schmaltz. System Architecture as an Ordinary Engineering Discipline. Lecture Notes in Computer Science Version 0.76, Saarland University, Saarbrücken, 2014.

[3] T. B. Berg. Maintaining I/O Data Coherence in Embedded Multicore Systems. *IEEE Micro*, 29(3):10–19, 2009. ISSN 0272-1732.

[4] K. Coloma, A. Choudhary, W. keng Liao, L. Ward, and S. Tideman. Dache: Direct access cache system for parallel I/O. In *in Proceedings of the 2005 International Supercomputer Conference*, 2005.

[5] E. A. Emerson and V. Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 144–159, 2003. ISBN 3-540-00898-5.

[6] J. Handy. *The Cache Memory Book*. Academic Press Professional, Inc., San Diego, CA, USA, 1993. ISBN 0-12-322985-5.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123704901.

[8] M. Kovalev, S. M. Müller, and W. J. Paul. *A Pipelined Multi Core MIPS Machine - Hardware Implementation and Correctness Proof*. Saarland University, January 2014.

[9] J. Magee and J. Kramer. *Concurrency: State Models &Amp; Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999. ISBN 0-471-98710-7.

[10] S. M. Müller and W. J. Paul. *Computer Architecture, Complexity and Correctness*. 2000.

[11] F. Pong and M. Dubois. A Survey of Verification Techniques for Cache Coherence Protocols, 1996.

[12] S. Schmaltz. *Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C*. PhD thesis, Saarland University, Saarbrücken, 2013. URL `http://www-wjp.cs.uni-saarland.de/publikationen/Schmaltz13.pdf`.

[13] S. Schmaltz. Mips-86 - A Multi-Core MIPS ISA Specification. Technical report, Saarland University, Saarbrücken, 2013. URL `http://www-wjp.cs.uni-saarland.de/publikationen/SchmaltzMIPS.pdf`.

[14] A. J. Smith. Cache Memories. In *ACM Computing Surveys*, volume 14, Berkeley, California 94720.

[15] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *SIGARCH Computer Architecture News*, 14(2):414–423, May 1986. ISSN 0163-5964.

[16] D. L. Weaver. Opensparc Internals. *Sun Microsystems*, 2008.

Glossary

Acronyms

(A) automata construction

(HW) hardware construction

(IH) induction hypothesis

CAS compare-and-swap

FGPA field-programmable gate array

MIPS-86 model of a multi-core MIPS machine

OC open collector

Technical Terminology

atomic protocol a cache protocol, which assumes sequentialized atomic memory accesses, p. 2

bus shared communication wire, e.g. between caches and main memory, p. 1

byte address has a length of 32 bits and is composed of 29 bits for the line address and 3 bits offset, p. 7

cache coherence protocol provides the necessary mechanisms for interaction with the memory layers and ensures certain properties, p. 1

cache data **(*ca.d*)** the memory lines (byte values), which were transfered from the main memory into the cache, p. 5

cache line a single entry of the cache, containing one entry for each component cache state, tag and data, p. 5

cache state **(*ca.s*)** for each cache line one of the states is possible: modified (M), owned (O), exclusive (E), shared (S), invalid (I), p. 5

cache tag **(*ca.t*)** used to identify the corresponding memory address, p. 5

caches the memory levels close to the processor, p. 5

data intervention fetching a cache line from other caches in case of a cache miss, p. 13

eviction when a cache location is currently occupied for another memory address and has to be evicted before the cache entry can be used for a new memory address, p. 1, 8

gate level design the implementation of the cache model, p. 16

line address consists of τ bits for the tag and a cache line address of ℓ bits to address the RAMs constituting the cache, p. 7

memory consistency model defines rules for the order and visibility of memory updates, p. 8

propagation delay necessary time until the value of registers or gates is changed, p. 12

sequential consistency the answers of reads in the presented gate level design are the same as if the operations of all processors were executed in some sequential order. For each processor, this order is consistent with the local order, which is specified by the executed program., p. 3

write back the entry is forwarded to the main memory, when it is evicted from the cache, p. 1, 8

write through each write operation of a cache is forwarded to the main memory, p. 1, 8

Basic Notations

$:=$	signal condition, $x := c$ signal x is rased if condition c is fulfilled
$\equiv$	definition
$:\Leftrightarrow$	both hand sides are defined to be logically equivalent
$\Leftrightarrow$	logical equivalence, both sides have the same truth value
$\Rightarrow$	implication in the mathematical sense
$\rightarrow$	either used as function arrow or as implication
$a \circ b$	concatenation of (bit)-string a with string b

$s \in \{M,O,E,S,I\}$	for state s, this is the short notation for $s[4:0] \in \{M,O,E,S,I\}$, e.g. abbreviations like $s = M$ or $s \in \{O,E\}$
X	when X is used as boolean value, this denotes that signal X is active (has a meaningful value 0 or 1)
$\overline{X}$	denotes that the signal X is inactive (value 0)

Symbols and Definitions

aca	abstract cache	an abstraction from the cache implementation consisting of two components: the cache state $aca.s : \mathbb{B}^{29} \rightarrow S$ and the cache data $aca.data : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$,	p. 6
S	set of states	set of possible cache states $S = \{wm00001, wm00010, wm00100, wm01000, wm1000\}$,	p. 20
ms	memory systems	the memory abstraction *ms* consists of two components: the line addressable memory $ms.mm : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$ and a sequence of abstract caches $ms.aca : [0 : P-1] \rightarrow K_{aca}$,	p. 9
$\Pi(ms,a)$	memory system slice	contains the main memory line $ms.mm(a)$ and the cache contents $ms.aca(i).X(a)$ for $X \in \{data, s\}$,	p. 9
lwms	local write mode switch	the write policy of the current memory access is different from the previous one $lwms \equiv acc.m \neq wm$,	p. 22
wb	write back	the memory accesses are handled in write back policy $wt(c) \Leftrightarrow \neg wb(c)$,	p. 19
$wm(c)$	current write mode	the write mode of the processor configuration c: $wm(c) \equiv wb(c) = 1$. This value determines the write mode for memory accesses *acc.m*,	p. 19
wm	write mode of a cache line	the write mode with which cache line *aca.a* was last accessed, i.e. $wm \equiv aca.s(acc.a).wm = 1$,	p. 21
$clean(a)$	clean line	a cache line for address a is clean, if the memory and cache data are consistent $clean(a) \equiv aca(i).data(a) = mm(a)$,	p. 28
M	Master	set containing all states of the master control automaton,	p. 45
S	Slave	set of all slave states,	p. 45
L	Local	set of local states *idle* and *localw*,	p. 45

G	Global	set of *global* states containing the states *wait, flush, m0, m1, m2, mdata, w, mupdate*,	p. 45
U	Update	set of $global'$ states, which can be attended in case of a $global'$ transaction,	p. 45
W'	Warm$'$	subset of Update, i.e. $U \backslash \{wait\}'$,	p. 45
W	Warm	subset of Global, i.e. $G \backslash \{wait\}$,	p. 45
H	Hot	global states, which run in sync with the slave automata,	p. 45
H'	Hot$'$	hot phase of new protocol part, in particular the states during participation in the protocol $mlr0$, $mlr1$ and $mupdate$,	p. 45