Practical Training Report

The Multi Micro Processor

Memory System and Caches

by Arjan Mels

Author: Arjan Mels (415826)
Coach: Dr. ir. A.C. Verschueren

January 2000
Summary

In this report the memory system necessary for the Multi-MicroProcessor is analysed. A large part is also implemented using IDaSS.

The memory system falls apart into two main blocks: the code sub-system and the data sub-system. The code system supplies the instructions to the cores of the processor. Via the data sub-system the cores can load from and store data to the main memory.

To optimise the performance caches are added both to the data side and to the code side. These caches utilise synchronous RAM, and are able to sustain a data rate of one item per clock cycle while reading from them.

Because all the cores have their small private instructions caches, the code sub-system can be relatively simple. It only has to arbitrate between incoming requests. (And as stated a global cache was added to enhance the performance even further.)

The data sub-system is a lot more complicated. Because locality in space is lost due to the shared use of Load and Store Units by the cores, a single global cache is necessary which allows multiple simultaneous accesses. The choice was made to use a banked approach. This way exotic hardware like multi-ported memory is avoided.

The design as presented is a reasonable compromises between performance and hardware size. Although nothing definite can be stated about the performance without extensive simulation, the offered solutions are very likely to suffice.
# Contents

Summary ......................................................................................................................... 2  

Contents ......................................................................................................................... 3  

1. Introduction .................................................................................................................. 5  
   1.1. General .................................................................................................................. 5  
   1.2. Multi Micro Processor ......................................................................................... 5  
   1.3. Memory System .................................................................................................. 6  
   1.4. Requirements ...................................................................................................... 6  

2. Overview of the Memory System .................................................................................. 8  
   2.1. Arbitration and Cache ....................................................................................... 8  
   2.2. Partitioning ......................................................................................................... 8  
   2.3. Conclusion .......................................................................................................... 10  

3. Basic Load & Store Unit .............................................................................................. 11  
   3.1. Introduction ......................................................................................................... 11  
   3.2. Analysis ............................................................................................................... 11  
      3.2.1. Instructions .................................................................................................... 11  
      3.2.2. Structure ....................................................................................................... 12  
   3.3. Interfaces ........................................................................................................... 12  
   3.4. Implementation ................................................................................................... 14  
   3.5. Design verification .............................................................................................. 15  
   3.6. Conclusions & Recommendations ........................................................................ 15  

4. Code Subsystem ......................................................................................................... 17  
   4.1. Introduction ......................................................................................................... 17  
   4.2. Analysis ............................................................................................................... 17  
      4.2.1. Cache ............................................................................................................. 17  
      4.2.2. Arbitration ................................................................................................... 18  
      4.2.3. Structure ....................................................................................................... 19  
   4.3. Interfaces ........................................................................................................... 19  
   4.4. Implementation ................................................................................................... 20  
      4.4.1. Cache ............................................................................................................. 20  
      4.4.2. Arbitration ................................................................................................... 22  
   4.5. Design verification .............................................................................................. 23  
   4.6. Conclusions & Recommendations ........................................................................ 24
1. Introduction

1.1. General

With the continuing miniaturisation of the feature sizes of integrated circuits and the ever-increasing maximum size of them, it has become possible to put more than one processor core on a die. Instead of a system with multiple processors one might use a single multiple core processor system, which could be more powerful (no bus bottleneck) and probably cheaper.

A possible design of such a multiple core processor is explored with the Multi Micro Processor [Lit. 1]. The structure suggested combines multiple RISC cores with different specialised coprocessors that are shared between the cores. In addition, the basic functionality of an operating system (task switching and messaging) is implemented in hardware. This is much faster and also easier in use than a software implementation. These advantages make the design especially useful for embedded applications.

During this practical training, I occupied myself with the design and implementation of the memory system of the Multi Micro Processor (MnP). This memory system has to arbitrate between the different parts of the MnP that require access to the system memory and also has to optimise the access speed of the memory. The design of the system was done and large parts were implemented with IdaSS. However, due to the amount of time available, not everything could be implemented.

In the rest of this introduction, I will give you a general outline of the MnP, with the stress on the memory system. In section 2, an overview of the memory system is given. In section 3 the design of a basic load and store unit is discussed; this load and store unit can serve as a clear and clean example of a general processing unit. In the 4th section the design and implementation of the code side of the memory system is detailed, while in the 5th section the data side is elaborated. While in section 6 the arbitration between the code and data side is discussed. Ultimately the overall conclusions are drawn and recommendations are made in section 7.

1.2. Multi Micro Processor

The basic idea behind the Multi Micro Processor is that you can put more cores (or Local Processing Unit = LPU) on one die and that those cores could share some common resources. Amongst these resources are the General Processing Unit (GPU's). These specialised coprocessors are not used every cycle by every core, so the access to these GPU's can be efficiently time multiplexed. Examples of GPU's are the Load and Store Unit (LSU, this unit accesses the memory), the Floating-Point Unit and the Complex Integer Unit.

The LPU's forward certain commands to the function switch (see also Figure 1). This unit forwards them then to the correct GPU, where the instruction is then executed. The result of the operation is then sent to the Result Switch, which forwards it to the correct LPU. The LPU can continue op-

![Figure 1 Functional structure of the Multi Micro Processor](image-url)
peration without waiting for this result. So, it is free to continue executing local (and even global) instructions as soon as it has forwarded the global command to the function switch and as long as the subsequent instructions do not use the same registers.

Each core can be busy with entirely different pieces of code. If there are more tasks (pieces of code that need to be run simultaneously) at hand than there are cores, some kind of task scheduling will be needed. This is often handled by software, but in the MuP it is handled in hardware by the Task Control Unit (which also handles messaging between the tasks).

When a task is scheduled out, but one or more GPU’s are still processing an instruction for that task, the result is forwarded to the register cache instead of the core (which is running a new task and thus running in an entirely different context).

1.3. Memory System

We can now proceed with examining which parts of the processor have to interface with the main memory. First of all the cores have to be supplied with instructions. Furthermore the cores have to be able to load and store data in the memory, this is done via one or more specialised GPU’s called Load and Store Units (LSU’s). If there are many tasks at hand, you do not want to make the register cache so large as to hold all the contexts, so the register cache also has to interface with the memory. Subsequently there might be any number of other GPU’s that have to access the memory (e.g. vector processors, communication processors etc.), because this kind of direct memory access can take a lot of the burden of the cores away.

Besides the parts that regulate and try to speed up the accesses to the main memory, also some other memory-related components still have to be designed. One of them is the Load and Store Unit. This unit makes some global commands available to the cores, which can use them to read data from and write data to the main memory. Other functions might be block move operations etc., but the exploration of such things is left to section 3. The other memory related part is the Register Cache. However, I have not applied myself to this part during this practical training, because there are some fundamental choices to be made in this block that relate to the switching of tasks and the storing of registers to memory and I already had my work carved out for me without these additional burdens.

Before the overall structure of the memory subsystem can be addressed (which will be done in section 2), one has to look at the demands made on such a system and what goals need to be (and seem reasonable to be) reached.

1.4. Requirements

There are two functional demands that the memory subsystem has to live up to:

- It has to be fast: All components have to be served as soon as possible, for the highest processor performance attainable.

- It has to be fair: All components have to be served at some time, otherwise deadlock or starvation situations could occur which is of course not acceptable. This is not to say that all components have to get an even share of memory access: some parts might be more critical to the performance than others!

In addition, three other considerations have to be taken into account:

- The design has to be easily scalable: The Multi Micro Processor is meant to be a very flexible design: It has to be (and is) possible to alter the number of cores and GPU’s in the design. Therefore, the memory subsystem must be easily adaptable to other configurations.
• The critical path of the memory subsystem must be no longer than the critical path of other parts of the processor and as short as possible: the slowest part of the processor determines the overall clock speed.

• The size of the circuit on the die should not be too large. Although this constraint is a bit fuzzy, it calls for realistic choices in the design process. You can set your goals very high, and with unlimited resources, almost everything is obtainable, but this is of course not the case.
2. Overview of the Memory System

2.1. Arbitration and Cache

Although a rough outline of the memory system is already given in [Lit. 1] (See also Appendix A), this section gives the rationale behind such a structure and some changes in the structure as given by [Lit. 1] are suggested.

As can be seen in Figure 2 many components want to access the memory. Thus, the access to the memory has to be arbitrated. However, most of the time more components will want to access the memory simultaneously (Figure 2). The processor cores for example need an instruction every clock cycle.

If all the accesses to the memory would be to addresses that are close to each other, the problem could be easily solved by a (very) wide data bus. The cores all work independently though. Therefore, this solution does not solve the problem.

Another rather blunt approach would be to use multi-ported memory. This would however require some separate data and address buses, not to mention the cost of multi-ported memory. The multi-ported memory can be circumvented by using split memories. One division springs to mind immediately. This is the division between the code side and the data side. Processors that implement separate code and data address spaces do exist (the so-called Harvard architecture). I do not think that is a good solution in this case though, because the disadvantage of multiple buses is not taken care of and because there still are many components wanting to gain access to a limited number of memories, so the basic problem is not solved. Moreover, it would involve some special structures to be able to store values to the code memory (to load programs). Every other division that you can think of is just as or even more problematic.

In multiprocessor systems, one also has to do with multiple systems accessing the same memory. There it is solved by giving each processor its own cache. This way the processor does not have to access the main memory for every load or store action. Caches were added to the processors already anyway, because high-speed memory is usually too expensive for the (large) main memory. The necessity for arbitration still exists, because all these caches still have to access the main memory. The problem has now become manageable however, since there are many fewer accesses to the main memory.

2.2. Partitioning

As stated, the normal solution in multiprocessor systems is to give every processor its own cache, thus relieving the burden on the bus. This approach can in part be followed for the Multi Micro Processor. The cores can each have a separate code cache. However, as the cores share common General Processing Units that might access the memory (amongst which the Load and Store Units), it is not a good idea to just add a separate data cache per GPU.

A cache works well if there is a certain locality of the data used in space and time. (I.e. the data is grouped tightly in memory and it is accessed in a relatively small time span.) This condition is not met in this set-up: multiple unrelated tasks will access the
same cache memory; thus breaking the space and time locality. Even more important is the factor that the separate caches have to keep coherent views of the data: if one task gets access to a different LSU each time, the data for that task gets scattered amongst the caches. The time necessary for keeping the caches coherent gets forbidding.

Therefore, the distinction between the data side and the code side (as it is already outlined in [Lit. 1]) is a good division, but on the data side something different than a cache per GPU has to be employed. Another reason why it is a good division is that the cores only read from the cache on the code side, but read from as well as write to the cache on the data side.

In [Lit. 1] no mention is made of any caching on the data side. At first glance, this seems understandable: as was stated in the previous paragraph, it is quite useless to add a cache per GPU. If you look at it from a performance point-of-view it also looks reasonable: not every processor will forward a Load or Store request every clock cycle.

However, when you look more closely, it becomes clear that some kind of caching on the data side is also inevitable. When the number of components (cores or GPU's) increases, the chance of simultaneous accesses will not be negligible. In tight computational or data shifting loops one memory access per two or three clocks per core is quite easily reached. Moreover, the main memory can be very slow in comparison to the processor: recent affordable dynamic RAM has an access time of 60ns; this amounts to a cycle time of at least 120ns and thus a maximum rate of about 8.3MHz. If you compare this to a maximum clock rate of 43MHz (mentioned in [Lit. 3]), then it becomes clear that a main memory access can take at least as long as five clock cycles.

Because a cache per GPU is not a good solution and a further division of the data memory not desirable, a single cache, which can handle multiple accesses, has to be considered. This might be implemented using true multi-ported memory or a kind of emulation of this type of memory (e.g. by using banked memory).

We have now reached a concept with separate code and data subsystems (see also Figure 3). The two subsystems still have to access the same external memory, so some kind of arbitration between these two is still necessary.

In this structure, the coherency between code and data subsystems is not mentioned. If this coherency would not be necessary, this would make matters a lot simpler. Luckily, it is also reasonable to do this: the code that is executed is not updated often and it is still possible to reconcile the two types of caches, by programmatically flushing them (thereby writing the updated content of the data cache back to memory). Therefore, if a mechanism to flush the caches is available, the maintenance of coherency between the data and code caches is not strictly necessary. (In practice, a number of processors do not do this, e.g. the Alpha 21164PC [Lit. 2] p. 2-13.)
2.3. Conclusion

A separate treatment of code and data is a logical structure to speed up the memory accesses. However, totally separate data and address buses cost too much in the way of pins and do not solve the main problem.

To be able to serve more requests to the memory simultaneously, cache structures are necessary. On the code side, separate caches per core are a good solution. Yet, this is not adequate for the data side. It is not necessary to keep the data and code sides coherent in hardware.

Before the code and data subsystems can be discussed (in sections 4 and 5 respectively), it is convenient to first look at a Load and Store Unit. This will give insight in the interface to the memory and additional requirements for the data subsystem. The LSU will therefore be the topic of the next section.
3. Basic Load & Store Unit

3.1. Introduction

The Load & Store Unit (LSU) accepts commands from the processor cores (via the Function Switch) and translates these commands to requests of the memory. The system can contain more than one LSU to increase the bandwidth from the cores to the memory.

The essential functionality of the LSU consists of a Read (also called Load) and a Write (also called Store) command. The Read command has to fetch the data from memory on the specified location and has to return this data to the core (via the Result Switch). The Write command has to store data from the core to the specified location in the memory.

The basic functionality of the LSU could be extended by several extra commands. Locked Read/Write cycles are very useful for inter-process and inter-processor communication. The addition of block operations (block fill, copy and compare) could take a lot of processing burden away from the cores for some applications.

In this section, only a basic LSU will be discussed: only a read, a write and a locked read/write instruction are implemented and the unit is not optimised for speed. The design is an example of a simple General Processing Unit, with a relatively clean implementation. It was used as a finger exercise for IDaSS and to explore the interface employed between the General Processing Units, the Function Switch and the Result Switch. It also was used to explore the interface that has to be used between the LSU's and the data-memory subsystem.

The LSU was not further developed at this stage because it became apparent during the design that a good (and thus fast design) is closely related to (and might even be interwoven with) the data-memory subsystem. Therefore, I thought it better to investigate the data-memory subsystem, before finalising the LSU. This is done in section 5.

3.2. Analysis

3.2.1. Instructions

To start, there only will be three commands implemented: a read command, a write command and a locked exchange command. The Read command has only one source operand (the address) and a result (the data). The Write command has two operands (the address and the data to store), but no result! (This means that internally the dummy register has to be used as destination.) The Locked Exchange has two operands (the address and the new data to store) and a result (the old data from the location).

The necessity of a Read and a Write command is self-explanatory. The Locked Exchange command can be used to implement semaphores. This isn't strictly necessary for inter-process communication (as something of the kind is implemented in the Task Control Unit, see also [Lit. 4]), but it is useful (as the number of semaphores is now only limited by the amount of memory). Moreover, it is indispensable for inter-processor communication on a shared memory basis. It also makes the addition of an extra signal to the memory necessary (M_LCK); therefore, it should be taken into account when designing the data subsystem.

The choice was made for the Locked Exchange instruction instead of a Locked Add or any other useful locked operation, because the Locked Exchange does not require additional hardware, whereas the Locked Add for example uses an extra adder.
Block operations would be a good addition for the LSU’s: they can relieve the load of the cores. Prime examples are the block fill, which fills a block of specified length with the specified data, the block copy, which copies the contents of one memory range to another, and the block compare, which searches for a specified value in a block.

There is one problem with this and that is that there are only two source operands. For most block operations, you need three operands however: the source location, the destination location and a count or the destination, the count and a value. This problem cannot be solved by simply adding an instruction for —say— setting the count, because there could be more than one LSU in the system. In this case, it would only be chance if the next instruction, which would execute the block move, would reach the same LSU. A solution could be to allow only one LSU with block operations. I have chosen though not to implement any block instructions, to keep the design as general as possible.

3.2.2. Structure

As soon as the LSU receives a command, the command and the matching data should be stored, because the Function Switch only keeps the information valid one clock cycle. Furthermore, the LSU should keep the amount of processing done immediately upon the receiving of a command restricted to a bare minimum: the Function Switch is a large combinatorial circuit with a corresponding delay. If the LSU were to act upon the received command immediately, the combinatorial delay would get even larger and thus the maximum clock rate of the processor would decrease.

The LSU should now continue with the decoding of the command. It then can act upon the specific command. To execute the commands several clock cycles will be necessary. E.g. for a Read command the LSU first has to issue a request to the memory and then when the result has become available it has to forward this to the Result Switch. This means sequential logic and suggests a state machine. This state machine controls a block of combinatorial circuitry, which routes the data to the correct place and generates the control signals.

During the processing of a command, the memory will be accessed. If the memory returns a result, it will again be necessary to buffer this data, because the memory will only keep it valid for one cycle and the LSU possibly needs to keep it valid for several cycles for the Result Switch.

This analysis suggests a structure with four main parts: an input buffer (for the command and data from the Function Switch), a switching unit (a combinatorial unit which generates the control signals and controls the data flow), a control unit (which is a state machine that controls the overall operation) and a memory buffer (for storing the data received from the memory).

3.3. Interfaces

The LSU receives commands from the processor cores (via the Function Switch). As described in [Lit. 3] this is done with a 15-bit wide bus called the Global Instruction (F_GI) bus from the Function Switch to the LSU and a 1-bit idle signal (F_IDLE) in the opposite direction. The 15-bit GI consist of 4 parts: a 4-bit task identification which is needed by the Result Switch to store the result at the correct location, a 5-bit instruction code, a 5-bit return address which again is needed by the Result Switch and a 1-bit acknowledge signal which indicates that the instruction and data are valid. Together with the command, the LSU receives the two data operands via the 64-bit wide Global Source (F_GS) bus.

The idle signal (F_IDLE) is asserted (high) when the LSU is busy processing the previous command. When this condition is true, no commands will be forwarded to the
LSU. When the Function Switch has a new command for the LSU, it puts this command on the F_GI bus. (The acknowledge part is of course high.)

To store the result of a command back to the processor cores (or the register cache) the LSU's have a 32-bit bus called Global Result (R_GR) to the Result Switch which transports the data, a 9-bit Global Instruction (R_GI) bus which contains the 4 bit task identification and the 5-bit return address and a 1-bit Global Instruction Complete (R_GIC) bus. In addition, the interface contains a 1-bit signal Result Acknowledge (R_RAC) back from the Result switch to the LSU.

When the LSU has processed a command, it can store its result back to the processor cores via the Result Switch. To do this it puts the data on the R_GR bus and the task identification and return address that it received from the Function Switch on the R_GI bus. It also asserts the R_GIC signal to signify that the other two buses are valid. The LSU has to keep the data on the buses valid until the Result Switch asserts the R_RAC line to signify that it has stored the results. Care should be taken, because the R_RAC signal can already be asserted during the clock cycle that R_GIC was asserted.

The interface to the memory subsystem is not defined in [Lit. 1] nor in [Lit. 3] or [Lit. 4]. For memory accesses, you need at least a data, an address and a control bus. The data bus could be a bi-directional bus, but for on chip connections it is better to use separate Data In (M_DI) and Data Out (M_DO) buses. The data buses are both 32-bits wide; the same holds for the address bus (M_ADR). Four 1-bit control signals are used. The M_REQ signal signifies that there is a request for the memory and that the data on the other buses is valid. The M_RW signals if it is a read or a write request. The M_LCK signal indicates that the request should be treated as a locked request. The M_ACK signal signifies to the LSU that the request has been processed. Some of these signals could be combined in larger buses, but this is not done at the moment for reasons of clarity.

If the LSU wants to read data from the memory subsystem, it puts the address on M_ADR, keeps R_RW low and asserts M_REQ (and if necessary also M_LCK). These values should be kept valid until the data is available (on M_DI) and M_ACK is asserted by the memory.

If the LSU wants to write to the memory, it puts the data on M_DO and the address on M_ADR and asserts both M_REQ and M_RW (and M_LCK if necessary). The memory responds with the assertion of M_ACK as soon as it has stored the data and at this moment, the LSU can release its buses.
3.4. Implementation

In Figure 4 the highest level of the implementation can be seen (see also Appendix B). The four blocks mentioned in the section 3.2.2 (input buffer, switching unit, control unit and memory buffer), are easily distinguishable.

The GPU_IN_BUFFER stores the signals received on the F_GI and F_GS busses when the 1-bit valid signal (that is contained in the F_GI bus) is high. This bit is also (unbuffered) available at the I_ACK output. The signals are already available in the cycle in which GPU_IN_BUFFER receives them. This is done in order to enable the state controller (CONTROL) to start decoding the instruction and it is accomplished by multiplexing the inputs with the stored values (for details see Appendices B.2 and B.5).

The GPU_IN_BUFFER also splits the F_GI and F_GS busses in their separate components. These components are I_S1 and I_S2 (the two source operands) for the F_GS bus. It is I_IC for the F_GI bus. I_IC is the 4-bit instruction code (the 5th bit, that signifies if the status word should be written back [Lit. 1], is dropped). I_GI is a combination of the task identification and the return address, which is needed by the Result Switch.

The MEM_IN_BUFFER stores the results, which come from the memory. The value on M_DI is stored when M_ACK is high.

The INSTR block contains combinatorial logic with four different functions. The IDLE function makes all the control signals low and thus is the default state in which nothing happens. There are two functions related to the reading of data (READ1 and READ2). When the block is executing READ1 the data on I_S1 is copied onto M_ADR and the control signals for the memory are set for reading (M_RW=0, M_REQ=1). During the execution of READ2 the data read from memory (M_DI) is forwarded to the Result Switch (R_GR). In addition, the R_GIC signal is asserted. Finally, the WRITE function forwards the data from I_S1 to M_ADR, but it also forwards the data from I_S2 to M_DO. It sets the control signals correct for writing (M_RW=1, M_REQ=1).
A separate constant generator (LOCK) was added for the generation of the M_LCK signal. This precludes the necessity of the replication of the READ1, READ2 and WRITE states in the INSTR block with an asserted M_LCK.

The CONTROL block is a state machine. It controls the INSTR and LOCK blocks. In Figure 5 the state diagram can be seen. At the moment only three instructions are implemented. The WRITE instruction is the simplest with only one state. The READ instruction has two states: one to fetch the data from memory and one to store it to the Result Switch. The EXCHANGE instruction has states similar to those of both other instructions. The choice was made to write the data to memory first and only then forward the result to the Result Switch, because in this way the M_LCK signal is asserted shortest. In this way, the memory subsystem is not tied up for an unnecessarily long period. If commands with other instruction codes (i.e. with code 0 or 4 to 15 inclusive) are issued to the LSU these are ignored and thus become functional NOP's.

3.5. Design verification

It should be verified, whether the LSU adheres to the interface specification as stated in section 3.3 and performs the tasks as laid out in 3.1.

The Load & Store Unit was tested by attaching a constant generator to all inputs. Viewers were attached to all outputs and internal signals on the highest level as well as to the CONTROL and INSTR blocks. Tests were performed for the all three commands and the command with code 0 (as NOP) under varying conditions: the behaviour under the assertion of both acknowledge signals (M_ACK and R_RAC) in the same cycle as a request was made and also acknowledgement in the next cycle and in later cycles was verified. The external behaviour was monitored, but also the state transitions in the CONTROL block and the function of the INSTR block. All tests gave the expected results in the final implementation and the external behaviour is as specified.

3.6. Conclusions & Recommendations

The unit as designed is fully functional and seems to work correct. The implementation is -I think- clean and clear. It can serve as a good example for the design of other General Processing Units. It can be developed further however.

The addition of extra instructions (like block operations) is quite easy in the chosen implementation and can be very worthwhile for some applications as already stated earlier. There is one problem with this and that is that there are only two source operands. For a block copy instruction, you need three operands however: the source location,
the destination location and a count. Simply adding an instruction for setting —say— the count cannot solve this problem, because there could be more than one LSU in the system. In this case, it would only be chance if the next instruction, which would execute the block move, would reach the same LSU.

In this stage of the design of the complete memory system, the LSU has made it clear what kind of interface is needed to the memory. Before continuing with using this complete interface for the data memory subsystem, first the design of the code memory subsystem is discussed in the next section. This is done, because the code memory subsystem is less complex than the data memory subsystem.
4. Code Subsystem

4.1. Introduction

The code subsystem has to arbitrate between the cores which all want to access the memory to get instructions. In addition, it has to optimise the performance by caching the data that is used most often.

An important requirement for the arbitration scheme is that it is fair (as stated in section 1.4). Fair does not necessarily mean that all cores should get equal shares of access to the memory; this could for instance depend upon the priority of the tasks involved (as is suggested in [Lit. 1]). It does however mean, that all cores have to be served eventually, otherwise starvation occurs.

As we have already seen in section 2 the performance can be enhanced by giving every core its own cache (the local cache). This reduces the number of accesses to main memory and makes it possible for the cores to fetch instructions simultaneously. A small cache (4KB = 1K words) per processor is already present in the design of the cores (see [Lit. 1]).

Keeping the data in these caches coherent is not really a problem. As already mentioned earlier, it is sufficient to have the possibility to flush the caches. In the code subsystem, this simply means that the data in the cache has to be invalidated. This is because no writes to the memory are performed. The local caches already have this facility (see [Lit. 1]), so everything is fine on this side. There is even an instruction implemented in the cores that starts the flushing of the caches.

The technologies that are commonly used make it difficult to synthesise asynchronous RAM's. Therefore, synchronous RAM has to be used for the caches. This is not yet done for the cache in the cores. This change would however require a complete redesign of the pipeline of the cores. It was not feasible to do that during this practical training and therefore the existing interface of the cores is used. In section 4.3, recommendations are made to improve the interface; the few small changes in the design of the code subsystem that are required, are indicated in the section on the implementation (section 4.4).

4.2. Analysis

4.2.1. Cache

It can be expected that the cores will all execute certain shared code often (e.g. operating system code). This suggests one additional cache which the cores share and which will contain this kind of code. Although the exact impact on the performance is difficult to estimate, it can be expected, that with such a shared cache the performance will be slightly better than in a system with no shared cache, but the same total amount of cache.

It would be an important improvement of this shared cache, if it would be possible to continue performing cache look-ups, while there is a main memory access in progress. A cache which posses this feature is often called non-blocking (i.e. it does not block access to the cache while a main memory access is performed).

What are left to determine about the global cache are its dimensions. Optimum dimensions can only be determined after extensive usage analyses and simulation of different configurations and even then it is highly application dependent. Therefore, a configuration is chosen based on certain assumptions that seem reasonable. The
configuration chosen and the assumptions on which the choices were made are outlined in the rest of this subsection.

The line size chosen for the local caches (4 words yielding 128 bits) seems reasonable on the ground that on average every fourth or fifth instruction is a branch instruction. Therefore, the same size is chosen for the shared cache.

It is not necessary for the cache to have the same size data bus to the cores as it has to the memory. For better performance, a wider memory data bus is very effective: a bus that is twice as wide simply doubles the bandwidth to the main memory. The cost (mainly pin count) however is also rather high. A data bus of 64-bits is feasible and seems reasonable for the Multi Micro Processor design.

It can be expected that the data in the shared cache is less related in space (and time) than the data in the local caches: the data will often consist of library routines that are not necessarily close to each other in memory for instance. Therefore, a high number of ways should yield a better performance. The cost in hardware however is prohibitive for a large number of ways. In existing mainstream processors, two and four way caches are used (almost) exclusively and thus one may assume that these are the reasonable alternatives. Thus, four ways seems a good compromise for this design.

The optimum size for this global code cache is still to be determined. For the time being a cache of 32KB (= 8K words) will be used. This is relatively large for an on-chip cache (although the most recent Intel Pentium or DEC/Compaq Alpha processors have a multiple of this). However, it seems justified, because there is more than one core (four in the current configuration) that requires an instruction (almost) every clock cycle. If it should become clear, that the cache chosen is too large, it is easily reduced in size.

4.2.2. Arbitration

As already stated, arbitration is necessary for requests which arrive at the same time or when a previous request is still being processed. Therefore, there are two places in the code subsystem where arbitration is required: first, when several cores try to access the shared cache at the same time; second, when a cache look-up fails, while another main memory access is already pending.

Optimum arbitration schemes would make use of a task’s priority. The involvement of this priority gives rise to a number of problems however (see also [Lit. 3]): the circuits tend to get quite large and fairness is sometimes in jeopardy. If the size and type of the local cache is chosen well, it may be expected that there are not too many requests forwarded to the code subsystem. It thus is tolerable not to consider the priority in an arbitration scheme.

To make the interval between request and answer as small as possible for the cores, a “first come, first served” scheme seems a good solution. However, this does not solve the problem entirely, because more cores could make a request simultaneously (i.e. in the same clock cycle). Therefore, additional arbitration is necessary. A round robin scheme (which cycles between the requests) fulfils the requirement of fairness (it divides the time evenly between the cores) and it is easily implemented. Other schemes (like random selection) do not hold (significant) advantages over these schemes and cost more in terms of hardware.

This gives a two-layer arbitration: first a round robin scheme is used to arbitrate between the incoming requests from the cores; second a first come, first served scheme is used for the regulation of the accesses to main memory.

It may seem that the “first come, first served” (FCFS) scheme could be substituted with an additional round robin scheme. Indeed the average access times to the memory would not increase. Still, the maximum access time as observed by a single core
would increase. At a glance, the hardware required for the FCFS scheme does not seem to be very much larger than for the round robin scheme. Therefore, the FCFS scheme was used.

4.2.3. Structure

The combination of the two arbitration schemes, the global cache and the non-blocking requirement, give a global structure as can be seen in Figure 7. The Arbitration block receives the requests from the cores (LPU’s) and makes a choice between them. The chosen request is then looked-up in the cache. If the data is not available in the cache, the Memory Control block receives the request. Other look-ups to the cache can take place while the Memory Control block fetches the data from memory. When the data is fetched, the Memory Control block stores this data to the cache and returns it to the correct core.

4.3. Interfaces

The code sub-system has two major interfaces: on one side, the cores request data from the shared cache; on the other side, these requests sometimes need to be forwarded to the main memory. Both interfaces perform the same function really: a request for data from a memory. Therefore, the two interfaces used look very similar.

The interface from the cores to the code subsystem is described in [Lit. 1] and consists of a 32-bits address bus (A0-3), a 32-bits data bus (D0-3), a 1-bit request signal (R0-3) to the code subsystem and a 1 bit ready signal (RDY0-3) back to the cores. When a core wants new data it puts the address of the desired data on the address bus together with a high signal on the request line. It keeps these signals valid until the code subsystem returns the requested data on the data bus accompanied by a high signal on the ready line. In this way, the fetching of data always costs at least two cycles: in the first cycle the request is made; after this there might be a number of cycles in which nothing happens, because the result is not yet available; in the last cycle the data is returned.

Two optimisations to this interface are possible. First, the data bus could be made wider. Optimal would be a bus as wide as the line-size; however, a 64-bits wide bus (this is the same width as the external data bus) is easiest to implement. Second, it is possible to overlap the returning of data with a new request. (Thus if the request line is still kept high when the ready line is made high by the code subsystem this signifies a new request). In this way one request per clock cycle per core is made possible.

The interface from the code subsystem to the memory (via an arbitrator) comprises an address bus of 31 bits (ADDR), a 64-bit data bus, a 1-bit request signal (READ) and a 1-bit ready/acknowledge signal (ACK). The usage of these lines is exactly the same as for the interface between core and code subsystem.

The second optimisation mentioned (overlap of requests) could also have been applied to this last interface. I have not done so however, because the memory is ex-
pected to be relatively slow (and thus the speed improvement would be marginal), it is slightly more difficult to implement and because it would need special measurements for the memory.

The code subsystem receives one additional signal. This is the Flush Cache (FC) signal. As the name implies, the cache will be flushed (for the code subsystem this simply means, that all the entries will be invalidated) upon receipt of this signal.

4.4. Implementation

4.4.1. Cache

The shared cache embedded in the code subsystem can still be seen as a separate entity. If constructed in a clear and general way it can also be used in other contexts. This cache was designed first; the rest of the code sub-system was build around it.

The cache designed by dr.ir. A.C. Verschueren [Lit. 5] served as the basis of the cache design. However, a complete overhaul was necessary, because in the original design asynchronous memory was used and the desired cache has four ways instead of two.

The cache is build around a central 'CONTROL' operator (as can be seen in Figure 7). Connected to this operator are all the in- and outputs, the memory banks for data and tags and some registers for the temporary storage of some information. Exact implementation details are to be found in Appendix C.

An important issue in the design of a multiple way cache is the administration of the access order of the ways. This information is needed, when new data has to be stored in the cache and no way is free. The efficient functioning of the cache depends on this replacement policy. Usually a Least Recently Used (LRU) policy is used.

In a two way cache system the administration of which way was accessed last is simple: one bit that is set when way 0 is accessed and reset when way 1 is accessed suffices. In a four-way system, however, this is not the case. For a true LRU algorithm a FIFO queue with four entries of two bits would be necessary [Lit. 6]. Additionally this would involve a read-modify-write cycle even for read operations. With the static RAM

![Figure 7 Overview of the cache used for the code subsystem](image-url)
that has to be used, this means a minimum read operation duration of two clock cycles.

This is of course not desirable. An alternative can be found in the so-called pseudo-LRU algorithm [Lit. 6]. This algorithm uses three bits. The first bit points to the group of two ways (0 & 1 or 2 & 3) which was accessed last; the other two bits point to the way last used in their respective group. Now only a write to the LRU bits is necessary, but the LRU bits have to be accessible separately: the bit that points to the specific way in the group that was not accessed most recently has got to keep its old value. Therefore, a separate block for the LRU bits is included; this block receives separate write signals for each of the LRU bits. Because there are 2K lines in the cache 6K LRU bits are necessary; this is too much to use a register file, therefore normal RAM has to be used.

Two basic operations are available: a read from the cache and a write to the cache. In addition, the possibility to flush the cache is implemented. The flowchart for these commands can be seen in Figure 9. For the implementation, it is very important to know the timing of the operations that the cache can perform. As the read operation will be used most often, this operation should take as few clock cycles as possible.

![Diagram of the state machine (CONTROL)](image)

To read data from the cache three actions have to take place: first, check if the data is available and if so in which way it is stored (so the tag memory has to be read); second, the data should be read and third, the LRU information should be updated. Each of these actions can be implemented in one cycle (even the LRU updating as shown earlier in this section). By reading all four ways at once, it is possible to return the correct data already after one cycle (or a Miss signal if the data was not found.). However, it is not possible to update the LRU information in this same cycle: the updating depends on the way in which the data was stored and this is only known after the reading of the tag memory.
So it looks like two cycles is the minimum for the read operation. Yet, it is possible to let the updating of the LRU information be overlapped by a new read request. The timing can schematically be seen in Figure 9.

<table>
<thead>
<tr>
<th>First read request</th>
<th>Read Data RAM</th>
<th>Read Tag RAM</th>
<th>Write LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Second read request</td>
<td>Read Data RAM</td>
<td>Read Tag RAM</td>
<td>Write LRU</td>
</tr>
<tr>
<td>First write request</td>
<td>Read Tag RAM</td>
<td>Write LRU</td>
<td></td>
</tr>
<tr>
<td>Second write request</td>
<td>Read Tag RAM</td>
<td>Write LRU</td>
<td></td>
</tr>
<tr>
<td>Read request</td>
<td>Read Data RAM</td>
<td>Read LRU</td>
<td>Write LRU</td>
</tr>
<tr>
<td>Write request</td>
<td>Read Tag RAM</td>
<td>Write LRU</td>
<td></td>
</tr>
</tbody>
</table>

During the gray cycles no new operations can be accepted and the busy signal is thus asserted.

- Figure 9 Timing of the read and write operations

When writing data to the cache 5 actions have to be taken: first, it has to be determined if there is already data in the cache with the same tag (in which case it has to be overwritten); second, if no data was stored for that address, the least recently used way has to be determined; third the data has to be written; fourth the Tag information has to be updated and fifth the LRU information has to be updated. Schematically this also can be seen in Figure 9.

It can be seen that two cycles are required and no overlap is possible. Thus, during the second cycle, no new operation can be accepted and the busy signal is asserted. A problem occurs when a write operation immediately follows a read command. Now, a read of the LRU information and a write to this information is required at the same time. This is not possible and hence an extra clock cycle has to be inserted.

From the description above of the operations, it is clear that some kind of state machine is necessary. Nevertheless, no state controller is included in the schematic in Figure 7: the state machine is implemented in the CONTROL block and the state is stored in the Status register. This solution is not the clearest way of implementing the cache. However, it was the direct result of converting the original implementation to synchronous memory. Moreover, the implementation of state controllers in IDaSS makes it difficult to drive many lines separately, because in a state controller no lines can be driven directly; the state controller can only give commands to other blocks. You would thus need either multiple constant generators or an operator block with different functions. The default implementation of the function is to use a multiplexer and demultiplexer. This is clearly not always the most efficient implementation.

### 4.4.2. Arbitration

With the cache implemented and its exact behaviour known, it is now possible to build the rest of the code subsystem around it. At one point or another the signals of the separate cores and the memory have to be combined to be able to regulate the accesses to the cache. Therefore, a central control unit is necessary. It is possible to connect all the cores and the memory directly to this control unit. However, this makes the control block fairly complicated and makes it more difficult to extend the system to a larger number of cores. Hence, the choice was made to keep the control unit mostly a simple multiplexer and implement most of the arbitration and control signal generation logic in separate units. Every core is connected to one of these units. The overall structure this results in can be seen in Figure 10.
The round robin arbitration scheme is implemented by passing a token along the RR (Round Robin) units. The unit that 'owns' the token can access the cache (if it is not busy of course). If the unit that owns the token does not want to make a request, it indicates this to the next unit. In this way, the timeslot is not lost.

When a unit has the token (or a previous block indicated that it does not use its turn) and wishes to make a request, the RR unit signals this to the control block. The control block then redirects the address bus from this core to the cache. When a hit results the data is put on all data outputs simultaneously, and the ROY line of the correct core is asserted. When the data is not in the cache (i.e. there is a cache miss), the RR blocks gets a sequence number from the control block. This sequence number has the same function as taking a number in a shop: you take a number and when this number turns up on the board, it is your turn. In this way, a first come, first served system is implemented.

When the previous memory request is finished (or when there was no previous request) the address is put on the memory address bus (Addr) together with a read signal. When the data becomes available (indicated by an asserted acknowledge signal), it is immediately forwarded to the correct core. This transfer takes precedence over the reading of data from the cache. During the same clock cycle as the delivery of the data, the new address is obtained from the RR unit next in line.

Both the main memory and the cache deliver 64 bits of data. However, the current interface to the cores is only 32 bits wide. Although the core implementation has to be changed in the future to make use of synchronous RAM, the choice was made to use the existing interface anyhow, to have at least a functioning system. The control block puts the correct 32 bits on the data bus to the cores (depending on the least significant bit of the address). When the interface to the cores is redesigned, this selection mechanism can be removed.

4.5. Design verification

Verified should be if the design adheres to the specifications. There is a very obvious, but important and non-trivial demand: the code subsystem must deliver the correct data. In addition, the design should be fair (allow all cores access to the memory) and should have an access time that is as short as possible. The fairness is obtained by
the arbitration schemes chosen, so verification consists of checking if the schemes are implemented correctly. The same holds for the access time.

First, the design of the cache was verified separately. Testing took place on a functional level, but during this testing the internal operation was also verified. The design as can be seen in Figure 11 was used. Besides patterns consisting of increasing addresses and "random" patterns, patterns that specifically tested the pseudo-LRU scheme were used. All test patterns mixed read and write requests. The writes were verified by reading the data back later, but also reads to addresses that were not previously written to were tested of course. Before any test pattern could be used the cache has to be flushed. This is achieved by setting the Flush constant generator to 1 during one cycle after which the necessary number of cycles (512 in this implementation) was waited for the completion of the flushing. This process was checked by inspecting the internal data, tag en LRU RAM's manually.

The total code subsystem was tested with the circuit as shown in Figure 12. Instead of real main-memory, a system is used in which the address is returned as data after a certain number of clock cycles. In this way a read from every possible address can be done, yielding a known result. The cores are simulated by two ROM's and a counter. The first ROM (RA?) contains the addresses that are to be read. The second ROM (RR?) only contains a sequence of one bit read signals. The counters function as control unit to step through the ROM's. With this system, every possible access order can be tested.

Tests were performed in which all cores asked for access at the same time, one after each other and in a "random" order. The same address was repeated multiple times at different intervals to check the functioning of the cache. In all cases the code subsystem functioned as prescribed and expected.

4.6. Conclusions & Recommendations

In this section, a design for the code subsystem is described which can serve the cores at a rate of one word per clock. This rate should be sufficient to serve a number of cores, because the cores each have there own first level cache. An overall cache
was implemented, because it is expected that this can even further improve the performance.

If a higher throughput rate is required, the two optimisations mentioned in section 3.3 (Interfaces) can be used. The most useful one is the widening of the data bus to 64 bits, because this doubles the total bandwidth. The faster bus protocol only doubles the speed with which a specific core can access the code subsystem.

The dimensions of the cache are expected to perform well under a broad range of conditions. Nevertheless, to optimise the performance, additional analysis and simulations are necessary.

Two different arbitration schemes were used: A round robin scheme for the initial selection of a request, because requests can arrive simultaneously. And a first come, first served scheme if access to the main memory is necessary. Although the round robin scheme probably uses less hardware, the cost of the first come, first served scheme is not excessive.

It is not expected that by using other arbitration schemes that do make use of a task’s priority, the performance can be enhanced significantly. Moreover, it can be seen from [Lit. 3] that it complicates things very much.

The only measurement for keeping the data coherent is the implementation of a flush sequence. Nothing more is needed for the code side, but it does mean that for the data subsystem a different approach is needed.
5. Data Subsystem

5.1. Introduction

This section describes the design of the data subsystem and how it could be implemented. However, due to time restrictions it proved not to be possible to implement it with IdaSS during this practical training.

As already mentioned in section 2 multiple units have to access the memory to fetch data and sometimes they might even try to access it simultaneously. Also it became evident that caching is necessary on the data side.

On the code side, the choice was made to use a small cache per core. On the data side, this is not a good solution, because it cannot be assured that the memory access of a core is handled by the same LSU every time. Therefore the caches connected to the separate LSU would constantly need reconciliation and this would generate so much traffic, that the cache would be a hindrance instead of help.

Therefore, a single cache, which can handle multiple accesses, has to be considered. As already mentioned, this might be implemented using true multi-ported memory. However multi-ported memory is not available in many technologies and (therefore) relatively expensive. Running the memory at double the clock speed can make virtually multi-ported memories, however this is again not possible in many technologies, relatively expensive and not possible to implement in IdaSS.

Hence, some sort of banked scheme is necessary: the separate banks can be accessed simultaneously. The way the memory is split into banks has to be considered carefully to obtain as even a spread of accesses over the banks as possible.

Because two units might want to access a bank simultaneously, an arbitration scheme is still necessary. An important requirement for the arbitration scheme is that it is fair.

Coherency of the data and code caches was not really an issue, because the code cache is read only. Keeping the data cache coherent with the main memory is not a problem as long as no system components external to the Multi-MicroProcessor modify the memory. It was decided that this is not the type of application for which the Multi-MicroProcessor is intended. Therefore, a cache flush feature suffices. For the data cache this means that data that was written to it by the cores (or other parts) has to be written back to main memory, before the entries are invalidated.

5.2. Analysis

5.2.1. Structure

The system as sketched in the previous sub-section, consist of a number of banks, which can be accessed by any of the LSU's and other units that need to access the memory. This can be achieved by using a large crossbar switch like the ones used in the function and result switch. Actually two crossbar switches are necessary: one from the LSU’s to the cache banks and one in the opposite direction. In Figure 13, this overall structure can be seen.

The arbitration block in this figure has to resolve conflicts between the units that want to access the same banks. A round robin scheme is recommended, because of its simplicity and fairness. This scheme has to be implemented separately for every bank.
Additional arbitration is required to determine which bank can access the main memory. This arbitration block is not shown in Figure 13, because this might be combined with the block that has to arbitrate between the accesses from the code subsystem and the data subsystem.

It is still left to determine how accesses should be spread over the banks. A good system to assure that accesses are spread as evenly balanced as possible, is to use the lower bits of an address for the bank selection. This works well, because it can be expected that memory addresses that lie close together are accessed shortly after each other.

Yet, it was decided to use a 64 bit wide interface to the main memory for performance reasons. This means that the least significant bit has to be used for distinguishing between the lower and upper 32 bits. Moreover, it also introduces some extra problems, because the LSU's only handle 32 bit wide data. Two solutions are obvious. The first is to let the LSU's handle it. However when a 32 bit write from the cores is issued, this necessitates a full read-modify-write cycle. The second is to let the banks handle it. It can be handled in a variety of ways inside the banks, but this will be looked into in the next sub-section.

The number of banks should be scalable, however for the moment four banks are suggested. This is enough for a reasonable probability of multiple accesses at once and is still small enough to be able to implement it. It is also enough to explore all problems involved.

The maximum clock speed of the data sub-system might be a problem. A request should be made known, arbitrated and presented to the memories in one clock cycle. This is about the same as needs to be done in the function switch; the arbitration in the function switch is more complex than in the data sub-system, but the data sub-system has to address memories, while the function switch only stores values in Registers. So nothing definite can be said until the timing results of an implementation are available. It is something to keep in mind however when trade-offs have to be made.

5.2.2. Crossbar Switches & Arbitration

The crossbar switches enable each of the requesting units to be connected to each of the cache banks. It has to be able to handle multiple connections at the same time. Actually two switches are necessary: one to transport addresses, data and control signals to the cache banks and one to return data to the requesting units. These two switches need to be separately configurable, because data is returned at least one cycle after the request.

The switches can use a design very similar to those used for the function and result switches (see [Lit. 3]). This means a design with a multiplexer per bank for the switch to transport requests and a multiplexer per requesting unit for the other switch.

The switch to transport requests to the cache banks needs to be controlled via an arbitration scheme because multiple requests can arrive within the same clock cycle.
This arbitration has to take place for each bank separately. The outcome of the arbitration controls the switch directly.

The switch to return the results can be controlled directly by the cache bank that wants to return data. Because the requesting units can only have one request pending, no arbitration is necessary. The cache banks can store which unit made the request and control the switch with this info. The resulting structure of the two switches can be seen in Figure 14.

The arbitration scheme necessary for the switching of the request to the banks has to be cheap in hardware, because it has to be implemented for each bank, it also has to be fair, allowing all requested units access in due time. For similar reasons as discussed in the section on the code system, a round robin scheme is an acceptable compromise. One exception to this scheme has to be allowed; in section 3 about the basic LSU, it was seen that a locked read/write operation might be a handy feature and a special control line (M_LCK) was included in the interface for this purpose. This feature could be implemented by altering the arbitration scheme. After a request is processed the unit should not go on to the next requesting unit if the M_LCK signal is asserted, but continue with the next request from the unit that has M_LCK asserted.

It is possible that two or more banks want to access the main memory in the same cycle. This last problem will be addressed in the next section, section 6, because the code subsystem also has to access the main memory. And this can be solved with one arbitration system.
5.2.3. Cache

The cache for the data subsystem is far more complicated than for the code subsystem, because it has to be possible to read data from the cache as well as write to it. When the cache is written to the main memory also has to be updated eventually. Two common strategies can be employed: write-through and write-back [Lit. 6]. Write-through is somewhat simpler to implement, but write-back has superior performance. Because performance is an important issue, write-back cache should be implemented. The write-back cache does not write data immediately back to the main memory (as write-through does), but only when a cache line has to be freed for other data (evicted).

In systems that implement write-back cache, usually a feature can be found to choose different caching strategies for different memory ranges. For example for memory mapped I/O, the caching can be disabled; write protected memory ranges can be indicated and write-through caching is an option for example for frame buffers of video-cards. All these features could (quite easily) be implemented in the data subsystem, because they are all simplifications of the write-back algorithm. Nevertheless it will be necessary to implement an additional GPU, to be able to define the ranges. This costs quite a lot of hardware due to the extra hardware necessary in the Function Switch. It might however be possible to implement this functionality in an existing GPU, for example the Task Switch (there can only be one of these). Another alternative is to provide a separate I/O General Processing Unit. All the extra features are then extraneous and the I/O Processor probably is a more effective solution.

The size of the banks is of course also scalable, but for the moment a total cache size equal to the global code cache was chosen: 32KB. This gives 8KB per bank. To keep the amount of logic down and because the memory is already divided into 4 banks, only two ways are used. Because the memory interface is 64 bits wide, it is obvious to use a line size of at least 2 words. Although it is possible to use more words per cache line, this complicates matters a lot; furthermore, it restricts the effectiveness of balancing over the different banks.

To enable the cache banks to also handle 32 bit requests it is necessary to use separate RAM's for the lower and upper 32 bits. In this way, the two words can be written to separately. It also means that separate dirty bits have to be maintained. In addition, special care has to be taken when writing back the data to main memory. The easiest solution for this is if the main memory has separately writable memory halves.

---

Figure 15 Flowchart of Write-Back cache
The same three operations as for the code-cache have to be implemented: a read operation, a write operation and a flush operation. The main difference is that the cache has to access the main memory independently. This might require wait states in the instructions. A high level flowchart of the three instructions for a write-back cache can be seen in Figure 15.

To obtain a throughput that is as high as possible each of the cache banks should be able to accept a read request every cycle and return the data the next cycle (if it is available from the cache memory). As was seen with the code sub-system this requires a very careful design.

For the code sub-system the choice was made not to use an IDaSS state controller, because it is a bit convoluted if you have to use a lot of control lines. However, because the data cache is a lot more complex it is advisable to use a state controller in the implementation. The overall structure of the cache can be seen in Figure 16. It might be handier not to use one solid memory switch, but smaller units. For example separate units for the address and data parts and a separate part to handle the data in the Tag RAM, might make the implementation a lot easier.

5.3. Interfaces

The data sub-system receives commands from the LSU’s and other units that want to access the memory.

The interface from the requesting units to the data sub-system was already specified in the section on the basic LSU (section 3.3). This interface is directly usable. For requesting units that can handle 64 bit wide data, the same interface can be used, just the data buses have to be extended to 64 bits.

The interface from each of the cache banks to the main memory (via an arbitrator) is the same as for the code sub-system. Because the data sub-system also has to write to the main memory, a write request signal has to be added. Considering the need for separately writable memory halves, two write signals should in fact be implemented; one for the lower 32 bits (WRITE0) and one for the upper 32 bits (WRITE1).

The data sub-system needs one additional interface for the flushing of the caches. The flush operation stores modified data back to the main memory. The data sub-system will receive the Flush Cache (FC) signal, which initiates the flushing. Because the flushing can take a variable amount of cycles a status report is necessary. Therefore, a signal that indicates that the cache is busy flushing itself is proposed. The LSU’s should not start fetching any new data before this signal is deactivated again. This can be implemented by blocking request from the code cache to the main memory.

5.4. Conclusions & Recommendations

The design for the data sub-system as presented in this section, is able to handle multiple requests simultaneously. To achieve this, without using for example multi-ported memory, a fairly complex system is necessary. The system as described is expected to be able to cope with quite large amounts of requests.

Due to a lack of time, the system was not implemented. With the analyses from this section the implementation should be pretty straightforward. However, it might turn out
to be more work than it seems at first glance. Especially the implementation of the cache is a tough nut to crack. Implementing a non-optimised (this means without overlapping requests or unnecessary wait cycles) version is quite easy; implementing an optimised version is a lot harder (especially because of the use of synchronous RAM).

If the system is implemented as proposed, it should be able to cope with the amount of data that can be expected in the current configuration. If the configuration changes it should be doable to change the data sub-system along with it, because the proposed realisation offers several opportunities to scale the design.
6. Coupling of Code and Data Caches

6.1. Introduction

There are multiple blocks in the Multi-MicroProcessor that have to access the main memory. At the moment there are the code subsystem and the 4 cache banks in the data sub-system. However in the future, this number might increase (due to the addition of an I/O GPU for example). These blocks might attempt simultaneous accesses and therefore access to the main memory has to be arbitrated.

Some blocks might need more priority than others. Most often, it is seen that loading and storing of data has the highest priority, because otherwise the pipelines will stall. Next comes the fetching of code and last should come the fetching of large blocks of code by for example a vector GPU. Simulations have to show however if this is the correct order for the Multi-MicroProcessor. It might be that the code fetching has to have higher priority, because otherwise starvation occurs. In any case, it is obvious that the data sub-system should be handled as one unit for any kind priority scheme, because otherwise it will get too large a share of the access time.

Again due to a lack of time, this block was not implemented, but implementation is a minor issue.

6.2. Analysis

A “first come, first served” arbitration scheme is the most obvious fair one. The problem with this scheme is that multiple requests might arrive in the same clock cycle and hence it is necessary to implement additional arbitration for those requests. This seems to be overdoing this a bit, because there should not be too many main memory request left with all the caching implemented.

Therefore, it seems to be a better idea to just implement a round robin scheme. This way only one arbitration scheme is necessary. On average the round robin scheme has the same waiting period as the “first come, first served” scheme anyway.

An added advantage of this scheme is that priorities are quite simple to implement. For units that need a higher priority just use more places in the round robin scheme, they will be granted access more often automatically in this way.

The round robin scheme can be implemented in a very similar way to that used for the code sub-system, with a token that is passed along.

6.3. Conclusions & Recommendations

It should be possible to implement the proposed scheme fairly quickly. It is expected that the round robin scheme performs well enough, especially if the priorities of the blocks are tweaked as necessary. To do this correctly simulations are again necessary.
The memory system for the Multi-MicroProcessor is analysed in this report. For the mayor part the outline as given in [Lit. 1] is followed. In [Lit. 1], however, no caching on the data side was mentioned. It became apparent that this is necessary for performance reasons.

The analysis and implementation of the memory system proved to be more complex and time consuming than was thought. Therefore the data subsystem was not implemented with IDaSS, only a detailed analysis is provided. Large parts of the data subsystem can be implemented using existing parts from the code sub-system (and the function switch as described in [Lit. 3]). However, an important part (the actual cache) cannot be based on an existing part.

The parts that are implemented have been tested, but only isolated. The parts work according to specification and it should be directly possible to couple the parts into the Multi-MicroProcessor framework.

Scalability is an important issue for the Multi-MicroProcessor. The approach used for the memory system is quite modular and extendible. As for many parts of the Multi-MicroProcessor, however, scaling is not as straightforward as desirable, due to many interdependencies between the number of blocks and bus widths, the limited usability of parameters in IDaSS, etc.

For a fair arbitration extensive use the round-robin scheme was made. It is easy to see that this scheme is fair, because it gives an equal chance to all the requesting parts. Although it might have some benefits to include a tasks priority in the arbitration schemes this proves to be very difficult and might lead to deadlocks if not implemented very careful. It did not seem worth the trouble.

For the implementation, only synchronous RAM is used. This introduces extra complexity, but makes the actual realisation in hardware easier. To achieve as high as possible a performance, a form of pipelining is used, which makes it possible to get 1 item of data from the cache per cycle.

The timing might become a bit critical in the data sub-system. The speed should be in the same order as that of the function switch, but it has to be kept in mind during implementation.

Overall it can be concluded that reasonable trade-offs between performance and usage of hardware have been made. For an evaluation of the real performance however extensive simulations are necessary. Even then the use of such simulation is limited, because the performance will be very dependant on the precise memory access patterns, which can be widely different from application to application. To be able to do these simulations a sort of high level simulator would need to be constructed. This is far easier than implement the complete system and change it until it performs well.

I think, however, that the proposed system should be able to handle almost all situations already reasonably well.
Literature


Appendix A Overview of the Multi Micro Processor
Appendix B Schematics & Listings of the Load & Store Unit

B.1. Schematic of the Load & Store Unit

B.2. Schematic of the GPU_IN_BUFFER block

B.3. Schematic of the MEM_IN_BUFFER block
B.4. Listing of the Load & Store Unit

'\LSU' is a BBB schematic. This BBB schematic has the following connectors:

A bus feedthrough connector with name 'F.GI':
This bus feedthrough is 15 bits wide.

A bus feedthrough connector with name 'F.GS':
This bus feedthrough is 64 bits wide.

A bus feedthrough connector with name 'F.IDL':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'M.ACK':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'M.ADR':
This bus feedthrough is 32 bits wide.

A bus feedthrough connector with name 'M.DO':
This bus feedthrough is 32 bits wide.

A bus feedthrough connector with name 'M.LCK':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'M.REQ':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'M.RW':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'R.GI':
This bus feedthrough is currently unconnected.

A bus feedthrough connector with name 'R.GIC':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'R.GR':
This bus feedthrough is 32 bits wide.

A bus feedthrough connector with name 'R.RAC':
This bus feedthrough is 1 bit wide.

Contents of this BBB schematic are unique in this system.

Blocks for BBB schematic '\LSU':

---

'\LSU\CONTROL' is a state machine controller.

This state machine controller has 6 states.
No stack is available for 'subroutine' calls.
This controller is enabled following system reset.

This state machine controller has no connectors.

Text for state number 1 (reset state) of '\LSU\CONTROL':

---

START:
INSTR idle;
[ GPU_IN_BUFFER\\I.ACK
|1|]
|GPU_IN_BUFFER\\I.IC
|01h - - - -
|02h
-> READ1
|03h
|EXCHG1
]
---

37
Text for state number 2 of '\LSU\CONTROL':
--------------------------------------------
READ1:
INSTR read1;
[ M_ACK\M_ACK
|%1
-> STORE
|%0
<<
]
--------------------------------------------

Text for state number 3 of '\LSU\CONTROL':
--------------------------------------------
STORE:
INSTR read2;
[ R_RAC\R_RAC
|%1
-> START
|%0
<<
]
--------------------------------------------

Text for state number 4 of '\LSU\CONTROL':
--------------------------------------------
WRITE1:
INSTR write1;
[ M_ACK\M_ACK
|%1
-> START
|%0
<<
]
--------------------------------------------

Text for state number 5 of '\LSU\CONTROL':
--------------------------------------------
EXCHG1:
INSTR read1;
LOCK setto: 1;
[ M_ACK\M_ACK
|%1
-> EXCHG2
|%0
<<
]
--------------------------------------------

Text for state number 6 of '\LSU\CONTROL':
--------------------------------------------
EXCHG2:
INSTR write1;
LOCK setto: 1;
[ M_ACK\M_ACK
|%1
-> STORE
|%0
<<
]
--------------------------------------------

End of state descriptions.

'\LSU\INSTR' is an operator.

This operator has 4 functions.
The default function is 'READ1'.

This operator has the following connectors:

An output connector with name 'I_IDL':
This output is 1 bit wide.
An input connector with name 'I_S1':
This input is 32 bits wide.

An input connector with name 'I_S2':
This input is 32 bits wide.

An input connector with name 'M_ACK':
This input is 1 bit wide.

An output connector with name 'M_ADR':
This output is 32 bits wide.

An input connector with name 'M_DI':
This input is 32 bits wide.

An output connector with name 'M_DO':
This output is 32 bits wide.

An output connector with name 'M_REQ':
This output is 1 bit wide.

An output connector with name 'M_RW':
This output is 1 bit wide.

An output connector with name 'R_GIC':
This output is 1 bit wide.

An output connector with name 'R_GR':
This output is 32 bits wide.

An input connector with name 'R_RAC':
This input is 1 bit wide.

Invisible control channel attached to this operator:

From state machine controller '\LSU\CONTROL' via path 'INSTR':
Commands issued are 'read2', 'writel', 'read1' and 'idle'.

Text for function 'IDLE' of '\LSU\INSTR':
----------------------v----------------------
I_IDLE:=1.
M_RW:=0.
M_REQ:=0.
R_GIC:=0.

Text for function 'READ1' of '\LSU\INSTR':
----------------------v----------------------
I_IDLE:=0.
R_GIC:=0.
M_ADR:=I_S1.
M_RW:=0.
M_REQ:=1.

Text for function 'READ2' of '\LSU\INSTR':
----------------------v----------------------
I_IDLE:=0.
R_GR:=M_DI.
M_RW:=0.
M_REQ:=0.
R_GIC:=1.

Text for function 'WRITE1' of '\LSU\INSTR':
----------------------v----------------------
I_IDLE:=0.
R_GIC:=0.
M_ADR:=I_S1.
M_DO:=I_S2.
M_RW:=1.
M_REQ:=1.

End of function descriptions.
'\LSU\LOCK' is a constant generator.
This constant generator is 1 bit wide.
It's value range is therefore 0..1.
The default value is 0.
This constant generator has the following connector:
An output connector without a name:
This output is 1 bit wide.
Invisible control channel attached to this constant generator:
From state machine controller '\LSU\CONTROL' via path 'LOCK':
The only issued command is 'setto:' (constant: 1).

B.5. Listing of the GPU\_IN\_BUFFER block

'\LSU\GPU\_IN\_BUFFER' is a BBB schematic. This BBB schematic has the following connectors:
A bus feedthrough connector with name 'F\_GI':
This bus feedthrough is 15 bits wide.
A bus feedthrough connector with name 'F\_GS':
This bus feedthrough is 64 bits wide.
A bus feedthrough connector with name 'I\_ACK':
This bus feedthrough is 1 bit wide.
Invisible test channel attached to this bus feedthrough:
From state machine controller '\LSU\CONTROL' via path 'GPU\_IN\_BUFFER\I\_ACK':
Tests on channel value only.
A bus feedthrough connector with name 'I\_GI':
This bus feedthrough is 9 bits wide.
A bus feedthrough connector with name 'I\_IC':
This bus feedthrough is 4 bits wide.
Invisible test channel attached to this bus feedthrough:
From state machine controller '\LSU\CONTROL' via path 'GPU\_IN\_BUFFER\I\_IC':
Tests on channel value only.
A bus feedthrough connector with name 'I\_S1':
This bus feedthrough is 32 bits wide.
A bus feedthrough connector with name 'I\_S2':
This bus feedthrough is 32 bits wide.
Contents of this BBB schematic are unique in this system.
Blocks for BBB schematic '\LSU\GPU\_IN\_BUFFER':

'\LSU\GPU\_IN\_BUFFER\MUX2\_1' is an operator.
This operator has 2 functions.
The default function is 'MUXA'.
This operator has the following connectors:
A control input connector without a name:
This control input is 1 bit wide.
Control specification:
%1 MUXA.
%0 MUXB.

40
An input connector with name 'A':
This input is 64 bits wide.

An input connector with name 'B':
This input is 64 bits wide.

An output connector with name 'O':
This output is 64 bits wide.

Invisible control channel attached to this operator:
A local channel from an unnamed control input:
Commands issued are 'muxb' and 'muxa'.

Text for function 'MUXA' of '\LSU\GPU_IN_BUFFER\MUX2_1':
----------------------v----------------------
O:=A.
----------------------^----------------------

Text for function 'MUXB' of '\LSU\GPU_IN_BUFFER\MUX2_1':
----------------------v----------------------
O:=B.
----------------------^----------------------

End of function descriptions.

'\LSU\GPU_IN_BUFFER\MUX2_1' is an operator.

This operator has 2 functions.
The default function is 'MUXA'.

This operator has the following connectors:
A control input connector without a name:
This control input is 1 bit wide.

Control specification:
----------------------v----------------------
%1 MUXA.
%0 MUXB.
----------------------^----------------------

An input connector with name 'A':
This input is 14 bits wide.

An input connector with name 'B':
This input is 14 bits wide.

An output connector with name 'O':
This output is 14 bits wide.

Invisible control channel attached to this operator:
A local channel from an unnamed control input:
Commands issued are 'muxb' and 'muxa'.

Text for function 'MUXA' of '\LSU\GPU_IN_BUFFER\MUX2_2':
----------------------v----------------------
O:=A.
----------------------^----------------------

Text for function 'MUXB' of '\LSU\GPU_IN_BUFFER\MUX2_2':
----------------------v----------------------
O:=B.
----------------------^----------------------

End of function descriptions.

'\LSU\GPU_IN_BUFFER\A GI' is a register.

This register is 14 bits wide.
The default function is 'hold'.
This register has the following connectors:

A control input connector without a name:
This control input is 1 bit wide.

Control specification:
----------------------v----------------------
%1 load.
%0 hold.
----------------------^----------------------

An input connector without a name:
This input is 14 bits wide.

An output connector without a name:
This output is 14 bits wide.

Invisible control channel attached to this register:

A local channel from an unnamed control input:
Commands issued are 'load' and 'hold'.

This register is 64 bits wide.
The default function is 'hold'.
The value is set to UNK after system reset.
The value loaded for the 'reset' command is 0.

The value is set to UNK after system reset.
The value loaded for the 'reset' command is 0.

This register has the following connectors:

A control input connector without a name:
This control input is 1 bit wide.

Control specification:
----------------------v----------------------
%1 load.
%0 hold.
----------------------^----------------------

An input connector without a name:
This input is 64 bits wide.

An output connector without a name:
This output is 64 bits wide.

Invisible control channel attached to this register:

A local channel from an unnamed control input:
Commands issued are 'load' and 'hold'.

This operator has 1 function.
The default function is 'SPLIT'.

This operator has the following connectors:

An input connector with name 'GIS':
This input is 14 bits wide.

An input connector with name 'GSS':
This input is 64 bits wide.

An output connector with name 'I_GI':
This output is 9 bits wide.

An output connector with name 'I_IC':
This output is 4 bits wide.
An output connector with name 'I_S1':
This output is 32 bits wide.

An output connector with name 'I_S2':
This output is 32 bits wide.

Text for function 'SPLIT' of 'LSU\GPU_IN_BUFFER\SPLIT':
---------------------------------------------------------------------
I_S1:= GSS from: 0 to: 31.
I_S2:= GSS from: 32 to: 63.
I_IC:= GIS from: 5 to: 8.
I_RA:= GIS from: 0 to: 4.
I_ID:= GIS from: 10 to: 13.
I_GI:= _ID, _RA.
---------------------------------------------------------------------

End of function descriptions.

'LSU\GPU_IN_BUFFER\STRIP_ACK' is an operator.
This operator has 1 function.
The default function is 'SPLIT'.
This operator has the following connectors:
A control input connector without a name:
This control input is 1 bit wide.
Control specification:
---------------------------------------------------------------------
%0 disable: GSS; disable: GIS.
%1 enable: GSS; enable: GIS.
---------------------------------------------------------------------

An output connector with name 'ACK':
This output is 1 bit wide.

An input connector with name 'F_GI':
This input is 15 bits wide.

An input connector with name 'F_GS':
This input is 64 bits wide.

A TS output connector with name 'GIS':
The default state is disabled.
This TS output is 14 bits wide.
Invisible control channel attached to this TS output:
A local channel from an unnamed control input:
Commands issued are 'enable' and 'disable' (ILLEGAL COMMAND).
A TS output connector with name 'GSS':
The default state is disabled.
This TS output is 64 bits wide.
Invisible control channel attached to this TS output:
A local channel from an unnamed control input:
Commands issued are 'enable' and 'disable' (ILLEGAL COMMAND).

Text for function 'SPLIT' of 'LSU\GPU_IN_BUFFER\STRIP_ACK':
---------------------------------------------------------------------
ACK := (F_GI at: 14).
GIS := F_GI from: 0 to: 13.
GSS := F_GS.
---------------------------------------------------------------------

End of function descriptions.
B.6. Listing of the MEM_IN_BUFFER block

'\LSU\MEM_IN_BUFFER' is a BBB schematic. This BBB schematic has the following connectors:

- A bus feedthrough connector with name 'I_ACK': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'I_DATA': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'M_ACK': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'M_DATA': This bus feedthrough is 32 bits wide.

Contents of this BBB schematic are unique in this system.

Blocks for BBB schematic '\LSU\MEM_IN_BUFFER':

-------------------------------------------------------------------------------
'\LSU\MEM_IN_BUFFER\DATA' is a register.
This register is 32 bits wide.
The default function is 'hold'.
The value is set to UNK after system reset.
The value loaded for the 'reset' command is 0.

This register has the following connectors:

- A control input connector without a name: This control input is 1 bit wide.
  Control specification:
  ----------------------v----------------------
  %1 load.
  %0 hold.
  ----------------------v----------------------

- An input connector without a name: This input is 32 bits wide.
- An output connector without a name: This output is 32 bits wide.

Invisible control channel attached to this register:
A local channel from an unnamed control input:
Commands issued are 'load' and 'hold'.

-------------------------------------------------------------------------------
'\LSU\MEM_IN_BUFFER\MUX2_1' is an operator.
This operator has 2 functions.
The default function is 'MUXA'.

This operator has the following connectors:

- A control input connector without a name: This control input is 1 bit wide.
  Control specification:
  ----------------------v----------------------
  %1 MUXA.
  %0 MUXB.
  ----------------------v----------------------

- An input connector with name 'A': This input is 32 bits wide.
- An input connector with name 'B': This input is 32 bits wide.
- An output connector with name 'O': This output is 32 bits wide.
Invisible control channel attached to this operator:
A local channel from an unnamed control input:
Commands issued are 'muxb' and 'muxa'.

Text for function 'MUXA' of '\LSU\MEM_IN_BUFFER\MUX2_1':
-------------------------------v-------------------------------
O:=A.
-------------------------------v-------------------------------

Text for function 'MUXB' of '\LSU\MEM_IN_BUFFER\MUX2_1':
-------------------------------v-------------------------------
O:=B.
-------------------------------v-------------------------------

End of function descriptions.
Appendix C Schematics & Listings of the Code Subsystem

C.1. Schematic of the Code Subsystem

C.2. Schematic of the RR? blocks
C.3. Schematic of the CONTROL block

C.4. Schematic of the CACHE block
C.5. Schematic of the LRU block

\[\text{\GICACHE} \text{ is a BBB schematic.}\]

This BBB schematic has the following connectors:

- A bus feedthrough connector with name 'AO': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'A1': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'A2': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'A3': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'Ack': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'Addr': This bus feedthrough is 31 bits wide.
- A bus feedthrough connector with name 'DO': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'D1': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'D2': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'D3': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'Data': This bus feedthrough is 64 bits wide.
- A bus feedthrough connector with name 'FC': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'RO': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'R1': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'R2': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'R3':


\[\text{\GICACHE} \text{ is a BBB schematic.}\]

This BBB schematic has the following connectors:

- A bus feedthrough connector with name 'AO': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'A1': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'A2': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'A3': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'Ack': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'Addr': This bus feedthrough is 31 bits wide.
- A bus feedthrough connector with name 'DO': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'D1': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'D2': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'D3': This bus feedthrough is 32 bits wide.
- A bus feedthrough connector with name 'Data': This bus feedthrough is 64 bits wide.
- A bus feedthrough connector with name 'FC': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'RO': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'R1': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'R2': This bus feedthrough is 1 bit wide.
- A bus feedthrough connector with name 'R3':
This bus feedthrough is 1 bit wide.
A bus feedthrough connector with name 'RDY0':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'RDY1':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'RDY2':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'RDY3':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'Read':
This bus feedthrough is 1 bit wide.

Contents of this BBB schematic are unique in this system.

C.7. Listing of the RR? Blocks

'\ICACHE\RRO' is a BBB schematic.
This BBB schematic has the following connectors:

A bus feedthrough connector with name 'Ack':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'Hold':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'IncSeq':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'MemAck':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'Ready':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'ReqIn':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'ReqOut':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'SeqReq':
This bus feedthrough is 3 bits wide.

A bus feedthrough connector with name 'SeqRes':
This bus feedthrough is 3 bits wide.

A bus feedthrough connector with name 'TPI':
This bus feedthrough is 2 bits wide.

A bus feedthrough connector with name 'TPO':
This bus feedthrough is 2 bits wide.

Contents of this BBB schematic are unique in this system.

Blocks for BBB schematic '\ICACHE\RRO':

---------------------------------------------

'\ICACHE\RRO\CONTROL' is an operator.
This operator has 1 function.
The default function is 'DEFAULT'.
This operator has the following connectors:
An input connector with name 'CacheAck':
This input is 1 bit wide.
An output connector with name 'IncSeq':
This output is 1 bit wide.

An input connector with name 'MemAck':
This input is 1 bit wide.

An input connector with name 'ReqOut':
This output is 1 bit wide.

Text for function 'DEFAULT' of '\ICACHE\RRQ\CONTROL':
----------------------v----------------------
"Split Token Input and status into their elements:"
  _TS := TPI at: 0.
  "A Previous block has the token, but does not use it"
  _TP := TPI at: 1. "I will have the token in the next turn"
  _TI := StatusIn at: 0. "The Token is mine now"
  _State := StatusIn at: 1 width: 2. "The state of the statemachine"
  _Seq := StatusIn at: 3 width: 3. "Sequence number for access to the main memory"
  _SO := State=0. "Waiting for a request"
  _S1 := State=1. "Waiting for an answer from the cache (only 1 block in this state)"
  _S2 := State=2. "Waiting for access to the main memory"
  _S3 := State=3. "Waiting for the result from the main memory (only 1 block in this state)"

  "Calculate the token output, pass the token along if it is mine now;
  skip this turn if the token is mine or the previous block skipped its turn
  and there was no request or I am waiting for a result."
  TPO := _TI, (_TI \/ _TS) \ (Req not \ / _SO not)).

  "Calculate if this block may (and should) pass its address.
  There is a request for data, I am not waiting for data and
  I have the token or my predecessor does not want to use it:"
  _Req := _SO \ (_TI \ / _TS) \ / Req \ / Hold not.

  "I am next in line to access the memory:"
  _ReqMem := (SeqRes=Seq) \ / _S2.

  "If the memory has data to pass back this takes priority"
  ReqOut := MemAck if1: _ReqMem if0: _Req.

  "The Ready signal is asserted if I get a result from the cache in
  state 1 and when I get a result from the memory:"
  RDY := (S1 \ / CacheAck) \ / (S3 \ / MemAck).

  "Increase the sequence number if I tyhe data was not in the cache
  (and thus it has to be fetched from memory):"
  IncSeq := S1 \ / CacheAck not.
"Calculate the new state:
S0: If there was a request: goto state 1
S1: If the data was in the cache: goto state 0
S1: If the memory is not busy: goto state 3 else: goto state 2
S2: If result from memory and I am next in line to access the memory: goto state 3
S3: If result from memory: goto state 0"

\[ \texttt{NewState} = \begin{cases} \texttt{S0} & \text{if \ } \text{Req} \text{ if1: (1 width: 2) if0: 0} \\
\text{if0: (S1} & \text{if1: (CacheAck} \\
\text{if1: (0 width:2)} & \text{if0: ((SeqReq=SeqRes} \\
\text{if1: (3 width:2)} & \text{if0: (2 width:2)})} \\
\text{if0: ((MemAck \land (\_ReqMem \lor \_S3))} & \text{if1: (State inc} \\
\text{if0: (State } & )).
\end{cases} \]

"The new status consist of: the new sequence number if I am in state one (Waiting for a result from the cache otherwise the old one; the calculated new state; and weather or not I have the token:"

\[ \text{StatusOut} = \begin{cases} \text{TI if1: SeqReq if0: \_Seq)} \text{, NewState, (Hold if1: \_TI if0: \_TP).} \\
\end{cases} \]

End of function descriptions.

\[ \text{\texttt{ICACHE\_RR\_Status} is a register.} \]

This register is 6 bits wide.
The default function is 'load'.
The value is 1 after system reset. "1 for RR0, 0 for RR1, RR2 & RR3"
The value loaded for the 'reset' command is 1. "1 for RR0, 0 for RR1, RR2 & RR3"

This register has the following connectors:
An input connector without a name:
This input is 6 bits wide.
An output connector without a name:
This output is 6 bits wide.

C.8. Listing of the CONTROL block

\[ \text{\texttt{ICACHE\_CONTROL} is a BBB schematic.} \]

This BBB schematic has the following connectors:
A bus feedthrough connector with name 'Addr0':
This bus feedthrough is 32 bits wide.
A bus feedthrough connector with name 'Addr1':
This bus feedthrough is 32 bits wide.
A bus feedthrough connector with name 'Addr2':
This bus feedthrough is 32 bits wide.
A bus feedthrough connector with name 'Addr3':
This bus feedthrough is 32 bits wide.
A bus feedthrough connector with name 'CAddr':
This bus feedthrough is 31 bits wide.
A bus feedthrough connector with name 'CBusy':
This bus feedthrough is 1 bit wide.
A bus feedthrough connector with name 'CDataIn':
This bus feedthrough is 64 bits wide.
A bus feedthrough connector with name 'CDataOut':
This bus feedthrough is 64 bits wide.

A bus feedthrough connector with name 'CHit':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'CRead':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'CWrite':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'DataOut':
This bus feedthrough is 32 bits wide.

A bus feedthrough connector with name 'Hold':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'MemAck':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'MemAckO':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'MemAddr':
This bus feedthrough is 31 bits wide.

A bus feedthrough connector with name 'MemDataIn':
This bus feedthrough is 64 bits wide.

A bus feedthrough connector with name 'MemRead':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'R0':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'R1':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'R2':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'R3':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'S0':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'S1':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'S2':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'S3':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'SeqReq':
This bus feedthrough is 3 bits wide.

A bus feedthrough connector with name 'SeqRes':
This bus feedthrough is 3 bits wide.

Contents of this BBB schematic are unique in this system.

Blocks for BBB schematic '\ICACHE\CONTROL':

\ICACHE\CONTROL\CONTROL' is an operator.

This operator has 1 function.
The default function is 'DEFAULT'.

This operator has the following connectors:

An input connector with name 'Addr0':
This input is 32 bits wide.
An input connector with name 'Addr1':
This input is 32 bits wide.

An input connector with name 'Addr2':
This input is 32 bits wide.

An input connector with name 'Addr3':
This input is 32 bits wide.

An output connector with name 'CAddr':
This output is 31 bits wide.

An input connector with name 'CBusy':
This input is 1 bit wide.

An input connector with name 'CDataIn':
This input is 64 bits wide.

An output connector with name 'CRead':
This output is 1 bit wide.

An output connector with name 'CWrite':
This output is 1 bit wide.

An output connector with name 'DataOut':
This output is 32 bits wide.

An input connector with name 'DI':
This input is 2 bits wide.

An output connector with name 'DO':
This output is 2 bits wide.

An input connector with name 'DTI':
This input is 32 bits wide.

An output connector with name 'DTO':
This output is 32 bits wide.

An output connector with name 'Hold':
This output is 1 bit wide.

An input connector with name 'MemAck':
This input is 1 bit wide.

An output connector with name 'MemAckO':
This output is 1 bit wide.

An output connector with name 'MemAddr':
This output is 31 bits wide.

An input connector with name 'MemDataIn':
This input is 64 bits wide.

An output connector with name 'MemRead':
This output is 1 bit wide.

An input connector with name 'RO':
This input is 1 bit wide.

An input connector with name 'R1':
This input is 1 bit wide.

An input connector with name 'R2':
This input is 1 bit wide.

An input connector with name 'R3':
This input is 1 bit wide.

An input connector with name 'SO':
This input is 1 bit wide.

An input connector with name 'S1':
This input is 1 bit wide.

An input connector with name 'S2':

This input is 1 bit wide. 
An input connector with name 'S3':
This input is 1 bit wide. 
An input connector with name 'SHI':
This input is 3 bits wide. 
An output connector with name 'SHO':
This output is 3 bits wide. 
An input connector with name 'STI':
This input is 3 bits wide. 
An output connector with name 'STO':
This output is 3 bits wide. 
An input connector with name 'TAddrIn':
This input is 32 bits wide. 
An output connector with name 'TAddrOut':
This output is 32 bits wide. 

Text for function 'DEFAULT' of '\ICACHE\CONTROL\CONTROL':

```
----------------------v----------------------
"Multiplex the incoming addresses:"
_REQ := R0 \/ R1 \/ R2 \/ R3.
_ADDRIN := ((32 copiesof: MemAck) \/ TAddrIn) \/
(32 copiesof: R0) \/ Addr0 \/
(32 copiesof: R1) \/ Addr1 \/
(32 copiesof: R2) \/ Addr2 \/
(32 copiesof: R3) \/ Addr3).
CAddr := _ADDRIN at: 1 width: 31.
"Split the delayed values and update:"
_WSI := DI at: 0.
_MemAckO := DI at: 1.
_DO := MemAck, (_ADDRIn at: 0).

"Output buffered data or data from cache to the data output:"
_DST := (TAddrIn at: 0) if0: (MemDataIn at: 0 width: 32) if1: (MemDataIn at: 32 width: 32).
_CDataIn := _WSI if0: (CDataIn at: 0 width: 32) if1: (CDataIn at: 32 width: 32).
_DataOut := _MemAckO if1: DTI if0: _CDataIn.

"Output some control signals:"
MemAck0 := _MemAck0.
_CRead := (MemAck \/_MemAck0) not \/_Req.
_CWrite := MemAck.
_Hold := _Busy \/ MemAck.

"Update the sequence numbers:"
_IncSH := S0 \/: S1 \/: S2 \/: S3.
_SHO := _IncSH if1: SHI inc if0: SHI. "(Increase the head if a new request is made"
_STO := MemAck if1: STI inc if0: STI. "(Increase the tail if the memory delivers new data"

"The memory should be read when there is a new request and the sequence numbers had the chance to get updated:"
_MemRead := (SHI -> STI) \/_IncSH. "without pause between two memory read cycles"
_MemRead := ((SHI -> STI) \/_IncSH) \/_MemAck0 not. met 1 clock pause
_MemRead := _MemRead.

"Store the new address for later use if there was a request and not already busy with getting data from memory:"
.Addr := (_Req \/_MemAckO \/(SHI=STI) \/_IncSH not)) if0: TAddrIn if1: _AddrIn.
TAddrOut := _Addr.
MemAddr := _Addr at: 1 width: 31.
```

End of function descriptions.

'\ICACHE\CONTROL\Data' is a register.
The value is 0 after system reset.
The value loaded for the 'reset' command is 0.

This register has the following connectors:

An input connector without a name:
This input is 32 bits wide.

An output connector without a name:
This output is 32 bits wide.

'\ICACHE\CONTROL\DELAY' is a register.
This register is 2 bits wide.
The default function is 'load'.
The value is 0 after system reset.
The value loaded for the 'reset' command is 0.

This register has the following connectors:

An input connector without a name:
This input is 2 bits wide.

An output connector without a name:
This output is 2 bits wide.

'\ICACHE\CONTROL\SeqHead' is a register.
This register is 3 bits wide.
The default function is 'load'.
The value is 0 after system reset.
The value loaded for the 'reset' command is 0.

This register has the following connectors:

An input connector without a name:
This input is 3 bits wide.

An output connector without a name:
This output is 3 bits wide.

'\ICACHE\CONTROL\SeqTail' is a register.
This register is 3 bits wide.
The default function is 'load'.
The value is 0 after system reset.
The value loaded for the 'reset' command is 0.

This register has the following connectors:

An input connector without a name:
This input is 3 bits wide.

An output connector without a name:
This output is 3 bits wide.

'\ICACHE\CONTROL\TAddr' is a register.
This register is 32 bits wide.
The default function is 'load'.
The value is 0 after system reset.
The value loaded for the 'reset' command is 0.

This register has the following connectors:

An input connector without a name:
This input is 32 bits wide.
C.9. Listing of the CACHE block

'\GICACHE\CACHE' is a BBB schematic.

This BBB schematic has the following connectors:

A bus feedthrough connector with name 'Address':
This bus feedthrough is 31 bits wide.

A bus feedthrough connector with name 'Busy':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'DataIn':
This bus feedthrough is 64 bits wide.

A bus feedthrough connector with name 'DataOut':
This bus feedthrough is 64 bits wide.

A bus feedthrough connector with name 'Flush':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'Hit':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'Read':
This bus feedthrough is 1 bit wide.

A bus feedthrough connector with name 'Write':
This bus feedthrough is 1 bit wide.

Contents of this BBB schematic are unique in this system.

Blocks for BBB schematic '\GICACHE\CACHE':

===================================================================

'\GICACHE\CACHE\CONTROL' is an operator.
This operator has 1 function.
The default function is 'default'.

This operator has the following connectors:

An input connector with name 'Addr':
This input is 31 bits wide.

An output connector with name 'Busy':
This output is 1 bit wide.

An output connector with name 'C0':
This output is 2 bits wide.

An output connector with name 'C1':
This output is 2 bits wide.

An output connector with name 'C2':
This output is 2 bits wide.

An output connector with name 'C3':
This output is 2 bits wide.

An input connector with name 'D0':
This input is 64 bits wide.

An input connector with name 'D1':
This input is 64 bits wide.

An input connector with name 'D2':
This input is 64 bits wide.
An input connector with name 'D3':
This input is 64 bits wide.

An output connector with name 'DataLoad':
This output is 1 bit wide.

An output connector with name 'DataOut':
This output is 64 bits wide.

An input connector with name 'Flush':
This input is 1 bit wide.

An input connector with name 'FlushIn':
This input is 9 bits wide.

An output connector with name 'FlushOut':
This output is 9 bits wide.

An output connector with name 'Hit':
This output is 1 bit wide.

An output connector with name 'LRUAddr':
This output is 9 bits wide.

An input connector with name 'LRUIn':
This input is 3 bits wide.

An output connector with name 'LRUOut':
This output is 3 bits wide.

An output connector with name 'LRUWrite':
This output is 4 bits wide.

An output connector with name 'RAMAddr':
This output is 10 bits wide.

An input connector with name 'Read':
This input is 1 bit wide.

An input connector with name 'StatusIn':
This input is 3 bits wide.

An output connector with name 'StatusOut':
This output is 3 bits wide.

An input connector with name 'TAddrIn':
This input is 31 bits wide.

An output connector with name 'TAddrOut':
This output is 31 bits wide.

An output connector with name 'TagAddr':
This output is 9 bits wide.

An input connector with name 'TagInHi':
This input is 46 bits wide.

An input connector with name 'TagInLo':
This input is 46 bits wide.

An output connector with name 'TagOutHi':
This output is 46 bits wide.

An output connector with name 'TagOutLo':
This output is 46 bits wide.

An output connector with name 'TagWrite':
This output is 2 bits wide.

An input connector with name 'Write':
This input is 1 bit wide.

Text for function 'default' of '\GICACHE\CACHE\CONTROL':
------------------------------------------v----------------------
"These values specify the cache parameters:"
_WordBits := 1. "#bits necessary to encode #Words/Line"
_ValidBits := 2. "2**_WordBits, equals #Words/Line"
_LineBits := 9. "#bits necessary to encode #Words/Way"

"Calculate some other field width from the parameters:"
_TeqBits := Addr width - _LineBits - _WordBits.
_SetBits := _TagBits + _ValidBits.

"Split the Tag into its elements"
_Tag0 := TagInLo at: 0 width: _TagBits.
_Valid0 := TagInLo at: _SetBits width: ValidBits.
_Tag1 := TagInLo at: _SetBits width: _TagBits.
_Valid1 := TagInLo at: _SetBits + _TagBits width: _ValidBits.
_Tag2 := TagInHi at: 0 width: _TagBits.
_Valid2 := TagInHi at: _TagBits width: _ValidBits.
_Tag3 := TagInHi at: _SetBits width: _TagBits.
_Valid3 := TagInHi at: _SetBits + _TagBits width: _ValidBits.

"Split the LRU into its elements:"
_LRU_0_1 := LRUIn at: 2.
_LRU_0_2 := LRUIn at: 1.
_LRU_2_3 := LRUIn at: 0.

"Split the Previous Status:"
_DWrite := StatusIn at: 2.
_PrevWrite := StatusIn at: 1.
_PrevRead := StatusIn at: 0.

"Check for Flush and increment Flush counter if necessary:"
_Flush := FlushIn=0 \ Flush.
_Normal := _Flush not.
FlushOut := _Flush if1: FlushIn inc if0: FlushIn.

"The only situation in which this cache can be busy is during flush or when there was a write request in the previous cycle:"
_Busy := _Flush \ _PrevWrite \ _DWrite.

"Compute the internal write state"
_Write := (Write \ _PrevRead not) \ _DWrite.

"Split the Address into its elements:"
_Addr := (_PrevWrite \ _DWrite) if0: Addr if1: TAddrIn.
_Word := _Addr at: 0 width: _WordBits.
_Line := _Addr at: _WordBits width: _LineBits.
_Tag := TAddrIn at: _WordBits+LineBits width: _TagBits.

"Calculate the which bit in of the valid state should be checked:"
_ValidMask := (1 width: _ValidBits) shl: _PrevWord.

"Update Status:"
_StatusOut := _Flush if1: 3 zeroes if0: (Write \ _PrevRead \ _PrevWrite not \ _DWrite not), _Write, Read.
_TAddrOut := (Write \ Read) if0: Addr if1: TAddrIn.
_DataLoad := Write \ (_PrevRead \ _DWrite not).

"Address the memories:"
_RAMAddr := _Line, _Word.
_TAddrAddr := _Flush if1: FlushIn if0: _Line.
_LRUAddr := _Flush if1: FlushIn if0: (Write if1: _NewLine if0: _OldLine).

"Check with which Tag in memory the Tag in the address correspons:"
_TEq0 := _Tag = _Tag0.
_TEq1 := _Tag = _Tag1 \ _TEq0 not.
_TEq2 := _Tag = _Tag2 \ _TEq0 not \ _TEq1 not.
_TEq3 := _Tag = _Tag3 \ _TEq0 not \ _TEq1 not \ _TEq2 not.
_TEq := (_TEq0 \ _TEq1 \ _TEq2 \ _TEq3).

"Check if the data on that location in the cache is valid:"
_Hit0 := (~Valid0 \ ~ValidMask) = 0 \ _TEq0.
_Hit1 := (~Valid1 \ ~ValidMask) = 0 \ _TEq1.
_Hit2 := (~Valid2 \ ~ValidMask) = 0 \ _TEq2.
_Hit3 := (~Valid3 \ ~ValidMask) = 0 \ _TEq3.
"Make the Hit line high, if the data is found in memory and if there was a request for
data and if the cache is not flushing itself."

\[ \_Hit := (_Hit0 \lor _Hit1 \lor _Hit2 \lor _Hit3). \]

\[ _Hit := \_Hit \land \_PrevRead \land \_Normal. \]

"Put the correct 'hit' data on the output if there was a read request and the cache is
not flushing:

\[ \text{DataOut} := _\text{Hit}0 \text{ if } 1 : _\text{Hit}0 \text{ if } 0 : \neg \text{Hit}2 \text{ if } 0 : \neg \text{Hit}3 \text{ if } 0 : 0\). \]

"Write the data to the correct way:

If there was already data in the cache with the same tag then update that word,
otherwise evict the least recently used line (determined by the pseudo LRU
algorithm):

\[ _\text{CO} := \_TEq0 \lor \_LRU_0_1 \land \_TEq \not. \]
\[ _\text{C1} := \_TEq1 \lor \_LRU_0_1 \not \land \_TEq \not. \]
\[ _\text{C2} := \_TEq2 \lor \_LRU_0_2 \not \land \_LRU_2_3 \land \_TEq \not. \]
\[ _\text{C3} := \_TEq3 \lor \_LRU_0_2 \not \land \_LRU_2_3 \not \land \_TEq \not. \]

\[ \text{ActMem} := (\text{Read} \land \_PrevWrite) \lor \_Normal. \]
\[ _\text{C0} := _\text{ActMem}, (\_PrevWrite \text{ if } 1 : _\text{C0} \text{ if } 0 : 1 \text{ zeroes}). \]
\[ _\text{C1} := _\text{ActMem}, (\_PrevWrite \text{ if } 1 : _\text{C1} \text{ if } 0 : 1 \text{ zeroes}). \]
\[ _\text{C2} := _\text{ActMem}, (\_PrevWrite \text{ if } 1 : _\text{C2} \text{ if } 0 : 1 \text{ zeroes}). \]
\[ _\text{C3} := _\text{ActMem}, (\_PrevWrite \text{ if } 1 : _\text{C3} \text{ if } 0 : 1 \text{ zeroes}). \]

"Update the tag and valid information:

if a word is written to the cache, set the tag to the upper address bits,
and set the valid bit for that word:

\[ _\text{NewTag}0 := _\text{CO if } 0 : _\text{Tag}0 \text{ if } 1 : _\text{Tag}. \]
\[ _\text{NewValid}0 := _\text{CO if } 0 : _\text{Valid}0 \text{ if } 1 : \neg _\text{Valid}0. \]
\[ _\text{NewTag}1 := _\text{C1 if } 0 : _\text{Tag}1 \text{ if } 1 : _\text{Tag}. \]
\[ _\text{NewValid}1 := _\text{C1 if } 0 : _\text{Valid}1 \text{ if } 1 : \neg _\text{Valid}1. \]
\[ _\text{NewTag}2 := _\text{C2 if } 0 : _\text{Tag}2 \text{ if } 1 : _\text{Tag}. \]
\[ _\text{NewValid}2 := _\text{C2 if } 0 : _\text{Valid}2 \text{ if } 1 : \neg _\text{Valid}2. \]
\[ _\text{NewTag}3 := _\text{C3 if } 0 : _\text{Tag}3 \text{ if } 1 : _\text{Tag}. \]
\[ _\text{NewValid}3 := _\text{C3 if } 0 : _\text{Valid}3 \text{ if } 1 : \neg _\text{Valid}3. \]

"Update the LRU information:

Set the LRU_0_2 bit if the hit (or update) took place in way 2 or 3;
set the LRU_0_1 bit if the hit (or update) was in way 1, reset it if it was in way 0
otherwise keep the old bit;
set the LRU_2_3 bit if the hit (or update) was in way 3, reset it if it was in way 2
otherwise keep the old bit.

\[ \text{LRUOut} := \_\text{Flush if } 1 : 0 \text{ if } 0 : \neg \text{LRU}_0_2, \_\text{LRU}_0_1, \neg \text{LRU}_0_2. \]
\[ \text{LRUWrite} := (\_\text{Flush} \lor \_\text{PrevWrite} \lor \_\text{Read}), (\_\text{Flush} \lor \_\text{PrevWrite}). \]

End of function descriptions.

\[ \text{\GICACHE\CACHE\FlushCount} \text{ is a register.} \]

This register is 9 bits wide.
The default function is 'load'.
The value is 0 after system reset.
The value loaded for the 'reset' command is 0.
This register has the following connectors:
An input connector without a name:
This input is 9 bits wide.

An output connector without a name:
This output is 9 bits wide.

\'\GICACHE\CACHE\RAM0\' is a RAM. (same for RAM1, RAM2 & RAM3)
This RAM uses the 'Single ported synchronous RAM' technology.
It contains 1024 words of 64 bits each.
There is no contents file attached.
The contents are set to UNK after system reset.
The functions of the (non-control) connectors are as follows:
The following port is available:

Combined read/write port:
Address input connector: 'A'.
Data input connector: '01'.
Data output connector: '~O'.

This port's technology is called 'Synchronous read/write port'.
Cycle and mode settings are as follows:
A read cycle takes 1 clock.
The internal state is sampled in clock cycle 1.
The output becomes valid in clock cycle 1, and automatically enables it's TS output in that cycle.
The default reading command is 'noread'.

A write cycle takes 1 clock.
The input must be valid when the write cycle is started.
The memory cells are updated in clock cycle 1.
The default writing command is 'nowrite'.
A read command may not be issued before clock cycle 1 of a write command.
A write command may not be issued before clock cycle 1 of a read command.
(These connectors are explained in more detail below.)

This RAM has the following connectors:
A control input connector without a name:
This control input is 2 bits wide.

Control specification:
------------------------------------------v----------------------
%11 write.
%10 read.
------------------------------------------'----------------------

An input connector with name 'A':
This input is 10 bits wide.

An input connector with name 'DI':
This input is 64 bits wide.

An output connector with name 'DO':
This output is 64 bits wide.

Invisible control channel attached to this RAM:
A local channel from an unnamed control input:
Commands issued are 'write' and 'read'.

\'\GICACHE\CACHE\RAM1\' is a RAM. (see for listing RAMO)
'\GICACHE\CACHE\RAM2' is a RAM. (see for listing RAM0)

'\GICACHE\CACHE\RAM3' is a RAM. (see for listing RAM0)

'\GICACHE\CACHE\Status' is a register.
This register is 3 bits wide.
The default function is 'load'.
The value is 0 after system reset.
The value loaded for the 'reset' command is 0.
This register has the following connectors:
An input connector without a name:
This input is 3 bits wide.
An output connector without a name:
This output is 3 bits wide.

'\GICACHE\CACHE\TAddress' is a register.
This register is 31 bits wide.
The default function is 'load'.
The value is set to UNK after system reset.
The value loaded for the 'reset' command is 0.
This register has the following connectors:
An input connector without a name:
This input is 31 bits wide.
An output connector without a name:
This output is 31 bits wide.

'\GICACHE\CACHE\TagHi' is a RAM. (same for TagLo)
This RAM uses the 'Single ported synchronous RAM' technology.
It contains 512 words of 46 bits each.
There is no contents file attached.
The contents are set to UNK after system reset.
The functions of the (non-control) connectors are as follows:
The following port is available:
Combined read/write port:
  Address input connector: 'A'.
  Data input connector: 'DI'.
  Data output connector: 'DO'.
This port's technology is called 'Synchronous read/write port'.
Cycle and mode settings are as follows:
  A read cycle takes 1 clock.
The internal state is sampled in clock cycle 1.
The output becomes valid in clock cycle 1,
and automatically enables its TS output in that cycle.
The default reading command is 'noread'.
  A write cycle takes 1 clock.
The input must be valid when the write cycle is started.
The memory cells are updated in clock cycle 1.
The default writing command is 'nowrite'.
A read command may not be issued before clock cycle 1 of a write command.
A write command may not be issued before clock cycle 1 of a read command.
This RAM has the following connectors:

A control input connector without a name:
This control input is 2 bits wide.

Control specification:
-------------------------------v-------------------------------
%11 write.
%10 read.
-------------------------------^-------------------------------

An input connector with name 'A':
This input is 9 bits wide.

An input connector with name 'DI':
This input is 46 bits wide.

An output connector with name 'DO':
This output is 46 bits wide.

Invisible control channel attached to this RAM:

A local channel from an unnamed control input:
Commands issued are 'write' and 'read'.

'\GICACHE\CACHE\TagLo' is a RAM. (see for rest of listing TagHi)

'\GICACHE\CACHE\TDataln' is a register.

This register is 64 bits wide.
The default function is 'hold'.
The value is set to UNK after system reset.
The value loaded for the 'reset' command is 0.

This register has the following connectors:

A control input connector without a name:
This control input is 1 bit wide.

Control specification:
-------------------------------v-------------------------------
1 load.
-------------------------------^-------------------------------

An input connector without a name:
This input is 64 bits wide.

An output connector without a name:
This output is 64 bits wide.

Invisible control channel attached to this register:

A local channel from an unnamed control input:
The only issued command is 'load'.

C.10. Listing of the LRU block

'\GICACHE\CACHE\LRU' is a BBB schematic.

This BBB schematic has the following connectors:

A bus feedthrough connector with name 'A':
This bus feedthrough is 9 bits wide.

A bus feedthrough connector with name 'I':
This bus feedthrough is 3 bits wide.
A bus feedthrough connector with name 'W':
This bus feedthrough is 4 bits wide.

A bus feedthrough connector with name 'O':
This bus feedthrough is 3 bits wide.

Contents of this BBB schematic are unique in this system.

Blocks for BBB schematic '\GICACHE\CACHE\LRU':

This RAM uses the 'Single ported synchronous RAM' technology. It contains 512 words of 1 bit each. There is no contents file attached. The contents are set to UNK after system reset.

The functions of the (non-control) connectors are as follows:

Combined read/write port:
Address input connector: 'A'.
Data input connector: 'I'.
Data output connector: 'O'.

This port's technology is called 'Synchronous read/write port'.

Cycle and mode settings are as follows:

A read cycle takes 1 clock.
The internal state is sampled in clock cycle 1.
The output becomes valid in clock cycle 1, and automatically enables its TS output in that cycle.
The default reading command is 'noread'.

A write cycle takes 1 clock.
The input must be valid when the write cycle is started.
The memory cells are updated in clock cycle 1.
The default writing command is 'nowrite'.

A read command may not be issued before clock cycle 1 of a write command.
A write command may not be issued before clock cycle 1 of a read command.

(These connectors are explained in more detail below.)

This RAM has the following connectors:

A control input connector without a name:
This control input is 2 bits wide.

Control specification:

A local channel from an unnamed control input:
Commands issued are 'write' and 'read'.

'\GICACHE\CACHE\LRU\LRU_01_23' is a RAM. (same for LRU_0_1 & LRU_2_3)
'\GICACHE\CACHE\LRU\LRU_2_3' is a RAM. (for listing see LRU_01_02)

'\GICACHE\CACHE\LRU\LRU_SPLIT' is an operator.

This operator has 1 function.
The default function is 'DEFAULT'.

This operator has the following connectors:

An input connector with name 'I':
This input is 3 bits wide.

An input connector with name 'I0':
This input is 1 bit wide.

An input connector with name 'I1':
This input is 1 bit wide.

An input connector with name 'I2':
This input is 1 bit wide.

An output connector with name 'O':
This output is 3 bits wide.

An output connector with name 'O0':
This output is 1 bit wide.

An output connector with name 'O1':
This output is 1 bit wide.

An output connector with name 'O2':
This output is 1 bit wide.

An input connector with name 'W':
This input is 4 bits wide.

An output connector with name 'W0':
This output is 2 bits wide.

An output connector with name 'W1':
This output is 2 bits wide.

An output connector with name 'W2':
This output is 2 bits wide.

Text for function 'DEFAULT' of '\GICACHE\CACHE\LRU\LRU_SPLIT':

```
<table>
<thead>
<tr>
<th>Act</th>
<th>W at: 3.</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>_Act,</td>
</tr>
<tr>
<td>W1</td>
<td>(W at: 2)</td>
</tr>
<tr>
<td>W0</td>
<td>_Act,</td>
</tr>
<tr>
<td></td>
<td>(W at: 0)</td>
</tr>
<tr>
<td>O2</td>
<td>I at: 2.</td>
</tr>
<tr>
<td>O1</td>
<td>I at: 1.</td>
</tr>
<tr>
<td>O0</td>
<td>I at: 0.</td>
</tr>
<tr>
<td>O</td>
<td>I2, I1, I0.</td>
</tr>
</tbody>
</table>
```

End of function descriptions.