# An instruction cache in IDaSS

Hu, Y.C.

*Award date:*
1992

May - December 1989.

Master thesis report:

# AN INSTRUCTION CACHE IN IDASS.

By: *Y.C. Hu*
Supervisor: *Prof. Ir. M.P.J. Stevens*
Coaches: *Ir. W.J. Withagen*
*Ir. F.P.M. Budzelaar*

Eindhoven University of Technology,
Department of Electrical Engineering,
Digital Systems Group.

# ABSTRACT.

This master thesis report is the result of my graduation project in the Digital Systems group of the department of Electrical Engineering of the Eindhoven University of Technology. After a theoretical study on instruction caches, and after several trace driven simulations of the cache performance with varying parameters, an optimum instruction cache implementation is chosen. The choice of the cache parameters is targeted towards an instruction cache needed for the C-processor. The C-processor is in development at the Digital Systems group, but the cache has a more general character. The traces, needed for the simulations, are artificially generated due to the lack of a compiler for the C-processor. The optimum instruction cache is implemented as a prototype with the Interactive Design and Simulation System (IDaSS) CAD tool.

The prefetch_lookup_on_hits algorithm is used as prefetch method and the Least Recently Used algorithm is used as replacement method. The cache has a 2-way-set-associative mapped organisation. The used parameters are:

- a cache size of 4K Byte.
- a block size of 32 quads (4 Bytes).
- a transfer block size of 8 quads.
- a set size of 16 blocks.

A read buffer and a fetch buffer are used to improve the performance of the cache.

The prototype has passed several functional tests successfully. To create a cache model with variable parameters, it is better to implement the cache in the Helix environment instead of using IDaSS. The reason is the lack of flexibility of the IDaSS system.

## *ACKNOWLEDGEMENTS.*

In the first place I would like to thank my coaches Ir. Willem Jan Withagen and Ir. Frank Budzelaar and professor Stevens. They supported me, but gave me also large freedom in this graduation project. I also would would like to thank Ir. Jean Paul Smeets and Ir. Jos Bormans, with whom I had several valuable discussions, and especially thanks to Ir. Ad Verschueren, from whom I received a lot of help with the IDaSS CAD tool.

# TABLE OF CONTENTS.

# 1.
# INTRODUCTION

The development in the integration technology made it possible to create extremely complicated integrated circuits. This is the reason that efficient design tools are needed to handle the complexity of our designs. At this moment there is still a big gap between the idee of the designer and an useful design tool. At the Digital Systems Group, of the Department of Electrical Engineering of the Eindhoven University of Technology, a project Structured Analyzes and Structured Design (SASD) is started to gain useful design tools to fill this gap. To test the developed design tools of SASD a VLSI project, the C-processor, was started. This processor is a dedicated microprocessor for effective execution of programs written in the C programming language, and related high level languages.

The C-processor project was started by F.P.M. Budzelaar [Budzelaar] in 1987, coached by Prof. Ir. M.P.J. Stevens. He specified the requirements to execute compiled C-programs efficiently and defined the global processor architecture. W.J. Withagen [Withagen] worked out the instruction set and made a software model to test the defined instruction set and its functionality. He also developed a hardware model that can be used as a testing vehicle for further decompositions and gate level implementations. J.E.H.M. Bormans [Bormans] studied different cache organisations and made trace driven simulations to investigate the performance of the different organisations.

The C-processor is a high performance microprocessor, however the rate at which data can be supplied to the microprocessor has become a limiting factor. The causes of this limiting factor are the pin count constraints and the relative small progress in the cost versus speed ratio of the memory. Therefor cache memories are used to level out the speed mismatch between the processor and the main memory. The processor contains separated cache memories, the instruction and data caches.

My graduation project was to implement an Instruction cache generator, comparable to the already existing RAM and PLA generators. First a prototype of the cache implementation has to be made with the design tool, Interactive Design and Simulation System for ULSI. This system is developed by Ir. A Verschueren as part of the mentioned SASD project. Due to the lack of time no attention could be spend to the parameterize the implemented prototype. Thus, this master thesis raport contains only the prototype implementation of the instruction cache.

The theory of the instruction caches, as studied by J.E.H.M. Bormans, is repeated in chapter 2. The interface definitions are presented in chapter three. The fourth chapter treats the decomposition of the cache in smaller modules. The next two chapters describe the main modules. And finally the chapter seven contains the conclusions and recommendations.

The cache is developed with Interactive Design and Simulation System (IDaSS) as CAD tool. This is done because the expectations were that interactive designing would be faster than the Silvar Lisco packet. And the cache was also a testing vehicle for IDaSS. The theory of Cache organisations are described in the master thesis report of J.E.H.M. Bormans [Bormans]. Only the results of this report are described in this chapter, for further reading it is recommended to read this report.

## 2.1. The cache organisation.

There are three common ways of cache organisation:

1. fully associative mapped.
2. direct mapped.
3. set associative mapped.

The fully associative mapped and the direct mapped are special cases of set associative mapping. If set size is 1, the mapping is direct. If the set size is equal to the total number of blocks, the mapping is fully associative. The set associative mapped organisation will be used for the prototyping and therefor only this organisation will be discussed, the other organisations are described in [Bormans]. In a set associative mapped cache is the memory address divided in four fields: Tag field, set field, transfer

block field and word field. A model of a two way set associative mapped cache is illustrated in Figure 1.

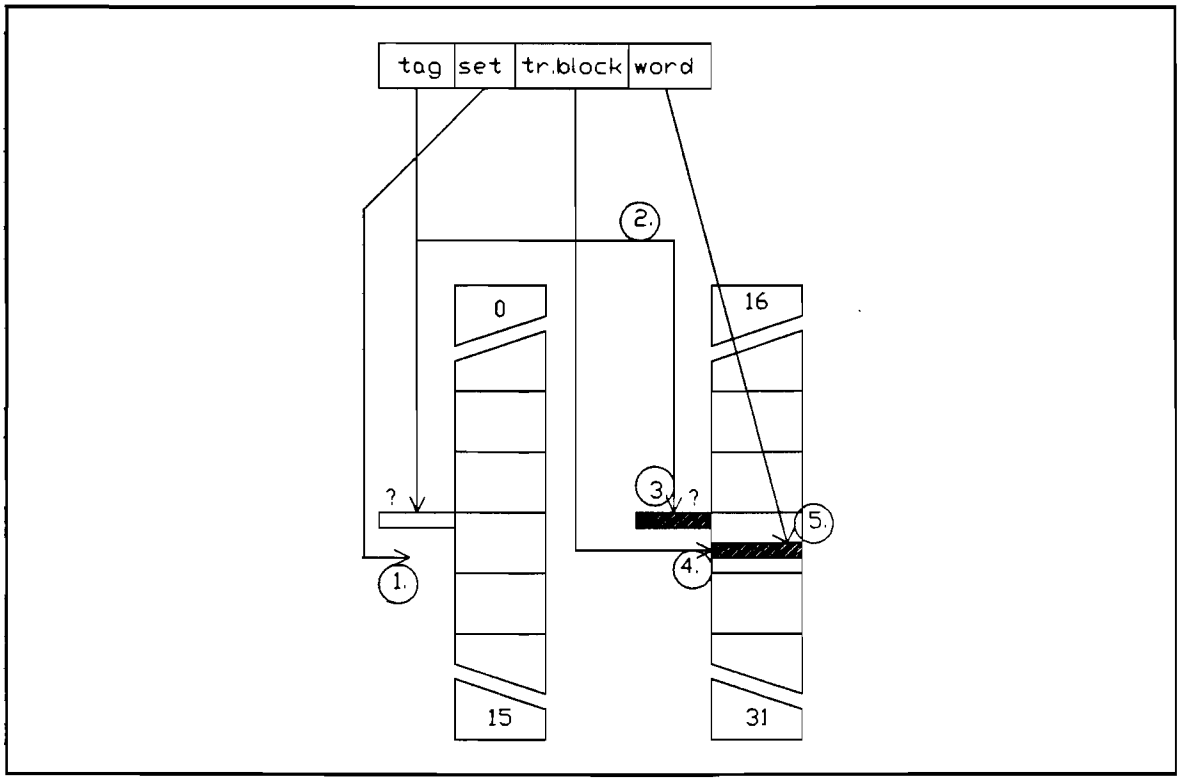


**Figure 1.** The set associative mapped cache.

The following items can be distinguished in processing the request for data.

1. the set field points to the set of cache blocks that may contain the block with the requested quad.
2. the tag field is compared against the tags of the blocks in the set.
3. if one of the tags match, the status information of that block is checked.
4. if there is a hit, the transfer block field is used to select the transfer block within the block.
5. after the transfer block is selected, the word field selects the quad within the transfer block.

## 2.2. The cache parameters.

Besides the mapping method, also the sizes of a number of cache parameters have to be chosen. The following parameters are of influence on the performance of the cache. In this paragraph the cache size, set size and block size will be discused. Transfer blocks will be introduced in the section about block sizes. The size of these transfer blocks will be discused in the section 2.2.4.

### 2.2.1. Cache size.

The area required for a cache can be divided into three parts:

- RAM for data
- RAM for overhead information (tag bits, status information, etc.)
- area required for multiplexers, buses, control logic, etc.

A numerical example that is used for the prototype of the instruction cache, its size is 1K quads, is:

| | | | |
|---|---|---|---|
| - data RAM | : 1024 X 32 | = 32,768 | bits. |
| - tag/status RAM | : 2 X (37 + 34) | = 142 | bits. |
| - multiplexers | : 4 X 256 | = 1024 | bits wide. |

It is obvious that the size of the data RAM is the most important parameter in the calculation of the used area on the chip.

The number of quads, which can be stored in the cache, is known as the most important cache parameter, since it highly affects the miss ratio and the cache size. The larger the cache, the more old information can be saved. This decreases the miss ratio. Since the usefulness of old information declines in time, only a slight improvement of cache performance is found beyond a certain cache size.

### 2.2.2. Set size.

Set size is just a different term for the scope of associative search. The larger the set size, the lower the miss ratio will be. This is because a larger placement freedom improves the performance of the replacement algorithm. However, beyond a certain set size, this improvement is likely to increase very little if any at all. Therefore the miss ratio will not decrease significantly beyond a certain set size. A smaller set size, on the other hand, results in a faster and less expensive implementation.

### 2.2.3. Block size.

The main advantage of grouping the quads in blocks is that the amount of the address tags, replacement information, buffer circuitry, etc, decreases. This block overhead decreases when the block size is enlarged. Besides the decrease in block overhead, there are some other positive effects of enlarging the block size.

- More adjacent data is stored at once, which tends to reduce the miss ratio. Prefetching could be, from this point of view, an alternative for large blocks.
- A block can be read in burst mode, which reduces the average main memory access time.
- A smaller penalty of cache flushes, which occur in many caches during task switching, is obtained by enlarging the block size. This way the cache will be filled faster.

There are two important disadvantages of enlarging the block size. Firstly, the miss ratio will increase, because a complete block instead of one quad has to be fetched from main memory. Secondly, the usefulness of extra loaded quads becomes less than the usefulness of the replaced quads. Quads probably will be loaded into the cache which never will be used. This effect is called memory pollution. Both miss and traffic ratio, the traffic between the main memory and the cache, increase because of memory pollution. Another negative effect of enlarging the block size is that placement freedom will be reduced, since all quads in a block must have the same tag.

There is however a way to limit the increase in traffic ratio and miss penalty, without sacrificing the savings in overhead bits. This is possible by dividing the blocks in smaller transfer blocks. This will be discussed in the next section.

It will be clear that the block size has a great impact on the performance of the cache. The size, the access time, the miss ratio and the traffic ratio are influenced by it, some of these because of different effects.

### 2.2.4. Transfer block size.

In the remainder of the text "blocks" will be used to indicate groups of quads, which have the same address tags, "transfer blocks" for groups of quads which are transferred as one unit between main memory and cache.

Each quad in a transfer block requires a data_valid bit, since not all quads of a transfer block have to be stored in the cache (paragraph 3.4). Other status information, like replacement information, is required for a complete block only. Dividing blocks in transfer blocks increases the number of overhead bits thus only very slightly.

Although dividing blocks in transfer blocks is positive for the miss penalty and the traffic ratio., it is not for the miss ratio. The miss ratio will be generally be higher, because only a part of a block contains valid data. Since the usage of small transfer blocks permits larger blocks, it could also be considered as a way to reduce overhead bits and thus to obtain a cache which is able to store more quads. Therefore the miss ratio will not necessary be higher. Another disadvantage is that the placement freedom will be reduced. This can be seen by considering that instead of dividing a large block in smaller transfer blocks, a number of small (transfer) blocks are enforced to have the same tag.

## 2.3. Prefetch algorithm.

There are two ways to fetch a block from main memory:
- demand fetch: the block is fetched when it is needed.
- prefetch: the block is fetched before it is needed.

A simple method to store sequential information is the usage of large blocks. However, this has many disadvantages as we have seen in section 3.2.3. The simplest prefetch method used is known as one-block-lookahead: the subsequent block of the currently referenced one is considered for prefetching.

If too much information is prefetched, memory pollution appears as it did for large block sizes. The smaller the cache, the larger this effect will be. Prefetching will always increase the traffic ratio. Another problem is that the storage prefetched quads will delay the servicing of requests from the processor, unless multiported RAM is used. These effects can be limited by dividing the blocks into smaller prefetch blocks. The link to the previously introduced transfer block will be clear.

The one-transfer-block-lookahead method can be used in different ways. One method is called prefetch_lookup: the prefetch of the next transfer block is based on its absence in the cache. This method has the disadvantage of an extra cache access. Such extra access are likely to increase the average cache access time. To limit these extra cache access this method can be refined by only prefetching in case of a hit (prefetch_lookup_on_hits).

## 2.4. Replacement algorithm.

The replacement algorithm has to decide which block in the set will be removed to store a main memory block. This happens when:

- a miss occurs. However, replacement is not always necessary. The block to which the missing transfer block belongs, can already be in the cache.
- the block to which the prefetched transfer block belongs is not in the cache.

A replacement algorithm is the LRU (least-recently-used) algorithm. The LRU algorithm is usage-based, which take the record of usage of a block into account. LRU is based on the theory that if a block has been often and recently used, it is likely to be referenced in the near future.

For LRU a table is required for each set, containing the replacement order of the blocks in the set. If a new block has to be stored in the set, the block referred to by the top of the table is used. That block reference will be placed at the bottom of the table and the other references will be shifted one place up. If a block is used which is in the cache, its reference will be put at the bottom of the table and the reference, which were below it, will be moved one place up. However , if the set size is 2, just one bit is required to decide which block has to be replaced. Updating the replacement information is just a matter of setting or resetting it, depending on which of the 2 blocks is used.

## 2.5. Architectural considerations.

The cache has to be able to service quad requests from the processor while it is prefetching. The best way to achieve this parallelism is to use separated working modules. One which controls requests from the processor, one which controls the fetching. The former will be called server, the latter fetcher.

To limit the increasing in the cache access time due to prefetching is to decrease the duration of storing a prefetched block. This can be done by placing a buffer between the main memory and cache memory. The advantage of this buffer is:

- The quads are stored in the buffer at main memory read speed and the whole fetched transfer block is stored at once in the cache RAM at cache RAM write speed. This requires a "wide" RAM.

The above mentioned buffer will be called fetch buffer from now on.

Another way to limit the above mentioned increase of the cache access time is to limit the number of cache accesses to the cache RAM. This can be obtained in the following way:

- Decreasing the number of accesses to the RAM by the fetcher. This is possible by the above mentioned fetch buffer and by avoiding unnecessary prefetches by using the prefetch_lookup method.
- Decreasing the number of accesses to the RAM by the server. This is possible by using a buffer between the cache and the instruction unit. If a request from the instruction unit results in a cache memory hit, the total transfer block is copied in this buffer. The following quads do not require access to the cache memory, as long as they belong to the same transfer block and their valid bits are set. The valid bits are necessary because it is possible that only a part of the transfer block is loaded into the cache memory.

The last mentioned buffer, which will be indicated by read buffer in the remainder of the text, has another and even larger advantage: This small buffer can be read much faster than the cache RAM itself, for it is much smaller and no associative search has to be done.

A miss is likely to be caused by a jump in the reference pattern, which itself can be caused by a conditional expression, a function call, a function return, a task switch, an interrupt, etc. A miss can also occur when quads of a transfer block are requested which

is still being prefetched. It is also possible that another transfer block is being prefetched when a miss occurs. Thus, three different cases can be distinguished when a miss occurs:

1. The fetcher is not busy. This represents the miss penalty in its basic form.
2. The fetcher is still prefetching the transfer block which caused the miss.
3. The fetcher is busy prefetching a transfer block different from the needed one.

If the first case occurs, the fetching of the required quad can be accelerated by using wrap around. The requested quad is fetched first, followed by the quads in the transfer block which have a higher address and finally those which have a lower address. If wrap around is used, the fetching can be accelerated more by using fetch bypass: the requested quad pair is passed directly from the main memory to the instruction unit and is stored in the RAM at the same time or later.
Wrap around can be improved by fetching only the required quad pair and the quads, which have a higher address, of the same transfer block. The lower quads will be skipped. The last mentioned idee is based on the fact that it is not likely that these quads will be requested. In this way the penalty of jumps will be minimised.

If the second case occurs, the same problems as mentioned above are present. But now it is likely that the next fetched quad pair will be the required one. Therefor is it better to wait for the quads and not to interrupt the fetcher. It will be clear that the usage of a fetch buffer, with quad_valid bits for each quad, is the best solution.

If the third case occurs, it is wise to stop fetching the other quads in the fetched transfer block, and write the quads, with the correct quad_valid bits, in the cache memory. Start a new fetch cycle of the required transfer block, using wrap around. The above mentioned idee will be called "stop prefetching", in case of stopping prefetch actions, and "stop demand fetching", in case of stopping demand fetch actions. The fetched quads can not be skipped, otherwise it would theoretically be possible that the cache is trashing: repeatedly losing the fetched quads and a moment later fetching them again.

In Figure 2 the global architecture of the cache is drawn, including buffers.

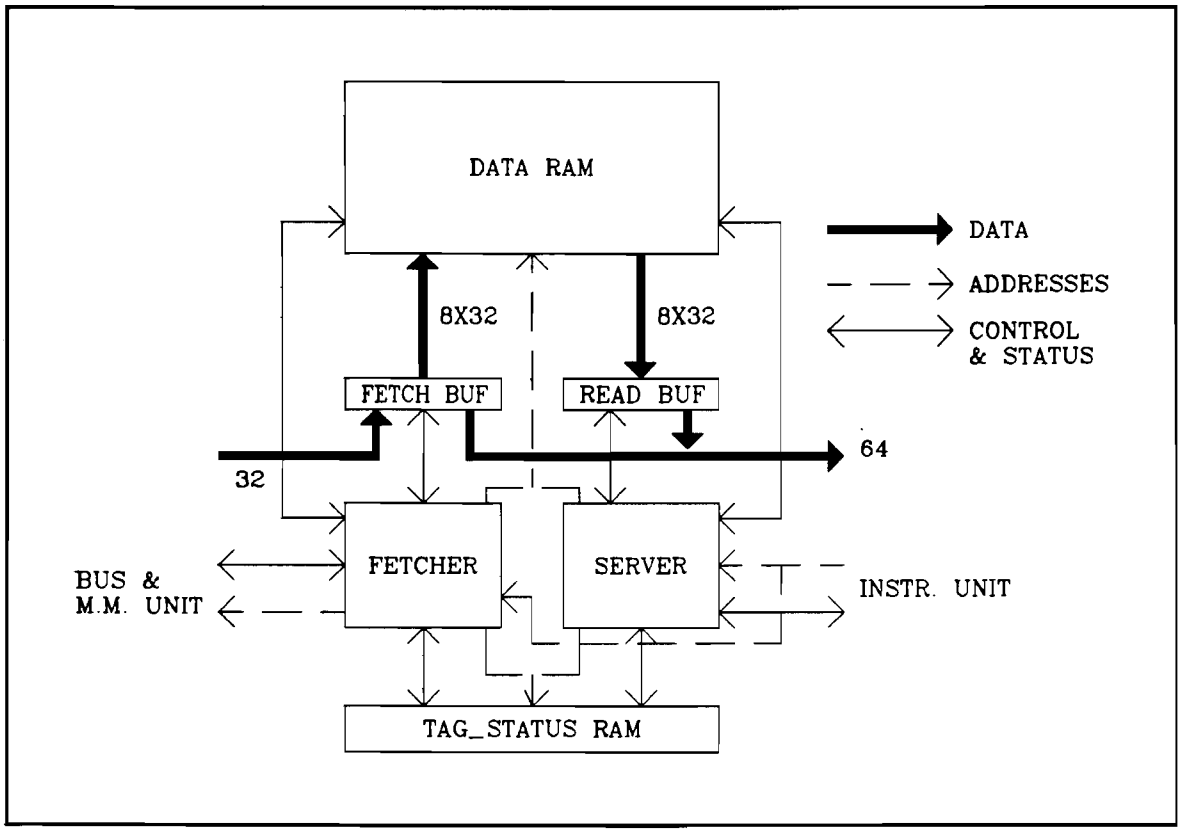


**Figure 2.** The global cache architecture.

## 2.6. Simulation results.

The trace driven simulations of J. Bormans [Bormans] show some interesting results. The cache model of the prototype is extracted from these results. But at the moment he ran his simulator, there was not and still is not a C compiler for the C-processor available. This is the reason he used estimated traces, generated by himself. So the results can be changed if real traces are used. The conclusions drawn from his simulations are written in Table 1.

**Table 1:** The conclusions of the trace driven simulation.

| | |
|---|---|
| Cache size | : 1024 quads. |
| Set size | : 2 sets. |
| Block size | : 32 quads. |
| Transfer Block size | : 8 quads. |
| Fetch algorithm | : prefetch hit. |
| Cache option | : stop fetcher, read buffer and fetch buffer. |

The parts of the C-processor which are of interest for the instruction cache are illustrated in Figure 3. At the main memory side the instruction cache is connected to the bus unit and at the processor side to the instruction unit. The memory management unit (MMU) is placed parallel to the cache.

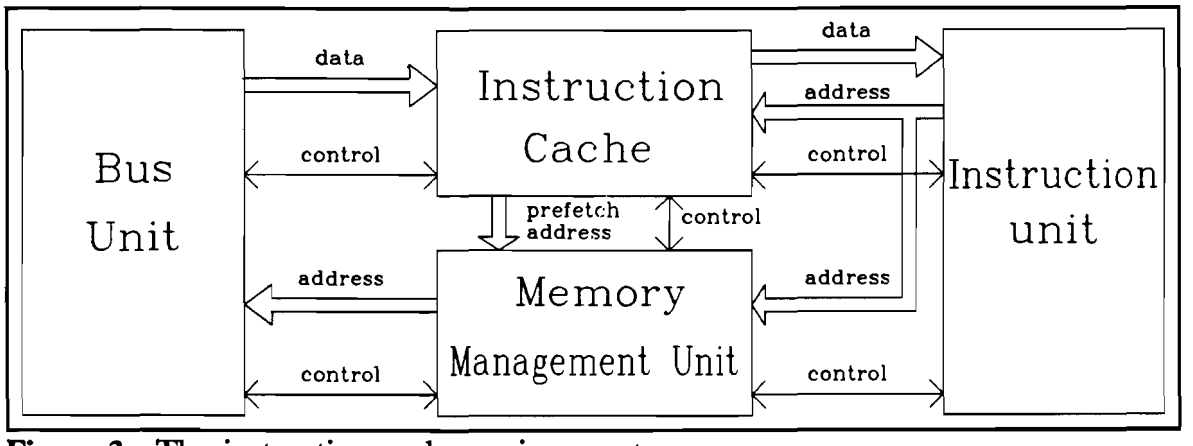


**Figure 3.** The instruction cache environment.

## 3.1. The bus unit.

The Bus Unit is the interface to the outside world. Besides requests from the instruction cache, it receives also requests from the data cache. The bus unit is able to serve these requests simultaneously. There are separate buses to the outside world for instructions

and data. (Harvard organisation.) The instruction bus is 32 bits wide and from now on this quantity of 32 bits will be called quads. All signals coming from or going to the outside world go through the bus unit for adaption of the signal levels. The bus unit takes also care of the external bus control.

### 3.1.1. The bus unit interface.

The interface between the bus unit and the instruction cache has to be a flexible protocol, because of the wide range of memory performance. The protocol has to be as fast as possible because the C-processor may be connected to an external cache. In this case memory request can be serviced in one clock cycle. In this situation a handshake protocol is used to fulfil the specifications. An example of this protocol is illustrated in Figure 4.

The protocol starts with making the valid signal active to indicate that the instruction cache needs a memory access and it puts the address on the address bus. To use the bus unit efficiently a count signal is added to the protocol to indicate how many quads the cache wants. After a while the bus unit responds by puting the instructions on the data bus and then enables the ready signal. Only an active ready signal means that the instruction on the data bus is valid. It is possible that a jump is detected by the instruction unit in the middle of the protocol and therefor a cancel signal is needed. An active cancel signal means that the protocol has to be terminated at the next clock cycle.
All data transfer and control signals will be synchronous but only the ready signal is asynchronous to speed up the protocol.

To ensure that the cache will not have a deadlock, the bus unit or the MMU will activate an error signal to indicate that it is not able to catch the requested quads. This can happen in case of a page fault (by the MMU) or a timeout event (by the bus unit) . To distinguish the two different cases a pagefault/error signal is added to the protocol.

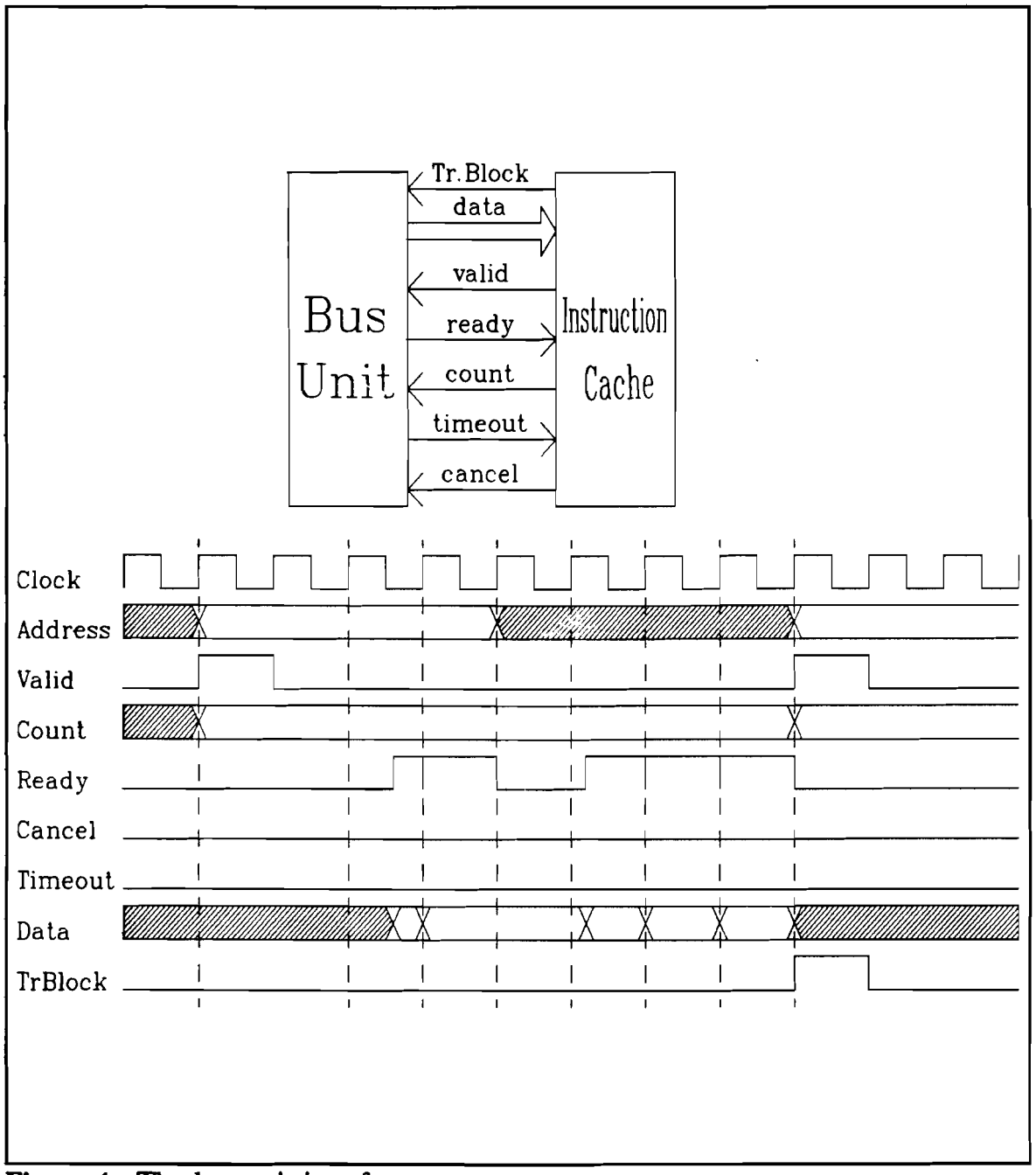


**Figure 4.** The bus unit interface.

If an error has taken place and the cache has to inform the processor of this error event then the status of the error event (pagefault or timeout event) will be put on the data bus.

If the static Data Ram of the cache is too slow writing the data from the fetchbuffer, the access time of the Ram is more than one clock cycle, then it can be solved by adding a Tr.Block acknowledge signal to the protocol. After the contents of the fetchbuffer is written in the data RAM, the Tr.Block acknowledge signal will be activated untill the next clock period.

## 3.2. The instruction unit.

A 64 bits data bus is used to transport the instructions between the cache and the instruction unit. The available bandwidth between the cache and the instruction unit is thus twice as large as the bandwidth between the main memory and the cache. The necessity of the 64 bits bus towards the instruction unit will be clear from the considering that many instructions will be longer than four bytes. The 32 bits connection with the outside world is imposed by pin constraints.

Instead of specifying the address of the 64 bits, the instruction unit specifies the address of the first of the two quads. The cache has to deliver the quads at the specified address and the quad at the next address. By allowing the 64 bits to be quad aligned in stead of 64 bits aligned, a more efficient data transport is possible between the cache and the instruction unit.

The number of clock cycles between the current and the next request for the cache depends on:

- the time required by the cache to deliver the two quads.
- the number of instructions contained in the double quads.

- the execution time of these instructions.

### 3.2.1. The instruction unit interface.

To serve the requests from the instruction unit without any necessary delay, this interface has to be as fast as possible. All instructions that are placed in the cache have to be transported to the instruction unit in one clock cycle. Therefor again a handshake protocol is used and some asynchronous behaviour is added to fulfil the speed specification. This protocol is shown in Figure 5.

This protocol starts by putting a valid address on the address bus and activating the request signal. After the cache delivered the first requested quad, it will activate the first ready signal. After finishing the second it will activate the second ready signal. When the instruction unit receives two ready signals and its ready to receive the requested quads then it activates the acknowledge signal. The cache can go to his rest state after the instruction unit activate the acknowledge signal. This sequence is the normal one, but it is possible that the instruction unit detects a jump instruction while it is waiting for instructions from the instruction cache. In this situation the instruction unit can stop the instruction request in the next cycle by activating the acknowledge signal, while the cache has not activated the ready signals.

As we have seen is it possible for the cache to receive an error signal from the bus unit or from the MMU. If the cache is prefetching, it can be ignored because the processor has not requested for these instructions. But if the cache is demand fetching, it is necessary to inform the processor of this error because the processor is expecting the requested instructions. To inform the processor there are some error signals added to the instruction unit interface.

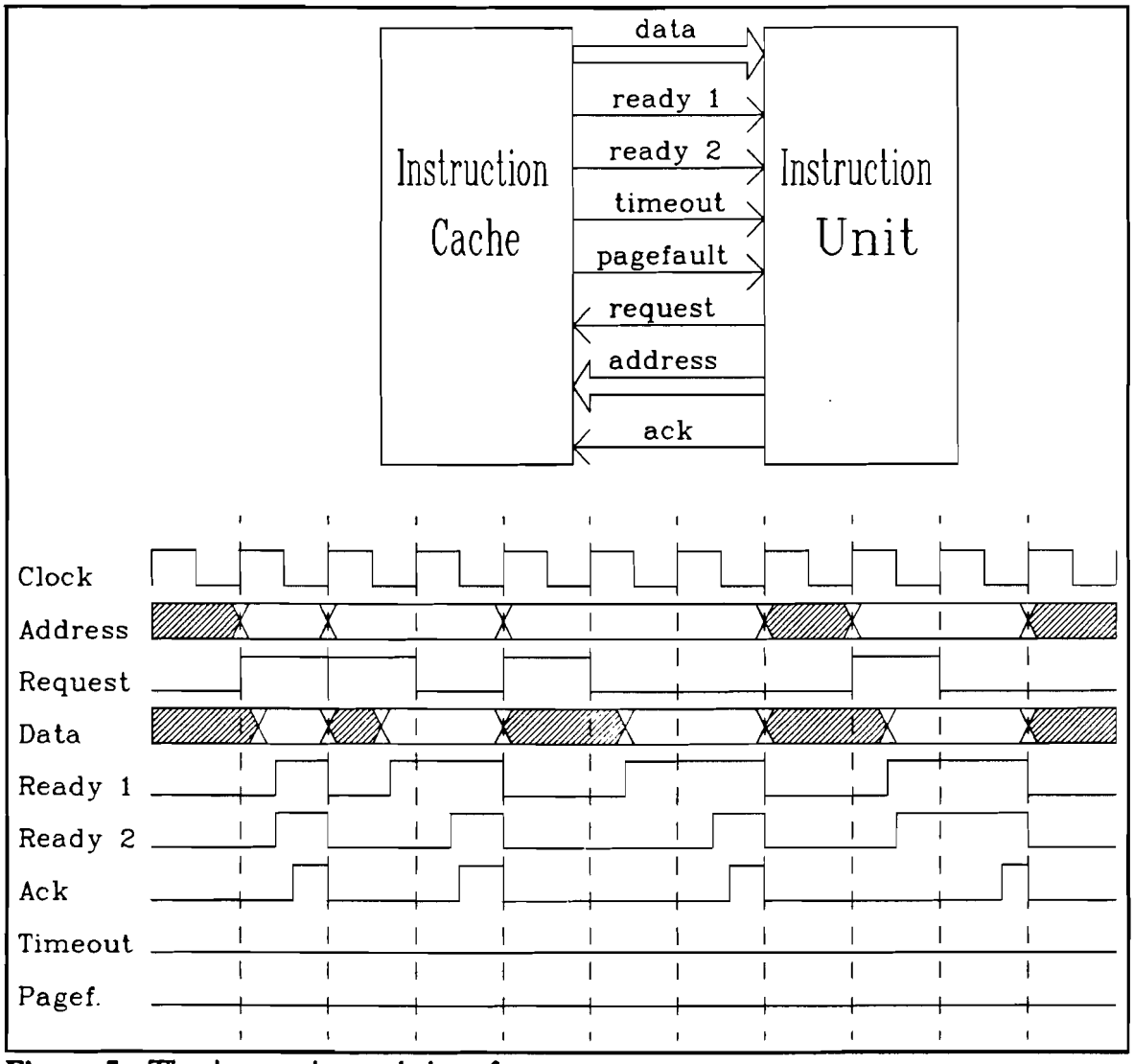


**Figure 5.** The instruction unit interface.

## 3.3. The memory management unit.

The C-processor is able to address 4 gigabyte. Since quads, instead of single bytes, are addressed, 30 bits are required for an address. However, the available amount of RAM will be less than needed for the complete address space. The major part of the information will be stored on magnetic storage media. This requires a memory management unit, which, among other things, has to translate the virtual addresses into real or physical addresses. To reduce the amount of cache flushes, normaly after each

process switch in a multitasking mode, a Proces Indentification Number (PIN code) is added to the address. Now a cache flush is only needed after all PIN codes are used and the new process will have the oldest unused PIN code. The PIN code has an arbitrary width of 16 bits thus, the memory management unit receives addresses of 46 bits.

The memory management unit is placed parallel of the instruction cache. The cache thus uses virtual addresses and is therefore called a virtual address cache. Most caches are real address caches. These caches have the disadvantage that for each cache reference the virtual address has to be translated into the real address. Although some parallelism is possible in this translation and the location of data in the cache, the cache access time is increased by this translation time. To reduce the cache access time, a virtual cache is used. An address translation is only required after a miss. The translation time can be shortened by translating the virtual address as soon as the cache receives a request. In this case the MMU is placed parallel of the cache.

If the memory management unit receives a request for a page which is not in the physical address space, it generates an error signal to indicate that the processor has to run an operating routine, e.g. to swap in a page from harddisk. Since this routine will use the cache itself, an error signal has to be activated. Otherwise the cache will wait indefinitely until a valid ready signal is returned by the bus unit and the operating routine never can be run.

As mentioned before a distinguish has to be made between an error during a prefetch action or during a demand fetch action. In case of a prefetch action the error signal is ignored and the cache stops the prefetch actions. In case of a demand fetch action the error signal has to be delivered to the instruction unit.

### 3.3.1. The memory management interface.

At this moment there is no interface protocol defined, because the MMU is a subject which still has to be studied. Only the prefetch addresses and status signals of the cache are implemented to inform the MMU in what mode the cache wants to fetch data. After the cache and the MMU receive a request from the instruction unit the cache will search for the requested data in itself and the MMU will translate the virtuel addresses into real addresses. If the data is in the cache and prefetching is not active than the MMU will not proceed, if the data is not in the cache than the MMU will deliver the translated address to the bus unit. When the data is found in the cache and the cache wants to prefetch then the MMU will skip the translation of the virtual addresses delivered by the instruction unit and it begins to translate the virtual prefetch address delivered by the instruction cache. After the translation it will deliver the real address to the bus unit.

The interface between the instruction cache and the MMU excists of a prefetch address, a pagefault signal and a DemandPre signal. The last mentioned signal has a width of 2 bits. This signal indicates the fetch mode of the cache. The following cases are possible (DemandPre1,DemandPre2):

- 00: The fetcher is non active, the MMU is non active.
- 10: The fetcher is demand fetching, the MMU has to translate the address delivered by the instruction unit.
- 01: The fetcher is prefetching, the MMU has to translate the prefetch address.
- 11: The fetcher is demand fetching, but the prefetcher is updating the cache status, the MMU has to translate the address delivered by the instruction unit.

By polling this signal the MMU knows exactly the state of the fetcher and knows exactly which address to translate. The meaning of the pagefault signal is ussumed to be clear. An example of this protocol is illustrated in Figure 6.

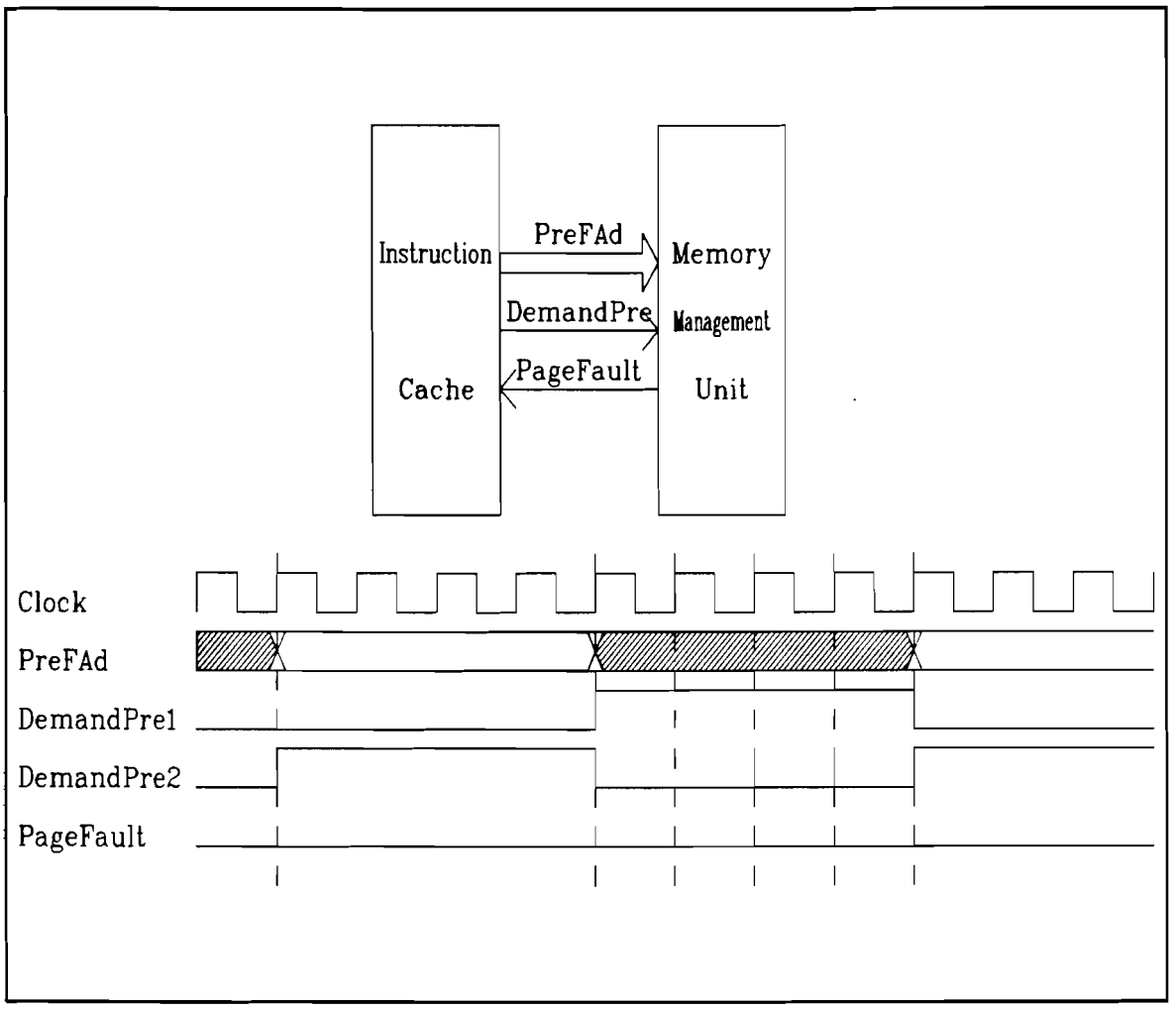


**Figure 6.** The memory management unit interface.

# 4.
# THE INSTRUCTION CACHE.

The second chapter contained an overview of the theory of instruction caches. Besides the theory, some practical specifications were defined for the instruction cache of the C-processor:

- the cache, if possible, has to deliver the quads, which are in the cache, in one clock cycle.
- the instruction unit requires a quad pair instead of a single quad.
- the MMU must function parallel with the cache.
- the cache can be in four different modes.
    * normal mode.
    * transparent mode.
    * cache flushing mode.
    * self test mode.

As illustrated in Figure 2 the cache can be split up in smaller modules. The more complex modules are the server and the fetcher and they are described in the next two chapters. The rest of the modules (memories and buffers) are described in this chapter. Table 1 is used to extract the dimensions of the memories and the buffers. As stated in table 1 is the block size 32 quads. Thus there are 32 blocks in the cache and there are 16 sets (see Figure 1). This means that the address is divided into a word field of 3 bits, a transfer block field of 2 bits, a set field of 4 bits and the remaining field is the tag field of 37 bits.

## 4.1. The decomposition of the cache.

As mentioned in the second chapter the cache has two sub modules, the server and the fetcher. The other modules are a read buffer, a fetch buffer, a data RAM and a tag/status RAM. Two other modules are added: First a status register, which contains the transfer block addresses of the transfer block in the read buffer and in the fetch buffer. Second a control unit, which collects the status signals from each unit in the cache and distributes them through the system. The schematic of the cache is illustrated in Figure 7. The used modules in this schematic are described in appendix B.

## 4.2. The cache RAM organisation.

In Figure 2 the tag/status RAM and data RAM are separated in the cache. This is an effective organisation for several reasons. Only one tag is required for all quads in a block. Some status information, like LRU bits, is also required per block. Other status information, like data_valid bits, is required per quad. If CAM is used it will be evident that the complete block must be stored on one RAM row. The quads have to be selected by means of the output decoders.

Traditional RAM could also organized this way. A RAM row is selected by the address decoder, using the set field of the address. (Assume a set size of 1, set associativity is discussed later in this paragraph). The tag, the status or one of the quads are selected by the output decoders. This is depicted in Figure 8a. It is of course not necessary to locate tag and status at the same location of the quads. One possibility is drawn in Figure 8b. This approach has many disadvantages:

- The address necessary to select tag/status information has a different format than the address of the quad. (Or tags should be stored in an even more inefficient way).

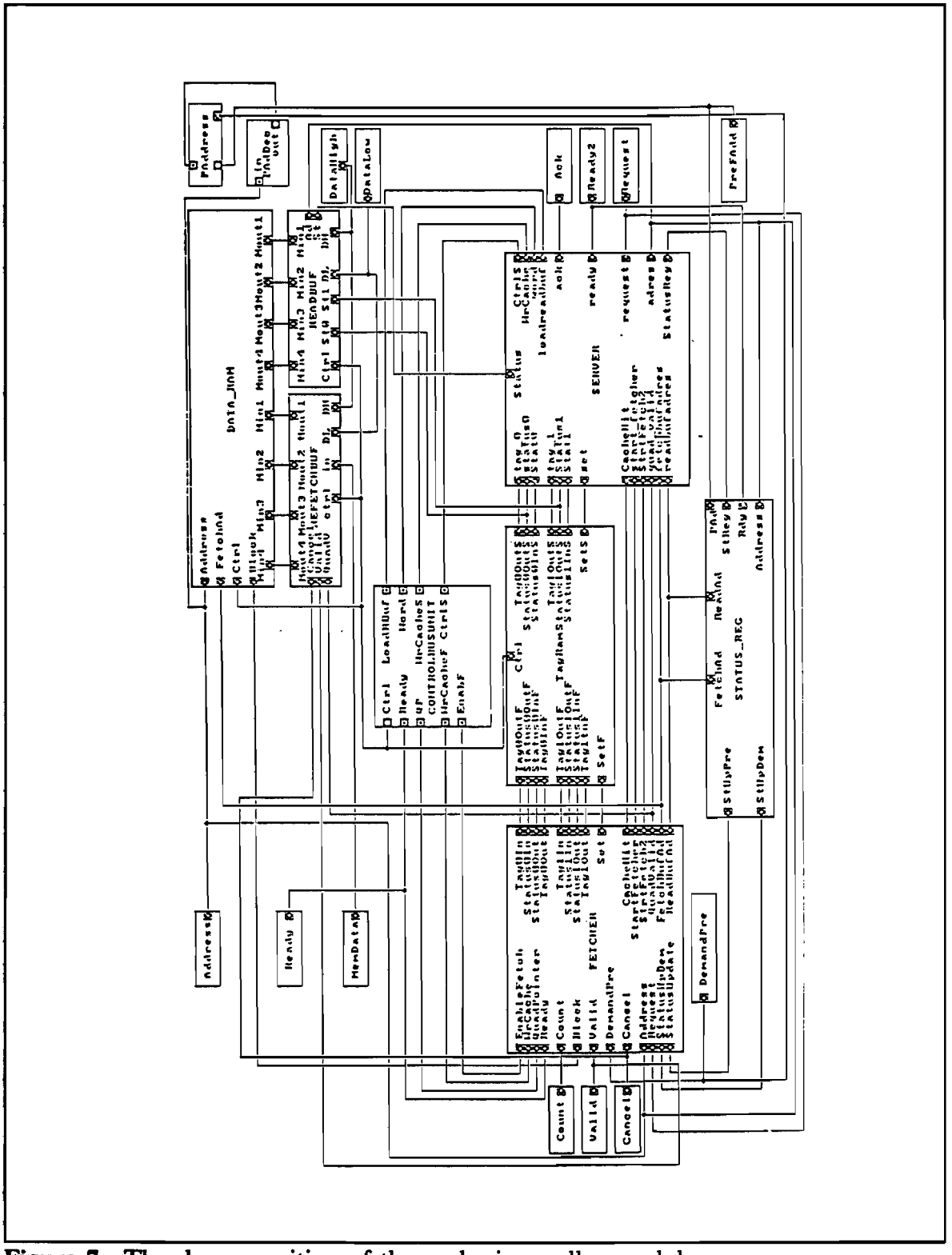

**Figure 7.** The decomposition of the cache in smaller modules.

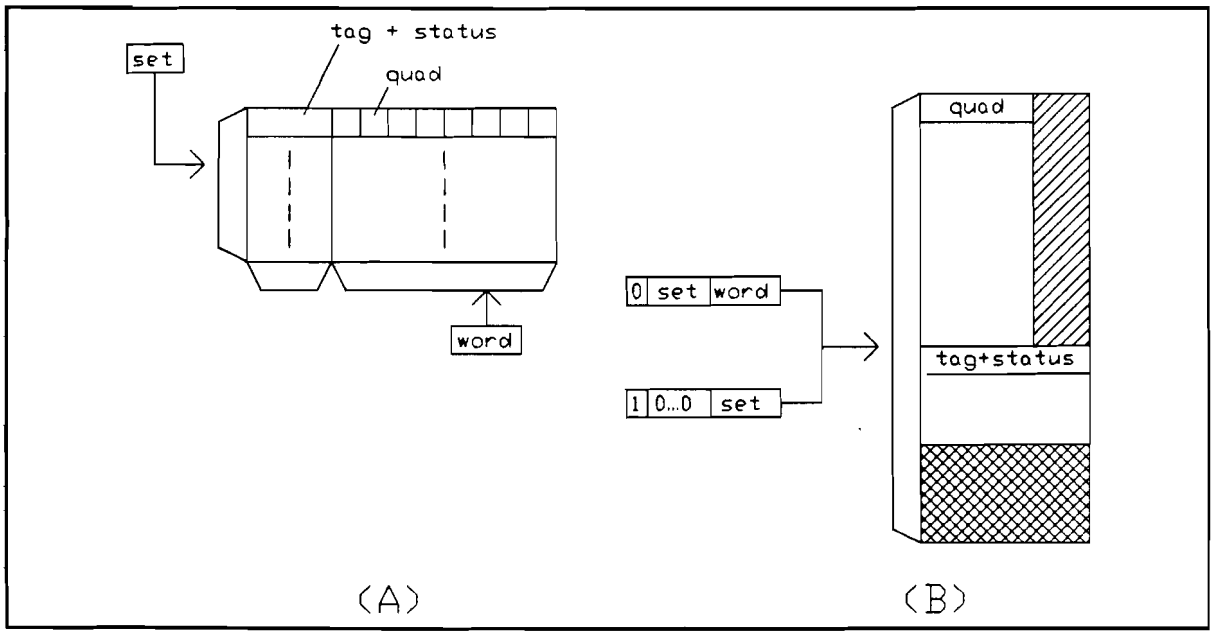


**Figure 8.** The data and status information combined in one RAM.

- Not all bits of the rows containing quads are used. (The length of tag plus status is longer than 32 bits). This is represented by the single shaded area.
- The address space is not used efficiently, since it is not a power of 2. This is represented by the double shaded area.

Therefor, it is better to use separated data RAM and tag/status RAM. This has also certain advantages over the first approach:

- No wide RAM is necessary.
- The RAMs are smaller and thus slightly faster.
- Simultaneous access to data (quads) and tag/status information is possible.

Two different modules were drawn in Figure 1 to illustrate a two way associative set. The tag and status information of the same set do indeed have to be accessible simultaneously. Different status RAMs could be used. But since these will be very small, it is better to use one status RAM, containing tag and status information of all the blocks of one set in one row. This is depicted in Figure 9 for a two way set associative mapped cache with LRU replacement and lookup prefetching.

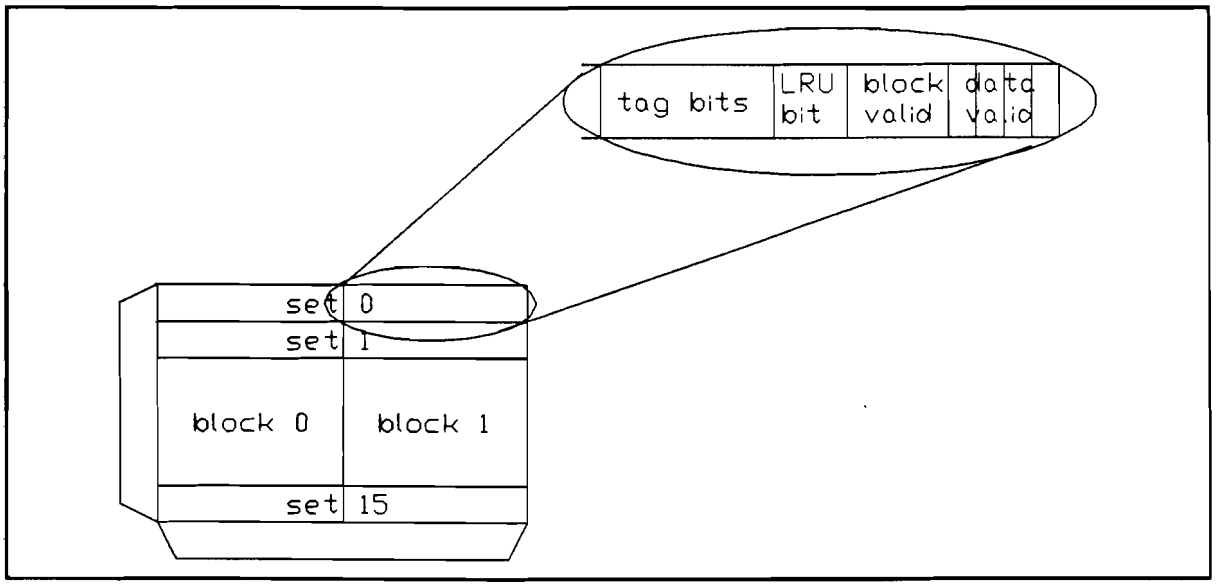


**Figure 9.** The status RAM of a two way set associative cache.

Quads of different blocks within a set do not have to be accessible at the same time. Thus they can be placed on different rows of one data RAM. The read and fetch buffer, introduced in chapter two, have a big influence on the cache performance. After all, their benefits are exploited best when complete transfer blocks can be copied to or from them. This requires a wide data RAM. (However not as wide as represented by Figure 8a, where tag, status and a complete block were stored on one row). The status RAM is not affected by the requirement that complete transfer blocks have to be stored on one row. An example of the data RAM of a cache with set size 2 is depicted in Figure 10. Blocks of 32 quads, divided over 4 transfer blocks, are assumed. The address which has to be supplied to the address decoder, consists successively of the set field, the block field and the transfer block field.

The memories in IDaSS have the restriction that they can not be wider as the maximum width of the busses. Normally is this maximum set to 64 bits, thus a memory wider than 64 bits is not possible in IDaSS. Because of this restriction, the status memory in the implementation of the prototype is divided in separate memories for the tag of the first block, the tag of the second block, the status of the first block and the status of the

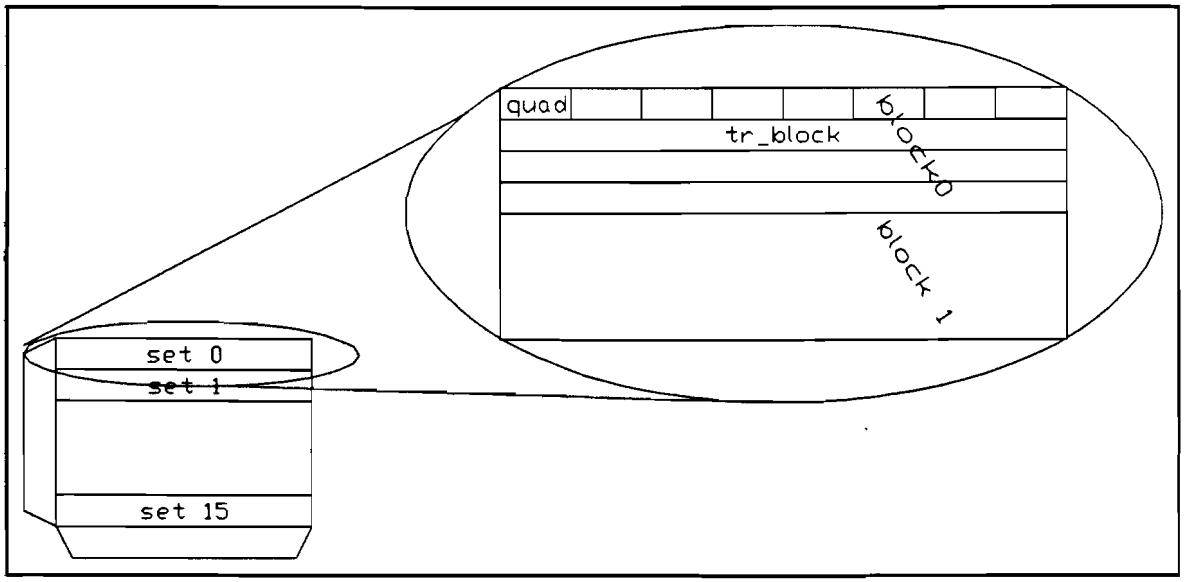


**Figure 10.** The data RAM of a two way set associative cache.

second block. They are respectively named TAG0, TAG1, STATUS0, STATUS1. The tag memories have a width of 37 bits and a depth of 16 words (number of sets), the status memories have a width of 34 bits ( 1 LRU bit, 1 Block_valid bit and 4 * 8 data_valid bits) and a depth of 16 words. Physically this decomposition of the memories is more realistic, because a memory of 144 bits wide and a depth of 16 words is rarely used. A memory, with a square layout, is more realistic. To address the memories the set field of the address is used as mentioned earlier in this chapter. Because the detection of the first quad, the detection of the second quad and the detection of the next transfer block are done simultaneously, the memories are implemented triple ported. This is realistic due of the small size of the used memories (the total size of the memories is, 16 * 17,75 = 284 bytes). The advantage of using triple ported memories is of course that no wait cycle has to be inserted for access by the fetcher or the server. In Figure 11 the implementation of the tag and status memories is illustrated.

The data RAM is, 8 * 32, 256 bits wide (the transfer block size) and a depth of, $2^4$ * $2^1$ * $2^2$, 128 words (this is the maximum range for the concatenation of the set, block and transfer block field as data memory address) as mentioned earlier in this paragraph. Sometimes the server and the fetcher want to have access to this RAM simultaneously.

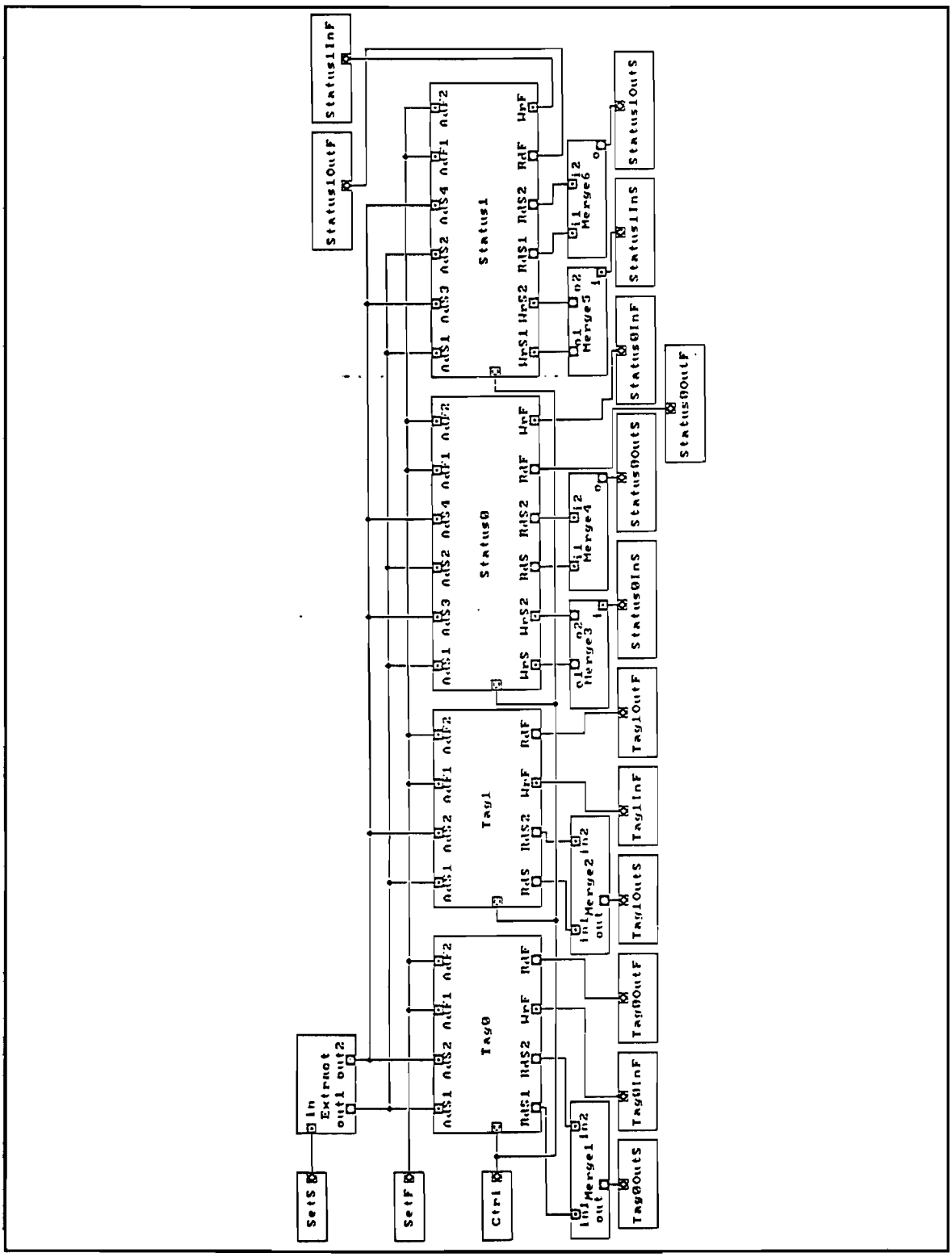


**Figure 11.** The tag and status memories implemented in IDaSS.

But this is impossible due of the single ported aspect of this memory. To implement this memory as dual ported or even triple ported is not very wise because of the large size of the memory. The size of the data RAM is , 1024 * 4, 4K Byte. Because the single ported character of the data RAM an arbitter is used to control the access to the RAM. If the server and the fetcher want to have access to the data RAM simultaneously, the server has always the highest priority to avoid unneccessary delay. The order of delivering the required quads are assumed to be clear, only the special case of cache memory hit of both quads, belonging to different transfer blocks, needs more explaining. Due the single ported data RAM both quads can not be read from this RAM. Therefor the first quad is copied in the read buffer, which cause a read buffer hit instead of cache memory hit. Then the last quad is delivered to the instruction unit.

The modules, which are used in the IDaSS implementation, are described in appendix C. For each module the name, the pin constrains, the functional description and the logical description are explained. The schematic of the IDaSS implementation is drawn in Figure 12.

## 4.3. The read buffer.

As mentioned in chapter two, the read buffer and the fetch buffer are used to avoid repeatingly access to the data RAM. The read buffer copies the transfer block with the required quads. The following quads do not require access to the data RAM, as long as they belong to the copied transfer block. Because not all quads may be fetched of a transfer block, due the "demand or prefetching stop" option, the status of the transferblock also has to be copied. Thus the read buffer is build of data registers, containing the quads, and status registers, containing the data_valid bits. Bypassing the quads from the cache memory is used to be able to deliver a quad from the cache memory and a quad from the read buffer. If this is not implemented then the above

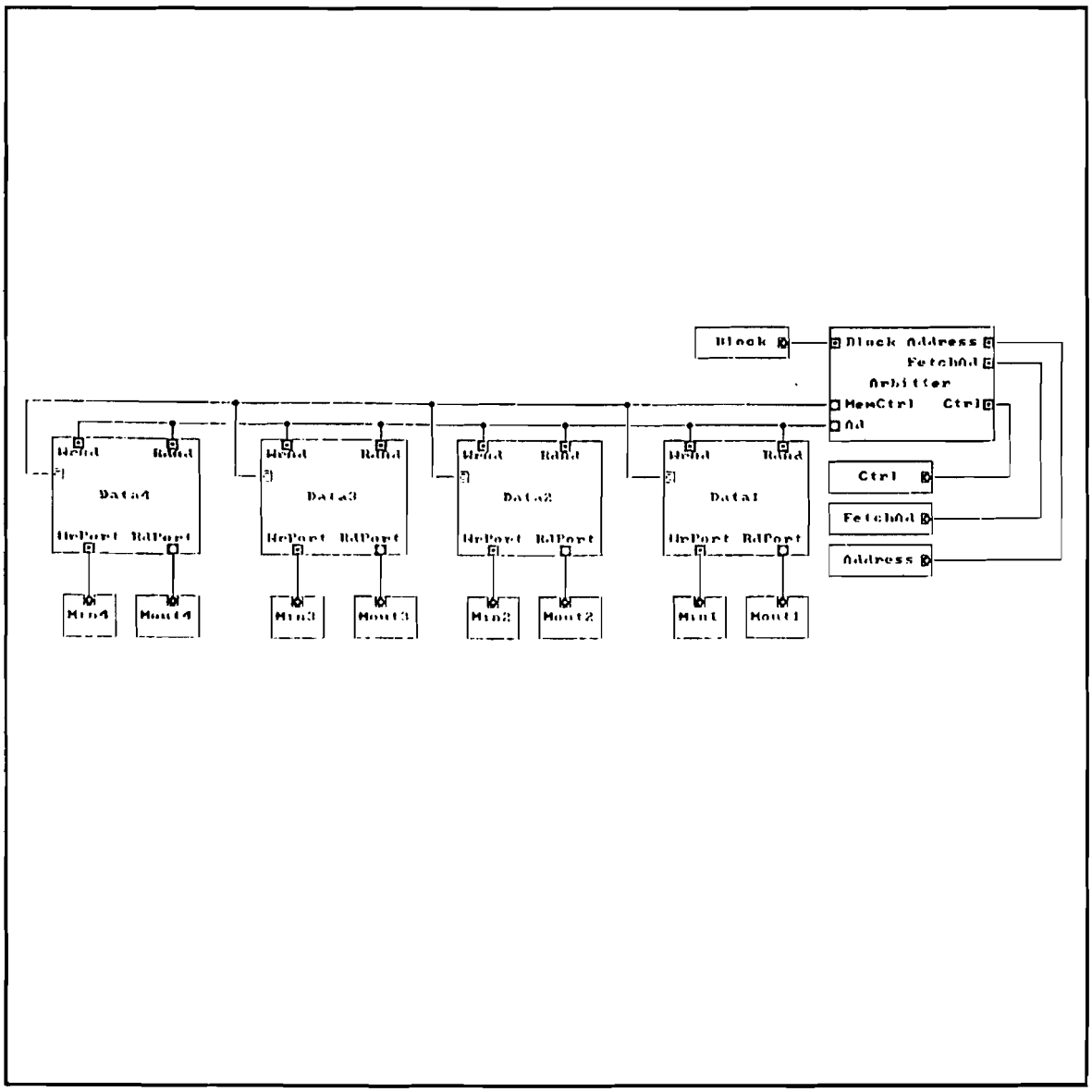


**Figure 12.** The IDaSS implementation of the data RAM.

mentioned case will cause problems due the old information in the read buffer. A required quad is placed in the read buffer and is overwritten by the transfer block, which is containing the older quad, at the next clock edge. This bypass feature is implemented by special registers, which contains a normal register and a multiplexer. In case of a read buffer hit the multiplexer selects the output of the register, thus the quad in the read buffer, and in case of a cache memory hit it selects the input of the register, thus the quad from the data RAM. The width of the read buffer is 8 quads as

mentioned earlier in the text. In Figure 13 the schematic of the read buffer is drawn and in Figure 14 the special data register is shown. The used modules of the read buffer are described in appendix D.

## 4.4. The fetch buffer.

Because of the same reason given with the read buffer, a fetchbuffer is used in this cache prototype. The fetch buffer is also build of data registers and status registers. Bypassing also is used from the bus unit to the instruction unit. Therefor two large multiplexers, 256 bits input and 32 bits output, are used to collect the two required quads. The multiplexer selects one of the eight quads in the buffer and pass it to the instruction unit. A control unit is implemented in the fetch buffer to generate the correct control signals to the multiplexers and the registers. This unit selects the quads via the multiplexers in case of a fetch bypass, it loads the data register with the appropriate quad and set the status register to the correct value. Because this unit is complete combinatorial, it is implemented as an operator in IDaSS. The schematic of the fetch buffer is drawn in Figure 15 and the description of the used modules in the implementation are described in appendix D.

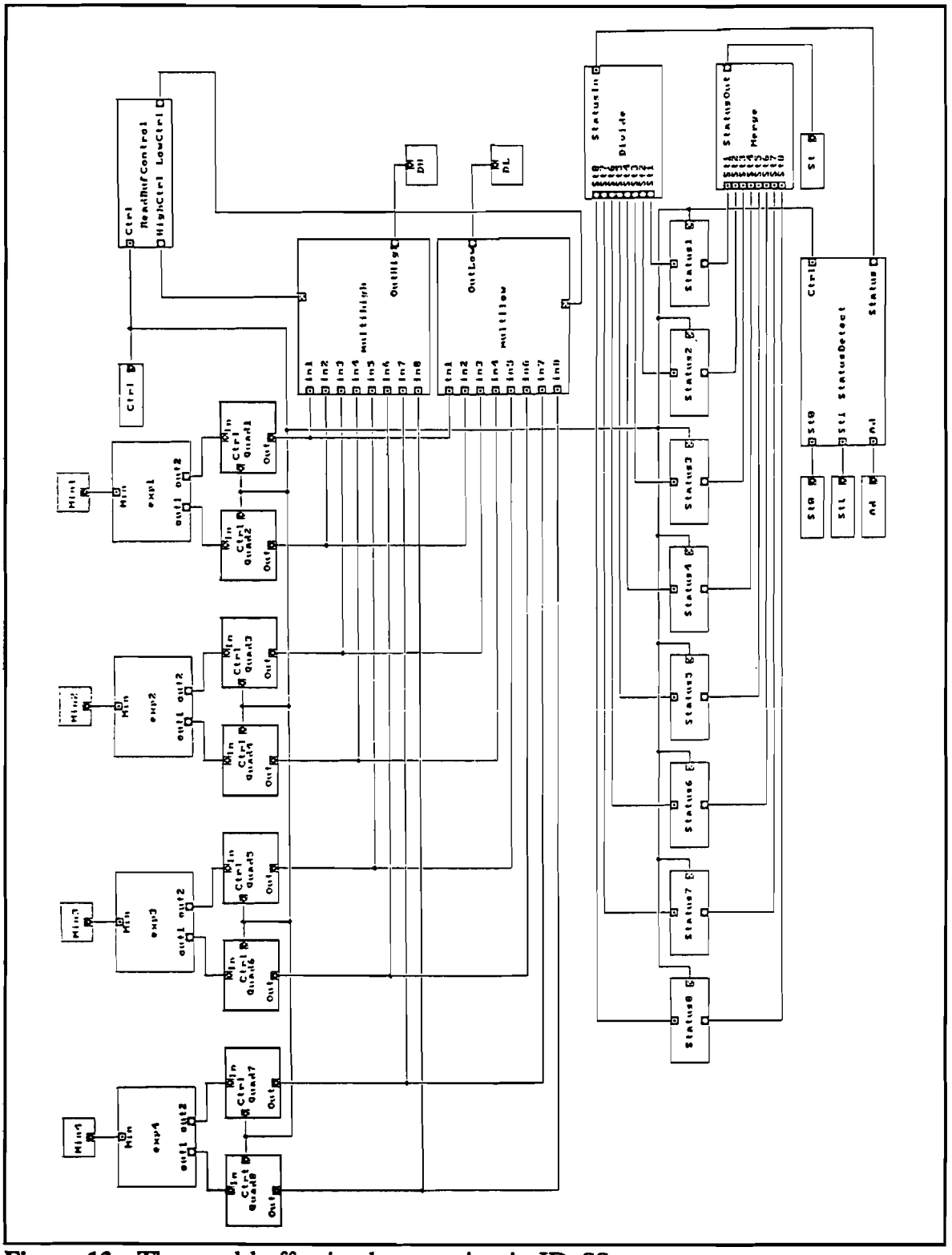

**Figure 13.** The read buffer implementation in IDaSS.

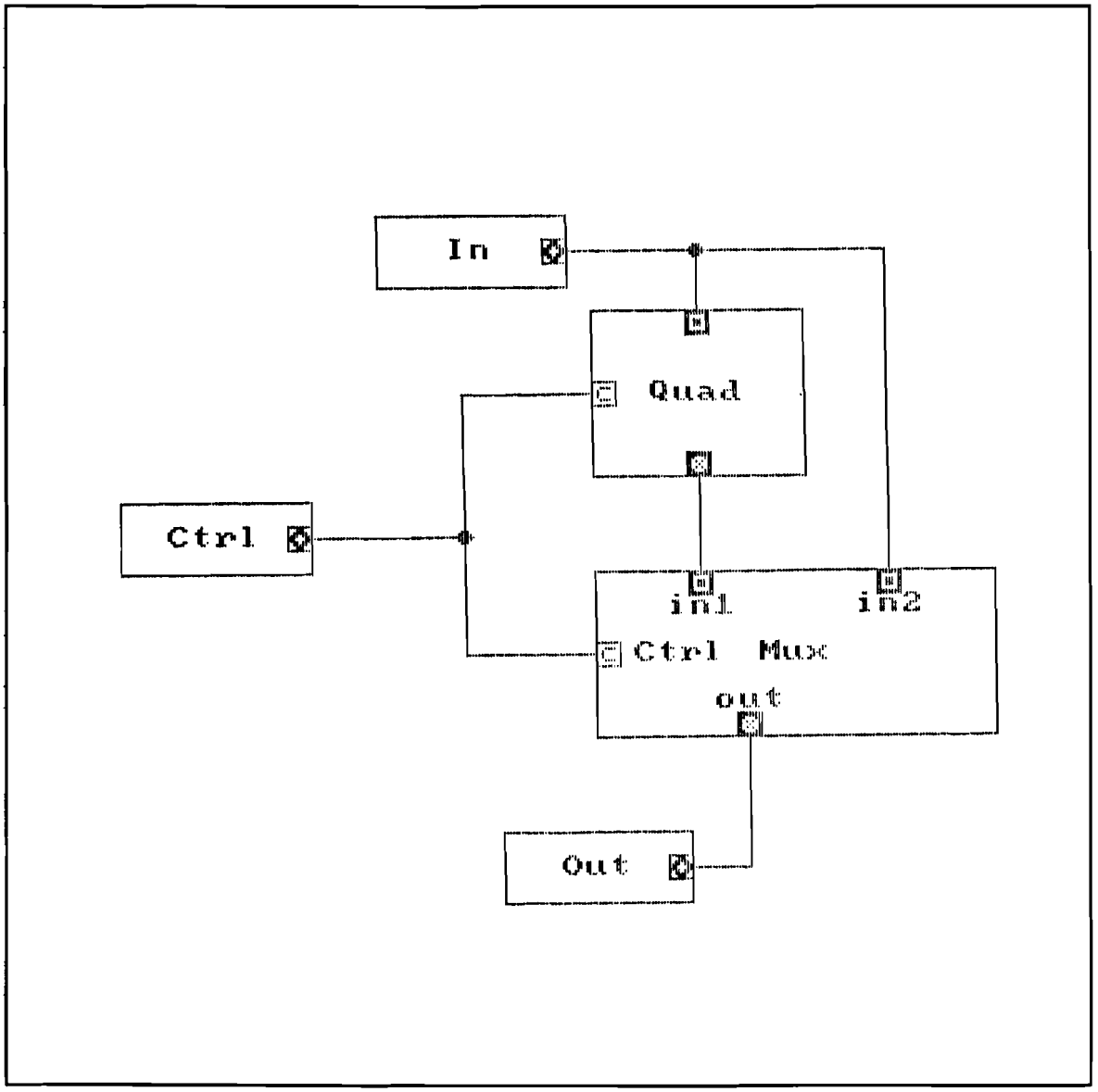


**Figure 14.** The implementation of the special data register.

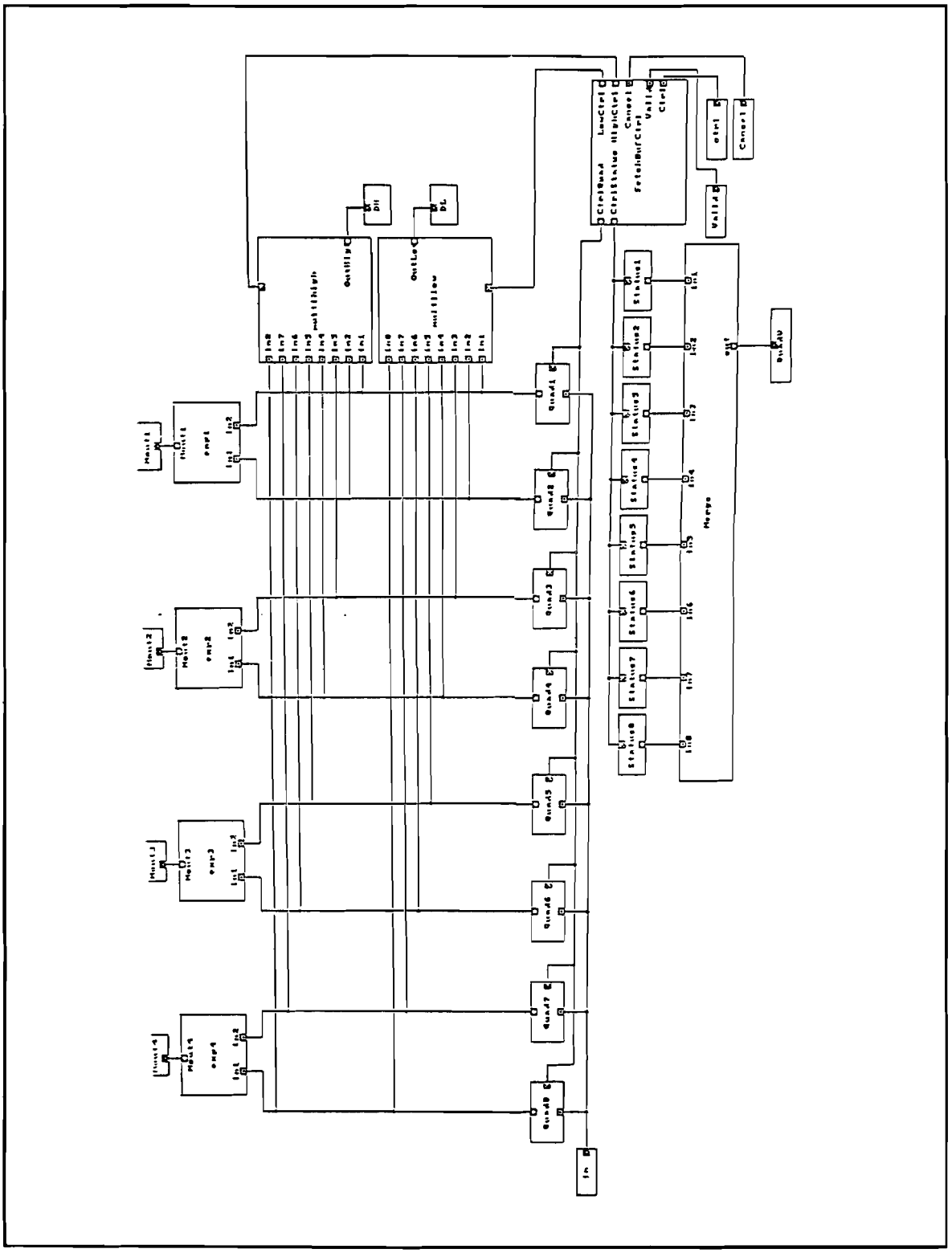

**Figure 15.** The fetch buffer implementation in IDaSS.

The server has to find the required quads and to take the appropriate actions to deliver those quads. The quads can be at three different places in the cache:

- in the fetch buffer.
- in the read buffer.
- in the cache RAM (see Figure 2).

Furthermore two different situations have to be distinguished. The first case is the situation where both quads are in the same transfer block. The second case is that the second requested quad is in a different transfer block. Due to the use of single ported data RAM it is not possible to read data from two different rows in the RAM. Last mentioned can slow down passing the second quad to the instruction unit.

## 5.1. The analysis.

All different states of the server can be illustrated as text but it is more clear to illustrate it in a figure. Figure 16 illustrates all modes and a detailed set of modes of the server is represented by Figure 17. The advantage of this method is the clear overview of the modes of the server. Also some timing can be illustrated in this kind of flow charts. In this chart some definitions are used.

*** Status signals:**

- READBUFHIT? : Set if the quad is in the read buffer.

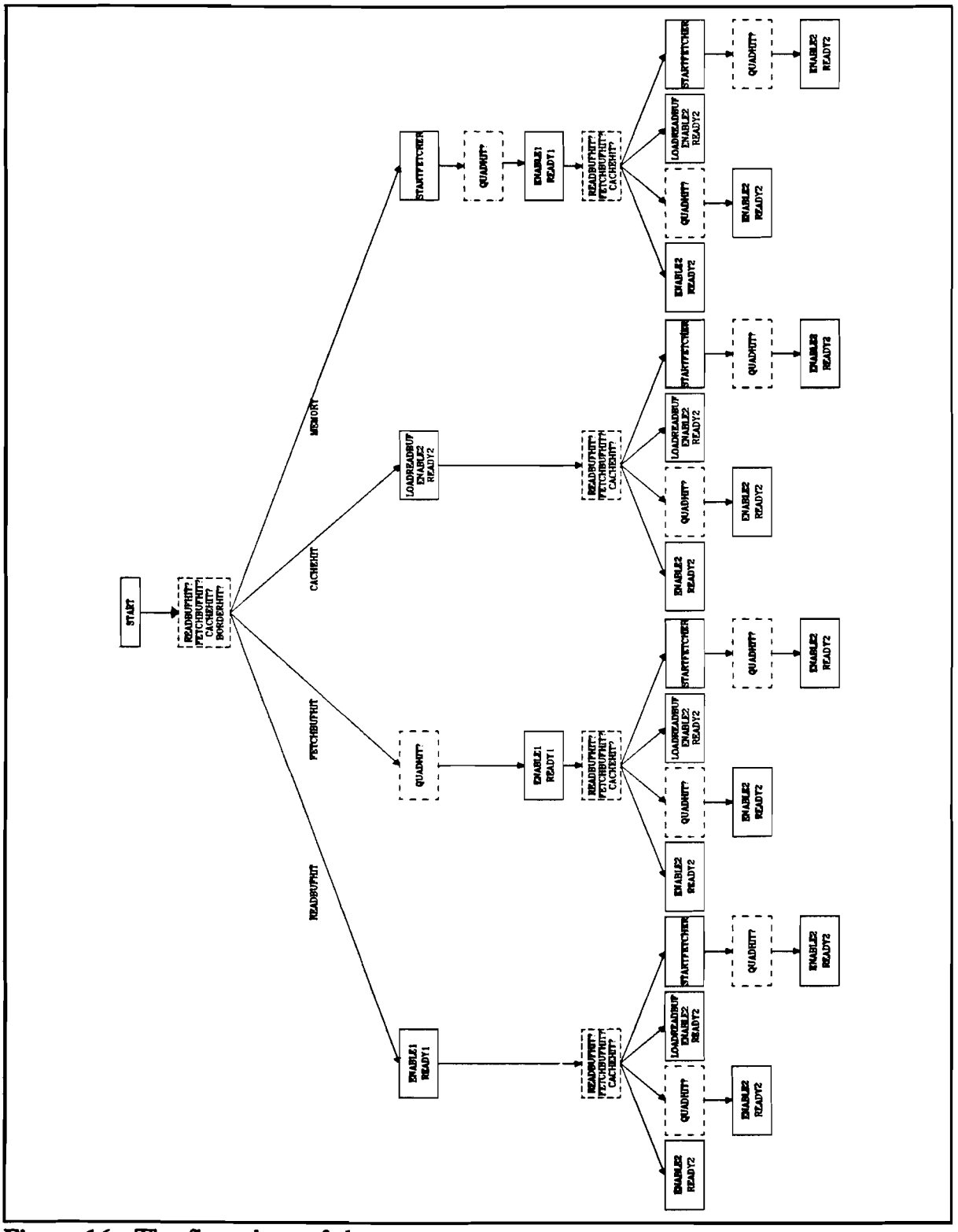


**Figure 16.** The flow chart of the server.

- FETCHBUFHIT? : Set if the transfer block, which could contain the quad, is in the fetch buffer.
- CACHEMEMHIT? : Set if the quad is in the cache memory.
- QUADHIT? : Set if the quad is in the fetch buffer.

*** Control signals:**

- ENABLE1 : Set if the first quad can be passed to the instruction unit.
- ENABLE2 : Set if the second quad can be passed to the instruction unit.
- READY1 : Set if the first quad can be passed to the instruction unit.
- READY2 : Set if the second quad can be passed to the instruction unit.

In Figure 16 the rectangles, where status signals have to be read and a decision has to be made, are drawn with dotted lines. The rectangles, where status signals have to be read and the server might have to wait (wait states), are drawn with dashed lines. The rectangles , where control signals have to be set, are drawn with solid lines. To complete the chart, the settings of the status signals have to be drawn in the figure. But the reason, why this is omitted in figure 16, is to have a clear drawing. But it is easily to extract the settings of the status signals from the place in the drawing. After a dotted rectangle four different possibilities exist.

- The first possibility is a read buffer hit.
- The second a fetch buffer hit.
- The third a cache memory hit.
- The last possibility is a main memory access.

This order is always the same after a dashed rectangle. The dashed dotted rectangle means a wait cycle if the signal, written in the rectangle, is not set. If the signal is set the flow continues.

The timing aspects can be described in clock cycles. If there is not a dashed rectangle from the top of the chart to the bottom, the time of the flow is one clock cycle. If there is a dashed rectangle the time of the flow can be more than one clock cycle. This

depends on when the signal in the dashed dotted rectangle has been set. In case the required quads are placed in different transfer blocks and both transfer block are placed in the cache memory, the time of the flow chart is two clock cycles due of the single ported data RAM.

As mentioned above, the first stage of processing the quad request is to detect the required quads. The read buffer, fetch buffer and the Cache Tag/Status Ram have to be checked on the presence of the quads. It is clear that comparators can be used to perform the above mentioned checking. The advantage is that they are combinatorial, thus fast response is possible. The results of the comparators is used to take further actions by the server.

The second stage of processing the quad request is to set the control signals, the following possibilities are possible, in case of a read buffer hit of the first quad. Figure 17 illustrates this part of figure 16.

1. The required quads are in the read buffer. In this case it is possible to enable the appropriate places in the read buffer in less then one clock cycle, and the ready signals, Ready1 and Ready2, are activated.
2. The first quad is in the read buffer, and the second is in the fetch buffer. In this case the server passes the first quad to the instruction unit, it sets the Ready1 signal, and it eventually has to wait for the activating of the signal, quadhit2. If this signal is set, the required second quad is available in the fetch buffer. If not, the fetch unit is fetching the second quad, and the server has to wait for it. If the Quadhit2 signal is set, the server passes the second quad to the instruction unit, and it sets the Ready2 signal.
3. The first quad is in the read buffer, and the second is in the cache memory. The required quads are passed to the instruction unit in less then one clock cycle and the Ready1 and Ready2 signals are activated.
4. The first quad is in the read buffer, and the second is outside the cache. The server passes the first quad in less then one clock cycle to the instruction unit, and it set

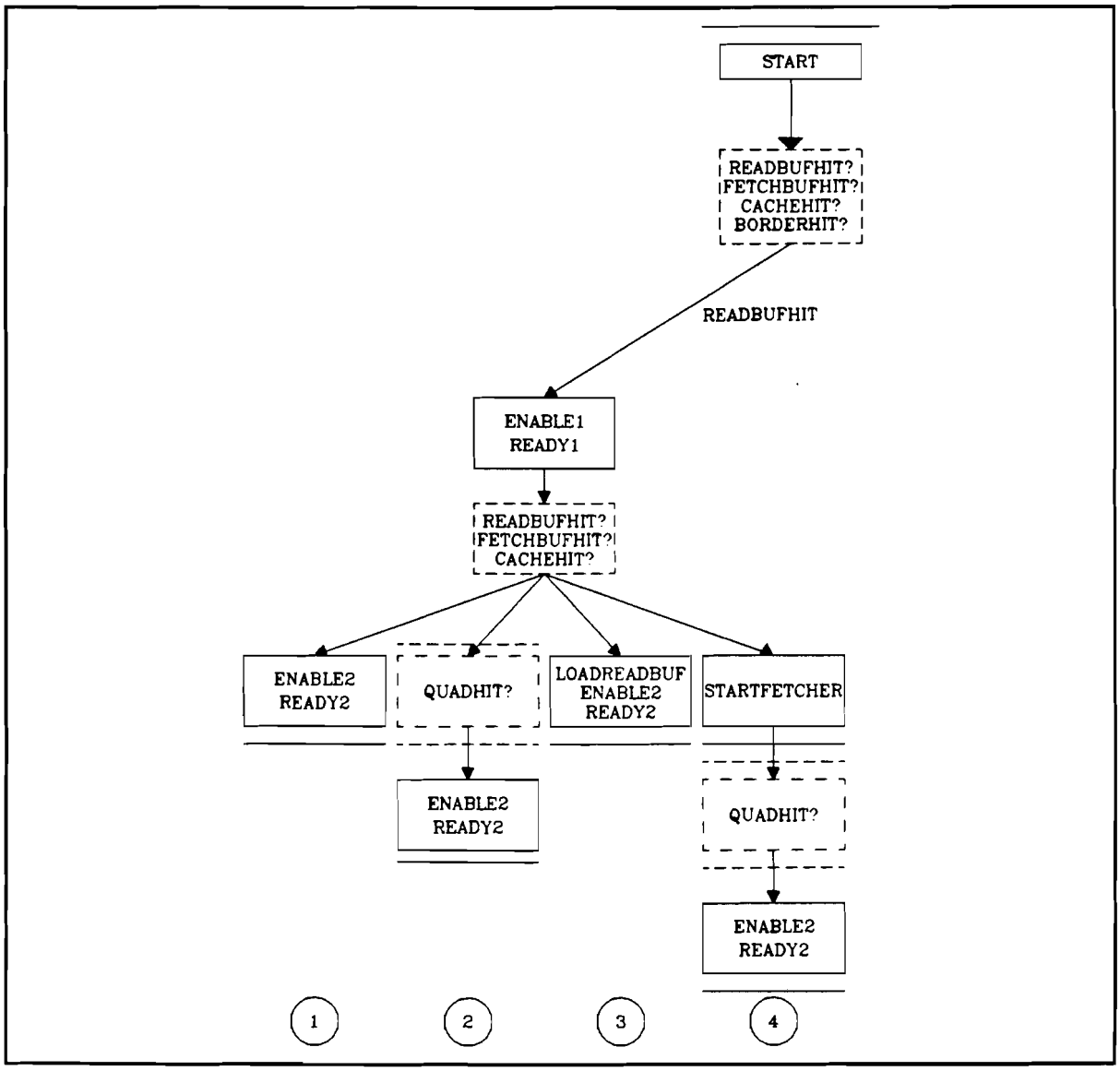


**Figure 17.** The set of modes in case of readbufhit of the first quad.

the Ready1 signal. The second quad is not present in the cache and therefor the server starts a demand fetch at the fetcher unit. After one or two clock edges the fetch buffer status is updated by the new transfer block address and the server regards from now on the situation as a fetch buffer hit.

In Figure 17 some horizontal solid and dashed lines are drawn to indicate the number of clock cycles, which the server needs to deliver the requested quads. The solid line

**Description:**

This state machine is implemented to indicate in which mode the fetcher is fetching. Two ways of fetching are possible:

- prefetching: fetching quads, which are not yet required by the instruction unit.
- demand fetching: fetching quads, which are required by the instruction unit.

The output signal is implemented as constant generator, named DemandOrPre.

**State Description:**

Rest:

DemandOrPre Setto:0;

<<

DemF3:

DemandOrPre Setto:3;

-> DemF

DemF:

DemandOrPre Setto:2;

<<

PreF:

DemandOrPre Setto:1;

<<

**Control Connector Specification:**

"The specification of the control connector input:
Request,StartFetcher,TrBlockHit,Ready,QuadPointer,State,CacheHit."

%x0xx1001010 Goto:Rest.

%x0xx1001100 Goto:Rest.

%x1xxxxxx00x Goto:DemF.

%x1xxxxxx10x Goto:DemF.

%x1xx100101x Goto:DemF.

%x1xx0xxx01x Goto:DemF3."The prefetcher is stoppped by an active StartFetcher signal."

%x1xx100001x Goto:DemF3."The prefetcher is stoppped by an active StartFetcher signal."

%100x000000x Goto:PreF.

## F.10 The demand and prefetching indication.

**Module:** a constant generator.

**Name:** DemandOrPre.

**Pin Constraints:**

Input: none.

Output: no name.

**Description:**

This constant generator is used to generate the output signals of the state controller, named FController.

is equal to one clock cycle, the dashed line is equal to zero or more clock cycles depending when the signal quadhit? is set.

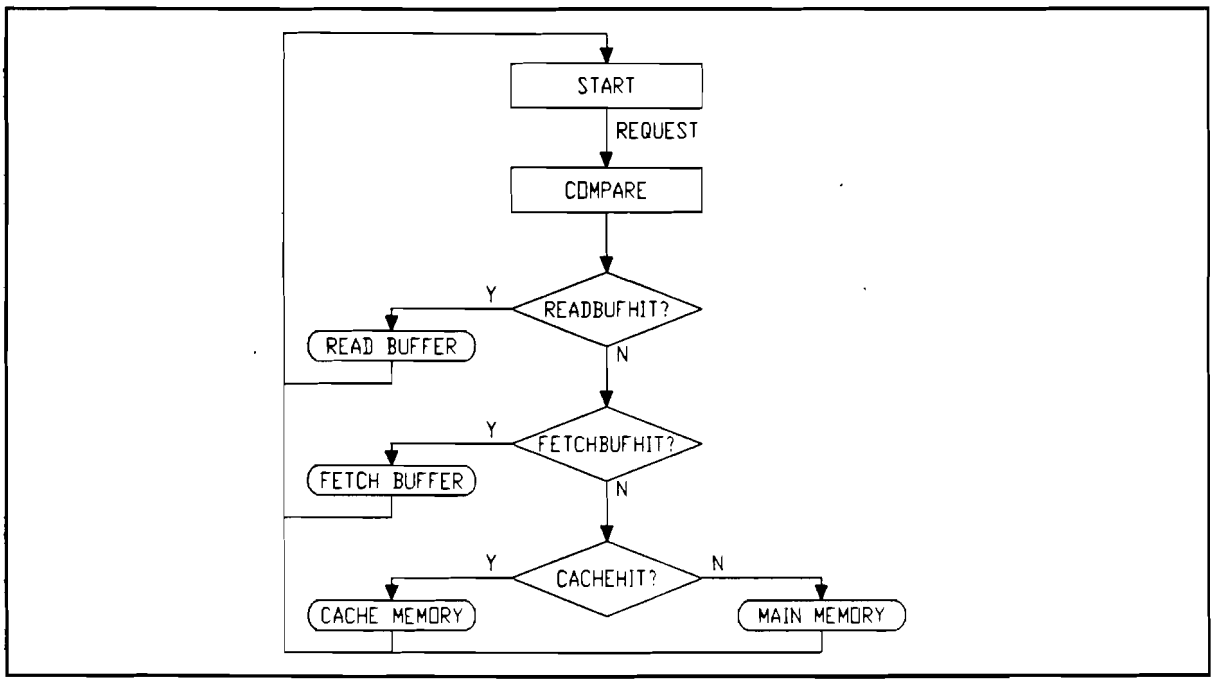


**Figure 18.** The ASM chart of the server.

From this flow chart an ASM chart is extracted and this ASM chart is drawn in Figure 18. Only the situation of a read buffer hit of the first quad is specified in detail (see Figure 19).

An alternative method to illustrate the modes of the server is an old IBM method. The basis is to draw a vertical line for each state and the state transitions can be presented by arrows drawn from the present state to the next state. An example of this method is illustrated in Figure 20.

A disadvantage with this method is that the figure is very long, but the state transitions are very clear. A reason why this method is not used is the miss of timing aspects in this method.

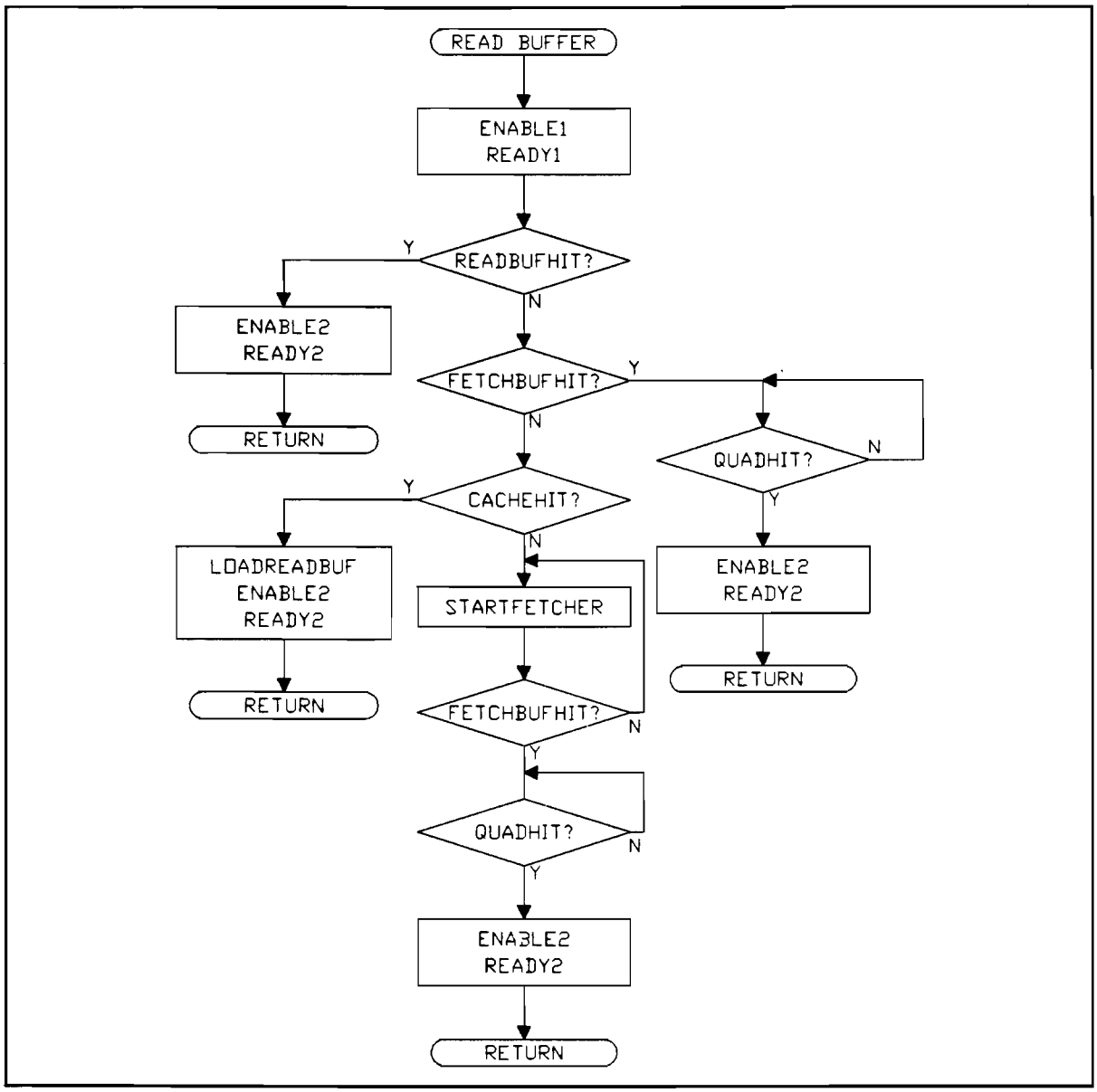


**Figure 19.** The ASM chart in case of a read buffer hit of the first quad.

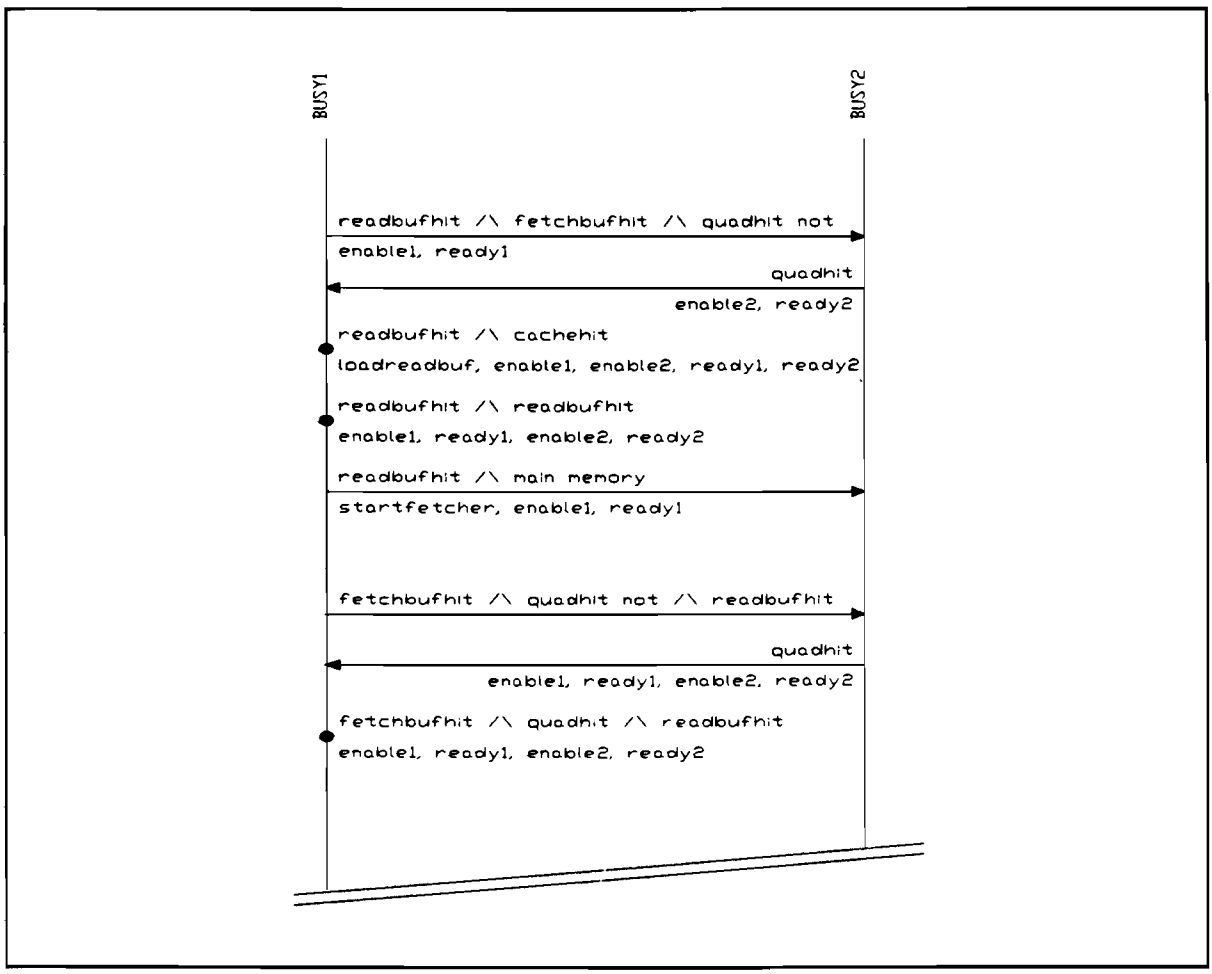


**Figure 20.** The modes of the server illustrated by an old IBM method.

## 5.2. The implementation.

The server is divided in three parts, the comparators and two combinatorial logic blocks. The comparators has to detect the vallidity of the required quads, the first combinatorial logic block has to set the control signals for the first quad, and the second combinatorial logic block has to set the control signals for the second quad. Besides the control signals mentioned in the previous paragraph some other control signals are added to update the

status register and to load the read buffer in case of a cachehit. The names of these control signals are:

- LOADREADBUF : set, if the read buffer has to load the data of the data RAM.
- LOADREADBUFAD : set, if the read buffer address of the first quad has to be written in the status register.
- LOADINCREADBUFAD: set, if the read buffer address of the second quad has to be written in the status register.

The module, which contains the comparators, is called "comparators" in the remainder of the text. And the module, which contains the combinatorial logic for the first quad, is called "comb1" and the other is called "comb2". The "comparators" is divided into three parts, "Quadfind1" which detects the presence of the first quad, "Quadfind2" which detects the presence of the second quad and the module "LRU" which updates the LRU bits of the cache blocks. To address the "tag/status" RAM and the "data" RAM the set field of the address and the block, where the required quad is found, are needed. These signals are easily to extract from the comparators, because the detection of the quads is done by these comparators. The signal names added are:

- SET1 : Indicates the set field of the lower quad address.
- SET2 : Indicates the set field of the higher quad address.
- BLOCK0 : Set, if the lower quad uses cache block1.
- BLOCK1 : Set, if the higher quad uses cache block1.

To make a distinction between status signals belonging to the first quad and the status signals belonging to the second quad, a number (1 or 2) is apended to the name of the status signals as described in the previous chapter, a "1" for the first quad and a "2" for the second quad, thus "readbufhit" of the second quad becomes "readbufhit2".

The outputs of the modules "Quadfind1" and "Quadfind2" are all the time available, as are the outputs of the LRU module. This means that no control is needed for the module "comparators". Because the modules "comb1" and "comb2" are combinatorial

logic, they are simulated in IDaSS by "operators". The two mentioned modules are described in a special language, defined by IDaSS, as functions to set the control signals as defined in the previous paragraph. The functions of the operators "comb1" and "comb2" represents the detection of the required quads. The names of the functions are obvious to understand. The control signals depend on the outputs of the module "comparators", named status signals. The control of these two blocks are defined with a control connector, which can be specified like a PLA, to set a specific function. Thus the status signals set the control signals to the appropriate settings.

Besides the mentioned three parts some extra modules are added to the implementation of the server. These modules merge and extract the busses due to the restriction in IDaSS to have only one control connector available on an operator. These operators are named "signalmerge" and "signalextract". The functions of these operators are defined by default, and thus they are always active. Physically this operators do not exist, because they have no logical meaning.

In case of a cache hit of the first and the second quad, there may arise some problems due of the single ported data RAM. If the quads belong to the same transfer block, there is no problem. But if the quads belong to different transfer blocks, it is impossible to deliver those quads in one clock cycle. A solution for this problem is to copy the first transfer block into the read buffer and deliver the first quad to the instruction unit and during the next clock cycle deliver the next quad. Therefor a distinction has to be made between the first clock cycle and the next ones. A constant generator, named "next", is used as indication in which clock cycle the server is functioning. A Moore machine is used to control the "next" constant generator. Because IDaSS has the restriction that only clocked buses can be used as input for the state controllers, it is impossible to set the "next" generator in time. A solution to that problem is to use a control connector, which forces the state controller to a specified state. Every clock edge the state controller evaluates the control connector, and if nothing is specified it evaluates its state description.

In case the first quad is in the read buffer and the next quad is in the cache memory, it is impossible to copy the transfer block, which contains the last quad, into the read buffer. In this case the old contents of the read buffer is lost. If the instruction unit sets the acknowledge signal in the same clock cycle , there is no problem. But when the acknowledge signal is not set in the same clock cycle the above described problem occurs. To solve this problem the status register updates after the acknowledge signal is set by the instruction unit.

A description of the used operators and state controllers are described in appendix E. Of each module the name, pin constraints, the functional description of the module and the logical description are described. The schematic of the server is drawn in Figure 21.

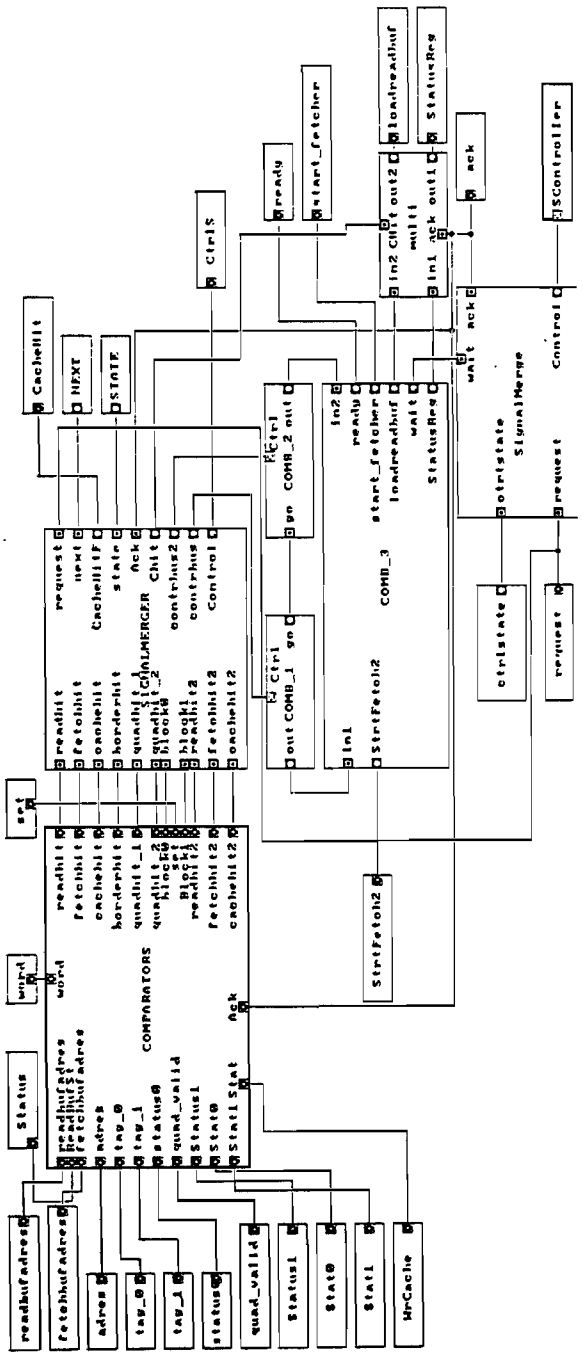


**Figure 21.** The server implemented in IDaSS.

The task of the fetcher is to fetch the quads from the main memory. As mentioned in chapter two there are two ways to fetch, demand and prefetching. The handshake interface protocol between the fetcher and the bus unit is already defined in chapter three. The used prefetch algorithm is a prefetch_lookup_on_hits. This means that only in the case of a cache hit the next transfer block will be prefetched, if and only if it is not already in the cache. To start a prefetch only in the case of a cache hit, has the advantage that the number of cache accesses is minimized (see paragraph 2.3). The replacement algorithm has to decide which block in the set will be removed to store a main memory block. The used replacement algorithm is the Least Recently Used algorithm. The block, that is the least recently used, is replaced by the new block (see paragraph 2.4).

## 6.1. The analysis.

The prefetch algorithm includes some comparators, which are used to check the existence of the next transfer block in the cache. Thus, the fetcher is divided into three parts:

- comparators.
- prefetch module.
- demand fetch module.

The last two mentioned modules have to fetch the data from the main memory. Due to the synchronous actions of the fetch algorithm and the replacement algorithm, they are build with Moore machines.

Only wrap around is still not discussed. The wrap around feature is based on the idee that a jump may occur into the middle of the transfer block. In this case it is wise to fetch the required quad first and not the first quad of the transfer block. The following quads are those which have a higher address in the same transfer block. As mentioned in chapter two the lower quads will be skipped.

In case of a jump the fetcher may be fetching unnecessary data, and in this case the fetcher has to be stopped. The question is "when is the data unnecessary?". In case of prefetching, the instruction unit did not require those quads yet. Therefor the prefetch action must not continue, when a jump occurs. In case of demand fetching, obviously the instruction unit is required for other data, which is not being fetched at this moment. Therefor the demand fetch action also has to stop immediately. To avoid that the data, which is already fetched, is lost, the contents of the fetch buffer have to be copied into the data memory. This stopping feature is called "stop prefetch" or "stop demand fetch". Because the fetch operation may be stopped in the middle of its sequence, status bits have to be used to indicate which quads of the transfer block are valid. The similarity between the earlier mentioned quad_valid bits and these data_valid bits is assumed to be clear.

The sequence of the fetching operation can be divided into two parts. The first part is an initialization part. The initialization part initializes the interface protocol between the fetcher and the bus unit. The second part is the following up part. This part only takes care of guiding the quads to the correct place in the fetch buffer. And when the last quad is fetched then the cache status is updated by the fetcher. A register is used for the guiding of the quads to the correct place in the buffer. The content of this register is used to address the data registers of the fetch buffer. This register is called quadpointer in the remainder of the text. The replacement and fetch algorithm are not

very complicated and therefor they are converted to ASM charts. The ASM chart of the prefetcher and the demand fetcher are illustrated in figure 22, 23 and 24.

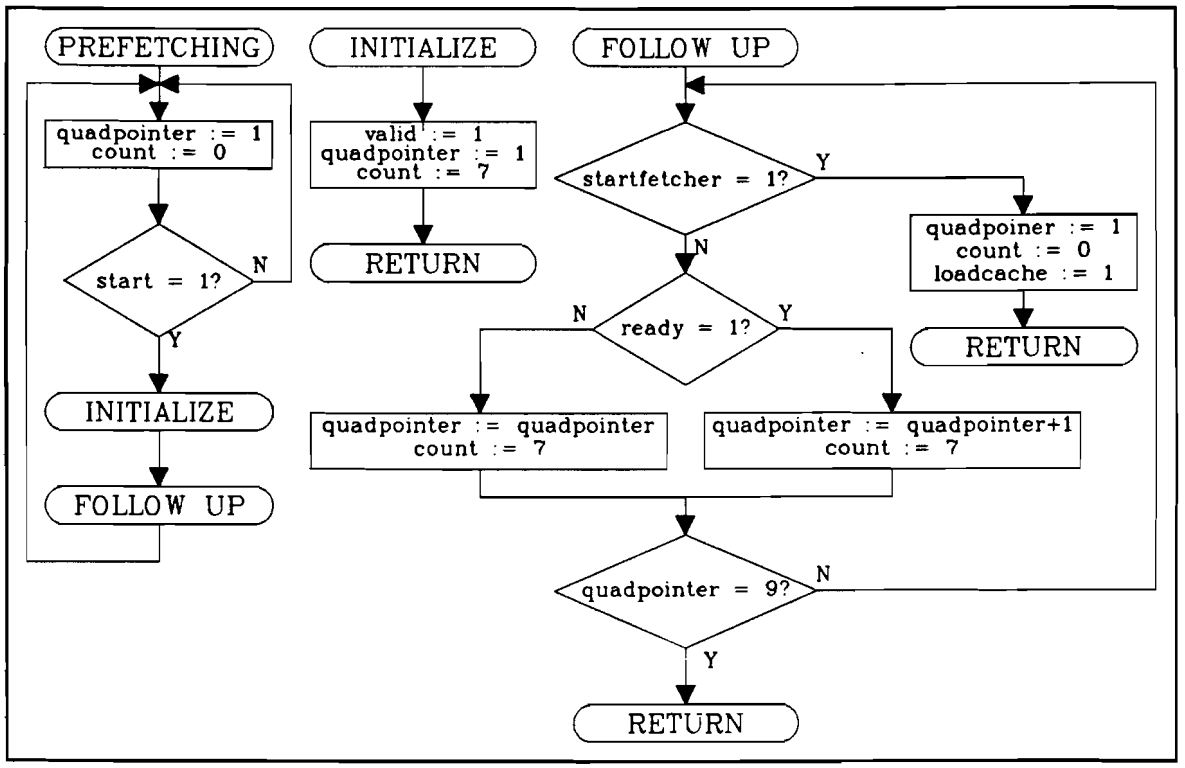


**Figure 22.** The ASM chart of the prefetcher.

The prefetcher state controller is initialized by an active start signal. The start signal is set, if the server did not started the fetcher (the used signal is named startfetcher), the instruction unit required for data (request), and the next transfer block (trblock) is not present in the cache. The initialization of the protocol is done by setting the valid and count signal. The valid signal indicates the validity of the delivered address, the count signal represents the number of required quads by the fetcher. Of course the quadpointer points to the first quad place of the fetch buffer. Immediately after the initialization the startfetcher and the ready signal are polled. If the startfetcher signal is set, the prefetch actions have to be stopped ("stop prefetch" option) and the contents of the fetch buffer are written to the data RAM. If the bus unit delivers a quad to the cache, the bus unit sets the ready signal to indicate the presence of a valid quad. If the

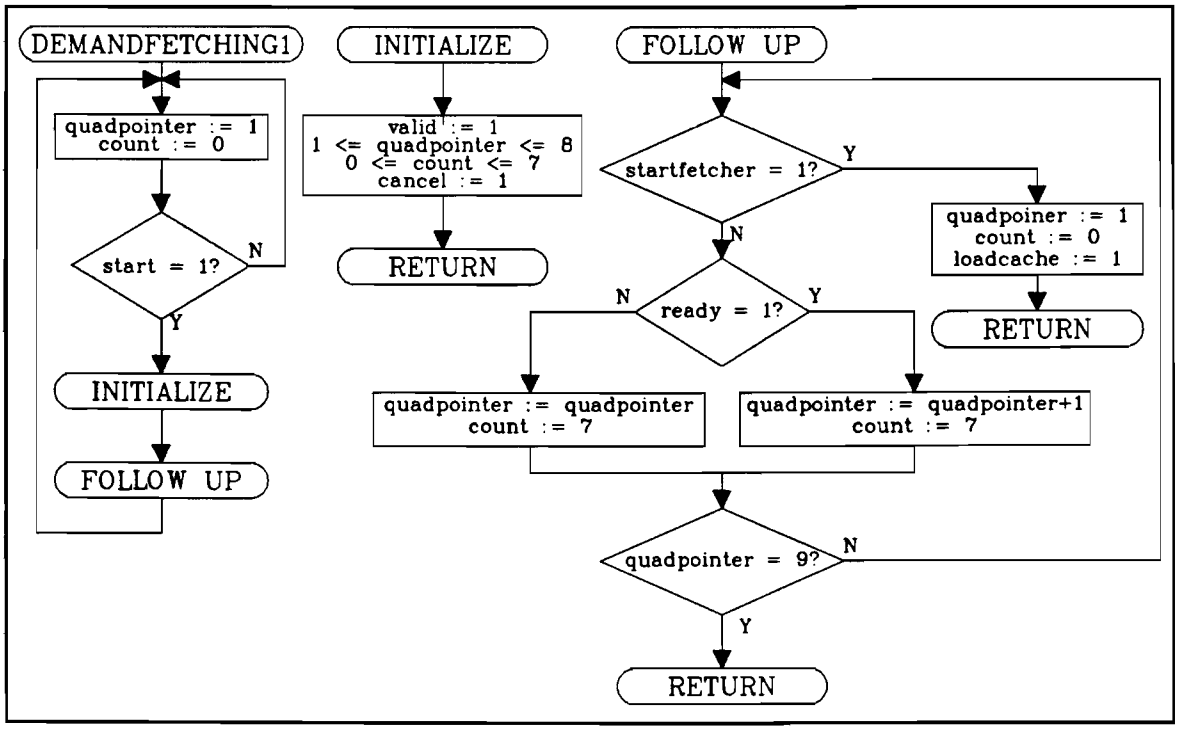


**Figure 23.** The first ASM chart of the demand fetcher.

ready signal is set, the quadpointer has to point the next place in the fetch buffer. This can be done by incrementing the quadpointer register. A last test is done to check if all places in the fetch buffer are filled by comparing the quadpointer with 9. If the quadpointer is equal to nine then the pointer reached over the end of the fetch buffer. Thus a new transfer block can be fetched and the status of the fetch buffer has to be stored in the tag/status RAM.

The same can be told about the ASM chart of the demand fetcher. There is only one difference between these ASM charts. The count signal is this time not always 7, but a value between 0 and 7. This is caused by the wrap around principle of the fetcher algorithm. In case a demand fetch is started, the demand fetch starts if the startfetcher signal is set, the count signal is calculated by the number of quads from the required quad to the end of the transfer block. Two different demand fetch ASM charts are illustrated in the figures 23 and 24. The reason of this difference is explained later in text.

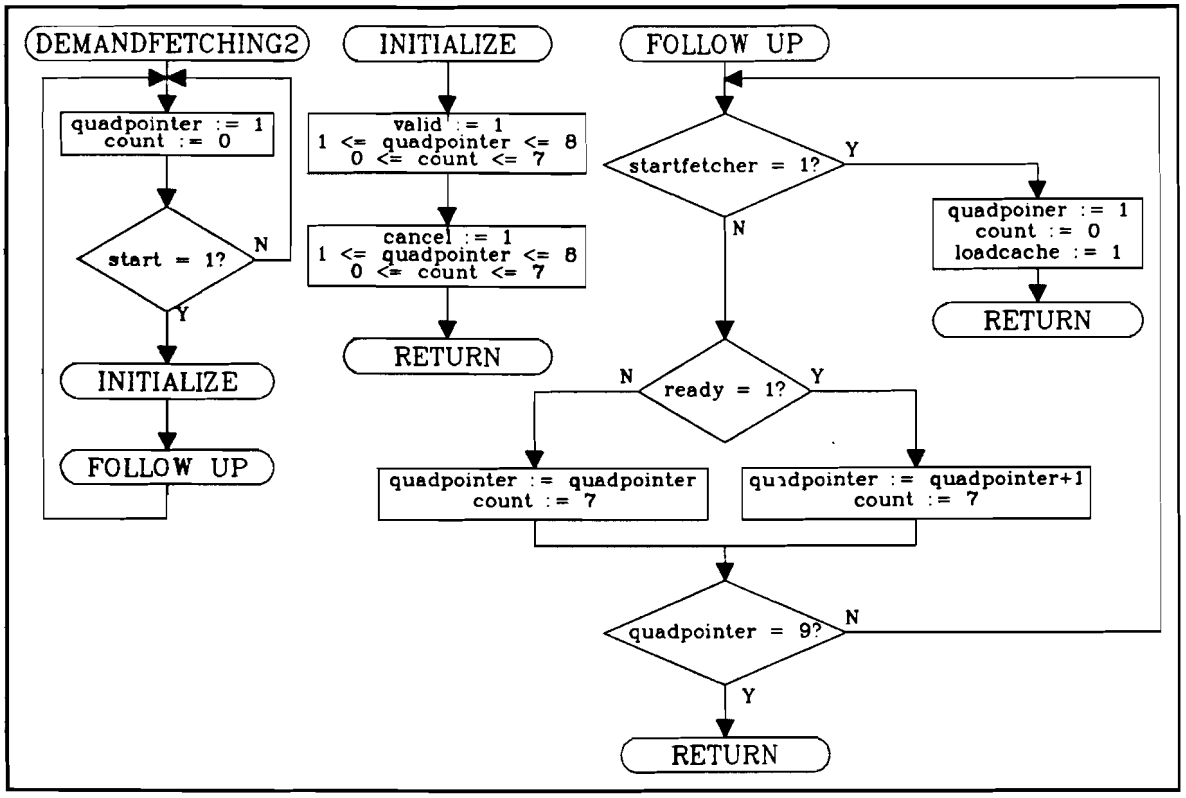


**Figure 24.** The second ASM chart of the demand fetcher.

The demand and prefetcher can not be active simultaneously. Therefor a fetch controller is added to the fetcher. This controller arbitrates the two fetch modules. The controller is also used for indicating in which mode the fetcher operates. This indication is important to the MMU, and the rest of the implementation of the fetcher.

The state diagram of this fetch controller is illustrated in Figure 25. In this figure is the demand fetching divided into two parts:

- initialization (DemandFetch1, DemandFetch2).
- resume part (DemandFetch).

The reason for splitting the demand fetch actions is the difference in initialization of the demand fetch algorithm. In case the fetcher is in rest or prefetching, both valid and cancel signals are set. The cancel signal indicates that the bus unit has to stop all the

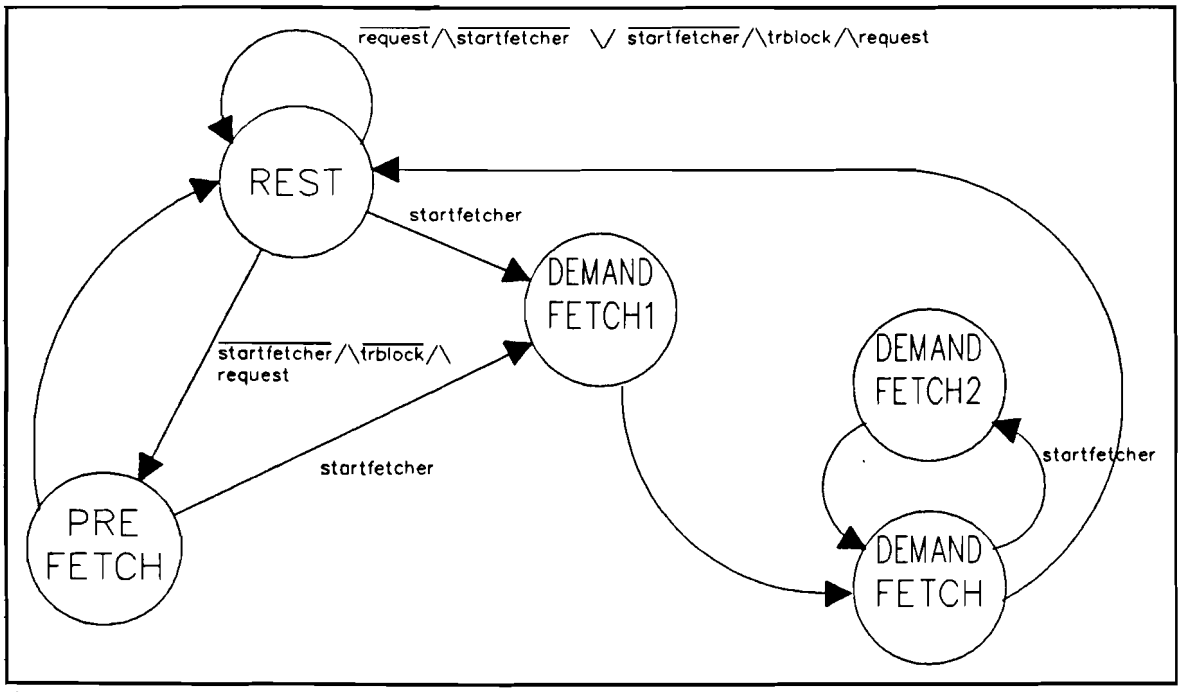


**Figure 25.** The state diagram of the fetch controller.

fetch actions and start the required one. In case the fetcher is demand fetching then the cancel signal is set first and afterwards the valid signal is set. The reason for this delay is the fact that the demand fetch module first has to save the contents in the fetch buffer and than it can start a new fetch cycle. Above mentioned is the promised explanation of using two demand fetcher ASM charts.

The spheres in figure 25 represent the ASM charts of the fetching and replacement methods. Thus, the sphere containing the name prefetch represents the ASM chart of the prefetcher (see figure 22). The first demand fetch ASM is placed into the DemandFetch1 and DemandFetch spheres. The second demand fetch ASM chart is placed into the DemandFetch2 and DemandFetch spheres. The startfetcher signal is set by the server to indicate that the server could not detect the required quads in the cache, thus the quads have to be fetched by the fetcher. The request signal is set when the instruction unit requires for data and the trblock signal is set if the next transfer block is detected inside the cache.

Besides the fetching operation the demandfetch and prefetch sub modules have to take care of the updating of the tag/status RAM with the correct status of the transfer block. In case of a replacement of a new block the tag bits also have to be updated.

## 6.2. The implementation.

The implementation of the fetcher is also divided into three parts:

- comparators.
- prefetch.
- demandfetch.

As in paragraph 6.1. the sub modules have the same mentioned tasks. The "comparators" sub module has to check if the next transfer block is available in case of a cache hit. If the fetcher is busy, the result of the "comparators" is ignored. The "prefetch" sub module takes care of the prefetch operations and the "demandfetch" sub module the demand fetch operations. The schematic of the fetcher is drawn in Figure 26.

Furthermore, three IDaSS operators are implemented to merge the control signals of both fetch sub modules together by multiplexers. In case of prefetching the multiplexers select the "prefetch" sub module and in case of demand fetching the multiplexers select the control signals of the other sub modules. Finally a operator is implemented to merge some status signals into one control bus.

The fetch controller, mentioned in paragraph 6.1., is implemented as a state register (see figure 26), which is controlled by a control connector. The content of this register represents the present state. The value of this state register is used to start a prefetch operation by the "prefetch" sub module or to start a demand fetch operation by the "demandfetch" sub module. No real state controller is used, because if a state controller is used, the following would happen (specific IDaSS problem). If a prefetch operation has to be started then the fetch controller has to be set in the correct state. After this

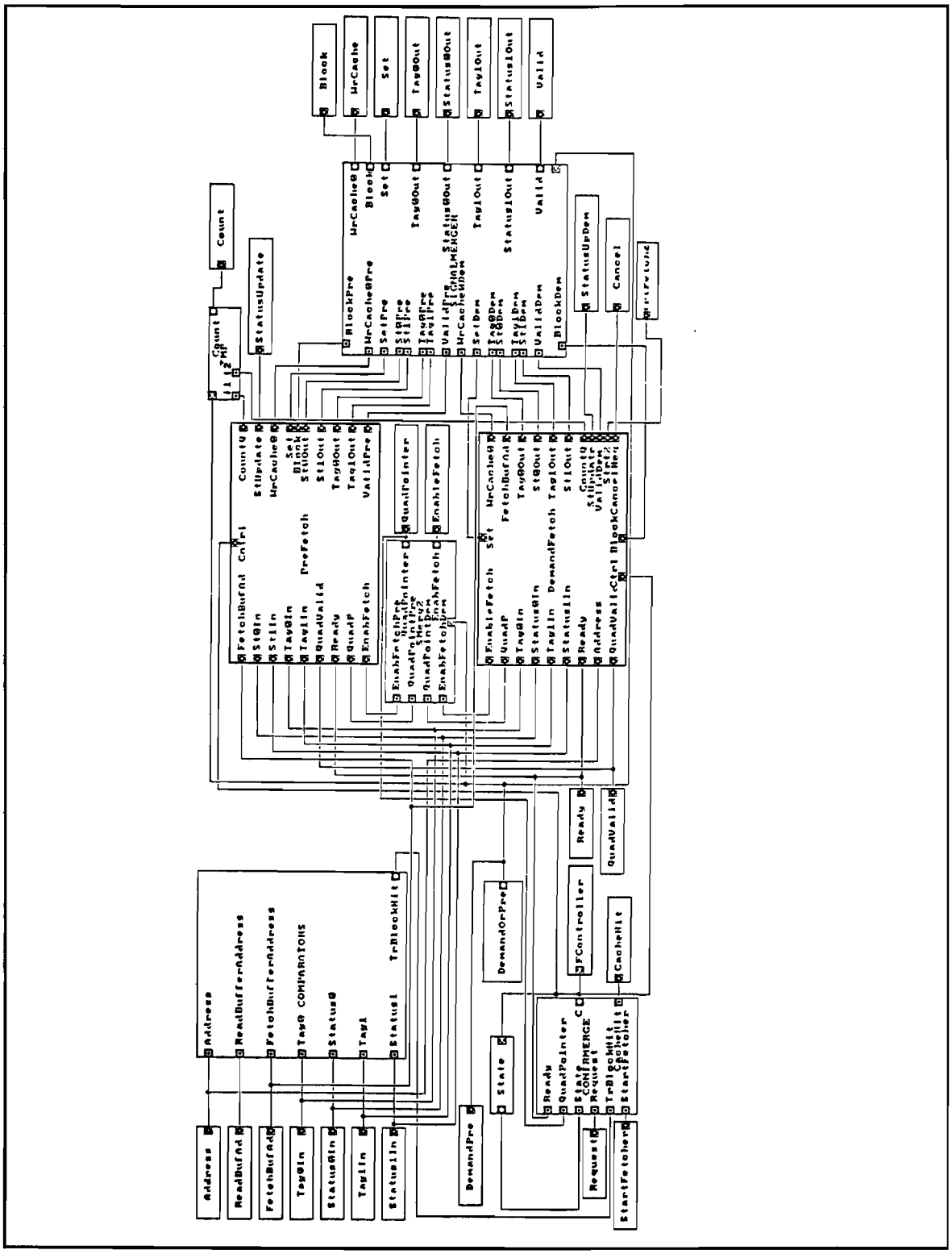

**Figure 26.** The implementation of the fetcher.

is done than the prefetch controller of the "prefetch" sub module has to be set in the correct state. Thus, before the prefetch operation has been started two clocks have passed. To avoid losing one unnecessary clock the input of the fetch controller is connected to the input of the prefetch state controller. The fetch controller indicates only in which fetch method the fetcher is operating. To make an output signal of the state controller a constant generator is used. No combinatorial outputs on a state controller are possible without constant generators in IDaSS. Thus there are two independent state controllers implemented, which execute the algorithms, and one state controller to inform the other modules in which mode the fetcher is operating.

A description of the used modules is described in appendix F.

## 6.3. The "prefetch" sub module.

A state controller is used to implement the "prefetcher" ASM chart of Figure 22. This is very easy because the states can be described as text. Besides the state controller, an operator is added to the "prefetch" sub module, which updates the status of the fetched transfer block at the end of the fetch operation. It also updates the tag bits of the transfer block in case of block replacement. The above mentioned operator is activated by an active loadcache signal. A register is used as a pointer to the correct place in the fetch buffer. This register is called quadpointer. This quadpointer has to increment if and only if a quad is received. The ready signal is set by the bus unit to indicate the presence of the required quad on the data bus. An operator is used to increment the pointer register at the next clock edge if the ready signal is set. The control signals of the interface protocol are implemented as constant generators, which are controlled by the prefetch controller because the restriction is that no output connectors on state controllers are available in IDaSS. The schematic of the "prefetch" module is drawn in Figure 27.

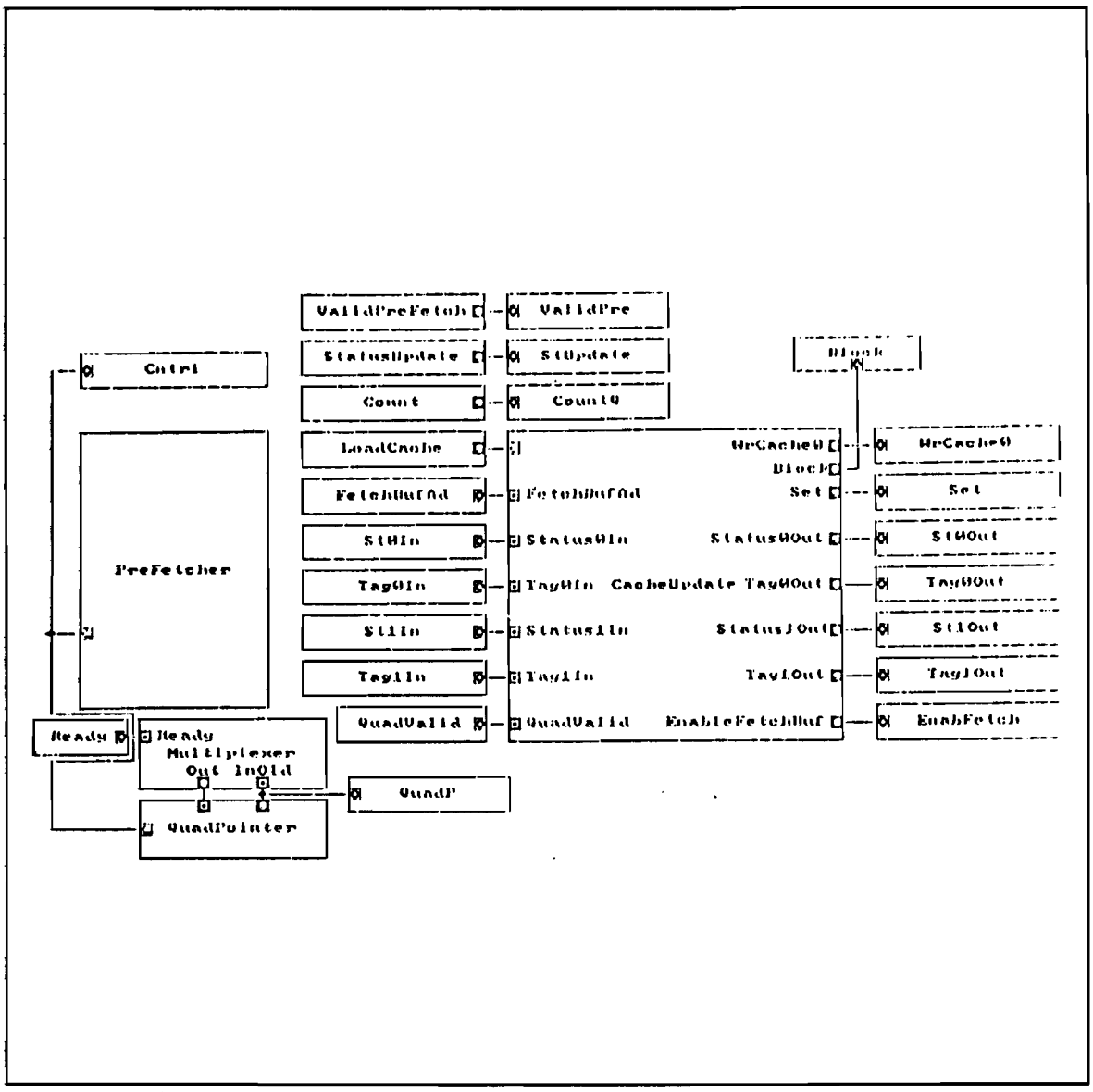


**Figure 27.** The implementation of the prefetcher.

## 6.4. The "demandfetch" sub module.

As in the "prefetch" sub module, a state controller is used to implement both "demandfetch" ASM charts. An operator is also used to implement the updating of the cache status. A register is also used to point to the correct place in the fetch buffer. And even constant generators are used to implement the control signals. Only a more

complex described operator is used to get the correct initialization of the pointer and the correct count value. This is caused by the wrap around option. To control this operator without unnecessary delays the state of the state controller, indicated by the state register, controls this operator directly by control connectors. The schematic of this sub module is illustrated in Figure 28.

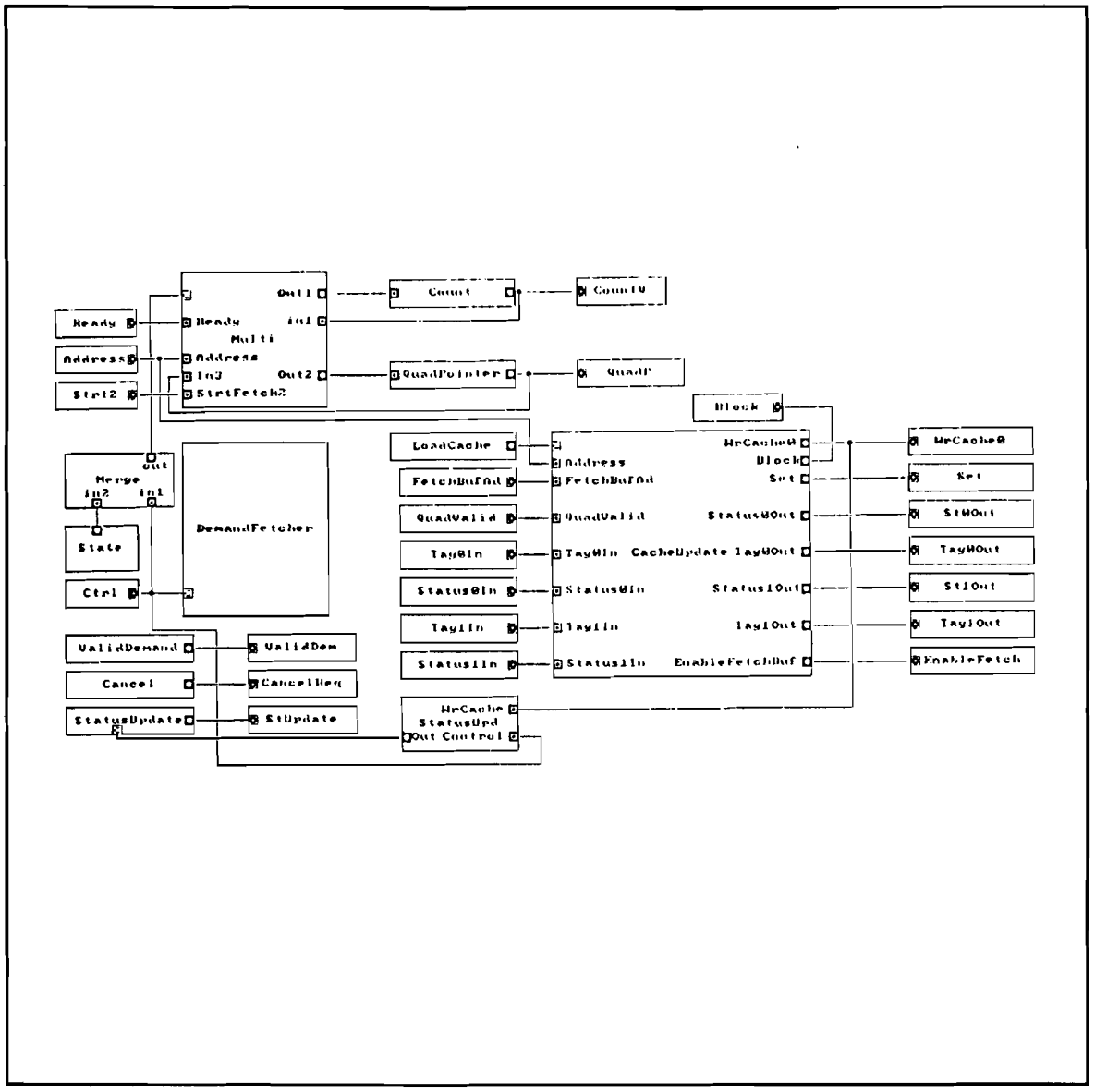


**Figure 28.** The implementation of the demand fetcher.

## 7.1. The conclusions.

This mater thesis describes an implementation of an instruction cache based on the master thesis of J.E.H.M. Bormans. J. Bormans presented a study on instruction caches in general and developed a simulator to estimate the performance of caches with varying the following parameters: cache size, set size, block size, etc. The conclusions of these simulations are used to implement a prototype of the optimum instruction cache. The simulations used an estimated trace, because there was and still is no realistic trace available. The optimum cache organisation may be changed when real traces are used. The implementation of the prototype is described in this master thesis.

The prototype was developed with the IDaSS CAD tool. The most important advantage of using IDaSS was that interactive designing and simulation was possible. While changing the implementation, it is possible to see the results of these changes immediately. Changing the implementation in the Helix environment requires more time, first the changes have to be compiled, the net list of the schematic has to be extracted and a new simulation has to be ran. Further advantages are that state controllers can be described textually. State machines are not very complicated to implement in IDaSS. It is possible to describe the combinatorial logic textually in IDaSS. The text of this descriptions are on a higher level than the complicated gate levels.

IDaSS has also some disadvantages, but most of these will be removed in its next version. One big disadvantage is the restriction that no combinatorial input connectors are available at the state controllers. So, the system can not simulate real Mealy machines, but also no Moore machines, if combinatorial inputs are used. No high level text descriptions are possible for schematics. The consequence of this restriction is that the test environment of the cache had to be implemented on register level. If the designs are complicated, the simulation time grows exponentially. This disadvantage can be reduced by using high performance machines, like the 80386 computers. The last major disadvantage is the lack of timing information in IDaSS. Because of this restriction no estimation can be made of the delay times of the combinatorial logic blocks.

The cache implementation had several functional tests. They are:
- sequential instructions with random jumps.
- the traces of Jos Bormans.

These tests are done successfully, only it is still not certain that the cache is implemented correct for 100 %. Therefor it is better to test the cache with all the situations that are illustrated in Figure 15. Because the lack of time, this is not done at this moment.

## 7.2. Recommendations.

The total cache is implemented as a prototype. Several functions, which are mentioned in chapter four, are not implemented. These are:
- transparent mode.
- cache flushing mode.
- self testing mode.

But the expectation is that this is not much work. Only some multiplexers and one counter have to be implemented. The multiplexers can be used to select between the normal mode and the transparent mode. The counter and some multiplexers are used

to implement the cache flushing mode. By resetting all the block valid bits in the tag/status RAM all data in the cache are lost. The counter is used to generate the address of the tag/status RAM and the multiplexer can be used to select a zero bit as input of the tag/status RAM to reset the block valid bits. By using scan flipflops the testing of the combinatorial logic is no problem. The stuck_at fault model can be used to generate the test patterns. This can be done manually, but it is better to use a computer to calculate these patterns. Only the memories need a special treatment. Because they will be generated by RAM generators, it is impossible to test the memories with the stuck_at fault model. Several test algorithms are developed and one of these can be used. The RAM test algorithm can be implemented as a state controller in IDaSS. This is very easy because the state controller can be described by a textual description. Because of the use of the stuck_at fault model, the self testing can only be implemented when the cache is implemented on gate level. In IDaSS the implementation is on register level. This is the reason that developing self testing circuits is not yet possible. Only the memory test algorithm can be developed, because this is independent of the rest of the implementation.

To be able to vary the width of the buffers and the sizes of the memories, which is needed for a cache generator, it is better to implement the cache in the Helix environment. IDaSS is not yet suitable for this kind of flexibility.

Some timing aspects can be studied if the cache is implemented in Helix. The delay of the combinatorial logic blocks has a big influence on the maximum clock rate of the C-processor. This delay can be reduced by optimizing the combinatorial logic.

# *LITERATURE.*

[Bormans] Bormans, J.E.H.M., "Instruction cache for the C-processor.", Master thesis report EB 205, Technical University Eindhoven, 1989.

[Budzelaar] Budzelaar, F.P.M., "The structured design of a processor for the language C", Master thesis report EB 085, Technical University Eindhoven, 1988.

[Withagen] Withagen, W.J., "Definition and high level desription of the C-processor", Master thesis report EB 168, Technical University Eindhoven, 1988.

# A
# THE TOP LEVEL.

## A.1 The cache.

**Module:** a lower level schematic.

**Name:** Cache.

**Pin Constraints:**

Input: Request, Address, Ack, Ready, MemData.

Output: Ready2, DataLow, DataHigh, Valid, Cancel, Count, PreFAdd, DemandPre.

**Description:**

The total instruction cache is implemented, except several functions of the cache. The expectation is that this is not difficult to implement. The transparent function can be implemented by using multiplexers. The flushing functions can be done by resetting the BlockValid bits in the Tag RAM. Thus by using a simple address counter, from 0 to 15, in 16 clock periods the total cache can be flushed.

The interface between the instruction unit and the cache contains the pins:

- Request, set if the instruction unit request for instructions.
- Address, the instruction address.
- Ack, set if the instruction unit received the required data.

- Ready2 (2 bits), first bit is set if the cache delivered the first quad to the instruction unit, second bit is set if the second quad is delivered to the instruction unit.
- DataLow, the lower data bus.
- DataHigh, the higher data bus.

The interface between the bus unit and the cache contains the pins:

- Ready, set if the bus unit delivered the required data to the cache.
- Valid set if the cache request for data.
- Cancel, set if the cache wants to stop the request for data.
- Count (3 bits), indicates the number of quads, which the cache requires..
- MemData, the data bus.

The interface between the MMU and the cache contains the pins:

- DemandPre (2 bits), if 00 no actions by the MMU are required,
  if 01 translation of the prefetch address is required.
  if 10 translation of the address delivered by the instruction unit is required.
  if 11 translation of the address delivered by the instruction unit is required.
- PreFAd, the address of the prefetch instructions.

## A.2 The bus unit.

**Module:** a lower level schematic.
**Name:** BusUnit.
**Pin Constraints:**
Input: Valid, Count, Cancel, Ready2, DemandPre, Address, PreFAdd.
Output: MemAd, Ready.

**Description:**

This module simulates the bus unit and the MMU. It is created for testing reasons. It calculates the memory address (MemAd) from the address delivered by the instruction unit in case of a demand fetch . And in case of a prefetch it calculates the memory address from the prefetch address (PreFAd). The memory address has to be calculated, because the width of the memory address is 11 bits instead of 46 bits.

## A.3 The test data ROM.

**Module:** ROM, 2048 words of 32 bits each.

**Name:** Data.

**Pin Constraints:**

Input: Ad1, Ad2, Ad3.

Output: D1, D2, D3.

**Description:**

The contents of this ROM is used to test the cache. The contents is used as data from the main memory. The ROM is filled with its own addresses.

## A.4 The test address decoder.

**Module:** an operator, 1 function.

**Name:** AdDec.

**Pin Constraints:**

Input: i1.

Output: o1, o2.

**Description:**

This module converts the address supplied by the instruction unit into a memory address with width of 11 bits. The o1 output presets the higher quad address and the o2 output presents the lower quad address.

**Logical Description:**

Function "Default"

"The higher quad address."

o1 := i1 From:0 To:10.

"The lower quad address."

o2 := i1+1 From:0 To:10.

## A.5 The test data comparator.

**Module:** an operator, 1 function.

**Name:** Compare.

**Pin Constraints:**

Input: in1, in2 d1,d2, Ack

Output out.

**Description:**

This module checks the outputs of the cache with the data in the Data memory. It checks only in the case that the instruction unit received the data, thus only in the case of an active ack signal. The in1 and in2 pins are the data, which has to be checked. The d1 and d2 pins are the test data.

**Logical Desription:**

Function "Compare"

out := ack if0:1

if1:( (in1 = d1) /\ (in2=d2) ).

"check the data from the cache with the data from the data ROM, if and only if ack is active ."

## A.6 The clock counter.

**Module:** a register, 32 bits.
**Name:** teller.
**Pin Constraints:**
Input: none.
Output: no name.
**Description:**
This register is used as a counter for testing reasons. The value of the counter represents the number of the clock. In case of a mismatch of the data from the cache with the data from the ROM, the clock number is saved into a LIFO memory.

## A.7 The test errors FIFO memory.

**Module:** FIFO, 8 words of 32 bits each.
**Name:** Errors.
**Pin Constraints:**
Input: in, no name.
Output: out.
**Description:**
This LIFO memory is used to store the clock number, delivered by the register "teller", for testing reasons. If a missmatch between the data from the cache and data from the data ROM occurs, then the FIFO stores the clock number and generates an error to stop the simulator. It is possible to continue the simulator until 8 missmatches are detected. To do last mentioned, the incorrect function call has to be removed from the control connector specification.
**Control Connector Specification:**

%0 write;errorInCache.
"in case of a missmatch between data from the cache and from the data ROM, the FIFO will store the clock number. By calling the function errorinCache, the simulator stops."

## A.8 The request signal generator.

**Module:** a register, 1 bit.
**Name:** Request.
**Pin Constraints:**
Input: no name (Control Connector).
Output: no name.
**Description:**
This register simulates the request signal of the instruction unit. If this register is set, then the instruction unit requirs for data from the cache.
**Control Connector Specification.**
%1 Setto:1.
"If the ack signal is set, then the instruction unit has taken the data from the instruction bus and a new request can start."

## A.9 The acknowledge signal generator.

**Module:** a constant generator, 1 bit
**Name:** Ack.
**Pin Constraints:**
Input: no name (Control Connector).
Output: no name.
**Description:**

This constant generator is used to simulate the asynchronuous ack signal of the instruction unit. The ack signal is set in case the instruction unit has taken the data from the instruction bus. It is assumed that the instruction unit takes the data immediatly after delivering by the cache. Therefor the ack signal is set after setting the ready signals.

**Control Connector Specification.**

"If the cache has not finished is actions, then the last ready signal (Ready2_2) is not set." %x0 Setto:0.

"If the cache has delivered both quads, then both Ready2_1 and Ready2_2 are set." %11 Setto:1.

## A.10 The trace address counter.

**Module:** a register, 13 bits.

**Name:** Counter.

**Pin Constraints:**

Input: no name (Control Connector).

Output: no name.

**Description:**

This register is used as a address counter for the trace memory.

## A.11 The trace ROM.

**Module:** ROM, 6545 words of 46 bits each.

**Name:** Trace.

**Pin Constraints:**

Input: ad.

Output: d.

**Description:**

This ROM is used to store the test trace. The trace is generated by the trace genarator of Jos Bormans. The trace generator is available in the directory //station_6/user/stud/hu/bormans. The command to start the trace generator is "tracegen", it asks for the minimal length of the desired trace. A special convert program is written to convert the trace output of Jos Bormans into Intel Hex format. This format can be loaded into the IDaSS memories directly. The conversion program is available in the IDaSS directory //station_6/user/stud/hu/idass and is named "hu.exe". The object file and C source are also available in the IDaSS directory (resp. hu.obj and hu.c).

## B.1 The fetcher.

**Module:** a lower level schematic.

**Name:** Fetcher.

**Pin Constraints:**

Input: Tag0In, Status0In, Tag1In, Status1In, CacheHit, StartFetcher, StrtFetch2, QuadValid, FetchBufAd, ReadBufAd, Request, Address, Ready.

Output: Tag0Out, Status0Out, Tag1Out, Status1Out, Set, StatusUpdate, StatusUpdDem, Cancel, DemandPre, Valid, Block, QuadPointer, WrCache, EnableFetch.

**Description:**

The fetcher fetches the quads from the main memory via the bus unit. There are two ways to fetch the quads:

- Demand Fetching: Fetching quads, which are required by the instruction unit.
- PreFetching: Fetching quads, which are not yet required by the instruction unit.

The used prefetch algorithm is one_transfer block_lookup on hits and the used placement algorithm is the Least Recently Used algorithm.

## B.2 The server.

**Module:** a lower level schematic.

**Name:** Server.

**Pin Constraints:**

Input: Tag_0, Status0, Tag1, Status1, Quad_Valid, FetchBufAdres, ReadBufAdres, Adres, Request, Ack.

Output: Stat0, Stat1, Set, CacheHit, Start_fetcher, StrtFetch2, StatusReg, Ready, LoadReadBuf, Word, WrCache, CtrlS.

**Description:**

The server receives the request signal from the instruction unit. If the request signal is set, then the server locates the required quads. If they are in the cache, then the server delivers the required quads to the instruction unit and it sets the appropriate control signals. If one or both quads are not in the cache, then the server starts the fetcher by setting the start_fetcher signal. If the second quad is not in the cache then the server sets also the StrtFetch2 signal to indicate the second quad is requested.

## B.3 The tag RAM.

**Module:** a lower level schematic.

**Name:** TagRam.

**Pin Constraints:**

Input: Tag0InF, Status0InF, Tag1InF, Status1InF, SetF, Tag0InS, Status0InS, Tag1InS, Status1InS, SetS, Ctrl.

Output: Tag0OutF, Status0OutF, Tag1OutF, Status1OutF, Tag0OutS, Status0OutS, Tag1OutS, Status1OutS.

**Description:**

This module contains the tag/status memories. Not one wide memory is used, because the restriction of a maximum width in IDaSS. Besides this restriction a memory, which is very wide, is not realistic. Therefore the tag/status memory is divided into Tag0, Tag1, Status0 and Status1 memories.

## B.4 The data RAM.

**Module:** a lower level schematic.

**Name:** Data RAM.

**Pin Constraints:**

Input: Address, FetchAd, Ctrl, Block, Min1, Min2, Min3, Min4.

Output: Mout1, Mout2, Mout3, Mout4.

**Description:**

The fetched quads are stored in this module. The dimension of the used memory is: 128 word * 256 bits. This very wide memory is divided into four parts with dimension: 128 words * 64 bits. This is caused by the restriction of the memories. The memories can not be wider than the maximum bus width (default 64 bits).

## B.5 The prefetch buffer.

**Module:** a lower level schematic.

**Name:** PreFetchBuffer.

**Pin Constraints:**

Input: Cancel, Valid, Ctrl, in.

Output: QuadValid, DL, DH, Mout1, Mout2, Mout3, Mout4.

**Description:**

This schematic is the implementation of the fetch buffer. Quad bypassing is also implemented to speed up the delivering of the required quads to the instruction unit. The fetch buffer is divided into a data part, which contains the quads, and a status part, which contains the status information. A control unit is used to guide the quad to the correct data register and set the appropriate status register.

## B.6 The read buffer.

**Module:** a lower level schematic.
**Name:** ReadBuf.
**Pin Constraints:**
Input: Min1, Min2, Min3, Min4, Ctrl, St0, St1, Ad.
OutPut: DH, DL, St.
**Description:**
This schematic is the implementation of the read buffer. In case of a cache memory hit the quads are bypassed to the instruction unit. After this unit has taken the quads off the instruction bus, the transfer block, which contains the delivered quads, is copied into the data registers of the read buffer. Besides the transfer block also the status of this transfer block is copied to the status registers of the read buffer. The reason of copying the status of the transfer block is the fact that not all quads have been loaded into the fetch buffer. So not all quads in the transfer block are valid.

## B.7 The status register.

**Module:** a lower level schematic.
**Name:** Status_Reg.

**Pin Constraints:**

Input: PreFAd, StReg, Ready, Address, StUpDem, StUpPre.
Output: FetchAd, ReadAd.

**Description:**
The addresses of the transfer block in the read buffer and in the fetch buffer have to be stored. The reason is that the server and the fetcher has to distinguish which transfer block is in the buffer. The addresses of the transfer block are stored in this module.

## B.8 The prefetch address decoder.

**Module:** an operator, 1 function.
**Name:** PAdDec.
**Pin Constraints:**
Input: in.
Output: out.
**Description:**
This module extracts the prefetch address from the address, which is delivered by the instruction unit. The prefetch address is used to set the appropriate transfer block address into the status register, and it is delivered to the Memory Management Unit.
**Logical Description:**
Function: decode.
"The transfer block address is extracted from the address value and the extracted address is incremented to get the next transfer block address. The Prefetch address is gotten by a concatenation of 3 zeroes. The 3 zeroes represent the word field of the prefetch address."
out := (in from:3 to:45) inc, 3 zeroes.

## C.1 The tag RAM.

**Module:** RAM, 16 words of 37 bits each.

Name: Tag0.

**Pin Constraints:**

Input: WrF, AdS1, AdS2, AdF1, AdF2, no name (Control Connector).

Output: RdS, RdS2, RdF.

**Description:**

This Ram contains the tag of the second block in the set. It controlled by a control connecter, which enables the several write ports of the RAM.

Control Specification:

(13) "WrCacheF."

%1 Write.

## C.2 The status RAM.

**Module:** RAM, 16 words of 34 bits each.

**Name:** Status0.

**Pin Constraints:**

Input: WrS1, WrS2, WrF, AdS1, AdS2, AdS3, AdS4, AdF1, AdF2, no name (Control Connector).

Output: RdS1, RdS2, RdF.

**Description:**

This Ram contains the status of the first block in the set. It is controlled by a control connecter, which enables the several write ports of the RAM.

**Control Connector Specification:**

(15,14,13) "WrCacheS2, WrCacheS1, WrCacheF."

%10x Write: WrS1.

%11x Write: WrS1.

%01x Write: WrS2.

%11x Write: WrS2.

%001 Write: WrF.

## C.3 Splitting and merging busses.

**Module:** an operator, 1 function.

**Name:** Extract.

**Pin Constraints:**

Input: in.

Output: out1, out2.

**Description:**

The server delivers two addresses to the tag/status RAM. One merged bus is used instead of two separated busses. This is done to reduce the number of I/O busses. This operator divides the wide merged bus into two sub busses.

**Logical Description:**

out1 := in From:0 To:3.

out2 := in From:4 To:7.

**Module:** an operator, 1 function.
**Name:** Merge1.
**Pin Constraints:**
Input: in1, in2.
Output: out.
**Description:**
The server gets the tag bits of the first quad and the tag bits of the second quad. The two busses are merged into one bus. This is implemented to reduce the number of the I/O busses.
**Logical Description:**
Function merge:
out := in1,in2.

**Module:** an operator, 1 function.
**Name:** Merge3.
**Pin Constraints:**
Input: i.
Output: o1, o2.
**Description:**
The server delivers the status dependent of the first quad and the status dependent of the second quad. One merged bus is used instead of two seperated busses. This is done to reduce the number of I/O busses. This operator divides the wide merged bus into two sub busses.
**Logical Description:**
Function merge:
o1 := i From:0 To:33.
o2 := i From:34 To:67.

**Module:** an operator, 1 function.
**Name:** Merge4.

**Pin Constraints:**

Input: i1, i2.

Output: o.

**Description:**

The server gets the status bits of the first quad and the status bits of the second quad. The two busses are merged into one bus. This is implemented to reduce the number of the I/O busses.

**Logical Description:**

Function merge:

out := i1,i2.

## C.4 The data RAM.

**Module:** 128 words of 64 bits each.

**Name:** Data1.

**Pin Constraints:**

Input: RdAd, WrAd, WrPort, no name (Control Connector).

Output: RdPort.

**Description:**

This RAM contains the quads, which are stored in the cache. It contains the seventh and the eighth quad of the transfer block. The added control connector enables the write port of the RAM.

**Control Connector Specification:**

%x1 Enable:RdPort. "set by default."

%1x Write.

## C.5 The data RAM arbiter.

**Module:** an operator, 1 function.

**Name:** Arbiter.

**Pin Constraints:**

Input: Block, Address, FetchAd, Ctrl.

Output: MemCtrl, Ad.

**Description:**

The data RAM is implemented as a single ported RAM. Therefor a arbiter is used to arbitrate when the server and the fetcher want to have access to the data RAM.

**Logical Description:**

"Ctrl : <QuadPointer>,<Ready>,<EnableFetch>,<WrCacheS2>,<WrCacheS1>, <WrCacheF>,<Word>,<LoadReadBuf>,<Block0>,<Block1>,<ReadHit>, <FetchHit>,<CacheHit>,<ReadHit2>,<FetchHit2>,<CacheHit2>,<Next>."

"Generating the status signals."

```
_CacheHit       := Ctrl at:4.
_CacheHit2      := Ctrl at:1.
_WrCacheF       := Ctrl at:13.
_WrCacheS1      := Ctrl at:14.
_WrCacheS2      := Ctrl at:15.
_EnableFetch    := Ctrl at:16.
_Next           := Ctrl at:0.
```

"Generating the address of the requested transfer block."

```
_Address        := ((Address From:0 To:2) ~= 7)
                     if1:(_CacheHit
                             if1:Address
                             if0:(_EnableFetch
                                  if1:(FetchAd, 3 zeroes)
                                  if0:Address))
                     if0:(_CacheHit /\ _CacheHit2
```

```
                        if1:Address
                        if0:( _CacheHit /\ _CacheHit Not
                                if1:Address
                                if0:(_CacheHit /\ _Next Not
                                        if1:Address
                                        if0:(_CacheHit2 /\ _CacheHit Not
                                                if1:Address+1
                                                if0:(_CacheHit2 /\ _Next
                                                        if1:Address+1
                                                        if0: (_EnableFetch
                                                                if1:(FetchAd,3zeroes)
                                                                if0:Address ))))))).
_TrBlock    := _Address From:3 To:4.
_Set        := _Address From:5 To:8.
_Block      := ((Address From:0 To:2) ~= 7)
            if1: ( _CacheHit
                    if1: (Ctrl at:8)
                    if0: ( _CacheHit2
                            if1:(Ctrl at:7)
                            if0:((_EnableFetch) if1: Block if0: 1 zeroes ) ) )
            if0: ( _CacheHit /\ _CacheHit2
                    if1:(Ctrl at:8)
                    if0:(_CacheHit /\ _CacheHit2 Not
                        if1: (Ctrl at:8)
                        if0: (_CacheHit /\ _Next Not
                                If1: (Ctrl at:8)
                                If0:(_CacheHit2 /\ _CacheHit Not
                                    If1:(Ctrl at:7)
                                    If0:(_CacheHit2 /\ _Next
                                        if1: (Ctrl at:7)
                                        if0: (_EnableFetch
```

```
                                        if1: Block
                                        if0: 0) ) ) ) ) ).
"Generating the write enable signal."
_Write      := _WrCacheF /\ _WrCacheS1 not /\ _WrCacheS2 not.

MemCtrl     := _Write,1 ones.
Ad          := _Set , _Block , _TrBlock
```

## D.1 The data register of the read buffer.

**Module:** a lower level schematic.
**Name:** Quad8.
**Pin Constraints:**
Input: In, Ctrl.
Output: Out.
**Description:**
This schematic is the implementation of a special data register. It contains a data register and a multiplexer, which select between the input from the register or input from the cache memory. The selection of the inputs and the control of the registers are done by the ctrl input connector.

## D.2 The status register of the read buffer.

**Module:** a register, 1 bit.
**Name:** Status8.

**Pin Constraints:**

Input: no name, no name (Control Connector).

Output: no name.

**Description:**

This register contains the status of the quad in the transfer block. If the quad is present in the loaded transfer block, then this bit is set. Other wise it is reset.

**Control Connector Specification:**

(9) "load read buffer signal"

%1 load.

## D.3 The transfer block status.

**Module:** an operator, 1 function.

**Name:** StatusDetect.

**Pin Constraints:**

Input: St0, St1, Ctrl, Ad.

Output: Status.

**Description:**

This operator extracts the status of the transfer block, which is being loaded into the read buffer.

**Logical Description:**

"Status signal generation."

_CacheHit := Ctrl at:4.

_CacheHit2 := Ctrl at:1.

_Block0 := Ctrl at:8.

_Block1 := Ctrl at:7.

"Transfer block field generation."

_TrBlock := _CacheHit if1: (Ad From:3 To:4 )

```
                                        if0:(_CacheHit2     if1:((Ad+1) From:3 To:4)
                                                            if0:0).
"Block status of the second quad."
_St0                    := St0 From:34 To:67.
_St1                    := St1 From:34 To:67.
"Transfer block status of the first quad."
_Status1                := _CacheHit        if1: (_Block0
                                                    if0:(St0 At:(_TrBlock,3 zeroes) Width:8)
                                                    if1:(St1 At:(_TrBlock,3 zeroes) Width:8))
                                                if0: ( 8 zeroes).
"Transfer block status of the second quad."
_Status2 := _CacheHit2 if1: (_Block1 if0: (_St0 At:(_TrBlock,3 zeroes) Width:8)
                                    if1: (_St1 At:(_TrBlock,3 zeroes) Width:8) )
                        if0: ( 8 zeroes).
Status := _CacheHit if1: _Status1 if0:_Status2.
```

## D.4 The read buffer controller.

**Module:** an operator,1 function.

**Name:** ReadBufControl.

**Pin Constraints:**

Input: Ctrl

Output: HighCtrl, LowCtrl.

**Description:**

This operator generates the selection bits of the multiplexers.

**Logical Description:**

```
" C o n t r o l B u s :
<QuadPointer>,<Ready>,<EnableFetch>,<WrCacheS2>,<WrCacheS1>,
```

```
<WrCacheF>,<Word>,<LoadReadBuf>,<Block0>,<Block1>,<Readhit>,
<FetchHit>,<CacheHit>,<ReadHit2>,<FetchHit2>,<CacheHit2>,<Next>."
"The word field of the address."
_Word       := Ctrl From:10 To:12.
"Status signals generation."
_ReadHit    := Ctrl at:6.
_ReadHit2   := Ctrl at:3.
_CacheHit   := Ctrl at:4.
_CacheHit   := Ctrl at:1.
"Selection."
HighCtrl    := (_Word = 7) if0:((_ReadHit2 \/ _CacheHit2)
                        if0: %0000 "The quads are not located in the read buffer
                                    or in the cache memory."
                        if1:((_word+1),1 ones))
                    if1:((_ReadHit2 \/ _CacheHit2)
                        if0: %0000 "The quads are not located in the read buffer
                                    or in the cache memory."
                        if1:((3 zeroes),1 ones)).
LowCtrl     := (_Word = 7) if0:((_ReadHit \/ _CacheHit)
                        if0: %0000 "The quads are not located in the read buffer
                                    or in the cache memory."
                        if1:((_word),1 ones))
                    if1:((_ReadHit \/ _CacheHit)
                        if0: %0000 "The quads are not located in the read buffer
                                    or in the cache memory."
                        if1:((3 ones),1 ones)).
```

## D.5 The multiplexers.

**Module:** an operator, 8 function.
**Name:** multihigh.
**Pin Constraints:**
Input: in1, in2, in3, in4, in5, in6, in7, in8, no name (Control Connector).
Output: outhigh.
**Description:**
This operator is implemented as a multiplexer, which selects the higher quad. The selection is controlled by a control connector, which selects the function of the operator. The function selects the input connector.

**Module:** an operator, 8 functions.
**Name:** multilow.
**Pin Constraints:**
Input: in1, in2, in3, in4, in5, in6, in7, in8, no name (Control Connector).
Output: outlow.
**Description:**
This operator is implemented as a multiplexer, which selects the lower quad. The selection is controlled by a control connector, which selects the function of the operator. The function of the operator selects the input connector.

## D.6 The fetch buffer controller.

**Module:** an operator, 1 function.
**Name:** FetchBufCtrl.
**Pin Constraints:**
Input: Cancel, Valid, Ctrl.
Output: CtrlQuad, CtrlStatus, LowCtrl, HighCtrl.
**Description:**

This module generates the control signals of the multiplexers and the registers.

**Logical Description:**

"Generating the status signals."

_WrCacheS := Ctrl at:14 width:2.

_QP := Ctrl From:18 To:21.

_Rst := Ctrl at:16.

_Word := Ctrl From:10 To:12.

_FetchHit := Ctrl at:5.

_FetchHit2 := Ctrl at:2.

"Checking if the cache can receive the data."

_Rdy := (Valid Not /\ Cancel) if0:(Ctrl at:17) if1:0.

"Generating the control signals."

HighCtrl := (_Word = 7) if0:((_FetchHit2) if0: %0000 if1:((_word+1),1 ones))
if1:((_FetchHit2) if0: %0000 if1:((3 zeroes),1 ones)).

LowCtrl := (_Word = 7) if0:((_FetchHit ) if0: %0000 if1:((_word),1 ones))
if1:((_FetchHit ) if0: %0000 if1:((3 ones),1 ones)).

CtrlStatus := ((_WrCacheS ~= 0) /\ _Rst) if0:(_QP,_Rdy,_Rst) if1:(_Qp,_Rdy,1 zeroes).

CtrlQuad := _Qp,_Rdy.

## E.1 The comparators.

**Module:** a lower level schematic.
**Name:** Comparators
**Pin Constraints:**
Input: readbufadres, fetchbufadres, ReadBufSt, adres, tag_0, tag_1, status0, status1, quad_valid, ack.
Output: word, readhit, fetchhit, cachehit, borderhit, quadhit_1, quadhit_2, block0, block1, set, readhit2, fetchhit2, cachehit2, stat, stat0, stat1.
**Description:**
This schematic is the implementation of the comparators. They locate the required quads. In case of a cache memory hit, the LRU bit is updated.

## E.2 The status signal merger.

**Module:** an operator, 1 function.
**Name:** SignalMerger.

**Pin Constraints:**

Input: readhit, fetchhit, cachehit, borderhit, quadhit_1, quadhit_2, block0, block1, readhit2, fetchhit2, cachehit2, request, next, ack, state.

Output: CacheHitF, Chit, contrbus2, contrbus, Control.

**Description:**

This operator generates three status signal bus and two other status signals. The most of the status signals are delivered by the comparators sub module.

**Logical Description:**

"Merging of comparators status signals to input buses."

contrbus := readhit,fetchhit,cachehit,borderhit,quadhit_1,quadhit_2,next, request,1 ones,cachehit2.

contrbus2 := readhit2,fetchhit2,cachehit2,quadhit_2,next,request,borderhit.

CacheHitF := Ack if1:(cachehit \/ cachehit2) if0:0.

Control := block0,block1,readhit,fetchhit,cachehit,readhit2,fetchhit2,cachehit2,Next.

Chit := CacheHit,(CacheHit2/\borderhit).

## E.3 The first combinatorial block.

**Module:** an operator, 16 functions

**Name:** Comb_1.

**Pin Constraints:**

Input: ctrl (Control Connector).

Output: out, go.

**Description:**

This module generates the servers control signals of the first quad. A control connector is used to call a function of this operator. The functions set the appropriate signals. The control signals are merged into one bus and the signals of this bus is extracted by the module Comb_3.

**List of functions:**

cachehit1

cachehit2

cachehit3

default

fetchhit1

fetchhit2

fetchhit3

memory1

memory2

memory3

readhit1

readhit2

wait1

wait2

wait3

wait4

**Control Connector Specifications:**

%xxxxxx001x default. "Rest state"

"No transfer block border cross"

%1000xx011x , %1000xx101x readhit1. "First quads is in the read buffer."

%01000x011x , %01000x101x wait1. "The transfer block is located in the fetch buffer."

%010010011x , %010010101x fetchhit1. "The first quad is located in the fetch buffer."

%010011011x , %010011101x fetchhit2. "First quad is in the fetch buffer."

%0010xx011x , %0010xx101x cachehit1. "First quad is in the Cache memory."

%00000x011x , %00000x101x wait3. "First quad is not in the Cache."

%000010011x , %000010101x memory1. "First quad is requested from the memory."

%000011011x , %000011101x memory2. "First quad is requested from the memory."

"Transfer block border cross."

%1001xx011x , %1001xx101x readhit2. "First quad is in the read buffer."

%01010x011x , %01010x101x wait2. "Transfer block is in the fetch buffer."

%01011x011x , %01011x101x fetchhit3. "First quad is in the fetch buffer."

%0011xx0111 , %0011xx1011 cachehit2. "Both quads are in the Cache memory."

%0011xx0110 , %0011xx1010 cachehit3. "First quad is in the Cache memory."

%00010x011x , %00010x101x wait3. "First quad is not in the Cache."

%00011x011x , %00011x101x memory3. "First quad is requested from the memory."

## E.4 The second combinatorial block.

**Module:** an operator, 7 functions

**Name:** Comb_2

**Pin Constraints:**

Input: Ctrl (Control Connector), go.

Output: out.

**Description:**

This module generates the servers control signals of the second quad. A control connector is used to call a function of this operator. The functions set the appropriate signals. The control signals are merged into one bus and the signals of this bus is extracted by the module Comb_3.

**List of Functions:**

cachehit

default

fetchhit

memory

readhit

wait1

wait2

**Control Connector Specifications:**

%xxxx00x default.

%100x10x , %100x01x readhit. "The second quad is in the read buffer."

%010010x , %010001x wait1. "The transfer block of the second quad is in the fetch buffer."

%010110x , %010101x fetchhit. "The second quad is in the fetch buffer."

%001x10x , %001x01x cachehit. "The second quad is in the cache buffer."

%000010x , %000001x wait2. "The second quad is not in the cache."

%000110x , %000101x memory. "The second quad is requested from the memory."

## E.5 The state controller.

**Module:** a state machine with 2 states.

**Name:** SContrroller.

**Pin Constraints:**

Input: no name (Control Connector).

Output: none

**Description:**

This state controller controls the wait cycles of the server by setting the "next" constant generator to 1.

**State Description:**

Busy1:

Next setto:0;

ctrlstate setto:2;

<<

Busy2:

Next setto:1;

ctrlstate setto:3;

<<

**Control Connector Specification:**

"control = <ctrlstate>,<wait>,<ack>,<request>."

%10000 goto: busy1. "no actions required by the instruction unit."

%10001 goto: busy2. "waiting for active ack sinal."

%10x11 goto: busy1. "ack is set, server can go to the rest state."

%10101 goto: busy2. "wait signal is set, server has not finished."

%11x00 goto: busy2. "waiting for active ack signal."

%11x1x goto: busy1. "ack is set, server can go to the rest state."

## F.1 The comparators.

**Module:** an operator, 1 function.

**Name:** Comparators.

**Pin Constraints:**

Input: Address, ReadBufferAddress, FetchBufferAddress, Tag0, Status0, Tag1, Status1.

Output: TrBlockHit.

**Description:**

This module locates the next transfer block of the requested one. If the next transfer block is in the cache, the trblockhit output signal is set. If the next tranfer block is not located in the cache, the trblockhit output signal is not activated.

**Logical description:**

Function Compare.

"Extracting the next transferblock address out of the delivered address."

```
_Address                 := (Address From:3 To:45) inc , (0 Width:3).
_TransferBlockAddress    := _Address From:3 To:45.
_Tag                     := _Address From:9 To:45.
_TransferBlock           := _Address From:3 To:4.
```

```
"In case the requested transferblok is in the ReadBuffer.
_ReadHit                := (_TransferBlockAddress = ReadBufferAddress). "
"In case the requested transferblok is in the FetchBuffer."
_FetchHit               := (_TransferBlockAddress = FetchBufferAddress).
"In case the requested transferblok is in the Cache memory."
_Block0                 := (Tag0 = _Tag) /\ (Status0 at:32) /\
                        ((Status0 at:_TransferBlock , 3 zeroes Width:8) = 0FFh).
_Block1                 := (Tag1 = _Tag) /\ (Status1 at:32) /\
                        ((Status1 at:_TransferBlock , 3 zeroes Width:8) = 0FFh).
_CacheHit               := _Block0 \/ _Block1.
"In case of the transferblock that realy is requested is in the fetchbuffer it has to
remain in the buffer."
_Test                   := (Address From:3 To:45) = (FetchBufferAddress).
"Mapping the temporal variables on the outputs."
TrBlockHit              := _Test if0:(_FetchHit \/ _CacheHit) if1:1.
```

## F.2 The prefetcher.

**Module:** a lower level schematic.

**Name:** PreFetch.

**Pin Constraints:**

Input: FetchBufAd, St0In, St1In, Tag0In, Tag1In, QuadValid, Ready, Ctrl.

Output: CountQ, StUpdate, WrCache0, Set, Block, St0Out, St1Out, Tag0Out, Tag1Out, QuadP, EnabFetch, ValidPre.

**Description:**

This module is the implementation of the prefetcher. It is activated on the following conditions:

* an active request signal of the instruction unit and
* a nonactive trblockhit signal of the comparators in the fetcher and

* a nonactive startfetcher signal of the server.

After finishing prefetching a transfer block, this module has to update the status of the block, which contains the transfer block. In case of a block replacement the tag bits of the block has to be updated.

## F.3 The demand fetcher.

**Module:** a lower level schematic.

**Name:** DemandFetch.

**Pin Constraints:**

Input: Tag0In, Tag1In, Status0In, Status1In, Address, QuadValid, Ctrl, Ready, Address, Strt2, FetchBufAd.

Output: EnableFetch, Set, WrCache0, Tag0Out, Tag1Out, Status0Out, Status1Out, CountQ, StUpdate, ValidDem, CancelReq, Block, QuadP.

**Description:**

This module represents the implementation of the demand fetcher. The demand fetcher is activated by an active startfetcher signal of the instruction unit. When the demand fetcher is activated, the count signal and the quad pionter have to be set to the correct values. The following cases have to be distinguished:

- fetching both quads.
- fetching the second quad.

To be able to make this distinguish a signal is added to the interface between the fetcher and the server. The added signal is named StrtFetch2.

After finishing the fetch cycle, the status and the tag of the fetched transfer block has to be updated.

## F.4 The first signal merger.

**Module:** an operator, 3 functions.

**Name:** SignalMerger.

**Pin Constraints:**

Input: BlockDem, ValidDem, St1Dem, Tag1Dem, St0Dem, Tag0Dem, SetDem, WrCache0Dem, BlockPre, ValidPre, St1Pre, Tag1Pre, St0Pre, Tag0Pre, SetPre, WrCache0Pre, no name (control connector).

Output: Block, Valid, St1, Tag1, St0, Tag0, Set, WrCache0.

**Description:**

This module is implemented as a multiplexer, which selects between the signals of the prefetcher and the signals of the demand fetcher. The following cases are possible:

- The prefetcher is active and the demand fetcher is inactive. The multiplexer selects the prefetch signals.
- The demand fetcher is active and the prefetcher is inactive. The multiplexer selects the demand fetch signals.
- Both fetch modules are active. This happens when the prefetcher is stopped by an active startfetcher signal. The prefetcher has to update the status of the prefetched transfer block in the tag/status RAM. The multiplexer selects the tag/status RAM signals of the prefetcher and the interface control signals of the demnad fetcher. Mentioned interface is the interface between the cache and the bus unit.
- None of the fetch modules are active. The multiplexer selects the signals of the demand fetcher.

**Locical Description:**

Function Prefetching:

"The prefetcher is active and the demand fetcher is inactive."

WrCache0 := WrCache0Pre.

Tag0Out := Tag0Pre.

Tag1Out := Tag1Pre.
Status0Out := St0Pre.
Status1Out := St1Pre.
Valid := ValidPre.
Set := SetPre.
Block := BlockPre.

Function DemandFetching:

"The demand fetcher is active and the prefetcher is inactive."

WrCache0 := WrCache0Dem.
Tag0Out := Tag0Dem.
Tag1Out := Tag1Dem.
Status0Out := St0Dem.
Status1Out := St1Dem.
Valid := ValidDem.
Set := SetDem.
Block := BlockDem.

Function DemandPreFetching:

"Both fetch modules are active. This happens when the prefetcher is stopped by an active startfetcher signal. The prefetcher has to update the status of the prefetched transfer block in the tag/status RAM."

WrCache0 := WrCache0Pre.
Tag0Out := Tag0Pre.
Tag1Out := Tag1Pre.
Status0Out := St0Pre.
Status1Out := St1Pre.
Valid := ValidDem.
Set := SetPre.
Block := BlockPre.

**Control Connector Specification:**

"Both fetch modules are inactive or only the demand fetch module is active."

%10,%00 DemandFetching.

"Only the prefetch module is active."

%01 PreFetching.

"Both fetch modules are active. This happens when the prefetcher is stopped by an active startfetcher signal of the server. Before the prefetcher goes to its rest state, it has to update the status of the prefetched transfer block."

%11 DemandPreFetching.

## F.5 The second signal merger.

**Module:** an operator, 4 functions.

**Name:** Merge2.

**Pin Constraints:**

Input: QuadPointPre, EnabFetchPre, QuadPointDem, EnabFetchDem, no name (Control Connector).

Output: QuadPoint, EnabFetch.

**Description:**

This module selects between the quadpointer and enable fetch buffer signal of the prefetcher and those of the demand fetcher. The following cases can occcur:

- The prefetcher is active and the demand fetcher is inactive. The multiplexer selects the prefetch signals.
- The demand fetcher is active and the prefetcher is inactive. The multiplexer selects the demand fetch signals.
- Both fetch modules are active. This happens when the prefetcher is stopped by an active startfetcher signal. The prefetcher has to update the status of the prefetched transfer block in the tag/status RAM. The multiplexer selects the prefetch EnabFetch signal and the demand fetch QuadP signal.
- None of the fetch modules are active. The multiplexer selects the zero signals.

**Logical Description:**

"None of the fetch modules are active."

QuadPointer := 0.

EnabFetch := 0.

"Both fetch modules are active. This happens when the prefetcher is stopped by an active startfetcher signal."

QuadPointer := QuadPointDem.

EnabFetch := EnabFetchPre.

"The prefetcher is active and the demand fetcher is inactive."

QuadPointer := QuadPointPre.

EnabFetch := EnabFetchPre.

"The demand fetcher is active and the prefetcher is inactive."

QuadPointer := QuadPointDem.

EnabFetch := EnabFetchDem.

**Control Connector Specificaton:**

"None of the fetch modules are active."

%00 Default.

"The demand fetcher is active and the prefetcher is inactive."

%10 Demand.

"The prefetcher is active and the demand fetcher is inactive."

%01 Pre.

"Both fetch modules are active. This happens when the prefetcher is stopped by an active startfetcher signal."

%11 DemandPre.

## F.6 The third signal merger.

**Module:** an operator, 3 functions.

**Name:** CountMerge.

**Pin Constraints:**

Input: CountPre, CountDem, no name (Control Connector).

OutPut: Count.

**Description:**

This module operates as a multiplexer. The multiplexer selects between the count signal of the prefetcher an the count signal of the demand fetcher. The following cases are possible:

- The prefetcher is active and the demand fetcher is inactive. The multiplexer selects the count signals of the prefetcher.
- The demand fetcher is active and the prefetcher is inactive. The multiplexer selects the demand fetch signal.
- Both fetch modules are active. This happens when the prefetcher is stopped by an active startfetcher signal. The prefetcher has to update the status of the prefetched transfer block in the tag/status RAM. The multiplexer selects the count signal of demand fetcher.
- None of the fetch modules are active. The multiplexer selects a zero signal.

**Logical Description:**

Funtion Default:

"None of the fetch modules are active."

Count := 0.

Function Pre:

"The prefetcher is active and the demand fetcher is inactive."

Count := CountPre.

Function Demand:

"The demand fetcher is active and the prefetcher is inactive, or both fetch modules are active."

Count := CountDem.

**Control Connector Specifications:**

"None of the fetch modules are active."

%00 Default.

"The demand fetcher is active and the prefetcher is inactive, or both fetch modules are active."
%10,%11 Demand.
"The prefetcher is active and the demand fetcher is inactive."
%01 Pre.

## F.7 The present state register.

**Module:** a register, 2 bits.
**Name:** State.
**Pin Constraints:**
Input: none, no name (control connector).
Output: no name.
**Description:**
The content of this register represents the present state of the total fetch module. The state of the prefetch, demand fetch and the fetch controller depends o the value of this register. The following states are possible:
- rest state, indicated by the code 00b.
- prefetch state, indicated by the code 01b.
- demand fetch follow up state, indicated by the code 10b.
- demand fetch initialisation state, indicated by the code 11b.

**Control Connector Specification:**
"rest state."
%x0xx1001010 Setto:0. "old state:prefetching."
%x0xx1001100 Setto:0. "old state:demand fetching."
"demand fetch initialisation state."
%x1xxxxxx00x Setto:3. "old state:rest state."
%x1xxxxxx10x Setto:3. "old state:demand fetching."

%x1xxxxxx01x Setto:3. "old state:prefetching."

"prefetch state."

%100x000000x Setto:1.

"demand fetch follow up state."

%xxxxxxxx11x Setto:2.

## F.8 The control signals merger.

**Module:** an operator, 1 function.

**Name:** ContrMerge.

**Pin Constraints:**

Input: Ready, QuadPointer, State, Request, TrBlockHit, StartFetcher, CacheHit.

Output: C.

**Desription:**

This operator merges all control signals into one control bus.

**Locical Description:**

C := Request,StartFetcher,TrBlockHit,Ready,QuadPointer,State,CacheHit.

## F.9 The fetch controller.

**Module:** a state machine with 4 states.

**Name:** FController.

**Pin Constraints:**

Input: no name (Control Connector).

Output none.

**Description:**

This state machine is implemented to indicate in which mode the fetcher is fetching. Two ways of fetching are possible:

- prefetching: fetching quads, which are not yet required by the instruction unit.
- demand fetching: fetching quads, which are required by the instruction unit.

The output signal is implemented as constant generator, named DemandOrPre.

**State Description:**

Rest:

DemandOrPre Setto:0;

<<

DemF3:

DemandOrPre Setto:3;

-> DemF

DemF:

DemandOrPre Setto:2;

<<

PreF:

DemandOrPre Setto:1;

<<

**Control Connector Specification:**

"The specification of the control connector input:
Request,StartFetcher,TrBlockHit,Ready,QuadPointer,State,CacheHit."

%x0xx1001010 Goto:Rest.

%x0xx1001100 Goto:Rest.

%x1xxxxxx00x Goto:DemF.

%x1xxxxxx10x Goto:DemF.

%x1xx100101x Goto:DemF.

%x1xx0xxx01x Goto:DemF3."The prefetcher is stoppped by an active StartFetcher signal."

%x1xx100001x Goto:DemF3."The prefetcher is stoppped by an active StartFetcher signal."

%100x000000x Goto:PreF.

## F.10 The demand and prefetching indication.

**Module:** a constant generator.

**Name:** DemandOrPre.

**Pin Constraints:**

Input: none.

Output: no name.

**Description:**

This constant generator is used to generate the output signals of the state controller, named FController.