Design and implementation of an advanced instruction fetch unit for the MOVE framework

Steven Roos

Laboratory of Computer Architecture and
Digital Techniques (CARDIT)
Faculty of Electrical Engineering
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft,
The Netherlands

June 13, 1997

Delft University of Technology
Faculty of Electrical Engineering

Type	:	Master's Thesis
Number of pages	:	146
Date	:	June 13, 1997

Lab./Dept.	:	Laboratory of Computer Architecture and Digital Techniques (CARDIT)
Codenumber	:	1-68340-28(1997)07
Author	:	Steven Roos
Title	:	**Design and implementation of an advanced instruction fetch unit for the MOVE framework**
Supervisor	:	Dr. H. Corporaal
Mentor	:	Dr. Ir. I. Karkowski

Design and implementation of an advanced instruction fetch unit for the MOVE framework

Abstract

The main characteristics of the MOVE processor architecture are flexibility and scalability, which are exploited by the MOVE framework for the (semi-) automatic synthesis of application specific processors. The central unit of a MOVE processor is the instruction fetch unit, which reads instructions from memory and distributes them over the other parts of the processor. The instruction fetch unit is also responsible for a large part of the processor control.

A new design template for the instruction fetch unit had to be developed because previous versions lacked some important functions, like support for long immediates (constants in the instruction stream). Furthermore, the new instruction fetch unit should have support for an advanced, time-sharing operating system, which is a new topic in the MOVE framework.

The instruction fetch unit has successfully been developed. It has support for virtual memory, and for privileged instructions, which are two of the main requirements for operating system support. Some of the functions of the instruction fetch unit are implemented in two independent units. The state unit is responsible for some processor variables, and the immediate unit offers support for immediates. These units have been developed successfully too.

Contents

1 Introduction

Microprocessors can be found in all types of electronic equipment, not only in general purpose computers. Most processors are used in other equipment, like phones, television sets or microwave ovens. They are called embedded processors, and their main tasks are usually in the signal processing or control domain

Many of the embedded processors are application specific processors (ASP), which differ from the general purpose processors (GPP) because they are optimized for a small range of tasks. Application specific processors may therefore have a better price/performance ratio. Due to the rapid developments in computer technic, the advantages of ASPs are temporal. The price of the GPPs can be low due to the large production quantities. ASPs are only preferred if they are available before a GPP reaches the same level of price and performance. This is why a short design time is crucial.

The ultimate goal of the MOVE project at the Laboratory of Computer Architecture and Digital Techniques of the Delft University of Technology is to develop a framework for the automated design of ASPs. Starting from a high level description (C or C++) of an application, this framework should generate a processor which is optimized for the selected application with a minimum of user interaction. Extremely short processor design times can be reached with this framework.

The processor architecture for the MOVE framework is selected on characteristics like scalability, flexibility and modularity. The selected architecture, the MOVE architecture [Cor95], is a transport triggered VLIW architecture. Transport triggering removes the transport capacity bottleneck of traditional (operation triggered) VLIW architectures. Those VLIW architectures need too much transport capacity when many functional units (FUs) are added, because the transport network must be designed for worst case. By specifying the data transports instead of operations, the transport network is under full control of the compiler, which can use the transport capacity as efficiently as possible.

Within the MOVE processor architecture, a design template for the instruction fetch unit is needed. The instruction fetch unit is the central unit of the processor, which reads the instructions out of the external memory, and distributes them over all units. The instruction fetch unit is also responsible for the control flow instructions and exceptions. The design and implementation of an instruction fetch unit for MOVE processor is the main subject of this report.

This chapter gives a short introduction to the backgrounds of this thesis. Section 1.1 gives a short explanation of the goals of the MOVE project. Section 1.2 describes why a new instruction fetch unit is needed, and what the main requirements for this unit are. That section also gives a short introduction on the design trajectory. Finally, section 1.3 presents an overview of the contents of this report.

1.1 The MOVE project

To decrease to design time of application specific processors, the MOVE framework is being developed within the MOVE project. Starting with an application, described in a high-level programming language like C or C++, the framework generates the silicon layout of a MOVE processor which is optimized for the selected application. The framework also generates a compiler for the selected processor. The user interaction in the design path is kept as low as possible.

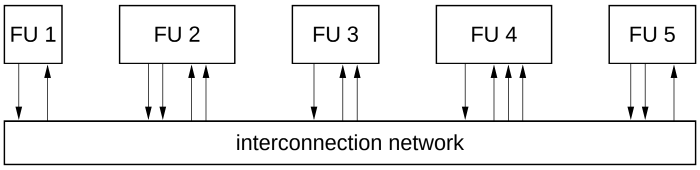


Figure 1.1: Organization of a TTA.

The MOVE processors are based upon a Transport Triggered Architecture (TTA). In this kind of computer architecture, bus transports instead of operations are programmed. Figure 1.1 shows a simplified picture of the organization of a TTA. It shows a TTA containing five functional units (FUs) and an interconnection network. Register units are considered FUs too. The processors are programmed by specifying a source FU port, and a destination FU port.

TTAs are very flexible, and have several parameters which can be adjusted to optimize the processor for its target application. There are no constraints on the number and types of FUs, and on the number of read and write ports on each unit. The interconnection network can be very simple, e.g. one bus, or can be very complex. Their scalability and modularity make TTAs particularly suited for automatically designed application specific processors.

A prototype of the MOVE processor has to be developed. Designing this processor has several goals:

- It shows that the MOVE concept is a realistic principle for processor designs. A working prototype proves that MOVE processors are a serious option for embedded computer systems. It also shows the performance capabilities of the MOVE architecture. Simulation gives a lot of information, but cannot prove everything. Especially timing behavior is very difficult for most simulators.

- The whole design path can be checked. Creating a fully functional processor will show all the problems in the design trajectory that cannot be expected on beforehand. Developing implies that placing and routing have to be done. The problems involved with this step can only be detected if the step is taken. For example, a choice has to be made between the software tools that can be used for the generation of the layout.

- A prototype will show how well the hardware and the software subsystems of the MOVE framework match together. This is where the previous prototypes [Cor93, Lam93] failed. These processors and their compilers both used their own interfaces and requirements, resulting in incompatibilities.

- If operating system (OS) functions like memory management are supported, an OS like Linux could be ported. This will show the usefulness of the MOVE principle for general purpose processors.

The demo processor should be placed inside a host computer on a special processor board. An interface has to be designed to provide programs for the processor and to read its responses from the host. After that, the processor can be tested. Eventually, an operating system may run on the processor. This is far too much work to fit in this thesis, but is required for a working prototype.

1.2 The design of the instruction fetch unit

The main task of an instruction fetch unit is to load the instructions for the processor from memory, and to distribute them over the rest of the processor. The instruction fetch unit is the central unit and is responsible for a large part of the global control within the processor. This makes it one of the most complex units.

Some instruction fetch units have been developed before, but they all have their disadvantages. The new design should aim at a better integration of the hardware and software subsystems of the framework. This includes support for long immediates in the instruction stream and for function calls. A new processor should have support for advanced (timesharing) operating systems, which is a new topic in the MOVE framework. This requires the implementation of a memory management system and the redesign of the exception system. The instruction fetch unit should not be designed for a prototype only. The unit has to be flexible, and useful for a large range of MOVE processor configurations.

Starting from the shortcomings of the previous versions, the instruction fetch unit is designed. The whole design trajectory is shown in figure 1.2. The list of requirements is generated from the intended use of the instruction fetch unit within the MOVE framework and the problems which were detected in the previous versions. For every subproblem, a number of possible solutions (preferably more than one) is worked out. The best solution for every problem is selected, and described in terms of datapath and control model, which are implemented in VHDL. This code contains parameters, by which the desired configuration can be chosen.

The first part of the design path, from architecture to implementation, is executed manually. The second part, from implementation to realization, should need as little user interaction as possible. First the collection of parameterized VHDL files is combined into one large VHDL file, in which all parameters are resolved. In the next steps, this design is worked out to a layout, which can be mapped onto silicon directly.

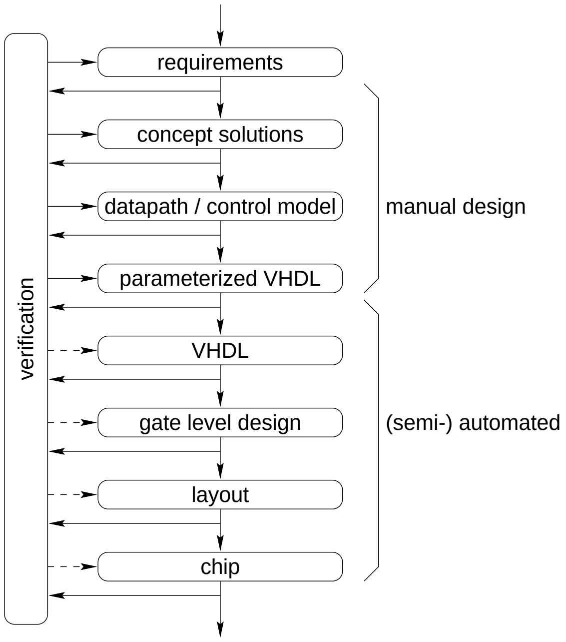


Figure 1.2: Design trajectory for the instruction fetch unit.

In parallel to the design path, the verification path is executed. Testing should guarantee that all the requirements which are formulated at the top of the design trajectory are met after the final step, when the processor chip is produced. As the second part of the design trajectory should need as little user interaction as possible, the feedback from the verification part should preferably only affect the first part of the design path.

This report will mainly deal which the upper part of the design path, from the requirements until the VHDL design. The design of the instruction fetch unit will be described as parameterized VHDL. The design is tested up to the non-parameterized VHDL level. A little synthesis is done, to make sure that the resulting VHDL file is accepted by the synthesizer, and to get some indications of the chip size and timing of the instruction fetch unit.

1.3 Report overview

The design trajectory is reflected in the structure of this report. An overview of the MOVE framework and architecture is given in chapter 2. This framework dictates the functions and the requirements of the instruction fetch unit, which are given in chapter 3. Chapter 4 describes how the requirements are worked out and implemented. Chapter 5 describes the automatic part of the design path. This chapter also describes the test methodology and results.

Those who need to work with the instruction fetch unit, but who are not interested in the exact implementation, are referred to chapter 6. This chapter can be regarded as a manual of the instruction fetch unit. Chapter 7 describes some extensions which can be used to optimize the instruction fetch unit and to add functionality. This report finishes with the conclusions and some recommendations.

The appendices give the implementation issues which have a too detailed nature to describe in the normal text. A complete list of all instructions is presented in appendix A. Appendix B gives the complete transition table for the finite state machine which controls the instruction cache. The parameters in the VHDL code of the instruction fetch unit are described in appendix C. The structure of the VHDL modules, and how they can be combined to a complete VHDL description, is described in appendix D. Appendix E gives the contents of all VHDL source files. Finally, appendix F described a way to generate a components package out of a VHDL description, which is needed to complete the VHDL code after the parameter references are resolved.

Overview of the MOVE framework

2

The functions and requirements of the instruction fetch unit are determined by its environment: the MOVE framework. Therefore a short overview of the MOVE framework and the MOVE processor architecture is presented here. The information in this chapter can be found in several other publications. A very complete overview is given in [Cor95]. The software subsystem is explained extensively in [Hoo96].

First, the MOVE framework itself is described. The hardware subsystem of this framework generates the MOVE processor, which is constructed as a Transport Triggered Architecture (TTA). The main component of the software subsystem is a compiler for this TTA processor. The MOVE processor architecture and the TTA principle are explained in the second section. This section also describes the timing of the MOVE processor and its functional units.

2.1 The MOVE framework

Application specific processors (ASP) have only temporal advantages above general purpose processors (GPP). Due to the rapid developments in computer architecture, a GPP may offers same performance as an ASP at a comparable cost. The cost of GPPs can be low due to the large production quantities. ASPs are only preferred if they are available before a GPP reaches the same level of performance and price. A short design time is therefore critical.

The MOVE framework is an automated design framework, which offers this short design time. It is a set of software tools by which an application specific processor can be developed semi-automatically. The necessary compiler is developed in parallel. As the framework hardly needs any user interaction, the design time can be very short. The development of this framework is the topic of the MOVE project.

The processors which are generated by the framework are based on a transport triggered architecture: the MOVE architecture. The scalability of this architecture allows that the processor configuration can be optimized for a selected application. Furthermore, the flexibility and simplicity of the MOVE processor architecture allow functions specific to the program in question easily to be integrated with the framework to enhance performance. The task of the MOVE framework is to exploit these advantages.

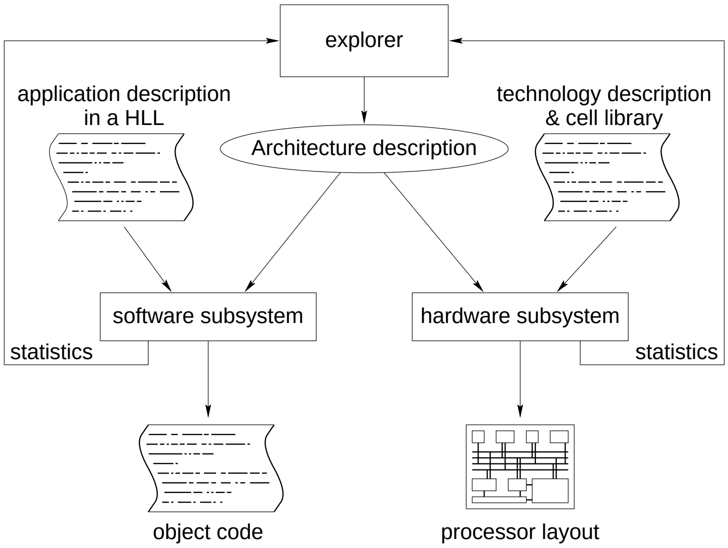


Figure 2.1: The MOVE framework.

The framework consists of a *hardware subsystem* and a *software subsystem*, which are used by an *explorer* to explore the architecture design space, depending on several parameters. By varying several architecture parameters, such as number of transport buses, number and type of function units, etc, the explorer tries to find the processor configuration that yields a good cost/performance ratio for the given application. The MOVE framework components are described in following subsections.

2.1.1 Explorer

Two main design evaluation criteria of the explorer are cost and performance, where performance is defined as the inverse of execution time. Cost may include the amount of chip area, the number of pins, the power dissipation, and the code size. Execution time is dependent on the number of executed operations, latencies, cache misses, and the clock cycle time.

The solution space is given by all possible design points in the 2-dimensional cost-performance space, that can be generated by the MOVE framework. Figure 2.2 shows many generated solutions for a particular application. Each marked solution corresponds to a particular MOVE configuration. As shown in this figure, the solution space is bounded by a curve connecting so-called Pareto points. These points are local optimal solutions for a certain cost or performance.

The explorer finds its way through this solution space by iteratively trying different architecture solutions, and letting the software and the hardware subsystems produce relevant information

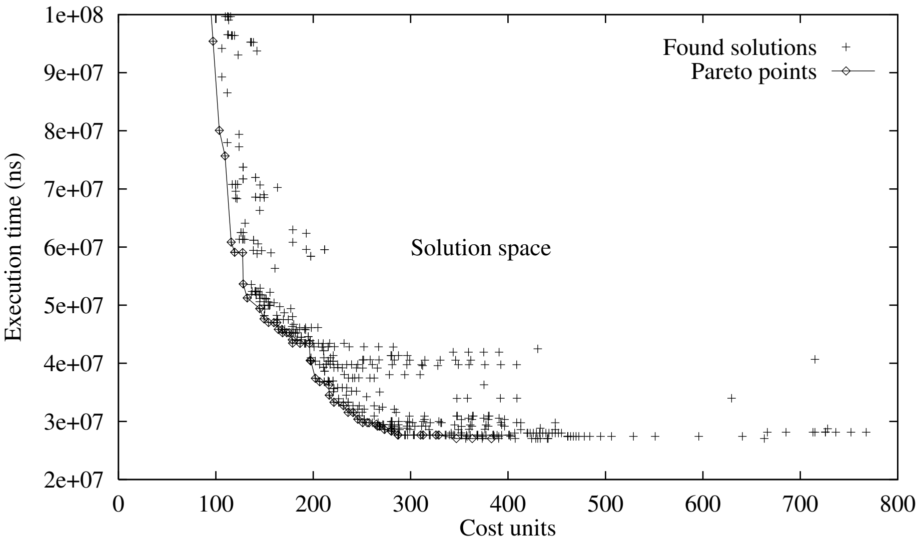


Figure 2.2: Possible solutions and Pareto points.

about these solutions, like cycle time, cost and the number of cycles needed to run the application. Based on this information a next design point is chosen by modifying the parameters. The user has to specify an initial (large) configuration from which the explorer starts the exploration process. The explorer does not add resources which were not available in this initial configuration. So if special FUs are used, one or more of them should be part of the initial configuration.

2.1.2 Hardware subsystem

The hardware subsystem of the framework is responsible for the realization of a MOVE processor in silicon. Figure 2.3 shows its organization. It is based on three components: a processor generator, a silicon compiler, and a hardware modeler. The *processor generator* accepts an architecture description and generates a description in VHDL, in which all parameter dependencies are resolved. This VHDL description can be used as input for many available silicon compilers and other synthesis tools. The *silicon compiler* takes the VHDL description, the technology information and a cell library as input, and produces a VLSI layout of the generated processor.

The *hardware modeler* is a tool which can quickly make area and timing estimations. This is required in order to be able to research many different architecture descriptions during the synthesis process. The hardware modeler exploits the fact that during the synthesis process many configurations are researched which only slightly differ from each other. The timing and area information of unchanged parts is kept in a database for quick access.

2.1.3 Software subsystem

The software subsystem of the framework is responsible for the generation of the object code for the selected MOVE processor. It contains a compiler, simulators for sequential and parallel code, profiling and trace analysis tools, and code viewers. The main component is the compiler, which is responsible for generating instruction level parallel code for the given MOVE processor,

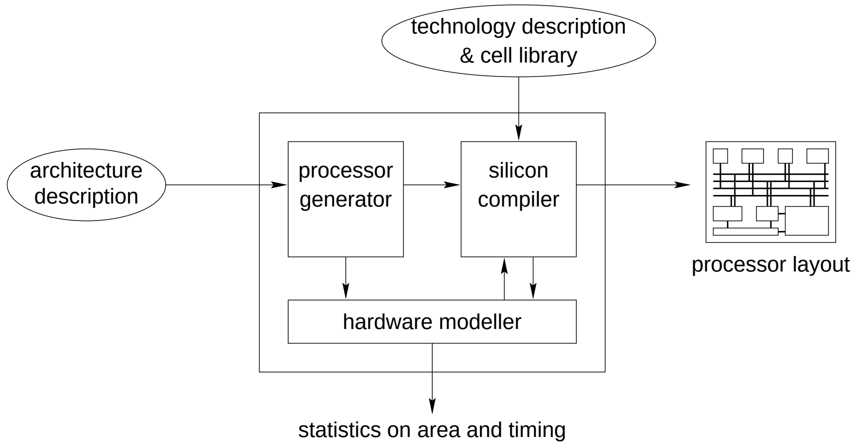


Figure 2.3: The hardware subsystem.

and producing statistics about cycle counts and usage of hardware resources. The compilation trajectory is shown in figure 2.4.

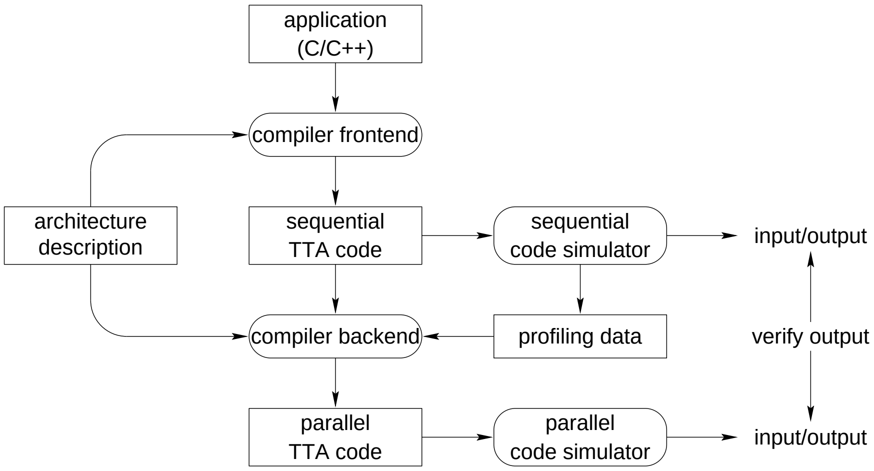


Figure 2.4: The TTA compiler trajectory of the software subsystem.

The compiler front-end is based on GNU gcc. The output of this compiler is sequential TTA code. The *sequential code simulator* may simulate this code to generate profiling data. The back-end may use the profiling information when it schedules the sequential code to produce parallel (scheduled) TTA code. The *parallel code simulator* can verify the correctness of this code by comparing the results with the output of the sequential code simulator.

The compiler back-end maps the sequential TTA code onto the available hardware of the target MOVE processor. It exploits advanced inter basic block scheduling and software pipelining techniques in order to compact the code as much as possible and to efficiently exploit all the available hardware. It generates code under the assumption that the target processor provides sufficient functionality. E.g., when the compiler front-end generates *shift* instructions, the hardware needs a *shift* unit because otherwise the scheduler cannot generate correct code.

2.2 Architecture of a MOVE processor

The core of the move processor is formed by the transport busses. Sockets connect these busses to the functional units (FUs), which are responsible for the data operations. Figure 2.5 shows a picture of a MOVE architecture. It contains four transport busses. Every socket connects one of FUs with one or more busses. There are input and output sockets, and every FU has one or more sockets. Not indicated in the figure is the fact that one or more FUs have connections to the outside world (off chip connections).

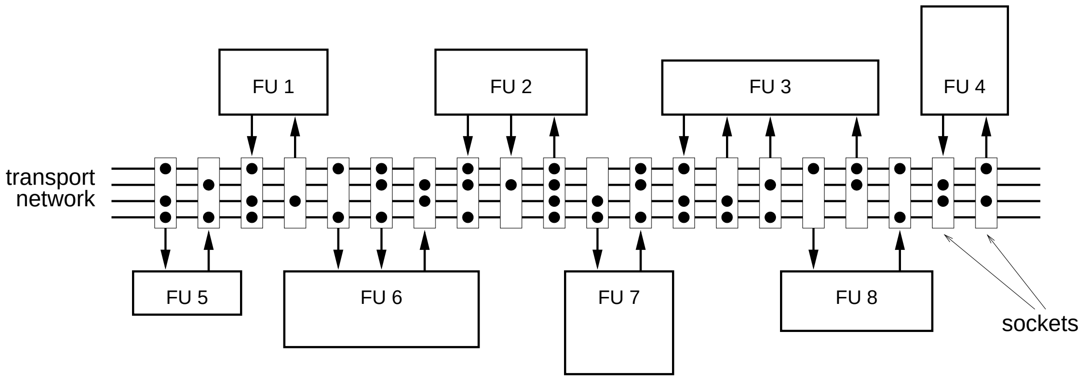
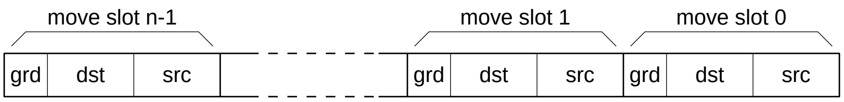


Figure 2.5: Transport network of the MOVE processor

2.2.1 The TTA principle

The processor is programmed by specifying the data transports over the transport busses. The only type of operation which is supported, is the *move*. Instructions specify the (virtual) source register and the (virtual) destination register for every move. Several virtual registers may map onto one physical registers. An instruction for a TTA with n busses may have the following format:



Each move slot specifies one move from source to destination. The guard field is used for conditional moves. If the guard expression specified by the guard field evaluates to *false*, the corresponding move is *squashed*, i.e. not executed. The data operations are executed as a side-effect of moves. One RISC-like operation typically requires three moves, e.g.:

```
add r3,r2,r1  ⇒  r2 -> add_o; r1 -> add_t
                 add_r -> r3
```

The notation used is: `add_t` and `add_o` are the trigger and operand registers of the adder, and `add_r` denotes the result (output) of this adder. The difference between a trigger and operand is that a trigger starts a new operation, whereas the operand does not. The precise difference is also influenced by the used timing discipline.

The reason this is called a Transport Triggered Architecture (TTA) originates from the fact that not operations, but transports are programmed. In Operation Triggered Architectures (most well known processors), the operations have data transports as a side effect, but with TTAs, transports have operations as a side effect. The most important advantages of TTAs are:

- The compiler can exploit optimizations like bypassing, operand sharing and dead result move elimination [CH95].
- The design is scalable and has a very modular structure. The transport bandwidth and the number of FUs can be designed independently. Adding a new FU type is very easy.
- The connectivity can be lower. The TTA needs less general purpose registers and less ports to the register file. This removes one of the major bottlenecks of (operation triggered) VLIWs
- The cycle time can be very short. The global pipeline and the internal pipelines of the FUs can be designed independently. The cycle time does therefore not depend on the FUs.

TTAs can easily be made fully programmable. This requires support for control flow operations and conditional execution. Control flow operations can be implemented by making the program counter a visible source and destination location of the instruction fetch unit. Conditional execution can be supported by guarding move operations; all data transports then become conditional on a boolean expression.

2.2.2 Timing of the MOVE processor

Pipelining is an important mechanism that exploits parallelism to reduce cycle time. It is also used in MOVE processors. The general MOVE pipeline can be divided into three stages:

Instruction fetch (IF) In the IF stage, the instruction fetch unit fetches the next instruction from the cache or memory (upon a cache miss)

Decode (DC) During this stage the source-, destination- and guard-IDs are distributed to the sockets, and decoded locally.

Move (MV) During the move stage the actual data transport takes place. As a result, operations in FUs may be started (after a trigger move).

Though other pipeline schemes are possible too, the instruction fetch unit from this thesis only has to support two and three stage pipelines. In the two stage pipeline, the *decode* and the *move* stages are combined, while all three stages are separate in the three stage pipeline. The two stage pipeline scheme is shown in figure 2.6. Figure 2.7 shows a three stage pipeline.

Figure 2.8 shows a more detailed picture of the two stage pipeline. The bars show which actions take place at which point in the pipeline. All timing data given here are approximations. As the decode and the move stages are combined, the critical path is most likely in this stage. After the instructions are distributed and decoded in the sockets, the data transports must be executed in the same cycle. The functional units are allowed to execute their functions during the instruction transport and decode. If this is not enough, they can be pipelined.

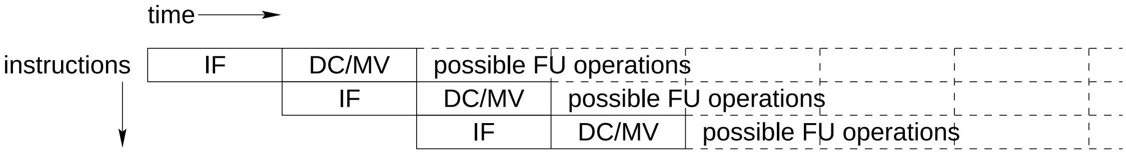


Figure 2.6: A two stage pipeline.

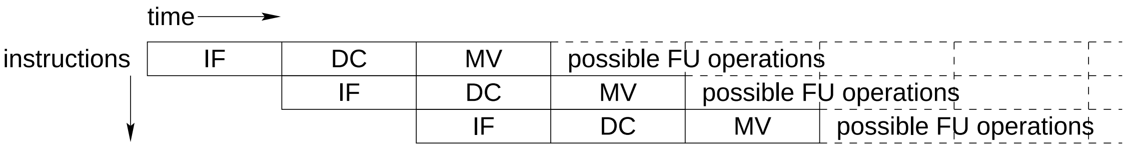


Figure 2.7: A three stage pipeline.

Figure 2.9 shows a similar picture for the three stage pipeline. As the decode and the move both get a pipeline stage on their own, the critical path is now probably somewhere else. The cache access may now become a limiting factor. One of the FUs, or the guard-squash path may also determine the minimal cycle time. Most FUs will need a result register, and jumps may get an extra delay cycle, which is clearly a disadvantage of the three stage pipeline.

All pictures show the virtual time horizontally. The virtual time proceeds at every clock cycle when the processor is not locked. During a lock, the whole processor is waiting, except the unit that requested the lock. Locks may be caused by cache misses and some exceptions.

2.2.3 Structure of functional units

In a MOVE processor all calculations and data operations are executed by functional units. Figure 2.10 shows a general picture of a functional unit. It has zero or more operand registers and a trigger register. If the FU can execute more than one function, it has also an opcode register, which specifies the intended operation. E.g., in an integer unit, the opcode register may select the

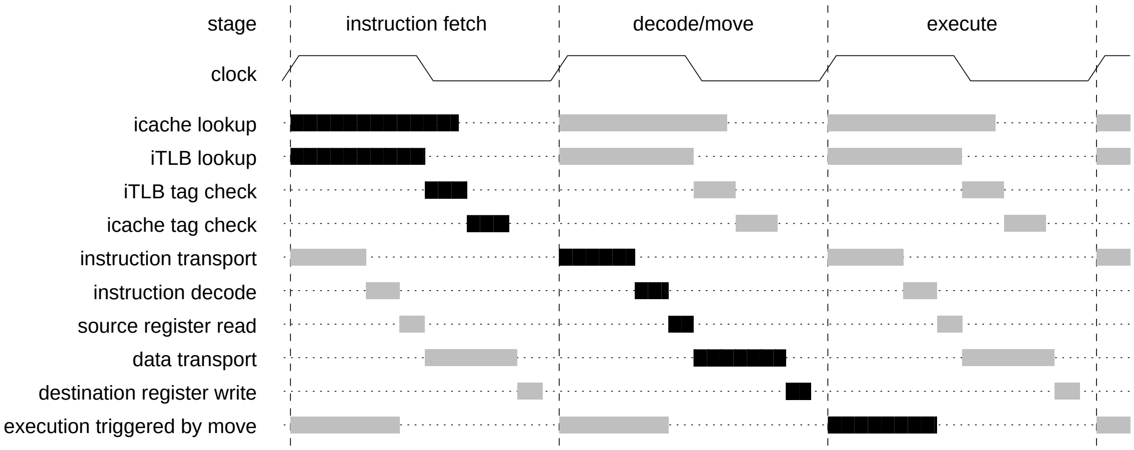


Figure 2.8: The timing of a two stage pipeline.

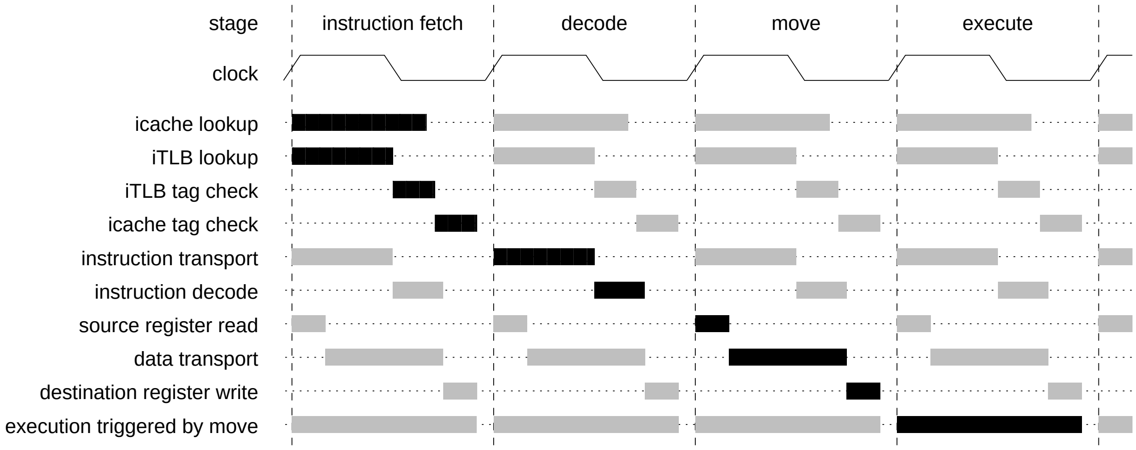


Figure 2.9: The timing of a three stage pipeline.

add or the subtract function. Some of the destination ID bits are written into the opcode register when the trigger register is written. The FU has therefore several virtual trigger registers (one for every function), which all map onto the same physical register.

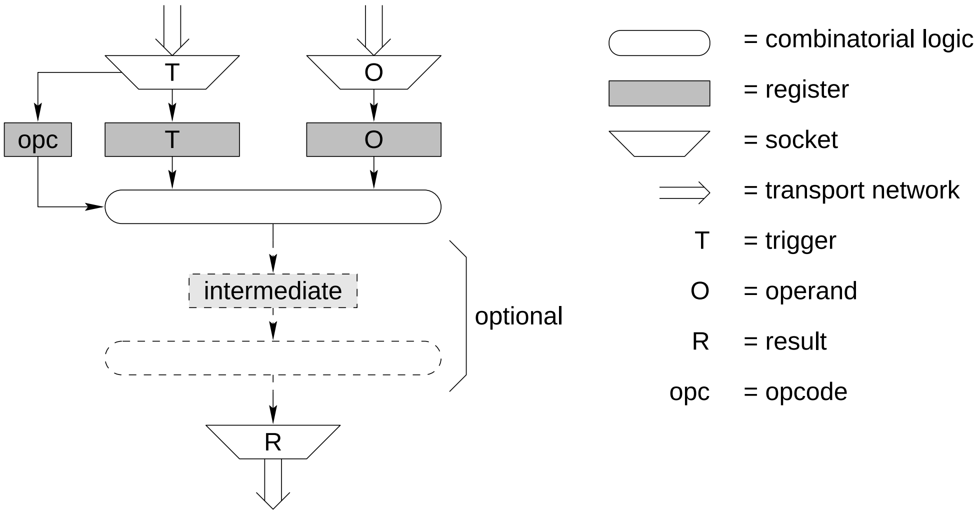


Figure 2.10: General structure of a functional unit

If the functional unit is internally pipelined, it has one or more intermediate pipeline stage registers. This pipeline runs synchronously with the instruction transport pipeline when virtual time latching (VTL) is applied. True virtual time latching (TVTL) means that both operand and trigger moves start an operation. With semi virtual time latching (SVTL), only trigger moves can start a new operation. For two stage transport pipelines, TVTL is the easiest scheme to implement. When a three stage transport pipeline is used, the result must be available very fast. Therefore often a result register (an intermediate stage register, with no extra combinatorial logic) is added. SVTL is then also possible.

The state of a single cycle FU, (i.e. not pipelined) can usually easily be saved and restored at exceptions. The result is available directly, and the operand register can be read when an identity operation (e.g. add with zero) is available. The state restore function has to write the operand register, and then trigger an operation, such that the result is also restored. This mechanism only works with TVTL FUs if the compiler does not exploit the fact that operations can also be started by writing into the operand register.

State saving and restoring for pipelined units is more complex. As the pipeline always proceeds, the results must be saved directly after the exception, and restored just before the return from exception. This requires at least one MOVE bus for each internally pipelined FU, which is usually not available. Other options are the usage of a push/pull mode, or busflashing.

3 Instruction fetch unit requirements

The main task of the instruction fetch unit is to load instructions from external memory and to distribute them over the processor. Of course, there are more functions which should be supported:

- Jump, branch and function call instructions are necessary.
- It should be possible to specify immediates in the instruction stream.
- A cache is needed to speedup the memory access. External memory is usually much slower than modern processors. A cache may help to avoid this latency.
- Exceptions must be supported. This provides programs an easy way to handle some unusual conditions without constantly checking those conditions.
- Virtual memory is required when an advanced operating system is running on the CPU. It provides tools to make memory management easier and to protect memory parts from usage by other programs. Its first property makes virtual memory also useful for processors not running an OS.
- Some of the instructions must be *privileged*. This means that they cannot be executed by user programs. They are restricted to the operating system, which runs in *kernel* mode.

The instruction fetch unit has to meet some non-functional requirements too:

- It must fit in the MOVE framework. This framework defines the interfaces to other parts of the processor. As a result, the instruction fetch unit has to be implemented in VHDL, and the macro processor tool m4 may be used for parameterizations.

- It should reach a high performance at a minimal cost (chip area and power consumption).
- The design must be customizable. When some users don't need all functions, they should be able to leave them out to gain area and performance.
- The implementation should be maintainable and expandable. Future designers might want to add or modify some functionalities. This should be possible without rewriting the whole design.
- The design should be testable. Making a real prototype is very expensive. You have to be sure the design is correct before you send it to the manufacturer.

This chapter will describe all requirements in more detail.

3.1 Jumps, branches and function calls

Jumps are control flow instructions which change the program counter (PC). Branches and function or procedure calls and returns are special forms of jumps. Branches are conditional jumps. A function call is a jump at which (at minimum) the return address is saved somewhere. The function return uses that return address as its jump target. The instruction fetch unit must be able to execute all these sorts of jumps.

In pipelined processors, jumps may cause serious performance loss. When a jump instruction is loaded, it is usually recognized as a jump after the next instruction is fetched. The instructions that are fetched incorrectly, will have to be discarded, and the correct instruction must be loaded. In that time, the processor is doing nothing. For a three stage MOVE processor, this would result in two stall cycles:

jump instruction	IF	DC	MV					
jump target		*stall*	*stall*	IF	DC	MV		
jump target + 1			*stall*	*stall*	IF	DC	MV	
jump target + 2				*stall*	*stall*	IF	DC	MV

This has a great effect on the performance because (conditional) jumps are frequent instructions. The instruction fetch unit has to provide means to reduce the effect of these *control hazards*.

The easiest way is adding delay cycles to every jump. The previous version of the MOVE processor [Lam93] used this principle. The execution cycle of a jump with a delay of length n is:

i	jump instruction
$i+1$	delay slot 1
$i+2$	delay slot 2
$\vdots$	
$i+n$	delay slot n
j	jump target

The job of the compiler is to fill the delay slots with useful instructions. This is not always possible. Therefore, the number of delay slots should be kept as low as possible. Another reason to keep that number low is because at an exception multiple PCs (one plus the length of the delay) are needed to restore the state correctly.

Usually, a jump is executed using an immediate as the target. As long immediates consume very much instruction bandwidth (or performance) and most jumps don't reach far, using short immediates should be possible, in addition to the normal, full sized jumps. Two approaches exist for this problem: *PC-relative* jumps and *page relative* jumps.

A PC-relative jump with n jumps n instructions forward. When n is negative, a backward jump is executed. When short immediates can be used to generate small positive and negative numbers, providing support for PC-relative jumps is sufficient. The disadvantage is that calculating the jump target requires a full adder, which is expensive and slow. It adds one cycle to every jump.

Page relative jumps assume that the PC is split up in a page number, and a page offset:

page number	page offset

A page relative jump just replaces the page offset. When the page size is matched with the size of short immediates, page relative jumps form an efficient solution. They hardly require extra hardware, and are relatively fast. The disadvantage is that the compiler cannot easily apply short jumps. Both the jump instruction and the jump target have to be on the same page. Whether this is the case is generally not known before the program has been linked. The linker has to do extra work.

3.2 Immediates

The previous version of the MOVE processor [Lam93] supported short immediates only. The source id fields of the move operations were available as data via the immediate unit. For practical configurations, the size of the immediates could not exceed 5 to 7 bits without significantly increasing the instruction word size. The immediate unit must be able to generate immediates of data word size ('long immediates'). This immediate unit may be a module in the instruction fetch unit, or may be an independent, separate unit.

The requirements for the immediate unit are:

- All possible data word sizes must be supported, even if they are very large. Several immediate sizes should be possible. As short immediates are more frequent, it is not efficient to use the space for large immediates when just a short immediate is needed.

- It must use the instruction bandwidth efficiently. It is unacceptable that a significant part of the instruction bits is reserved for long immediate generation only when they are rarely used.

- Loading a long immediate must not require much time. Otherwise, using several immediates short after each other would be too slow.

3.3 Cache

Modern processors are usually much faster than the external memory they are using. Reading one word from memory takes several CPU cycles. A cache may help to avoid this latency. When all instructions would have to be loaded from memory, the processor would spend most of its time waiting for the memory. Not implementing a cache would result in an unacceptable performance decrease.

The big disadvantage of a cache is that it consumes a lot of chip area. It is very important that the cache parameters can be adapted, to find a balance between cost (area) and performance (hit rate). Another way to reduce the cache size is to use special RAM cells. These are highly optimized memory structures, that make it possible to implement a large memory on a fairly small area.

It must be possible to enable and disable the cache. There are several reasons for this:

- It may increase the performance. When a procedure is executed only once, the old instructions are overwritten with useless ones that will not be used in the near future. When the cache would be disabled, the possibly useful old values are conserved.
- If (part of) the cache does not work correctly, the instruction fetch unit can still be used when the cache is disabled. This is why caches should be disabled upon reset.
- When the cache is disabled, all memory operations are visible on the outside. This may be useful when testing the chip.

The load-store unit also contains a cache [Baa96]. If possible, the instruction cache and the data cache should be similar. The control may differ, because instruction and data caches have slightly different requirements. A load-store unit can both read and write data, while an instruction fetch unit only reads instructions. The exception support mechanism in the instruction fetch unit may directly influence the behavior of the instruction cache, while the data cache does not need to know about exceptions.

3.4 Exceptions

The instruction fetch unit has to be able to support exceptions. For small, highly specialized applications, exception support may not be necessary. In general, when the application is more complex, or the processor tasks are less specialized, exception support can make the programmer's job easier.

Exception support allows the programmer to concentrate on the main task of the processor only. Handling rare conditions becomes much easier because checking for those conditions is not necessary any more. Those conditions are detected because they pull an external interrupt line. The interrupt handler then takes the necessary actions. An external interrupt line is required.

Splitting up an application in several independent tasks is possible using exceptions. A process switch is executed after an external interrupt (e.g. from a timer) or an exception requested by the program itself (*software trap*). Tasks like basic I/O operations or operating system functions can be implemented as a routine called after a software trap. Software traps must be implemented in the instruction fetch unit.

Exceptions are the base of an operating system (OS). When process switches are possible, and some tasks are implemented as exception routines, we have a very basic OS. When a fully featured OS (e.g. a ported version of Linux) must run on the processor, some additional requirements have to be met. They are described in section 3.5.

From now on, the term *interrupt* will be used to indicate an exception caused by pulling the external interrupt line. The term *trap* or *software trap* will be used to indicate an exception that is requested by a user program. The term *exception* will be used for exceptions in general, and when the type of the exception is not important.

Most exceptions will not terminate the running process. This means that the interrupted process must be restarted after the exception has been handled. This requires that the processor state at the moment of the exception can be saved, and restored later. According to Hennessy and Patterson [HP90], exceptions are precise if the pipeline can be stopped so that the instructions issued before the faulting instruction are completed and those issued after it can be restarted from scratch. This guarantees that the interrupted process can be restarted. Precise exceptions are required for some applications. For other applications, precise exceptions just simplify the software design. The instruction fetch unit should support precise as good as possible, though implementing precise exceptions in TTA processors is hard [Cor95].

Ideally, the faulting instruction does not change any processor state before the excepting condition is detected, and the pipeline is locked. When the faulting instruction writes its results before the exception can be handled, the hardware must be prepared to retrieve the source operands, even if they are overwritten by the same instruction.

To ensure the exception handler can do its job correctly, it must not be interrupted by new exceptions. The instruction fetch unit has to provide at least two different operating modes: exception mode and user mode. In user mode, exceptions are enabled, i.e. all exceptions are possible. In exception mode, exceptions cannot interrupt the processor because exceptions are disabled.

3.5 Operating system support

Support for an operating system is not a usual feature for application specific processors (ASP). There are however some good reasons why the processor should be able to support an OS:

- Implementing OS support forces the designer to look better at the whole design. It will learn the designers of both the hardware and the software about limits of the MOVE concept.

- In the future, an operating system like Linux might be ported to the MOVE processor. This will demonstrate that the MOVE processor can be used as a general purpose processor. It will also be possible to make a valid comparison between commercially available processors and a MOVE processor of comparable feature size, thus showing the usefulness of the MOVE concept for general purpose processors. It will show that the MOVE processor is especially interesting for embedded applications which combine general purpose and application specific functionality, like multimedia.

- The applications for which an ASPs are used, are growing more complex. The embedded processor usually runs more than one program. It becomes unacceptable that the whole processor crashes when one program makes an error.

- The need for a standard environment is growing. When the development environment, where a fully featured OS is available, and the target environment are the same, software design and testing will be much easier. The hardware will be more expensive, but the developing the software will be cheaper.

- Testing the software on the processor itself becomes easier. The operating system can provide some useful debugging functions. Checking for null pointer assignments is cheap when then the first memory page is not mapped. Programs can be tested on a specialized emulator too, but the emulator will be slower and less accurate than a real processor.

Though OS support is not needed for many current applications, it may become an important feature in the future. Therefore we accepted the challenge to implement OS support in the new version of the MOVE framework. Of course, operating system support should be seen as an option.

3.5.1 Operating system tasks

To be able to derive the requirements for operating system support, we must have a complete view of the tasks of an OS. The tasks of an operating system include:

- Initializing the processor after a reset.

- Basic I/O operations. The low level I/O operations like disk access and console functions are a task of the OS. The user programs may ask the OS for such operations by software traps.

- Executing process switches. A simple OS switches to the user program when it is started, and receives control again when the user program finishes. More sophisticated operating systems may start several processes together, and run them together using timesharing, thus providing a form of multitasking. A process switch will be executed after a timer interrupt (pre-emptive multitasking) or a software trap.

- Handling interrupts. The OS may eventually invoke a user program to handle the interrupt request (e.g. a driver for an I/O device).

- Handling software traps. Software traps are used to ask the OS to perform some tasks the user program cannot do itself. These tasks may be a process switch, or an I/O operation.

- Memory management. The operating system has to decide how the available physical memory is divided over all the processes. It also has to decide which actions should be taken when a process needs more memory than available or when a process tries to access data which belong to another process. A virtual memory system is a common memory management system which has some important additional advantages too.

- Protecting processes against each other. A user program should not be able to read or write the data of other programs. Sending a message to another program should be possible, though. The memory management system should ensure that a user process cannot read or write the memory which is in use by another process. The state of the processor which is necessary for the operating system to perform its tasks should not be accessible for user programs. When a user program tries to violate any protection, an exception routine should be called to avoid damage.

If precise exceptions are supported, the operating system can correctly do most of these tasks. The memory management and the protection tasks require some additional features.

3.5.2 Virtual memory

In a virtual memory system, every process gets its own, independent piece of virtual memory, which may have the size of the full address space. When a process tries to access a virtual memory address, a combination of hardware and software translates this address to a physical memory address, which is used to access main memory. This principle is shown in figure 3.1.

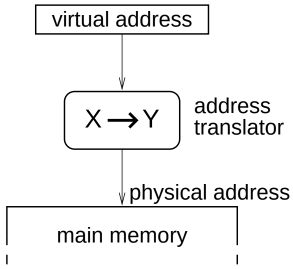


Figure 3.1: The principle of virtual memory.

Not every virtual address needs to have a translation. Usually, most of the virtual memory is not used, and therefore not allocated yet. Part of the data may be written to disk, to save physical memory space. When these data are needed, they are reloaded and another piece of memory is written to the disk if there is not enough physical memory available. The operating system decides which blocks are replaced.

Protection is easy in a virtual memory system. When we make sure that a virtual address does not translate to a physical address that is in use by another process, a process cannot access the data of other processes if it is not explicitly allowed to. When the program tries to access a virtual address for which no translation exists, an exception routine should be called, to let the operating system take the necessary actions.

Because a part of the virtual memory may be resident on a disk, all processes together may use more memory than which is physically available. Programmers need not make sure their programs entirely fit in main memory. This is the reason why virtual memory was invented, and why its usefulness is not restricted to processors running a complete OS.

The virtual address can translated by doing a lookup in a *page translation table*. With this system, memory is divided into *pages*, which are typically fixed at 512 to 8192 words. The page translations are kept in a page table. If this table is in memory only, every load or store would require two memory operations: one to access the page table, and one to access the requested data.

To prevent the double memory accesses, a *translation-lookaside buffer* or *TLB* must be implemented. The TLB is a cache dedicated to the page translation table. A TLB entry is like a cache entry where the tag holds the virtual page number and the data portion holds the physical page number.

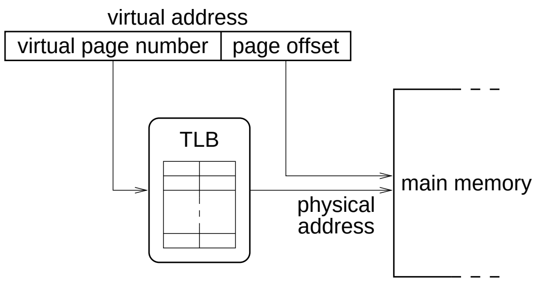


Figure 3.2: The principle of a paging virtual memory system using a TLB.

The TLB organization and size should guarantee a very low miss rate. As a TLB miss causes and exception, the miss penalty is extremely high. A large portion of the processor state has to be saved, which takes quite some time. It may be wise to disable the caches during the exception routine to prevent cache misses in the user program, but this will cause a cache miss at every instruction and every load or store.

3.5.3 Privileged instructions

One of the tasks of the operating system is to prevent user processes from influencing other processes when this is not explicitly allowed. A user mode process should not be able to access any data from other processes running on the same CPU, including the operating system. A portion of the information used by the OS is kept in the CPU itself, even if a user process is running. That information should not be accessible by user processes. Of course, a process should not be able to break down its own protection shell.

The registers and operations which are restricted to operating system use only are called *privileged*. The source and destination IDs connected to those registers are *privileged opcodes*. When running a user process, the CPU is in *user mode* (exceptions enabled, privileged instructions not allowed). When OS code is running, and privileged opcodes are allowed, the processor is said to be in *kernel mode*.

The kernel mode can be the same as the exception mode as defined in section 3.4, but then exceptions would not be possible while running OS code, which may be useful in many situations. There are two ways to allow exceptions while running OS code:

1. Running the interruptible OS code in user mode. This solution has two disadvantages. First, disabling the exceptions is only possible by generating a trap. The trap handler will start saving the process state, before it jumps to the right piece of code. This will take quite some time. Another option is to let the trap exception handler execute other code after running a user mode OS process. This is possible by either using another exception handler starting address, or by temporarily overwriting the exception handler code, but these solutions have disadvantages too. Second, the interruptible OS code cannot execute any privileged instructions. This will inhibit the execution of some OS tasks, like updating TLBs.

2. Splitting up the kernel mode into two separate modes. Exceptions are enabled in the first mode, and disabled in the second mode. The third processing mode, user mode, does not change. The main disadvantage of this solution is that the control is more complex. Executing privileged instructions when exceptions are possible may be dangerous. Privileged instructions may affect the exception support, e.g. writing to privileged registers which are used to save state at exceptions may affect the correct state saving. The processor has to make sure that executing privileged instructions just before an exception does not affect the correct behavior of both the exception routine and the interrupted program. Compiler conventions may relieve this constraint.

When a process illegally tries to execute a privileged instruction, an exception should stop the process before it can do any harm. Usually, the exception routine will terminate the offending process. However, in some special situations, it might be necessary to be able to restart that process. One of those situations is arises when the operating system is emulating a processor for another operating system.

3.5.4 Virtual processor emulation

An uncommon, but interesting task of the operating is to emulate the processor on which it runs as if the operating system were not present at all. This makes it possible to run another operating system on the virtual, emulated processor, as if it were a normal user program.

Support for virtual processor emulation is no requirement. It is hard to implement, and will rarely be used. On the other hand, it will ensure that the design is relatively clean, and looking at it will teach a lot about the possibilities of the MOVE architecture.

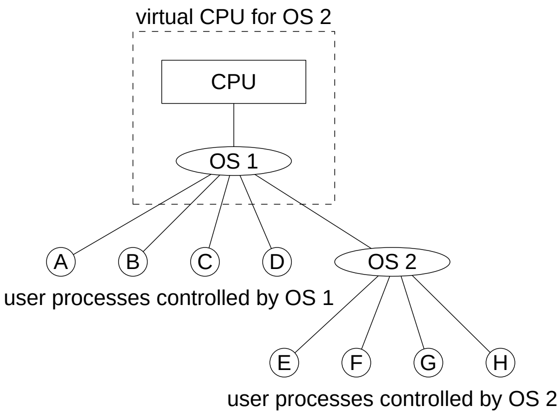


Figure 3.3: Emulating a virtual processor to run a second OS.

Figure 3.3 shows schematically how a virtual processor emulator works. The processes A, B, C, D and OS 2 are user mode processes which are controlled by OS 1. When OS 2 is the active process, OS 1 and the CPU together behave as a virtual processor for OS 2. The user processes E, F, G and H see only OS 2, and use the OS functions as provided by OS 2. OS 2 cannot see the difference between a real CPU and the virtual processor on which it is actually running.

Every exception caused by E, F, G or H will be intercepted by OS 1. The OS 1 exception handler may, depending on the exception source, handle the exception itself, or restart at the first

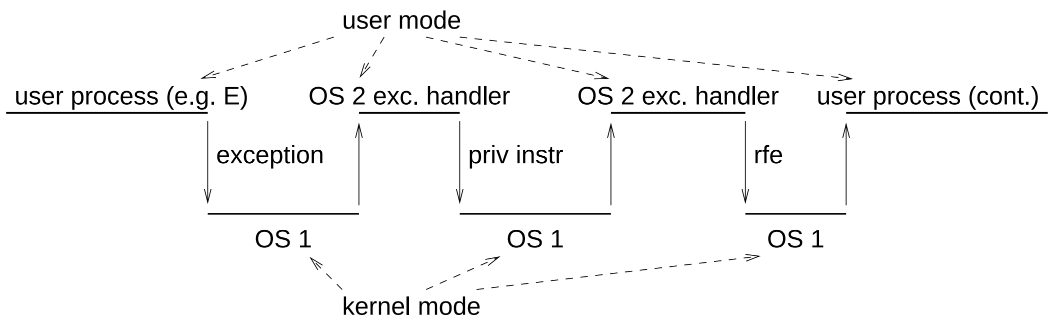


Figure 3.4: Procedure for handling exceptions caused by a user process which is under control of OS 2.

instruction of the OS 2 exception handler. When OS 2 tries to execute a privileged instruction, the exception routine emulates the effect of that operation in a way that OS 2 thinks its instruction was executed directly. This procedure is also shown in figure 3.4.

Every privileged register for OS 2 has to have an equivalent in memory, which is controlled by OS 1. OS 1 must emulate a write to that register by writing to the memory and taking care of the side effects, which are caused by writing to a trigger register. Reading that register must be emulated by reading the memory location.

Support for virtual processor emulation is not easy to implement in a processor design. It means that code which was intended to run with exceptions disabled, has to run correctly when exceptions are enabled. Every instruction that behaves different in user mode than in other modes, has to generate an exception when a user mode process tries to execute it. An instruction executed by OS 2 (user mode) should behave exactly the same as if this instruction would have been executed by OS 1 (exception mode). Otherwise, the instruction should cause an exception when OS 2 tries to execute it. Note that the load-store unit [Baa96] violates this rule. After an exception, it goes into a lock mode in which the internal pipeline is locked. The pipeline is restarted after the first read of the load result, or the first load trigger. In user mode, such a lock mode is not implemented.

3.6 Interfaces

The interfaces from the instruction fetch unit to the other parts of the MOVE framework are defined by the MOVE framework. A part of the framework was already described in chapter 2. This section describes the interfaces in more detail.

3.6.1 Transport network

The interface between the instruction fetch unit and the transport network is formed by *sockets*. The instruction and data busses are connected to the FUs via the input and output sockets. The optional signal socket provides signals which are not logically part of a data socket. Most FUs don't need a signal socket, but the instruction fetch unit does. Figure 3.6 shows the interfaces to the sockets.

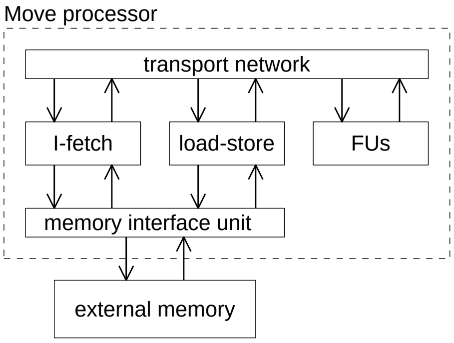


Figure 3.5: The interfaces of the instruction fetch unit.

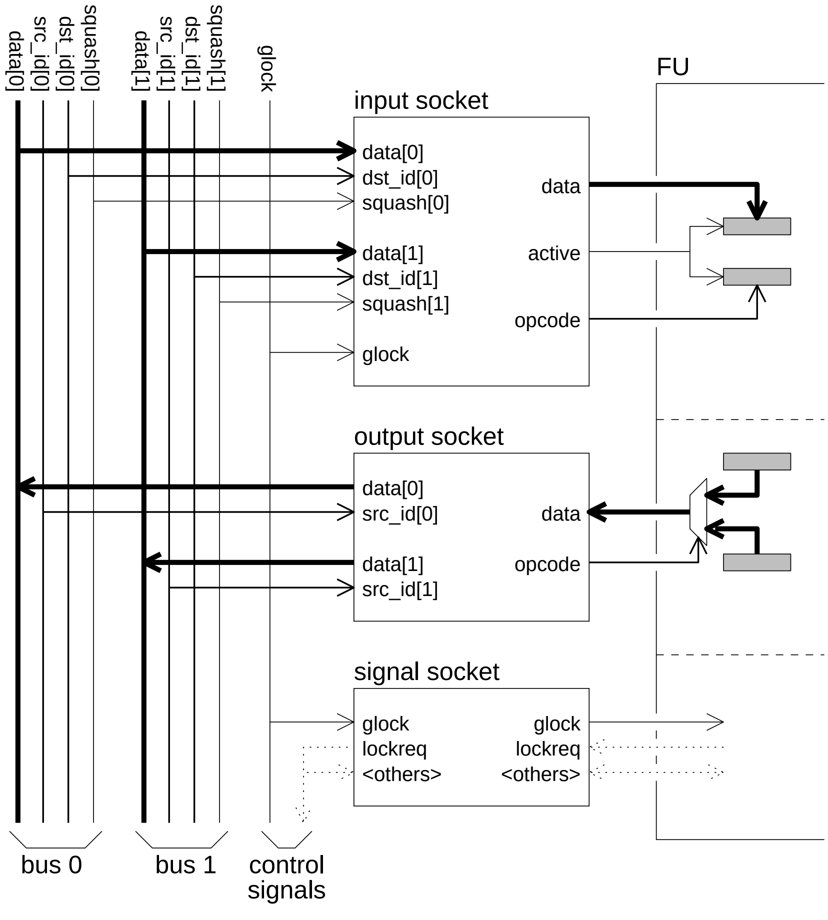


Figure 3.6: Overview of the interface to the transport network.

The control signals from the input socket to the FU are defined as:

$$select[i] = decode[i] \cdot \overline{squash[i]}$$

$$active = (select[1] + \cdots + select[n]) \cdot \overline{glock}$$

In these formulas, $decode[i]$ denotes that $dst\_id[i]$, the destination ID for bus i, is within the address range of the socket for bus i. When $select[i]$ is true, the data and opcode signals from bus i are passed to the FU. The compiler has to make sure that no more than one select signal is true.

The output socket works slightly different. The select signals are given by:

$$select[i] = decode[i]$$

In this formula, $decode[i]$ is true when $src\_id[i]$ is within the address range of the socket for bus i. If $select[i]$ is true, the socket sends the opcode from $src\_id[i]$ to the FU (if the FU needs an opcode), and puts the data from the FU onto the data bus. Otherwise, the socket data output to bus i is high impedance. The compiler has to make sure that the used opcodes are the same when one output is used by more than one bus.

The $squash[i]$ signals are not a part of the output socket equation. This is only allowed if a read operation does not change any state inside the FU, and one bus cannot be shared by several (mutually exclusive) result moves.

All signals from the input socket to the FU arrive late during the MV pipeline stage. The opcode signal from the output socket to the FU arrives early during the MV pipeline stage. The data signal of the output socket is expected to be valid very soon after the opcode has become valid. The FUs have to meet these timing requirements.

3.6.2 Memory interface unit

The instruction fetch unit and the load-store unit use a shared interface to external memory. The memory interface unit controls the external memory directly, or communicates with a (off chip) memory controller. A memory interface unit is described in [Baa96].

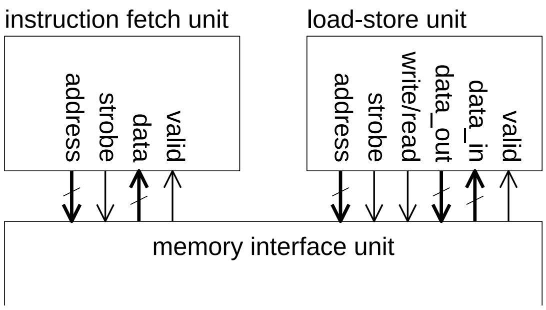


Figure 3.7: The interfaces to the memory interface unit.

The memory interface unit and the instruction fetch unit communicate via the signals shown in figure 3.7. The instruction fetch unit raises the *strobe* signal when it needs an instruction from memory. When the memory unit has loaded the desired instruction from memory, it raises the *valid* signal and puts the instruction on the *data* signals.

The signal *address* specifies the instruction address. As the instructions are subsequently numbered, the instruction address is usually not equal to the memory address of the instruction. When an instruction uses n memory words, the memory interface unit has to multiply the instruction address by n to get the memory address.

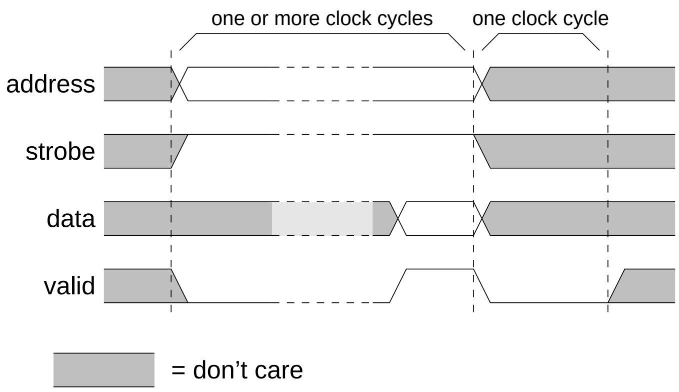


Figure 3.8: The timing of the interface to the memory interface unit.

The memory interface unit can respond to requests in the same cycle as it receives the *strobe* signal if memory is very fast. Usually, the instruction fetch unit will have to wait a few cycles before the instruction is available. The *address* has to be stable during that time. A memory access cannot be interrupted.

The instruction fetch unit can ask for a new instruction immediately after it has received the previous instruction. This gives a 100% utilization of the available memory bandwidth. The memory unit cannot respond immediately to the second request. The *valid* signal must be low for at least one cycle after it was high. As a consequence, the maximum memory access frequency is one instruction per two cycles.

3.6.3 Control signals

Both the instruction fetch unit and the load-store unit need some control input signals to turn on and off caches and TLBs. These signals should get default values at an exception, and should be restored to the old values at the return from exception. Because exceptions are a task of the instruction fetch unit, controlling the cache and TLB enabled signals is a task of the instruction fetch unit too.

A way to save internally pipelined functional units correctly is to lock them at an exception. The pipelined is then restarted at the first access. The load-store [Baa96] uses the exception enabled signal for this purpose. This means that the exception enabled signal should be available outside the instruction fetch unit.

3.7 General requirements

The design must meet some very general requirements. Unfortunately, most of these constraints are not specified well. They will be discussed in this section. The requirements include:

- minimizing the chip area
- reducing the power usage
- maximizing the highest possible clock rate

- keeping the design testable
- making the design maintainable and expandable
- offering enough possibilities to customize the design

The price of the design highly depends on the used chip area. Memory structures, especially normal flip-flops, and global wiring use very much area. The number of registers should be kept as low as possible, though it might cost some performance. Wiring cannot easily be reduced. Making as few connections as possible, and making sure that connected areas can be placed together may help.

When static CMOS circuits are used, the power consumption highly depends on the number of logic transitions per second. As reducing the clock rate is not acceptable, power consumption is very hard to control. Reducing state transitions is possible, but not easy.

To achieve the highest possible clock rate, the critical path should be shortened. Reducing the delay in a non-critical path has no effect, as long as the critical path is not affected. All optimizations should aim at reducing the critical path, even at the expense of more delay in a non-critical path. When using an 1.0 μm process and a 3 stage pipeline, 60 to 70 MHz should be possible [Arn95].

To keep the design testable and maintainable, the design should be as clear and neat as possible. When possible, the design should be split in several highly independent modules. Reducing the number of states makes a design more testable, and will make it more easy for other people to read the design. Good comments in the source files are needed too.

The MOVE-architecture is designed for a wide range of applications. This implies that the design will be adjusted to match the specific requirements for the application before it will be synthesized. All functions that might not be needed for all applications, should only be available as an option. All parameters which may have an significant impact on the chip area or the performance should be controllable. Both a 2 (IF-DC/MV) and a 3 (IF-DC-MV) stage pipeline should be possible.

4 Implementation of the instruction fetch unit

For every requirement from chapter 3 there are several possible implementations which meet that specification. All these solutions provide the desired functions, but differ in terms of cost, performance and testability. The final implementation is selected after a comparison on these criteria.

This chapter describes the design trajectory from the requirements to the implementation in terms of datapaths and control models. The implementation alternatives are given and compared. The selected implementation is explained in more detail. The exact VHDL structure and source codes are added in the appendices for those who want to know all the details. Appendix D describes the function of all (VHDL source) files and the dependencies between them. The contents of the VHDL source files are added in appendix E.

As shown in figure 4.1, the functions of the instruction fetch unit are distributed over three separate units:

- The immediate unit extracts the immediates out of the instructions, and places them on the data busses. It generates both short and long immediates.
- The state unit controls some variables which have a wider range than the instruction fetch unit only. It is responsible for the processor modes and the enabled signals for the instruction cache and TLB as well as the data cache and TLB.
- The third unit is the unit that remains after the state unit and immediate unit are split off. From now on, this stripped version of the instruction fetch unit will simply be called the instruction fetch unit.

In this chapter, the design of the instruction fetch unit is split in a datapath design (section 4.1), and a control model (section 4.2). The state unit is explained as a part of the control model, though most of the parameters it controls are not instruction fetch unit specific.

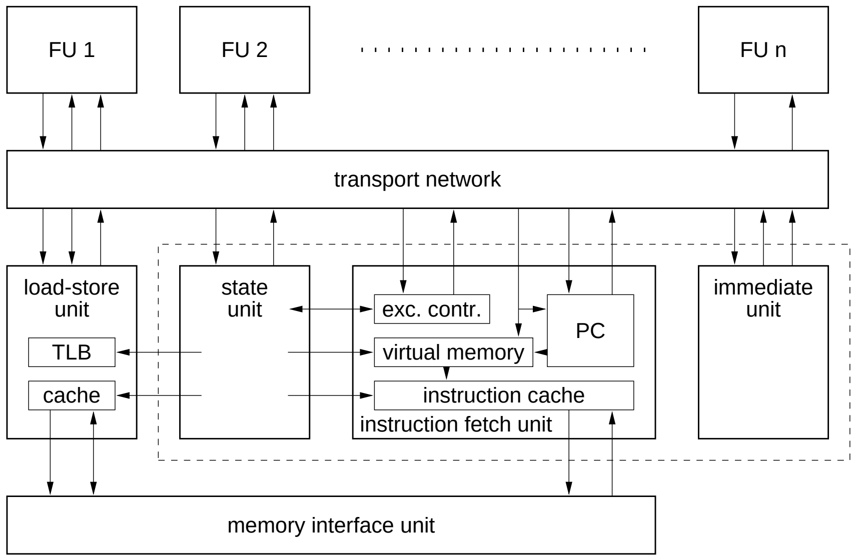


Figure 4.1: A processor environment with an instruction fetch unit, a state unit and an immediate unit, which are together responsible for the instruction fetch unit functions.

Within the instruction fetch unit the cache and TLB use the same generic cache template. The organization and size of this cache can be controlled by simple parameters. This cache may also be useful to the load-store unit, though the current implementation does not use it. Section 4.3 describes the interfaces and the implementation of the generic cache.

The selected way to generate long immediates allows that the immediate unit is separated from the instruction fetch unit. Section 4.4 explains this decision and describes the immediate unit.

4.1 Datapath model

The datapath of the instruction fetch unit consists of three modules, which is shown by figure 4.2. Module 1 is the program counter path which contains the program counter (PC), the PC selector, and the PC chain. The second module is the TLB, which is the main component of the virtual memory system. The third module consists of the instruction cache and the interface to the memory interface unit.

4.1.1 Program counter path

The program counter (PC) holds the number of the instruction in the IF stage of the transport pipeline. This instruction address is not necessarily the same as the memory address where the instruction is stored. First, the virtual memory system (section 4.1.2) may map the virtual address in the PC onto another, physical, address. Second, the memory interface unit may scale the instruction address if one instruction uses more than one memory word.

As the PC uses the instruction number, and not its corresponding memory address, the PC can be incremented by 1 every cycle. At exceptions, an exception vector is written into the PC.

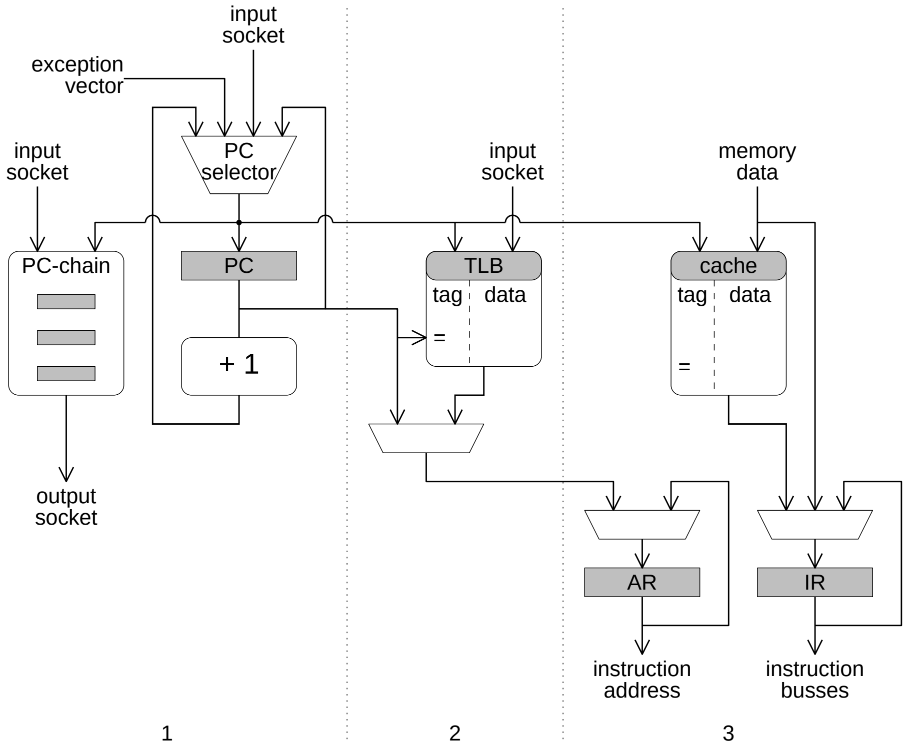


Figure 4.2: Datapath of the instruction fetch unit

The exception controller (section 4.2.1) determines which exception vector is used. If a jump instruction is triggered, the jump target is written into the PC. Conditional jumps are supported via the MOVE guard mechanism.

The instruction fetch unit uses the principle of delay slots to avoid the jump latency described in section 3.1. Another option would be to implement a branch predictor like branch target buffers. Such predictors are hard to implement and take quite some area. The performance is hardly any better than with delay slots, as long as the number of delay slots can be kept low (1 or 2). Therefore jump delay slots are implemented.

Another important issue is the choice between PC relative jumps and page relative jumps. With page relative jumps, the lower bits of the PC are just replaced by the jump operand. The operand is added to the operand when using the PC relative scheme. Both options need an absolute, full sized jump too. Figure 4.3 shows datapath implementations for both possibilities.

The main advantage of page relative jumps is that the jump can take effect immediately. The adder used for PC relative jumps is too slow to allow immediate PC relative jumps, which makes a jump register and an extra delay slot necessary. The extra delay slot causes an extra PC chain register[1]. The disadvantage of this construction for fast page relative jumps is the dependency between the next PC value and the input socket data. These data are usually available late during the cycle, and have to go through a number of selectors before they reach the PC register.

The last disadvantage is probably not severe because the instruction fetch unit is a central

[1]At an exception, the PC chain saves the addresses of the first instructions which must be executed after the return from exception

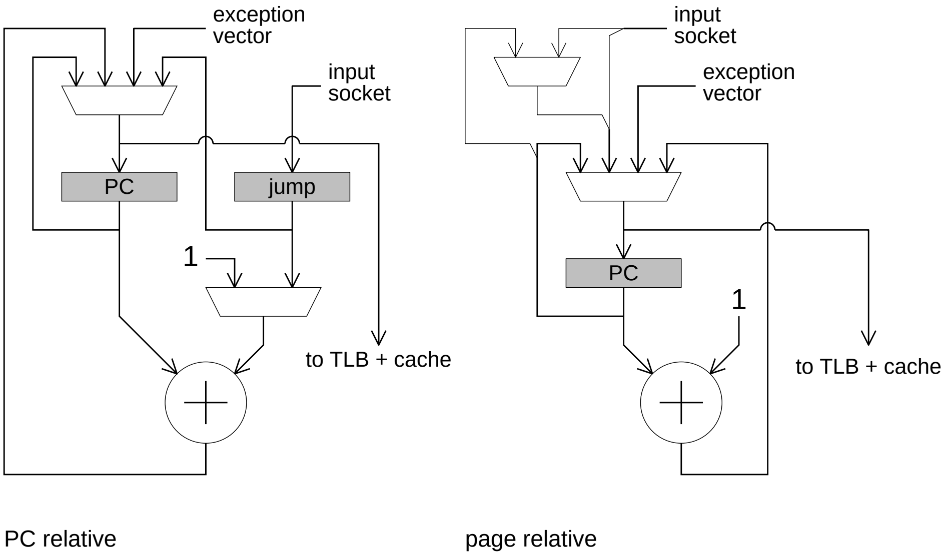


Figure 4.3: Datapaths for PC relative and page relative jumps.

unit, with short lines to all other units. The jump operand will probably arrive earlier than the data for other units. The extra delay cycle and the bigger hardware (two registers and a bigger adder) necessary for the PC relative jump compensate for the compiler difficulties using page relative jumps, so page relative jumps are implemented.

The implemented instruction fetch unit offers absolute jumps, which write the whole PC, and two different page sizes for page relative jumps. These page sizes are parameters which can be adopted to the available immediate bit sizes. Two different page sizes give more flexibility in scheduling jumps when immediates are used as the jump target than would be possible when only one page size would have been used.

At a function call, the return address has to be saved somewhere. The easiest way is to make the PC of the last delay slot plus 1 available via an output socket. This reduces the function call to a normal jump. The return address must be saved in a general purpose register by a second instruction. It is however better to save the return address automatically.

The page relative jump scheme from figure 4.3 can easily be extended to support such automatic return address saving. Note that the return address is already physically available at the time of the function call at the output of the incrementer. The return address register is connected there. This register is loaded at the same cycle the function address is written into the program counter.

At an exception, the PC has to be saved. As a jump takes effect $n-1$ cycles after it is ordered (using an n stage pipeline), the next n program counters must be saved. The PC chain contains these PCs. When exceptions are enabled (after a return from exception), the next PC is also saved in the first entry of the PC chain. The chain runs along with the pipeline. At an exception, the chain is locked, thus saving the first n instruction addresses to be executed after a return from exception.

In exception mode[2] the PC chain can be read and written. Only the PCC-MV register, which

[2]see section 4.2.2

contains the address of the first instruction which must be executed after the return from exception, can be read. Reading this register does not affect the chain. The only register that can be written, is PCC-IF, which contains the address of the n^{th} instruction after the return from exception. A write lets the chain proceed one cycle. If a three stage transport pipeline is used, the third (middle) PC chain register is called PCC-DC.

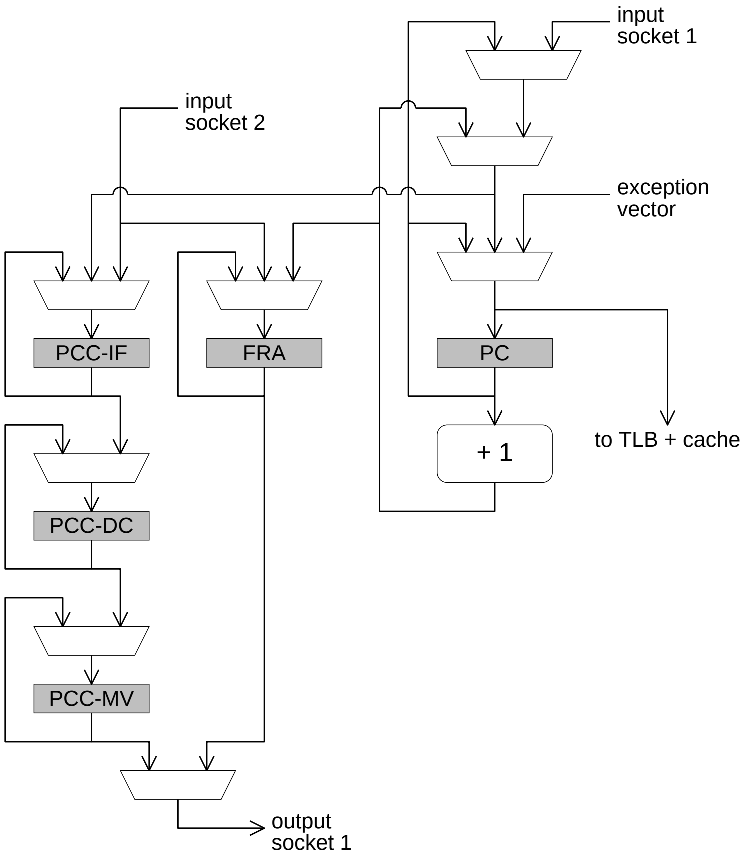


Figure 4.4: PC data path for a 3 stage pipeline. The PC chain register PCC-DC is removed in the 2 stage version.

The complete implementation of the program counter path, including the function address register and the PC chain, is shown in figure 4.4. This figure also shows all possibilities to read and write the registers. The tables below show all instructions which access one or more of these registers.

Input socket 1 functions:

opcode	*action*		*description*
000	$src_{(p-1):0}$	$\rightarrow pc_{(p-1):0}$	page relative jump; page offset: p bits
001	$src_{(p-1):0}$	$\rightarrow pc_{(p-1):0}$	page relative call; page offset: p bits
	$pc+1$	$\rightarrow fra$	
010	$src_{(q-1):0}$	$\rightarrow pc_{(q-1):0}$	page relative jump; page offset: q bits ($p > q$)
011	$src_{(q-1):0}$	$\rightarrow pc_{(q-1):0}$	page relative call; page offset: q bits ($p > q$)
	$pc+1$	$\rightarrow fra$	
100	src	$\rightarrow pc$	absolute jump
101	src	$\rightarrow pc$	absolute call
	$pc+1$	$\rightarrow fra$	
110	src	$\rightarrow pc$	return from exception

Input socket 2 functions:

opcode	*action*		*description*
0/00	src	$\rightarrow fra$	write the function return address
1/01	src	$\rightarrow pcc\_if$	write the PC chain
	pcc_if	$\rightarrow pcc\_dc$	
	pcc_dc	$\rightarrow pcc\_mv$	

Output socket 1 functions:

opcode	*action*		*description*
0	fra	$\rightarrow dst$	read the function return address
1	pcc_mv	$\rightarrow dst$	read the PC chain

The return from exception function behaves like a normal jump, but has the side effect that exceptions are enabled, the PC chain is restarted and the saved cache and TLB variables are restored. This is explained more extensively in section 4.2 about the control model. Input socket 2 can also be used by the TLB. If a TLB is included, the opcode for that socket has two bits, otherwise the opcode has only one bit.

4.1.2 Virtual memory system

One of the issues in designing virtual memory systems is to decide whether the caches should be addressed with virtual addresses, or with real addresses. Virtually addressed caches must be invalidated at every process switch because the old and the new process may use the same virtual address for different memory locations[3]. A TLB lookup followed by a cache lookup is usually too slow to fit in one clock cycle, so this option is also not acceptable.

The solution which is used in the instruction fetch unit uses the real addresses to access the cache, but does the TLB and cache lookup simultaneously. The page offset is not changed by the TLB, and can therefore immediately be used for the cache lookup. The translated page number is needed for the cache tag check. As the TLB is usually much smaller than the cache, the TLB lookup will be finished when the cache tag is checked. No time is wasted.

[3]The cache invalidating can be avoided with some tricks, like increasing the width of the cache-address tag with a process ID tag [HP90], but they all have their impact on performance, and may have other disadvantages.

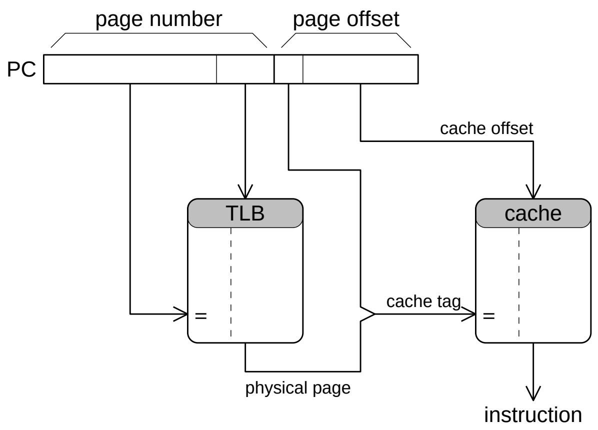


Figure 4.5: Simultaneous TLB lookup and cache lookup.

The number of cache addressing bits may not be greater than the number of page offset bits. This constrains the cache size. When a direct mapped cache is used, the cache size cannot be greater than the page size. When an n-way set associative cache is used, the cache size is restricted to n times the page size. The IBM 3033 used a 16-way set associative cache to be able to use a large, physically addressed cache, though miss rates hardly improve above 4-way set associativity.

The instruction fetch unit and the load-store unit may have their own TLBs. The other possibility is to create a shared TLB, which is used by both the load-store unit and the instruction fetch unit. The advantage of a shared TLB is clear: you need only one TLB, which saves area and OS overhead. The disadvantage is that the load-store unit and the instruction fetch unit cannot access the TLB simultaneously. A usual way to overcome this restriction is to implement local micro-TLBs in both units, which work like caches for the big TLB. But the area which was saved by sharing the TLB will be consumed largely by those micro-TLBs, and the control will be more difficult.

Separate TLBs are implemented because the area advantage of a shared TLB nearly vanishes when micro-TLBs are implemented, thus leaving the disadvantages. An extra advantage of the separate TLBs is that the implementation parameters for both TLBs can be chosen independently. The load-store TLB may have another size and organization and use another page size than the instruction fetch unit TLB.

Figure 4.6 shows the datapath of the virtual memory system. The generic cache template(section 4.3) is used for the TLB itself. It is accessed with the next value of the PC (new_pc) instead of the PC register value, because of the timing of the generic cache interface. The output of the TLB is used as the physical page number, which is used by the cache and the memory interface. When the TLB is disabled, the virtual page number from the PC register is directly passed to the output.

When a new TLB entry is written, the virtual page number must be written first. The TLB looks up this value, to set some values which are needed at a write. The page number is saved in the page register. The next step is writing the new page translation, together with a valid bit.

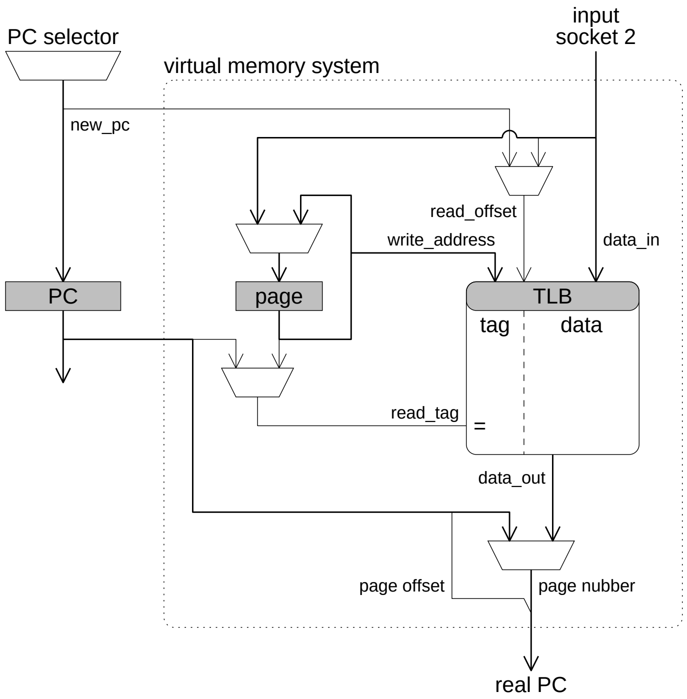


Figure 4.6: The datapath of the virtual memory system.

The written real page number is connected to the virtual page number in the page register at that moment.

When a TLB miss causes an exception, the missing instruction can be found in the PCC-MV register (see figure 4.4). This register can be written directly into the TLB when the virtual page address is written, because the page numbers must be aligned as if they are regular instruction addresses.

The number of bits needed for the virtual and the real instruction address need not be the same. The virtual address space can therefore be bigger than the physical address space. If the TLB is disabled, the most significant bits of the virtual instruction address are discarded.

The next table gives the instructions which are needed for the TLB. Input socket 2 means that the second input socket of the instruction fetch unit is used. When writing the virtual and real page numbers, the data must be aligned as if they were complete addresses. When using p bits for the page number, and q bits for the page address, the page number which is written must be on the bit positions q to $p + q - 1$. The valid bit has to be on position 0.

Input socket 2 functions:

opcode	*action*		*description*
10	src	$\rightarrow page\_reg$	write the virtual page number
11	src	$\rightarrow tlb\_data$	write the real page number
	$page\_reg$	$\rightarrow tlb\_tag$	

4.1.3 Instruction cache

The last part of the datapath is the real instruction fetching part. The physical instruction number is used to fetch the desired instruction from memory. The cache is a way to speed up this memory access. When the cache is enabled and the requested instruction is in the cache, the instruction is read from the cache, and not from memory. This gives a better performance because the cache is usually much faster than main memory.

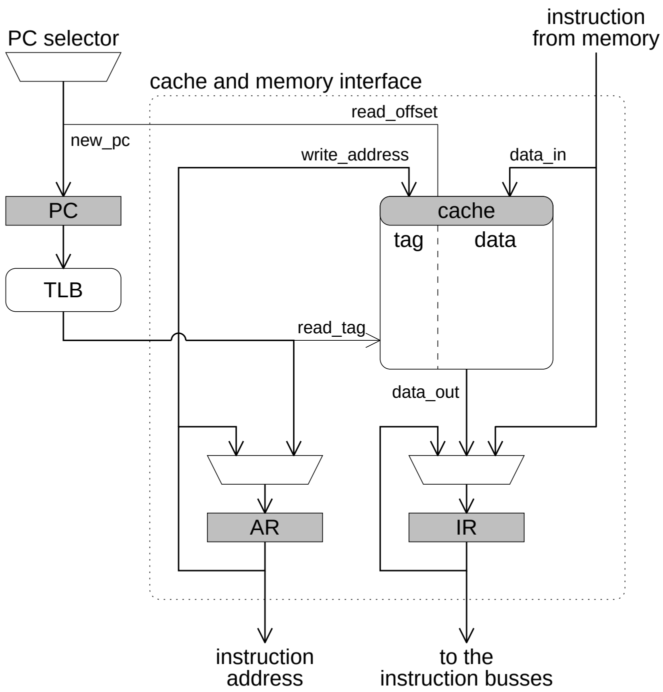


Figure 4.7: The datapath of the instruction cache and the memory interface.

Figure 4.7 shows the datapath of the instruction cache and the interface to the memory interface unit. The instruction register (IR) holds the instruction during the decode stage. This means that the instruction on the instruction busses is valid from the beginning of the decode cycle, which is especially important for two stage pipeline systems. As the decode and move stage are combined in these systems, the decode logic will most likely be part of the critical path.

The address register (AR) holds the instruction address. This address is not necessarily the same as the memory address, because the memory interface unit may scale the instruction address if one instruction uses more than one memory word, or if one memory word contains more than one instruction. This unit assumes that the instruction address is stable during the complete memory access. The register is not overwritten until the interface unit has returned the requested instruction.

The generic cache as described in section 4.3 is used for the instruction cache, just as it was used for the TLB. The generic cache is highly parameterizable, and all configurations which are supported by the generic cache are available for the instruction cache. The generic cache provides set associativity and subblocking.

The generic cache access time is one cycle. This means that when the cache address is stable just before the rising clock edge, the corresponding instruction is valid and the tag is checked just before the next rising clock edge. When cache access is in the critical path, shifting the tag check and the selection of the correct cache bank (when set-associative caches are used) to the next (DC) stage may help to improve the performance. This is especially true for three stage pipelines. The current design does not use such a pipelined, two cycle cache because it needs more hardware, and a completely different generic cache, which may be useless for the TLB and the load-store unit. Section 7.3 will discuss a two cycle cache access.

4.2 Control model

The control of the instruction fetch unit is responsible for all state changes. The control is distributed over the instruction fetch unit and the state unit, as shown in figure 4.8. The five controllers are:

- The exception controller is part of the instruction fetch unit and decides which exception requests are granted, and controls the registers which show the origin of the exception. These registers, the exception source register and the trap register, can be read and written by the exception handler. Subsection 4.2.1 describes this controller.
- The processor mode controller in the state unit keeps the processor mode, which tells the exception controller whether exceptions are enabled, and whether privileged instructions are allowed. Subsection 4.2.2 describes the processor modes.
- The virtual memory controller, which is described in subsection 4.2.3, controls the state of the instruction TLB.
- A cache controller (subsection 4.2.4 is responsible for the correct operation of the instruction cache and the communication with the memory interface unit.
- The controller which enables and disables the instruction cache and TLB as well as the data cache and TLB is part of the state unit. Reading and writing to this controller is possible to change the enabled signals. Its functions are described in subsection 4.2.5.

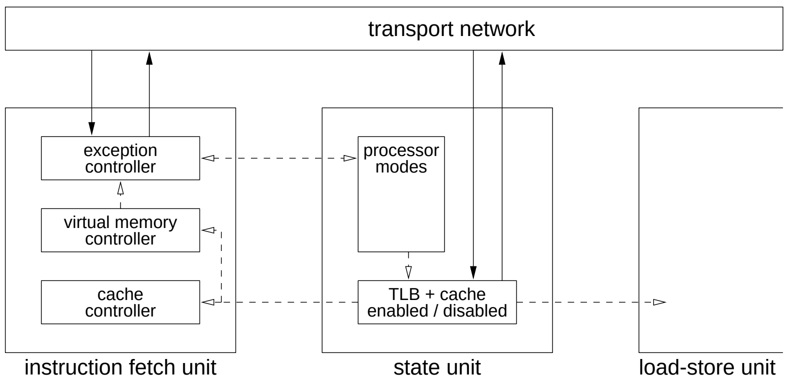


Figure 4.8: The control path of the instruction fetch unit and the state unit.

4.2.1 Exceptions

The exceptions which are supported by the exception controller are (in order of precedence):

reset A reset is a special kind of interrupt. It has absolute and immediate precedence, and cannot be masked. Even a memory access is interrupted by a reset. The reset also does some initializations. The reset handler is treated like a normal exception handler.

data TLB miss When the load-store unit does not find a translation for a load or store address, it gives an exception request. A data TLB miss is the only exception request which is available late. The request is available one cycle after the execution of the faulting instruction.

interrupt Interrupts are generated from outside the processor. It is an exception which is generated because one of the external devices (e.g. the timer) pulled the external interrupt line.

trap Traps are caused by a trap instruction. Writing to the trap register when exceptions are enabled causes an exception called a trap. Traps are useful for system calls.

instruction TLB miss The instruction TLB gives an exception request when it cannot find the translation for a program counter. All instructions before the faulting instruction are executed normally. When the last of those instructions is completed, an exception is given.

privileged instruction User mode processes cause this kind of exception when they try to execute a privileged instruction. The privileged instruction is intercepted before it can complete. This ensures that it will not be executed illegally.

After the reset, data TLB misses have the highest priority. A data TLB miss is generated by an instruction which completed the previous cycle. The exceptions comes after the faulting instruction. All other synchronous exceptions inhibit the execution of the instruction which caused the exception. When another exception condition occurred during the previous cycle, that exception routine is canceled, and the data TLB miss is handled, because it should have been handled before the previous instruction[4].

Interrupts have their origin outside the processor. If one of the external devices needs the processor, it gives an interrupt request. These requests should usually be handled fast, because otherwise the correct operation of the device cannot be guaranteed any more. This is why interrupt have the next highest priority.

Traps are handled before instruction TLB misses and privileged instructions. The execution of the instruction which orders the trap means that the translation of the virtual address of this instruction was available and this instruction does not illegally contain privileged opcodes. When a trap and an instruction TLB miss or a privileged opcode exception request come together, the trap belongs to an instruction before the instruction which caused the TLB miss or contained the privileged opcode. A trap is therefore handled earlier.

An exception vector is written into the program counter at an exception. The exception handler starts at that address. It is possible to use a separate exception address for each kind of exception mentioned above. This implies that there is only one trap exception vector. Another option

[4]The exception source register will cause an new exception if the trap bit or the interrupt bit is still high after the return from the data TLB miss exception. If the interrupted process caused an instruction TLB miss, or contained a privileged opcode, that process will generate the same exception again if it is restarted.

was to specify the exception address as an operand for the trap instruction, but this would give protection problems because it gives programs the possibility to jump to themselves in exception mode. Allowing only a small number of fixed exception addresses solves this problem, but significantly increases the complexity, without adding much functionality.

An exception request does not have to wait until the processor is not locked. It is therefore possible to generate a lock request and an exception request together when the pipeline must not proceed. Assuming a three stage pipeline, the exception sequence is:

instruction	i	IF	DC	MV			
instruction	$i+1$		IF	DC	*squash*		
instruction	$i+2$			IF	DC	*squash*	
exception	e				IF	DC	MV
exception	$e+1$					IF	DC

This sequence can be correct regardless of the locking state during the exception request. The exception request occurred during the MV stage of instruction i when the processor was not locked at that moment. Otherwise, the exception was requested at least one cycle after the execution of the move of instruction i. In both cases, instruction i is the last instruction that is allowed to complete before the exception. The instructions $i+1$ and $i+2$ are squashed to prevent their execution.

A two stage pipeline behaves slightly different. As only one instruction $i+1$ has entered the pipeline at the exception request, only one instruction needs to be squashed. This results in the following sequence:

instruction	i	IF	MV		
instruction	$i+1$		IF	*squash*	
exception	e			IF	MV
exception	$e+1$				IF

The exception handler can give the control back to the interrupted user process by executing a return from exception. The return from exception instruction is a jump instruction which has the side effect that exceptions are enabled after the delay slots. The saved cache and TLB enabled signals are also restored.

The return from exception procedure is shown below for a three stage pipeline. Instruction $e+n-2$ is the first return from exception instruction. The second delay slot is removed when a two cycle pipeline is used.

exception	$e+n-2$	IF	DC	MV			
exception	$e+n-1$		IF	DC	MV		
exception	$e+n$			IF	DC	MV	
instruction	$i+1$				IF	DC	MV
instruction	$i+2$					IF	DC

A usual return from exception procedure consists of two or three (depending on the pipeline depth) moves from the PCC-MV register to the PC using the return from exception instruction. However, one move to the PC using that return from exception instruction is enough to enable exceptions. The other returns from exception are necessary to restore the PC chain correctly, and to restart the second and third instructions with the correct cache and TLB enabled signals. The behavior of these signals will be described in section 4.2.5.

The exact behavior of the exception enabled/disabled signal is explained in section 4.2.2 about processor modes.

The exception source is stored in the exception controller. The exception source register has a bit for every exception source (except the reset). When the processor has more than one load-store unit with TLB, they all get their own bit. The bits in the register are always set when the corresponding exception condition is met, even when exceptions are disabled. It is therefore possible to check for interrupts requests when interrupts are not possible. When the data TLB would be enabled when exceptions are disabled, a TLB miss can still be detected by checking the corresponding bit.

The exception source register can be read and written. The exception handler can read the register to know exactly the origin of the exception. The whole register can be written to set or reset the exception source state. This can be necessary because the bits are not reset automatically. A new exception is generated immediately if the trap bit or the interrupt bit is still high after the return from exception. The format of the exception source register is:

(msb) 4	3	2	1	0 (lsb)
dTLB miss	iTLB miss	privileged opcode	trap	interrupt

The exception controller also contains a trap register. Writing into this register causes a trap when exceptions are enabled. A write always sets the trap bit in the exception source register. A user process can specify which function the exception handler must perform by writing a specific value to the trap register. The trap handler can read this register.

The instructions which access the trap and the exception source register are shown in the tables below. The registers can be read via output socket 2 and can be written by input socket 3.

Input socket 3 functions:

opcode	*action*	*description*
0	$src \rightarrow trap\_reg$	give a trap; write the trap register
1	$src \rightarrow exc\_source$	write the exception source register

Output socket 2 functions:

opcode	*action*	*description*
0	$trap\_reg \rightarrow dst$	read the trap register
1	$exc\_source \rightarrow dst$	read the exception source register

4.2.2 Processor modes

The processor can run in several modes. It can run in virtual or real mode (use/don't use the virtual memory system). Exceptions can be enabled or disabled. Privileged instructions may be allowed or forbidden. The caches can be used, or not used. The state unit controls those parameters. In this design the cache and TLB enabled signals are chosen to be completely independent. They can combine with all other modes, and are described separately in subsection 4.2.5.

Fundamentally, the exceptions disabled/enabled signal and the kernel mode signal (privileged instructions allowed/inhibited) are independent too. They are however combined in the state fetch unit to make the control easier. The next table shows all modes which are fundamentally possible. Of them, only the *exception mode* and the *user mode* are supported.

	mode	exceptions	priv. instr.
1	exception mode	No	Yes
2	*not supported*	No	No
3	*not supported*	Yes	Yes
4	user mode	Yes	No

A process which is not allowed to execute privileged instructions should not run with exceptions disabled. The OS cannot control a program which cannot be interrupted. Only trusted OS code may run with exceptions disabled. There is no reason to withhold this code from executing privileged instructions. Furthermore, illegal privileged instructions cannot be stopped by exceptions when exceptions are disabled. This is why mode 2 is not supported.

The opposite case is mode 3. Code which runs in this mode can be interrupted by exceptions, and can also execute privileged instructions. This mode might be useful. Interrupt requests may have to wait too long when the complete OS code is run with exceptions disabled. Making a part of the OS code interruptible may help to make the latency between the interrupt request and the starting of the interrupt handler shorter.

Making privileged instructions available when exceptions are possible, will make the control of the privileged registers more complex. A part of the privileged registers is used for exception support, like the PC chain registers. When exceptions are enabled, these registers may contain unpredictable data, and writing them may cause incorrect exception handling. The usefulness of these privileged registers is therefore very restricted when exceptions are enabled. This lead to the decision to allow privileged instructions only when exceptions are disabled.

The processor gets into exception mode at a reset. A return from exception is needed to go to user mode. An exception forces the processor to switch to exception mode again. This cycle is always repeated. The precise procedure for a three stage pipeline is shown in the finite state diagram (FSM) of figure 4.9.

In this FSM, the states *rfe_mv* and *enabled* correspond to user mode. In the other states, the processor is in exception mode. The states *rfe_mv* and *exc_mv* are not necessary for a correct exception behavior, but are needed for the correct writing of the data cache and TLB enabled signals (section 4.2.5). The exception behavior of these states is exactly the same as of the states *enabled* and *disabled*, respectively. When a two stage pipeline is used, the states *rfe_dc* and *exc_dc* are removed.

The processor gets into state *exc_if* at an exception. In the pipeline are now two instructions which should not be executed. They are squashed in the states *exc_if* and *exc_dc*. The first instruction of the exception routine is executed in state *exc_mv*, and the other instructions in state *disabled*, except the delay slots of the return from exception.

The state machine jumps to *rfe_if* when the exception handler executes a return from exception instruction. After two delay slots[5] (*rfe_if* and *rfe_dc*), the processor enters user mode in state

[5] If a two cycle transport pipeline is used, there is only one delay slot instruction, which is executed in state *rfe_if*.

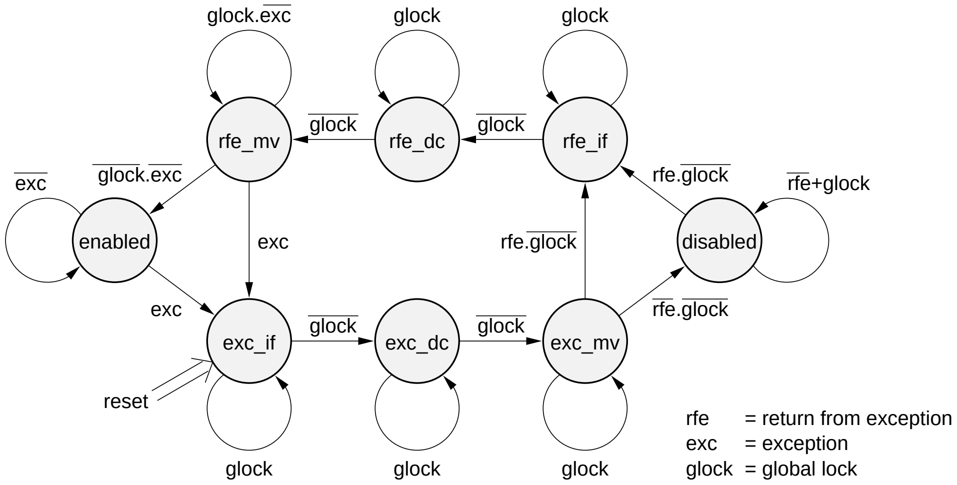


Figure 4.9: The state diagram which controls the exception state.

rfe_mv, in which the first user mode instruction is executed. Exceptions are possible from that moment on. All other user mode instructions are executed in state *enabled*.

The implemented state machine differs slightly from the one described above. The *exc* and *rfe* signals are generated by the instruction fetch unit. For performance reasons they are sent to the state unit with one cycle delay. The state which is shown is therefore not always equal to the state stored in the state register. The real state is constructed from the contents of the state register and the delayed *exc* and *rfe* signals. If the *exc* or *rfe* signal specifies that the state should have changed in the previous cycle, the real state gets the correct value, while the state register still contains the old value. The exceptions enabled signal is therefore not only a function of the state registers (Moore model), but also a function of the delayed *exc* and *rfe* signals (Mealy model).

The privileged instruction system ensures that some instructions are restricted to exception mode processes. There may be two reasons to declare an instruction privileged:

- Protection. A user mode program must not be able to affect the correct exception behavior or change the protection level of any process. Writing to the TLB or to registers which are needed for state saving at an exception is therefore privileged.

- Processor emulation. The principle of virtual processor emulation was explained in section 3.5. All instructions which do not behave equal in user mode and in exception mode should cause an exception when a process tries to execute them in user mode. This implies that reading the registers which are needed for state saving and tracking the origin of an exception is privileged. It also restricts the use of special *lock* or *push/pull* modes in pipelined FUs[6], which may be implemented to make saving and restoring the state of that unit easier.

[6] It does not forbid the use of such modes. A push/pull mode is allowed if a user mode process can also use that mode, which implies that it must be possible to save and restore the mode signal.

The return from exception instruction and writing to the TLB, the PC chain or the exception source register are privileged in the instruction fetch unit because of protection reasons. The processor emulation function requires that reading the PC chain, the exception source register or the trap register is privileged too. Within the state unit, reading or writing the current TLB enabled signal and the saved cache and TLB enabled signals is privileged. Writing the data TLB is privileged too.

The state unit provides a *user_mode* signal, which indicates whether privileged instructions are forbidden. In the selected implementation this signal is equal to the *exc_enabled* signal which specifies whether exceptions should be granted. The instruction fetch unit checks all the opcodes which it sends to the FUs. When one of the opcodes is privileged, and the *user_mode* signal is high, an exception is given. When the privileged instruction was already in the MV stage, a global lock is given too, to prevent the execution of the privileged instruction.

4.2.3 Virtual memory controller

The virtual memory control is distributed over two units. The state units provides the TLB enabled signal. This signal is used in the instruction fetch unit by the controller of the virtual memory system. The state unit also provides an enabled signal for the load-store unit TLB. The generation of the TLB enabled signal is covered by section 4.2.5.

A TLB lookup is started just before the clock edge at which that instruction enters the IF pipeline stage. The address for that generic cache access is the value which will be written into the PC register at the clock edge. The generic cache interface requires this behavior.

The TLB enabled signal which belongs to that lookup is not known before the IF stage. At the time a lookup is started, the controller does not know whether the translation is really needed. When the TLB turns out to be enabled, the tag is checked using the PC register. The output of the generic cache, which is used as the TLB, is then directed to the output. Otherwise the value in the program counter is send to the output.

A miss is reported to the exception controller when the TLB is enabled and at least one of the next conditions is true:

- The generic cache, which is used as the TLB, reports a miss.
- The valid bit which is connected to the translation is not set.

Updating the TLB requires two instructions. The first instruction writes the virtual page number, which is looked up in the TLB. Nothing is written into the TLB yet, but the results of the lookup are needed for the write caused by the second instruction. This second instruction writes the physical page number and the valid bit into the TLB, and connects them to the virtual page number from the first instruction.

As the TLB can do only one lookup or write in a cycle, it has to make a choice when something is written to the TLB, because the new PC may have to be translated too. The TLB handles the write instruction first. If the TLB turns out to be enabled, the translation for the PC is needed too. The virtual memory controller solves this problem by locking the processor for one cycle, which makes it possible to translate the TLB in the next cycle. This mechanism allows TLB updates when the TLB is enabled.

4.2.4 Cache controller

The cache controller takes care of the instruction cache and the communication with the memory interface unit. The cache controller uses two enabled signals, which are provided by the state unit. The state unit also contains an enabled signal for the load-store unit. Section 4.2.5 describes the generation of the enabled signals.

The cache controller uses a *read_enabled* and a *write_enabled* signal. The *read_enabled* signal controls whether a memory access is started when the desired instruction is found in the cache. Disabling cache reads may be useful when the cache may contain invalid instructions, e.g. after a reset, or after a write to an code page. The *write_enabled* signal controls whether an instruction from memory is written into the cache. Disabling cache writes may be useful to prevent a piece of code from overwriting the current cache contents.

The cache controller core is formed by a finite state machine (FSM) with seven states, as shown by figure 4.10. This figure does not show the transition conditions, because that would make the picture less clear. The complete transition table is given in appendix B. The best way to understand the seven states is to see them as a combination of the state of the instruction in the IF pipeline stage, and the state of the instruction in the DC or DC/MV pipeline stage.

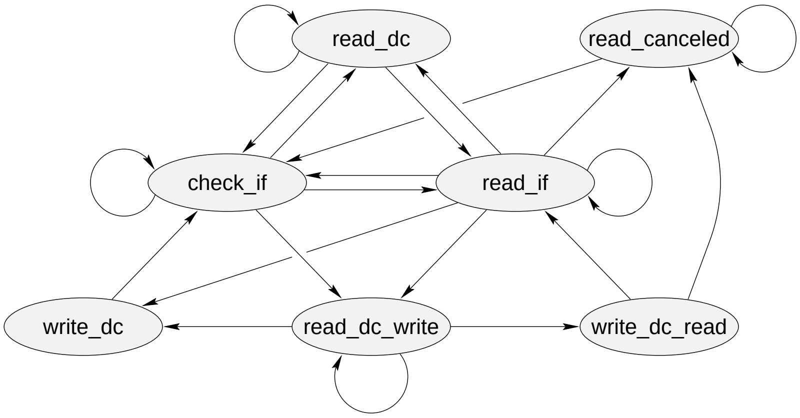


Figure 4.10: The finite state machine for the cache controller. The state transition conditions are specified in appendix B.

The address of the instruction in the IF stage is in the program counter. This instruction always starts with a cache lookup. When the instruction is not found in the cache, or cache reading is disabled, a memory access may be started speculatively as soon as the memory address register (AR) comes free. As long as the instruction has not propagated to the DC stage, the memory access remains speculative. It may become unnecessary at an exception, or after the write of the DC instruction into the cache when the IF instruction appears to be the same as the DC instruction. The memory access has to be finished, but the result will not be saved anywhere.

The instruction in the DC pipeline stage is ready for distribution and decode when it is in the instruction register (IR). It gets there after a cache hit, or a memory access. When an instruction causes a cache miss when it is in the IF stage, a memory access is started. The result is not written into the instruction register until the instruction propagates to the DC stage. If cache writing was enabled when the memory access was started, the result must be written into the cache too. This write prevents IF instructions from doing a valid cache lookup for one cycle. After the write, the cache contents have changed, and the IF instruction lookup must be repeated.

The state of the IF instruction and the state of the DC instruction are not independent. In the state machine of the cache controller they are combined because it leads to a significant reduction of the number of states. It makes the design a little clearer, and testing less complex. The resulting states are:

check_if The DC instruction is ready and in the instruction register. The cache is now performing a lookup for the IF instruction. The miss signal is available at the end of the cycle.

read_if The DC instruction is in the instruction register, ready to be executed. A speculative memory access for the IF instruction has been started. The IF instruction will not be written into the cache until it propagates to the DC stage.

read_dc_write The DC instruction is being read from memory. This instruction will be written into the cache when the memory access is finished. The cache is performing a lookup for the IF instruction, to be able to start a speculative memory access for that instruction immediately after the current memory access is finished. The state of the IF instruction might however change when the DC instruction is written into the cache.

read_dc In this state the DC instruction is also read from memory, but the cache will not be updated. The DC instruction will only be written into the instruction register, after which it can be executed. The IF instruction is looked up, and when a memory access is needed, it will be started immediately after the completion of the DC instruction has been read.

write_dc_read The DC instruction is being written into the cache, and a speculative memory read for the IF instruction has been started. The write to the cache prevents the final lookup of the IF instruction, to get its state after the write of the DC instruction.

write_dc The DC register is being written into the cache. The IF instruction is waiting until the cache is available for its final lookup. No memory access is pending.

read_canceled The DC instruction is ready and in the instruction register. The instruction which is now read from memory is not necessary any more, but has to be completed. The controller may get into this state after an exception, or when the DC instruction and the IF instruction are on the same address, and the write of the DC instruction into the cache also wrote the IF instruction.

The processor is locked in the *write_dc_read* and the *write_dc* states because the DC instruction is not known in these states. After a write of the DC instruction, the state of the IF instruction might have changed. It has therefore to be read again after a cache write. This is the reason why the processor is locked in the *write_dc* and the *write_dc_read* states. The processor is also locked in the *read_canceled* state because the memory address register is used in that state. When processor would not be locked, and the IF instruction causes a cache miss, the instruction address is lost. It cannot go to the memory address register, and the program counter is overwritten.

The default state of the cache controller is *check_if*. The controller will be in this state if the cache only gets hits.

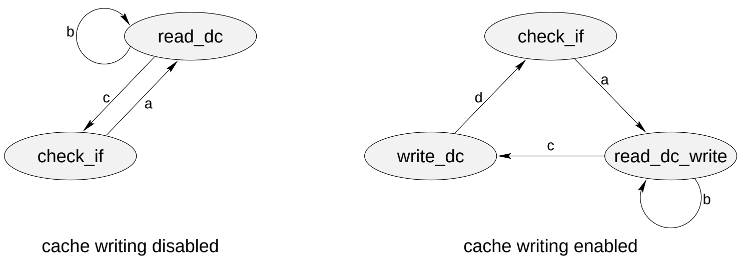


Figure 4.11: The cache controller sequences for one cache miss among many hits.

If one instruction causes a cache miss, the controller takes one of the sequences in figure 4.11. The left sequence is valid if cache writing is disabled, and the right sequence if cache writing is enabled. In *read_dc* or *read_dc_write*, the instruction is loaded from memory. In these states, the controller waits until memory is ready. If cache writing is enabled, the instruction is written into the cache in state *write_dc*.

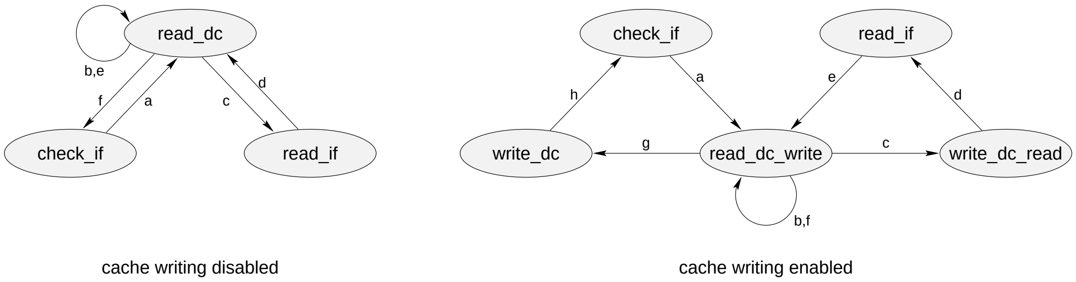


Figure 4.12: The cache controller sequences for two subsequent cache misses among many hits.

If two subsequent instructions cannot be found in the cache, the controller sequences from figure 4.12 are valid. The left sequence is used if cache writing is disabled; otherwise the right sequence is used. In the states *read_dc* or *read_dc_write*, the state of the next instruction is checked, which turns out to be a miss. After the memory is ready, the cache controller immediately starts a new load for the second missing instruction. If cache writing is disabled, this instruction is loaded in state *read_if*. If cache writing is enabled, the first missing instruction is written into the cache in state *write_dc_read*, before the controller gets to state *read_if*. The controller gets from state *read_if* to *read_dc* or *read_dc_write* as soon as the processor is not locked, and the first instruction that caused a miss is decoded. This clears the instruction register, which can be used if the memory has finished the load of the second instruction. As the third instruction is in the cache, the controller now gets back to *check_if* (after writing the second instruction into the cache in state *write_dc* if cache writing is enabled).

If the cache controller is processing a large number of subsequent cache misses (e.g. if cache reading is disabled), one of the loops from figure 4.13 is used (the left one if cache writing is disabled, otherwise, the right one). It becomes clear from this picture that the minimum memory cycle time is three if cache writing is enabled, and two otherwise. If cache writing is enabled, and

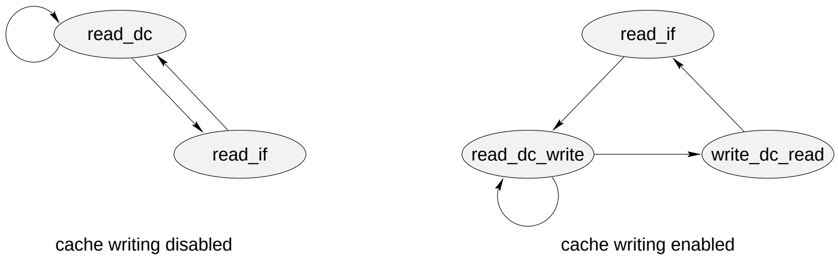


Figure 4.13: The cache controller sequences if all instructions cause a cache miss.

the memory has a new instruction ready two cycle after the load of the previous instruction was finished, the controller gets from state *read_if* to *write_dc* instead of *write_dc_read*. In that case, the loop is shown in figure 4.14. The effective memory cycle time remains three, as the memory interface is not used in the states *write_dc* and *check_if*. The controller cannot speculatively start a memory access by jumping to *write_dc_read* instead of *write_dc* because the correct memory address is not in the memory address register yet.

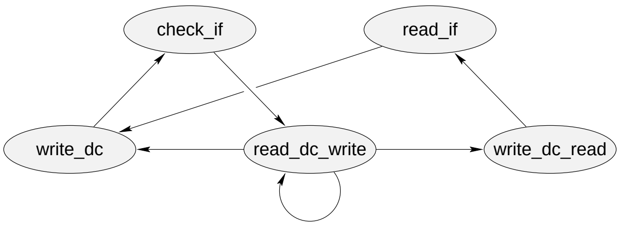


Figure 4.14: The cache controller sequence if the cache writing is enabled, all instructions cause a miss, and the memory access time is two cycles.

4.2.5 Cache and TLB enabling

The state unit provides the cache and TLB enabled signals for both the instruction fetch unit and the load-store units. The caches and the TLBs can be turned on and off by writing to the state unit. This unit also offers functions for saving and restoring the cache and TLB enabled signals at exceptions.

For each of the four kinds of enabled signals (instruction cache, data caches, instruction TLB, data TLBs) the state unit keeps two sets. One set is used for user mode, the other for exception mode. At an exception the exception mode sets are sent to the instruction fetch unit and the load-store unit, and after a return from exception the focus switches to the user mode sets. The user mode sets can be written in exception mode. At a reset all exception mode signals are set to disabled.

There are two or three registers in the user mode sets for the instruction cache enabled and TLB enabled signals (two for a two stage pipeline, three for a three stage pipeline). This is done to ensure correct exception behavior. Every saved instruction address now gets its own cache and TLB parameters. At an exception they must be saved, because of the same reason the PC chain registers are saved at an exception.

The load-store unit signals have effect from the first instruction after they are written, but the instruction fetch unit signals have a latency, which is equal to the number of pipeline stages minus one. When instruction i enables the instruction cache, instruction $i + 3$ is the first instruction for which the cache is used (assuming a three stage pipeline). When instruction i also enables the data cache, loads and stores triggered in instruction $i + 1$ will use the data cache.

This difference influences the exception behavior of the instruction fetch unit signals and the load-store unit signals. They both use the state machine which was also used for the processor modes (figure 4.9). At an exception the instruction signals change on the transition to *exc_if*, and the data signals on the transition to *disabled*. A return from exception becomes visible on the inputs of the instruction fetch unit signals when the state changes to *rfe_if*, while the the signals for the load-store unit are restored on the transition to *enabled*.

The state unit supports four functions for reading and writing the cache and TLB enabled signals. All functions read and write both the signals for the instruction fetch unit and the load-store unit. The instruction fetch unit signals are on the least significant bits; the load-store unit signals are next to them. The opcodes are the same for reading and writing. The next table shows the state unit functions.

opcode	*description*
0/00	read/write the current cache enabled signals
1/01	read/write the saved cache enabled signals
10	read/write the current TLB enabled signals
11	read/write the saved TLB enabled signals

Figure 4.15 shows the principle by which the data cache and TLB enabled signals are written. The read functions and the reset behavior are not included in the scheme. The user mode and exception mode signals are implemented symmetrically. All writes to one of these registers go via the *written* register. This register was implemented to make the implementation similar to the implementation of the instruction cache and TLB enabled signals.The reason why the *written* register was needed in that implementation is explained below.

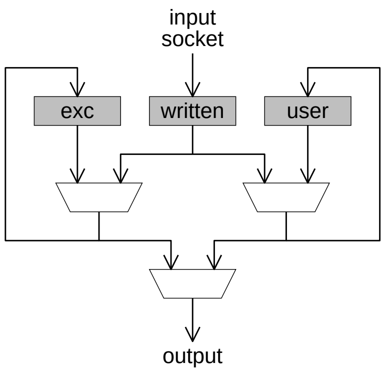


Figure 4.15: The principle by which the cache and TLB enabled signals are constructed.

The scheme is extended on the user mode side for the instruction cache and TLB enabled signals, which is shown in figure 4.16 for a three stage pipeline. As there are two or three user mode sets of enabled signals[7] needed for a correct exception behavior, there are two or three user mode registers. These registers react just like the PC chain registers. The enabled signals ripple through the chain at every write. The new value is written in the leftmost (IF) user mode register. A read shows the MV register.

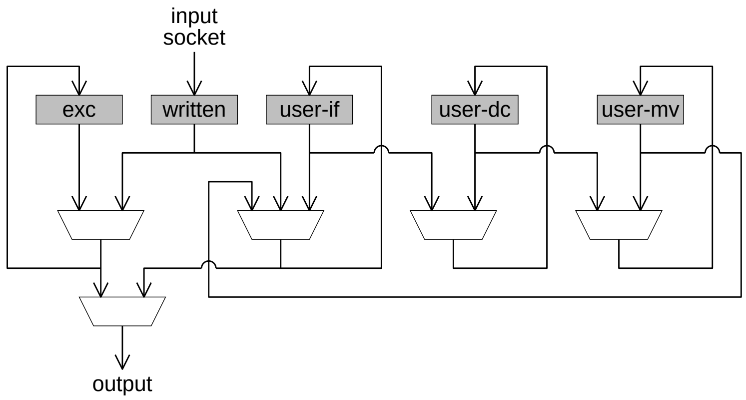


Figure 4.16: The extended scheme for the construction of the instruction cache and TLB enabled signals.

In user mode (and the *rfe_if* and *rfe_dc* states), the DC and MV registers save the previous versions of the IF signal. In exception mode, the chain is locked. The new IF enabled signal is only changed at writes and at a return from exception. After a return from exception instruction, the value which was saved in the MV register is used as the new value for the IF register. The saved cache and TLB enabled signals are restored this way.

The *written* register is needed in the implementation of the write functions of the instruction cache and TLB enabled signals because the *exc* and *rfe* signals arrive at the state unit with a delay of one cycle[8]. At the moment a write is triggered, the values of these registers are not known to the state unit, though they might be necessary to decide into which register (user mode, exception mode, or none) the write operand must be written.

An example of this situation is shown in the next instructions:

```
1: pcc -> rfe, "111" -> cache_saved; enable caches after rfe
2: pcc -> rfe,                      ; delay slot 1 of rfe
3: pcc -> rfe,                      ; delay slot 2 of rfe
```

In this code, the last three instructions of an exception handler are shown for a three stage pipeline. Together with the first return from exception instruction, a write to the saved (user mode) cache registers is triggered. If the instruction cache saved registers show the same behavior as the PC chain registers, this write does not affect the saved instruction cache signal chain. At the moment this write is triggered, the state unit does not know whether a return from exception is being triggered too. The write value is saved in the *written* register for one cycle. The next

[7] Two for a two stage transport pipeline, and three for a three stage pipeline

[8] The same *exc* and *rfe* signals are used for the FSM for the processor modes which is explained in section 4.2.2. This FSM influences the behavior of the cache and TLB enabled signals.

cycle, the *rfe* signal will become high, so the state unit knows that the written value must not be written into the *user-if* register.

4.3 Generic cache

The generic cache is designed as a cache template which can be used in every unit where caching is needed. The instruction fetch unit uses it in its cache and TLB. The generic cache can also be useful for the data cache and TLB, though the current implementation of the load-store unit does not use it.

The advantage of a generic cache is that it has to be designed, and tested only once, and can be used in several units, which can all offer the configuration options which are offered by the generic cache. The disadvantage is that, though the generic cache is very flexible, the cache design cannot be optimized for the unit in which it will be used as well as a dedicated cache design for that unit only. For the instruction fetch unit, this is not a major problem, unless the instruction cache is pipelined, and gets an access time of two cycles (see section 7.3).

The generic cache is highly parameterizable. The next table shows the which parameters can be adjusted, and what values are currently supported. Creating additional configurations is possible.

parameter	range	description
cache size	2^n	number of cache entries
associativity	1, 2	number of places where a block may be placed in the cache
replacement	LRU, FIFO	replacement algorithm for set-associative caches
block size	2^n	entries on one cache line
address size	≥ 1	bits in a cache address (tag + offset)
data size	≥ 1	bits in a data word

The basic datapath structure of the generic cache is given in figure 4.17, which shows the simplest configuration (direct mapped, block size 1). The cache data and tags are stored in an efficient memory structure called a *ram cell*. One generic cache implementation may contain more than one ram cell.

This section will describe the generic cache. subsection 4.3.1 defines the interfaces of the cache and the ram cells which are used inside the cache. The subject of subsection 4.3.2 is the implementation of set-associativity in the generic cache. Subblocking, which makes it possible to combine more than one cache word on a cache line, is explained in subsection 4.3.3.

4.3.1 Interfaces

The external interface for all configurations is the same. Internally the cache uses one or more efficient memory structures called ram cells. Every cell library uses its own ram cell interface, which made the definition of a uniform ram cell interface necessary. The ram cells which are used inside the cache should be constructed from a library cell and an interface shell, to let the ram cell meet the desired interface specifications

Figure 4.18 shows the external interface signals of the generic cache. The *read_offset* signal is used to select a set or a cache line for a cache lookup. The *read_tag* is used for the miss detection (tag check) and the selection of the correct cache word from the selected set. The output is sent

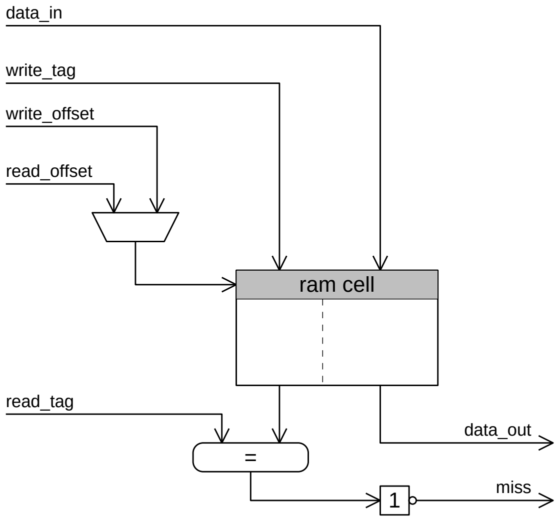


Figure 4.17: The datapath structure of a direct mapped cache.

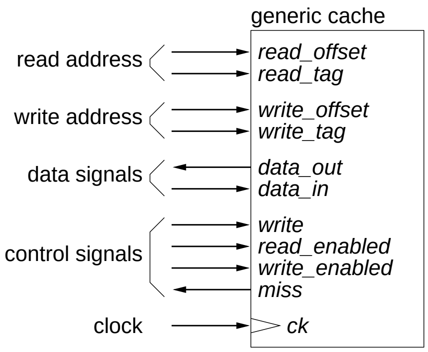


Figure 4.18: The signals in the generic cache interface.

to the *data_out* signal. At a write the *data_in* and *write_tag* signals are written to the cache set or line which is selected by the *write_offset* signal.

The *write* signal selects the read or the write function. After a read, the *miss* signal becomes high when the requested entry was not found in the cache. The *read_enabled* signal is only used with a set-associative cache with LRU replacement. This signal lets the cache know that the entry which was just read is being used. At a write the set-associative caches must know into which bank the data must be written. Subblocking caches need to know which valid bits must be set and reset at a write. This state is saved when the *write_enabled* signal gets high after a read.

The write procedure for the generic cache starts with a read of the target address. During the tag check stage, the write information must be saved by the *write_enabled* signal. The write may immediately follow the read. The write address must be the same as the address for which the information was saved after the read, otherwise the cache contents may get corrupted.

The timing of the generic cache interface is shown in figure 4.19. All actions take exactly one cycle to complete. The cache parameters, especially the associativity, may influence the exact

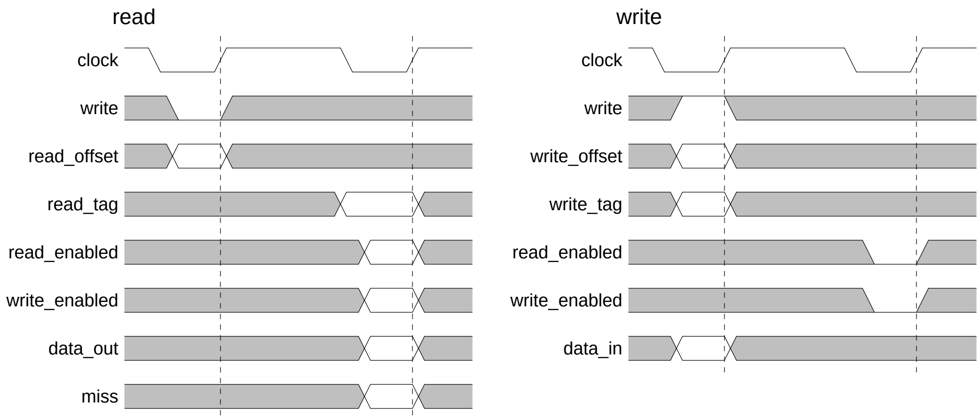


Figure 4.19: The timing of the generic cache.

time at which the cache lookup result is available, but this result will be available at the end of the cycle for all configurations.

The addresses which are used for a read or a write must be available just before the lookup or write cycle starts. They are clocked in on the rising clock edge. The tags are checked during the lookup cycle itself. At the end of the lookup cycle the results may be saved for a write (*write_enabled* is high), or the entry can be marked as the most recently used (*read_enabled* is high).

The setup and hold times are not exactly specified. They largely depend on the used ram cells. The ram cell interface shell has to make sure the ram cells and the cache environment are compatible

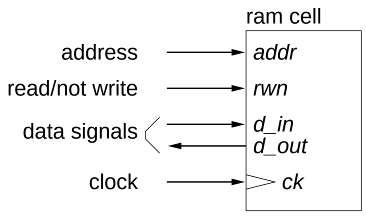


Figure 4.20: The interface signals of the standard ram cell.

The interface of the ram cells which are used in the generic cache is chosen to fit the normal synchronous ram cell interface. Figure 4.20 shows which signals are used. All signals are positive edge triggered. This implies that the address and data input signals have to be valid just before the clock edge which starts the read or the write.

The *read/not write* signal is just the opposite of the *write* signal from the interface of the whole cache.

4.3.2 Associativity

The generic cache accepts a parameter which specifies the associativity of the cache. The configurations which are currently supported are direct mapped and two-way set-associative. Least recently used (LRU) and least recently written or first-in–first-out (FIFO) replacement policies are supported for the two-way set-associative cache.

Two-way set-associative caches consist of two separate cache banks. Which bank is used is selected by the result of the cache tag check for both banks. The bank that contains the right tag sends its results to the output. The datapath for a 2-way set-associative cache is showed by figure 4.21.

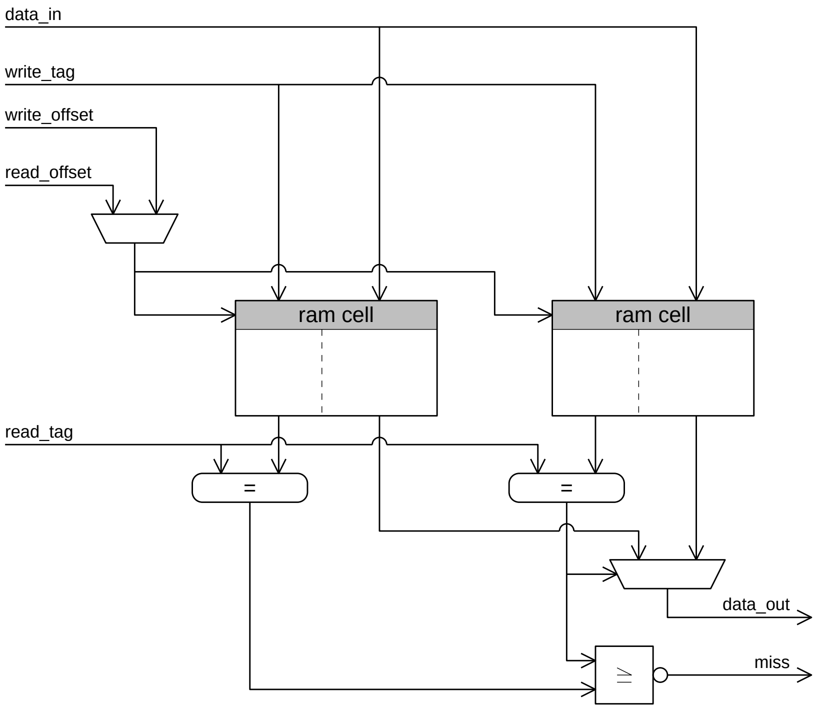


Figure 4.21: The datapath of a 2-way set-associative cache.

The cache needs to know to which bank a write must go. To save this information, the destination address must be looked up first, and the *write_enabled* signal must be high during the tag check stage. When one of the cache banks already contains an entry for the selected address, this entry will be overwritten. Otherwise, the selection of the bank into which the new entry will be written, depends on the used replacement policy.

The FIFO replacement strategy replaces the entry which was written the longest time ago. To keep track of which bank is the most recently written, both banks have one extra bit for each entry. When the write information is being saved, the bits from both banks are compared. If they are equal, the entry in the first bank will be replaced on a write, otherwise the other bank will be written. The bit in the selected bank will be adjusted on a write, making this entry to the most recently written.

The LRU replacement strategy replaces the entry which was used the longest time ago. The cache has a register for each pair of entries (one from each bank). This register contains one bit which points to the least recently used entry. This register is updated after every read which was executed with the *read_enabled* signal high during the tag check, and after every write. Which bank will be replaced on a write is decided when the write information is being saved (*write_enabled* is high). An access of one of the entries after the write information has been saved does not influence the decision which entry will be replaced.

Generally, the LRU replacement policy will give better hit rates. The advantage of the FIFO replacement policy is that it requires less chip area than the LRU policy, because it does not need a separate register to store the the bits which specify which entry should be the first to replace. It uses bits in the ram cells, which are generally much more efficient than registers. The LRU replacement algorithm cannot use the ram cells because it this algorithm requires that the bits are updated after each read, whereas the FIFO bits have to be updated after a write only.

The direct mapped cache is always smaller than a two-way set-associative cache of the same size. It uses only one ram cell, while the 2-way set-associative cache uses two ram cells of the half size. The ram cell overhead for a direct mapped cache is therefore less. Furthermore, the direct mapped cache does not save any additional control information. The performance of the direct mapped cache is worse too. The time needed for the lookup in the ram cells and the tag checks is approximately equal, but the direct mapped cache can sent its result directly to the output, before the tag check, while the 2-way set-associative cache needs the results of the tag check to select the correct output.

4.3.3 Subblocking

A cache uses a tag for every cache line. Instead of using one word per cache line or cache block, it is more efficient to put several cache words on the same cache line. This reduces the number of tag bits when compared to the number of data bits. As the cache words are still written one by one, the generic cache uses a subblocking system to remember which words on a cache line are valid.

Every cache word gets its own valid bit. All valid bits on a cache line keep their old values when new word is written to that line with the old tag. When the new word is written with a tag which is different from the old tag on the destination cache line, all valid bits on that line are reset. The valid bit of the word which is written is always set to valid.

The implementation of subblocking uses two separate ram cells. One big ram cell contains all data words. This unit is addressed with the complete cache address. This ram cell stores the cache words, not the cache lines. The other, smaller ram cell contains all tags and valid bits, and does not use the line offset bits when it is addressed. After a lookup the correct valid bit is selected using the saved cache line offset. When this bit is high and the tag check returns true, a hit is reported.

When the *write_enabled* signal is high during the check stage, the valid bits which should be used at a write are recorded. The write itself does not write the whole cache line. Only the tag, the valid bits and the selected cache word are written.

The hit rate of a cache using a large block or line size may be lower than the hit rate of a cache which has a tag for every word. The cache which employs subblocking can be much smaller. Using two instead of one ram cell generates some overhead, but this this will generally be compensated by the area which is saved because less tags have to be saved. As the number of tags has decreased, the tag lookup is faster and the tag check can be completed earlier. This is especially

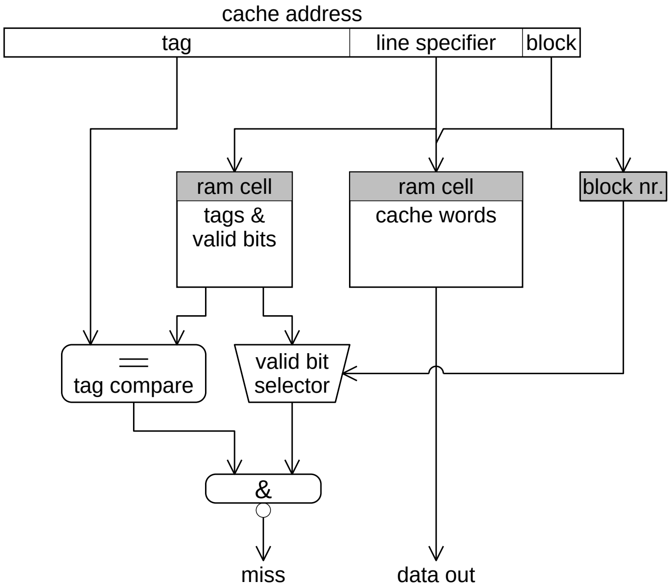


Figure 4.22: The implementation of subblocking in the generic cache.

important for set-associative caches.

4.4 Immediate unit

The previous versions of the MOVE processor generator supported short immediates only. The source fields of move operations could be used as data, as shown in figure 4.23. For practical configurations, the size of these short immediates could not exceed 5 to 7 bits. Using larger source fields, or dedicated immediate fields is easy to implement, but inefficient in terms of instruction bandwidth.

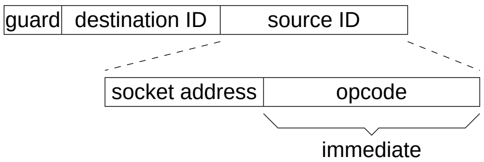


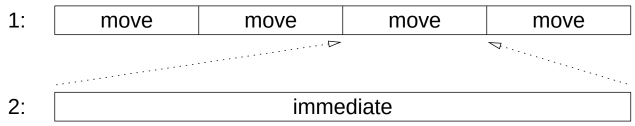
Figure 4.23: Using the opcode part of the source ID to create short immediates

To be able to use bigger immediate sizes more efficiently, the short immediate system must get an extension, which makes it possible to use instruction bits from other move slots as data. Eventually, the extension system may completely replace the short immediate system. Extensions for the immediate may be added before, within, or after the instruction which uses the immediate.

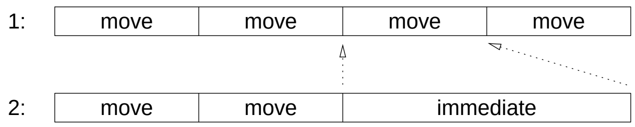
Two alternatives for the immediate construction are discussed below. They both use move slots around the instruction which uses the immediate. The first scheme uses the instruction after the one which uses the immediate. The other scheme specifies the immediate bits before the instruction which uses them.

Specifying an immediate after it is used

As the immediate extensions are specified after the instruction which uses the immediate, the first scheme does not need a separate instruction format to specify which move slots have to be used as immediates. The source field of the move which fetches the immediate specifies whether the next instruction must be used as an immediate, or as normal move operations:



The source field may also specify how much move slots must be used for the immediate. When not all move slots of the following instruction are needed, the other slots may be used for normal move operations:



The source field itself may also contain some immediate bits. The immediate is then constructed from the bits in the source field and the bits from the next instruction. If one instruction is not enough to specify all the necessary immediate bits, more than one instruction after the one which uses it may be used.

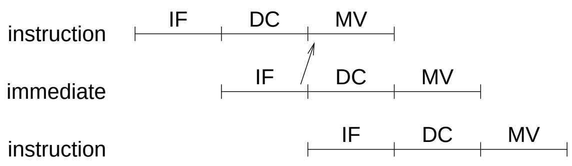


Figure 4.24: The timing of the immediate which is specified after it must be used in a three stage pipeline.

The advantage of this system is that it hardly uses any extra instruction bits. It is easy to implement when a three stage pipeline is used, and only the next instruction is used for the immediate extensions. When the immediate is needed, the instruction with the immediate has just finished the IF stage, and is therefore available directly. No locks are needed. This is shown in figure 4.24. Note that the DC and MV pipeline stages of the immediate can be removed if there are no other moves within the instruction that carries the immediate.

The two stage version is more complex. Figure 4.25 shows the timing when the immediate uses all move slots of the instruction which carries it. The exception behavior is what makes this option difficult. After an exception between $i-1$ and i, or between i and $i+1$, the process must be restarted at instruction i. An exception between $i+1$ and $i+2$ makes that the interrupted process has to be restarted at instruction $i+2$. With some extra hardware, this can be implemented in the program counter datapath of figure 4.4, but implementing this in the state unit, which records the instruction cache and TLB signals, is difficult.

If the immediate instruction also carries some normal move instructions, the timing of figure 4.26 must be used. This figure also removes the disadvantages of the timing scheme of fig-

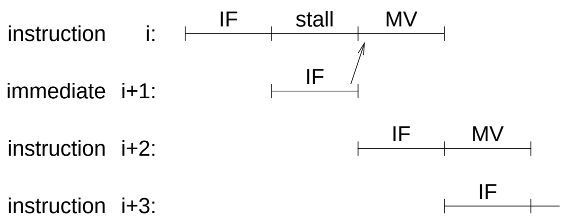


Figure 4.25: The timing for a two stage pipeline of an immediate which is specified after it must be used. The immediate instruction $(i+1)$ carries no moves.

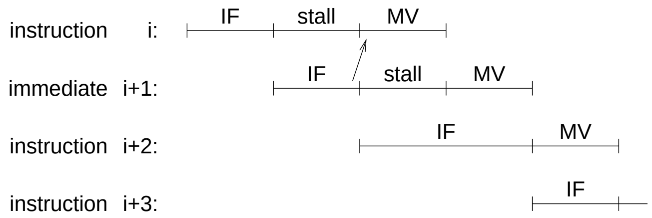
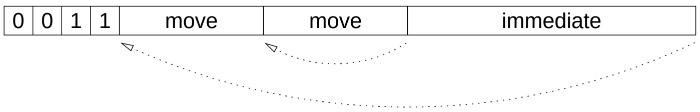


Figure 4.26: The timing for a two stage pipeline of an immediate which is specified after it must be used. The immediate instruction $(i+1)$ carries some moves too.

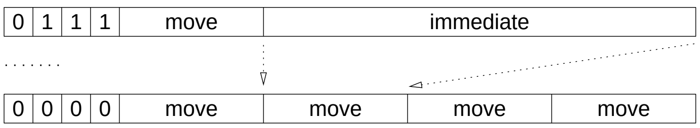
ure 4.25, but introduces another disadvantage. The cache lock signal cannot be synchronous with the global lock signal because the immediate bits of instruction $i+1$ are not available until the cache and the PC are not locked. The processor must be locked during the stall cycle of instruction i.

Specifying an immediate before it is used

The second system for the generation of immediates uses special bits in the instruction format to specify which move slots will be used as immediates. The immediate can be specified in the same instruction as the one which uses it:



When the number of bits in the same instruction is not enough, one of the previous instruction can be used instead:



The advantage of this system is that it is easy to implement. All information about which slots contain a move operation, and which an immediate, is contained in the same instruction. The timing scheme does not need any modifications. Another advantage is that this system is very flexible. The distance between the immediate bits and the move operation which uses them is variable. The number of slots which are used for an immediate can be different for every immediate. Adding a shift register to generate very long immediates is possible.

The extra instruction bits, which are needed to specify which move slots must be used as immediates, form the main disadvantage of this system.

Implementation of the immediate unit

The second system is selected, because it is easier to implement for both two and three stage pipelines, and because of its flexibility. It is designed as an extension system for the existing short immediate principle. The least significant bits of the immediate are taken from the source opcode, just like the short immediate system uses them. The most significant bits must be specified in move slots in the in the same instruction or before the one which uses the immediate. The format is the same as described above for the second option.

Every extension from a certain move bus always appears at the same place in the immediate. When some extensions are not specified, they are constructed using sign extension, or zero extension. It is possible to specify that the instruction for move bus 0 and 2 contain extensions, while the instruction for bus 1 and 3 do not contain an extension. In that case, the extensions from bus 1 and 3 are constructed as a sign extensions from the parts of the extension which were specified for the busses 0 and 2 (or zeros if sign extension was not implemented). This is shown in figure 4.27. In this example, move 3 uses an immediate which is specified within the same instruction.

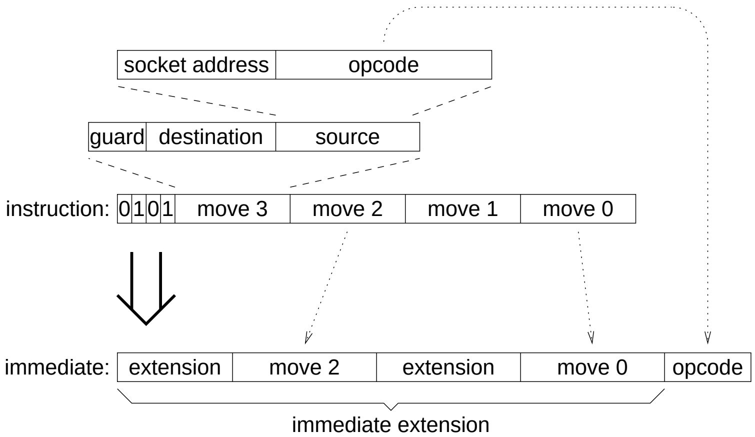


Figure 4.27: Specifying extensions for a short immediate.

Both the short immediate and the extension system are implemented in the immediate unit. This unit is separate from the instruction fetch unit, and uses the normal functional unit interface. It may have two kinds of outputs for every move bus: one for short immediates, and one for extended immediates. The short immediates use the opcode bits from the source ID only. The long immediates use the opcode bits together with zero or more extensions. Short immediates, or immediates which are not extended to the full bus width, can be sign extended as an option.

The simplest version of the immediate unit only accepts extensions from the same instruction. A more complex version also accepts extensions from instructions before the instruction which uses the immediate. This version uses a register inside to store the immediate extensions. The register can be written to support exceptions. The most complex version supports the generation of very long immediates by shifting.

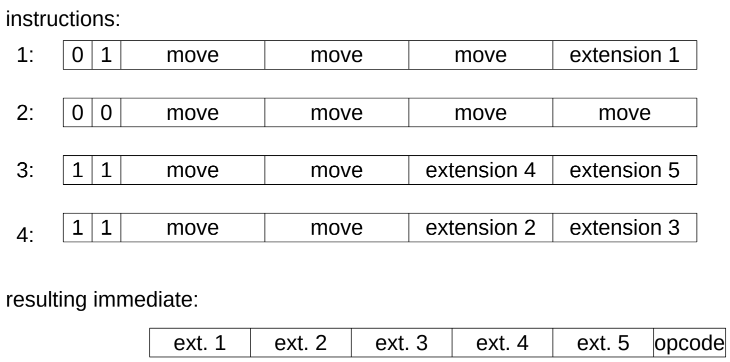


Figure 4.28: Example of an immediate which is constructed by shifting extensions.

Figure 4.28 shows an example where an immediate is constructed by shifting extensions from more than one instruction into the immediate extension register. This example assumes that only two move slots per instruction can be used for immediate extensions (this is a parameter), and that complete immediates require five extensions. The picture shows how the immediate looks when it is used in one of the move operations of the fourth instruction. The opcode field must be specified by the source ID of the move instruction which uses the immediate.

The main disadvantage of using separate instruction bits to specify which move slots are immediate extensions, is the increase of the number of instruction bits. Chapter 7.4 describes some methods to make this more efficient, but they are not implemented yet.

Realization and verification 5

This chapter describes the part of the design trajectory from the parameterized VHDL level to a complete processor chip layout. The complete design path was shown in figure 1.2. The verification procedure is also described in this chapter.

The path from parameterized VHDL to a processor layout is shown with more details in figure 5.1. First, the parameter references are resolved by the m4 preprocessor tool. Adding a components package[1] is necessary to make the VHDL description complete. A special tool has been developed to generate this components package. Appendix F describes this tool. During the second step, the synthesis tool *Synopsys* is used to generate a gate level description out of the VHDL description. The results of this step are described in section 5.1. In the last step a complete processor layout must be generated. The intended tool for this step, *Cadence*, turned out to be useless. We did not succeed in generating a layout with this program. It has very bad manuals, and the examples from the manual did not work. In the future *Compass* might be used for this step.

In the whole design trajectory errors can be made. Thorough testing should maximize the chance that all errors are discovered before the processor layout is sent to the manufacturer to be realized on silicon. The test procedure is explained in section 5.2.

5.1 Synthesis

The tool which is used for the synthesis step is Synopsys, with libraries from European Silicon Structures (ES2). The goal of this step was not only to generate a highly optimized gate level

[1]Each VHDL unit that uses subunits needs the complete interface description of these subunits. We choose to collect these interface descriptions in a components package, which is shared by all units. The components package is described in appendix F.

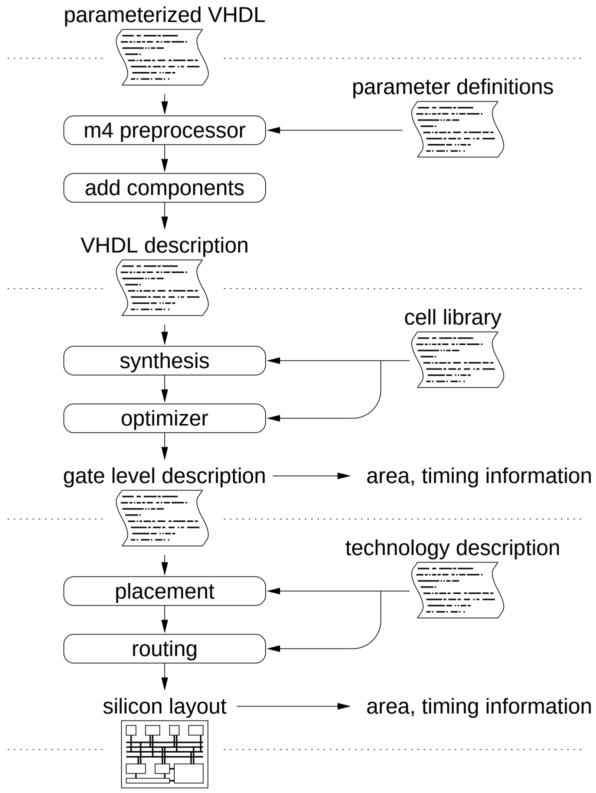


Figure 5.1: The design path from parameterized VHDL to a processor layout.

design, but especially ensuring that the VHDL description of the instruction fetch unit is accepted by Synopsys. As Synopsys accepts a relatively large subset of VHDL, this does not guarantee that other synthesis tools also accept this description.

5.1.1 Area usage

The area usage is estimated with Synopsys. As this is not a layout tool, it cannot calculate the exact area usage. All numbers specify the area without the wiring area, because the wiring cost is not known before the routing step has been taken[2]. Routing was not possible because Cadence (the layout and routing tool) did not work properly.

Table 5.1 shows the used area for several different configurations of the instruction fetch unit. The numbers are calculated by Synopsys, when the wiring area and the ram cells inside the cache and the TLB are not taken into account. The design is optimized for area with medium effort, a timing constraint of a 100 MHz system clock, under typical operating conditions with the ES2 1.0 μm library. It uses a 32 bit instruction address (both virtual and real), and 2 move busses with 15 instruction bits each. The instruction cache contains 1024 instructions on 64 cache lines, and is two-way set-associative (FIFO replacement). The TLB (if available) uses a page size of 4096 in-

[2]Synopsys has an area and timing estimator for the wiring which is based on wire load models. It is difficult to predict how representative this approximation will be. It reports that wiring is responsible for more than 50% of the total area of the instruction fetch unit.

structions, and has 16 entries (1 per line), using a two-way set-associative (LRU replacement) configuration.

Configuration			**Area**
pipeline stages	privileged instructions	virtual memory	(mm^2)
2	no	no	1.5
2	no	yes	1.8
2	yes	yes	2.0
3	no	no	1.7
3	no	yes	2.2
3	yes	yes	2.2

Table 5.1: The area needed for the instruction fetch unit, without wiring area and ram cell area.

The given area specifications must be seen as approximations. Synopsys is not very stable in its optimizations. It is therefore difficult to reproduce these results, and using other optimization parameters may decrease the reported area significantly. The order in which the subunits are processed also influences the results. The area reported for the subunits which are the same in more than one configuration (e.g. the cache) was not the same for each configuration. This explains why kernel mode support increases the used area by 0.2 mm^2 for a two stage pipeline, while it does not significantly increase the reported area for a three stage pipeline.

The ram cells are not included in this area report, because Synopsys cannot handle them. The ram cells are pre-compiled structures, which must be included in the design during the placement step. Fortunately it was possible to generate the ES2 library ram cells, and to get an area report for them. If the cache and TLB parameters which are specified above are used, the instruction cache uses four ram cells with a total area of 15.5 mm^2, and the TLB uses two ram cells which together take 0.7 mm^2. These numbers do not include the interface which is needed to let those ram cells meet the uniform ram cell specifications from section 4.3.1.

The area used by the state unit is specified in table 5.2 for four configurations. The wiring area is not included. If the state unit has support for privileged instructions, its user_mode output signal is the same as the exc_enabled signal, which does therefore not influence the used area. This is the reason why versions with privileged instruction support are not included.

Configuration		**Area**
pipeline stages	virtual memory	(mm^2)
2	no	0.11
2	yes	0.17
3	no	0.13
3	yes	0.18

Table 5.2: The area needed for the state unit, without wiring area.

The area usage of the immediate unit is highly determined by the number of registers. This is clearly shown by table 5.3, which gives the area reports of Synopsis for four different configurations of the immediate unit. A three stage transport pipeline causes a register because the immediate extension is read by the immediate unit during the DC pipeline stage, while it is used in the MV pipeline stage. If immediate extensions can be specified in an instruction before the one which uses it, another register is needed. The results are valid for an immediate unit with one short immediate output (5 bits), and one long immediate output (extension + 4 bits). An extension can be 15 or 28 bits long (1 or 2 instructions). The full bus width is 32 bits.

Configuration		**Area**
pipeline stages	place of instruction with extension	(mm^2)
2 stages	same	0.04
2 stages	same + before	0.22
3 stages	same	0.17
3 stages	same + before	0.32

Table 5.3: The area needed for the immediate unit, without wiring area.

5.1.2 Critical path analysis

Timing information for the instruction fetch unit is not available because the placement and routing could not be done. An estimation by Synopsys was not possible because the ram cells are not supported by Synopsys. Furthermore, the timing of the external signals is not known because a complete processor design is not available yet[3].

Though the exact timing data are not available, we can expect the selectors before the PC to be in the critical path. These selectors are responsible for the jump functions, and for writing the exception vector into the PC at an exception. If the jump function constrains the performance, the following path is part of the critical path inside the instruction fetch unit:

1. The data, or the squash signal of input socket 1, through which a jump or function call is triggered,

2. The selectors, which select the input value for the PC register,

3. The selector inside the virtual memory datapath, which selects the read_offset signal for the TLB (see figure 4.6),

4. The selector, which selects between the read_offset and the write_offset inputs in the TLB (see figure 4.17 of the generic cache),

5. The address input of one of the ram cells inside the TLB.

Closely related is another possibly critical path. This path also leads through the PC selectors, but has a slightly different origin:

1. The data, or the squash signal of input socket 3, through which a trap instruction is given,

[3]The processor core which is used for testing (section 5.2) cannot be used because its VHDL code is not optimized in any way, and would therefore generate useless results.

2. The exception controller, which must select the exception vector,
3. The selectors, which select the input value for the PC register,
4. The selectors between the input of the PC register and the address inputs of the ram cells of the TLB.

These paths are part of the move (MV) stage of the transport pipeline. This pipeline stage is probably the pipeline stage which restricts the cycle time, especially if the decode (DC) and move (MV) pipeline stages are combined, as in a processor with a two stage transport pipeline. If the cache and the TLB are large, and use a set-associative organization, the instruction fetch (IF) stage may be critical. In that case, the critical path will be:

1. The rising clock edge, which starts a TLB lookup,
2. The tag check for the TLB outputs,
3. The selection of the correct physical page number from the outputs of the TLB ram cells,
4. The tag check of the cache outputs, using the physical page number from the TLB[4],
5. The selection of the correct instruction from the outputs of the cache ram cells.
6. The selectors, which select the new input of the instruction register.
7. The input of the instruction register.

5.2 Testing the design

The goal of testing is to maximize the chance that the processor chip meets the requirements which are defined at the start of the design process. All design steps are shown in figure 5.2. If all steps are guaranteed to be taken correctly, the fabricated processor must be correct.

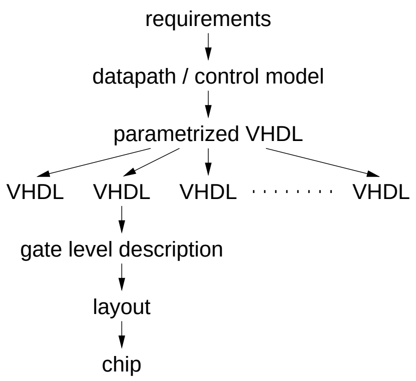


Figure 5.2: All design steps have to be verified to ensure that the processor chip will meet its requirements.

[4] This step might also be delayed by the cache lookup itself

One of the main problems of the verification procedure is that not for one, but for a large number of processor configurations the correctness must be proven. A good design can help to make this easier. The modules in the instruction fetch unit are chosen such that each module depends on as few parameters as possible.

The instruction fetch unit is tested on the VHDL level. The instruction fetch unit design had to be specified as parameterized VHDL, and should therefore be tested on that level. As testing with unresolved parameters is not possible, several VHDL configurations are tested. The goal of the verification procedure is therefore redefined as ensuring that the requirements are met at the VHDL level.

The step from datapath and control model is verified by testing each module separately for each configuration. It is generally not possible to test all combinations of configurations, states and input signals because this usually leads to a huge number of possible combinations. Therefore the test is restricted to all state transitions and the critical combinations of input signals of the main configurations. This should be enough to filter out all errors, though it cannot be proven.

The tests which are described above can verify that the VHDL code implements the datapaths and control models correctly, but cannot ensure that the used datapath and control models themselves are correct. To test this step, a whole processor environment was created. This processor was connected to a piece of memory, with test instructions. The instruction fetch unit run correctly in this test environment, so the chance is high that it will also operate correctly in its final environment, though this test cannot prove it.

We implemented a very small, but complete, timesharing operating system in the test program. On this operating system environment, two different user mode processes were run. This test showed clearly that it is possible to run an advanced operating system on the instruction fetch unit.

The simulator which is used for the tests is the Optium by Vantage Analysis Systems. A sample output generated by this simulator is shown in figure 5.3. The picture shows a part of the test program for the complete processor. At time point 62985 ns an interrupt is given manually, which starts the interrupt handler (address 0x00000040) at time 62990 ns (the first rising clock edge). At the same time a load instruction is triggered, which causes a data TLB miss. The data TLB miss is detected at time 63000 ns, and the data TLB miss exception handler is started, while the interrupt handler is stopped. As the interrupt bit in the exception source register (the rightmost bit in exc_sources) is still high, it will cause an interrupt immediately after the data TLB exception handler returns.

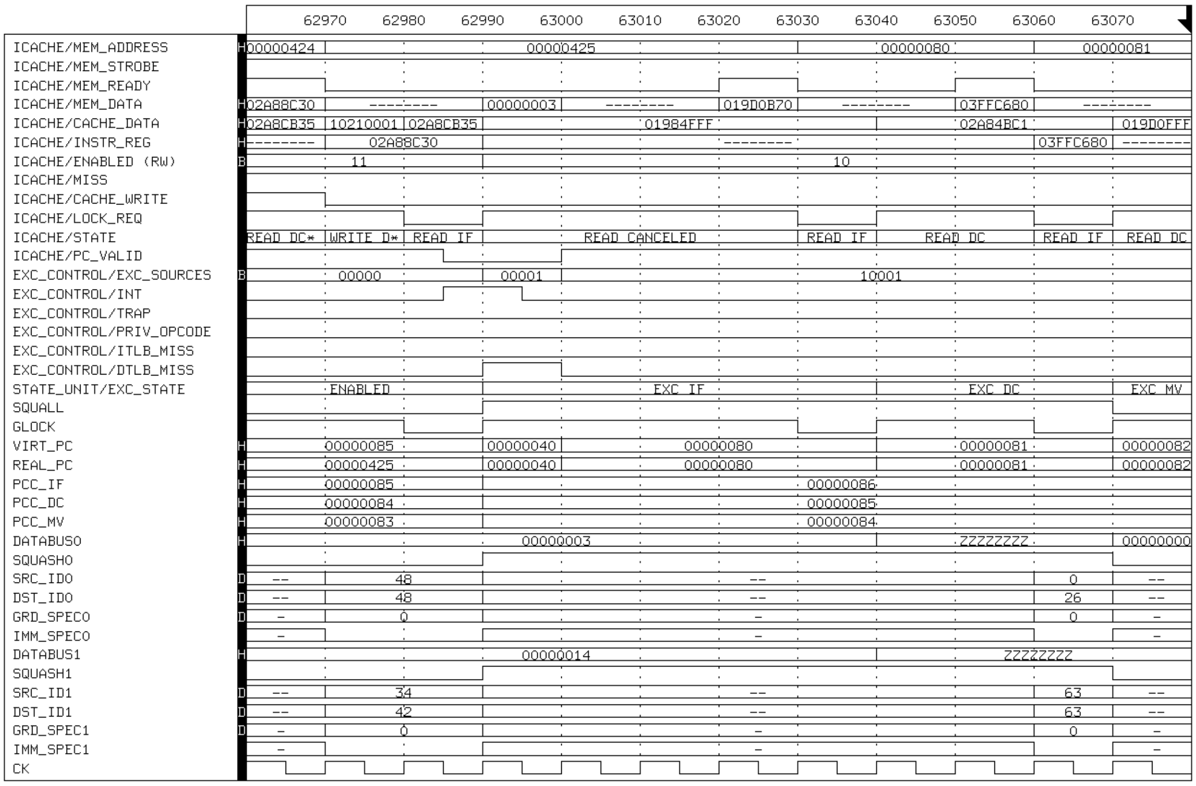


Figure 5.3: Example of the output of the VHDL simulator.

6 Usage of the instruction fetch unit

This chapter describes the instruction fetch unit from a behavioral point of view. The functions of the instruction fetch unit, the immediate unit, and the state unit are described, without specifying how these functions are implemented. In general, the most complex versions of these units are described. Some of the functions, like virtual memory, or privileged instructions are not available in all versions. Every function which was specified in the requirements from chapter 3 will be described here.

6.1 Executing jumps and function calls

The instruction fetch unit uses page relative jumps and function calls. Three different jump offsets are possible. These three sizes can be specified by parameters. The number of (virtual) instruction address bits is defined as the largest possible jump size. Every instruction address specifies an instruction, and not the memory location of the instruction. The instructions are therefore always numbered 0, 1, 2, … regardless of their corresponding size in memory.

The page relative jumps are relative to the page of the last delay slot instruction. When using a four bit page size, a typical jump looks like:

```
0x0E: 8 -> jump_s, ... ; short jump instruction
0x0F: ...              ; delay slot 1
0x10: ...              ; delay slot 2
0x18: ...              ; jump target
```

There are two delay slots in this example. This means that a three stage pipeline is used. A two cycle pipeline would have caused only one delay slot.

The only difference between jump and call instructions is that a call saves the return address, while the jump does not save anything. The call saves the address of the last delay slot plus 1, which is the address of the first instruction which was not executed before the function call. In the case of the previous example, the address *0x11* would have been saved in the function return address register.

6.2 Using the instruction cache

The instruction cache has no invalidate instruction, because it would have required a lot of hardware. The procedure to initialize the cache is to execute a number of subsequent instructions, with the cache writing enabled. These instructions will overwrite the cache entries. When the number of executed instructions is at least the number of cache instructions, the whole cache is overwritten, and therefore guaranteed to be valid.

As the cache is addressed with physical addresses, an invalidate is only needed after a reset, and after a write to a code page. The cache does not have to be invalidated at process switches. After a reset, cache reading must not be turned on until the cache is initialized.

The state unit provides a write enabled and a read enabled signal for the instruction cache. Cache writing can be disabled to prevent the cache from overwriting its possibly useful contents. Cache reading must be disabled when the contents are not reliable, e.g. after a reset, or when the cache is broken. Turning these signals on and off is described in section 6.6, which describes the state unit.

6.3 Exception handling

The processor has two modes: user mode and exception mode. The main difference between them is that exceptions are only possible in user mode. In exception mode, all exception requests are ignored. The exception types which are supported by the instruction fetch unit are:

reset A reset is a very special kind of exception. It cannot be masked. Even exception mode cannot stop a reset. A reset does some initializations too.

interrupt An interrupt is requested by external devices.

trap A trap instruction can be used by user processes to ask the OS to do some operations (system calls). A trap is generated after a user mode process writes into the trap register. The value in this trap register is available in exception mode. The size of the trap register is parameterizable.

privileged instruction executed by a user mode process. Instructions which are restricted to exception mode processes are called privileged. They cause an exception when a user mode process tries to use them. Section 6.5 describes them in more details.

TLB miss A TLB miss is a page translation which cannot be found in the instruction TLB, or the data TLB in the load-store unit. A data TLB miss causes an exception immediately after the instruction which issued the load or store. An instruction TLB miss causes an exception just before the faulting instruction would be executed.

unit	**register**	**words**
instruction fetch	PC chain	p
	trap	1
	function return	1
state	cache enable	1
	TLB enable	1
immediate	extension	1
load-store	load result	2
	store operand	1
register	register	r
other	operand	n
	result	n
total		$8 + p + r + 2n$

Table 6.1: The state which has to be saved at na exception. The PC chain, the trap register and the cache and TLB enable signals only have to be saved at a process switch.

When one of the above exception requests is detected by the instruction fetch unit, the processor goes into exception mode, and jumps to the exception handler code. Every type of exception may have its own starting address for its exception handler. The instruction which is in the *Move* (MV) pipeline stage at the time of the exception is executed if the processor is not locked. The other instructions in the pipeline are squashed.

The exception source is recorded in a special register (*exc_sources*) inside the instruction fetch unit. The interrupt, trap, and the privileged opcodes each have a bit. Every TLB gets a separate bit. The register can be read and written in exception mode. The format which should be used is:

(msb) 4	3	2	1	0 (lsb)
dTLB miss	iTLB miss	privileged opcode	trap	interrupt

The bits in this register are always set if their corresponding exception condition occurs, even in exception mode. They must be reset manually. When the trap bit or the interrupt bit is still high after the return from exception, a new exception is generated immediately.

An exception handler usually starts with state saving, and ends with state restoring. All registers which may be used before the control is returned to the interrupted process should be saved. The state of the instruction fetch unit which may have to be saved, is in the program counter chain, the function return address register, and the TLB. In some cases, the contents of the trap register are also part of the state. The function return address register was described in section 6.1, and saving the TLB state will be described in section 6.4.

Table 6.1 shows which processor state must be saved at an exception. The contents of the TLB cannot be read, and can therefore not be saved. The trap register may have to be saved to specify whether the process still has a waiting trap request and if yes, which trap request. The state of the instruction fetch unit depends on the number of pipeline stages (p). We assume that the saved cache enable signals and the saved TLB enable signals each fit in one word. The state which has to be saved for the functional units depends on the number and type of the FUs. We assume there are *n* FUs which are not specified separately in the table, and that these FUs require that one result and one operand register have to be saved.

The program counter (PC) chain contains the addresses of the first few instructions which must be executed after the return from exception. The number of registers in the PC chain is equal to the number of pipeline stages. In user mode, this PC chain runs along with the pipeline, but is locked in exception mode. In exception mode, the first register of the chain (*PCC-IF*) can be written, and the last (*PCC-MV*) can be read. Writing the PC chain lets the chain proceed one step.

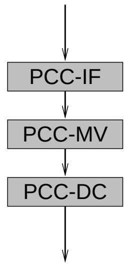


Figure 6.1: The structure of the PC chain (three stage pipeline).

The return from exception procedure uses the PC chain. All registers of the chain are moved to the program counter one by one using the special return from exception (*rfe*) instruction. This instruction is an absolute jump instruction, which has some side effects. The PC chain is restarted again at the first *rfe* instruction, and the processor goes into user mode after the last delay slot. To ensure that all pipeline stages are restored correctly, the delay slots must also contain moves from the PC chain to the PC using the *rfe* instruction. For a three stage pipeline, the return from exception code is:

```
e+n-3: pcc -> rfe, ... ; return from exception instruction
e+n-2: pcc -> rfe, ... ; delay slot 1
e+n-1: pcc -> rfe, ... ; delay slot 2
u    : ...             ; first user mode instruction
```

6.4 Using virtual memory

The virtual memory system of the instruction fetch unit uses a *Translation Look-ahead Buffer*. This TLB contains a part of the page translation table. The virtual page number from every instruction is looked up in the TLB during the instruction fetch (IF) pipeline stage. If a program tries to execute an instruction for which no page translation was available, an exception is given.

The exception handler may update the TLB, or terminate the program in case of a program error. The faulting instruction number is in the *PCC-MV* register. Updating the TLB requires two steps:

1. Write the virtual page number to the TLB. This is usually the page of the faulting instruction.

2. Write the corresponding physical page number into the TLB. A valid bit is written too.

The page numbers must be written into the TLB with the same alignment as if they were normal instruction numbers. It is possible to write the faulting instruction address from *PCC-MV* directly into the TLB as the virtual page number. The corresponding physical instruction number, which must be written into the TLB to specify the physical page, can be calculated from its memory address by dividing the memory address by the number of memory words needed for one instruction. The valid bit for the physical page number is on the least significant bit position. The data which are written to the TLB should have the format:

page number	not used	X

The *X* bit is the valid bit which specifies whether the page is executable. The *X* bit is only needed when a physical page is written. The number of bits for the virtual page and for the physical page number is not necessarily the same. The virtual address space may therefore be larger than the physical address space.

The TLB can be enabled and disabled via the state unit, which is described in section 6.6. In general, the virtual memory will be enabled (TLBs are used) in user mode. It is advisable to disable virtual memory in exception mode, because TLB misses cannot be handled. If the TLB is disabled, the translated address is the same as the virtual address. The instruction TLB can be updated while it is in use.

Invalidating the TLB may be necessary after a reset, and during a process switch. If the TLB contains n entries, writing n subsequent virtual pages and their translations will initialize the TLB completely. The TLB cannot be read to retrieve its contents, which makes it impossible to save the contents of the TLB at a process switch. An alternative may be to let the TLB exception handler maintain a copy of the TLB contents in memory. When a (set-associative) TLB uses a *Least Recently Used* replacement algorithm, the exception handler cannot know which entry will be replaced. In that case, the memory table should use another algorithm, like *Least Recently Written*.

6.5 Using privileged instructions

The OS support requires that some instructions are not available to user processes. These instructions are called privileged, and generate an exception when a user mode process tries to access them. An instruction should be declared privileged if it may affect the correct behavior of the OS or any other process if the instruction is executed by a user mode process. In the instruction set summary in appendix A, it is indicated which instructions of the instruction fetch unit and the state unit are privileged.

The privileged instruction detector scans the instruction stream for privileged opcodes. It generates an exception if a privileged instruction reaches the MV pipeline stage. The detector does not look at the squash signals. As a result, privileged instructions which will not be executed because the guard expression evaluates to *false*, also cause an exception. The exception occurs just before the instruction which contains the privileged opcode. This ensures that the instruction cannot do any damage before the exception. The exception handler may decide to terminate the process which tried to execute the privileged instruction.

When the operating system (OS) emulates a virtual processor, the exception handler may have to simulate the privileged instruction, and then restart the interrupted process. Virtual processor emulation is an advanced OS task, in which the main OS emulates a processor in a way that a second OS can run as a user process, but still operate correctly. The second OS cannot see the difference between a bare processor on which it thinks it is running, and the combination of the processor and the main OS, on which it actually runs. This requires that all instructions which may behave differently in user mode and in kernel mode should be declared privileged.

The instruction fetch unit, state unit and the immediate unit support the processor emulation function. The exception handler of the main OS knows which instruction contained the privileged opcode because the instruction address is in the *PCC-MV* register. It can emulate all moves in that instruction if the main OS keeps the privileged registers of the second OS in memory. The exception handler must restart the interrupted process on the first cycle after the privileged instruction.

The described procedure only works if state changes caused by the privileged instruction take effect immediately. The jumps do not meet this requirement because they take effect after one or two delay slots. In the case of the jumps, this is no problem, because the PC chain can catch the necessary effects. The only instruction which is a real problem, is the return from exception. The transition from exception mode to user mode is delayed by the number as delay slots, and is not simulated by something like a mode chain. Therefore, the only way to emulate the return from exception correctly is to emulate its delay slots too. The exception handler must then restart at the process which was restarted by the emulated return from exception.

6.6 Using the state unit

Enabling and disabling the caches and the TLBs is the function of the state unit. The state unit is also responsible for the saving and the restoring of the enabled signals at an exception. The state unit controls the instruction cache and the instruction TLB as well as the data cache and the data TLB. At a reset the state unit disables all caches and TLBs.

The state unit has two functions for enabling and disabling the caches and the TLBs at runtime. One instruction writes the cache enabled signals for both the instruction fetch unit and the load-store unit. The other instruction writes the TLB enabled registers. In exception mode, there are two functions which write the saved enabled signals. One writes the saved cache enabled signals, the other writes the saved TLB enabled signals. The same four registers can be read too.

Each enabled signal has two versions in the state unit. One version is only used in user mode, the other in exception mode. At an exception, the focus switches from the user mode signals to the exception mode signals. After a return from exception the user mode signals are used again. This principle makes that the cache and TLB enabled signals after an exception are the same as the cache and TLB enabled signals which were used by the previous exception mode process just before the return from exception. When the exception handler writes the saved signals, it updates the user mode signals. At a reset, the caches and TLBs are turned off. The enabled signals for exception mode are set to zero.

The format for the current and for the saved enabled signals is the same. The bit order of the cache enabled signals is:

(msb) 2	1	0 (lsb)
dcache read/write	icache read	icache write

The format of the TLB enabled registers is:

(msb) 1	0 (lsb)
dTLB enabled	iTLB enabled

When a parameter for the load-store unit is changed, the effects are visible at the next instruction. Changing parameters for the instruction fetch unit has a delayed effect. When instruction i changes an enabled signal, the first instruction which uses the changed parameter is $i + n$, where n is the number of pipeline stages.

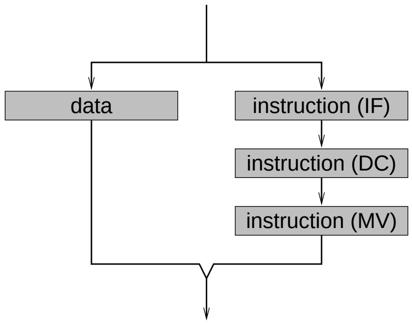


Figure 6.2: Structure of the user mode enabled signals when an exception mode process uses them (assuming a three stage pipeline).

The exception handler must be able to save the effects of this delay. For this reason, there are n registers which save the instruction cache enabled signals, and n registers which save the instruction TLB enabled signals (n is the number of pipeline stages). Figure 6.2 shows how these signals can be read and written in exception mode. The instruction fetch unit signals work just like the PC chain. In exception mode, the chain is locked, and only proceeds after a write. The chain is restarted after a return from exception. In exception mode, the upper (IF) register is updated on a write, while the MV register is visible for reads. A write immediately updates the signal for the load-store unit. This exception mode behavior does not affect the user mode behavior. In user mode, only the IF register can be accessed.

The MV register is used for every instruction which is started by a return from exception (rfe) instruction. The first rfe instruction starts the chain. As a result, the values in the MV registers are used for the first instruction after the return from exception, and the chain proceeds one cycle. If the instructions in the delay slots of the first rfe instruction are also rfe instructions, the values from the DC (three stage pipeline) and the IF registers are used for the instructions which are started by those rfe instructions. This is the reason why the delay slots of the first rfe instruction must also contain rfe instructions, instead of normal jumps.

6.7 Using immediates

The programmer can specify immediates in the instruction stream. For short immediates, only the opcode of the source ID is available. Figure 6.3 shows this principle. A processor generator parameter decides whether this short immediate should be zero extended or sign extended to the full bus width. Using short immediates can be very efficient, because it does not need extra instruction fields. The disadvantage of using short immediates is that the number of immediate bits cannot be greater than 6 or 7 for practical configurations. Therefore, an extension for the short immediate system is designed.

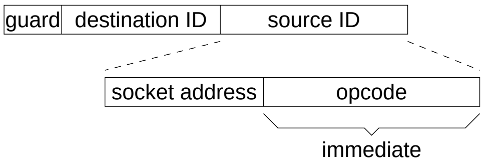


Figure 6.3: Using the opcode part of the source ID to create short immediates

Extensions may be provided by the same instruction as the one which uses the immediate, or an instruction before it. If one instruction cannot provide immediate extensions which are long enough, a shifting mechanism can be used. This leads to three different versions of the immediate unit, which only differ on their way of constructing long immediates. Processor generator parameters specify which version is used. In order of increasing complexity, the three versions are:

- Immediate extensions can only be specified in the same instructions as the one which uses them. If a long immediate is used, but no extensions are specified in that instruction, the immediate is treated as a short immediate, i.e. only the opcode in the source field is used.

- Immediates can not only be specified in the same instruction, but also before the one which uses them. The immediate unit now uses a register to store the extension. As a result, this version requires more chip area. An instruction which contains an immediate extension overwrites the whole extension register. An instruction which uses a long immediate will use an extension from the same instruction, if it is available. Otherwise, the value which was stored in the extension register will be used. The extension register can be written via an input socket for state restoring purposes.

- If one instruction is not sufficient to specify extensions which have enough bits, the third version can be used. This version does not throw away the contents of the extension register if another instruction specifies a new extension, but it shifts the old contents towards the most significant side of the register. The new extension is placed on the least significant position. This shift mechanism offers the possibility to create very long immediates by using several instructions which all specify a part of the extension.

Whether an instruction contains extensions is specified by some extra instruction bits. For every move specifier (a combination of the guard, destination and source ID for a move operation) there is one extra bit. The number of move specifiers that can be used for an extension is calculated from the necessary number of bits to extend the short immediate to the full bus width. Optionally, the designer can specify a lower maximum. In the case not all move specifiers are needed for the extension, the busses with the lowest bus numbers are used.

If a move specifier is used as a part of the extension, it is always on the same part of the extension. The move specifier for bus 0 is always on the least significant position in the extension. The whole extension is overwritten if one of the move specifiers is used as an extension, unless extension generation by shifting is selected (the third version of the immediate unit is used). The parts of the extension for which the corresponding move specifiers are not used as an extension (the

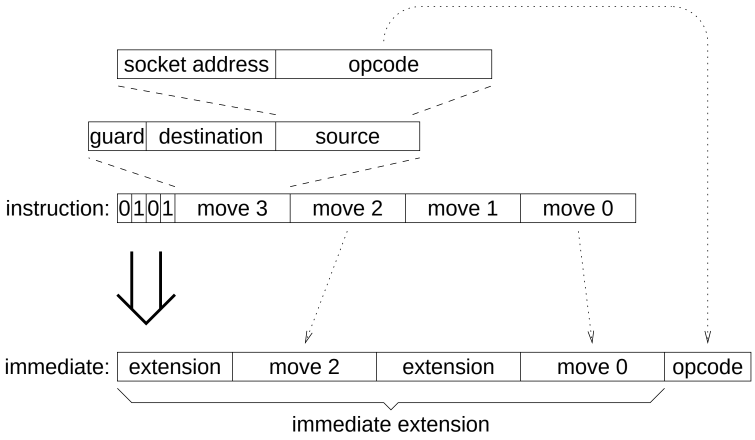


Figure 6.4: Specifying extensions for a short immediate

selecting bit is zero) are filled in by zero extension or sign extension. If four busses can be used for the extension, and only the bits for the busses 0 and 2 are set, the extensions for bus 1 and 3 are generated by zero or sign extensions of the move specifiers for the busses 0 and 2, respectively. Figure 6.4 shows this schematically.

The immediate unit has two output socket types. The first socket is only used by short immediate; the other by the extended immediates. Each output socket of the immediate unit should be connected to one bus only, but the immediate unit may have several output sockets of the same kind. This makes it possible to use immediates on more than one bus in the same instruction. Each bus uses its own opcode, and therefore gets its own immediate. The long immediate output sockets may also each use a separate opcode, but their extension will be shared. The two socket types may use their own opcode size (the number of bits for the short immediate, or the bits added to the extension to create the long immediate).

The extension register in two of the three versions of the immediate unit is part of the processor state which has to be saved at an exception. The exception handler can read the register via the long immediate output socket, with any opcode. The immediate units with the extension register also have an input socket to write the extension register. The extension must have the same alignment as if the extension were a part of a long immediate. As a result, the value which was saved by the state saving code can be used directly for the state restoring.

The state saving code generally uses the immediate unit before the extension register can be saved, because the load-store unit state must be saved first. The designer has to make sure that the exception handler code can bypass the extension register, or that the extension register can be saved first. A simple way to bypass the extension register is the usage of short immediates. Therefore an immediate unit with an extension register generally needs at least one short immediate output socket.

7 Extensions

Though the instruction fetch unit described in this report is very complete and well optimized, some extensions and improvements are possible. Some of the extensions are intended to make the MOVE processor applicable for a wider range of applications. Other extensions are intended to make the processor more efficient or faster in some configurations. For every extension, I will specify its use and requirements. Possible implementation principles are given too.

Section 7.1 describes busflashing, which is intended to improve the exception support. If exception support is not necessary at all, section 7.2 describes some possible optimizations. If the cache lookup is in the critical path, it may be useful to spread the cache access over two cycles. This is explained in section 7.3. Finally, section 7.4 describes how the immediate system can use fewer instruction bits.

7.1 Busflashing

The time which is needed for exceptions is largely determined by the time needed for the state saving and restoring. This time can be very high, especially for large processor configurations. Busflashing is a way to decrease the amount of state that has to be saved after an exception. The busflashing exception support mechanism is described in [Cor95].

Busflashing also solves a problem that occurs in MOVE processors with a relatively high number of internally pipelined functional units. These units require that their results are saved in the first cycles of the state saving code. The results must be restored in the last cycles of the state restoring code. Every internally pipelined unit needs at least one move bus to restore its state. Therefore the number of busses can easily become too low for a correct exception behavior.

The main principle of busflashing is that a special FIFO buffer, called the *busflash*, saves the

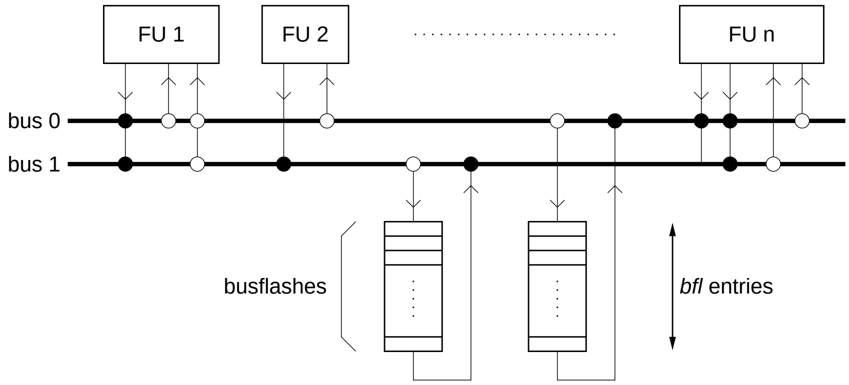


Figure 7.1: The hardware required for busflashing

destination ID and the transported data of every (user mode) instruction (see figure 7.1). The buffer has *bfl* entries. When the busflash is frozen at an exception, it contains the state changes which are caused by the *bfl* instructions before the exception. If no state has a lifetime larger than *bfl* instructions, the state of the processor can be reconstructed completely by restarting at *bfl* instructions before the instruction at which the processor was interrupted. The necessary data operands and destination IDs are in the busflash.

Busflashing restricts the number of state that has to be saved after an exception because it is not necessary to save and restore all results from all functional units. The trigger operations which created the results are still available. All registers with a lifetime shorter than the busflashing window do not have to be saved. The busflash itself only has to be saved at process switches.

Busflashing removes the need to save and restore the results from all internally pipelined functional units in parallel. When the busflash is executed again after the exception, the results are reconstructed in the same way as they were created. The number of busses is not restricted with busflashing exception support.

The busflashing mechanism has two major disadvantages. First, the busflash buffers use a lot of chip area. Second, the scheduling freedom decreases because of the restricted lifetime of the processor state. Especially general purpose registers and operand registers within functional units may have a very long lifetime. Compiler conventions may require that these registers are saved and restored on the traditional way.

The instruction fetch unit needs some modifications when busflashing is implemented. The exception state diagram (figure 4.9) has to be extended between the *disabled* and the *rfe_if* states. In the new states, and the *rfe_if* and *rfe_dc* states, the instructions from the busflash are executed again. Busflashing also influences the way the program counters (and the corresponding instruction cache and TLB enabled signals) are saved. The instruction fetch unit only has to save the address of the first instruction after the return from exception (the equivalent of the *PCC-MV* register), which is written into the program counter on the transition to the *rfe_if* state. The busflash contains the information which is needed to restore jumps and calls.

The instruction fetch unit is not the only unit that needs modifications. The whole instruction distribution structure has to be extended in a way that the destination IDs and the source registers can be overruled by the busflash unit. The changes which were made to the PC chain, are also needed for the saved instruction cache and TLB signals in the state unit. The load-store unit needs some modifications which ensure that the memory contents are not changed accidentally, and that memory mapped I/O operations are not triggered twice.

Implementing busflashing is not an easy operation, but for large processor which use exceptions extensively, busflashing can be useful to decrease the penalty of exceptions. In the decision whether a busflashing mechanism is implemented, the large amount of chip area that busflashing consumes should also be taken into account. The area can be decreased if the length of the busflash window is shortened, but this may affect the code efficiency.

7.2 Removing exception support

In some small or highly specialized configurations, exception support may not be necessary. There are however currently no versions of the instruction fetch unit without exception support. These versions can be smaller because the PC chain is not needed, and the control becomes easier. The sophisticated exception controller can be replaced by a simple reset controller, and one state and a number of transitions are removed from the cache controller.

The state unit can get much simpler too. As exceptions are not possible, the cache enabled signals do not have to be stored. All that remains is a small register with two bits for the instruction cache, and one bit for every data cache.

The immediate unit does not change significantly. The only simple optimization which is possible is the removal of the write socket for the extension register. When exceptions are removed, the alternative version of the immediate unit, in which the extensions are specified in the instructions after the one which uses the immediate, gets much simpler. As the main disadvantage was its complex exception behavior, this version may get better than the current implementation when exception support is removed. Especially in the three stage pipeline version, that version is a strong alternative, because it uses the instruction bandwidth more efficiently, while the hardware requirements are not higher.

In functional units which use true virtual time latching (TVTL), another optimization is possible. They can be triggered safely by writing into the operand register. This is not possible when exceptions are supported, because the state saving and restoring code assumed that only writes into trigger registers can start a new operation.

7.3 Pipelined cache access

In the instruction fetch (IF) pipeline stage the complete cache access is performed, including the tag check. If the cache lookup is critical, and the decode is not, the performance may improve if the tag check is shifted into the decode (DC) pipeline stage. Pipelining the cache lookup can be particularly useful in processors with a three stage pipeline (IF-DC-MV), because it is not unlikely that the cache access in these processors is in the critical path, while the DC stage is not entirely used in most cases.

The two cycle cache access has some disadvantages too. The most important are:

- The two cycle cache needs significantly more area. For an n-way set-associative cache, $n + 1$ extra tag registers are needed (one for every cache bank, and one for the PC). The cache also needs n instruction registers instead of one, because the selection between those

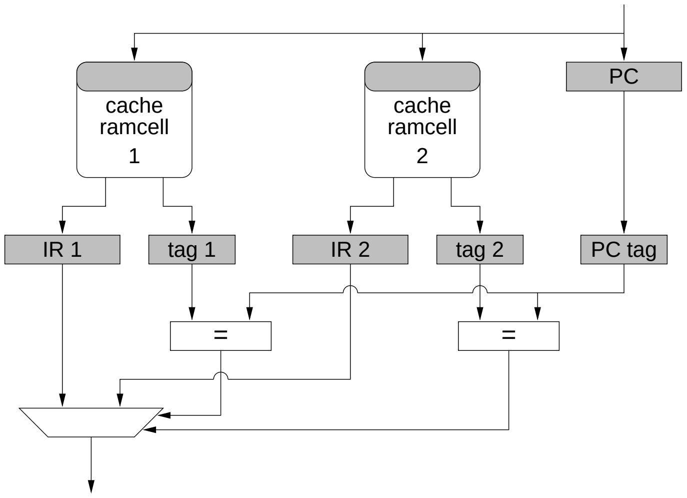


Figure 7.2: The datapath of a (2-way set-associative) pipelined two cycle cache.

registers is shifted towards the DC pipeline stage. Figure 7.2 shows where the extra registers are created for a 2-way set-associative cache when the TLB is left out. Only the PC and one instruction register (IR) is needed for a one cycle cache.

- There is only a one cycle implementation of the generic cache available, but the pipelined instruction cache needs a two stage version, with a different interface. Using only the two cycle generic cache is possible, but requires that the TLB lookup gets two cycles as well, and inhibits the use of one cycle loads in the load-store unit.

- Set-associative caches need the result of the tag check to determine which cache bank contains the correct instruction. The tag check delays the start of the decode, especially when a set-associative, two cycle TLB is used, for which the cache tag is not available until the TLB tags are checked.

It is possible, but not easy, to make some of the above disadvantages less severe. Combining the designs of the cache and the TLB can remove one of the tag registers. Adding some extra comparators may speed up the tag check when a set-associative TLB is used, though this requires extra hardware, and a more complex generic cache interface.

The cache control will get more complex. One of the major changes in the state machine is the lock request signal which will become a function of the tag compare results and the state, instead of a function of the state only. Another modification is the increase of the minimum memory cycle time to three cycles, because after a write cycle two additional cycles are needed. The new address must be read (cycle one) and its tags must be compared (cycle two) before a new write is possible (cycle three). The state diagram for a one cycle cache accepts a new instruction from memory after two cycles, though it adds wait cycles when cache writing is enabled. The two cycle cache will therefore not decrease the effective maximum memory bandwidth.

7.4 Specifying immediate extensions efficiently

For every move specifier (a combination of a source ID, a destination ID, and a guard specifier) that can be used as an immediate extension, one bit is needed to specify whether it should be used for an immediate, or for a regular move. These bits may take up quite some instruction bandwidth. There are more efficient ways to specify which move specifiers should be used for the long immediate construction.

The first option is to use the guard specifiers. When grd guard bits are used, 2^{grd} different guard expressions are possible, of which only one expression is needed for the immediate generation. For a typical number of $grd = 3$ guard bits, only one of the eight guard expressions is spent on immediate constructions. One of the disadvantages of this option is the lower number of bits which is available for the immediate extension, because the guard bits cannot be used any more. Another disadvantage is that it mixes the guard function and the immediate function. These are independent functions, and should therefore preferably be implemented orthogonally.

The second option is specifying only the number of move specifiers, instead of specifying exactly which move specifiers should be used as the immediate extensions. If move specifiers from n busses can be used as immediate extension, $n+1$ combinations (0, 1, …, n) are possible. This requires $\log(n+1)$ bits, instead of n bits, and will save instruction bits if $n \geq 3$. The only disadvantage is that it offers less flexibility in choosing the place of the extensions. If the number of move specifiers which are used as immediate extension is i, all move specifiers on the busses 0 to $i-1$ are used as immediate extensions. The current implementation allows the programmer not to use one of those busses. The missing bits are added by zero or sign extension. That is generally not very useful, because if more significant busses are needed for the immediate, the less significant bits are not likely to have their default values.

8 Conclusions and further research

All processor functions which are related to instruction fetching and processor control in the MOVE processor architecture have successfully been implemented in three highly independent units:

- The instruction fetch unit which fetches the MOVE instructions from memory via a memory interface unit. This instruction fetch unit has full support for advanced operating systems, which includes a virtual memory system based on a TLB, and support for privileged instructions.
- The function of extracting immediates out of the instruction is implemented in a separate immediate unit. This immediate unit offers support for very long immediates, while shorter immediate sizes are supported too.
- The control signals which do not only control the instruction fetch unit are implemented in a state unit. This unit controls the processor modes and the enable signals for the caches and TLBs in both the instruction fetch unit and the load-store unit.

In the implementation trajectory, the designer has to take numerous design decisions, of which the effects generally cannot be predicted precisely. The effects of those decisions may also depend on the application for which the processor will be used. Therefore the instruction fetch unit, the immediate unit and the state unit accept a lot of parameters, so the final configuration can be specified when the target application is known.

A generic cache template has been developed too. This cache implementation offers the designer a large number of cache configurations, which all use a uniform interface. In the instruction fetch unit, this cache template is used in the instruction cache and in the TLB. The generic cache can also be used in the load-store unit, though the current implementation of this unit does not use it.

Testing turned out to be very difficult. For each possible configuration, all states and all possible combinations of input signals have to be checked. Splitting up the design in several smaller subunits slightly reduced this problem, but the total number of combinations remained too high. The designs are verified by testing the state transitions and the critical combinations of the main configurations of the subdesigns, though this cannot prove that all configurations are completely correct. Testing a complete processor should guarantee that the interfaces are defined correctly, and that all necessary functions are implemented.

In one of the tests, the instruction fetch unit, the state unit, and the immediate unit were included in a complete processor description. On this processor, a small, but complete operating system was run successfully, which proved that it is possible to implement an advanced operating system for the MOVE architecture.

Some of the topics which can be used for further research are:

- Going down the whole design trajectory, and designing a complete processor layout. This will show how much time is needed to generate a processor layout when the final configuration is determined. It will also show how well the ram cells fit in the design.

- Adding a cache and TLB simulator to the MOVE framework. These simulators should be able to optimize the cache and TLB size and organization. This can be very important because the cache and TLB are responsible for a large part of the area needed for the instruction fetch unit.

- Implementing support for busflashing exception support. This will solve two problems of current MOVE processors. First, busflashing will make it possible to save and restore internally pipelined functional units efficiently, which is not possible in the current implementation. Second, it will reduce the penalty of exceptions because the busflash can take over most of the state saving and restoring which is needed at exceptions.

- Implementing test functions in the design of the instruction fetch unit. When a design is tested at the VHDL level, it is possible to inspect all internal signals, and sometimes to modify them directly if a subunit design is tested. When a physical processor is tested, this is generally not possible any more. Adding some functions which make it possible to inspect and modify some of the state registers might help to improve the testability.

Bibliography

[Arn95] Marnix Arnold. Synthesis and characterisation of MOVE configurations. Technical report, Delft Univ. of Technology, June 1995.

[Baa96] T.A. Baan. Design and implementation of a load-store unit for MOVE. Technical report, Delft Univ. of Technology, August 1996.

[CH95] Henk Corporaal and Jan Hoogerbrugge. Code generation for Transport Triggered Architectures. In *Code Generation for Embedded Processors*, pages 240–259. Kluwer Academic Publishers, 1995. Chapter 14.

[Cor93] Henk Corporaal. MOVE32INT, Architecture and Programmer's Reference Manual. Technical report, Delft Univ. of Technology, 1993.

[Cor95] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

[Hoo96] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, February 1996. ISBN 90-9009002-9.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture, a quantitative approach*. Morgan Kaufmann publishers, 1990.

[Lam93] Reinoud Lamberts. A MOVE Processor Generator. Technical report, Delft Univ. of Technology, November 1993.

A Instruction set

This appendix gives a list of the functions of the instruction fetch unit, the state unit and the immediate unit. This list only gives a very short description of each function. The details can be found in the chapters 4 and 6, which describe the implementation and the usage of all functions.

instruction fetch unit input socket 1

opcode	*description*	*privileged*
000	page relative jump (short offset)	no
001	page relative function call (short offset)	no
010	page relative jump (medium offset)	no
011	page relative function call (medium offset)	no
100	absolute jump	no
101	absolute function call	no
110	return from exception	yes

instruction fetch unit input socket 2

opcode	*description*	*privileged*
00	function return address	no
01	program counter chain	yes
10	write a virtual page number into the iTLB	yes
11	write a physical page number into the iTLB	yes

instruction fetch unit input socket 3

opcode	*description*	*privileged*
0	trap register	no
1	exception source register	yes

instruction fetch unit output socket 1

opcode	*description*	*privileged*
0	function return address	no
1	program counter chain	yes

instruction fetch unit output socket 2

opcode	*description*	*privileged*
0	trap register	yes
1	exception source register	yes

state unit input socket

opcode	*description*	*privileged*
00	current cache enabled signals	no
01	saved cache enabled signals	yes
10	current TLB enabled signals	yes
11	saved TLB enabled signals	yes

state unit output socket

opcode	*description*	*privileged*
00	current cache enabled signals	no
01	saved cache enabled signals	yes
10	current TLB enabled signals	yes
11	saved TLB enabled signals	yes

immediate unit input socket

opcode	*description*	*privileged*
-	write the immediate extension register	no

immediate unit output socket 1

opcode	*description*	*privileged*
xx…x	short immediate (xx…x)	no

immediate unit output socket 2

opcode	*description*	*privileged*
xx…x	long immediate (extension + xx…x)	no

B Finite state machine of the cache controller

This appendix gives the complete list of transition conditions of the finite state machine (FSM) of the cache controller, which was described in section 4.2.4, and is shown again in figure B.1. The state transitions are controlled by five input signals:

glock is high when the processor is locked.

pc_valid is low means that the instruction in the IF pipeline stage will not be executed, and will therefore not be looked up. If the memory lookup for this instruction already started, it is finished, but the results are not written into the cache. This signal gets low if an exception is started, or if the TLB lookup failed.

mem_ready gets high when the memory interface unit has finished the memory lookup for an instruction.

miss specifies whether a memory lookup is needed for an instruction. This signal is high when the cache reports a miss, or cache reading is disabled.

write_enabled specifies whether writing the cache is allowed.

Table B.1 gives the complete transition table. If one of the input signals is specified as '-', the signal may have any value (don't care). If the next state is specified as don't care, this means that this combination of the state and the input signals is not allowed.

state	glock	pc_valid	mem_ready	miss	write_enabled	next state
check_if	-	0	0	-	-	check_if
	-	1	0	0	-	check_if
	0	1	0	1	0	read_dc
	0	1	0	1	1	read_dc_write
	1	1	0	1	-	read_if
	-	-	1	-	-	-
read_if	-	0	0	-	-	read_canceled
	-	0	1	-	-	check_if
	0	1	0	0	-	read_canceled
	0	1	0	1	0	read_dc
	0	1	0	1	1	read_dc_write
	0	1	1	0	-	check_if
	0	1	1	1	0	check_if
	0	1	1	1	1	write_dc
	1	1	-	-	-	read_if
read_dc_write	1	-	0	-	-	read_dc_write
	1	0	1	-	-	write_dc
	1	1	1	0	-	write_dc
	1	1	1	1	-	write_dc_read
	0	-	-	-	-	-
read_dc	1	-	0	-	-	read_dc
	1	0	1	-	-	check_if
	1	1	1	0	-	check_if
	1	1	1	1	-	read_if
	0	-	-	-	-	-
write_dc_read	1	0	0	-	-	read_canceled
	1	1	0	-	-	read_if
	1	-	1	-	-	-
	0	-	-	-	-	-
write_dc	1	-	0	-	-	read_if
	1	-	1	-	-	-
	0	-	-	-	-	-
read_canceled	1	-	0	-	-	read_canceled
	1	0	1	-	-	check_if
	1	1	1	0	-	check_if
	1	1	1	1	-	read_if
	0	-	-	-	-	-

Table B.1: The complete state transition table for the finite state machine of the cache controller.

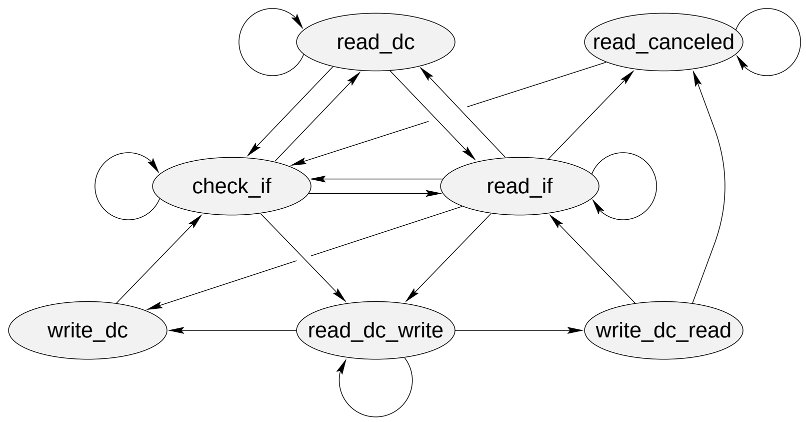


Figure B.1: The finite state machine for the cache controller. The state transition conditions are specified in table B.1.

Parameter description C

This appendix contains a complete list of parameters which are accepted by the instruction fetch unit, the state unit, and the immediate unit. There are two files which contain these parameters:

- `params_user` contains all parameters which must be changed to select another process configuration. Only this file should have to be changed if the processor configuration is modified.
- `params_derv` contains all constants and derived parameters. The parameters in this file should be independent of the selected configuration, but some parameters are specified as constants, where they should be calculated. This is because the complete processor core has not been defined yet, so most of the constants it needs are not defined yet. These parameters include the opcodes for the sockets, and the number of instruction bits.

The contents of these files are listed below. The way these files are used during the conversion from parameterized VHDL to a complete VHDL description without unresolved parameters is explained in appendix D.

The generic cache parameters are defined by the units which use the cache template. These parameters are not in the parameter files directly. The main parameters were included in the description of the generic cache implementation (section 4.3), but the complete list of parameters is shown below in table C.1. If these (m4) parameters are defined when the cache code is included with the m4 command `include PATH/cache.vhdl.m4`, the source code of the desired cache configuration is inserted on that place. The cache gets the name which is specified with the parameter CACHENAME.

parameter	range	description
CACHENAME	-	name of the generated cache configuration
ADDRSIZE	≥ 2	bits in a cache address (tag + offset)
DATASIZE	≥ 1	bits in a data word
CACHESIZE	2^n	number of cache entries
BLOCKSIZE	2^n	entries on one cache line
ASSOCIATIVITY	1, 2	number of places where a block may be placed in the cache
REPLACEMENT	LRU, FIFO	replacement algorithm for set-associative caches
OFFSETSIZE	≥ 1	log (CACHESIZE/ASSOCIATIVITY)
TAGSIZE	≥ 1	ADDRSIZE-OFFSETSIZE

Table C.1: All parameters which are used by the generic cache.

C.1 params_user

```
#================================================================================
#        ------ user parameters ------
# To modify the configuration of the move-processor, this file is the only
# one which needs to be changed.
#
# Connection defines specification (copied from Marnix Arnold):
#
#   Most units have sockets connecting to move busses. For each socket it
#   can be specified to which of the busses it will connect. This is done
#   by defines such as the following:
#
#     define({CINTFU1_TBUS0}, 0)
#     define({CINTFU1_TBUS1}, 0)
#     define({CINTFU1_TBUS2}, 1)
#     define({CINTFU1_TBUS3}, 1)
#
#   In this example, the trigger socket connections for the second
#   integer FU are defined, for a processor with PARMOVES == 4. The
#   socket will connect to busses 2 and 3, but not to 0 and 1.
#
#   For other FUs and sockets, the parameter naming is similar, see
#   further below in this file. The fields in the identifiers are as
#   follows:
#
#       CINTFU1_TBUS2
#       |\   /|||\ /|
#       | \ / ||| V |
#       |  V  ||| | Bus number
#       |  |  ||| |
#       |  |  ||| Always 'BUS'
#       |  |  |||
#       |  |  ||Socket selector: T->trigger, O->operand, R->result
#       |  |  ||selectors can have subscripts (e.g. 'R2')
#       |  |  ||
#       |  |  |Always '_'
#       |  |  |
#       |  |  FU number, starting at 0
#       |  |
#       |  FU type
#       |
#       Always 'C' (for Connect)
#
#
```

```
#==============================================================================

#------------------------------------------------------------------------------
#        ------ general processor parameters ------
#------------------------------------------------------------------------------

# STAGES: 2 for a 2 stage pipeline, 3 for a 3 stage pipeline
define({STAGES}, 2)

# PARMOVES: defines the number of move busses.
define({PARMOVES}, 2)

# BUSWIDTH: defines the data (move-) bus width.
define({BUSWIDTH}, 32)

# VIRTMEM: if defined, TLBs are used to generate virtual memory.
define({VIRTMEM})

# KERNEL: if defined, supports kernel and user mode. Privileged opcodes
#   are used to protect the processor state from modifications by user
#   programs.
# define({KERNEL})

#------------------------------------------------------------------------------
#        ------ Ifetch unit parameters ------
#------------------------------------------------------------------------------

# IFE_TRAPVECTORS: defines the number of possible trapvectors
#   Must be a power of two, and > 0
define({IFE_TRAPVECTORS}, 16)

# IFEFULLSIZE, IFEMEDSIZE, IFESHORTSIZE: define the jump/call offset sizes.
#   If these parameters are undefined, default values will be used.
#   IFEFULLSIZE >= IFEMEDSIZE >= IFESHORTSIZE > 0
define({IFEFULLSIZE},  32)
define({IFEMEDSIZE},   16)
define({IFESHORTSIZE},  4)

# IFEADDRSIZE: defines the instruction address size. IFEADDRSIZE gives the
#   number of instruction address bits to the memory interface.
#   0 < IFEADDRSIZE <= IFEFULLSIZE
define({IFEADDRSIZE}, 32)

# The ifetch cache sizes:
# IFECACHESIZE: The number of icache entries
#   Must be a power of 2.
# IFECACHEASSOCIATIVITY: The associativity of the icache
#   Must be 1 or 2
# IFECACHEREPLACEMENT: Replacement strategy used for a set-associative icache
#   Must be FIFO, or LRU
# IFECACHEBLOCKSIZE: The number of instructions per cache line
#   Must be a power of 2.

define({IFECACHESIZE},        1024)
define({IFECACHEASSOCIATIVITY},  2)
define({IFECACHEREPLACEMENT}, FIFO)
define({IFECACHEBLOCKSIZE},     16)

# The sizes of the ifetch tlb:
# IFETLBSIZE: The number of itlb entries
#   Must be a power of 2.
# IFETLBASSOCIATIVITY: The associativity of the itlb
#   Must be 1 or 2
# IFETLBREPLACEMENT: Replacement strategy used for a set-associative tlb
```

```
#   Must be FIFO, or LRU
# IFETLBBLOCKSIZE: The number of entries on one line
#   Must be a power of 2.
# IFEPAGESIZE: The number of instructions on one virtual memory page
#   Must be a power of two, and at least IFECACHESIZE/IFECACHEASSOCIATIVITY
define({IFETLBSIZE},          16)
define({IFETLBASSOCIATIVITY},  2)
define({IFETLBREPLACEMENT},  LRU)
define({IFETLBBLOCKSIZE},      1)
define({IFEPAGESIZE},       4096)

# Connection defines
define({CIFEFU0_T1BUS0}, 1)
define({CIFEFU0_T1BUS1}, 1)
define({CIFEFU0_T2BUS0}, 1)
define({CIFEFU0_T2BUS1}, 1)
define({CIFEFU0_T3BUS0}, 1)
define({CIFEFU0_T3BUS1}, 1)
define({CIFEFU0_R1BUS0}, 1)
define({CIFEFU0_R1BUS1}, 1)
define({CIFEFU0_R2BUS0}, 1)
define({CIFEFU0_R2BUS1}, 1)


#------------------------------------------------------------------------------
#        ------ state unit parameters ------
#------------------------------------------------------------------------------

# Connection defines
define({CSCLFU0_TBUS0}, 1)
define({CSCLFU0_TBUS1}, 1)
define({CSCLFU0_RBUS0}, 1)
define({CSCLFU0_RBUS1}, 1)

#------------------------------------------------------------------------------
#        ------ immediate unit parameters ------
#------------------------------------------------------------------------------

# IMM_SHORTBITS:  defines the number of bits to be used for a short immediate.
#   The short immediates that can be formed are 0 to 2**IMM_SHORTBITS-1 when
#   IMM_SIGNEXT is not defined, and when it is defined,
#   - 2**(IMM_SHORTBITS-1) to 2**(IMM_SHORTBITS-1)-1
define({IMM_SHORTBITS}, 5)

# IMM_LONGBITS:  defines the number of bits to use as the opcode part of a
#   long immediate. To create a long opcode, these bits are concatenated with
#   the extension bits.
define({IMM_LONGBITS}, 4)

# IMM_REG:  if defined, extension from previous instructions can be used to
#   generate longer immediates than possible when only the current instruction
#   would be used. To ensure correct exception support, at least one bus
#   must be able to generate short immediates (to address the load-store
#   unit).
define({IMM_REG})

# IMM_SHIFT:  if defined, a long immdiate can be generated by shifting. This
#   means that the extension register can be filled by more then one
#   instruction. When another instruction tries to write the extension
#   register, its previous contents are shifted to the more significant side.
#   IMM_REG must be defined.
#define({IMM_SHIFT})

# IMM_SIGNEXT: if defined, all immediates use sign extension
```

```
define({IMM_SIGNEXT})

# IMM_BUSSES:   specifies the maximum number of busses to be used to generate
#   extensions for long immediates. If undefined, as much busses are used
#   as necessary to generate a full size immediate.
undefine({IMM_BUSSES})

# r1 socket: short immediates
# r2 socket: long immediates
# Connection defines:

define({CIMMFU0_TBUS0},  1)
define({CIMMFU0_TBUS1},  1)
define({CIMMFU0_R1BUS0}, 1)
define({CIMMFU0_R1BUS1}, 0)
define({CIMMFU0_R2BUS0}, 0)
define({CIMMFU0_R2BUS1}, 1)

#------------------------------------------------------------------------------
#        ------ Load store unit parameters (copied from Theo Baan) ------
#------------------------------------------------------------------------------

# DATACACHE: if defined, a cache is generated for the load store unit:
define({DATACACHE})

# speculative memory access when ya don't know whether it is a data cache hit
define({NO_MEMORY_ACCESS_ON_CACHE_MISS})

# BITS_ON_A_PAGE gives maxsize for cache
#   - this is without LDST_IGNORE_LOWER_X_BITS
define({BITS_ON_A_PAGE}, 4)

# DTLB_BITS_LINES: 2log(nr of lines) for the 2-way associative tlb cache
define({DTLB_BITS_LINES}, 2)

# DCACHE_BITS_LINES <= Words_on_a_page since direct mapped
define({DCACHE_BITS_LINES}, 4)

# Connection defines:
define({CLDSFU0_TBUS0},  1)
define({CLDSFU0_TBUS1},  1)
define({CLDSFU0_OBUS0},  1)
define({CLDSFU0_OBUS1},  1)
define({CLDSFU0_R1BUS0}, 1)
define({CLDSFU0_R1BUS1}, 1)
define({CLDSFU0_R2BUS0}, 1)
define({CLDSFU0_R2BUS1}, 1)

#------------------------------------------------------------------------------
#        ------ External memory interface (copied from Theo Baan) ------
#------------------------------------------------------------------------------

# watch it if you discard some bits from memory (e.g. 32 bits bus,
# 48 bits instr, and 1 instruction uses 2 words)

# number of pins to external memory
define({EXTMEM_DATASIZE},  32)
define({EXTMEM_ADRESSIZE}, 16)

# bits needed to represent latency to memory
define({EXTMEM_BITS_FOR_DELAY}, 4)
```

C.2 params_derv

```
#==============================================================================
#          ------ derived parameters ------
# All parameters in this file are either constants that may not be changed,
# or parameters which are derived from user constants.
#==============================================================================

# There can be only one ifetch unit and state unit:
define({IFEFUS}, 1)
define({SCLFUS}, 1)
# The current implementation of the immediate unit assumes there is only
# one immediate unit. More than one is useless anyway.
define({IMMFUS}, 1)
# The load-store unit also assumes it is the only one.
define(LDSFUS, 1)

# Check the immediate unit paramerters, and define defaults if some parameters
#   are not defined yet.
# IMM_BUSSES is the number of move busses that can be used for immediate
#   extensions. Cannot be greater than the total number of move busses
ifdef({IMM_BUSSES},, {
  define({IMM_BUSSES}, PARMOVES)
})
ifelse(eval(IMM_BUSSES>PARMOVES), {
  define({IMM_BUSSES}, PARMOVES)
})

# TEMP counts the number of busses to which a long immediate output socket
#   is connected. If zero, then IMM_BUSSES = 0, IMM_REG undefined, IMM_shift
#   undefined because long immediates are not used.
define({TEMP}, 0)
FOR(BUSNR, 0, PARMOVES-1, {
  define({TEMP}, eval(TEMP+translit(CIMMFU0_R2BUS\BUSNR,\))) })
ifelse(TEMP, 0, {
  define({IMM_BUSSES}, 0) })
undefine({TEMP})

ifelse(IMM_BUSSES, 0, {
  undefine({IMM_REG})
    FOR(BUSNR, 0, PARMOVES-1, {
      define({translit(CIMMFU0_R2BUS\BUSNR,\)}, 0) }) })

ifdef({IMM_REG},, { undefine({IMM_SHIFT}) })

#------------------------------------------------------------------------------
# socket opcode size:
define({IFEFUT1S},  7)
define({IFEFUT2S},  ifdef({VIRTMEM}, 4, 2))
define({IFEFUT3S},  2)
define({IFEFUR1S},  2)
define({IFEFUR2S},  2)

define({SCLFUTS},  ifdef({VIRTMEM}, 4, 2))
define({SCLFURS},  ifdef({VIRTMEM}, 4, 2))

define({IMMFUTS},  1)
define({IMMFUR1S}, eval(2**IMM_SHORTBITS))
define({IMMFUR2S}, eval(2**IMM_LONGBITS))

define({LDSFUTS},  ifdef({VIRTMEM}, 4, 3))
define({LDSFUR1S}, 1)
define({LDSFUR2S}, 1)
```

```
#------------------------------------------------------------------------------
# SRCSIZE?, DSTSIZE?, GRDSIZE: define the number of bits used for the
#   source and desitination IDs and the guard specifier.
define({SRCSIZE0}, 6)
define({DSTSIZE0}, 6)
define({SRCSIZE1}, 6)
define({DSTSIZE1}, 6)
define({GRDSIZE},  2)

#------------------------------------------------------------------------------
# IMM_EXTSIZE: size of full extensions. Will be calculated later:
define({IMM_EXTSIZE},    0)

# IMM_EXTENSIONS:  the number of busses needed to generate a full size
#   extension. Will be calculated later:
define({IMM_EXTENSIONS}, 0)

# INSTRSIZE: defines the number of bits used for every instruction. Every
#   instruction specifies PARMOVES moves. The space reserved in memory for
#   every instruction may be > INSTRSIZE. Will be calculated later:
define({INSTRSIZE},      0)

#------------------------------------------------------------------------------
# definitions for IMM_EXTSIZE, IMM_EXTENSIONS and INSTRSIZE:
FOR(BUSNR, 0, IMM_BUSSES-1, {
  ifelse(eval(IMM_EXTSIZE < BUSWIDTH-IMM_LONGBITS), 1, {
    define({IMM_EXTENSIONS}, eval(IMM_EXTENSIONS+1))
    define({IMM_EXTSIZE},
      eval(translit(IMM_EXTSIZE+SRCSIZE\BUSNR+DSTSIZE\BUSNR+GRDSIZE,\)))
  })
})

ifelse(eval(IMM_EXTSIZE < BUSWIDTH-IMM_LONGBITS), 1, {
  ifdef({IMM_SHIFT}, {
    define({IMM_SHIFTSIZE}, IMM_EXTSIZE)
    define({IMM_EXTSIZE}, eval(BUSWIDTH-IMM_LONGBITS))
  })
},{
  define({IMM_EXTSIZE}, eval(BUSWIDTH-IMM_LONGBITS))
  undefine({IMM_SHIFT})
})

FOR(BUSNR, 0, PARMOVES-1, {
  define({INSTRSIZE},
    eval(translit(INSTRSIZE+SRCSIZE\BUSNR+DSTSIZE\BUSNR+GRDSIZE,\)))
})
define({INSTRSIZE}, eval(INSTRSIZE+IMM_EXTENSIONS))

#------------------------------------------------------------------------------
# IFEFULLSIZE, IFEMEDSIZE, IFESHORTSIZE: define the jump/call offset sizes.
#   Assign default values to the undefined parameters:
ifdef({IFEFULLSIZE},,{
  define({IFEFULLSIZE},  ifdef({VIRTMEM}, BUSWIDTH, IFEADDRSIZE)) })
ifdef({IFEMEDSIZE},,{
  define({IFEMEDSIZE},   eval(SRCSIZE0+DSTSIZE0+GRDSIZE+IMM_LONGBITS)) })
ifdef({IFESHORTSIZE},,{
  define({IFESHORTSIZE}, eval(IMM_SHORTBITS)) })
ifelse(eval(IFEMEDSIZE>IFEFULLSIZE),  1, { define({IFEMEDSIZE},  IFEFULLSIZE) })
ifelse(eval(IFESHORTSIZE>IFEMEDSIZE), 1, { define({IFESHORTSIZE}, IFEMEDSIZE) })


#------------------------------------------------------------------------------
# exception handler addresses (can be regarded user parameters too).
```

```
# IFE_RESET_VECTOR  : The start address of the reset handler
# IFE_INT_VECTOR    : The start address of the interrupt handler
# IFE_TRAP_VECTOR   : The start address of the trap handler
# IFE_ITLB_VECTOR   : The start address of the itlb miss handler
# IFE_DTLB_VECTOR   : The start address of the dtlb miss handler
# IFE_OPCODE_VECTOR : The start address of the privileged opcode handler
define({IFE_RESET_VECTOR}, "FOR(CNT,8,IFEFULLSIZE-1,{0})00100000")
define({IFE_INT_VECTOR},   "FOR(CNT,8,IFEFULLSIZE-1,{0})01000000")
define({IFE_TRAP_VECTOR},  "FOR(CNT,8,IFEFULLSIZE-1,{0})01100000")
define({IFE_ITLB_VECTOR},  "FOR(CNT,8,IFEFULLSIZE-1,{0})10000000")
define({IFE_DTLB_VECTOR},  "FOR(CNT,8,IFEFULLSIZE-1,{0})10000000")
define({IFE_OPCODE_VECTOR},"FOR(CNT,8,IFEFULLSIZE-1,{0})10100000")

#------------------------------------------------------------------------------

# SCL_ICACHE_BITS: number of icache enabled bits
# SCL_ITLB_BITS  : number of itlb   enabled bits
# SCL_DCACHE_BITS: number of dcache enabled bits
# SCL_DTLB_BITS  : number of dtlb   enabled bits

define({SCL_ICACHE_BITS}, 2)
ifdef({VIRTMEM}, {
  define({SCL_ITLB_BITS}, 1)
},{
  define({SCL_ITLB_BITS}, 0)
})

ifdef({DATACACHE}, {
  define({SCL_DCACHE_BITS}, LDSFUS)
},{
  define({SCL_DCACHE_BITS}, 0)
})
ifdef({VIRTMEM}, {
  define({DATATLB})
  define({SCL_DTLB_BITS}, LDSFUS)
},{
  define({SCL_DTLB_BITS}, 0)
})

#------------------------------------------------------------------------------
# Socket addresses:
#------------------------------------------------------------------------------

# Base addresses for the sockets of the several units.
# These should be calculated automatically in the future,
# but are defined manually for now.

# Socket ID’s have the following form (copied from Marnix Arnold):
#
#   AINTFU1_TBUS2
#   |\   /| |\ /|
#   | \ / | | V \_ Bus number
#   |  V  | |  \__ Always ’BUS’
#   |  |  |  \____ Socket selector (’T’rigger, ’O’perand, ’R’esult)
#   |  |  \_______ FU number, starting at ’0’
#   |  \__________ FU type
#   \_____________ Always ’A’ddress
#
#------------------------------------------------------------------------------

define({AIFEFU0_T1BUS0},  0)
define({AIFEFU0_T1BUS1},  0)
define({AIFEFU0_T2BUS0},  8)
define({AIFEFU0_T2BUS1},  8)
```

```
define({AIFEFU0_T3BUS0}, 12)
define({AIFEFU0_T3BUS1}, 12)
define({AIFEFU0_R1BUS0},  0)
define({AIFEFU0_R1BUS1},  0)
define({AIFEFU0_R2BUS0},  2)
define({AIFEFU0_R2BUS1},  2)

define({ASCLFU0_TBUS0},  16)
define({ASCLFU0_TBUS1},  16)
define({ASCLFU0_RBUS0},   4)
define({ASCLFU0_RBUS1},   4)

define({ALDSFU0_TBUS0},  20)
define({ALDSFU0_TBUS1},  20)
define({ALDSFU0_OBUS0},  24)
define({ALDSFU0_OBUS1},  24)
define({ALDSFU0_R1BUS0},  8)
define({ALDSFU0_R1BUS1},  8)
define({ALDSFU0_R2BUS0},  9)
define({ALDSFU0_R2BUS1},  9)

define({AIMMFU0_TBUS0},  25)
define({AIMMFU0_TBUS1},  25)
define({AIMMFU0_R1BUS0}, 32)
define({AIMMFU0_R2BUS1}, 32)
```

Structure of the VHDL modules D

The VHDL subunits and their hierarchical structure are shown in figure D.1. The icache and itlb_cache are implementations of the generic cache, and contain one or more ram cells, depending on the configuration. The names of the subunits of the instruction fetch unit all start with ”ife”, which is an abbreviation of instruction fetch unit. The last three characters indicate the function of the subunits, which are ”fui” (functional unit internals) for each top level design[1]. This naming convention was also used in [Arn95].

The functions of the subunits are:

ifefui The top level of the instruction fetch unit. Its only function is to connect the subdesigns.

ifeexc Contains the exception controller, the exception source register, and the trap register.

ifeopc Checks the instruction in the decode pipeline stage, and reports to the exception controller whether it contains privileged opcodes.

ifesel Contains the program counter (PC), the PC selector, and the PC chain.

ifetlb Contains the virtual memory datapath and the virtual memory controller. The main part of the datapath, the TLB, is in itlb_cache, which is an implementation of the generic cache template.

ifecch Contains the cache controller, the instruction address register (for the memory interface unit) and the instruction register. The cache itself is in icache, which is an implementation of the generic cache template.

[1]The suffix ”fu” (functional unit) is reserved for the combination of the functional unit internals and the sockets.

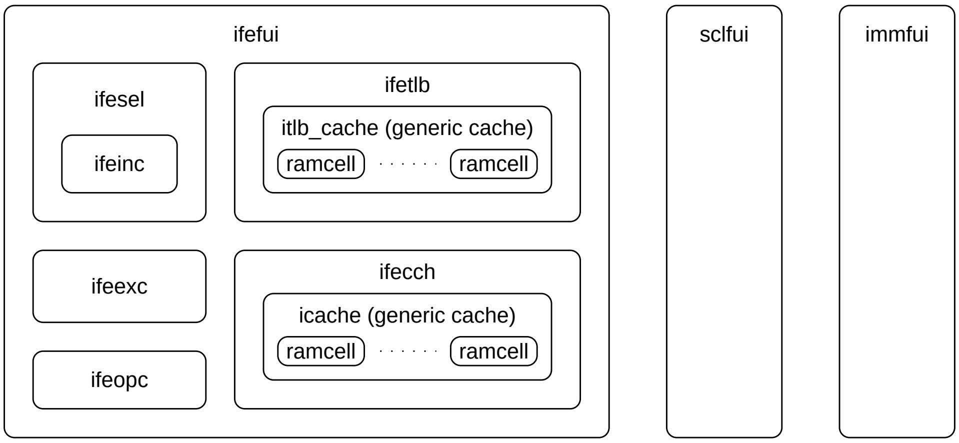


Figure D.1: The hierarchical structure of the VHDL subdesigns of the instruction fetch unit, the state unit and the immediate unit.

sclfui The state unit, which controls the processor modes and the enabled signals for the caches and the TLBs.

immfui The immediate unit generates both long and short immediates.

The files with these units carry the same name as the unit they contain, but with the suffix `.vhdl.m4` to specify that the file contains parameterized VHDL. The generic cache definition is in the file `cache.vhdl.m4`, and the definitions of the ramcells which are used by the generic cache should be in `ramcell.vhdl`

Including the generic caches is the responsibility of the units which use it. These units must specify the necessary parameters, and then include the cache description with the m4 command `include PATH/cache.vhdl.m4`. The parameters which are used by the generic cache are described in appendix C.

The opcodes of the functional units are specified in the file `pkg_opcodes.vhdl.m4` The opcodes in this file do not include the socket IDs, but only the opcode bits which are passed to the functional unit internals themselves. All opcodes use the same format. An example is IFE_JUMP_JUMP_T1. In this opcode, the first three characters contain the type of the functional unit (in this case the instruction fetch unit), and the characters after the last underscore specify the type of the socket (in this case the first trigger socket). The socket number is not needed if the unit has only one socket of that kind. The middle field specifies the function of the opcode.

There are two steps needed to generate a complete VHDL description out of the parameterized descriptions of the subunits:

- The parameter references must be resolved by running m4 on the file `genpass.m4`. This file includes the parameter files `params_user` and `params_derv`, and the necessary VHDL source files. The file `params_user` contains all parameters which are necessary to specify a processor configuration. If another configuration is needed, only this file has to be changed. The file `params_derv` contains some constant parameters, and parameters which must be calculated from the user parameters and the constants. The parameter files are given in appendix C.

- Every VHDL description which uses another subdesign must include the entity specification of all its subunits in its architecture body. This can make the code very unclear. There is however an alternative. Those component specifications may also be collected in a special package, which can be used by every VHDL design. This package is automatically generated by the tool *vhdl_components*, which is described in appendix F.

The contents of the file `genpass.m4` are included below. Running m4 on this file generates an large VHDL description of the complete instruction fetch unit, state unit, and immediate unit. In this description, all m4 parameters are resolved. After this step, only adding the components package is needed to make the VHDL description is complete.

```
divert(-1)
#-----------------------------------------------------------------------------
# genpass.m4
#
# m4 preprocessor file for the ifetch, state and immediate unit
#-----------------------------------------------------------------------------

changequote({, })

define({FOR}, {ifelse(eval($2<=$3), 1, {define({$1}, $2)$4FOR({$1}, incr($2),$3,{$4})})undefine({$1})})

define({LOG}, {ifelse(eval($1<=1), 1, 0, {eval(1+LOG($1-eval($1)/2))})})
# Aangepast om LOG(7) gelijk aan 3 te maken, evenals LOG(8)

# shift and index are predefined m4 macros which may be used as normal
# identifiers in vhdl
undefine({shift})
undefine({index})

# Path is the directory with vhdl templates:
define({PATH}, .)

# define parameters:
include(params_user)
include(params_derv)

# enable m4 output and include all FU's
divert

include(PATH/pkg_opcodes.vhdl.m4)

include(PATH/ramcell.vhdl)

include(PATH/ifeinc.vhdl.m4)

include(PATH/ifesel.vhdl.m4)

include(PATH/ifecch.vhdl.m4)

include(PATH/ifetlb.vhdl.m4)

include(PATH/ifeopc.vhdl.m4)

include(PATH/ifeexc.vhdl.m4)

include(PATH/ifefui.vhdl.m4)

include(PATH/sclfui.vhdl.m4)

include(PATH/immfui.vhdl.m4)
```

E VHDL sources

This appendix contains the parameterized VHDL templates from which the complete VHDL descriptions of the instruction fetch unit, the state unit, and the immediate unit are generated. They might appear to be difficult to read at fist sight because of the m4 descriptions that are used. It may be helpful to compare these descriptions with the result after the preprocessing, to see how the m4 commands evaluate.

Each file starts with a short header which describes the function of the unit, and a list of the m4 parameters on which it depends. The functions of the files is also described in appendix D.

E.1 pkg_opcodes.vhdl.m4

```
-------------------------------------------------------------------------------
-- Opcodes package
--
-- author: Steven Roos
-------------------------------------------------------------------------------

-- parameter dependencies:
--   {VIRTMEM}       : ifdef({VIRTMEM}, defined, not defined)


library IEEE;
use IEEE.std_logic_1164.all;

-- Opcode name format:
--  IFE_JUMP_FULL_T1
--  \ / \___ ___/ ||
--   V      V     |\_ Socket number specifier (optional)
--   |      |     \__ Socket selector ('T'rigger, 'O'perand, 'R'esult)
--   |      \________ Opcode name (optional)
--   \_______________ FU type

library IEEE;
use IEEE.std_logic_1164.all;

package opcodes is
```

```
  -- ifetch FU T1 (jump/call):
  constant IFE_JUMP_SHORT_T1  : std_logic_vector := "000";
  constant IFE_CALL_SHORT_T1  : std_logic_vector := "001";
  constant IFE_JUMP_MED_T1    : std_logic_vector := "010";
  constant IFE_CALL_MED_T1    : std_logic_vector := "011";
  constant IFE_JUMP_FULL_T1   : std_logic_vector := "100";
  constant IFE_CALL_FULL_T1   : std_logic_vector := "101";
  constant IFE_JUMP_RFE_T1    : std_logic_vector := "110"; -- privileged opcode

    ifdef({VIRTMEM}, {
  -- ifetch FU T2 (state restoring, TLB):
  constant IFE_RETADDR_T2     : std_logic_vector := "00";
  constant IFE_PCC_T2         : std_logic_vector := "01"; -- privileged opcode
  constant IFE_TLB_VIRT_T2    : std_logic_vector := "10"; -- privileged opcode
  constant IFE_TLB_REAL_T2    : std_logic_vector := "11"; -- privileged opcode
    },{
  -- ifetch FU T2 (state restoring, TLB):
  constant IFE_RETADDR_T2     : std_logic_vector := "0";
  constant IFE_PCC_T2         : std_logic_vector := "1"; -- privileged opcode
    })
  -- ifetch FU T3 (trap, exception sources):
  constant IFE_TRAP_T3        : std_logic_vector := "0";
  constant IFE_EXC_SOURCE_T3  : std_logic_vector := "1";

  -- ifetch FU R1 (state saving)
  constant IFE_RETADDR_R1     : std_logic_vector := "0";
  constant IFE_PCC_R1         : std_logic_vector := "1"; -- privileged opcode

  -- ifetch FU R2 (exception sources):
  constant IFE_TRAP_R2        : std_logic_vector := "0"; -- privileged opcode
  constant IFE_EXC_SOURCE_R2  : std_logic_vector := "1"; -- privileged opcode

    ifdef({VIRTMEM}, {
  -- state FU T
  constant SCL_CACHE_STATE_T  : std_logic_vector := "00";
  constant SCL_CACHE_SAVED_T  : std_logic_vector := "01"; -- privileged opcode
  constant SCL_TLB_STATE_T    : std_logic_vector := "10"; -- privileged opcode
  constant SCL_TLB_SAVED_T    : std_logic_vector := "11"; -- privileged opcode

  -- state FU R
  constant SCL_CACHE_STATE_R  : std_logic_vector := "00";
  constant SCL_CACHE_SAVED_R  : std_logic_vector := "01"; -- privileged opcode
  constant SCL_TLB_STATE_R    : std_logic_vector := "10"; -- privileged opcode
  constant SCL_TLB_SAVED_R    : std_logic_vector := "11"; -- privileged opcode
    },{
  -- state FU T
  constant SCL_CACHE_STATE_T  : std_logic_vector := "0";
  constant SCL_CACHE_SAVED_T  : std_logic_vector := "1";

  -- state FU R
  constant SCL_CACHE_STATE_R  : std_logic_vector := "0";
  constant SCL_CACHE_SAVED_R  : std_logic_vector := "1";
    })
  -- load-store FU T
  constant LDS_LOAD_T         : std_logic_vector := "00";
  constant LDS_STORE_T        : std_logic_vector := "01";
  constant LDS_LOAD2_T        : std_logic_vector := "10";  ifdef({VIRTMEM}, {
  constant LDS_TLB_T          : std_logic_vector := "11"; -- privileged opcode})

end opcodes;
```

E.2 ifeinc.vhdl.m4

```
------------------------------------------------------------------------------
-- Ifetch FU PC incrementer (final version)
--
------------------------------------------------------------------------------

-- parameter dependencies:
--   {IFEFULLSIZE}   : IFEFULLSIZE


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity ifeinc is
  port(
    i : in  std_logic_vector(IFEFULLSIZE-1 downto 0);
    y : out std_logic_vector(IFEFULLSIZE-1 downto 0));
end ifeinc;
```

```
architecture func of ifeinc is
begin
      y <= conv_std_logic_vector(unsigned(i) + 1, y'length);
end func;
```

E.3 ifesel.vhdl.m4

```
-------------------------------------------------------------------------------
-- Ifetch FU PC selector (final version)
--
-- author: Steven Roos
-------------------------------------------------------------------------------

-- parameter dependencies:
--   {STAGES}        : STAGES
--   {BUSWIDTH}      : BUSWIDTH
--   {IFEFULLSIZE}   : IFEFULLSIZE
--   {IFEMEDSIZE}    : IFEMEDSIZE
--   {IFESHORTSIZE}  : IFESHORTSIZE

--   {IFEFUT1S}      : IFEFUT1S
--   {IFEFUT2S}      : IFEFUT2S
--   {IFEFUR1S}      : IFEFUR1S


-- This unit contains the PC (program counter) and the PC-chain. The
-- PC-chain is used at exceptions to store the first addresses that
-- didn't execute when the processor was interrupted. These addresses
-- are the first that the processor must execute after a return from
-- exception.

library IEEE;
use IEEE.std_logic_1164.all;
use work.opcodes.all;
use work.components.all;

entity ifesel is
  port (
    ck          : in  std_logic;
    glock       : in  std_logic;
    exc_request : in  std_logic;
    exc_vector  : in  std_logic_vector (IFEFULLSIZE-1 downto 0);
    exc_enabled : in  std_logic;
    exc_call    : out std_logic;
    exc_return  : out std_logic;

    t1_data     : in  std_logic_vector (BUSWIDTH-1 downto 0);
    t1_active   : in  std_logic;
    t1_opcode   : in  std_logic_vector (LOG(IFEFUT1S)-1 downto 0);
    t2_data     : in  std_logic_vector (BUSWIDTH-1 downto 0);
    t2_active   : in  std_logic;
    t2_opcode   : in  std_logic_vector (LOG(IFEFUT2S)-1 downto 0);
    r_data      : out std_logic_vector (BUSWIDTH-1 downto 0);
    r_opcode    : in  std_logic_vector (LOG(IFEFUR1S)-1 downto 0);

    new_pc      : out std_logic_vector (IFEFULLSIZE-1 downto 0);
    pc          : out std_logic_vector (IFEFULLSIZE-1 downto 0));
end ifesel;

architecture func of ifesel is
  -- pc_reg (= pc) is the register that contains the PC;
  -- new_pc_reg (= new_pc) is its next value.
  -- next_pc is the next value of PC when exceptions are ignored
  signal pc_reg     : std_logic_vector (IFEFULLSIZE-1 downto 0);
  signal new_pc_reg : std_logic_vector (IFEFULLSIZE-1 downto 0);
  signal next_pc    : std_logic_vector (IFEFULLSIZE-1 downto 0);
  -- pcc_?? are the next values for the PC after a return from exception
  signal pcc_if : std_logic_vector (IFEFULLSIZE-1 downto 0); ifelse(STAGES, 3, {
  signal pcc_dc : std_logic_vector (IFEFULLSIZE-1 downto 0); })
  signal pcc_mv : std_logic_vector (IFEFULLSIZE-1 downto 0);
  -- pcc_lock specifies whether the PC-chain should run along with the PC
  signal pcc_lock : std_logic;

  signal pc_plus_1 : std_logic_vector (IFEFULLSIZE-1 downto 0);
  signal function_return_address : std_logic_vector (IFEFULLSIZE-1 downto 0);

  for pcincrement: ifeinc use entity work.ifeinc (func);
begin
  pcincrement: ifeinc port map (
    i => pc_reg,
    y => pc_plus_1);
```

```
  process (glock, t1_active, t1_opcode, t1_data, pc_reg, pc_plus_1)
  begin
    -- this next_pc signal is divided in three parts:
      -- IFEFULLSIZE-1 downto IFEMEDSIZE
      -- IFEMEDSIZE-1 downto IFESHORTSIZE
      -- IFESHORTSIZE-1 downto 0
    -- Every of the three parts can come from three sources:
    -- pc_reg, t1_data and pc_plus_1

    next_pc <= pc_reg;
    if (glock = '0') then
      if    (t1_active = '1' and
             (t1_opcode = IFE_JUMP_SHORT_T1 or
              t1_opcode = IFE_CALL_SHORT_T1 )) then
        next_pc (IFESHORTSIZE-1 downto 0) <= t1_data (IFESHORTSIZE-1 downto 0);
      elsif (t1_active = '1' and
             (t1_opcode = IFE_JUMP_MED_T1 or
              t1_opcode = IFE_CALL_MED_T1 )) then
        next_pc (IFEMEDSIZE-1 downto 0) <= t1_data (IFEMEDSIZE-1 downto 0);
      elsif (t1_active = '1' and
             (t1_opcode = IFE_JUMP_FULL_T1 or
              t1_opcode = IFE_CALL_FULL_T1 or
              t1_opcode = IFE_JUMP_RFE_T1  )) then
        next_pc (IFEFULLSIZE-1 downto 0) <= t1_data (IFEFULLSIZE-1 downto 0);
      else
        next_pc <= pc_plus_1;
      end if;
    end if;
  end process;

  new_pc_reg <=
    exc_vector when (exc_request = '1') else
    next_pc;

  process
  begin
    wait until ck'event and (ck = '1');

    pc_reg <= new_pc_reg;

    if (t2_active = '1' and t2_opcode = IFE_RETADDR_T2) then
      function_return_address <= t2_data (IFEFULLSIZE-1 downto 0);
    elsif (t1_active = '1' and (t1_opcode = IFE_CALL_SHORT_T1 or
                                t1_opcode = IFE_CALL_MED_T1   or
                                t1_opcode = IFE_CALL_FULL_T1  )) then
      function_return_address <= pc_plus_1;
    end if;

    if (t2_active = '1' and t2_opcode = IFE_PCC_T2) then
      pcc_mv <= ifelse(STAGES, 3, {pcc_dc;
      pcc_dc <= })pcc_if;
      pcc_if <= t2_data (IFEFULLSIZE-1 downto 0);
    elsif ((glock = '0' and pcc_lock = '0') or
           (t1_active = '1' and t1_opcode = IFE_JUMP_RFE_T1)) then
      pcc_mv <= ifelse(STAGES, 3, {pcc_dc;
      pcc_dc <= })pcc_if;
      pcc_if <= next_pc;
    end if;

    exc_call   <= '0';
    exc_return <= '0';
    if (exc_request = '1') then
      exc_call <= '1';
      pcc_lock <= '1';
    elsif (exc_enabled = '0' and
           t1_active = '1' and t1_opcode = IFE_JUMP_RFE_T1) then
      exc_return <= '1';
      pcc_lock <= '0';
    end if;
  end process;

  new_pc <= new_pc_reg;
  pc <= pc_reg;
    ifelse(eval(BUSWIDTH>IFEFULLSIZE), 1, {
  r_data (BUSWIDTH-1 downto IFEFULLSIZE) <=
    "FOR(CNT, IFEFULLSIZE, BUSWIDTH-1, 0)"; })
  r_data (IFEFULLSIZE-1 downto 0) <=
    function_return_address
      when (r_opcode = IFE_RETADDR_R1) else
    pcc_mv
      when (r_opcode = IFE_PCC_R1) else
    "FOR(CNT, 0, IFEFULLSIZE-1, {-})";

end func;
```

E.4 ifecch.vhdl.m4

```
--------------------------------------------------------------------------------
-- Ifetch FU cache + memory controller (final version)
--
-- author: Steven Roos
--------------------------------------------------------------------------------

-- parameter dependencies:
--   {IFEADDRSIZE}          : IFEADDRSIZE
--   {IFEFULLSIZE}          : IFEFULLSIZE
--   {INSTRSIZE}            : INSTRSIZE
--   {IFECACHESIZE}         : IFECACHESIZE
--   {IFECACHEBLOCKSIZE}    : IFECACHEBLOCKSIZE
--   {IFECACHEASSOCIATIVITY}: IFECACHEASSOCIATIVITY
--   {IFECACHEREPLACEMENT}  : IFECACHEREPLACEMENT


-- This version of the cache controller for the ifetch unit uses the generic
-- cache implementation in cache.vhdl. That cache implementation represents
-- a one cycle cache. Two cycle caches (in which the tag check is delayed
-- one cycle) are not spported (yet). These require another state machine.
--   When this state machine starts up a memory access, it is finished too.
-- Memory reads are only interrupted at a reset. When an exception occurs,
-- or when the current PC generates a tlb-miss, pc_valid must be lowered.
-- When that entry is being read from memory, the memory read is completed,
-- but the value will not be written to the cache.

--# m4-parameters for the cache:
define({CACHENAME},     icache)
define({ADDRSIZE},      IFEADDRSIZE)
define({DATASIZE},      INSTRSIZE)
define({CACHESIZE},     IFECACHESIZE)
define({BLOCKSIZE},     IFECACHEBLOCKSIZE)
define({ASSOCIATIVITY}, IFECACHEASSOCIATIVITY)
define({REPLACEMENT},   IFECACHEREPLACEMENT)
define({OFFSETSIZE},    LOG(CACHESIZE/ASSOCIATIVITY))
define({TAGSIZE},       eval(ADDRSIZE-OFFSETSIZE))
-- First, the cache architecture code is included:
include(PATH/cache.vhdl.m4)

--------------------------------------------------------------------------------
-- start of ifecch
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use work.components.all;

entity ifecch is
  port (
    ck          : in  std_logic;
    -- addresses used to access the cache:
    new_pc      : in  std_logic_vector (IFEFULLSIZE-1 downto 0);
    real_pc     : in  std_logic_vector (ADDRSIZE-1 downto 0);
    -- signals for the memory interface:
    mem_data    : in  std_logic_vector (DATASIZE-1 downto 0);
    mem_ready   : in  std_logic;
    mem_address : out std_logic_vector (ADDRSIZE-1 downto 0);
    mem_strobe  : out std_logic;
    -- next instruction:
    instruction : out std_logic_vector (DATASIZE-1 downto 0);
    -- control signals:
    reset       : in  std_logic;
    glock       : in  std_logic;
    pc_valid    : in  std_logic;
    enabled     : in  std_logic_vector (2-1 downto 0);
    lock_req    : out std_logic);
end ifecch;

architecture func of ifecch is
  type states is
    (check_if, read_if, write_dc_read, write_dc,
     read_dc_write, read_dc, read_canceled);
    -- Description of the states:
    --   check_if:
    --     The DC instruction is ready, and the IF instruction is being
    --     looked up in the cache (and TLB).
    --   read_if:
    --     The DC instruction is still ready, but the IF instruction was
    --     not found in the cache, and is therefore read from memory.
    --   write_dc_read:
    --     The DC instruction is being written into the cache and a
    --     speculative memory read of the IF instruction is started.
    --   write_dc:
    --     The DC instruction is being written into the cache. No
```

```
    --     speculative memory read is started.
    --   read_dc_write:
    --     The DC instruction is being read from memory, and has to be
    --     written into the cache.
    --   read_dc:
    --     The DC instruction is being read from memory, but will not be
    --     written into the cache.
    --   read_canceled:
    --     A useless, canceled instruction is being read from memory. The
    --     DC instruction is still ready.

  signal state : states;

  signal write_enabled : std_logic; -- cache_enabled (0)
  signal read_enabled  : std_logic; -- cache_enabled (1)

  signal address     : std_logic_vector (ADDRSIZE-1 downto 0); -- mem_address
  signal cache_data  : std_logic_vector (DATASIZE-1 downto 0); -- cache output

  signal cache_save_read  : std_logic;   -- report that an entry is used
  signal cache_save_write : std_logic;   -- prepare the cache for a write
  signal cache_write      : std_logic;   -- write into the cache
  signal cache_miss       : std_logic;   -- the cache reports a miss
  signal total_miss       : std_logic;   -- cache_miss or not read_enabled

  for cch_unit: icache use entity work.icache (func);

begin
  cch_unit: icache
    port map (
      ck            => ck,
      read_offset   => new_pc (OFFSETSIZE-1 downto 0),
      read_tag      => real_pc (ADDRSIZE-1 downto OFFSETSIZE),
      write_offset  => address (OFFSETSIZE-1 downto 0),
      write_tag     => address (ADDRSIZE-1 downto OFFSETSIZE),
      data_in       => mem_data,
      data_out      => cache_data,
      read_enabled  => cache_save_read,
      write_enabled => cache_save_write,
      write         => cache_write,
      miss          => cache_miss);

  write_enabled <= enabled (0);
  read_enabled  <= enabled (1);

  total_miss <=
    cache_miss or not read_enabled;

  mem_address <= address;

  mem_strobe <=
    '1' when (state = read_if       or
              state = write_dc_read or
              state = read_dc_write or
              state = read_dc       or
              state = read_canceled) else
    '0';

  lock_req <=
    '1' when (state = write_dc_read or
              state = write_dc      or
              state = read_dc_write or
              state = read_dc       or
              state = read_canceled) else
    '0';

  cache_save_read <=
    '1' when (glock = '0' and pc_valid = '1' and total_miss = '0') else
    '0';

  cache_save_write <=
    '1' when (glock = '0' and pc_valid = '1' and total_miss = '1') else
    '0';

  cache_write <=
    mem_ready when (state = read_dc_write) else
    mem_ready when (state = read_if     and
                    glock         = '0' and
                    pc_valid      = '1' and
                    total_miss    = '1' and
                    write_enabled = '1') else
    '0';

  process
  begin
    wait until ck'event and (ck = '1');
```

```
case state is

  when check_if =>
    if (pc_valid = '1' and total_miss = '1') then
      if (glock = '1') then
        state <= read_if;
      elsif (write_enabled = '1') then
        state <= read_dc_write;
      else
        state <= read_dc;
      end if;
    end if;

  when read_if =>
    if (glock = '1' and pc_valid = '1') then
      state <= read_if;
    elsif (mem_ready = '0') then
      if (pc_valid = '0' or total_miss = '0') then
        state <= read_canceled;
      elsif (write_enabled = '1') then
        state <= read_dc_write;
      else
        state <= read_dc;
      end if;
    else
      if (pc_valid = '0' or total_miss = '0' or write_enabled = '0') then
        state <= check_if;
      else
        state <= write_dc;
      end if;
    end if;

  when write_dc_read =>
    if (pc_valid = '1') then
      state <= read_if;
    else
      state <= read_canceled;
    end if;

  when write_dc =>
    state <= check_if;

  when read_dc_write =>
    if (mem_ready = '1') then
      if (pc_valid = '0' or total_miss = '0') then
        state <= write_dc;
      else
        state <= write_dc_read;
      end if;
    end if;

  when read_dc =>
    if (mem_ready = '1') then
      if (pc_valid = '0' or total_miss = '0') then
        state <= check_if;
      else
        state <= read_if;
      end if;
    end if;

  when read_canceled =>
    if (mem_ready = '1') then
      if (pc_valid = '0' or total_miss = '0') then
        state <= check_if;
      else
        state <= read_if;
      end if;
    end if;

  when others =>
    -- There should be no others!
    state <= check_if;

end case;

if (reset = '1') then
  state <= check_if;
end if;

if (state = check_if) then
  if (glock = '0') then
    instruction <= cache_data;
  end if;
elsif (state = read_if) then
  if (glock = '0') then
    instruction <= mem_data;
  end if;
```

```
    elsif (state = read_dc_write or
           state = read_dc) then
      instruction <= mem_data;
    end if;

    if (total_miss = '1') then
      if (state = check_if) then
        address <= real_pc;
      elsif (state = read_dc_write or
             state = read_dc       or
             state = read_canceled) then
        if (mem_ready = '1') then
          address <= real_pc;
        end if;
      end if;
    end if;

  end process;
end func;
```

E.5 ifetlb.vhdl.m4

```
ifdef({VIRTMEM},{
--------------------------------------------------------------------------------
-- Ifetch TLB (final version)
--
-- author: Steven Roos
--------------------------------------------------------------------------------

-- parameter dependencies:
--   {VIRTMEM}            : ifdef({VIRTMEM}, defined, not defined)
--   {BUSWIDTH}           : BUSWIDTH
--   {IFEFULLSIZE}        : IFEFULLSIZE
--   {IFEADDRSIZE}        : IFEADDRSIZE
--   {IFEPAGESIZE}        : IFEPAGESIZE
--   {IFETLBSIZE}         : IFETLBSIZE
--   {IFETLBBLOCKSIZE}    : IFETLBBLOCKSIZE
--   {IFETLBASSOCIATIVITY}: IFETLBASSOCIATIVITY
--   {IFETLBREPLACEMENT}  : IFETLBREPLACEMENT
--   {IFEFUT2S}           : IFEFUT2S


-- This version of the intruction address tlb uses the generic cache
-- implementation in cache.vhdl.m4. In fact, a tlb is no more than a
-- cache which contains the address translations which are most likely
-- to be used in the future. Though direct mapped implementations can
-- be used, 2 or more way set associative implementations are preferred.
--   The tlb can be written when it is disabled. To write the tlb, the next
-- procedure must be followed: first a virtual address must be written
-- to the tlb (opcode: IFE_TLB_VIRT_T2). This address should be aligned
-- as it is when you would normally use it, with page number, and a
-- dummy page offset after it. Then the real translation for that
-- address should be given (opcode: IFE_TLB_REAL_T2). The alignment of
-- this translated address is the same as the alignment of the virtual
-- address. The least significant bit is a valid bit. A '0' means
-- that this is an invalid translation. This gives the possibility
-- to use less translations than the number of tlb entries.
--   A miss will be reported when a PC is not found in the internal
-- cache, and the tlb is enabled. When the cycle after a write (real
-- or virtual address) the enabled signal is high, a miss is reported
-- too because in that case a valid lookup cannot be done.
--   When the tlb is disabled, the addresses are not translated.

-- In normal operation every read is started up on the rising edge of
-- a clock cycle, using new_pc as the cache address. The tag is
-- checked with virt_pc, which also contains the previous offset.
-- When an entry is written into the tlb, the virtual address is used
-- to look in the cache for a previous translation for that address.
-- The virtual address is saved in address_reg. That register is used
-- for the tag check. Nothing is really written into the cache when a
-- virtual address is written to the tlb. Only when a command is given
-- to write a real address to the tlb, the cache will be changed.
-- The address where the translated address is written to, is the
-- address which was stored in address_reg.

--# m4-parameters for the cache:
define({CACHENAME},     itlb_cache)
define({ADDRSIZE},      eval(IFEFULLSIZE-LOG(IFEPAGESIZE)))
define({DATASIZE},      eval(IFEADDRSIZE-LOG(IFEPAGESIZE)+1))
define({CACHESIZE},     IFETLBSIZE)
define({BLOCKSIZE},     IFETLBBLOCKSIZE)
define({ASSOCIATIVITY}, IFETLBASSOCIATIVITY)
```

```
define({REPLACEMENT},   IFETLBREPLACEMENT)
define({OFFSETSIZE},    LOG(CACHESIZE/ASSOCIATIVITY))
define({TAGSIZE},       eval(ADDRSIZE-OFFSETSIZE))
-- First, the cache code is included:
include(PATH/cache.vhdl.m4)

------------------------------------------------------------------------------
-- start of ifetlb
------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.opcodes.all;
use work.components.all;

entity ifetlb is
  port (
    ck       : in  std_logic;
    new_pc   : in  std_logic_vector (IFEFULLSIZE-1 downto 0);
    virt_pc  : in  std_logic_vector (IFEFULLSIZE-1 downto 0);
    real_pc  : out std_logic_vector (IFEADDRSIZE-1 downto 0);
    t_data   : in  std_logic_vector (BUSWIDTH-1 downto 0);
    t_active : in  std_logic;
    t_opcode : in  std_logic_vector (LOG(IFEFUT2S)-1 downto 0);
    glock    : in  std_logic;
    pc_valid : in  std_logic;
    enabled  : in  std_logic;
    lock_req : out std_logic;
    miss     : out std_logic);
end ifetlb;

architecture func of ifetlb is
  signal read_offset, write_offset   : std_logic_vector (OFFSETSIZE-1 downto 0);
  signal read_tag, write_tag         : std_logic_vector (TAGSIZE-1 downto 0);
  signal page_reg                    : std_logic_vector (ADDRSIZE-1 downto 0);
  signal data_in, data_out           : std_logic_vector (DATASIZE-1 downto 0);
  signal read_enabled, write_enabled : std_logic;
  signal cache_write, cache_miss     : std_logic;

  type tlb_states is (st_read, st_write_virt, st_write_real);
    -- Description of the states:
    --   st_write_virt:
    --     The clockcycle after a virtual page number is written
    --   st_write_real:
    --     The clockcycle after a real page number is written
    --   st_read:
    --     The TLB was not written in the previous cycle
  signal state : tlb_states;

  for cache_unit : itlb_cache use entity work.itlb_cache (func);

begin
  cache_unit: itlb_cache
    port map (
      ck            => ck,
      read_offset   => read_offset,
      read_tag      => read_tag,
      write_offset  => write_offset,
      write_tag     => write_tag,
      data_in       => data_in,
      data_out      => data_out,
      read_enabled  => read_enabled,
      write_enabled => write_enabled,
      write         => cache_write,
      miss          => cache_miss);

  real_pc <=
    data_out (DATASIZE-1 downto 1) & virt_pc (LOG(IFEPAGESIZE)-1 downto 0)
      when (enabled = '1') else
    virt_pc (IFEADDRSIZE-1 downto 0);

  lock_req <=
    '0' when (enabled = '0') else
    '1' when (state = st_write_virt or
              state = st_write_real) else
    '0';

  miss <=
    '0' when (enabled = '0') else
    '1' when (state = st_write_virt or
              state = st_write_real) else
    cache_miss or not data_out (0);

  read_offset <=
    t_data (eval(LOG(IFEPAGESIZE)+OFFSETSIZE)-1 downto LOG(IFEPAGESIZE))
      when (t_active = '1' and t_opcode = IFE_TLB_VIRT_T2) else
```

```
    new_pc (eval(LOG(IFEPAGESIZE)+OFFSETSIZE)-1 downto LOG(IFEPAGESIZE));
  read_tag <=
    page_reg (ADDRSIZE-1 downto OFFSETSIZE)
      when (state = st_write_virt) else
    virt_pc (IFEFULLSIZE-1 downto eval(LOG(IFEPAGESIZE)+OFFSETSIZE));

  write_offset <=
    page_reg (OFFSETSIZE-1 downto 0);
  write_tag <=
    page_reg (ADDRSIZE-1 downto OFFSETSIZE);

  data_in <=
    t_data (IFEADDRSIZE-1 downto LOG(IFEPAGESIZE)) & t_data (0);

  read_enabled <=
    '1' when (enabled = '1' and glock = '0' and pc_valid = '1') else
    '0';

  write_enabled <=
    '1' when (state = st_write_virt) else
    '0';

  cache_write <=
    '1' when (t_active = '1' and t_opcode = IFE_TLB_REAL_T2) else
    '0';

  process
  begin
    wait until ck'event and (ck = '1');

    if    (t_active = '1' and t_opcode = IFE_TLB_VIRT_T2) then
      state <= st_write_virt;
    elsif (t_active = '1' and t_opcode = IFE_TLB_REAL_T2) then
      state <= st_write_real;
    else
      state <= st_read;
    end if;

    if (t_active = '1' and t_opcode = IFE_TLB_VIRT_T2) then
      page_reg <= t_data (IFEFULLSIZE-1 downto LOG(IFEPAGESIZE));
    end if;

  end process;
end func;
})
```

E.6 ifeopc.vhdl.m4

```
ifdef({KERNEL}, {
--------------------------------------------------------------------------------
-- Ifetch FU opcode tester (final version)
--
-- Author: Steven Roos
--------------------------------------------------------------------------------

-- parameter dependencies:
--   {STAGES}         : STAGES
--   {PARMOVES}       : PARMOVES
  translit(FOR(BUSNR, 0, PARMOVES-1, {
--   {SRCSIZE\BUSNR}       : SRCSIZE\BUSNR
--   {DSTSIZE\BUSNR}       : DSTSIZE\BUSNR }),\)
--   {GRDSIZE}        : GRDSIZE
--   {IMM_EXTENSIONS} : IMM_EXTENSIONS

--   C???FU?_??BUS?
--   A???FU?_??BUS?
--   ???FU?S


-- This unit tests all opcodes on the instruction busses. When a privileged
-- opcode is found, the signal priv_move is raised when that instruction is
-- about to execute. That signal may be used to generate an exception.
-- Privileged opcodes in user code will always raise an exception, even
-- when the privileged instruction is squahed by a guard that evaluates
-- to false. This is done for performance reasons. An exception will not
-- be raised when the instruction is part of an immediate extension.

translit(

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

```
use work.opcodes.all;

entity ifeopc is
  port ( FOR(BUSNR, 0, PARMOVES-1, {
    src_id\BUSNR   : in  std_logic_vector (SRCSIZE\BUSNR-1 downto 0);
    dst_id\BUSNR   : in  std_logic_vector (DSTSIZE\BUSNR-1 downto 0); ifelse(eval(BUSNR<IMM_EXTENSIONS), 1, {
    imm_spec\BUSNR : in  std_logic; }) })
    priv_move : out std_logic);
end ifeopc;

architecture func of ifeopc is
  signal priv_src_opcode : std_logic_vector (PARMOVES-1 downto 0);
  signal priv_dst_opcode : std_logic_vector (PARMOVES-1 downto 0);
begin
    FOR(BUSNR, 0, PARMOVES-1, {
  priv_src_opcode (BUSNR) <= ifelse(eval(BUSNR<IMM_EXTENSIONS), 1, {
    '0' when (imm_spec\BUSNR = '1') else })
      FOR(FUNR, 0, IFEFUS-1, { ifelse(translit(CIFEFU\FUNR\_R1BUS\BUSNR,\), 1, {
    '1' when (src_id\BUSNR = std_logic_vector(conv_unsigned(AIFEFU\FUNR\_R1BUS\BUSNR/eval(2**LOG(IFEFUR1S)),
                             SRCSIZE\BUSNR-LOG(IFEFUR1S))) & IFE_PCC_R1) else }) })
      FOR(FUNR, 0, IFEFUS-1, { ifelse(translit(CIFEFU\FUNR\_R2BUS\BUSNR,\), 1, {
    '1' when (src_id\BUSNR = std_logic_vector(conv_unsigned(AIFEFU\FUNR\_R2BUS\BUSNR/eval(2**LOG(IFEFUR2S)),
                             SRCSIZE\BUSNR-LOG(IFEFUR2S))) & IFE_EXC_SOURCE_R2) else
    '1' when (src_id\BUSNR = std_logic_vector(conv_unsigned(AIFEFU\FUNR\_R2BUS\BUSNR/eval(2**LOG(IFEFUR2S)),
                             SRCSIZE\BUSNR-LOG(IFEFUR2S))) & IFE_TRAP_R2) else }) })
      FOR(FUNR, 0, SCLFUS-1, { ifelse(translit(CSCLFU\FUNR\_RBUS\BUSNR,\), 1, {
    '1' when (src_id\BUSNR = std_logic_vector(conv_unsigned(ASCLFU\FUNR\_RBUS\BUSNR/eval(2**LOG(SCLFURS)),
                             SRCSIZE\BUSNR-LOG(SCLFURS))) & SCL_CACHE_SAVED_R) else  ifdef({VIRTMEM}, {
    '1' when (src_id\BUSNR = std_logic_vector(conv_unsigned(ASCLFU\FUNR\_RBUS\BUSNR/eval(2**LOG(SCLFURS)),
                             SRCSIZE\BUSNR-LOG(SCLFURS))) & SCL_TLB_STATE_R) else
    '1' when (src_id\BUSNR = std_logic_vector(conv_unsigned(ASCLFU\FUNR\_RBUS\BUSNR/eval(2**LOG(SCLFURS)),
                             SRCSIZE\BUSNR-LOG(SCLFURS))) & SCL_TLB_SAVED_R) else }) }) })
      FOR(FUNR, 0, LDSFUS-1, { ifdef({VIRTMEM}, { ifelse(translit(CLDSFU\FUNR\_R2BUS\BUSNR,\), 1, {
    '1' when (src_id\BUSNR = std_logic_vector(conv_unsigned(ALDSFU\FUNR\_R2BUS\BUSNR,
                             SRCSIZE\BUSNR))) else }) }) })
    '0';

  priv_dst_opcode (BUSNR) <= ifelse(eval(BUSNR<IMM_EXTENSIONS), 1, {
    '0' when (imm_spec\BUSNR = '1') else })
      FOR(FUNR, 0, IFEFUS-1, { ifelse(translit(CIFEFU\FUNR\_T1BUS\BUSNR,\), 1, {
    '1' when (dst_id\BUSNR = std_logic_vector(conv_unsigned(AIFEFU\FUNR\_T1BUS\BUSNR/eval(2**LOG(IFEFUT1S)),
                             DSTSIZE\BUSNR-LOG(IFEFUT1S))) & IFE_JUMP_RFE_T1) else }) })
      FOR(FUNR, 0, IFEFUS-1, { ifelse(translit(CIFEFU\FUNR\_T2BUS\BUSNR,\), 1, {
    '1' when (dst_id\BUSNR = std_logic_vector(conv_unsigned(AIFEFU\FUNR\_T2BUS\BUSNR/eval(2**LOG(IFEFUT2S)),
                             DSTSIZE\BUSNR-LOG(IFEFUT2S))) & IFE_PCC_T2) else  ifdef({VIRTMEM}, {
    '1' when (dst_id\BUSNR = std_logic_vector(conv_unsigned(AIFEFU\FUNR\_T2BUS\BUSNR/eval(2**LOG(IFEFUT2S)),
                             DSTSIZE\BUSNR-LOG(IFEFUT2S))) & IFE_TLB_VIRT_T2) else
    '1' when (dst_id\BUSNR = std_logic_vector(conv_unsigned(AIFEFU\FUNR\_T2BUS\BUSNR/eval(2**LOG(IFEFUT2S)),
                             DSTSIZE\BUSNR-LOG(IFEFUT2S))) & IFE_TLB_REAL_T2) else }) }) })
      FOR(FUNR, 0, IFEFUS-1, { ifelse(translit(CIFEFU\FUNR\_T3BUS\BUSNR,\), 1, {
    '1' when (dst_id\BUSNR = std_logic_vector(conv_unsigned(AIFEFU\FUNR\_T3BUS\BUSNR/eval(2**LOG(IFEFUT3S)),
                             DSTSIZE\BUSNR-LOG(IFEFUT3S))) & IFE_EXC_SOURCE_T3) else }) })
      FOR(FUNR, 0, SCLFUS-1, { ifelse(translit(CSCLFU\FUNR\_TBUS\BUSNR,\), 1, {
    '1' when (dst_id\BUSNR = std_logic_vector(conv_unsigned(ASCLFU\FUNR\_TBUS\BUSNR/eval(2**LOG(SCLFUTS)),
                             DSTSIZE\BUSNR-LOG(SCLFUTS))) & SCL_CACHE_SAVED_T) else  ifdef({VIRTMEM}, {
    '1' when (dst_id\BUSNR = std_logic_vector(conv_unsigned(ASCLFU\FUNR\_TBUS\BUSNR/eval(2**LOG(SCLFUTS)),
                             DSTSIZE\BUSNR-LOG(SCLFUTS))) & SCL_TLB_STATE_T) else
    '1' when (dst_id\BUSNR = std_logic_vector(conv_unsigned(ASCLFU\FUNR\_TBUS\BUSNR/eval(2**LOG(SCLFUTS)),
                             DSTSIZE\BUSNR-LOG(SCLFUTS))) & SCL_TLB_SAVED_T) else }) }) })
      FOR(FUNR, 0, LDSFUS-1, { ifdef({VIRTMEM}, { ifelse(translit(CLDSFU\FUNR\_TBUS\BUSNR,\), 1, {
    '1' when (dst_id\BUSNR = std_logic_vector(conv_unsigned(ALDSFU\FUNR\_TBUS\BUSNR/eval(2**LOG(LDSFUTS)),
                             DSTSIZE\BUSNR-LOG(LDSFUTS))) & LDS_TLB_T) else }) }) })
    '0';

    })
  priv_move <=
    '1' when (priv_src_opcode /= "FOR(CNT, 0, PARMOVES-1, 0)") else
    '1' when (priv_dst_opcode /= "FOR(CNT, 0, PARMOVES-1, 0)") else
    '0';

end func;

,\)
})
```

E.7 ifeexc.vhdl.m4

```
--------------------------------------------------------------------------------
-- Ifetch FU exception controller (final version)
--
-- Author: Steven Roos
--------------------------------------------------------------------------------
```

```
-- parameter dependencies:
--   {STAGES}           : STAGES
--   {VIRTMEM}          : ifdef({VIRTMEM}, defined, not defined)
--   {KERNEL}           : ifdef({KERNEL}, defined, not defined)
--   {BUSWIDTH}         : BUSWIDTH
--   {IFEFUT3S}         : IFEFUT3S
--   {IFEFUR2S}         : IFEFUR2S
--   {LDSFUS}           : LDSFUS
--   {IFE_TRAPVECTORS}  : IFE_TRAPVECTORS

--   {IFE_RESET_VECTOR} : IFE_RESET_VECTOR
--   {IFE_INT_VECTOR}   : IFE_INT_VECTOR
--   {IFE_TRAP_VECTOR}  : IFE_TRAP_VECTOR
--   {IFE_ITLB_VECTOR}  : IFE_ITLB_VECTOR
--   {IFE_DTLB_VECTOR}  : IFE_DTLB_VECTOR
--   {IFE_OPCODE_VECTOR}: IFE_OPCODE_VECTOR


define({EXC_SOURCES}, 2)
ifdef({VIRTMEM}, {
  define({EXC_SOURCES}, eval(EXC_SOURCES+LDSFUS+1))
})
ifdef({KERNEL}, {
  define({EXC_SOURCES}, eval(EXC_SOURCES+1))
})

library IEEE;
use IEEE.std_logic_1164.all;
use work.opcodes.all;

entity ifeexc is
  port (
    ck          : in  std_logic;
    -- bus interfaces:
    t_data      : in  std_logic_vector (BUSWIDTH-1 downto 0);
    t_active    : in  std_logic;
    t_opcode    : in  std_logic_vector (LOG(IFEFUT3S)-1 downto 0);
    r_data      : out std_logic_vector (BUSWIDTH-1 downto 0);
    r_opcode    : in  std_logic_vector (LOG(IFEFUR2S)-1 downto 0);
    -- exception requests:
    reset       : in  std_logic;
    int         : in  std_logic;  ifdef({VIRTMEM}, {
    itlb_miss   : in  std_logic;
    dtlb_miss   : in  std_logic_vector (LDSFUS-1 downto 0); }) ifdef({KERNEL}, {
    priv_opcode : in  std_logic;  })
    -- miscellaneous control signals:
    glock       : in  std_logic;
    exc_enabled : in  std_logic;  ifdef({KERNEL}, {
    user_mode   : in  std_logic;  })
    lock_req    : out std_logic;
    pc_valid    : out std_logic;
    exc_request : out std_logic;
    exc_vector  : out std_logic_vector (IFEFULLSIZE-1 downto 0));
end ifeexc;

architecture func of ifeexc is
  type exception_requests is (exc_reset, exc_int, exc_trap, ifdef({VIRTMEM}, {
                              exc_itlb, exc_dtlb, }) ifdef({KERNEL}, {
                              exc_opcode,}) no_request);
  signal exception_request : exception_requests;
    ifdef({KERNEL}, {  ifelse(STAGES, 3, {
  signal priv_opcode_dc, priv_opcode_mv : std_logic; },{
  signal priv_opcode_mv : std_logic; }) })
    ifdef({VIRTMEM}, { ifelse(STAGES, 3, {
  signal itlb_miss_if, itlb_miss_dc, itlb_miss_mv : std_logic; },{
  signal itlb_miss_if, itlb_miss_mv : std_logic; })

  signal dtlb_exc_enabled : std_logic;

  signal pc_valid_reg : std_logic; })

  signal trap : std_logic;
  signal trap_reg : std_logic_vector (LOG(IFE_TRAPVECTORS)-1 downto 0);
  signal exc_sources : std_logic_vector (EXC_SOURCES-1 downto 0);

begin
  lock_req <=
    '0' when (exc_enabled = '0') else
    '1' when (exc_sources (1) = '1' or
              exc_sources (0) = '1') else  ifdef({VIRTMEM}, {
    '1' when (dtlb_miss      /= "FOR(CNT, 0, LDSFUS-1, 0)") else
    '1' when (itlb_miss_mv    = '1') else  }) ifdef({KERNEL}, {
    '1' when (priv_opcode_mv  = '1' and user_mode = '1') else })
    '0';

  pc_valid <=
```

```
    '0' when (exception_request /= no_request) else ifdef({VIRTMEM}, {
    '0' when (itlb_miss_if = '1') else
    pc_valid_reg; },{
    '1'; })

  trap <=
    '1' when (t_active = '1' and t_opcode = IFE_TRAP_T3) else
    '0';

  process (reset, int, trap, exc_sources, ifdef({KERNEL}, {
           user_mode, priv_opcode_mv, ifelse(STAGES, 3, {priv_opcode_dc, }) })
           ifdef({VIRTMEM}, {dtlb_miss, itlb_miss_mv, ifelse(STAGES, 3,
           {itlb_miss_dc, }, {itlb_miss_if, })
           dtlb_exc_enabled, })exc_enabled, glock)
  begin
    exception_request <= no_request; -- default

    if (reset = '1') then
      exception_request <= exc_reset; ifdef({VIRTMEM}, {
    elsif (dtlb_exc_enabled = '1' and
           dtlb_miss /= "FOR(CNT, 0, LDSFUS-1, 0)") then
      exception_request <= exc_dtlb; })
    elsif (exc_enabled = '1') then
      if    (int = '1') then
        exception_request <= exc_int;
      elsif (glock = '1') then
        if    (exc_sources (0) = '1') then -- int waiting
          exception_request <= exc_int;
        elsif (exc_sources (1) = '1') then -- trap waiting
          exception_request <= exc_trap; ifdef({VIRTMEM}, {
        elsif (itlb_miss_mv = '1') then
          exception_request <= exc_itlb; }) ifdef({KERNEL}, {
        elsif (priv_opcode_mv = '1' and user_mode = '1') then
          exception_request <= exc_opcode; })
        end if;
      else
        if    (trap = '1') then
          exception_request <= exc_trap; ifelse(STAGES, 3, { ifdef({VIRTMEM}, {
        elsif (itlb_miss_dc = '1') then
          exception_request <= exc_itlb; }) ifdef({KERNEL}, {
        elsif (priv_opcode_dc = '1' and user_mode = '1') then
          exception_request <= exc_opcode; }) },{ ifdef({VIRTMEM}, {
        elsif (itlb_miss_if = '1') then
          exception_request <= exc_itlb; }) })
        end if;
      end if;
    end if;
  end process;

  exc_request <=
    '1' when (exception_request /= no_request) else
    '0';

  exc_vector <=
    IFE_RESET_VECTOR
      when (exception_request = exc_reset ) else
    IFE_INT_VECTOR
      when (exception_request = exc_int    ) else
    IFE_TRAP_VECTOR
      when (exception_request = exc_trap  ) else ifdef({VIRTMEM}, {
    IFE_ITLB_VECTOR
      when (exception_request = exc_itlb  ) else
    IFE_DTLB_VECTOR
      when (exception_request = exc_dtlb  ) else }) ifdef({KERNEL}, {
    IFE_OPCODE_VECTOR
      when (exception_request = exc_opcode) else })
    (others => '-');           ifdef({VIRTMEM}, {

  itlb_miss_if   <= itlb_miss; }) ifdef({KERNEL}, { ifelse(STAGES, 3, {
  priv_opcode_dc <= priv_opcode;  },{
  priv_opcode_mv <= priv_opcode;  }) })

  process
  begin
    wait until ck'event and (ck = '1');

      ifdef({VIRTMEM}, {
    if (exception_request /= no_request) then
      pc_valid_reg <= '1';
    elsif (itlb_miss_if = '1' and glock = '0') then
      pc_valid_reg <= '0';
    end if;

    if (glock = '0') then
      dtlb_exc_enabled <= exc_enabled;
        ifelse(STAGES, 3, {
      itlb_miss_dc   <= itlb_miss_if;
```

```
      itlb_miss_mv   <= itlb_miss_dc; },{
      itlb_miss_mv   <= itlb_miss_if; })
    end if; })
      ifdef({KERNEL}, { ifelse(STAGES, 3, {
    if (glock = '0') then
      priv_opcode_mv <= priv_opcode_dc;
    end if; }) })

    if (t_active = '1') then
      case t_opcode is
        when IFE_TRAP_T3 =>
          trap_reg <= t_data (LOG(IFE_TRAPVECTORS)-1 downto 0);
        when IFE_EXC_SOURCE_T3 =>
          exc_sources <= t_data (EXC_SOURCES-1 downto 0);
        when others =>
          null;
      end case;
    end if;

    if (int = '1') then
      exc_sources (0) <= '1';
    end if;
    if (trap = '1') then
      exc_sources (1) <= '1';
    end if; ifdef({KERNEL}, {
    if (exception_request = exc_opcode) then
      exc_sources (2) <= '1';
    end if; }) ifdef({VIRTMEM}, {
    if (exception_request = exc_itlb) then
      exc_sources (eval(EXC_SOURCES-LDSFUS-1)) <= '1';
    end if;

    if (exception_request = exc_dtlb) then ifdef({KERNEL}, {
      exc_sources (2) <= '0'; })
      exc_sources (eval(EXC_SOURCES-LDSFUS-1)) <= '0';
    end if;
      FOR(FUNR, 0, LDSFUS-1, {
    if (dtlb_miss (FUNR) = '1') then
      exc_sources (eval(EXC_SOURCES-LDSFUS+FUNR)) <= '1';
    end if; }) })

    if (reset = '1') then
      exc_sources <= (others => '0');
    end if;
  end process;

  process (r_opcode, trap_reg, exc_sources)
  begin
    r_data <= (others => '0');
    case r_opcode is
      when IFE_TRAP_R2 =>
        r_data (LOG(IFE_TRAPVECTORS)-1 downto 0) <= trap_reg;
      when IFE_EXC_SOURCE_R2 =>
        r_data (EXC_SOURCES-1 downto 0) <= exc_sources;
      when others =>
        r_data <= (others => '-');
    end case;
  end process;
end func;
```

E.8 ifefui.vhdl.m4

```
--------------------------------------------------------------------------------
-- Ifetch FU internals (final version)
--
-- author: Steven Roos
--------------------------------------------------------------------------------

-- parameter dependencies:
--   {STAGES}        : STAGES
--   {VIRTMEM}       : ifdef({VIRTMEM}, defined, not defined)
--   {KERNEL}        : ifdef({KERNEL}, defined, not defined)
--   {PARMOVES}      : PARMOVES
--   {BUSWIDTH}      : BUSWIDTH

--   {IFEFUT1S}      : IFEFUT1S
--   {IFEFUT2S}      : IFEFUT2S
--   {IFEFUT3S}      : IFEFUT3S
--   {IFEFUR1S}      : IFEFUR1S
--   {IFEFUR2S}      : IFEFUR2S
  translit(FOR(BUSNR, 0, PARMOVES-1, {
--   {SRCSIZE\BUSNR}      : SRCSIZE\BUSNR
```

```
--   {DSTSIZE\BUSNR}      : DSTSIZE\BUSNR }),\)
--   {GRDSIZE}         GRDSIZE
--   {IMM_EXTENSIONS}: IMM_EXTENSIONS
--   {INSTRSIZE}     : INSTRSIZE

--   {IFEFULLSIZE}   : IFEFULLSIZE
--   {IFEADDRSIZE}   : IFEADDRSIZE
--   {LDSFUS}        : LDSFUS


-- This is the complete ifetch unit. It is basically no more than a shell
-- around its components. It hardly does anything with the signals itself.
-- It just defines how the components are connected, and which signals are
-- made visible to the rest of the processor.
-- It contains code and signals which are only generated when kernel mode
-- is enabled.

-- The ifetch unit has a three input sockets. The first one is used for jumps
-- and calls. The second is used to write the fuction return address and
-- the PC-chain, which is used as a return address for exceptions. When
-- virtual memory is supported, the tlb can be written via this socket too.
-- The third socket is used for traps and writing the exception state.
--   Jumps and calls may have three address sizes. Their sizes can be specified
-- using m4 parameters. A convenient value for the smallest size is the
-- size of a short immediate. An appropriate value for the medium size may be
-- the size of a short immediate plus the immeiate extension from one bus.
-- The largest size must be at least the number of instruction address bits.
--   All jumps can be carried out with, or without saving the address of the
-- first instruction that is not executed. When that address is saved, it is
-- called a call. A special jump instruction is the return from exception.
-- This privileged instruction works as a full sized jump, but has the side
-- effect of restoring the previously saved state and enabling exceptions.
--   The second socket can write the PC-chain and the function return address.
-- When the PC-chain is written, the values in this chain ripple through the
-- chain. The whole chain can be written by writing STAGES addresses into it.
-- Writing the PC-chain is a privileged operation.
--   The function return address is written automatically at a call. For
-- state saving and restoring purposes, it can be written via a socket too.

-- The registers that can be read are the fuction return address, and the
-- PC-chain. To perform a function return, the value that is read from the
-- function return address register must be used to perform a normal jump.
-- The only value that can be read from the PC-chain is the address of the
-- first instruction that was not completed before the exception. Reading
-- this register does not cause the PC-chain to shift forward. To read the
-- other values of the PC-chain, new values must be put in it to force the
-- other entries to ripple into the readable register. Reading from the
-- PC-chain is privileged.
--   Via the second output socket the exception source registers can be read.
-- One register contains a bit for all possible exception sources. The other
-- register contains the last value that was written into the trap register.
-- Reading these registers is privileged.

library IEEE;
use IEEE.std_logic_1164.all;
use work.opcodes.all;
use work.components.all;

entity ifefui is
  port (
    ck            : in  std_logic;
    -- socket interfaces:
    t1_data       : in  std_logic_vector (BUSWIDTH-1 downto 0);
    t1_active     : in  std_logic;
    t1_opcode     : in  std_logic_vector (LOG(IFEFUT1S)-1 downto 0);

    t2_data       : in  std_logic_vector (BUSWIDTH-1 downto 0);
    t2_active     : in  std_logic;
    t2_opcode     : in  std_logic_vector (LOG(IFEFUT2S)-1 downto 0);

    t3_data       : in  std_logic_vector (BUSWIDTH-1 downto 0);
    t3_active     : in  std_logic;
    t3_opcode     : in  std_logic_vector (LOG(IFEFUT3S)-1 downto 0);

    r1_data       : out std_logic_vector (BUSWIDTH-1 downto 0);
    r1_opcode     : in  std_logic_vector (LOG(IFEFUR1S)-1 downto 0);

    r2_data       : out std_logic_vector (BUSWIDTH-1 downto 0);
    r2_opcode     : in  std_logic_vector (LOG(IFEFUR2S)-1 downto 0);
    -- instruction bus interface:   translit(FOR(BUSNR, 0, PARMOVES-1, {
    src_id\BUSNR       : out std_logic_vector (SRCSIZE\BUSNR-1 downto 0);
    dst_id\BUSNR       : out std_logic_vector (DSTSIZE\BUSNR-1 downto 0);
    grd_spec\BUSNR     : out std_logic_vector (GRDSIZE-1 downto 0); })
      FOR(BUSNR, 0, IMM_EXTENSIONS-1, {
    imm_spec\BUSNR     : out std_logic; }),\)
    -- memory unit interface:
    mem_data      : in  std_logic_vector (INSTRSIZE-1 downto 0);
```

```
    mem_ready     : in  std_logic;
    mem_address   : out std_logic_vector (IFEADDRSIZE-1 downto 0);
    mem_strobe    : out std_logic;
    -- exception request inputs:
    reset         : in  std_logic;
    int           : in  std_logic; ifdef({VIRTMEM}, {
    dtlb_miss     : in  std_logic_vector (LDSFUS-1 downto 0); })
    -- control signals:
    glock         : in  std_logic;
    cache_enabled : in  std_logic_vector (2-1 downto 0); ifdef({VIRTMEM},{
    tlb_enabled   : in  std_logic; }) ifdef({KERNEL},{
    user_mode     : in  std_logic; })
    exc_enabled   : in  std_logic;
    exc_return    : out std_logic;
    exc_call      : out std_logic;
    lock_req      : out std_logic);
end ifefui;

architecture func of ifefui is
  signal pc, new_pc : std_logic_vector (IFEFULLSIZE-1 downto 0);
  signal real_pc : std_logic_vector (IFEADDRSIZE-1 downto 0);

  signal cache_lockreq : std_logic; ifdef({VIRTMEM}, {
  signal tlb_lockreq   : std_logic; })
  signal exc_lockreq   : std_logic;
  signal exc_request   : std_logic; -- forces the next PC to be exc_vector
  signal exc_vector    : std_logic_vector (IFEFULLSIZE-1 downto 0);
  signal pc_valid      : std_logic; -- indicates that the current PC points to
                                    -- an instruction that will not be executed.
    ifdef({VIRTMEM},{
  signal itlb_miss   : std_logic; }) ifdef({KERNEL},{
  signal priv_move   : std_logic;
  signal priv_opcode : std_logic; })

  signal instruction : std_logic_vector (INSTRSIZE-1 downto 0);
  -- instruction format (n+1 busses):
  -- |IMMm|...|IMM0|GRDn|DSTn|SRCn|...|...|...|GRD1|DST1|SRC1|GRD0|DST0|SRC0|
    translit( FOR(BUSNR, 0, PARMOVES-1, {
  signal source_id\BUSNR :
    std_logic_vector (SRCSIZE\BUSNR-1 downto 0);
  signal destination_id\BUSNR :
    std_logic_vector (DSTSIZE\BUSNR-1 downto 0);
  signal guard_specifier\BUSNR :
    std_logic_vector (GRDSIZE-1 downto 0); })
      FOR(BUSNR, 0, IMM_EXTENSIONS-1, {
  signal immediate_specifier\BUSNR : std_logic; }),\)

  for exc_control : ifeexc use entity work.ifeexc (func);
  for pc_selector : ifesel use entity work.ifesel (func);
  for cache       : ifecch use entity work.ifecch (func); ifdef({VIRTMEM}, {
  for tlb         : ifetlb use entity work.ifetlb (func); }) ifdef({KERNEL},{
  for opcode_test : ifeopc use entity work.ifeopc (func); })

begin
  -- The exception controller:
  exc_control: ifeexc
    port map (
      ck          => ck,
      t_data      => t3_data,
      t_active    => t3_active,
      t_opcode    => t3_opcode,
      r_data      => r2_data,
      r_opcode    => r2_opcode,
      reset       => reset,
      int         => int,         ifdef({VIRTMEM}, {
      itlb_miss   => itlb_miss,
      dtlb_miss   => dtlb_miss,   })  ifdef({KERNEL}, {
      priv_opcode => priv_opcode, })
      glock       => glock,
      exc_enabled => exc_enabled, ifdef({KERNEL}, {
      user_mode   => user_mode,   })
      lock_req    => exc_lockreq,
      pc_valid    => pc_valid,
      exc_request => exc_request,
      exc_vector  => exc_vector);

  -- The PC selector. This unit also contains the PC-chain:
  pc_selector: ifesel
    port map (
      ck          => ck,
      glock       => glock,
      exc_request => exc_request,
      exc_vector  => exc_vector,
      exc_enabled => exc_enabled,
      exc_return  => exc_return,
      exc_call    => exc_call,
      t1_data     => t1_data,
```

```
      t1_active   => t1_active,
      t1_opcode   => t1_opcode,
      t2_data     => t2_data,
      t2_active   => t2_active,
      t2_opcode   => t2_opcode,
      r_data      => r1_data,
      r_opcode    => r1_opcode,
      new_pc      => new_pc,
      pc          => pc);

  -- The cache and int controller:
  cache: ifecch
    port map (
      ck          => ck,
      new_pc      => new_pc,
      real_pc     => real_pc,
      mem_data    => mem_data,
      mem_ready   => mem_ready,
      mem_address => mem_address,
      mem_strobe  => mem_strobe,
      instruction => instruction,
      reset       => reset,
      glock       => glock,
      pc_valid    => pc_valid,
      enabled     => cache_enabled,
      lock_req    => cache_lockreq);  ifdef({VIRTMEM}, {

  -- The TLB and its controller:
  tlb: ifetlb
    port map (
      ck       => ck,
      new_pc   => new_pc,
      virt_pc  => pc,
      real_pc  => real_pc,
      t_data   => t2_data,
      t_active => t2_active,
      t_opcode => t2_opcode,
      glock    => glock,
      pc_valid => pc_valid,
      enabled  => tlb_enabled,
      lock_req => tlb_lockreq,
      miss     => itlb_miss); }) ifdef({KERNEL}, {

  -- The unit which scans instructions for privileged opcodes:
  opcode_test: ifeopc
    port map ( translit(FOR(BUSNR, 0, PARMOVES-1, {
      src_id\BUSNR   => source_id\BUSNR{,}
      dst_id\BUSNR   => destination_id\BUSNR{,} ifelse(eval(BUSNR<IMM_EXTENSIONS), 1, {
      imm_spec\BUSNR => immediate_specifier\BUSNR{,} }) }) ,\)
      priv_move => priv_move);

  priv_opcode <= priv_move and not cache_lockreq; }) ifdef({VIRTMEM}, {
  lock_req <= cache_lockreq or tlb_lockreq or exc_lockreq; },{
  lock_req <= cache_lockreq or exc_lockreq;
  real_pc <= pc (IFEADDRSIZE-1 downto 0); })

  -- instruction format (n+1 busses):
  -- |IMMm|...|IMM0|GRDn|DSTn|SRCn|...|...|...|GRD1|DST1|SRC1|GRD0|DST0|SRC0|
  translit( define({UPPER}, 0)
        FOR(BUSNR, 0, PARMOVES-1, {
      define({LOWER}, UPPER)
      define({UPPER}, eval(translit(UPPER+SRCSIZE\BUSNR,\)))
  source_id\BUSNR           <= instruction (UPPER-1 downto LOWER);
      define({LOWER}, UPPER)
      define({UPPER}, eval(translit(UPPER+DSTSIZE\BUSNR,\)))
  destination_id\BUSNR      <= instruction (UPPER-1 downto LOWER);
      define({LOWER}, UPPER)
      define({UPPER}, eval(UPPER+GRDSIZE))
  guard_specifier\BUSNR     <= instruction (UPPER-1 downto LOWER); })
    FOR(BUSNR, 0, IMM_EXTENSIONS-1, {
      define({UPPER}, eval(UPPER+1))
  immediate_specifier\BUSNR <= instruction (UPPER-1); })
    undefine({UPPER}) undefine({LOWER})
    FOR(BUSNR, 0, PARMOVES-1, {
  src_id\BUSNR   <= source_id\BUSNR;
  dst_id\BUSNR   <= destination_id\BUSNR;
  grd_spec\BUSNR <= guard_specifier\BUSNR; ifelse(eval(BUSNR<IMM_EXTENSIONS),1,{
  imm_spec\BUSNR <= immediate_specifier\BUSNR; })
    }),\)

end func;
```

E.9 sclfui.vhdl.m4

```
-------------------------------------------------------------------------------
-- State & control FU internals (final version)
--
-- Author: Steven Roos
-------------------------------------------------------------------------------

-- parameter dependencies:
--   {STAGES}         : STAGES
--   {VIRTMEM}        : ifdef({VIRTMEM}, defined, not defined)
--   {KERNEL}         : ifdef({KERNEL}, defined, not defined)
--   {BUSWIDTH}       : BUSWIDTH
--   {SCLFUTS}        : SCLFUTS
--   {SCLFURS}        : SCLFURS

--   {SCL_ICACHE_BITS}: SCL_ICACHE_BITS
--   {SCL_DCACHE_BITS}: SCL_DCACHE_BITS
--   {SCL_ITLB_BITS}  : SCL_ITLB_BITS
--   {SCL_DTLB_BITS}  : SCL_DTLB_BITS


library IEEE;
use IEEE.std_logic_1164.all;
use work.opcodes.all;

entity sclfui is
  port (
    ck            : in  std_logic;
    reset         : in  std_logic;
    t_data        : in  std_logic_vector (BUSWIDTH-1 downto 0);
    t_active      : in  std_logic;
    t_opcode      : in  std_logic_vector (LOG(SCLFUTS)-1 downto 0);
    r_data        : out std_logic_vector (BUSWIDTH-1 downto 0);
    r_opcode      : in  std_logic_vector (LOG(SCLFURS)-1 downto 0);
    glock         : in  std_logic;
    exc_call      : in  std_logic;
    exc_return    : in  std_logic;
    exc_enabled   : out std_logic;
      ifelse(eval((SCL_ICACHE_BITS+SCL_DCACHE_BITS) > 0), 1, {
    cache_enabled : out std_logic_vector (eval(SCL_ICACHE_BITS+SCL_DCACHE_BITS)-1 downto 0); })
      ifelse(eval((SCL_ITLB_BITS+SCL_DTLB_BITS) > 0), 1, {
    tlb_enabled   : out std_logic_vector (eval(SCL_ITLB_BITS+SCL_DTLB_BITS)-1 downto 0); })
      ifdef({KERNEL}, {
    user_mode     : out std_logic; })
    squall_req    : out std_logic);
end sclfui;

architecture func of sclfui is
  type exc_states is (exc_if, ifelse(STAGES, 3, {exc_dc, })exc_mv, disabled,
                      rfe_if, ifelse(STAGES, 3, {rfe_dc, })rfe_mv, enabled);
  signal exc_state, exc_state_reg : exc_states;

  signal prev_glock : std_logic;

  type   write_actions is (cache_state, cache_saved, ifdef({VIRTMEM}, {
                           tlb_state  , tlb_saved  , })no_action);
  signal write_action : write_actions;
    ifelse(eval(SCL_ICACHE_BITS+SCL_DCACHE_BITS > SCL_ITLB_BITS+SCL_DTLB_BITS), 1, {
  signal write_reg : std_logic_vector (eval(SCL_ICACHE_BITS+SCL_DCACHE_BITS)-1 downto 0); },{
  signal write_reg : std_logic_vector (eval(SCL_ITLB_BITS+SCL_DTLB_BITS)-1 downto 0); })

    ifelse(eval(SCL_ICACHE_BITS>0), 1, {
  -- icache enabled signals:
  signal icache_written     : std_logic_vector (SCL_ICACHE_BITS-1 downto 0);
  signal icache_user_if     : std_logic_vector (SCL_ICACHE_BITS-1 downto 0);
  signal icache_user_if_reg : std_logic_vector (SCL_ICACHE_BITS-1 downto 0); ifelse(STAGES, 3, {
  signal icache_user_dc     : std_logic_vector (SCL_ICACHE_BITS-1 downto 0);
  signal icache_user_dc_new : std_logic_vector (SCL_ICACHE_BITS-1 downto 0);
  signal icache_user_dc_reg : std_logic_vector (SCL_ICACHE_BITS-1 downto 0); })
  signal icache_user_mv     : std_logic_vector (SCL_ICACHE_BITS-1 downto 0);
  signal icache_user_mv_reg : std_logic_vector (SCL_ICACHE_BITS-1 downto 0);
  signal icache_exc         : std_logic_vector (SCL_ICACHE_BITS-1 downto 0);
  signal icache_exc_reg     : std_logic_vector (SCL_ICACHE_BITS-1 downto 0); })

    ifelse(eval(SCL_ITLB_BITS>0), 1, {
  signal itlb_written       : std_logic_vector (SCL_ITLB_BITS-1 downto 0);
  signal itlb_user_if       : std_logic_vector (SCL_ITLB_BITS-1 downto 0);
  signal itlb_user_if_reg   : std_logic_vector (SCL_ITLB_BITS-1 downto 0); ifelse(STAGES, 3, {
  signal itlb_user_dc       : std_logic_vector (SCL_ITLB_BITS-1 downto 0);
  signal itlb_user_dc_new   : std_logic_vector (SCL_ITLB_BITS-1 downto 0);
  signal itlb_user_dc_reg   : std_logic_vector (SCL_ITLB_BITS-1 downto 0); })
  signal itlb_user_mv       : std_logic_vector (SCL_ITLB_BITS-1 downto 0);
  signal itlb_user_mv_reg   : std_logic_vector (SCL_ITLB_BITS-1 downto 0);
  signal itlb_exc           : std_logic_vector (SCL_ITLB_BITS-1 downto 0);
  signal itlb_exc_reg       : std_logic_vector (SCL_ITLB_BITS-1 downto 0); })
```

```
    ifelse(eval(SCL_DCACHE_BITS>0), 1, {
  -- dcache enabled signals:
  signal dcache_written     : std_logic_vector (SCL_DCACHE_BITS-1 downto 0);
  signal dcache_user        : std_logic_vector (SCL_DCACHE_BITS-1 downto 0);
  signal dcache_user_reg    : std_logic_vector (SCL_DCACHE_BITS-1 downto 0);
  signal dcache_exc         : std_logic_vector (SCL_DCACHE_BITS-1 downto 0);
  signal dcache_exc_reg     : std_logic_vector (SCL_DCACHE_BITS-1 downto 0); })

    ifelse(eval(SCL_DTLB_BITS>0), 1, {
  -- dtlb enabled signals:
  signal dtlb_written       : std_logic_vector (SCL_DTLB_BITS-1 downto 0);
  signal dtlb_user          : std_logic_vector (SCL_DTLB_BITS-1 downto 0);
  signal dtlb_user_reg      : std_logic_vector (SCL_DTLB_BITS-1 downto 0);
  signal dtlb_exc           : std_logic_vector (SCL_DTLB_BITS-1 downto 0);
  signal dtlb_exc_reg       : std_logic_vector (SCL_DTLB_BITS-1 downto 0); })

begin
 --=========================================================================
  -- Local state changes:
  --=========================================================================

  squall_req <=
    '1' when (exc_state = exc_if) else ifelse(STAGES, 3, {
    '1' when (exc_state = exc_dc) else })
    '0';

  exc_enabled <=
    '1' when (exc_state = enabled) else
    '1' when (exc_state = rfe_mv ) else
    '0';

  ifdef({KERNEL}, {
  -- The user mode is chosen to be the same as the mode in which
  -- exceptions are enabled. These modes can be used independantly,
  -- but combining them made the control easier.
  user_mode <=
    '1' when (exc_state = enabled) else
    '1' when (exc_state = rfe_mv ) else
    '0'; })

  exc_state <=
    exc_if when (exc_call   = '1') else
    rfe_if when (exc_return = '1' and exc_state_reg = disabled) else
    exc_state_reg;

  process
  begin
    wait until ck'event and (ck = '1');

    prev_glock <= glock;

    if    (exc_state = exc_if and glock = '0') then ifelse(STAGES, 3, {
      exc_state_reg <= exc_dc;
    elsif (exc_state = exc_dc and glock = '0') then })
      exc_state_reg <= exc_mv;
    elsif (exc_state = exc_mv and glock = '0') then
      exc_state_reg <= disabled;
    elsif (exc_state = rfe_if and glock = '0') then ifelse(STAGES, 3, {
      exc_state_reg <= rfe_dc;
    elsif (exc_state = rfe_dc and glock = '0') then })
      exc_state_reg <= rfe_mv;
    elsif (exc_state = rfe_mv and glock = '0') then
      exc_state_reg <= enabled;
    else
      exc_state_reg <= exc_state;
    end if;

    write_reg    <= t_data (write_reg'length-1 downto 0);
    write_action <= no_action;

    if (t_active = '1') then
      case t_opcode is
        when SCL_CACHE_STATE_T => write_action <= cache_state;
        when SCL_CACHE_SAVED_T => write_action <= cache_saved; ifdef({VIRTMEM}, {
        when SCL_TLB_STATE_T   => write_action <= tlb_state;
        when SCL_TLB_SAVED_T   => write_action <= tlb_saved; })
        when others            => null;
      end case;
    end if;
  end process;

  ifelse(eval(SCL_ICACHE_BITS>0), 1, {
  --=========================================================================
  -- Icache state changes:
  --=========================================================================
```

```
icache_written <=
  write_reg (SCL_ICACHE_BITS-1 downto 0);

cache_enabled (SCL_ICACHE_BITS-1 downto 0) <=
  icache_user_if
    when (exc_state = rfe_if or ifelse(STAGES, 3, {
          exc_state = rfe_dc or })
          exc_state = rfe_mv or
          exc_state = enabled) else
  icache_exc;

process (icache_user_if_reg, ifelse(STAGES, 3, {icache_user_dc_reg, })icache_user_mv_reg,
         icache_exc_reg, icache_written,
         exc_state, write_action, prev_glock, exc_return)
begin
  icache_user_if <= icache_user_if_reg; ifelse(STAGES, 3, {
  icache_user_dc <= icache_user_dc_reg; })
  icache_user_mv <= icache_user_mv_reg;
  icache_exc     <= icache_exc_reg;

  if (prev_glock = '0') then
    if (write_action = cache_state) then
      if (exc_state = exc_if or ifelse(STAGES, 3, {
          exc_state = rfe_dc or })
          exc_state = rfe_mv or
          exc_state = enabled) then
        icache_user_if <= icache_written;
      else
        icache_exc     <= icache_written;
      end if;
    elsif (write_action = cache_saved) then
      if (exc_state = disabled) then
        icache_user_mv <= ifelse(STAGES, 3, {icache_user_dc_reg;
        icache_user_dc <= })icache_user_if_reg;
        icache_user_if <= icache_written;
      end if;
    end if;

    if (exc_state = exc_if or
        exc_state = rfe_if or ifelse(STAGES, 3, {
        exc_state = rfe_dc or })
        exc_state = rfe_mv or
        exc_state = enabled) then
      icache_user_mv <= ifelse(STAGES, 3, {icache_user_dc_reg;
      icache_user_dc <= })icache_user_if_reg;
    end if;

    if (exc_return = '1') then
      icache_user_if <= icache_user_mv_reg;
    end if;
  end if;
end process;

process
begin
  wait until ck'event and (ck = '1');

  icache_user_if_reg <= icache_user_if; ifelse(STAGES, 3, {
  icache_user_dc_reg <= icache_user_dc; })
  icache_user_mv_reg <= icache_user_mv;
  icache_exc_reg     <= icache_exc;

  if (reset = '1') then
    icache_exc_reg <= (others => '0');
  end if;
end process;

})
ifelse(eval(SCL_DCACHE_BITS>0), 1, {
--=========================================================================
-- Dcache state changes:
--=========================================================================

dcache_written <=
  write_reg (eval(SCL_ICACHE_BITS+SCL_DCACHE_BITS)-1 downto SCL_ICACHE_BITS);

cache_enabled (eval(SCL_ICACHE_BITS+SCL_DCACHE_BITS)-1 downto SCL_ICACHE_BITS) <=
  dcache_user
    when (exc_state = exc_if or ifelse(STAGES, 3, {
          exc_state = exc_dc or })
          exc_state = rfe_mv or
          exc_state = enabled) else
  dcache_exc;

process (dcache_user_reg, dcache_exc_reg, dcache_written,
         exc_state, write_action, prev_glock)
begin
```

```
  dcache_user <= dcache_user_reg;
  dcache_exc  <= dcache_exc_reg;

  if (prev_glock = '0') then
    if (write_action = cache_state) then
      if (exc_state = exc_if or
          exc_state = enabled) then
        dcache_user <= dcache_written;
      else
        dcache_exc  <= dcache_written;
      end if;
    elsif (write_action = cache_saved) then
      if (exc_state = rfe_if   or ifelse(STAGES, 3, {
          exc_state = rfe_dc   or })
          exc_state = disabled) then
        dcache_user <= dcache_written;
      end if;
    end if;
  end if;
end process;

process
begin
  wait until ck'event and (ck = '1');

  dcache_user_reg <= dcache_user;
  dcache_exc_reg  <= dcache_exc;

  if (reset = '1') then
    dcache_exc_reg <= (others => '0');
  end if;
end process;

})
ifelse(eval(SCL_ITLB_BITS>0), 1, {
--===========================================================================
-- Itlb state changes:
--===========================================================================

itlb_written <=
  write_reg (SCL_ITLB_BITS-1 downto 0);

tlb_enabled (SCL_ITLB_BITS-1 downto 0) <=
  itlb_user_if
    when (exc_state = rfe_if or ifelse(STAGES, 3, {
          exc_state = rfe_dc or })
          exc_state = rfe_mv or
          exc_state = enabled) else
  itlb_exc;

process (itlb_user_if_reg, ifelse(STAGES, 3, {itlb_user_dc_reg, })itlb_user_mv_reg,
         itlb_exc_reg, itlb_written,
         exc_state, write_action, prev_glock, exc_return)
begin
  itlb_user_if <= itlb_user_if_reg; ifelse(STAGES, 3, {
  itlb_user_dc <= itlb_user_dc_reg; })
  itlb_user_mv <= itlb_user_mv_reg;
  itlb_exc     <= itlb_exc_reg;

  if (prev_glock = '0') then
    if (write_action = tlb_state) then
      if (exc_state = exc_if or ifelse(STAGES, 3, {
          exc_state = rfe_dc or })
          exc_state = rfe_mv or
          exc_state = enabled) then
        itlb_user_if <= itlb_written;
      else
        itlb_exc     <= itlb_written;
      end if;
    elsif (write_action = tlb_saved) then
      if (exc_state = disabled) then
        itlb_user_mv <= ifelse(STAGES, 3, {itlb_user_dc_reg;
        itlb_user_dc <= })itlb_user_if_reg;
        itlb_user_if <= itlb_written;
      end if;
    end if;

    if (exc_state = exc_if or
        exc_state = rfe_if or ifelse(STAGES, 3, {
        exc_state = rfe_dc or })
        exc_state = rfe_mv or
        exc_state = enabled) then
      itlb_user_mv <= ifelse(STAGES, 3, {itlb_user_dc_reg;
      itlb_user_dc <= })itlb_user_if_reg;
    end if;

    if (exc_return = '1') then
```

```
      itlb_user_if <= itlb_user_mv_reg;
    end if;
  end if;
end process;

process
begin
  wait until ck'event and (ck = '1');

  itlb_user_if_reg <= itlb_user_if; ifelse(STAGES, 3, {
  itlb_user_dc_reg <= itlb_user_dc; })
  itlb_user_mv_reg <= itlb_user_mv;
  itlb_exc_reg     <= itlb_exc;

  if (reset = '1') then
    itlb_exc_reg <= (others => '0');
  end if;
end process;


})
ifelse(eval(SCL_DTLB_BITS>0), 1, {
--===========================================================================
-- Dtlb state changes:
--===========================================================================

dtlb_written <=
  write_reg (eval(SCL_ITLB_BITS+SCL_DTLB_BITS)-1 downto SCL_ITLB_BITS);

tlb_enabled (eval(SCL_ITLB_BITS+SCL_DTLB_BITS)-1 downto SCL_ITLB_BITS) <=
  dtlb_user
    when (exc_state = exc_if or ifelse(STAGES, 3, {
          exc_state = exc_dc or })
          exc_state = rfe_mv or
          exc_state = enabled) else
  dtlb_exc;

process (dtlb_user_reg, dtlb_exc_reg, dtlb_written,
         exc_state, write_action, prev_glock)
begin
  dtlb_user <= dtlb_user_reg;
  dtlb_exc  <= dtlb_exc_reg;

  if (prev_glock = '0') then
    if (write_action = tlb_state) then
      if (exc_state = exc_if or
          exc_state = enabled) then
        dtlb_user <= dtlb_written;
      else
        dtlb_exc  <= dtlb_written;
      end if;
    elsif (write_action = tlb_saved) then
      if (exc_state = rfe_if   or ifelse(STAGES, 3, {
          exc_state = rfe_dc   or })
          exc_state = disabled) then
        dtlb_user <= dtlb_written;
      end if;
    end if;
  end if;
end process;

process
begin
  wait until ck'event and (ck = '1');

  dtlb_user_reg <= dtlb_user;
  dtlb_exc_reg  <= dtlb_exc;

  if (reset = '1') then
    dtlb_exc_reg <= (others => '0');
  end if;
end process;

})

--===========================================================================
-- Socket outputs:
--===========================================================================

process (ifelse(eval(SCL_ICACHE_BITS > 0), 1, {icache_exc, icache_user_if, icache_user_mv, })
         ifelse(eval(SCL_DCACHE_BITS > 0), 1, {dcache_exc, dcache_user, })
         ifelse(eval(SCL_ITLB_BITS   > 0), 1, {itlb_exc, itlb_user_if, itlb_user_mv, })
         ifelse(eval(SCL_DTLB_BITS   > 0), 1, {dtlb_exc  , dtlb_user  , })
         exc_state, r_opcode)
begin
  r_data <= (others => '0');
```

```
    ifelse(eval(SCL_ICACHE_BITS > 0), 1, {
    -- icache output:
    if (r_opcode = SCL_CACHE_STATE_R) then
      if (exc_state = rfe_if or ifelse(STAGES, 3, {
          exc_state = rfe_dc or })
          exc_state = rfe_mv or
          exc_state = enabled) then
        r_data (SCL_ICACHE_BITS-1 downto 0) <= icache_user_if;
      else
        r_data (SCL_ICACHE_BITS-1 downto 0) <= icache_exc;
      end if;
    elsif (r_opcode = SCL_CACHE_SAVED_R) then
      r_data (SCL_ICACHE_BITS-1 downto 0) <= icache_user_mv;
    end if;
    })

    ifelse(eval(SCL_DCACHE_BITS > 0), 1, {
    -- dcache output:
    if (r_opcode = SCL_CACHE_STATE_R) then
      if (exc_state = rfe_mv or
          exc_state = enabled) then
        r_data (eval(SCL_DCACHE_BITS+SCL_ICACHE_BITS)-1 downto SCL_ICACHE_BITS) <= dcache_user;
      else
        r_data (eval(SCL_DCACHE_BITS+SCL_ICACHE_BITS)-1 downto SCL_ICACHE_BITS) <= dcache_exc;
      end if;
    elsif (r_opcode = SCL_CACHE_SAVED_R) then
      r_data (eval(SCL_DCACHE_BITS+SCL_ICACHE_BITS)-1 downto SCL_ICACHE_BITS) <= dcache_user;
    end if;
    })

    ifelse(eval(SCL_ITLB_BITS > 0), 1, {
    -- itlb output:
    if (r_opcode = SCL_TLB_STATE_R) then
      if (exc_state = rfe_if or ifelse(STAGES, 3, {
          exc_state = rfe_dc or })
          exc_state = rfe_mv or
          exc_state = enabled) then
        r_data (SCL_ITLB_BITS-1 downto 0) <= itlb_user_if;
      else
        r_data (SCL_ITLB_BITS-1 downto 0) <= itlb_exc;
      end if;
    elsif (r_opcode = SCL_TLB_SAVED_R) then
      r_data (SCL_ITLB_BITS-1 downto 0) <= itlb_user_mv;
    end if;
    })

    ifelse(eval(SCL_DTLB_BITS > 0), 1, {
    -- dtlb output:
    if (r_opcode = SCL_TLB_STATE_R) then
      if (exc_state = rfe_mv or
          exc_state = enabled) then
        r_data (eval(SCL_DTLB_BITS+SCL_ITLB_BITS)-1 downto SCL_ITLB_BITS) <= dtlb_user;
      else
        r_data (eval(SCL_DTLB_BITS+SCL_ITLB_BITS)-1 downto SCL_ITLB_BITS) <= dtlb_exc;
      end if;
    elsif (r_opcode = SCL_TLB_SAVED_R) then
      r_data (eval(SCL_DTLB_BITS+SCL_ITLB_BITS)-1 downto SCL_ITLB_BITS) <= dtlb_user;
    end if;
    })
  end process;
end func;
```

E.10 immfui.vhdl.m4

```
-------------------------------------------------------------------------------
-- Immediate FU internals (final version)
--
-- Author: Steven Roos
-------------------------------------------------------------------------------

-- parameter dependencies:
--   {STAGES}          : STAGES
--   {BUSWIDTH}        : BUSWIDTH
--   {PARMOVES}        : PARMOVES

--   {IMM_SHORTBITS}   : IMM_SHORTBITS
--   {IMM_LONGBITS}    : IMM_LONGBITS
--   {IMM_EXTSIZE}     : IMM_EXTSIZE
--   {IMM_SIGNEXT}     : ifdef({IMM_SIGNEXT}, defined, not defined)
--   {IMM_REG}         : ifdef({IMM_REG}, defined, not defined)
--   {IMM_SHIFT}       : ifdef({IMM_SHIFT}, defined, not defined)
--   {IMM_SHIFTSIZE}   : ifdef({IMM_SHIFTSIZE}, IMM_SHIFTSIZE, not defined)
```

```
--   {IMMFUR1S}       : IMMFUR1S
--   {IMMFUR2S}       : IMMFUR2S
  translit(FOR(BUSNR, 0, PARMOVES-1, {
--   {SRCSIZE\BUSNR}       : SRCSIZE\BUSNR
--   {DSTSIZE\BUSNR}       : DSTSIZE\BUSNR }),\)
--   {GRDSIZE}        : GRDSIZE
--   {IMM_EXTENSIONS} : IMM_EXTENSIONS
  translit(FOR(BUSNR, 0, PARMOVES-1, {
--   {CIMMFU0_R1BUS\BUSNR} : CIMMFU0_R1BUS\BUSNR
--   {CIMMFU0_R2BUS\BUSNR} : CIMMFU0_R2BUS\BUSNR }),\)


-- This immediate unit assumes that it is the only immediate unit.
-- r1: short immediates
-- r2: long immediates

translit(

library IEEE;
use IEEE.std_logic_1164.all;

entity immfui is
  port (
    ck         : in  std_logic;  ifdef({IMM_REG}, {
    t_data     : in  std_logic_vector (BUSWIDTH-1 downto 0);
    t_active   : in  std_logic;  })
      FOR(BUSNR, 0, PARMOVES-1, { ifelse(translit(CIMMFU0_R1BUS\BUSNR,\), 1, {
    r1_data\BUSNR   : out std_logic_vector (BUSWIDTH-1 downto 0);
    r1_opcode\BUSNR : in  std_logic_vector (LOG(IMMFUR1S)-1 downto 0); }) })
      FOR(BUSNR, 0, PARMOVES-1, { ifelse(translit(CIMMFU0_R2BUS\BUSNR,\), 1, {
    r2_data\BUSNR   : out std_logic_vector (BUSWIDTH-1 downto 0);
    r2_opcode\BUSNR : in  std_logic_vector (LOG(IMMFUR2S)-1 downto 0); }) })
      FOR(BUSNR, 0, IMM_EXTENSIONS-1, {
    src_id\BUSNR    : in  std_logic_vector (SRCSIZE\BUSNR-1 downto 0);
    dst_id\BUSNR    : in  std_logic_vector (DSTSIZE\BUSNR-1 downto 0);
    grd_spec\BUSNR  : in  std_logic_vector (GRDSIZE-1 downto 0);
    imm_spec\BUSNR  : in  std_logic; }) ifdef({IMM_REG},{
    squall     : in  std_logic; })
    glock      : in  std_logic);
end immfui;

ifelse(eval(IMM_EXTENSIONS>0), 1, {
-------------------------------------------------------------------------------
-- Immediate unit architecture which supports short and long immediates:
-------------------------------------------------------------------------------

architecture func of immfui is
      ifdef({IMM_REG},{
  -- The immediate register:
  signal imm_reg : std_logic_vector (IMM_EXTSIZE-1 downto 0); })
  -- The immediate extension that will be used in the current cycle:
  signal imm_ext : std_logic_vector (IMM_EXTSIZE-1 downto 0);
  -- The immediate information that was collected in the decode stage:
  signal imm_new : std_logic_vector (ifdef({IMM_SHIFT}, {IMM_SHIFTSIZE}, {IMM_EXTSIZE})-1 downto 0);
  -- The signal that specifies whether a new extension from the decode
  -- stage has to be used:
  signal new_ext : std_logic;

  -- The new immediate extension parts:       FOR(BUSNR, 0, IMM_EXTENSIONS-1, {
  signal imm_bus\BUSNR : std_logic_vector (eval(translit(SRCSIZE\BUSNR+DSTSIZE\BUSNR+GRDSIZE,\))-1 downto 0); })

begin
    FOR(BUSNR, 0, IMM_EXTENSIONS-1, {
  imm_bus\BUSNR <=
    grd_spec\BUSNR & dst_id\BUSNR & src_id\BUSNR
      when (imm_spec\BUSNR = '1') else
    (others => ifdef({IMM_SIGNEXT}, {ifelse(BUSNR, 0, {'0'},
               {imm_bus\eval(BUSNR-1) (imm_bus\eval(BUSNR-1)'high)})}, {'0'})); })

    ifelse(STAGES, 3, {
  process
  begin
    wait until ck'event and (ck = '1');

    if (glock = '0') then })
      define({LOW}, 0)
      FOR(BUSNR, 0, IMM_EXTENSIONS-1, {
       define({HIGH}, eval(translit(LOW+SRCSIZE\BUSNR+DSTSIZE\BUSNR+GRDSIZE,\)))
       ifelse(eval(HIGH>IMM_EXTSIZE), 1, { define({HIGH}, IMM_EXTSIZE) })
    imm_new (HIGH-1 downto LOW) <= imm_bus\BUSNR (eval(HIGH-LOW)-1 downto 0);
       define({LOW}, HIGH) })
       undefine({LOW})  undefine({HIGH})
    new_ext <= imm_spec0 FOR(BUSNR, 1, IMM_EXTENSIONS-1, {or
               imm_spec\BUSNR });
      ifelse(STAGES, 3, {
    end if;
```

```
  end process; })

    ifdef({IMM_REG},{
  process
  begin
    wait until ck'event and (ck = '1');

    if (glock = '0' and squall = '0') then
      if (t_active = '1') then
        imm_reg <= t_data (eval(IMM_EXTSIZE+IMM_LONGBITS)-1 downto IMM_LONGBITS);
      else
        imm_reg <= imm_ext;
      end if;
    end if;
  end process;

  imm_ext <=
    imm_reg when (new_ext = '0') else  ifdef({IMM_SHIFT}, {
    imm_reg (eval(IMM_EXTSIZE-IMM_SHIFTSIZE)-1 downto 0) & })imm_new;
    },{
  imm_ext <= imm_new;
    })

    FOR(BUSNR, 0, PARMOVES-1, { ifelse(translit(CIMMFU0_R1BUS\BUSNR,\), 1, {
  process (r1_opcode\BUSNR)
  begin  ifdef({IMM_SIGNEXT}, {
    r1_data\BUSNR <= (others => r1_opcode\BUSNR (IMM_SHORTBITS-1)); },{
    r1_data\BUSNR <= (others => '0'); })
    r1_data\BUSNR (IMM_SHORTBITS-1 downto 0) <= r1_opcode\BUSNR;
  end process; }) })

    FOR(BUSNR, 0, PARMOVES-1, { ifelse(translit(CIMMFU0_R2BUS\BUSNR,\), 1, {
  process (r2_opcode\BUSNR{,} imm_ext{,} imm_new, new_ext)
  begin  ifdef({IMM_REG}, { ifelse(eval(BUSWIDTH>IMM_LONGBITS+IMM_EXTSIZE), 1, {
        ifdef({IMM_SIGNEXT}, {
    r2_data\BUSNR <= (others => imm_ext (IMM_EXTSIZE-1)); },{
    r2_data\BUSNR <= (others => '0'); }) })
    r2_data\BUSNR (eval(IMM_EXTSIZE+IMM_LONGBITS)-1 downto IMM_LONGBITS) <= imm_ext; },{
      ifdef({IMM_SIGNEXT}, { ifelse(eval(BUSWIDTH>IMM_LONGBITS+IMM_EXTSIZE), 1,{
    if (new_ext = '1') then
      r2_data\BUSNR <= (others => imm_ext (IMM_EXTSIZE-1));
    else
      r2_data\BUSNR <= (others => r2_opcode\BUSNR (IMM_LONGBITS-1));
    end if; },{
    r2_data\BUSNR <= (others => r2_opcode\BUSNR (IMM_LONGBITS-1)); }) },{
    r2_data\BUSNR <= (others => '0'); })
    if (new_ext = '1') then
      r2_data\BUSNR (eval(IMM_EXTSIZE+IMM_LONGBITS)-1 downto IMM_LONGBITS) <= imm_ext;
    end if; })
    r2_data\BUSNR (IMM_LONGBITS-1 downto 0) <= r2_opcode\BUSNR;
  end process; }) })

end func;

},{
-------------------------------------------------------------------------------
-- Immediate unit architecture which only supports short immediates:
-------------------------------------------------------------------------------
architecture func of immfui is
begin
    FOR(BUSNR, 0, PARMOVES-1, { ifelse(translit(CIMMFU0_R1BUS\BUSNR,\), 1, {
  process (r1_opcode\BUSNR)
  begin  ifdef({IMM_SIGNEXT}, {
    r1_data\BUSNR <= (others => r1_opcode\BUSNR (IMM_SHORTBITS-1)); },{
    r1_data\BUSNR <= (others => '0'); })
    r1_data\BUSNR (IMM_SHORTBITS-1 downto 0) <= r1_opcode\BUSNR;
  end process; }) })
end func;

}),\)
```

E.11 cache.vhdl.m4

```
--------------------------------------------------------------------------------
-- Generic cache unit (final version)
--
-- Author: Steven Roos
--------------------------------------------------------------------------------

-- parameter dependencies:
--   {CACHENAME}    : CACHENAME
--   {ADDRSIZE}     : ADDRSIZE
```

```
--   {DATASIZE}     : DATASIZE
--   {CACHESIZE}    : CACHESIZE
--   {BLOCKSIZE}    : BLOCKSIZE
--   {ASSOCIATIVITY}: ASSOCIATIVITY
--   {REPLACEMENT}  : REPLACEMENT
--   {OFFSETSIZE}   : OFFSETSIZE
--   {TAGSIZE}      : TAGSIZE


-- Every time a cache is used, a new {CACHENAME} must be used. A new
-- entity and architecture called {CACHENAME} are generated, using the
-- m4 parameters {BLOCKSIZE, ASSOCIATIVITY and REPLACEMENT} for the
-- architecture, and the parameters {OFFSETSIZE, TAGSIZE, DATASIZE and
-- CACHESIZE} to determine the actual sizes.

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.components.all;

entity CACHENAME is
  port (
    ck            : in  std_logic;
    read_offset   : in  std_logic_vector (OFFSETSIZE-1 downto 0);
    read_tag      : in  std_logic_vector (TAGSIZE-1 downto 0);
    write_offset  : in  std_logic_vector (OFFSETSIZE-1 downto 0);
    write_tag     : in  std_logic_vector (TAGSIZE-1 downto 0);
    data_in       : in  std_logic_vector (DATASIZE-1 downto 0);
    data_out      : out std_logic_vector (DATASIZE-1 downto 0);
    read_enabled  : in  std_logic;
    write_enabled : in  std_logic;
    write         : in  std_logic;
    miss          : out std_logic);
end CACHENAME;

-- At cycle 0 the cache is read with read_offset. The tag is checked at cycle 1
-- with read_tag. When a LRU replacement strategy is used, the LRU state is
-- written when 'read_enabled' is '1'. The state which is used at a write is
-- written when 'write_enabled' is '1'. This state includes the bank in
-- which the entry must be written (set-associative caches) and the valid
-- bits for all subblocks.
-- 'write_offset' chooses the line to write to, and 'write_tag' is the tag
-- to be written.

ifelse(ASSOCIATIVITY, 1, {
ifelse(BLOCKSIZE, 1, {
-------------------------------------------------------------------------------
-- A direct mapped cache with a block size of 1:
-------------------------------------------------------------------------------

architecture func of CACHENAME is
  signal ram_address : std_logic_vector (OFFSETSIZE-1 downto 0);
  signal d_in, d_out : std_logic_vector (eval(DATASIZE+TAGSIZE)-1 downto 0);
  signal rwn : std_logic;  -- 1 = read; 0 = write

  for ram: ramcell use entity work.ramcell (func);
begin

  ram: ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => CACHESIZE,
      wordlength => eval(DATASIZE+TAGSIZE))
    port map (
      ck    => ck,
      addr  => ram_address,
      rwn   => rwn,
      d_in  => d_in,
      d_out => d_out);

  rwn <= not write;
  d_in <= data_in & write_tag;
  ram_address <= write_offset when (write = '1') else
                 read_offset;

  data_out <= d_out (eval(DATASIZE+TAGSIZE)-1 downto TAGSIZE);
  miss <= '0' when (d_out (TAGSIZE-1 downto 0) = read_tag) else
          '1';
end func;

}, {
-------------------------------------------------------------------------------
-- a direct mapped cache with a blocksize > 1:
-------------------------------------------------------------------------------

-- This implementation uses two ram cells. One cell contains all tags and
-- all control information. The other cell only contains data.
```

```
architecture func of CACHENAME is
  signal tag_hit : boolean;
  signal subblock : integer range 0 to BLOCKSIZE-1;
  -- decides which part of the cache line must be used.
  signal ram_address : std_logic_vector (OFFSETSIZE-1 downto 0);
  signal valid, valid_reg : std_logic_vector (BLOCKSIZE-1 downto 0);
  -- Vaild_reg contains the state that is saved at a read. In valid_reg is
  -- stored which entries on a cache line contain valid data, in respect to
  -- the current tag. valid contains the next valid_reg, and is used at
  -- a write.
  signal ctrl_in, ctrl_out : std_logic_vector (eval(TAGSIZE+BLOCKSIZE)-1 downto 0);
  signal rwn : std_logic;

  for ctrl_ram, data_ram: ramcell use entity work.ramcell (func);
begin

  ctrl_ram: ramcell
    generic map (
      addr_range => eval(OFFSETSIZE-LOG(BLOCKSIZE)),
      words      => eval(CACHESIZE/BLOCKSIZE),
      wordlength => eval(TAGSIZE+BLOCKSIZE))
    port map (
      ck    => ck,
      addr  => ram_address (OFFSETSIZE-1 downto LOG(BLOCKSIZE)),
      rwn   => rwn,
      d_in  => ctrl_in,
      d_out => ctrl_out);
  data_ram: ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => CACHESIZE,
      wordlength => DATASIZE)
    port map (
      ck    => ck,
      addr  => ram_address,
      rwn   => rwn,
      d_in  => data_in,
      d_out => data_out);

  rwn <= not write;
  ctrl_in <= write_tag & valid;
  -- The input of the ram cell that contains the valid bits and the tags.
  ram_address <= write_offset when (write = '1') else
                 read_offset;

  tag_hit <= (ctrl_out (eval(TAGSIZE+BLOCKSIZE)-1 downto BLOCKSIZE) = read_tag);
  miss <= '0' when (tag_hit and (ctrl_out (subblock) = '1')) else
          '1';
  -- miss when the tags don't match, or when that particlar entry on the cache
  -- line is not valid.

  process (write_enabled, tag_hit, subblock, valid_reg, ctrl_out)
  begin
    if (write_enabled = '0') then
      valid <= valid_reg;
    elsif (tag_hit) then  -- tag will not change at a write
      valid <= ctrl_out (BLOCKSIZE-1 downto 0);
      valid (subblock) <= '1';
    else                  -- at a write, another tag will be written.
      valid <= "FOR(CNT, 0, BLOCKSIZE-1, {0})";
      valid (subblock) <= '1';
    end if;
  end process;

  process
  begin
    wait until ck'event and (ck = '1');

    subblock <= conv_integer (unsigned (ram_address (LOG(BLOCKSIZE)-1 downto 0)));
    valid_reg <= valid;
  end process;
end func;

})
}, REPLACEMENT, FIFO, {
ifelse(BLOCKSIZE, 1, {
-------------------------------------------------------------------------------
-- a 2-way set associative cache with a block size of 1:
-- (this implementation uses a FIFO replacement policy)
-------------------------------------------------------------------------------

-- This implementation uses 2 ram cells: one for every bank.

architecture func of CACHENAME is
  signal ram_address: std_logic_vector (OFFSETSIZE-1 downto 0);
  signal d_in, d0_out, d1_out : std_logic_vector (eval(DATASIZE+TAGSIZE+1)-1 downto 0);
```

```
  signal tag_hit0, tag_hit1 : boolean;
  signal rwn0, rwn1 : std_logic;
  signal toggle, toggle_reg : std_logic;
  -- Toggle bits are used to detemine which of both ram cells must be updated.
  -- When a value is written to a line in one of both banks, the toggle bit on
  -- that line is changed, so the next time that line is read, the bank
  -- specifier will point to the other ram cell.
  signal ram_sel, ramsel_reg : std_logic;
  -- This bit specifies which of both banks will be written.

  for ram0, ram1: ramcell use entity work.ramcell (func);
begin

  ram0 : ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => eval(CACHESIZE/2),
      wordlength => eval(DATASIZE+TAGSIZE+1))
    port map (
      ck    => ck,
      addr  => ram_address,
      rwn   => rwn0,
      d_in  => d_in,
      d_out => d0_out);

  ram1 : ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => eval(CACHESIZE/2),
      wordlength => eval(DATASIZE+TAGSIZE+1))
    port map (
      ck    => ck,
      addr  => ram_address,
      rwn   => rwn1,
      d_in  => d_in,
      d_out => d1_out);

  d_in <= data_in & write_tag & toggle;
  ram_address <= write_offset when (write = '1') else
                 read_offset;
  rwn0 <= not (write and not ram_sel);
  rwn1 <= not (write and ram_sel);

  tag_hit0 <= (read_tag = d0_out (eval(TAGSIZE+1)-1 downto 1));
  tag_hit1 <= (read_tag = d1_out (eval(TAGSIZE+1)-1 downto 1));
  miss <= '0' when (tag_hit0 or tag_hit1) else
          '1';
  data_out <=
    d0_out (eval(DATASIZE+TAGSIZE+1)-1 downto eval(TAGSIZE+1))
      when (tag_hit0) else
    d1_out (eval(DATASIZE+TAGSIZE+1)-1 downto eval(TAGSIZE+1));

  process (write_enabled, tag_hit0, tag_hit1,
           ramsel_reg, toggle_reg, d0_out, d1_out)
  begin
    if (write_enabled = '0') then
    -- disabled, so keep old state:
      ram_sel <= ramsel_reg;
      toggle <= toggle_reg;
    elsif (tag_hit0) then
    -- bank 0 already contains the right tag; refresh the data:
      ram_sel <= '0';
      toggle <= not d1_out (0);
    elsif (tag_hit1) then
    -- bank 1 already contains the right tag; refresh the data:
      ram_sel <= '1';
      toggle <= d0_out (0);
    else
    -- determine which is the least recently written ram cell:
      ram_sel <= d0_out (0) xor d1_out (0);
      toggle <= not d1_out (0);
    end if;
  end process;

  process
  begin
    wait until ck'event and (ck = '1');

    ramsel_reg <= ram_sel;
    toggle_reg <= toggle;
  end process;
end func;

},{
--------------------------------------------------------------------------------
-- a 2-way set associative cache with a block size > 1:
-- (this implemetation uses a FIFO replacement policy)
```

```
------------------------------------------------------------------------------

-- This cache implementation uses 4 ram cells, two for every bank. The data
-- are contained in one cell, and the tags and control variables are contained
-- in the other cell.

architecture func of CACHENAME is
  signal ram_address: std_logic_vector (OFFSETSIZE-1 downto 0);
  signal data0_out, data1_out : std_logic_vector (DATASIZE-1 downto 0);
  -- datalines of the data ram cells
  signal ctrl_in, ctrl0_out, ctrl1_out : std_logic_vector (eval(TAGSIZE+1+BLOCKSIZE)-1 downto 0);
  -- data lines of the tag & control ram cells
  signal subblock : integer range 0 to BLOCKSIZE-1;
  -- specifies which part of the cache line should be used
  signal tag_hit0, tag_hit1 : boolean;
  signal rwn0, rwn1 : std_logic;
  signal valid, valid_reg : std_logic_vector (BLOCKSIZE-1 downto 0);
  -- specifies which parts of the cacheline will be valid after a write
  signal toggle, toggle_reg : std_logic;
  -- The toggle bits are used to implement the FIFO algorithm
  signal ram_sel, ramsel_reg : std_logic;
  -- selects which bank to write

  for ctrl0_ram, ctrl1_ram: ramcell use entity work.ramcell (func);
  for data0_ram, data1_ram: ramcell use entity work.ramcell (func);
begin

  ctrl0_ram: ramcell
    generic map (
      addr_range => eval(OFFSETSIZE-LOG(BLOCKSIZE)),
      words      => eval((CACHESIZE/2)/BLOCKSIZE),
      wordlength => eval(TAGSIZE+1+BLOCKSIZE))
    port map (
      ck    => ck,
      addr  => ram_address (OFFSETSIZE-1 downto LOG(BLOCKSIZE)),
      rwn   => rwn0,
      d_in  => ctrl_in,
      d_out => ctrl0_out);

  ctrl1_ram: ramcell
    generic map (
      addr_range => eval(OFFSETSIZE-LOG(BLOCKSIZE)),
      words      => eval((CACHESIZE/2)/BLOCKSIZE),
      wordlength => eval(TAGSIZE+1+BLOCKSIZE))
    port map (
      ck    => ck,
      addr  => ram_address (OFFSETSIZE-1 downto LOG(BLOCKSIZE)),
      rwn   => rwn1,
      d_in  => ctrl_in,
      d_out => ctrl1_out);

  data0_ram: ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => eval(CACHESIZE/2),
      wordlength => DATASIZE)
    port map (
      ck    => ck,
      addr  => ram_address,
      rwn   => rwn0,
      d_in  => data_in,
      d_out => data0_out);

  data1_ram: ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => eval(CACHESIZE/2),
      wordlength => DATASIZE)
    port map (
      ck    => ck,
      addr  => ram_address,
      rwn   => rwn1,
      d_in  => data_in,
      d_out => data1_out);

  ctrl_in <= write_tag & toggle & valid;
  ram_address <= write_offset when (write = '1') else
                 read_offset;
  rwn0 <= not (write and not ram_sel);
  rwn1 <= not (write and ram_sel);

  tag_hit0 <= (read_tag = ctrl0_out (eval(TAGSIZE+1+BLOCKSIZE)-1 downto eval(1+BLOCKSIZE)));
  tag_hit1 <= (read_tag = ctrl1_out (eval(TAGSIZE+1+BLOCKSIZE)-1 downto eval(1+BLOCKSIZE)));
  miss <= '0' when (tag_hit0 and (ctrl0_out (subblock) = '1')) else
          '0' when (tag_hit1 and (ctrl1_out (subblock) = '1')) else
          '1';
  data_out <=
```

```
    data0_out
      when (tag_hit0) else
    data1_out;

  process (write_enabled, tag_hit0, tag_hit1, ramsel_reg, toggle_reg,
           valid_reg, ctrl0_out, ctrl1_out, subblock)
  begin
    if (write_enabled = '0') then
    -- disabled, so keep old state
      ram_sel <= ramsel_reg;
      toggle <= toggle_reg;
      valid <= valid_reg;
    elsif (tag_hit0) then
    -- the tag from bank 0 matched; this bank should be refreshed at a write
      ram_sel <= '0';
      toggle <= not ctrl1_out (BLOCKSIZE);
      valid <= ctrl0_out (BLOCKSIZE-1 downto 0);
      valid (subblock) <= '1';
    elsif (tag_hit1) then
    -- the tag from bank 1 matched; this bank should be refreshed at a write
      ram_sel <= '1';
      toggle <= ctrl0_out (BLOCKSIZE);
      valid <= ctrl1_out (BLOCKSIZE-1 downto 0);
      valid (subblock) <= '1';
    else
    -- select the least recently written
      ram_sel <= ctrl0_out (BLOCKSIZE) xor ctrl1_out (BLOCKSIZE);
      toggle <= not ctrl1_out (BLOCKSIZE);
      valid <= "FOR(CNT, 0, BLOCKSIZE-1, {0})";
      valid (subblock) <= '1';
    end if;
  end process;

  process
  begin
    wait until ck'event and (ck = '1');

    subblock <= conv_integer (unsigned (ram_address (LOG(BLOCKSIZE)-1 downto 0)));
    ramsel_reg <= ram_sel;
    toggle_reg <= toggle;
    valid_reg <= valid;
  end process;
end func;

})
}, REPLACEMENT, LRU, {
ifelse(BLOCKSIZE, 1, {
-------------------------------------------------------------------------------
-- a 2-way set associative cache with a block size of 1:
-- (this implemetation uses a LRU replacement policy)
-------------------------------------------------------------------------------

-- This implementation uses one ram cell for each of both banks

architecture func of CACHENAME is
  signal lru : std_logic_vector (eval(CACHESIZE/2)-1 downto 0);
  -- the array containing the bits that specify the least recently used entries
  signal ram_address : std_logic_vector (OFFSETSIZE-1 downto 0);
  signal linenr : integer range 0 to eval(CACHESIZE/2)-1;
  signal d_in, d0_out, d1_out : std_logic_vector (eval(DATASIZE+TAGSIZE)-1 downto 0);
  signal tag_hit0, tag_hit1 : boolean;
  signal rwn0, rwn1, rwn0_reg, rwn1_reg : std_logic;
  signal ram_sel, ramsel_reg : std_logic;
  -- selects which bank to write

  for ram0, ram1: ramcell use entity work.ramcell (func);
begin

  ram0 : ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => eval(CACHESIZE/2),
      wordlength => eval(DATASIZE+TAGSIZE))
    port map (
      ck    => ck,
      addr  => ram_address,
      rwn   => rwn0,
      d_in  => d_in,
      d_out => d0_out);

  ram1 : ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => eval(CACHESIZE/2),
      wordlength => eval(DATASIZE+TAGSIZE))
    port map (
      ck    => ck,
```

```
      addr  => ram_address,
      rwn   => rwn1,
      d_in  => d_in,
      d_out => d1_out);

  d_in <= data_in & write_tag;
  ram_address <= write_offset when (write = '1') else
                 read_offset;
  rwn0 <= not (write and not ram_sel);
  rwn1 <= not (write and ram_sel);

  tag_hit0 <= (read_tag = d0_out (eval(TAGSIZE)-1 downto 0));
  tag_hit1 <= (read_tag = d1_out (eval(TAGSIZE)-1 downto 0));
  miss <= '0' when (tag_hit0 or tag_hit1) else
          '1';
  data_out <=
    d0_out (eval(DATASIZE+TAGSIZE)-1 downto eval(TAGSIZE))
      when (tag_hit0) else
    d1_out (eval(DATASIZE+TAGSIZE)-1 downto eval(TAGSIZE));

  process (write_enabled, tag_hit0, tag_hit1, ramsel_reg, lru, linenr)
  begin
    if (write_enabled = '0') then
    -- disabled, so nothing has changed
      ram_sel <= ramsel_reg;
    elsif (tag_hit0) then
    -- the tag of bank 0 matches, select it
      ram_sel <= '0';
    elsif (tag_hit1) then
    -- the tag of bank 1 matches, select it
      ram_sel <= '1';
    else
    -- select the least recently used bank
      ram_sel <= lru (linenr);
    end if;
  end process;

  process
  begin
    wait until ck'event and (ck = '1');

    linenr <= conv_integer (unsigned (ram_address));
    ramsel_reg <= ram_sel;
    rwn0_reg <= rwn0;
    rwn1_reg <= rwn1;

    if (rwn0_reg = '0') then
    -- bank 0 was written, and becomes the most recently used
      lru (linenr) <= '1';
    elsif (rwn1_reg = '0') then
    -- bank 1 was written, and becomes the most recently used
      lru (linenr) <= '0';
    elsif (read_enabled = '1') then
      if (tag_hit0) then
      -- bank 0 is being used
        lru (linenr) <= '1';
      elsif (tag_hit1) then
      -- bank 1 is being used
        lru (linenr) <= '0';
      end if;
    end if;
  end process;

end func;

},{
--------------------------------------------------------------------------------
-- a 2-way set associative cache with a block size > 1:
-- (this implemetation uses a LRU replacement policy)
--------------------------------------------------------------------------------

-- In this cache implementation, 4 ram cells are used. For both banks the are
-- a data, and a tag & control ram cell.

architecture func of CACHENAME is
  signal lru : std_logic_vector (eval((CACHESIZE/2)/BLOCKSIZE)-1 downto 0);
  -- array with bits that specify which entries are least recently used
  signal ram_address: std_logic_vector (OFFSETSIZE-1 downto 0);
  signal data0_out, data1_out : std_logic_vector (DATASIZE-1 downto 0);
  signal ctrl_in, ctrl0_out, ctrl1_out : std_logic_vector (eval(TAGSIZE+BLOCKSIZE)-1 downto 0);

  signal linenr : integer range 0 to eval((CACHESIZE/2)/BLOCKSIZE)-1;
  signal subblock : integer range 0 to BLOCKSIZE-1;

  signal tag_hit0, tag_hit1, hit0, hit1 : boolean;
  signal rwn0, rwn1, rwn0_reg, rwn1_reg : std_logic;
  signal valid, valid_reg : std_logic_vector (BLOCKSIZE-1 downto 0);
```

```
  -- specifies which parts of the cacheline will be valid after a write.
  signal ram_sel, ramsel_reg : std_logic;
  -- selects which bank should be written.

  for ctrl0_ram, ctrl1_ram: ramcell use entity work.ramcell (func);
  for data0_ram, data1_ram: ramcell use entity work.ramcell (func);
begin

  ctrl0_ram: ramcell
    generic map (
      addr_range => eval(OFFSETSIZE-LOG(BLOCKSIZE)),
      words      => eval((CACHESIZE/2)/BLOCKSIZE),
      wordlength => eval(TAGSIZE+BLOCKSIZE))
    port map (
      ck    => ck,
      addr  => ram_address (OFFSETSIZE-1 downto LOG(BLOCKSIZE)),
      rwn   => rwn0,
      d_in  => ctrl_in,
      d_out => ctrl0_out);

  ctrl1_ram: ramcell
    generic map (
      addr_range => eval(OFFSETSIZE-LOG(BLOCKSIZE)),
      words      => eval((CACHESIZE/2)/BLOCKSIZE),
      wordlength => eval(TAGSIZE+BLOCKSIZE))
    port map (
      ck    => ck,
      addr  => ram_address (OFFSETSIZE-1 downto LOG(BLOCKSIZE)),
      rwn   => rwn1,
      d_in  => ctrl_in,
      d_out => ctrl1_out);

  data0_ram: ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => eval(CACHESIZE/2),
      wordlength => DATASIZE)
    port map (
      ck    => ck,
      addr  => ram_address,
      rwn   => rwn0,
      d_in  => data_in,
      d_out => data0_out);

  data1_ram: ramcell
    generic map (
      addr_range => OFFSETSIZE,
      words      => eval(CACHESIZE/2),
      wordlength => DATASIZE)
    port map (
      ck    => ck,
      addr  => ram_address,
      rwn   => rwn1,
      d_in  => data_in,
      d_out => data1_out);

  ctrl_in <= write_tag & valid;
  ram_address <= write_offset when (write = '1') else
                 read_offset;
  rwn0 <= not (write and not ram_sel);
  rwn1 <= not (write and ram_sel);

  tag_hit0 <= (read_tag = ctrl0_out (eval(TAGSIZE+BLOCKSIZE)-1 downto BLOCKSIZE));
  tag_hit1 <= (read_tag = ctrl1_out (eval(TAGSIZE+BLOCKSIZE)-1 downto BLOCKSIZE));
  hit0 <= tag_hit0 and (ctrl0_out (subblock) = '1');
  hit1 <= tag_hit1 and (ctrl1_out (subblock) = '1');
  miss <= '0' when (hit0 or hit1) else
          '1';
  data_out <=
    data0_out
      when (tag_hit0) else
    data1_out;

  process (write_enabled, tag_hit0, tag_hit1, ramsel_reg, lru, linenr,
           valid_reg, ctrl0_out, ctrl1_out, subblock)
  begin
    if (write_enabled = '0') then
    -- cache disabled; keep all state
      ram_sel <= ramsel_reg;
      valid <= valid_reg;
    elsif (tag_hit0) then
    -- the tag of bank 0 matches; use this bank at a write
      ram_sel <= '0';
      valid <= ctrl0_out (BLOCKSIZE-1 downto 0);
      valid (subblock) <= '1';
    elsif (tag_hit1) then
    -- the tag of bank 1 matches; use this bank at a write
```

```
      ram_sel <= '1';
      valid <= ctrl1_out (BLOCKSIZE-1 downto 0);
      valid (subblock) <= '1';
    else
    -- select the least recently used bank with the lru-bits
      ram_sel <= lru (linenr);
      valid <= "FOR(CNT, 0, BLOCKSIZE-1, {0})";
      valid (subblock) <= '1';
    end if;
  end process;

  process
  begin
    wait until ck'event and (ck = '1');

    linenr <= conv_integer (unsigned (ram_address (OFFSETSIZE-1 downto LOG(BLOCKSIZE))));
    subblock <= conv_integer (unsigned (ram_address (LOG(BLOCKSIZE)-1 downto 0)));
    ramsel_reg <= ram_sel;
    valid_reg <= valid;
    rwn0_reg <= rwn0;
    rwn1_reg <= rwn1;

    if (rwn0_reg = '0') then
    -- bank 0 has just been written
      lru (linenr) <= '1';
    elsif (rwn1_reg = '0') then
    -- bank 1 has just been written
      lru (linenr) <= '0';
    elsif (read_enabled = '1') then
      if (hit0) then
      -- bank 0 is being used
        lru (linenr) <= '1';
      elsif (hit1) then
      -- bank 1 is being used
        lru (linenr) <= '0';
      end if;
    end if;

  end process;

end func;

})
})
```

The VHDL components package

F

If a VHDL description of one unit uses other units, the interfaces of the subunits must be known to the main unit. The normal way to do this, is to copy the interface specification from the entity declaration of the subunit into the architecture specification of the top level unit[1]. The entity declarations must appears as component specifications in the header of the architecture specification. Apart from this declaration, the architecture body must also contain the statement which calls the subunit. The component specifications therefore hardly give any information, while they can easily make the code less clear.

Fortunately, it is also possible to collect the necessary component specifications in a VHDL package, which can be used by all units which need it. The tool which is described in this appendix automatically searches the VHDL source code for entity declarations, and converts them into components specifications which are put in a package. The procedure is actually quite simple, but we did not succeed in specifying the correct commands for an command-line editor, like sed. Therefore this tool, *vhdl_components*, was developed.

An entity declaration has the format:

```
entity name is ... end name;
```

The *vhdl_components* tool converts this into an component specification:

```
component name ... end component;
```

[1]A VHDL unit consists of a entity specification and an architecture specification. The entity specification defines all input and output signals, and the architecture specification contains the function body.

All component specifications in a source file are collected in a package with the name *components*. To use the components package, the entity declaration of the unit which uses other components must be proceeded by the statement: `use work.components.all;`

The *vhdl_components* tool uses *stdin* as its input, and *stdout* as output. The (unix) command: `vhdl_components < input.vhdl > output.vhdl` has therefore the desired effect. The input file must contain a valid VHDL description (no m4 parameters). The output file will contain the components package.