Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Diplomarbeit

Formalising PC Hardware: A Model of the x86 Architecture

Sarah Hoffmann

17. September 2003

Betreuender Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuender Mitarbeiter:	Dr. Michael Hohmuth

Erklärung

Hiermit erkläre ich, die vorliegende Diplomarbeit eigenhändig und selbständig verfasst und keine als die angegebenen Hilfsmittel und Quellen verwendet zu haben.

Dresden, 17. September 2003

Sarah Hoffmann

Contents

List of Figures

All quotations taken from *The Marriage of Heaven and Hell, Proverbs from Hell* by William Blake [Bla94].

Chapter 1

Introduction

8. A fool does not see the same tree
a wise man sees.

Although the awareness of security issues has risen lately trust in computer software is largely based on faith and the common misconception that computers are infallible. Taking a look at the bug report list of an arbitrary software project can prove the contrary. Verification exists, if at all, only for small applications. They in turn rely on the correctness of the underlying operating system. Still, the most widespread operating systems work on the principle of hope for the best—or rather the lack of the worst. The VFiasco project [HTS02] at TU Dresden tries to fill that gap by verifying the Fiasco microkernel.

The verification of a monolithic operating systems is unrealistic because of their enormous size. Modern verification techniques utilise theorem provers to partly automate the verification process but they are still unusable for software beyond 50.000 lines of code. Linux is said to have more than 2 million lines of code [GT00]. Microkernels offer a solution to this problem. They aim at reducing the functionality of the operating system kernel by shifting as much work as possible to user applications.

L4 [Lie96] is a family of microkernels of the second generation. Its kernels implement only minimal functionality: address spaces (tasks), execution entities (threads) and inter-process communication (IPC). Management of hardware resources like physical memory or devices is left to user applications. The kernel solely provides the means to restrict access to these resources. It initially gives the rights to use them to one user task, which is free to use and distribute those rights.

Besides running normal applications it is even possible to execute a full monolithic operating system on top of such a kernel as the port of Linux to L4 [Hoh96] has shown. If the kernel can ensure that the applications are fully isolated security critical applications can run next to a standard production system and still give guarantees about themselves.

For the L4 microkernel such *security properties* include that address spaces are independent, i.e., that one task cannot spy out data from another, that a task cannot gain access to parts of the hardware it does not have the rights to and that user-space tasks cannot do damage to the kernel or other tasks.

The Fiasco [Hoh98] kernel is one of the members of the L4 family. It was developed at TU Dresden as a kernel with full real-time capabilities. Originally designed for IA32 processors, it has been successfully ported to other architectures. It is written almost entirely in C++ utilising assembler code when direct hardware access is needed. With currently about 30.000 lines of code for its core functionality[1] it has a reasonable size for verification.

There is another obstacle for the verification of an operating system. It is impossible to verify it without regarding the hardware it runs on. After all, its main task is to manage this

[1]Fiasco features a lot of useful extensions like a fully grown kernel debugger. As these extensions would not be included in a working system they can safely be disregarded for verification.

hardware. PCs undeniably constitute the most widespread hardware today. They are by no means less complex than the operating systems that run on top of it. Until now, verification projects tried to get around this problem by using either special hardware [K$^+$03] or idealised models that have no counterparts in real processors [Bev88]. Both solutions are not applicable to VFiasco because its goal is to verify Fiasco in the context of a real existing execution environment.

This thesis will show that it is possible to develop a model of general purpose hardware that is simple enough to be usable in software verification but still constitutes a model of existing processors. We will do so by abstracting the processor to its functional model and then reducing it further to those parts that are required by an operating system.

1.1 Structural Outline

This thesis aims at the development of an x86 hardware model that is suitable for the verification of the Fiasco microkernel. It is organised as follows: the next chapter introduces the VFiasco project and the task of verifying a microkernel. Chapter 3 gives an overview over the x86 architecture. It explores which parts are necessary for a verification model and which can be omitted. We continue this examination in more detail in Chapter 4 where we assemble the findings to the hardware model. Chapter 5 addresses a number of problems that arise when integrating the model into the VFiasco project. Chapter 6 explains how the model can be challenged to prove its correctness. The thesis concludes with a short summary and an overview of future work.

1.2 Acknowledgements

I would like to thank Prof. Härtig and the people of the TU Dresden operating systems group for the great opportunity to work there. In particular, I wish to thank Michael Hohmuth and Hendrik Tews for supervising me and introducing me to the deeper mysteries of software verification. A special kind of gratitude I owe to my brother Martin who always had the time and patience to listen to my lengthly explanations until they became fully clear to me.

There are so many other people whose technical and moral support made this work possible, among them Matthias Daum, Udo Steinberg, and Alexander Warg. Those I cannot name here are not forgotten.

Finally, I wish to thank my parents who supported my studies and always endorsed my ideas and plans, however unusual they seemed to get.

Chapter 2

VFiasco

33. What is now proved was once only imagin'd.

The verification of an operating system kernel, even if it is a microkernel, is not simply done by taking the kernel and putting it appropriately into a verification system. The kernel has to be translated into a specification language and the execution environment has to be defined and modelled.

This chapter first introduces PVS, the verification system used in the VFiasco project. Afterwards it describes shortly the steps taken to verify the Fiasco microkernel.

2.1 PVS

PVS is a verification system that contains a specification language and a semi-automated theorem prover. The PVS language features a higher-order logic with predicate subtypes. In this thesis we are going to utilise the PVS syntax at times to give definitions of types and functions. Below follows a simplified overview of those language constructs of PVS that are needed to understand these definitions. For a full introduction the reader is referred to the PVS manuals [OSRSC01a, OSRSC01b].

The PVS language is strongly typed. Next to the base types like `nat`, `int` or `bool` it knows the following complex types:

- Records collect a number of types. They are of the form

  ```
  [# a_1 : t_1, a_2 : t_2, ... #]
  ```

- Functions employ the usual mathematical notation. They can be given either in form of a definition or in form of a lambda calculus. For example, the `min` function over an unspecified data type `T` can be written as one of

  ```
  min(d1, d2 : T) : T = if d1 < d2 then d1 else d2 endif

  min : [ T, T -> T ] =
     Lambda(d1, d2 : T) : if d1 < d2 then d1 else d2 endif
  ```

  Functions are evaluated by their signatures, so it is possible to overload function names.

- Abstract data types (ADTs) are a very powerful type. In this thesis we will only use ADTs that are not recursive. They can be viewed as an extended form of an enumeration that allows each element to have a number of parameters. A general definition looks like the following:

  ```
  example_adt : Datatype
  Begin
    cons1(ac11 : type11, ac12 : type12) : rec1?
    cons2(ac21 : type21, ac22 : type22) : rec2?
  End adt
  ```

  PVS composes various functions and theorems for each ADT. A constructor function creates an element of the ADT. It has the same name as the element and takes the given parameters. One of the constructors of `example_adt` above is

  ```
  cons2(ac21 : type21, ac22 : type22) : example_adt
  ```

  A recogniser functions checks if a value is of a given element.[1]

  ```
  rec1?( the_adt : example_adt) : bool
  ```

  yields true iff `the_adt` has been assigned `cons1`. Finally accessor functions, which use the parameter names of the definition, extract the value of a parameter.

A number of type definitions together with associated theorems and axioms form a *theory.* A collection of potentially dependent theories forms a specification. Theories can be generalised over constants and types. The generalisation parameters are then given in square brackets after the theory name. Before such a theory can be used it has to be instantiated, either when importing the theory or when using a function or type of that theory. Again, the actual parameters have to be provided in square brackets.

2.2 Verifying Fiasco

The VFiasco project plans to prove security properties from the Fiasco source code. Figure 2.1 depicts the steps necessary. The verification on C++ level has been chosen to take advantage of the structure of the source code. It allows to split the verification into smaller steps. Properties can be established over single C++ functions first and later be combined to prove the security properties of the whole kernel.

PVS does not understand the C++ syntax, hence, the source code has to be translated into the PVS language first. This is done automatically by the *semantics compiler*. It compiles the semantics of the source code into a PVS specification. The base of this semantics is the C++ model. It provides a model of the C++ compiler used to compile Fiasco, or to be more specific, a model of the interpretation the compiler applies to C++ code. For a normal application this would complete the model for a source code verification if it can be proved that the compiler produces correct code. However, the purpose of an operating system is to control the underlying hardware. Therefore, the semantics of its source code cannot be complete without a model of this hardware. Together, source code specification, C++ model, and hardware model define the behaviour of the kernel. This has to be verified against the

[1]Note that the question mark is not mandatory at the end of the recogniser name. Using it there is just a convention. The question mark constitutes a legal identifier character in PVS. Thus, `rec1` an `rec1?` are different identifiers.

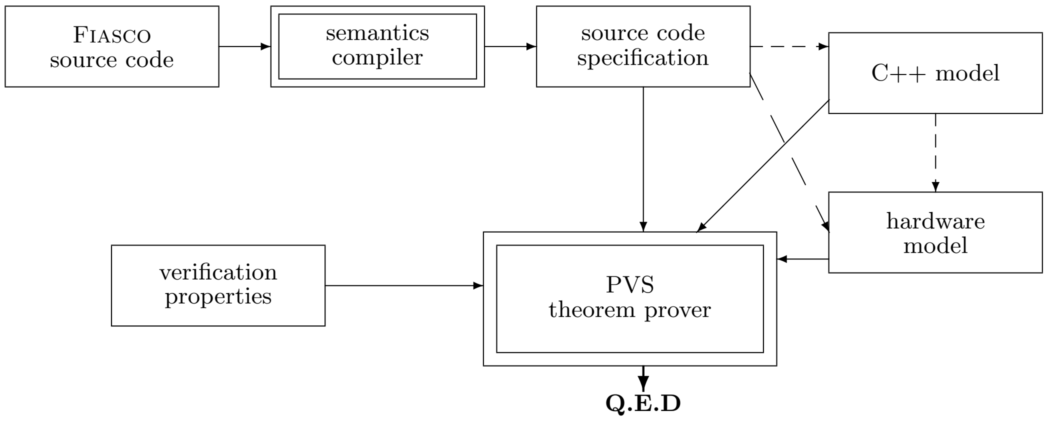


Figure 2.1: Verifying Fiasco

security properties we want to prove. In order to do so they have to be formulated in PVS first.

We will have a closer look at the semantics compiler and the C++ model before defining the function of the hardware model more precisely in this context.

2.2.1 Semantics Compiler

In its essence source code constitutes a sequence of operations that change the state of the underlying system—hopefully in a meaningful way. The semantics compiler utilises this view and translates the source code into a sequence of *state transformers.* They are functions that alter the system state: $State \rightarrow State$. The semantics compiler cannot know what the system state comprises. Therefore, it uses it as an unspecified type and leaves its exact type definition to the hardware model. In spite of that, the view of a state transforming system forms the foundation of the verification process. State transformers will also form the basic function type in the hardware model. That is why they are examined more closely in the following. A full description of them and the semantics compiler can be found in [Dau03].

A simple state transformer as defined above is not particularly powerful. When a state transformer represents a C++ expression it does not only yield the subsequent state but also a result. To reflect that, the co-domain of the state-transformer type has to be extended:

$$State \rightarrow (State, Data)$$

The type of `Data` depends on the concrete state transformer. Some state transformers, like those of statements, do not return any data. They simply use the data type `Unit`, which just contains one element `unit`. It signifies that the value is not of interest.

Furthermore, a state transformer can terminate abnormally. A reason might be a programming error like trying to read from uninitialised memory. So the co-domain has to be extended once more. Here, an abstract data type is suitable where each element models another termination state:

```
Result[State, Data] : Datatype
Begin
  OK(next_state : State, get_data : Data) : OK?
  Fatal : Fatal?
End Result
```

`OK` stands for a normal termination and includes the subsequent state and a return type. `Fatal` denotes an unrecoverable error. It does not need a subsequent state because the verification fails at this point. There are other abnormal termination states. Those that are of interest for the hardware model will be introduced in the appropriate sections.

Using the `Result` ADT we arrive at the definition of a standard VFiasco state-transformer type:

```
State_transformer[State, Data] : Type = [ State -> Result[State, Data]]
```

Simple state transformers can be connected to become complex ones that form language constructs like loops and functions. The most important connection functions are those that join two state transformers sequentially: the infix operator `##` and the function `eval_if_ok`. `##` is defined as follows:

$$\begin{aligned} \texttt{\#\#} \quad &: \quad \textit{State\_transformer} \times \textit{State\_transformer} \rightarrow \textit{State\_transformer} \\ & \quad (t_1, t_2) \mapsto \lambda(s : \textit{State}) : \begin{cases} t_2(\textit{next\_state}(t_1(s))) & \text{if } \textit{OK?}(t_1(s)) \\ t_1(s) & \text{otherwise} \end{cases} \end{aligned}$$

The definition of `eval_if_ok` reads similar. The difference is that it additionally allows to evaluate the return data.

2.2.2 C++ Model

The C++ model has to express the exact semantics that the C++ compiler assigns to the source code. We do not care so much for the object code representation the compiler produces but more for the semantics this object code gets on execution. One could say the C++ model constitutes the semantics of a virtual machine the C++ source code is directly executed on. This semantics suffices to verify a simple application program if we assume that the compiler works correctly. It is still inadequate for an operating system because system software regularly bypasses the C++ semantics and directly accesses the underlying hardware. Naturally, the semantics for such operations is undefined in C++ so they require a model of the hardware itself. Another problem is that every compiler makes assumptions about the state of the hardware. For example, the compiler expects that it can use a portion of memory exclusively. Circumventing the C++ semantics might hurt these assumptions. To still be able to use the C++ model we have to prove that its assumptions hold in spite of the hardware manipulation by the operating system. We can only do this if the C++ model builds its semantics on top of the hardware model as well.

The definition of the C++ model is currently a work in progress, its usage of the hardware model for the most part undefined. At the time being only the concept of a memory free of side-effects and a model for data types exist. Both are introduced shortly in the following two sections. A more detailed discussion can be found in [HT03].

Plain Memory

The C++ model defines memory by its state and exactly two functions: `read` and `write`. Consequently, a memory interface is a record of two state transformers:

```
Memory_struct[State] : Type =
  [#
    memory_read  : [Address -> [State -> Result[State, Byte]]],
    memory_write : [Address, Byte -> [State -> Result[State, Unit]]]
  #]
```

It depends on two more types: *Byte,* the smallest addressable unit in memory, and *Address,* the scope of bytes the memory offers.

Such a memory interface can specify an arbitrary memory. We already stated that the C++ model needs to have a part of the memory for its own usage. These regions need to form a memory free of side-effects, a *plain memory.* The C++ model expects the following properties to hold for those regions:

- Reading and writing a byte must be successful for each address in the region.
- Writing a byte to an address in the region and reading afterwards from the same address must retrieve exactly the same value.
- Writing a byte to one address must leave the rest of the memory in the given range untouched.

Data Types

The memory interface allows to read and write single bytes only. Few of the C++ data types are of that size. Nevertheless, they are used in the C++ model as base types. To load them from and store them to memory the C++ model defines a serialisation interface for each data type `Data`:[2]

```
Data_type_structure[Data : Type] : Type = [#
  size      : nat,
  to_byte   : [Data -> list[Byte]],
  from_byte : [list[Byte] -> lift[Data]]
#]
```

A data type always uses a fixed number of bytes. The function `to_byte` creates from the given value a byte list that can be stored as a sequence in memory. Accordingly, `from_byte` does the reverse with a byte list that has been read from memory. To become a C++ data type the interface has to fulfil the property that these two functions are bijective.

2.3 Hardware Model

Semantics compiler and C++ model have left the system state undefined. It is the main task of the hardware model to define its type and give it a semantics. The semantics compiler further expects a definition of those functions that are not part of the C++ specification. They can be identified by carefully checking the Fiasco source code [Dau02]. The requirements of the C++ model are currently confined to the plain memory interface. In any case, neither the semantics compiler nor the C++ model change the system state directly. They only use the state transformers the hardware model will provide. So they require only a functional semantics of the hardware. That is what makes a first simplification of the hardware model possible: VFiasco does not need a model that exactly reflects real processors down to the level of gates. Instead, it suffices to model the virtual processor the hardware is supposed to implement. The disadvantage of such an abstraction is that it would require to deliver a formal proof that the model correctly implements the real hardware Fiasco is supposed to run on. The purpose of this thesis is to define a functional model. It cannot give a proof of its correctness in context of real hardware. Indeed such a proof would be a major project on its own.

The virtual processor model of the x86 architecture is defined in the *Intel IA-32 Architecture Manual* ([Int99], IA32 manual for short). On its basis we will develop a formal model of the architecture. The IA32 manual is mostly informal, so we will be confronted with interpretation problems that emerge when going from an informal to a formal specification.

[2]The `lift` type is used to define partial functions. It is declared as: $\mathit{lift}[Data] = \bot \uplus Data$.

The hardware model needs to define the system state and the state transformers of the functions that Fiasco and the C++ model use. This can happen in one of two ways: axiomatic or definitional. An axiomatic specification only defines the state and function types, i.e., the signature, and provides properties of them in the form of axioms. A definitional specification, on the other hand, provides an exact definition of the system state and each function. The advantage of an axiomatic model is that it only gives a semantics to known facts. A definitional model runs the risk to define results that are not specified and thus make the entire verification void. Still, we are going to realise a functional model out of three reasons: First of all, errors in axiomatic definitions lead to inconsistencies that allow to conclude anything from the specification. Second, the architecture is for the most part well defined. And finally, a definitional model is much closer to the specification. It defines an exact state type that includes those system structures the IA32 manual describes. The semantics of instructions is even given in an informal version of VHDL and thus can easily be compared to the function definitions of the model. This way, we can hopefully strengthen the confidence in the correctness of the model even without a formal proof.

2.4 Notation

In the remainder of this thesis we will occasionally refer to specific parts of the IA32 manual. It is published in three volumes, so when citing a particular section we state the volume number in Roman numerals followed by the section[3] in Arabic numerals. Thus, the section about segment descriptors (III–3.4.3) can be found in the third volume of the IA32 manual in section 3.4.3.

2.5 Related Work

There are very view projects in software verification that try to include the hardware. The usual approach is to assume the existence of a virtual machine. One such project is the LOOP project [vdBJ01], which aims at the verification of JAVA code on base of a JAVA virtual machine.

The idea of a functional hardware model has been used in a similar way in the Kit study [Bev88]. However, the model was restricted to a very simple von-Neumann machine, which had no counterpart in a real processor. Furthermore, the model comprised a state machine on which object code was executed directly.

The project most closely related to VFiasco is the VeriOS project of the Saarland University [K$^+$03]. Its goal is the verification of an L4 kernel running on their own verified microprocessor VAMP [BJK$^+$03]. The processor is modelled down to the gate level thus using a different level of abstraction than this thesis aims at.

[3] Section numbering changes slightly between the various versions of the IA32 manual. The numbers in this thesis refer to the version cited in the bibliography.

Chapter 3

Hardware Requirements of Fiasco

46. You never know what is enough unless you know what is more than enough.

Since its birth in 1978 the x86 architecture has grown considerably. The original processor started out as a simple 16-bit CPU. Its successors then introduced numerous new features—fundamental ones like protected mode and paging as well as advanced units like MMX. In spite of all progress each new generation remained compatible with its predecessors, thus creating a complex and tangled architecture.

This chapter is meant to give an overview of the x86 architecture. It is going to single out the features that are required for the VFiasco hardware model. To be able to do so the criteria for such a selection have to be defined first.

3.1 Towards a Model

In the last chapter we have already state that the high abstraction level of the functional processor specification can considerably simplify the model. Still, the x86 architecture specification is highly complex when modelled in its entirety. Fortunately, completeness is not needed and even may not be wished for. Every feature in the model introduces additional proof obligations during the verification. Thus, excluding unused features can greatly reduce the amount of required proofs. Of course, we have to examine carefully that we do not undermine the correctness of the hardware model by omitting various parts.

To determine to what extend we have to model the architecture we first have to observe which of its functions, i.e., machine code instructions, are used. From this we can conclude which architectural parts must be modelled.

It is safe to leave out complete hardware functions that the kernel is not supposed to utilise. If the kernel uses them nevertheless the verification already fails with the semantics compiler because it is not able to translate an instruction that has no counterpart in the state transformers of the hardware model. One such function is the `UD2` instruction, which solely causes an invalid opcode exception.

If the semantics of a function is provided it must be modelled completely. The same is true for all parts of the architecture that are influenced by the function. This way we ensure that the kernel does not use them in an illegal way. However, it is possible to neglect the dynamic model of such a feature and make a static assumption about its usage—or non-usage. Verification then has to be rendered fatal if the function is used in another way. For example, many such static assumptions are made when writing to one of the control registers. Here, additional processor features like the debugging extensions can be switched on and off. Most of them are not needed. So, we just make the static assumption that they are switched off in the processor and forbid to switch them on when writing to the control registers. We call features *unsupported* if they are omitted either by not providing a functional interface or by making a static assumption about them.

Just observing the influence of a feature on the functional interface is not enough. For example, consider the translation look-aside buffer, which we will examine in detail in Section 4.2.3. Even if FIASCO would not use its interface we cannot simply omit it. It still has an influence on the correctness of linear address translation. So, a feature must also be modelled if it has an influence on another feature that is already included in the model. We trust that such dependencies only exist if they are explicitly stated in the IA32 manual.

When modelling a feature we normally expect its usage to be *mandatory.* Any attempt to circumvent it leads to a fatal result. However, some of the hardware functions FIASCO uses do not influence the overall correctness of the verification, either because the C++ model hides their semantics or because they only effect the internal state of the processor without being visible at the functional interface. These features are just *carried along.* We only need to provide the interface functions for them. Their internal functionality is not necessary and their state is required only up to the point that the semantics of the interface is correct. The most prominent examples of this kind of feature are memory caches. On a single processor system they do not influence the result of a memory access but only the time necessary to execute them. Still, the kernel should be allowed to set registers that influence the caching strategy and read the correct value afterwards. So, we only have to add a variable to the state that saves the current caching strategy.

Finally there are features that have been introduced in later processor generations. They cannot become mandatory because then the model is not a model of earlier processors anymore. Those *processor parameters* will become parameters over the model. By doing so one can choose during verification between model instantiations where the feature is either mandatory or unsupported.

The main objective for the selection of processor features are the demands of the FIASCO hardware interface and the C++ model. The extent of the hardware usage of FIASCO mainly depends on the properties that are to be verified. They fall roughly into three categories:

Correctness of kernel code. Those properties argue over the outcome of particular kernel functions. One such property states that the page-fault handler always succeeds for certain memory regions. The necessary hardware functions are relatively few. They are well defined by the source code of FIASCO.

Security of user-mode applications. This category comprises properties that guarantee that access restrictions are respected by user applications and that tasks are isolated from each other. To verify them the model needs the ability to switch between privilege levels. Further, it has to be extended by all those parts of the architecture that can be influenced by user tasks.

Correct bootstrapping. Every operating system kernel makes some assumptions about the normal operating mode of the processor. They do not necessarily match the state the processor is in after reboot. Therefore, a final verification goal is to prove that the bootstrapping procedure of the kernel sets up the processor correctly.

As already mentioned in 2.3, at the time being the C++ model requires only a memory interface, that is functions to read and write virtual memory.

3.2 Architecture Survey

Now we can classify the x86 architecture according to the verification needs. This section functions as a general overview of the architecture. During the development of the model specification in the next chapter we explore the parts to be modelled in more detail.

Much of the architecture of the two 16-bit processors 8086 and 80286 has become obsolete. It is nowadays included only to assure backward compatibility. As a general rule, such parts will be unsupported because FIASCO is a 32-bit kernel. To further emphasise that we are going to refer to the architecture from now on as *IA32*, short for Intel architecture 32 bit.

3.2.1 Modes of Operation

IA32 supports three operating modes: real-address mode, system management mode, and protected mode. *Real-address mode* is an emulation mode for the old 8086 processor and as such not of interest for newer software. There is one exception for operating systems: after power up or a reset the processor always finds itself in this mode. Therefore, it must be modelled when proving bootstrapping.

System management mode (SMM) provides means to handle power management and to control system hardware. It works transparently to any software including the operating system. The processor enters this mode when it receives a system management interrupt (SMI). SMM operates in its own execution environment and is supposed to restore the complete state of the processor on return to the mode it was called from. For this model we assume that it works indeed transparently and can therefore be ignored. FIASCO does not support power management. If it chooses to do so in a future version the SMM handler has to be proved in another model.

The standard operating mode is *protected mode.* It has a submode called *virtual-8086 mode*, which allows emulation of 8086 within protected mode. It is legacy as well. For kernel-code verification only the normal protected mode is required.

3.2.2 Memory Management

Memory management provides the foundation for the memory abstraction of the C++ model. Therefore, it is a mandatory part of the hardware model. It consists of physical memory and two independent memory management facilities.

First an address is interpreted in terms of *segmentation.* It allows to partition the memory into smaller regions. Most modern operating systems work around segmentation by providing segments that span over the entire memory. In spite of that, it is impossible to disregard segmentation because it provides the privilege system of the processor and is used during interrupt handling.

In a second stage the linear address calculated by the segmentation unit is translated by *paging*, which implements virtual memory with two levels of page tables.

To accelerate memory access there are various stages of caches for physical memory. They normally work transparently to software. In a single processor environment they are guaranteed to always be consistent with physical memory (III–9.1). On multi-processor machines consistency requires intervention by software. There is a number of functions to control the caching strategies. They only influence performance, which is not of interest for this hardware model. Even if FIASCO currently does not control caching it is highly possible that it will do so in the future. For this reason caches need to be carried along.

3.2.3 Arithmetic Operations

Arithmetic operations are those functions that alter their operands: binary and decimal arithmetic, logic, shift, rotate, bit and byte instructions. Direct manipulation of the hardware never influences their results, so they can all be hidden in the C++ semantics.

Later processor generations introduced additional arithmetic units to enhance performance for multi-media operations. In particular, there is the floating point unit (FPU) and the SIMD units MMX, SSE, SSE2 and 3dnow. FIASCO needs none of these units but user-mode applications are free to use them. As with general-purpose registers, the kernel has to save and restore their state correctly when switching between tasks. In Fiasco this is done lazily. On a task-switch the kernel sets a flag to indicate that an exception should be raised as soon as the FPU is used. The FPU state is saved only if such an exception occurs. Therefore, we can safely leave it to user-mode verification to model the state of these units. For kernel mode verification we just have to provide the ability to enable the FPU exception.

3.2.4 Execution of code

When the processor executes code it successively reads instructions from memory, decodes and executes them. Additionally, control transfer instructions can be used to change which instruction is read next. We need to trust that the C++ model provides a correct mapping between C++ source and machine code instructions. Otherwise, we would end up proving Fiasco on object code level. We have to assume further that code execution works transparently to any other functionality the model provides. This requires three conditions to hold: (1) Reading of code from memory must not change the system state. (2) The sequence of instructions in memory must match the state transformers to be executed. (3) All control transfers must happen within the C++ model so that the order in which the instructions are executed is the same as that of the state transformers. The first two assumptions can become proof obligations during the verification. The third one is more problematic. Next to the normal control transfer instructions, jumps and loops, there are some that the C++ model does not know about: call gates, far jumps, interrupts and exceptions. To be able to use them they need a semantics that only the hardware model can define. However, such a semantics is confined to what the processor does during the control transfer. To reestablish the third assumption we need a model that integrates this hardware control transfer into the context of the C++ model. Altogether, a model of code execution requires a closer connection between C++ model and hardware level. This problem will be addressed on its own in Chapter 5.

Fiasco code verification needs a model for page fault handling because it pages kernel data on demand. Other exceptions are not expected to be raised. Conditions that cause such an exception can always be considered a programming error. To be interruptible Fiasco further needs a model of timer interrupts. For user-mode verification all exceptions have to be modelled because we have to prove that the kernel can handle an arbitrary misuse of the processor by user-mode applications.

The IA32 architecture provides multitasking support in hardware. In order to use it the kernel needs to allocate a task state structure (TSS) for each task[1]. The hardware then automatically saves and restores the state of the processor when switching between tasks. Apart from the current processor state the TSS holds some static information for the execution of the task: the stack pointer to be used when changing privilege levels and the I/O bitmap, which restricts access to I/O ports. Fiasco does not use hardware multitasking because it turned out to be by far slower then manually saving and restoring the processor state. Thus, task switching itself can remain unsupported. Access to the I/O bitmap and the system stack pointers needs to be added for user-mode verification. Setting up the TSS is part of bootstrapping.

The processor offers many features for debugging: break points, debugging registers, stepwise execution, performance measurement, and others. All these are used for kernel development and are not of interest for a kernel in production use. In the model they are unsupported.

[1]At this point the terminology of Intel and L4 collide with each other. Intel calls an execution entity a task. L4 uses the same term for an address space. Intel‘s task concept maps to an L4 thread. This paper will follow the Intel notation. When speaking of task and threads in the L4 sense we refer explicitly to them as L4 tasks and L4 threads.

3.2.5 General-Purpose and Control Registers

The general-purpose registers and the flag register are part of the task state. They are used exclusively by the C++ model except for some flags in the flag register which we will explore in Section 4.2.6. All registers need to be modelled nonetheless, because the kernel saves and restores them on a task switch. The ESP register deserves special treatment. Although it belongs to the general-purpose registers it has the dedicated use of stack pointer. In this function it is interpreted and changed by the hardware.

The control registers CR0 and CR4 configure the operating mode of the processor in more detail. Most of the flags they contain can be considered architectural features that do not need to be changed at run time. Notable exceptions concern the page-global flag (for TLB, see Section 4.2.2) in the CR4 register and the task switch flag (to enable FPU exceptions, see Section 3.2.3) in CR0. Because of them the two registers have to be modelled as well.

3.2.6 Devices

I/O ports allow communication with devices. They can either be mapped into memory or be accessed directly. The processor provides two kinds of access control: the general right to do I/O communication can be changed by setting the I/O privilege level (IOPL) and access to single ports can be restricted in the I/O bitmap. As mentioned before a microkernel does not need to access devices. It suffices to model access control, namely IOPL and the I/O bitmap. For user-mode verification the access functions have to be added.

Devices may also directly transfer data to and from memory by using direct memory access (DMA). The usage of this mechanism is a general problem to secure operating systems because it subverts the protection mechanisms of the CPU and can do arbitrary damage [HLM$^+$03]. Using it in a verified microkernel is currently out of the question.

Table 3.1 summarises the usage of the IA32 architecture for the verification of Fiasco. For further reference you may consult the appendices A and B. They list and classify all architectural features as well as the x86 instruction set in detail.

The functional components we identified in the course of this chapter can be stripped further of unnecessary subparts. We will do so in the next chapter when developing the hardware model itself.

Verification of	**Kernel Code**	**User-Mode**	**Bootstrapping**
mandatory	protected mode physical memory paging with TLB segmentation stack interrupt and exception handling	privilege-level change task-state segment	real-address mode
carried along	memory caches general-purpose registers EFLAGS control registers	state of FPU, MMX, SSE, SSE2, 3dnow	
unsupported	system management mode virtual-8086 mode hardware multitasking debugging		
hidden by C++ model	arithmetic operations code execution		

Table 3.1: Usage of x86 architecture in the VFiasco hardware model

Chapter 4

Model Specification

56. To create a little flower is a labour of ages.

Based on the requirements definition of the last chapter we are now going to develop a model of the IA32 hardware. The chapter starts with a short overview of the general design of the model and then explores each part in detail.

4.1 General Design

This first realisation of the VFiasco hardware model will concentrate on the development of a model that allows the verification of the code of a minimally configured Fiasco kernel. It will be designed in a way that allows to add other aspects of the IA32 architecture gradually.

4.1.1 Layers

Figure 4.1 gives an overview of those parts of the architecture the model has to cover at least: memory, the address translation mechanisms, interrupt and exception handling, registers that belong to the task state, and the control registers. They can be divided into the following functional units:

RAM includes state and interface of physical memory to be used by other units.

Linear memory adds the paging mechanism, reads and interprets page tables, and knows about page faults.

Translation look-aside buffer (TLB) specifies the cache for linear address translation. Although it is part of linear memory it is consider on its own because of its complexity.

Segmented memory implements segment address translation, the public memory interface, and handles the global descriptor table (GDT).

Stack supplies the means to use the stack memory region.

Registers comprise those parts of the hardware that are touched directly by Fiasco but are not used by the hardware model itself.

Interrupt and exception handling provides means to enter and leave an interrupt handler and reads and interprets the interrupt descriptor table (IDT).

These units gradually depend on each other. To allow a minimal modularisation we use a layering model where each of the unit is modelled separately but builds on top of each other. When implemented every layer gets its own collection of theories. The main theory contains the model specification. It consists of the definition of a state type and a collection of functions to change the state.

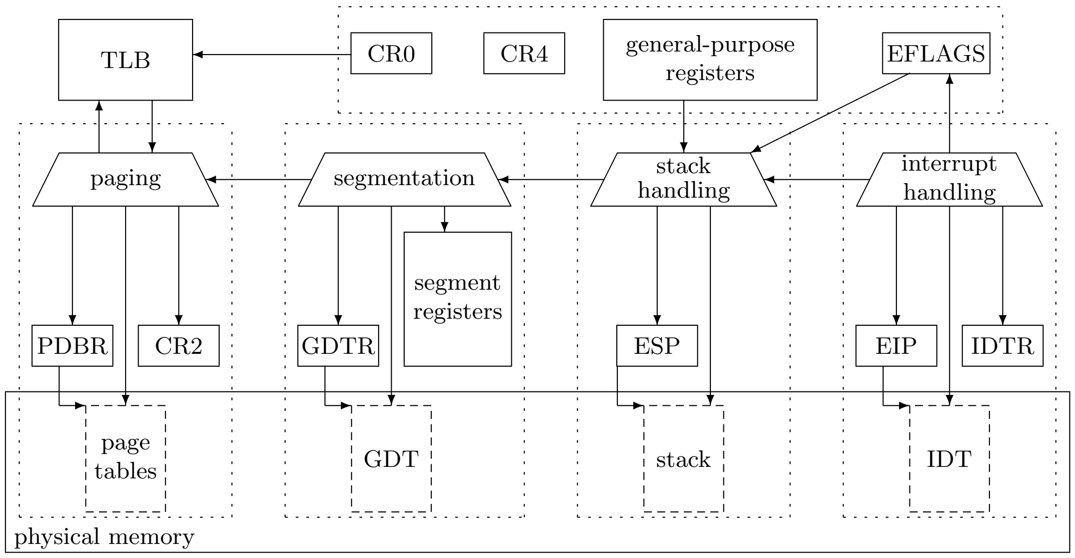


Figure 4.1: Overview over the model

4.1.2 Hardware State

The state of each layer is composed of the states of the system structures belonging to that layer. They are best collected in a record. Each layer extends the state of the underlying layer. PVS does not support the extension of record types, so we model this by defining its own state types for each layer and then add the state of the underlying layer. This way the uppermost layer determines the overall dynamic state of the hardware.

System structures that are stored in memory do not get their own state. Instead, they are always read directly from memory. Higher layers can use the memory interfaces of lower layers. The GDT, for example, is referenced by an linear address. Hence, the read function of linear memory is used to retrieve data from it.

For processor parameters the modularisation is not necessary because there are only few. Instead, they can easily be collected in a global record and passed to every theory as a theory parameter.

In Section 3.1 we established the condition that features can only be unsupported if it can be verified that they do not have side effects on modelled features. This model makes an exception to this rule by introducing some *unverifiable* features. The system structures stored in memory normally contain information the system software provides to the processor. As such the hardware model only needs to read them. However, the processor also modifies certain bits to provide feedback about the usage of these structures to the kernel. The current implementation of VFiasco does not make use of this information, so the model will not reflect these modifications. This way, the hardware model itself leaves physical memory untouched. That simplifies proofs over the model considerably. You should keep in mind, though, that it is impossible to formally prove within the hardware model that those bits are not used. They are just a part of memory. Once they are retrieved they are interpreted by higher levels of the verification model, outside of the control of the hardware model.

4.1.3 Functional Interface

In the processor the set of machine instructions defines the functional interface. Because IA32 is not a load-store architecture the execution of a single instruction normally includes several steps: fetching its operands, executing the appropriate operation on them and storing the result. In contrast to that, the C++ model strictly separates these three steps. It defines individual functions for reading a variable, for writing a result, and for the operation itself. The hardware model will follow the view of the C++ model and define the read and write operations separately. Those instructions that expect a very specific kind of operator prove the exception. An example is the `LGDT` instruction. It expects a pointer into virtual memory and reads its data directly from there.

Separating machine code instructions leads to problems when the processor uses instruction boundaries to define its atomic operations. For example, it guarantees that interrupts never occur during the execution of an instruction. We will meet this problem again in Chapter 5 when discussing interrupt and exception handlers.

4.1.4 Data Types

The functional interface of the IA32 defines three data types: a 32-bit word (`Mword`), a 16-bit word (`Word`) and an 8-bit byte (`Byte`). They can be interpreted twofold: as a number, for example, when referring to an address, or as a vector of bits as in the EFLAGS register. The hardware has no difficulty in changing between these interpretations because internally all values are represented as bits. In PVS this cannot be done that easily. Numbers and bit vectors are fundamentally different data types. The former is a basic type while the latter is represented as a function:[1]

```
bvec[number_of_bits] : Type = [below(number_of_bits) -> bit]
```

Conversions between those types are not trivial. Although PVS offers an impressive library to reason over bit vectors, proofs over such converted values require a lot of user interaction. For this reason we define a corresponding bit-vector type for each of the three data types: `Mword_vec`, `Word_vec` and `Byte_vec`. The internal state variables then use the representation that requires the least conversion effort. The operands and results of the public interface are always represented numerically to match the C++ data types.

4.2 Layer Analysis

The remainder of the chapter deals with the specification of the hardware model in detail. For each layer we first examine the hardware it models, classify its subparts, and try to further simplify it. Then we specify a state for the remaining parts and define its functionally. Where appropriate the properties of the layer are shortly discussed.

4.2.1 Physical Memory

The foundation of the model is the state of physical memory. It is the base upon which linear and virtual memory are constructed and it determines the state of those system structures that are stored in physical memory instead of having their own dedicated registers.

In Hardware

IA32 distinguishes between physical memory and physical address space. Physical memory refers to the physically available main memory (RAM). The physical address space is defined as every byte that is addressable on the address bus (III–3.3). Normally the address bus has a width of 32 bits, thus spanning an address space of 4 Gbyte. Later processors introduced four additional address lines resulting in a bus size of 36 bits. Making use of the extended address scope requires a special addressing mode, either PAE or PSE-36. Both are unused in Fiasco and therefore are unsupported in the model.

The RAM is always located at the lower end of the physical address space. Higher regions can be used to map other types of memory, for example, I/O ports or the memory for the

[1]The PVS type below is defined as a subtype of $\mathbb{N}$: $below(n) = \{x \in \mathbb{N} | x < n\}$.

system management mode. Some regions may be mapped to no physical memory at all. The IA32 manual does not define what happens if those physical addresses are accessed. For a verification model such a usage has to be fatal.

The Model

First of all, we can now define the types `Address` and `Byte` for the IA32 architecture. They are used for the memory interface of the C++ model as well as in the hardware model as a base data type. The width of the address bus defines the address range:

```
Address : Type = below(2^32)
```

The smallest addressable unit, the byte, has a width of 8 bit:

```
Byte : Type = below(2^8)
```

This layer confines itself to modelling RAM. The available memory is a fixed quantity of the processor and normally less than the full address space. We model its size as a processor parameter: `max_pm`. The state of physical memory then corresponds to the data that is currently stored in RAM. Each address has assigned a byte value:

```
Physical_memory : Type = [ below(max_pm) -> Byte ]
```

For access a read and a write function have to be defined:[2]

$$\begin{array}{rcl} \mathit{physical\_read} & : & \mathit{Address} \rightarrow (\mathit{State} \rightarrow \mathit{Result}[\mathit{State}, \mathit{Byte}]) \\ & & a \mapsto \lambda(s) : \begin{cases} \mathit{OK}(s, s(a)) & \text{if } a < \mathit{max\_pm} \\ \mathit{Fatal} & \text{otherwise} \end{cases} \\ \\ \mathit{physical\_write} & : & \mathit{Address} \times \mathit{Byte} \rightarrow (\mathit{State} \rightarrow \mathit{Result}[\mathit{State}, \mathit{Byte}]) \\ & & (a, b) \mapsto \lambda(s) : \begin{cases} \mathit{OK}(s \text{ with } s(a) = b, \mathit{unit}) & \text{if } a < \mathit{max\_pm} \\ \mathit{Fatal} & \text{otherwise} \end{cases} \end{array}$$

Plain Memory Properties

Physical memory constitutes the most simple form of plain memory. Reading and writing always fulfils the plain memory properties as long as the address is within the boundaries of existing physical memory.

4.2.2 Linear Memory

The layer of linear memory specifies the paging mechanism. It adds functions to read and interpret page tables, translate linear addresses, and it raises page faults.

In Hardware

IA32 paging uses two levels of page tables. Each *standard page* has a size of 4 Kbyte. Beginning with the Pentium processor it is possible to omit the second level of page tables and directly address a *super page* of 4 Mbyte in the first-level. All page tables reside in physical memory. The location of the current first-level page table, the *page directory,* is stored in the page directory base register (PDBR). when forming the base address The processor uses only the

[2] s with $(a := b)$ denotes a function update, formally: $s \text{ with } (a := b) \overset{\text{def}}{=} \lambda(x) : \begin{cases} b & \text{if } x = a \\ s(a) & \text{otherwise} \end{cases}$.

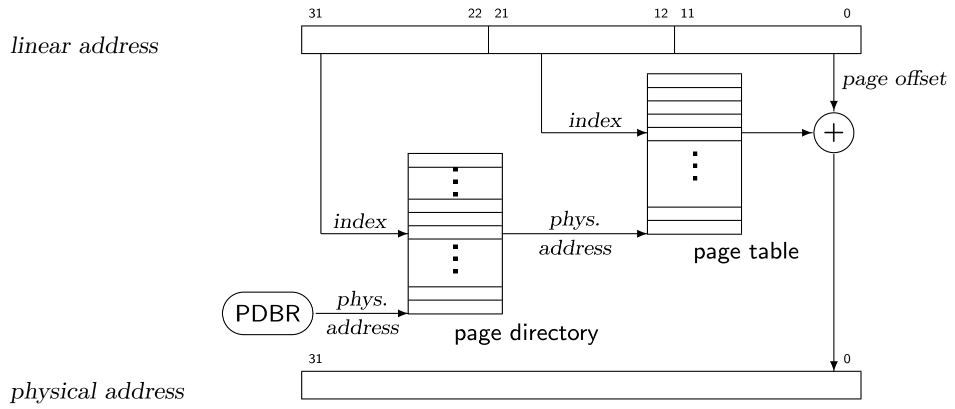


Figure 4.2: Address translation using paging

upper 12 bits of the 32-bit register and sets all others to zero. This way the page directory is guaranteed to be within a single page itself. Bits 3 and 4 of the PDBR contain flags that define the global memory caching strategy. In the page directory each entry references either a super page or a second level of page tables (*page tables*). They, in turn, point to a standard page in physical memory. The linear address is divided into three parts. They are used as indices of page directory and page table and as the offset into the page. Figure 4.2 illustrates the address translation.

Next to the address pointer the page-directory and page-tables entries contain various additional information about the pages they reference. Picture 4.3 shows their usage in the model.

- The *present* bit states whether there is any physical memory allocated to a page.
- The *user/supervisor* and *read/write* bits are used to restrict access. The privileges of the page-directory and the page-table entry are combined to determine the rights of the page. There is a peculiarity in the early 386 processors that allows supervisor to access any page regardless of the state of the read/write bit. This behaviour has been corrected with the introduction of the write-protect flag. When it is set, addresses in write-protected pages cannot be written to in any privilege level.
- *Accessed* and *dirty* bits state whether the page has been used or changed. The processor sets these bits as feedback for the operating system. Because VFiasco does not use this information they are not updated in the model making their usage unverifiable.
- The *global* bit is part of the TLB cache and as such discussed in Section 4.2.3.
- The *page table attribute index*, *write-through* and *cache-disabled* bits control the caching strategy for memory on page level. As mentioned in Section 3.2.4 those are only carried along. The same is true for any bits that are available for the system programmer's use.
- Unused bits in the page-directory and page-table entries are reserved and setting them can cause a page fault in certain processor states. Therefore, it is considered fatal to set them to anything else then zero, even if the processor does not perform this check.

Accessing linear memory can rise page faults. They have three kinds of causes: a non-present page, an access violation, or reserved bits in the page-table entries. If the processor comes upon one of the conditions it saves the faulting address in the CR2 register and sets up a page fault exception. The cause of the page fault is then passed to the handler as an error code.

Page-directory entry 4-Mbyte page

31 21 11 0

Page Base Address	–	PAT	Avail.	G	PS	D	A	PCD	PWT	U/S	R/W	P
M	0	C	C	M	1	C	C	C	C	M	M	M

Page-directory entry 4-Kbyte page

31 21 11 0

Page-Table Base Address	Avail.	G	PS	PAT	A	PCD	PWT	U/S	R/W	P
M	C	C	0	C	C	C	C	M	M	M

Page-table entry 4-Kbyte page

31 21 11 0

Page Base Address	Avail.	G	–	D	A	PCD	PWT	U/S	R/W	P
M	C	M	0	C	C	C	C	M	M	M

M - modelled, C - carried along

Figure 4.3: Layout of page table entries

As mentioned before paging is mandatory in the model. Super-page and write-protection support have been introduced in later processor generations, so they need to become processor parameters.

The Model

Linear memory adds the contents of the PDBR and the CR2 register to the system state. For the PDBR only the valid bits are stored. The two bits for cache control are carried along. Furthermore, it is essential to know the current privilege level when handling access control. So we add it already at this level although it is part of segmentation. IA32 supports four privilege levels. They are mapped to two levels for page-level protection. Level zero denotes supervisor level, all other levels are considered user level. The state of linear memory sums up to the following:

```
Linear_memory : Type = [#
    memory     : Physical_memory,
    priv_level : below(4),
    pdbr       : bvec[20],
    pdbr_cache : bvec[2],
    cr2        : Address
#]
```

This layer provides read and write functions for the PDBR and CR2. When setting the PDBR, unused bits must be zero or writing fails. Fiasco only reads the CR2, it never writes it. Even so, the layer provides a write function because the function that sets up the page fault handling needs to set the register to a valid value. From the point of view of the hardware model the register is only carried along.

Furthermore, we have to define the linear memory interface. For it the read and write functions of physical memory have to be extended. Before actually accessing physical memory the linear address operand has to be translated into a physical one. This is done by the look-up function `linear_l_t_p`. It involves three steps: First it reads the page-directory entry and, if necessary, the page-table entry as untyped 32-bit vectors from physical memory. If this is successful it checks that the two values constitute legal page table entries and that the desired access can be granted. Finally, it calculates the physical address. The new access functions `linear_read` and `linear_write` form the linear memory structure.

Looking up a linear address can yield a page fault if the requested page is not present or access is not allowed. Page faults constitute an abnormal termination state. As such we model them as another element of the `Result` type:

```
Result : Datatype =
  Begin
    ...
    Page_fault(pfa : Address,
               page_fault_flags : Pagefault_flag) : Page_fault?
    ...
  End
```

Both the cause of the page fault and the faulting address are returned. Note that the page fault condition does not yield a subsequent state. It is not needed because the system state is always reset to the state before the execution of the offending instruction when handling the page fault. A semantics for dealing with page faults, i.e., reacting on a `Result` of `Page_fault`, is not part of the hardware model and will be discussed in Chapter 5.

Plain Memory Properties

The access functions `linear_read` and `linear_write` can fail for various reasons. The following two properties guarantee that they terminate successfully:

- *Consistency:* Reading from physical memory while traversing the page directory never fails. If a physical address can be resolved it lies within the existing physical memory.
- *Existence:* The page-table entries are in a legal format, there is a page mapped at the requested address, and the desired access can be granted.

If they hold linear memory fulfils the first of the three condition that were stated in Section 2.2.2. They do not yet guarantee that the value behind an address does not change or that other addresses are unaffected by write operations. To allow that the following properties have to hold additionally:

- *Directory entries:* Addresses that should be written must not be part of a page-directory or page-table entry that belongs to a page where any other address of the plain memory region is located.
- *Sharing:* If two linear addresses point to the same physical address none of the two may be writable.

4.2.3 TLB Memory

Translation look-aside buffers (TLBs) are responsible for caching address translations from linear to physical memory. They store recently used page-directory and page-table entries to accelerate the look-up. Unlike memory caches the TLB is not entirely transparent to the operating system. Therefore, it cannot be dismissed.

In Hardware

Whenever the processor has to traverse page tables it caches the results in the TLB. There it keeps them until they need to be replaced by other entries. Thus, whenever the operating system changes the page directory it must explicitly remove affected entries from the TLB. To delete a single entry it can use the `INVLPG` instruction. If it wants to flush the entire TLB it has to reload the page directory by writing into the PDBR register.

Pentium Pro further introduced the concept of global pages. Entries for such pages are pinned to the TLB. Reloading the PDBR does not flush them. To remove them the global-pages feature has to be switched off temporarily by clearing the page-global flag in the CR4 register (see Section 4.2.6). This technique shall be called a *global flush.* Accordingly, a *local flush* denotes a TLB flush that is done by reloading the PDBR. If the global-paging mechanism is disabled both have the same effect. The `INVLPG` instruction always causes a flush of the affected entries regardless of their global state.

In real processors there are generally two independent TLBs for code and data. If the processor supports super pages they normally have their own cache as well. The IA32 manual leaves the contents and inner workings of the TLB entirely unspecified. It solely guarantees that the contents of the TLB and the page directory in memory are consistent as long as the following rules are obeyed:

1. Whenever a page-directory or page-table entry is changed or deleted, the operating system must immediately invalidate the corresponding entry in the TLB so that it can be updated the next time the entry is referenced. (III–3.11.)

2. When handling a page-fault exception the TLB must be flushed after the update, even if the faulting page was not present before. (III–3.7.6.)

3. Starting with Pentium 4 speculative execution (*snooping*) demands that there must be a flush between the change of the page directory and the access of the corresponding address. Flushing before the change is not enough even if it is done immediately beforehand and no address in the affected page is accessed in between. (III–9.7.)

Basically the operating system must flush the TLB after any changes of the address or the rights in the page directory. It only may delay this flush until the time the affected addresses are accessed again.

Currently Fiasco considers two cases where the TLB becomes inconsistent:

- *Deleting an entry or changing the physical address at the lowest level* leaves an inconsistent entry in the TLB.

- *Restricting rights* might grant access to a page that has been forbidden in the meantime.

All other changes are assumed to be safe. Even though this conforms to general TLB usage those changes are still perilous. It is possible to construct a TLB for each of the changes that conforms to the specification but exhibits an inconsistency:

- *Making a page-directory or page-table entry present* might lead to an unexpected page-fault if a TLB caches non-present pages.

- *Expanding rights of an entry* can raise an unexpected page fault if rights are checked against the TLB only.

- *Transforming a 4-Mbyte page into a set of 4-Kbyte pages (splitting)* might lead to problems with correctly combining rights. Assume the 4-Mbyte page was read-only, which is reflected in the new page set by giving read-only rights to the page-table entries and full rights in the page directory. On a write access a TLB implementation might now find the stale 4-Mbyte page entry with read access in its cache and decide to confirm the access violation in the page directory. There it finds that the page-directory entry now grants write access.

- *Transforming a set of 4-Kbyte pages into a 4-Mbyte page (merging)* might lead to an access to a page table that does not exist any longer. Consider a TLB that caches page-directory and page-table entries as they come from memory. Before merging a set of pages the page-directory entries has already been cached due to some access in one of the 4-Kbyte pages. After merging the pages an address in a different former 4-Kbyte page is accessed. The TLB page-directory entry would still be missing the super-page bit. Therefore, the TLB is checked for the Kbyte page-table entry. It is not found and read from memory instead, from a page table that is no longer valid and might have already been overwritten.

- *Redirecting a page-directory entry to another page table* leads to a faulty look-up under the same conditions as merging.

Note that the example TLB implementations given for merging and splitting rely on information from different states of the physical memory to translate an address. An implementation like that is highly unlikely. Still, it would be a valid one according to the IA32 manual. Therefore, it is valid in a model that implements the IA32 specification instead of a specific processor.

The Model

TLBs are the only part of the architecture whose functionality is not exactly defined. Therefore, the following model can only be a generalised abstract TLB model that incorporates the properties stated above.

When the processor needs to translate a linear address it first checks if it can find the corresponding page-directory and page-table entries in the TLB. If it cannot find them it loads them from the page directory, which is always consistent with itself. If the entries are cached in the TLB they must have been added at some time since the last flush of that address. Which state (or states, for that matter) constitutes the TLB entry depends on many factors: (1) whether the address has been access before, (2) how many other addresses have been accessed in between, (3) the size of the TLB, (4) its replacement strategy, (5) how long the address resolution was done before the execution of the instruction and many others. While it is possible to exactly model the first three factors, the latter two are well-kept secrets of processor design. Therefore, not much can be said about the actual content of the TLB. The only guarantee that can be given is the following:

> An address translation is ensured to be consistent if since the last flush the accompanying page-table entries has never been in a state that could have caused an inconsistent entry to be loaded into the TLB.

Note that this guarantee requires to examine every single state since the last flush. This eliminates a very simple model: The TLB model stores the state of the page directory after the last flush and compares before each address translation if the current and the stored page directory yield the same address. It might happen that one of the page-table entries was changed in between and then restored to its old value. Even if none of the affected addresses are accessed in between the inconsistent state might have entered the TLB due to snooping.

We could now express the consistency guarantee as an axiomatic property of the TLB. This requires an expression in temporal logic. A more simple approach is to formulate a predicate over all linear addresses that states whether an address is consistent. This predicate can be added to the system model as TLB state:

$$TLB : Address \rightarrow bool$$

True indicates that all system states since the last flush have had a consistent page directory for that address. Now we observe every change of the state of physical memory and, from that, recompute the state of the TLB:

$$TLB = \lambda(a : Address) : TLB(a) \wedge \neg tlb\_address\_changed(old\_state, new\_state)(a)$$

We define `tlb_address_changed` as a function that returns true for all addresses that exhibit inconsistencies between the two states. It is the heart of the TLB implementation. How exactly this functions is defined determines how restrictive the TLB model is. A very restrictive model would simply compare for each address the associated page-directory and page-table entries and mark the address changed if they are not identical. On the other hand, a very weak realisation could implement the current TLB usage of Fiasco. The implementation in this model strictly follows the IA32 manual because any other implementation requires a careful study of real processors to ensure that it is correct. Such a study is beyond the scope of this thesis.

The consistency predicate suffices to ensure that a memory access is consistent with the current page directory. If the model should allow to distinguish between global and local TLB flushes we further need to take into account the global state of an address.

A global flush simply leaves the entire address space consistent:

$$tlb\_flush\_global : TLB = \lambda(a : Address) : \mathrm{true}$$

A local flush, on the other hand, does not remove entries from the TLB that are inconsistent and marked global. So we need another predicate `TLB_global` that states whether since the last flush there has been an inconsistency, that was caused by a global page-table entry. Using this predicate a local flush can be defined as:

$$tlb\_flush\_local : TLB = \lambda(a : Address) : TLB(a) \vee \neg TLB\_global(a)$$

Like the consistency predicate the global predicate needs an update function of its state. When trying to define such a function we soon realise that a simple boolean predicate is not enough for the global state. Consider the following two examples:

- The global bit of a page-table entry is changed from local to global and back to its local state again. Afterwards the physical address resolution is changed.
- The address resolution of a local page-table entry is changed. Afterwards the global bit is first set and then unset again.

In both cases the entries are in an inconsistent local state after the changes. Still, in the first case a global flush is required while in the second a local flush suffices. The difference lies in the global entry that could be lingering in the TLB. In the second example it is still consistent, in the first case it is not. The problem is that not a single change in the physical memory causes the dangerous state with a global inconsistent entry but a particular sequence of state changes. Therefore we have to distinguish three cases of the global state: (1) all old entries are local, (2) an old entry is global but consistent with the current page directory and (3) there is a global stale entry in the TLB. A local flush then only fails in the third case. Figure 4.4 illustrates how the new global state of the TLB is determined when the state of the underlying memory changes[3].

The consistence and global state functions constitute the state of the TLB memory. To be able to decide whether to do a local or a global flush it further needs to know about the global-pages feature.

[3]There still exists a special case of manipulating the global bits that is not usable in this model but should work in reality: If a global entry is changed to some different local one and changed back afterwards to the global entry it had before, a local flush suffices in reality but not in our model. We consider a software that relies on that kind of behaviour as poorly designed.

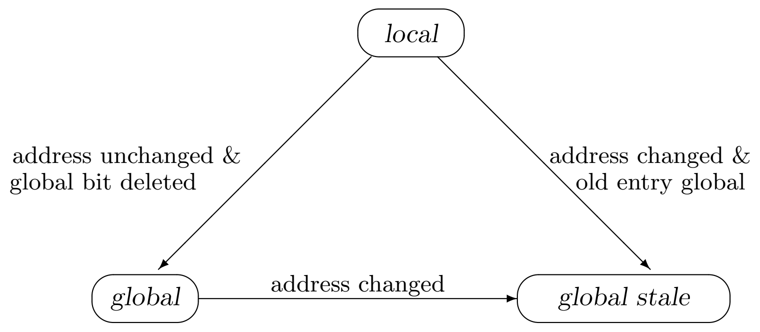

local
address unchanged & global bit deleted
address changed & old entry global
global
address changed
global stale

Figure 4.4: Determining the global state of an address: Changes to the page-table entries that change the global state of associated linear addresses.

```
Tlb_memory : Type =
    [# memory          : Linear_memory,
       tlb_consistent : [Address -> bool],
       tlb_global     : [Address -> {local, global, global_stale}],
       global_enable  : bool
     #]
```

The two memory access functions `linear_read` and `linear_write` have to be wrapped with the consistency check and TLB update functions. Before an address can be accessed we check that it is still consistent. After the access the TLB is updated.

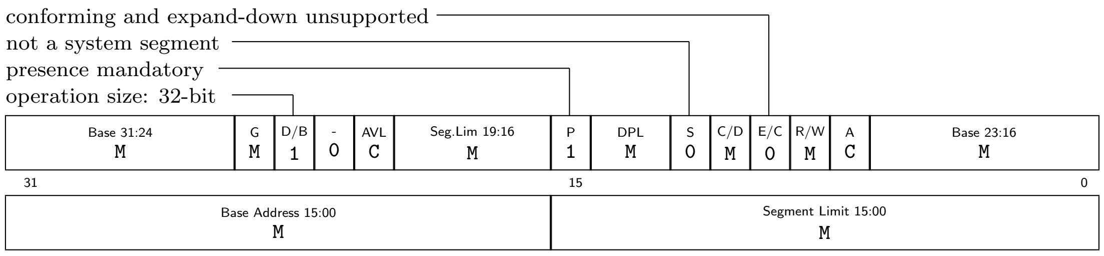
The TLB update function can also be used if the physical memory is changed in other ways then through the memory interface of the hardware model. For example, it is possible to integrate DMA from devices in the model without loosing TLB integrity. The associated functions only have to be wrapped with the `tlb_update` function that compares the states before and after the change and updates the TLB accordingly.

The write function of the PDBR register has to be extended to flush the TLB, either locally or globally. Changing the value of the `global_enable` bit causes a flush of the TLB as well. Setting the bit is done by writing into the CR4 register. This function will be implemented in the register layer in Section 4.2.6.

The `INVLPG` instruction takes an address as an operand and guarantees to flush all entries relevant for this address. Beyond that the exact semantics of the instruction are explicitly defined as implementation dependent (II–3.2). The TLB is guaranteed to store address translation for whole pages. Because the smallest page granularity is 4 Kbyte this is the minimal scope that is flushed. In the model we restrict the `tlb_invlpg` function to flush all addresses that belong to the same 4-Kbyte page as the indicated address. To support flushing of individual 4-Mbyte pages we would have to assure that the processor indeed caches super pages in their entirety and then model two individual TLBs for 4-Kbyte and 4-Mbyte pages. Since Fiasco does not yet use the `INVLPG` instruction we omit this step for the sake of simplicity.

Plain Memory Properties

The TLB introduces the inconsistency of its cache as a source of an unsuccessful memory access. These inconsistencies can arise only when a page-directory or page-table entry is changed. The page-directory property of linear memory already forbids writing to them. Therefore, the same properties apply for linear memory and TLB memory.



M - modelled, C - carrieds along

Figure 4.5: Segment descriptors for memory segments

4.2.4 Segmentation

Basically, segments provide the ability to divide memory into regions of different usage and protect them from each other. These regions constitute the IA32 virtual memory. In addition to that, the processor uses segments and their descriptors to store various system information.

In Hardware

Segment types can be divided into memory and system segments. System segments comprise interrupt, trap and call gates, the local descriptor table (LDT) and the task-state segment (TSS). Gates are explained in detail in Section 4.2.7, the LDT will be discussed later in this section. The TSS we already dismissed in Section 3.2.4.

Memory segments describe a region in linear memory. They can have one of two different types: code or data. Code segments are always executable and may optionally be readable. Additionally, the conformance property influences from which privilege levels the segment may be loaded. Data segments are at least readable and can also be made writable. In addition, it is possible to choose whether the addresses above (expand-down) or below (expand-up) the given limits constitute the segment. Neither conforming code segments nor expand-down data segments are used by Fiasco. They are unsupported in the model.

Segments are defined by segment descriptors. They contain the segment type, their position in linear memory, a limit, and a privilege level (DPL). The latter states which privilege level is at least required to access them. Like page tables, segments have an accessed bit, which is unverifiable, and an available bit for usage by software, which is carried along. Figure 4.5 depicts the usage of the descriptor information.

Segment descriptors are stored in segment descriptor tables of which there are three: the global descriptor table (GDT), the local descriptor table (LDT) and the interrupt descriptor table (IDT). The IDT only contains references to interrupt handler functions, which we will discuss in Section 4.2.7. GDT and LDT can both be used to store memory segment descriptors. The position and size of GDT and IDT are stored in a register, GDTR and IDTR respectively. The LDT in turn is indirectly referenced by an LDT segment descriptor within the GDT. Because the LDT is not used we keep the model simple and leave LDTs unsupported.

Software can never access linear memory directly, it always has to use a segment. In order to avoid recurring reading of segment descriptors from memory they are cached in segment registers. There is one register for code segments (CS), one for stack access (SS) and four for data access (DS, ES, FS, GS)[4]. Each register has a visible and a shadow part. The visible part contains the index into GDT or LDT and the requested privilege level (RPL). The RPL

[4]There are two more segment registers that are not modelled: one for the current task-state segment, and one for the LDT.

for data segments restricts access to that segment. The RPL of CS functions as the current privilege level (CPL) the CPU is running in. The shadow part caches information from the segment descriptor.

The processor checks the type and the privilege level of the segment already when loading the segment registers. What remains to be done before accessing memory is to ensure that the address lies within the boundaries of the segment and that the desired access is allowed.

The IA32 architecture additionally offers several functions (LAR, LSL, VERR, VERW) to check accessibility of memory through a segment descriptor. For this model we assume that the kernel knows about the contents of the GDT and leave these functions unsupported.

The Model

The state of the segmented memory layer consists of the GDTR register and the segment registers:

```
Segmented_memory : Type =
  [#  memory           : Tlb_memory,
      gdtr             : [# base : Address,
                            limit : Word #],
      segment_register : [{CS, DS, ES, FS, GS, SS} ->
                            [# register : Word_vec,
                               base     : Address,
                               limit    : Mword,
                               dpl      : below(4),
                               stype    : {System, Code, Code_Readable,
                                                     Data, Data_Writable}
                             #]
                         ]
  #]
```

For the GDTR the linear base address and the size of the global descriptor table are stored. The segment registers contain the currently loaded segment value and in the shadow part base address, size, access type, and DPL of the segment. The first one is used to calculate the linear address and the others are required to perform access checks. We cannot simply read this information from the GDT in memory. Once the segment register is loaded the processor no longer keeps it consistent with the GDT entries in memory.

All registers need functions to read and write them. The content of the GDTR is always laoded from and stored to memory directly. The respective functions take a virtual address as an operand. The write function is more restrictive than IA32 manual states. It fails if the given GDT would be outside linear memory.

When loading segment registers the corresponding segment descriptor is read from the GDT in linear memory and checked for the correct type and privilege level. The CS register cannot be loaded directly but only implicitly when performing far jumps (see Section 4.2.7).

Memory read and write functions have to be prepended by the address translation of segmentation. The address is checked to be within segment boundaries and, when writing the segment, the segment is checked to be writable. Then the address is translated into its linear form by adding the segment base address. Addresses in segmented memory always consist of a segment-register/address pair. This cannot be expressed by the general memory interface anymore. Therefore, we define for each segment its own memory interface by parameterising the interface over the segment selector.

Properties for Flat Segments

Like most modern operating systems FIASCO tries to avoid usage of segmentation as far as possible[5]. Segmentation cannot be switched off completely so FIASCO uses flat segments, which span over the entire linear memory and grant as much access as possible. One might argue that it would greatly simplify the model to support only these kind of segments. Unfortunately this is not the case. The most expensive part of segmentation is hidden in the loading process of segment registers. It cannot be omitted because it is impossible to make any static assumptions about it. The segment descriptors have to be read from the GDT, which resides in linear memory. Thus, it might be changed any time as a side effect of another function. Reloading the segment registers is done regularly when using interrupt gates. FIASCO reads the contents of segment register as well, so that the their state needs to be stored, too. For this reason we have modelled the complete segmentation mechanism with the exceptions already pointed out in the last section.

To simplify the reasoning over the model for software with a flat segmentation we provide the appropriate theorems. A flat segment is either a readable code segment when loaded into CS or a writable data segment when used with one of the other segment registers. It has to have a base address of 0, a limit of $2^{32} - 1$. If a segment descriptor that meets these conditions is loaded it can easily be proved that all subsequent reads and writes through this segment have the same result as if linear memory was accessed directly.

4.2.5 Stack

Stack is a region in virtual memory with the dedicated use as a LIFO memory. As such it is the responsibility of the C++ model to make sure it is set up correctly and used type-safely within the application. The problem is that the hardware uses the same stack as the C++ model. The processor needs stack in control transfer instructions to save system state information. This layer adds the hardware support for stack on top of which we are going to define interrupt and exception handling in Section 4.2.7. The C++ model might later utilise it to define its own memory allocation.

In Hardware

IA32 dedicates a special segment to stack. It has its own stack segment register (SS) and a stack pointer register (ESP) that points to the top of the stack. It is used with the `POP` and `PUSH` instructions. Apart from that, the stack can be used like any other memory region and the ESP like any other general-purpose register (see Section 4.2.6).

Stack grows downwards towards address 0. The ESP always points to the topmost 32-bit word on the stack. Thus, push functions first decrement the stack pointer and then write the value to the new address. Accordingly, pop functions first read the value from stack and increment the stack pointer afterwards.

The Model

The stack segment register has already been added to the state in the segmentation layer in Section 4.2.4. This layer only adds the stack pointer to the system state:

```
Stack_memory : Type = [# memory : Segmented_memory,
                         esp    : Address
                       #]
```

As usually there have to be functions to read and write the stack pointer itself. Push and pop functions save and restore data on stack and set the stack pointer accordingly. In this

[5]There is one extension to FIASCO that proves the exception to this rule: Small address spaces [Hof02] multiplex multiple tasks into one address space by putting them into different non-overlapping segments.

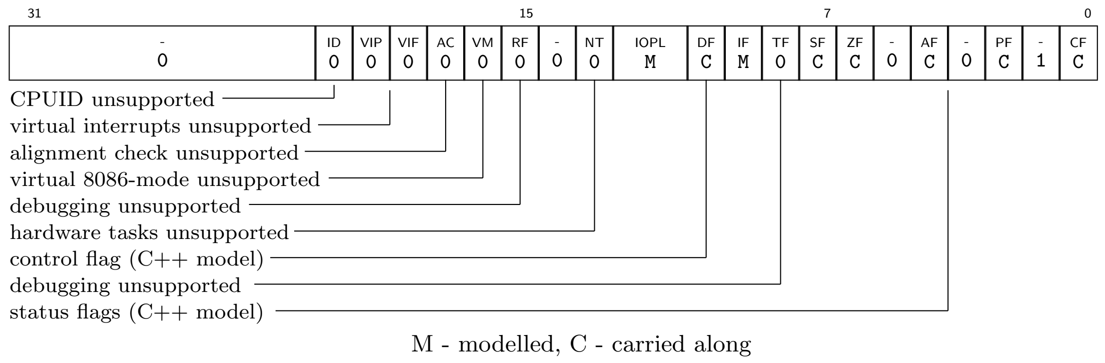


Figure 4.6: EFLAGS

layer we add the very general `stack_push` and `stack_pop` functions that handle 32-bit words. Other functions that save special CPU registers like EFLAGS on the stack will follow in the appropriate layers.

4.2.6 Registers

At this point we add those registers of the hardware state that are not used by the hardware model itself but need to be saved for direct access by the kernel: the general-purpose registers, the flag register and the control registers CR0 and CR4.

In Hardware

IA32 has seven 32-bit general-purpose registers[6]: EAX, EBX, ECX, EDX, EDI, ESI, EBP. They are at the free disposal of the C++ model; the hardware model does not manipulate them. Fiasco accesses them directly only when saving their state. All registers can be used either as 32-bit or as 16-bit words. In the latter case only the lower half of the register is used. EAX through EDX can also be accessed as 8-bit bytes.

The EFLAGS register is a 32-bit register containing status, control and system flags. Like the general-purpose registers it is part of the task state. The status flags indicate results of arithmetic operations, which are hidden by the C++ model. We will leave their value undefined although that renders their usage unverifiable because they are normally set by hardware. The only control flag DF controls the direction of string operations,[7] which are part of the C++ model as well, so the DF flag is just carried along. Of the system flags solely the interrupt enable flag (IF) and the IOPL are of interest. Picture 4.6 summarises how all flags are handled.

The control registers CR0 and CR4 are 32-bit flag registers that are used to control global processor settings. Pictures 4.7 and 4.8 show the flags in detail.

The Model

Of the EFLAGS, CR0 and CR4 registers we have to save the state of those flags that are carried along, so they can be correctly retrieved. The value of PGE, the only flag modelled, is stored in the TLB layer.

All three register require access functions that read and write a 32-bit value. To realise them we define conversion functions from and to a 32-bit word. When converting from 32-bit words they have to meet the conditions stated in pictures 4.6, 4.7 and 4.8. CR0 and CR4 are used with standard read and write functions that directly change the content of the

[6]We do not follow the view of the IA32 manual here, which counts the ESP register (see Section 4.2.5) as a general-purpose register because of its special usage.

[7]String operations are those that handle large data structures in memory with one single instruction.

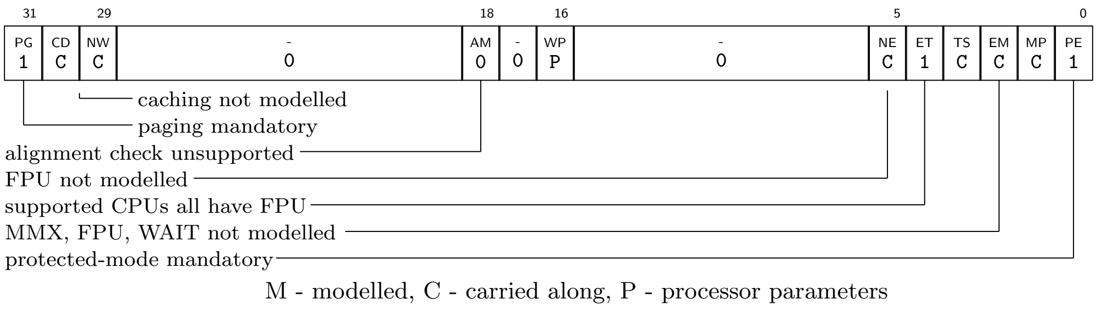


Figure 4.7: CR0

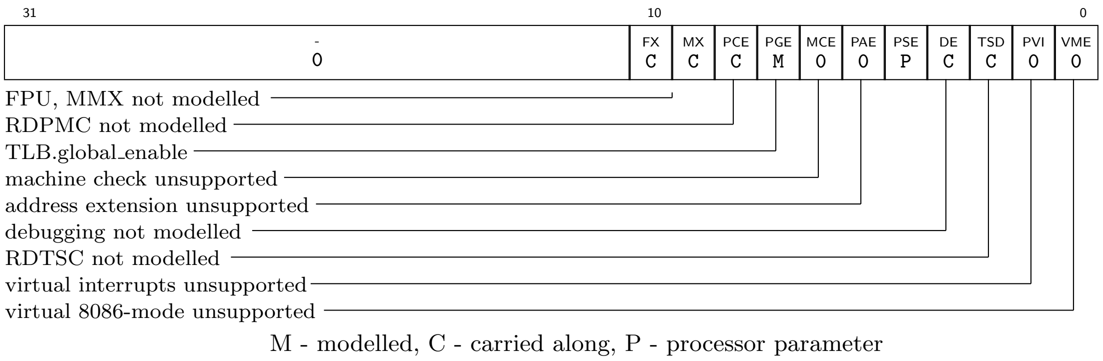

31
10
0
- 0
FX C
MX C
PCE C
PGE M
MCE 0
PAE 0
PSE P
DE C
TSD C
PVI 0
VME 0
FPU, MMX not modelled
RDPMC not modelled
TLB.global_enable
machine check unsupported
address extension unsupported
debugging not modelled
RDTSC not modelled
virtual interrupts unsupported
virtual 8086-mode unsupported
M - modelled, C - carried along, P - processor parameter

Figure 4.8: CR4

registers. EFLAGS can only be read and written by transferring the register content from and to the stack using the `POPF` and `PUSHF` instructions. Most of the flags of the EFLAGS register additionally have dedicated assembler instructions to read and write them. Fiasco only uses the `CLI` and `STI` instructions to manipulate the interrupt flag. All other functions are hidden by the C++ model.

Modelling the general-purpose registers is slightly more complicated. From the point of view of the C++ model there is not much difference between general-purpose registers and memory. Both are used to store variables. Therefore, it is a good idea to provide a memory interface for general-purpose register as well. We cannot use the standard address type as we have defined it in Section 4.2.1 because it covers only the physical address space. Instead, we define a new type `Register_address`. The C++ model can later combine both address types to define an address space that includes both memory types.

Following the example of physical memory the registers can now be modelled as a simple chunk of memory:

$$gp\_memory : Register\_address \to Byte$$

This bears one problem: registers have a size of 32 bits. That sums up to four bytes, not one. So we carefully have to choose the correct function domain. If there are `register_num` registers in the CPU, each of them having a width of `register_byte_size` bytes, the type `Register_address` is defined as follows:

```
    Register_address : nat = below(register_num * register_byte_size)
```

Now each general-purpose register can be assigned `register_byte_size` successive, mutually exclusive bytes. This results in a correct register model. However, as soon as we define the read and write functions `reg_gp_read` and `reg_gp_write` over this memory in the same way as we did for physical memory it becomes clear that the problem of the over-sized registers is not yet solved. Let us assume that EAX occupies bytes 0 to 3 of `gp_memory`. In this case `reg_gp_read(2)` reads the third byte of the EAX register, which cannot be read by itself in a real processor. If we simply restricted the domain of the read and write functions to reflect the restrictions of the hardware they could not be used as a general memory interface anymore. Remember how complex data types are accessed by the C++ model: each byte is read by itself and afterwards they are connected. This is also true when a 32-bit value is read from a register. So we leave the functions as they are and assume that the C++ model defines a correct use for them.[8]

The overall state of this layer sums up to:

```
Register_memory : Type = [#
    memory     : Stack_memory,
    cr0        : [# cr0_cd, cr0_nw, cr0_ne,
                    cr0_ts, cr0_em, cr0_mp : bool
                  #],
    cr4        : [# cr4_pce, cr4_de, cr4_tsd : bool
                  #],
    eflags     : [# iopl : Cpu_privilege,
                    of_flag, df_flag, if_flag, sf_flag,
                    zf_flag, af_flag, pf_flag, cf_flag : bool
                  #],
    gp_memory  : [Register_address -> Byte]

#]
```

The read and write functions for general-purpose registers can only be used by the memory interface. When a specific register needs to be accessed, i.e., when FIASCO needs to save the state of some of the registers, we need special access functions. They have to address the registers by their names and use 32-bit words as operands:

```
reg_gp_read(r : Registers)(s) : Result[Register_memory, Mword] =
  let ra = registers2registeraddress(r) in
    OK(s, s`gp_memory(ra) +
          max_byte * (s`gp_memory(ra + 1) +
          max_byte * (s`gp_memory(ra + 2) +
          max_byte * (s`gp_memory(ra + 3)))))
```

The function reads the four assigned bytes and concatenates them to become a 32-bit word. It resembles the `from_byte` function (see Section 2.2.2) for the 32-bit word data type. The function `registers2registeraddress` yields the lowest byte in `gp_memory` for the given register. The write function `reg_gp_write` is then defined similarly to `to_byte`. FIASCO directly touches general-purpose registers only by transferring them to and from the stack. For convenience, functions to transfer single registers as well as the complete set (including the ESP register) to and from the stack are provided as well.

4.2.7 Interrupt and Exception Handling

This layer contains the hardware infrastructure for interrupt and exception handling. It provides access to the interrupt table and functions to set up and return from an interrupt.

[8] As long as the C++ model generalises over the use of registers that does not even provide a problem: Assume a variable is saved at an arbitrary address $x \in$ *Register_memory* and then a program can be verified for all values of x. This additionally proves some cases that are not possible in reality (e.g., saving the variable in the third byte of an register) but also all possible cases.

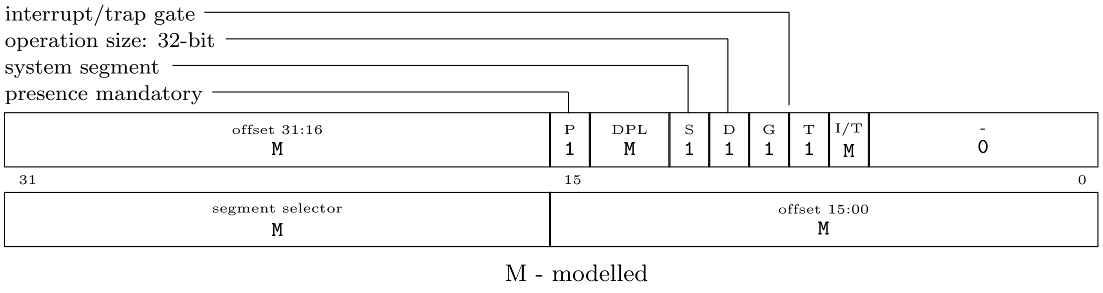


Figure 4.9: Interrupt and trap gate segment descriptors

In Hardware

Interrupts and exceptions mainly differ in where they arise. Exceptions emerge from faulty instructions and cannot be suppressed. Some exceptions provide an error code to describe their cause more closely. Interrupts normally originate in hardware outside the processor. They can occur at any given time although the processor guarantees to always call the interrupt handler on instruction boundaries. It is possible to forbid most interrupts by clearing the interrupt flag (IF) in the EFLAGS register. Nonmaskable interrupts are received nonetheless.

IA32 handles interrupts and exceptions in the same way. Therefore, when talking about interrupts below we refer to exceptions as well.

Interrupts are identified by an interrupt vector. Each vector can be assigned its own interrupt handler. Those handlers are found in system segment descriptors of the interrupt descriptor table (IDT). This table is similar to the GDT (see Section 4.2.4). It can only contain three types of descriptors: interrupt, trap and task gate descriptors. Task gates are used to handle an interrupt by switching to another hardware task. As hardware tasks are unsupported task gates are unsupported as well. Interrupt and trap gate descriptors define the interrupt handler by the code segment and address of the handler routine. Interrupt handlers can run at a higher privilege level than the code they interrupt. We leave this change of privilege to user-mode verification because kernel code runs at the highest privilege level already. Interrupt handlers run at the same level.

When an interrupt is raised the processor reads the corresponding entry from the IDT, saves the current EFLAGS, code segment (CS), and instruction pointer (EIP) to the stack, and loads the new CS and EIP from the gate descriptor. If an exception has assigned an error code it is pushed on the stack as well. Some exceptions, like page faults, are restartable: after the exception was handled the faulting instruction is executed again. Thus, the EIP that is pushed on the stack does not point to the next instruction but instead to the offending one. The sole difference between an interrupt and a trap gate is that in the former interrupts are disabled.

`IRET` is the companion instruction that leaves an interrupt handler. It pops all values from the stack and restores the old CS, EIP and EFLAGS.

The Model

Real processors use the EIP as a pointer to the next instruction to be loaded and executed. In Section 3.2.4 we already decided that the C++ model hides code execution. Therefore, the model does not need the current EIP. Indeed, it cannot even keep it current because it does not know where in memory the physical representation of a state transformer resides. Still, we need to model the EIP because, as we just saw, it is saved and restored during interrupt handling. Other than that it is simply carried along. In particular, the interrupt entry and `exec_iret` functions just write an uninterpreted address to the EIP register. It is the responsibility of the C++ model to give a meaning to this value.

Apart from the EIP this layer adds the IDT base register IDTR:

```
Executable_memory : Type = [#
    memory     : Register_memory,
    eip        : Address,
    idtr       : [# base : Address,
                    limit : Word   #]
#]
```

Reading and writing the IDTR register works exactly in the same way as for the GDTR (see Section 4.2.4).

In the processor the EIP can only be changed by using control transfer instructions like jumps or loops. In spite of that, we add both a read and a write function for the EIP to the model. This way it can be manipulated by higher layers when they choose to provide an interpretation for it.

For interrupt handling we model functions to enter an interrupt gate and the `exec_iret` function to return from it. The `INT` instruction allows the software to call an interrupt as well. Of course, such an invocation is not asynchronous anymore. It rather resembles a simple function call. When invoked by software the interrupt handler additionally tests the DPL of the gate so that a caller can only access it if it has sufficient privileges. Furthermore, the `INT` instruction never pushes an error code on the stack, even if an exception handler is called.

4.3 Model Assembly

The interrupt and exception handling component forms the uppermost layer of the model. Hence, its state represents the overall system state:

```
IA32_state : Type = Executable_memory
```

This concludes the analysis of the x86 architecture. The model now covers the most important parts of the processor state and all functions that are used in the run-time code of Fiasco. Table 4.1 summarises state and functions once more. One last issue that is left from chapter 3 is the execution of code. We are going to discuss that in the next chapter.

Layer	**State**	**Functions**
physical memory	physical memory	reading and writing physical addresses
linear memory	PDBR CR2 current privilege level	reading and writing PDBR reading and writing CR2 linear address translation
TLB	consistency global state	consistency check of linear addresses update of TLB state TLB flush
segmented memory	GDTR segment registers (CS, DS, ES, FS, GS, SS)	reading and writing GDTR reading and writing segment registers virtual address translation
stack	ESP	reading and writing ESP push and pop of 32-bit values
registers	general-purpose registers EFLAGS CR0, CR4	reading and writing general-purpose registers push and pop of general-purpose registers push and pop of EFLAGS reading and writing CR0, CR4 interrupt enabling and disabling
interrupts and exceptions	IDTR EIP	reading and writing IDTR reading and writing EIP interrupt entry interrupt return

Table 4.1: State and functions of the hardware model summarised

Chapter 5

Execution of Source Code

63. The crow wish'd every thing was black,
the owl that every thing was white.

A processor and C++ source code employ very different levels of abstraction. Therefore, the object code of a program functions as a mediator between them. The C++ compiler is able to translate source code into object code and the processor knows how to interpret it. In the verification model the different abstraction levels still exist between the hardware model and the C++ model. To connect them we need a model of the object code. This could be a very concrete one that separates the verification process into two steps: First it is proved that the source code is always compiled into one concrete object code representation and then the security properties are verified on a hardware model that interprets this object code. In Section 2.2 we already stated that VFiasco aims to model this connection on a more abstract level. It wants to formulate a number of conditions that allow a direct execution of C++ state transformers on the hardware model. We stated some of these conditions already in Section 3.2.4. This chapter provides solutions for the two most prominent problems: handling of exceptions and interrupts and the connection between state transformers and their physical representation.

5.1 Interrupt and Exception Handling

When the hardware intervenes in the program flow it is no longer guaranteed that the order of the state transformers matches the order of the object code created by the C++ compiler. Such an intervention takes place in the current model when an interrupt or exception is triggered. We will first explore the problem by the example of the page faults exception and afterwards return to general exception and interrupt handling.

5.1.1 Handling Page Faults

The ability to handle page faults is essential for kernel-code verification. Fiasco uses it to lazily allocate memory for thread control blocks and to propagate kernel pages into the page directories of the various L4 tasks.

Page faults occur at well defined points in time: when linear memory is accessed. As such they are restricted to the hardware model. The C++ model does not even need to know about them. Instead, a memory access should return `OK` if there was no page fault or one that has been handled and `Fatal` otherwise. To achieve this we need to wrap any functions `st` that access linear memory. They have to appropriately call a page fault handler `ia32_handle_page_fault`:

```
 catch_page_fault (st : [ IA32_state -> Result[IA32_state, Data]])(s) :
                                                 Result[IA32_state, Data] =
      cases st(s) of
        OK(state, value) : OK(state, value),
        Page_fault(pfa, page_fault_flags) :
                  (ia32_handle_pagefault(pfa, page_fault_flags)(s) ,
        Fatal : Fatal
      endcases
```

Before the processor calls a page fault handler it returns to the state before it started to execute the faulting instruction. Therefore, it is not necessary to know the subsequent state of an instruction that faulted. Instead, we can apply the same state to the page fault handler as we applied to the instruction state transformer. Still, this is not yet correct. After the page fault handler has returned the processor reruns the last instruction. If the page fault was not handled correctly the instruction might cause a page fault again. The semantics resembles that of a C++ while loop that terminates under the condition, that the page fault was handled successfully. Therefore, we can reuse the while semantics of the C++ model [Tew02]. That requires to change the `catch_page_fault` function to call itself recursively:

```
ia32_iterate_exception(index : nat,
                 st : [ IA32_state -> Result[IA32_state, Data]])(s) :
                        RECURSIVE Result[IA32_state, Data] =
  if index = 0 then
    st(s)
  else
    cases st(s) of
      OK(state, value) : OK(state, value),
      Page_fault(pfa, page_fault_flags) :
            (ia32_handle_pagefault(pfa, page_fault_flags) ##
                        ia32_iterate_exception(index-1, st))(s) ,
      Fatal : Fatal,
    endcases
  endif
Measure index
```

PVS demands to prove that the recursion is finite. It expects a measurement expression, which has to be a well-founded order relation. In above function the measurement is `index`, which constitutes the number of times the page fault handler has to be called before the page fault is resolved. We are only interested in the minimum of iterations, that is, the first occasion where the page fault has been handled. Its result determines the state of the system after the execution of the wrapped state transformer. The complete exception wrapper then reads:

```
ia32_wrap_exception(st : [ IA32_state -> Result[IA32_state, Data]])
                                      (s) : Result[IA32_state, Data] =
  if ( EXISTS(n : nat) : OK?(ia32_iterate_exception(n, st)(s)) ) then
    let i = min(Lambda (n : nat) :
                 OK?(ia32_iterate_exception(n, st)(s))) in
      ia32_iterate_exception(i, st)(s)
  else
    Fatal
  endif
```

When verifying this function, it must first be proved that the page fault is ever handled at all. We normally expect that it can be resolved on the first attempt. This is indeed one of the kernel-code properties to be proved.

One problem still persists. In section 4.1.4 we decided to separate reading and writing of operands from the execution of machine instructions. Hence, the exception wrapper handles the page fault only for part of a machine code instruction, for reading of the operands, for writing of the results, or for the executional part of the instruction. Consider the following assembler instruction:

```
ADD 0xE0000000, $4
```

It loads the contents of the memory at address `0xE0000000`, adds the constant value 4, and stores the result back in memory at the same address. The corresponding PVS specification looks like the following:

```
... ##
eval_if_ok(ia32_read(DS)(0xE0000000),
           Lambda ( b : Byte ) : ia32_write(DS)(0xE0000000, b + 4) ##
...
```

The functions `ia32_read` and `ia32_write` shall be the wrapped read and write functions for segmented memory, DS is the standard segment to be used. Consider further that the page behind the address `0xE0000000` is mapped read-only. While reading from the address still succeeds, writing causes a page fault. After this page fault has been handled only the write function is repeated. However, the IA32 manual requires that the read function is repeated as well. Worse, the verification is void if the page fault handler is faulty enough to change the value of the address that the `ia32_read` function has read from.

There are two ways to solve the problem. Either we ensure that the page fault handler does not change the content of the memory or we use the page fault wrapper only on state transformers that express the full semantics of a machine instruction. Both solutions require knowledge of the C++ semantics that goes beyond the scope of this thesis.

Finally we have to define the `ia32_handle_page_fault` function. The C++ model sees the page fault handler as a regular C++ function, which can be called like any other function. Nonetheless, it is not correct to simply apply this function when handling the page fault because the handler is set up by the hardware model. It does not know about C++ functions.

Let us recall what the hardware does on an exception: The execution is interrupted, the processor puts (among other things) the current instruction pointer on the stack, looks up the associated handler function and resumes execution at the instruction the handler points to. It executes the code until it meets an `IRET` instruction. At this point, it pops several values from the top of the stack, interprets one of them as an instruction pointer and jumps to the address it points to.

The entry and the `IRET` function are modelled in the hardware model. To handle the page fault correctly the interrupt entry function of the hardware has to be called first and then the C++ model page fault handler. To obtain a correct model three conditions have to be met:

1. The EIP that is loaded by the interrupt entry function belongs to the C++ model page fault handler.

2. The last instruction the page fault handler executes is an `IRET`.

3. The instruction pointer the `IRET` function restores is the same as the one that was pushed on the stack by the interrupt entry function.

The first problem can be solved by using a reverse function of the C++ function pointer. A C++ compiler can always assign an unambiguous address to a function. The other way around we can construct a partial function that delivers for an address the related state transformer. It is partial because not every address constitutes an entry point of a function.

```
addr2st : [ Address -> lift[State_transformer]]
```

returns the `bottom` element for all addresses that do not have a state transformer assigned.

To make sure that the last instruction of the state transformer is indeed an `IRET` instruction we introduce `Iret` as another abnormal termination state:

```
Result : Datatype =
  Begin
    ...
    Iret(next_state : State) : Iret?
    ...
  End
```

Now the page fault handler can be tested for this result element. In order to ensure that `IRET` was not called somewhere in the middle of the handler the `Iret` state must not be passed along when two state transformers are connected. Instead the execution of another state transformer after it is fatal.

The third problem can be solved by adding the proof obligation that the EIP before and after the execution is the same.

5.1.2 Handling Other Exceptions

There is no fundamental difference between page fault handling and the handling of other exceptions. Each exception needs to get its own dedicated `return` state. In Fiasco we do not expect any other exceptions to be raised, so no other exceptions are currently implemented.

5.1.3 Handling Interrupts

In contrast to exceptions interrupts can be raised at almost any time. The processor guarantees only that they are raised at instruction boundaries. Again this requires knowledge about the mapping between C++ source code and machine code instructions. Otherwise it is impossible to know where in the source code specification an interrupt can occur.

In this model we simply assume that in the kernel code interrupts are disabled. Another solution to this problem would be to formulate invariants about how interrupt handlers change the hardware state. Then it has to be proved that the code is still correct if only these invariants hold.

5.2 Physical Representation of State Transformers

In Section 3.2.4 we have established two proof obligations to ensure the correct mapping between state transformers and their physical representation in machine code: The machine code in memory must match the state transformer executed, and reading an instruction must not change the overall system state.

The most straightforward solution to this is to expect the whole kernel code to be always in memory and stay untouched. The following property must hold for all subsequent states of an initial state *is* where the code was mapped correctly in the code region *code*:

$$\begin{aligned}\forall s \in \mathit{State}, a \in \mathit{code} \quad : \quad & \mathit{OK?}(\mathit{seg\_read}(\mathit{CS})(a)(s)) \\ & \wedge\ \mathit{get\_data}(\mathit{seg\_read}(\mathit{CS})(a)(\mathit{is})) = \mathit{get\_data}(\mathit{seg\_read}(a)(s))\end{aligned} \tag{5.1}$$

This only ensures that the correct instruction is read. We also have to add the condition that reading does not change the state:

$$\forall s \in \mathit{State}, a \in \mathit{code} : \mathit{next\_state}(\mathit{seg\_read}(a)(s)) = s \tag{5.2}$$

This solution restricts the kernel in two ways: it forbids self-modifying code and demand paging of code. Self-modifying code is very dangerous and difficult to handle anyway. It cannot be verified on C++ level because the modifications have to be done on the machine-code level.

Demand paging of code, on the other hand, can be very useful: In FIASCO every address space has its own set of page tables. Kernel data and code is mapped at the upper end of each of them. Using demand paging these mappings could be added lazily when they are needed. The problem is that property 5.1 does not hold anymore when reading the instruction causes a page fault. Wrapping the read instruction with a page fault handler is not possible either. Handling a page fault is always accompanied by a change of the system state, thus it hurts condition 5.2.

Having said that it should be added that FIASCO currently does not use demand paging for code. All page-table entries that are needed for kernel code are copied into the page directories immediately after they have been created. Thus, above solution is applicable for FIASCO.

Chapter 6

Implementation

68. Where man is not, nature is barren.

The model as it has been described in the last two chapters found its realisation in a PVS specification. For each layer state and functions are implemented as described in chapter 4. A number of corollaries and theorems about the functional interface complete a layer specification. The first part of this chapter examines some implementation details of the realisation while the second part deals with challenging the model.

6.1 Realisation of the Specification

The means to modularise a specification in PVS are very few. Essentially, they are restricted to the usage of potentially parametrised theories. Therefore, the PVS realisation relies chiefly on coding conventions to minimise dependencies between layers. The extension of the system state proves one exception to this rule.

6.1.1 Casting Between Layers

In Section 4.1 we defined that the system state of the underlying layer is extended by defining a separate record type for each layer and adding the state of the lower layer in a separate field. Components of the state are only changed by functions of the layer they are defined in. Therefore, higher layers have to use the functions of the lower layers regularly. This leads to complications because those functions are state transformers over the lower-level state. So to be able to use them we first have to extract the state of the lower layer, apply the function, and afterwards reintegrate the resulting state.

To be able to hide this casting we first have to define an interface `Cast_struct` that comprises two functions: `down` casts the state of the upper layer to that of the lower one and `up` reintegrates the underlying state. The interface that casts between linear and physical memory is defined as follows:

```
linear2phy : Cast_struct[Linear_memory, Physical_memory] = (#
  down := Lambda (s) : s‘memory,
  up   := Lambda (s)(p : Physical_memory) : s with [memory := p]
#)
```

The interface can be used to define a function `cast_state` that wraps a function `g` of a lower layer to become a function of the higher layer:

```
cast_state(cs : Cast_struct[State_1, State_2])
           (g : [State_2 -> Result[State_2, Data]])
                             (s : State_1) : Result[State_1, Data] =
   cases g(cs`down(s)) of
     OK(next,value)           : OK(cs`up(s)(next),value),
     Fatal                    : Fatal,
     ...
   Endcases
```

Two casting interfaces can easily be concatenated:

```
cast_cast_struct( cs1 : Cast_struct[State_1, State_2],
                     cs2 : Cast_struct[State_2, State_3] ) :
                                    Cast_struct[State_1, State_3] =
  (# down := Lambda (s : State_1) : cs2`down(cs1`down(s)),
       up := Lambda (s : State_1)(s2 : State_3) :
                            cs1`up(s)(cs2`up(cs1`down(s))(s2))
   #)
```

Every layer includes a casting interface for each underlying layer. It defines the one to the layer that it directly inherits from and then casts the casting interfaces of that layer to its own state. This way, when another layer is added in between later only the casting interfaces of the layer directly above have to be adapted. The same principle is used for memory interfaces, which have to be cast in a similar way.

To hide the complexity of casting the theory IA32_Model, which defines the global state IA32_state, exports all interface functions as state transformers of this state.

6.1.2 Infrastructure

The implementation of the hardware model could rely on a basis of specifications and theories that have been formulated in earlier stages of the VFiasco project. The most important ones are the various theories about state transformers, the Result and Data_type type and the memory interface. Where the need arose they have been extended.

6.2 Challenging the Model

The theorems and lemmas provided with the specification have two functions: they challenge the correctness of the model and they are the base for reasoning on the model.

The theorems are formulated in a way that they are mostly independent from other layers. For system structures the proofs are formulated on the memory abstraction level they are stored in, i.e., on the physical, linear, or virtual memory interface. The correctness of these interface is then proved in the appropriate layers. One of the few prerequisites that was necessary to hold in all layers is that reading from memory never changes the state. This significantly simplifies proofs over functions that access system structures in memory. If reading from memory can arbitrarily change the state the influence that these functions have on the system state is arbitrary as well.

For each function that was defined in the model we provide two common lemmas for the case of their normal termination (theory *_Corollaries):

- *function_name*_ok_next_state proves that the given function realises the change of state it is supposed to implement. First of all, it provides the proof that the function does not have any unexpected side-effects on the system state. Furthermore, it functions as an automatic rewrite lemma for PVS[1].

- `function_name_ok_get_data` provides a similar lemma for the return data. It is not necessary when the return type is `Unit`.

Properties we formulated in the last chapter can be found in the `*_Properties` theories. They generally have lemmas associated that prove the correctness of them. For system structures in memory there are additionally properties about the memory they use. They are mostly used for the model challenges but might be useful later to prove that those structures stay unchanged.

The chief part of the model challenges is located in the `*_Challenge` theories. There are for each function a number of theorems to prove its conformance with the IA32 manual. They include proofs about the prerequisites for a normal termination of the state, proofs that unsupported features indeed do not influence the model, and proofs that illegal operators are handled correctly.

Proving the theorems helped to find a number of bugs in the model. Still, due to the limited time resources available for this work the collection of challenges is not yet complete. Most urgently needed are challenges over the model in its entirety. The VHDL function specifications from the IA32 manual Section II–3.2 should be reformulated in PVS and proved against the model.

[1]PVS can use automatic rewrite lemmas to simplify proves by itself. They need to have a special form of implication.

Chapter 7

Summary and Future Work

69. Truth can never be told so as to be understood,
and not be believ'd.

In the course of this thesis we developed and realised a model of the x86 architecture that is to be used in the verification of the FIASCO microkernel.

Base of the model was the Intel architecture manual, which describes the functional specification of the processor. An analysis of the requirements of FIASCO showed that the part of the hardware that it actively uses is relatively small. We were able to limit the model to the following components: memory and memory management, interrupt and exception handling, and the general purpose and control registers. We decided in favour of a definitional model because it resembles most closely to the processor specification. For each selected component we defined a state and the functions to manipulate it. While most components were well defined, two parts of architecture provided significant problems: the TLB and code execution.

The functionality of the TLB has not been specified precisely enough to allow a definitional specification. Therefore, we developed a model that incorporates its most important property: its consistency with the page directory in physical memory.

Code execution can only be modelled on the level of machine code. The VFIASCO project tries to avoid this level of abstraction. So, instead of including a model of code execution itself we established conditions and properties that allow to disregard it without undermining the correctness of the model. On base of these properties we were able to implement interrupt and exception handling.

The theoretical model was realised in a practical specification in the verification system PVS. The realisation presented a significant part of the work. The model covers about one fifth of Intel the architecture specification. It takes up almost 3000 lines of code in 44 theories. To prove the soundness of the model we provided 204 theorems. Proving them required about 2000 prove steps. In addition to that the model relies on a framework of 32 theories with another 1000 lines of code. About one third of it was developed during the model implementation.

The model is already useful for the verification of code of the FIASCO kernel. Still, there is a lot of work to be done to allow the verification of the security properties that we introduced in the beginning. First of all, those parts we left unverifiable have to be modelled fully, namely the accessed and dirty bits in page directory (see Section 4.2.2) and segment descriptors (see Section 4.2.4). Next the model has to be extended to allow user-mode and bootstrap verification. Most importantly, the privilege level system needs to be completed.

Another unsolved problem is the verification of the correctness of the model. The challenges we provided for single functions of the model have to be further developed to challenges of the model in its entirety. Thinkable is also a way of automated challenging as chosen by the semantic compiler: The identical behaviour of the model and a real processor can be proved with the help of small test programmes. The final goal is, of course, a formal prove of correctness of the model for each existent processor.

A last issue that has to be solved is the embedding of the hardware model in the C++ model. The propositions made in Chapter 5 have to prove their usefulness in the face of a practical realisation and a number of other problems are sure to arise that have to solved with the help of the hardware model.

Appendix A

Functional overview

The following table provides a detailed overview of the state of the model as it has been implemented in the course of this thesis. Its structure follows roughly the order of the third volume of the IA32 manual.

Feature	State	Ref
general - system registers		
global descriptor table	modelled	4.2.4
local descriptor table	unsupported	4.2.4
interrupt descriptor table	modelled	4.2.7
task-state segment	modelled	3.2.4
control registers	modelled	3.2.5
debugging registers	unsupported	3.2.4
model-specific registers	unsupported	
EFLAGS	modelled	4.2.6
stack pointer	modelled	4.2.5
general purpose registers	modelled	4.2.6
general - modes of operation		
real-mode	unsupported	3.2.1
protected mode	modelled	
virtual-8086 mode	unsupported	
system management mode	unsupported	
memory management - segmentation		
segment-descriptors:		4.2.4
granularity	modelled	
non-present segments	unsupported	
16-bit segments or descriptors	unsupported	
available bits	carried along	
accessed bits	unusuable	
code/data segments, except...	modelled	4.2.4
... expand-down data segments	unsupported	
... conforming code segments	unsupported	
system segment descriptors:		
32-bit trap or interrupt gates	modelled (in IDT only)	4.2.7
32-bit TSS	unsupported	3.2.4
call gates, task gates	unsupported	3.2.4

Feature	State	Ref
memory management - paging		
36-bit addressing (PAE, PSE-36)	unsupported	
paging	modelled	
no paging	unsupported	
page size extensions (large pages)	static	4.2.2
write-protect supervisor access	static	
accessed and dirty bits on page level	unsupported	
fatal reserved bits	modelled	
global pages	dynamic (availibility: static)	
TLBs	modelled (restricted)	4.2.3
memory management - caching		
caching options	carried along	3.2.2
memory type ranges	unsupported	
protection		
access control on segment level	modelled	4.2.4
access control on page level	modelled	4.2.2
privilege level (CPL, RPL, DPL)	modelled	4.2.4
privilege level change through call gates	unsupported	3.1
SYSENTER/SYSEXIT	unsupported	
Alignment checks	unsupported	3.2.4
interrupt and exception handling		
interrupt handler...		5.1
... without privilege level change	modelled	4.2.7
... with privilege level change	unsupported	
distinction interrupt/trap	modelled	
task gates	unsupported	
exceptions ...		
... page faults	modelled	5.1.1
... all other	carried along[1]	5.1.2
distinction fault/trap	unsupported[1]	
enabling/disabling interrupts	carried along	4.2.6
others		
task switching	unsupported	3.2.4
FPU. MMX, SSE, 3dnow	unsupported	3.2.3
debugging and perfomance monitoring	carried along	3.2.4
machine check architecture	unsupported	3.2.4

modelled fully available, use mandatory (occasionally even more restrictive than expected from the IA32 manual)

static fully available, use optional but a model parameter

dynamic fully available, use optional and part of the dynamic state (i.e. can be switched on and off)

carried along not part of the model but can be used by the program to be verified (i.e. it does not influence correctness and is therefore not fatal)

unsupported not part of the model and its use is fatal (i.e. because incorrect or simply considered outdated)

unusable not part of the model and its use is not fatal but leads to a wrong verification instead

Appendix B

IA32 Instruction Set

Instruction	Layer	Function
data transfer (conditional and unconditional)		
to/from memory	segmentation	seg_(read/write)
to/from registers	register	reg_gp_(read/write)
PUSH	stack	stack_push
POP	stack	stack_pop
PUSHA/PUAHAD	register	reg_pusha
POPA/POPAD	register	reg_popa
IN	unsupported	
OUT	unsupported	
CWD/CDQ/CBW/CWEQ	C++ model	
binary arithmetic	C++ model	
decimal arithmetic	C++ model	
logic instructions	C++ model	
shift and rotate	C++ model	
bit and byte instrunctions	C++ model	
jump (conditional and unconditional)	C++ model	
loop	C++ model	
CALL	unsupported	
RET	unsupported	
IRET w/o priv-level change	execution	exec_iret
IRET with priv-level change	unsupported	
INT/INTO w/o priv-level change	execution	exec_int
INT/INTO with priv-level change	unsupported	
BOUND	C++ model	
ENTER/LEAVE	C++ model	
string routines	C++ model	
PUSHF	register	reg_pushf
POPF	register	reg_popf
CLI	register	reg_cli
STI	register	reg_sti
other flag control instructions	unsupported	
LDS/LES/LFS/LGS/LSS	unsupported	
LEA	C++ model	
NOP	C++ model	
UD2	unsupported	
XLAT/XLATB	C++ model	
CPUID	unsupported	
MMX, SSE, SSE2, FPU instructions	unsupported	

Instruction	Layer	Function
LGDT	segmentation	seg_lgdt
SGDT	segmentation	seg_sgdt
LLDT	unsupported	
SLDT	unsupported	
LTR	unsupported	
STR	unsupported	
LIDT	execution	exec_lidt
SIDT	execution	exec_sidt
MOV segment registers	segmentation	seg_{read/write}_segment_register
MOV CR0	register	reg_{read/write}_cr0
MOV CR2	tlb	vmem_read_cr2
MOV CR3	tlb	vmem_{read/write}_pdbr
MOV CR4	register	reg_{read/write}_cr4
LMSW	unsupported	
SMSW	unsupported	
CLTS	unsupported	
ARPL	unsupported	
LAR	unsupported	
LSL	unsupported	
VERR	unsupported	
VERW	unsupported	
MOV debug registers	unsupported	
INVD	unsupported	
WBINVD	unsupported	
INVLPG	tlb	tlb_invlpg
LOCK	unsupported	
HLT	unsupported	
RSM	unsupported	
RDMSR	unsupported	
WRMSR	unsupported	
RDPMC	unsupported	
RDTSC	unsupported	

Bibliography

[Bev88] William R. Bevier. Kit: A study in operating system verification. Technical report, Computational Logic Inc., Austin, Texas, 1988.

[BJK$^+$03] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W.J. Paul. Instantiating uninterpreted functional units and memory system: functional verification of the VAMP processor. 2003.

[Bla94] William Blake. *Poems*. Everyman's Library Pocket Books. Alfred A. Knopf, Inc., New York, Toronto, 1994.

[Dau02] Matthias Daum. Entwicklung einer Implementationssprache für einen sicheren Mikrokern. Term paper, TU Dresden, 2002. in German.

[Dau03] Matthias Daum. Develoment of a Semantics Compiler for C++. Diploma thesis, TU Dresden, 2003. to be published.

[GT00] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *ICSM*, pages 131–142, 2000. Available from URL: `http://plg.uwaterloo.ca/~migod/papers/icsm00.pdf`.

[HLM$^+$03] H. Härtig, J. Löser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O Architecture for Microkernel-Based Operating Systems. Technical Report TUD-FI03-08-Juli-2003, Dresden University of Technology, Dresden, Germany, July 2003.

[Hof02] Sarah Hoffmann. Kleine Addressräume für FIASCO. Term paper, TU Dresden, 2002. in German.

[Hoh96] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, TU Dresden, August 1996. In German; with English slides. Available from URL: `http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/`.

[Hoh98] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`.

[HT03] M. Hohmuth and H. Tews. The semantics of C++ data types: Towards verifying low-level system components. In *Proceedings of Theorem Proving in Higher-Order Logics (TPHOLs), Emerging Trends*, Rom, Italy, September 2003. Accepted for publication.

[HTS02] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD–FI02–03–März 2002, Dresden University of Technology, 2002. Available from URL: `http://os.inf.tu-dresden.de/vfiasco/`.

[Int99] Intel Corp. *Intel Architecture Software Developers Manual, Volumes I-III*, 1999. add version.

[K$^+$03] Michael Klein et al. The VeriOS project, 2003. Information available from URL: `http://busserver.cs.uni-sb.de/forschung/forschung.php`.

[Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[OSRSC01a] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Language Reference*, 2001.

[OSRSC01b] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Prover Guide*, 2001.

[Tew02] Hendrik Tews. Programmverifikation und -spezifikation mit Coalgebren, 2002. Lecture notes, in German.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. *Lecture Notes in Computer Science*, 2031:299+, 2001.

70. Enough! or Too much.