Verification of Hypervisor Subroutines written in Assembler

Dissertation

zur Erlangung des Doktorgrads
der technischen Fakultät
der Albert-Ludwigs-Universität Freiburg im Breisgau
von

Stefan Maus

ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

8th September 2011

Dekan
Prof. Dr. Hans Zappe

Gutachter
Prof. Dr. Andreas Podelski
Prof. Dr. Wolfgang J. Paul

29th August 2011

Abstract

We present a methodology for the specification and verification of functional specifications of programs written in Assembler. We have evaluated the methodology in an industrial setting, the verification of the Microsoft Hypervisor.

Many industrial software projects are written in a high-level language like C. For performance reasons or for direct hardware access, some of the routines are implemented in Assembler. Our goal is the automatic modular verification of functional specifications for C programs with subroutines in Assembler. This goal entails the need for checking an Assembler procedure against its functional specification. The specification of the Assembler program is used also in the specification of the C code that calls the Assembler program as a subroutine. Therefore, we need to translate back and forth between specifications for C code and specifications for Assembler code.

The particular context of our work is the verification of the Microsoft Hypervisor where the static checker VCC is used to verify the part of the code written in C. VCC uses modular reasoning and comes with its own annotation language for functional specifications for C programs. The functional specifications for the Assembler routines in the Microsoft Hypervisor are given in the form of specifications for the C routines that call them.

In this thesis, we introduce the tool Vx86 and the corresponding methodology to verify Assembler code against functional specifications of the form described above. In our approach, we use Vx86 to translate the Assembler code into C code. We give the translation of each Assembler instruction into a short piece of C code. Some instructions of x86 are complex in that their behavior depends on the internal processor state such as flag or control registers. The high-level semantics of C does not, however, foresee such low-level featuers. In order to account for the processor state in the high-level semantics, we introduce a C structure that contains the complete processor state. The modification of this C structure during the execution of the C code simulates the modification of the internal processor state during the execution of the Assembler instructions. Since we can refer to this C structure in the annotation language, we can specify the functional correctness of Assembler routines also if the functional correctness refers to the hardware directly.

Our tool Vx86, integrated with VCC, leads to a methodology for the verification of functional specifications of programs written in Assembler and, thus, the pervasive verification of mixed programs such as the Microsoft Hypervisor which are written in C and Assembler. In this methodology, the program has to be

annotated manually with modular specifications, e.g., pre- and postconditions and loop invariants. The verification of the code against these specifications is then fully automatic. We have used the methodology and the tool Vx86 in order to verify all subroutines of the Microsoft Hypervisor which are written in Assembler.

Zusammenfassung

Wir stellen eine neue Methode zur Spezifikation und Verifikation von funktionalen Spezifikationen von Assembler-Programmen vor. Wir haben diese Methode in einem industriellen Kontext evaluiert, nämlich der Verifikation des Microsoft Hypervisor.

Viele industrielle Softwareprojekte sind in Hochsprachen wie C geschrieben. Aus Geschwindigkeitsgründen oder für direkten Hardwarezugriff werden einige Routinen in Assembler implementiert. Unser Ziel ist eine automatische, modulare Verifikation funktionaler Spezifikationen für C-Programme mit Unterroutinen in Assembler. Dies erfordert das Überprüfen einer Assembler-Prozedur gegen ihre funktionale Spezifikation. Die Spezifikation der Assembler-Prozedur wird auch benutzt für der Spezifikation des C-Codes, der die Assembler-Prozedur als Unterroutine aufruft. Dazu müssen C-Code Spezifikationen und Assembler-Code Spezifikationen in beiden Richtungen ineinander übersetzt werden.

Der besondere Kontext unserer Arbeit ist die Verifikation des Microsoft Hypervisor, wobei der *Static Checker* VCC für die Verifikation der Code Teile benutzt wird, die in C geschrieben sind. VCC ermöglicht modulare Beweise und führt eine eigene Annotationssprache für die funktionale Spezifikation von C-Programmen ein. Die funktionale Spezifikation von Assembler-Routinen im Microsoft Hypervisor werden in Form von Spezifikationen für deren Aufrufe im C-Programm bereit gestellt.

In dieser Arbeit führen wir das Werkzeug Vx86 und eine darauf beruhende Methode ein, um Assembler-Code gegen funktionale Spezifikationen der oben beschrieben Form zu verifizieren. In unserer Methode benutzen wir Vx86 um Assembler-Code in C-Code zu übersetzen. Wir präsentieren die Übersetzung jeder einzelnen Assembler-Instruktion in ein kurzes C-Code Fragment. Einige Instruktionen des x86 sind komplex in dem Sinne, dass ihr Verhalten von dem internen Prozessorzustand abhängt, wie den Flag- oder Kontrollregistern. Die Hochsprachen-Semantik von C sieht allerdings keine solchen systemnahen Features vor. Um in der Hochsprachen-Semantik auf den Prozessorzustand Bezug nehmen zu können, führen wir eine C-Struktur ein, die den kompletten Prozessorzustand abbildet. Die Veränderungen an dieser C-Struktur während der Ausführung des C-Codes simuliert die Veränderungen des internen Prozessorzustandes bei der Ausführung der Assembler-Befehle. Da wir in der Annotationssprache Bezug auf diese C-Struktur nehmen können, können wir auch die funktionale Korrektheit von Assembler-Routinen spezifizieren, wenn sich die funktionale Korrektheit direkt auf die Hardware bezieht.

Unser Werkzeug Vx86 führt, zusammen mit VCC, zu einer Methode der Verifikation von funktionalen Spezifikationen von Prozeduren, die in Assembler geschrieben sind, und führt damit zur durchgängigen Verifikation von gemischten Programmen wie dem Microsoft Hypervisor, die in C und Assembler geschrieben sind. Für diese Methode muss das Programm manuell mit modularen Spezifikationen annotiert werden, z.B. mit Vor- und Nachbedingungen oder Schleifeninvarianten. Die Verifikation des Codes gegen diese Spezifikationen erfolgt dann vollautomatisch. Wir haben die Methode und das Werkzeug Vx86 benutzt, um alle Unterroutinen des Microsoft Hypervisor, die in Assembler geschrieben sind, zu verifizieren.

Contents

Chapter 1

Introduction

In the normal software development cycle, testing is an integral part. Although people are trying to write correct programs, introducing errors in complex software is unavoidable. Finding these errors by testing is not always easy. Moreover, when errors are found and the program is changed, the testing cycle has to be performed again. Obviously, testing can only show the presence of errors, never their absence, unless all input variations are tested. The latter is only possible for very small input domains. For some projects testing is not enough, e.g., for security reasons, or it is difficult to perform, e.g., in embedded systems that have no diagnostic hardware. One possibility to prove the absence of errors is formal software verification, which involves a formal specification of the system and a mathematical proof that the implementation meets this specification.

Software projects are usually written in high-level languages like C++, Java, or C. One of the goals of these languages is the abstraction from the processor or other hardware. However, some program parts are written in Assembler to increase the performance or to directly access the hardware. The Assembler language provides only a minimal abstraction from the hardware. In our special case, we are looking at the source code of Microsoft Hypervisor, called Hyper-V, which is part of the Microsoft Server 2008. It contains about 50k lines of code, where 5k lines are Assembler code. The ultimate goal is to verify the complete system. Therefore, we require a method to describe and verify both the C code and the Assembler parts. Additionally, we need to verify the "language crossing". While a verification tool for C existed at the beginning of the project, no attention was paid to the Assembler part or "language crossing".

```
1  result = foo(a,b,c,d)
```

Figure 1.1: Example C code for a call to a function

```
   push   rax
   push   rcx
   push   rdx
4  push   r8
   push   r9
   push   r10
   push   r11
   mov    rcx, a
9  mov    rdx, b
   mov    r8, c
   mov    r9, d
   call   foo
   mov    result, rax
14 pop    r11
   pop    r10
   pop    r9
   pop    r8
   pop    rdx
19 pop    rcx
   pop    rax
```

Figure 1.2: Example Assembler implementation

1.1 Assembler

Programs are normally written in high-level languages like C. To execute these programs on a computer, they have to be compiled to Assembler for the specific platform to get an executable binary. This compilation process is complex: it has to translate the high-level statements to a number of low-level instructions; the functions need a stack; a heap has to be located somewhere in the memory, etc. The executable has much more instructions than its source code because all implicit knowledge has to be made explicit. All the abstraction of the programming language is gone at the end.

As an example, consider a simple function call. In high-level languages, subroutines are moved into functions to improve their readability and reusability. Those subroutines can then be called by a chosen name and a number of parameters, as shown in Figure 1.1. Results of such subroutines can be changes on a global state or returned values.

In Assembler, calling a function is much more complex. The Assembler developer has to

- save some registers on the stack

- put all parameter values in registers
- call the function
- read out the return value from a register
- restore registers from the stack

The complexity of the function call depends on the complexity of the parameters used. Creating subfunctions doubtlessly helps preventing to rewrite the same code over and over again, but it does not improve readability as the calls are so complicated (as shown in Figure 1.2 on the preceding page for a call to a function with 4 simple parameters). In high-level languages, those facts are abstracted away, the compiler automatically introduces additional allocations in heap or stack (this is implicit use of heap or stack), the complete stack handling, function calls, etc. High-level languages were introduced to get rid of that low-level burden.

Despite these disadvantages, there are reasons to use Assembler. The presented work is part of a verification project by Microsoft. Microsoft wishes for the verification of the Microsoft Hypervisor that is part of Windows Server 2008. This software has to use low-level instructions that are not available in C. For example the virtualization instructions of the x86 architecture are not included in C compilers, because they should never be accessed from a normal program. Besides such special instructions, the C compiler may also produce sub-optimal executable code. In Assembler, the programmer has direct influence on the instructions used and can make optimizations for cache lines, pipelining of the processor, etc. Some high performance optimizations are only possible thanks to special background knowledge. For example a copy routine could be more optimized if it is known that the amount of bytes is always 4096 and it is always page aligned. An arbitrary copy routine (for example by a library) has to care about unaligned access, arbitrary size and so on. Therefore, the Assembler versions can be a lot faster. This is useful for code sitting in a critical path. The usual way to combine Assembler with high-level languages, is to implement functions in Assembler code and call those Assembler functions from ordinary C code. To execute the program, everything is translated to machine code separetly.

For the verification it may seem benificial to use the compiler to translate the high-level code in Assembler and do the verification on the Assembler code only. However, most of the existing verification tools are written for high-level languages, at least C language, see [32, 46, 55]. These high-level languages abstract from the hardware, e.g., the memory layout, which is also useful for verification tools. In Assembler there are only processor registers and memory.

The programmer or the compiler has to explicitly allocate memory on the stack and compute the address. This can introduce bugs that are not possible in high-level languages, e.g., stack frame corruption. A verification tool for Assembler needs to check that the accessed memory is writeable and that the stack pointer is restored after the execution of the method. Moreover, verifiers for high-level languages can make assumptions like type safety. These assumptions are no longer valid for programs in Assembler code, since Assembler does not support data types and accesses pointers and integers in a polymorphic way. Verification tools usually assume that this compilation process is correct and only verify the high-level program code against the programming language semantics. With the translation approach the verifier would have to check these assumptions explicitly.

1.2 Microsoft Hypervisor

The Microsoft Hypervisor is a thin layer of software written in C and Assembler that sits directly on x64 hardware, turning a real multi-processor (MP) x64 machine into a number of MP x64 virtual machines (VMs). These VMs provide additional machine instructions (hypercalls) to create and manage VMs, hardware resources, and inter-VM communication. VMs are viewed as a key enabling technology for a variety of services, such as server consolidation, sandboxing of device drivers, testing, running multiple OSs on a hardware machine, live VM migration, snapshotting/recovery, and high availability. Moreover, it provides such functionality in an OS-neutral way, with a trusted computing base 2-3 orders of magnitude smaller than that of a typical commercial operating system.

Intel and AMD have developed hardware support for hypervisor systems. For example, they can switch to the hypervisor if hardware interrupts occur and provide multi-stage page tables. Operating systems (OS) usually run on highest privilege level and can use direct access to the hardware. Additional hardware support allows to run the OS on the original privilege level by running a hypervisor in an even higher level. For example, they can switch to the hypervisor if hardware interrupts occur and provide multi-stage page tables so that the operating systems do not see that they are working in translated mode. Unfortunately, both Intel and AMD have their own virtualization instructions. Since the AMD instruction set is older and the implementation in the hypervisor thus (hopefully) has less errors in it, we decided to first support AMD. In future work, there could also be an implementation of the Intel virtualization hardware. Both hardware types can be supported at the same time because they have different instruction names and different processor states. Compared to standard Assembler instructions, the virtualization instructions are very com-

plex. They are used for context switches between the hypervisor (host system) and the operating systems (guest systems). A typical scenario for a context switch consists of the following sequence of operations:

1. save the host state
2. load the guest state
3. run the guest
4. save the guest state
5. load the host state.

Properties about those virtualization instructions include facts like "the state of the host after the restoring process is the same as it was at the point of the saving". Such a property does not only include the values of all registers (visible and invisible), but also the stack that is administrated by the processor. If the stack has changed (either the place or the content) then the host will have a completely different state. Properties involving virtualization typically range over many registers and memory locations. Additionally, the processor state is usually available twice: once for the host system and once for the guest system. The verification tool then has to scale well to handle such complex functions and specifications, and we have seen verification times for virtualization function degrade (see below).

On the other hand, several functions in the Microsoft Hypervisor are only used for optimization reasons. The specifications for those functions are not too complicated. However, looking at an optimized implementation is often scary; algorithms are optimized for filling the pipeline most efficiently, to exploit branch prediction and caching. Verifiers however are good at keeping track of detail and so these algorithms are a great target for modern verification technology.

1.3 VCC and Boogie

VCC[16, 20, 18, 15, 17] (the verifying C compiler from Microsoft) is one from a series of verification tools. The first in the row was Spec#[9, 8, 47, 48, 50, 1]. Spec# was an extension of C# to enable specification and verification. It is integrated into Visual Studio, so that the verification can be made from the integrated development environment (IDE). This makes the specification process a lot easier, because the verification can be made on the fly.

Spec# consists of three parts (see also Figure 1.3 on the following page):

1. compiler, which compiles C# with specification to BoogiePL

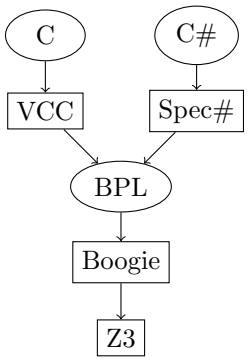

Figure 1.3: Workflow of the Spec# verification process

2. Boogie, which tries to verify the BoogiePL code by generating a verification condition and calling an SMT solver

3. Z3, which is used as automatic first-order SMT solver to handle requests from Boogie

VCC is developed to replace the C# translation to Boogie to enable a similar verification for C programs. It translates C programs augmented with specification into a BoogiePL program.

The C annotations are mostly similar to ESC/Java[33, 60, 65, 13] and Spec#. Modular reasoning is used for verification, which means a function by function verification. Function calls are exchanged by using their specification. Thus, only a small part of the code base has to be verified at a time. In contrast to ESC/Java and Spec#, VCC uses sound verification methods of low-level concurrent C code.

Annotations are implemented as Macros. This means that the annotations can be placed directly into the C code. For compiling of the program with a normal compiler, there exists a header file that allows the preprocessor to throw the annotations away, while the verification tool can make use of them.

It is very important that there is no data flow from the annotations into the normal code. In the annotations, only special variables can be written, they are placed in so called *ghost memory*. This memory is disjoint with normal C memory, thus normal C instructions cannot access it and read values out of it. VCC introduces also additional proof obligations to the manually inserted ones from the source file. Those will be checked for correctness, for example if *writes* clauses were given correctly.

BoogiePL is an intermediate language with just 10 statements, and it can be extended by a rich type system. The 10 statements are:

- assert, which means a proof obligation to the verifier
- assume, representing an assumption the verifier can make
- havoc, which means a non-deterministic choice for sets of variables
- assignment, which assigns a value to a variable
- call, which represents a function call to another Boogie function.
- if, which allows a conditional execution, depending on a boolean expression or non-deterministic choice
- while, which represents a loop where the execution is either depending on a boolean expression or non-deterministic choice
- break, which stops the execution of the closest enclosing loop
- return, which terminates any execution reaching this statement
- goto, which is a non-deterministic jump to one of the given labels

Note that labels are no normal statements in BoogiePL. BoogiePL programs are divided in so called *basic blocks*. Basic blocks start always with a label, followed by statements and are ending either with *return* or *goto*. For a complete description see also [7, 45, 49].

BoogiePL is not intended as a new programming language for execution, but only for verification purposes. This intermediate step has the advantage that the verification tool Boogie can be used for different languages. Otherwise, the functionality of this tool would have to be rebuilt for every verification tool from scratch. The tool Boogie is a verification condition generator. It produces one or more (SMT-) formulae. These formulae are then checked by a theorem prover such as Simplify or Z3, see [26, 27, 28]. Z3 can handle more complicated formulae than Simplify and is much faster.

1.3.1 Specification and Annotation with VCC

The source code needs to be annotated for the verification with VCC. There are implicit verification tasks for VCC (such as arithmetic overflows). But most of the verification tasks are given as explicit annotations.

Specification Variable The developer of the specification can introduce new variables. They can, for example, hold knowledge that is implicit in the source code. The specification variables are located in a different memory space in the VCC memory model. Besides the 64-Bit address space for normal variables,

there exists an extra 64-Bit address space for the specification variables. This means the specification variables cannot alias with normal variables. VCC also checks that there is no data flow from specification variables back to normal variables, since this would change the semantics of the program when compiled with an ordinary compiler. If VCC detects such data flow, it stops with an error message.

Expression Expressions are used in the different annotations. Expressions consist of boolean C expressions (first order logic). They can refer to source code variables as well as specification variables. Besides normal C arithmetic and logic, there exist some specification functions. They introduce, for example, all quantifiers, shortcuts for memory regions and rights for memory regions.

Writes Clause Writes Clauses specify all memory regions a function can write to. For every memory access, VCC checks whether the affected memory region is writable, i.e. it was included in the Writes Clause. If it was not included, VCC stops with an error message. All memory regions specified in the Writes Clause are set non-deterministicaly to an arbitrary value. This makes sure that underspecification can be used in a sound way.

Precondition A precondition limits the possible inputs for a function. If no precondition is given, all input parameters of a function can be chosen non-deterministically. To allow for example verification of arithmetic expressions, the possible input has to be limited. When a function is called, VCC verifies that the precondition is met by the calling function.

Postcondition A postcondition limits the possible outputs of a function. They can refer to the return value of a function but also to memory included in the Writes Clause. If no postcondition is given, the return value and all memory included in the Writes Caluse are set to arbitrary values. When a function is called, VCC uses the postcondition and the Writes Clause of the called function to verify the calling function.

Assertion VCC checks that expressions given by Assertions evaluate to true at the given point. If there is a possible valuation for the input paramters of function that cause the assertion expression to be evaluated to false, VCC gives an error message. Assertions can have different purposes. They can help the developer to verify values of variables at a given point in the source code. They can also help the verifier to link given information that is not given explicit but is implicit knowledge of the developer. Assertions (although they evaluate to

true if not given explicitly) can also speed up verification, because they can help verification tools to find the important information of a verification task.

Assumption These clauses specify conditions that are assumed to hold at a given point. The assumptions are added to the collection fo facts that the verifier uses to analyze the code. They should be used with care because they can easily destroy soundness of the verification. They can be used for debugging purpose or to give information explicitly that can not be inferred by VCC.

Data Structure Invariant Data Structure Invariants limit the possible values of the fields of data structures. VCC augments data structures with special fields. The fields are used for the ownership and virtual locking mechanisms. The locking is virtual because it is not given in source code. The locking mechanism is described in Section 4.3.

Specification Function Specification functions can be used like normal C functions. They are not given by a function body but by pre- and postcondition pairs. This means they are given as annotation expressions. Annotations cannot write into source code variables but only to specification variables. This means specification functions can only change the specification status. Calling source code functions out of specification annotations would lead to a change of the program status that would make the semantics of the verified program and the compiled program incoherent. This means that only calls to specification functions are allowed in annotations.

1.4 Contribution of this thesis

Recently, the verification of systems C code made much progress, see [66, 51, 21, 58]. Industrial systems code often consists of C code combined with subroutines implemented in Assembler. For existing verification tools, such a combination of languages is not manageable. We present an approach that allows verification of such so called mixed programs. For the presented verification task, we are interested in correctness of function specifications.

We want to verify memory safety and functional properties for mixed programs from an industrial setting. Thus, we need a machine readable semantics for Assembler. Existing Hoare logics for Assembler are complete, but they are too unwieldy and thus not feasible for our purpose. The level of abstraction must be precise enough to reason about the program correctness. On the other hand, it must be abstract enough to allow reasoning in realistic time. Our solution is a semantics for a low-level programming language (Assembler) in a

high-level language (C) (presented in Section 3.4), together with a processor model (presented in Section 3.3). Thus we can reuse verification tools for the high-level programming language (described in Section 1.3).

Like in all mixed language problems, the different type and variable systems prevent a common annotation language. Our approach uses two kinds of assertions. The first one is for the call of Assembler functions from C programs. These assertions are expressed in terms of C variables and serve as function contracts. From C point of view, the function can be seen as $\{p\}f(e)\{q\}$, where p represents the precondition, q represents the postcondition and $f(e)$ is the call to the function implemented in Assembler. p and q can be used for modular reasoning in the high-level program. The second one is for Assembler programs implementing the function. In fact, our assertions are expressions that refer to C variables for a new program, $\{p'\}simulated\_body\{q'\}$, where p' and q' represent the corresponding pre- and postconditions of the C description and *simulated_body* simulates the original Assembler program. p' and q' are evaluated in a C state (called abstract processor state), which corresponds to the processor state in a simulation. The translation of the pre- and postconditions is completely automated (described in Chapter 4). The *simulated_body* is expressed with the help of the introduced semantics and is a translation of the original Assembler implementation (shown in Chapter 2). With the same method, we can also handle the call of C functions out of Assembler programs. The assertion pairs are then translated in the other direction, where modular reasoning is used in the Assembler program. The C code verification is used to show that the C implementation meets the specification.

As case study we use the Microsoft Hypervisor that is already shipped. We have developed specifications of the Assembler functions. Our approach was used to verify that the specifications are met by the implementations and also to verify C programs calling them. This shows the usability in a real scenario with mixed programs, not only small examples. Because of copyrights, we only present a function that does not fall under the non-disclosure agreement and a simple example similar to a function in the Hypervisor in Chapter 5.

1.5 Limitations of the Approach

Our focus is on handwritten Assembler code. This means on one side more special instructions that are not used in translated code resulting from compilers. On the other side there are no optimizations that introduce difficult control flow. Assembler code written by developers usually has a reducible control flow because they are thinking in structures like loops. Assembler allows irreducible control flow but this is only used by compilers for optimizations.

Our approach considers Macro Assembler instead of pure Assembler. The advantage is the presence of labels in the code. This means we do not have to deal with addresses but Assembler jumps do use labels as goal. Dealing with addresses is difficult because instruction size and other address related information is introduced by the compiler. The addresses also depend on optimizations made by the compiler.

For the instructions, we are able to deal with general purpose instructions (like arithmetic operations and stack operations). We have also implemented special instructions like *cpuid* or *hlt*. We have not implemented instructions from the floating point or the multimedia extension units, because they are not used in the Microsoft Hypervisor. This is not a restriction of the approach, they can be added if needed.

We are not able to deal with instructions that access memory from another thread. The reason is that we assume that the address translation works in the same virtual memory area. This assumption would restrict the virtualization in the hypervisor. For guest execution, there exists an extra instruction. Our implementation of this instruction assumes that the guest is working in isolation and can not change the memory of the hypervisor. This assumption has to be discharged separated from our work. A complex proof including address translation of the processor and multiple page tables is necessary as well as arguing about the memory allocation for the guest systems. Context switches of different threads are also not possible with our approach because the address spaces of both threads would have to be used.

1.6 Related Work

In the following subsections, related work is presented for both: (1) verification of Assembler programs and (2) verification of micro kernels. For this work, both are important because our practical work was done on Microsoft Hypervisor that can be seen as a special micro kernel.

1.6.1 Assembler Verification

For the verification, low-level languages like Assembler are much more complicated than high-level languages like C. The complexity of the language comes from different facts. First of all, the program is unstructured. There exists no native loop but conditional jumps and labels which together represent the loop. Program verification uses the structure of a program to introduce invariants or frame conditions. If there does not exist a structure most of the existing verification tools do not work properly. Another fact that increases complexity is

the polymorphic use of numbers and bitvectors. Although this is also part of high-level languages, it is usually neglected in program verification. Assembly instructions always have this polymorphic use of registers. Bit shifts and logical Bit operations are widely used in Assembler and cannot be neglected at all.

Besides the level of complexity coming from the language itself, there is also a difference in the level of complexity coming from the specification and the verification properties used. Depending on the demands of the domain there can be type safety checks, memory safety checks (memory must be allocated, out of bounds checks for arrays etc.) or pre-/postcondition pairs representing the functional properties of a program. Invariants can either be found automatically or have to be given as part of the specification. More detailed verification also means more computational effort. The increase in verification time can be huge depending on the programs to check.

Existing Assembler verification programs usually have simple specifications to reduce the complexity of the system. Boyer and Yu [12] verified MC68020 assembler programs. They used the theorem prover Nqthm as their verification tool; they formalized the MC68020 as Nqthm theories, thus in effect giving an interpreter for the processor; assembler programs are then translated into expressions over this special logic. Vx86 differs in various dimensions from this early work, Vx86 works on the much more complex x86 architecture, Vx86 incorporates contracts (including framing) into the Assembler, Vx86 uses an automatic theorem prover (ATP), Vx86 has been used to verify parts of a real industrial strength operating system.

Another approach to guarantee that Assembler programs are safe are Typed Assembly Languages (TAL) [19, 53, 37, 14, 38, 67]. TALs are low-level, statically typed target languages. TALs guarantee type safety, which typically implies memory safety. However, TALs do not guarantee arithmetic safety, call safety, interrupt safety or other functional properties. Furthermore, TALs are often idealistic Assembler languages, they are only used as target languages for compilers; as such they do not deal with the whole instruction set of the processor. We, however, also have to deal with instructions like *HLT* or *CPUID* and the virtualization instruction set.

Proof carrying code (PCC) has a similar goal [56, 29, 31, 30, 57]. Instead of defining type safety for Assembler code, PCC adds proofs to untrusted Assembler files, which establish certain properties. The receiver of the untrusted code is then able to use a simple and fast proof validator to check that the proof is valid and hence the untrusted code is safe to execute. Like TAL, PCC focuses on memory safety; it is not a general verification architecture. There are no functional contracts as needed for real world applications like the Microsoft Hypervisor. In personal communication with the authors we found out

that using our approach could be interesting for PCC. Contracts are introduced into the binary code by the PCC compiler. There exists a tool to discharge those contracts later on. The current tool uses a weak proof system. Using our approach, the proof system is much stronger and could discharge more difficult contracts. On the other hand, it is not clear whether our approach can handle the control flow resulting from the compilers. Optimizations of compilers can introduce irreducible control flow. Our translation approach would not be able to deal with that.

Verification on source code and verification on binary code is also a question of trust. Verification on the binary level means that all compiler optimizations are also verified. The disadvantage is that it is not clear how to verify more difficult contracts on this level. Annotations from the high-level language do not refer to the binary representation and thus would have to be compiled in a difficult way. Verification on source code level means to assume a correct working compiler. This is the case for all software model checkers that are discharging annotations on source code. As adavantage they can deal with more difficult annotations refering directly to high-level constructs. Model checkers are already used on the C source code of the Microsoft Hypervisor.

Besides PCC and TAL, there exists a lot of verification on binary code. They all share the idea of verification after the compiler. They also share the limitation of very weak proof systems. They share the lack of modular reasoning for high-level languages, which is not needed if the whole program is compiled to one binary. They do not support functional specifications. There exists for example an extension of Codesurfer, called Codesurfer/x86, presented in [3, 6, 63, 4, 5, 35, 52, 62]

1.6.2 Hypervisor and Micro Kernel Verification

Up to our knowledge there does not exist a project aiming for the verification of a hypervisor. But a hypervisor can be seen as a kind of micro-kernel, extended with virtualization capabilities. Various projects aim for micro-kernel verification. The CLI stack project in the late 1980s [11] was the first project focusing on the pervasive verification of computer systems. In total, the system consisted of four levels: starting from a verified FM 8502 microprocessor via a simple Assembler language up to a verified operating system.

Verisoft [34, 24, 2, 22] is in spirit similar to the CLI project. Verisoft developed machine-models for Assembler, small step and big step semantics for more abstract programming languages, and programs for devices, kernels, operating systems and applications. However, the Verisoft project only dealt with idealistic processors, inline Assembler, and OS. The L4.verified project

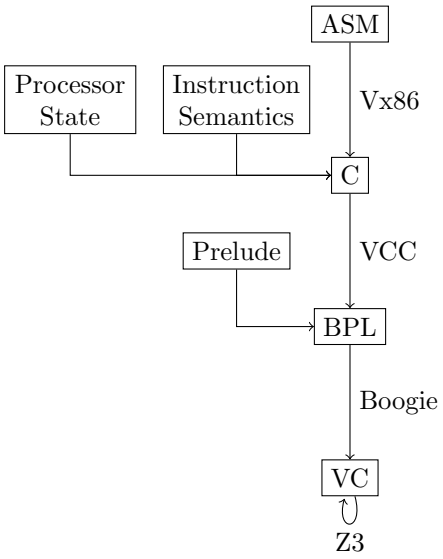

Figure 1.4: Data flow in our verification process

[39, 44, 43, 42, 41] aims at the formal verification of an industrial strength implementation of an L4 micro-kernel, which is highly optimized for the ARM platform. While the L4.verified project tries to do low-level C verification, it has – to the best of our knowledge – not yet started verifying Assembler code. Verisoft and the L4.verified project use the same verification technology. Both systems use the interactive theorem prover Isabelle and a Hoare calculus embedded in Isabelle [64] to verify properties of the micro-kernel. The automation was slightly improved with the integration of automatic tools that can verify parts of the proof obligations [23]. However, the resulting system does not yet achieve the automation level we achieved.

1.7 Structure of the Thesis

Our approach is to use existing C tools for Assembler verification. An overview on how the tool chain will work can be seen in Figure 1.4.

First of all, the Assembler files have to be translated into a format parsable by a C compiler. This is done by a syntax translation from Assembler to C, which is described in Chapter 2. This translation is implemented in the functional programming language *F#* (pronounced "F sharp"). It translates each Assembler instruction into a call of a function. The meaning of the instructions is provided by separate files that contain a C implementation for each function. These functions access the internal processor registers as C variables (the proces-

sor model). In Chapter 3, we present the processor model and show exemplary implementations of the Assembler instructions. The separate files are included in the automatically generated code. After this step, a C compiler would be able to compile the "Assembler" file (it is now a C file with a processor simulator) into a binary or executable. Executing this binary would not yield the same result as executing the original Assembler code, but it would only change the virtual processor state. This is also described in the chapter mentioned.

The previous translation enables us to verify pure Assembler code. However, in mixed programs, where some parts are written in C language and other in Assembler there is another important need. For the transition from C to Assembler the calling conventions have to be taken into account and the function specification of the functions called from C and written in Assembler have to be translated to the virtual processor model. This is discussed in Chapter 4.

We evaluate our approach on example code in Chapter 5. Due to copyright restrictions, we only present a function that does not fall under the non-disclosure agreement and a simple example similar to a function in the Hypervisor.

In Chapter 6, we conclude what was achieved. Furthermore, we mention future work: parts that are not yet finished and also some ideas in which direction this work could be extended. A short overview of the history of the x86 architecture is given in Appendix A. It should show the interested reader why some of the (sometimes) strange instructions exist.

Chapter 2

Translation of Assembler into C Programs

For our verification approach, we have to translate the Assembler input into a C compliant file for the verification tool. The translation has two main parts: A syntactical translation of the Assembler instructions to C function calls, and a translation of the operands. An operand can be a processor register or memory access. A memory access is translated to a corresponding pointer dereference, possibly using pointer arithmetic. A processor register is translated to an access of the virtual processor state. The syntactical translation consists of three phases: (1) a parser that reads in an assembler input file and produces a syntax tree, (2) an analysis phase that translates the operands and detects loops, and (3) a pretty printer that produces legal C code out of the syntax tree.

In the first phase, the parser generates a syntax tree that abstracts unnecessary information like empty lines and comments. Although this syntax tree is generated out of an Assembler program, we interpret the tree as a C program, i.e. the instructions are interpreted as function calls. In the second phase tree transformers are used to change memory accesses into pointer arithmetic, change back jumps into while-loops, and resolve other Assembler constructs that are not available in C. In the third phase, the pretty printer generates a C program from the syntax tree. It introduces type casts and parenthesis to make the output C compliant code.

Our current implementation does not support all Assembler constructs. For example, we do not support all prefixes (like **rep**) or floating point operations. This is because they are not used in Microsoft Hypervisor. The missing parts can be easily implemented in the future if our approach is extended to programs that need them. Our main goal at the moment is to verify the Assembler code

of the Microsoft Hypervisor.

2.1 Microsoft Macro Assembler Notation

We are using the Assembler syntax of the Microsoft Assembler that follows the Intel notation. A line in the Assembler file consists of an optional label, followed by an optional instruction and an optional comment, i.e., a line can also be empty. Most instructions are of the form *mnemonic destination*, *source* where *mnemonic* is the name of the instruction and *destination* and *source* are the operands. An operand can be a register, a memory access, or a constant expression. The *destination* operand can serve as input and output, while the *source* operand is always input. Some instructions have a different number of arguments, e.g., a jump instruction has a label as its single operand. A register operand is the name of a processor register, see Chapter 3. A memory operand can also involve a limited amount of pointer arithmetic, see Section 2.3.2).

For readers familiar with GNU Assembler syntax that is derived from AT&T notation, a short description of differences should help in understanding the Assembler examples. The general shape of the AT&T instruction format is *mnemonic source*, *destination*, whereas the Intel syntax swaps the operands. The register names in AT&T syntax are prefixed by a "%" sign, which is omitted in Intel syntax. If we want to store a copy of the value from register "rax" into the register "rbx", one would write in AT&T notation *mov* $\%rax, \%rbx$, whereas in Intel notation this looks like *mov* rbx, rax. Literal values in Microsoft syntax are written as decimal numbers or optionally as hexadecimal numbers followed by an "h", without the prefix "$" that is used in AT&T syntax. The number "$0x10" is written in Intel notation as "10h".

2.2 Parser

We use a parser generator to not invent everything from scratch. A parser generator takes as input a grammar, e.g. in Backus–Naur form, and produces a parser. Figure 2.1 on the following page presents the (simplified) grammar used for Vx86. We let the preprocessor of the Microsoft Macro Assembler run before the parsing process starts. The parser takes the preprocessed file as input, which means that macros from included files are inlined. Thus, we do not have to assemble the information from multiple Assembler files. The preprocessor also catches illegal code and produces appropriate error messages. For example, it will detect, if an assembler instruction is used with an illegal prefix (the supported combinations are described in the manuals). Also not

⟨digit⟩→ **0-9**
⟨hdigit⟩→ **0-9**|**a-f**|**A-F**
⟨letter⟩→ **a-z**|**A-Z**|_
STRING→ (⟨letter⟩| ⟨digit⟩)+
NL→ **\n**|**\r \n**
REQUIRES→ **;ˆ requires (** STRING **)**
ENSURES→ **;ˆ ensures (** STRING **)**
WRITES→ **;ˆ writes (** STRING **)**
SPEC→ **;ˆ spec (** STRING **)**
INVARIANT→ **;ˆ invariant (** STRING **)**
ASSERT→ **;ˆ assert (** STRING **)**
ASSUME→ **;ˆ assume (** STRING **)**

⟨Prog⟩→ STRING **equ** ⟨Para⟩ NL ⟨Prog⟩| **;** NL ⟨Prog⟩|
. altentry STRING NL ⟨Prog⟩|
REQUIRES NL ⟨FunSpec⟩ STRING **proc** NL ⟨StmtList⟩ **eproc** NL ⟨Prog⟩|
ENSURES NL ⟨FunSpec⟩ STRING **proc** NL ⟨StmtList⟩ **eproc** NL ⟨Prog⟩|
WRITES NL ⟨FunSpec⟩ STRING **proc** NL ⟨StmtList⟩ **eproc** NL ⟨Prog⟩|
STRING **proc** NL ⟨StmtList⟩ **eproc** NL ⟨Prog⟩|
extern STRING **:** STRING NL ⟨Prog⟩| **extern** STRING **: proc** NL ⟨Prog⟩|
STRING **qword** ⟨Para⟩ NL ⟨Prog⟩| NL ⟨Prog⟩| EOF

⟨FunSpec⟩→ NL ⟨FunSpec⟩| REQUIRES NL ⟨FunSpec⟩|
ENSURES NL ⟨FunSpec⟩| WRITES NL ⟨FunSpec⟩

⟨Expr⟩→ ⟨StmtList⟩

⟨StmtList⟩→ ⟨Statement⟩ NL| ⟨Statement⟩ NL ⟨StmtList⟩

⟨Statement⟩→ | ⟨Instruction⟩| ⟨Label⟩| ⟨Label⟩ ⟨Instruction⟩| SPEC| ASSERT|
INVARIANT| ASSUME

⟨Instruction⟩→ **lock** ⟨Instruction⟩| STRING| STRING ⟨Para⟩|
STRING ⟨Para⟩ **,** ⟨Para⟩| STRING **near** ⟨Parameter⟩

⟨Label⟩→ STRING **:**|**@:**

⟨Para⟩→ STRING| **@b**| **@f**| INT64| [⟨Para⟩]|
byte ptr [⟨Para⟩]| **word ptr** [⟨Para⟩]|
dword ptr [⟨Para⟩]| **qword ptr** [⟨Para⟩]|
byte ptr ⟨Para⟩ [⟨Para⟩]| **word ptr** ⟨Para⟩ [⟨Para⟩]|
dword ptr ⟨Para⟩ [⟨Para⟩]| **qword ptr** ⟨Para⟩ [⟨Para⟩]|
⟨Para⟩ **/** ⟨Para⟩|⟨Para⟩ ***** ⟨Para⟩| ⟨Para⟩ **+** ⟨Para⟩|⟨Para⟩ **-** ⟨Para⟩|
(⟨Para⟩ **)**|**NOT** ⟨Para⟩| Para [Para]|
rax|**rbx**|**rcx**|**rdx**|**rdi**|**rsi**|**rbp**|**rsp**|**r8-r15**|**cr0-cr8**|
eax|**ebx**|**ecx**|**edx**|**edi**|**esi**|**ebp**|**esp**|**r8d-r15d**|
ax|**bx**|**cx**|**dx**|**di**|**si**|**bp**|**sp**|**r8b-r15b**|
al|**bl**|**cl**|**dl**|**dil**|**sil**|**bpl**|**spl**|**r8l-r15l**|
cs|**ds**|**es**|**fs**|**gs**|**ss**|**dr0-dr7**|**xmm0-xmm15**

Figure 2.1: Grammar of x86 Assembler in Backus–Naur Form

every operand can be used with every instruction, for example, it is not allowed to copy from a debug register into another debug register. We do not need to check for these errors because the preprocessor already reports them. The given grammar from Figure 2.1 on the preceding page is not literally the one we use for the parser generation. For example, the constructs that are ignored by our tools like comments, procedure headers, etc. are missing in the grammar. In our real implementation, the corresponding grammar rules are included. They make the grammar huge and do not help in understanding the approach or the grammar.

Assembler files include information that is not necessary for the verification of functionality. For example, most of the functions have an artificial data block that makes sure the alignment is best for performance. Those performance issues are not important for our verification purpose and can be ignored. Assembler also has some constructs that are not available in other languages. For example, a function can have multiple entry points. So one function can have an entry point at the beginning and an entry point with a different name at a later point of the code. A developer can then call one of the two versions by referring to either of the entry points by their name. Fortunately, in the Microsoft Hypervisor such things are not used. If they occurred, one would have to copy the body twice in the C code, because C does not know any construct that has such a behavior.

Mainly, the Assembler file consists of constant definitions, annotations, function declarations (also external C functions), function bodies, and ignorable things. A function body consists of lines which can have labels, instructions, operands, and comments. Labels and comments are optional. All instructions have a fixed number of operands. The number of operands for each instruction can be found in the Intel instruction manuals.

The VCC function specifications, i.e., the pre- and postconditions as well as writes clauses, are defined between the parameters and an optional function body. In the Assembler files of the Hypervisor, the function headers are introduced by macros (also introducing framing), and they are expanded in a preprocessing step. The definition of the function specification cannot be placed at the same position as in the C version. Instead it should appear before the start of a function. There can be additional comments, but no constant definition may be written between the annotations and the function header.

2.3 Syntax Tree Transformer

To translate a language into another one can use a common syntax tree that supports the constructs of both languages. Syntax tree transformers can then be used to change the constructs that are specific to the input language into

```
int i = 10;
while (i > 0)
invariant (i >= 0)
{
        i--;
}
```

Figure 2.2: Example for a C loop with a simple loop invariant

semantically corresponding constructs of the output language. For some pairs of languages, such a translation can be easily developed. An example is a syntax translation from Java to C++. Mainly libraries are different for those languages. Some of the details have to be adapted. But most part of the languages is the same, as well syntactically as semantically. If (like in operating systems) no libraries are used and the languages are not too far away (like functional languages and imperative languages, object oriented languages etc.), the translation can be done with a syntax translation where some details have to be adapted. A change for example from functional languages to imperative languages cannot be done in such a simple way, because many constructs of one language result in very complex constructs in the other language.

Our abstract syntax tree serves as an intermediate representation for the translation from Assembler to C. The syntax tree produced by the parser still contains Assembler constructs that have no direct correspondance in C. The task of the syntax tree transformer is to translate these constructs to C.

The syntax tree is much simpler than the original Assembler file. It consists of constant definitions (which map a name to a number), variable definitions (which are real C variables), function declarations (which can be C or Assembler functions that are not located in the current file), and function definitions. A function definition contains a function body consisting of a list of statement. A statement is an annotation, a label, an instruction, a jump, a break, or a loop statement. An instruction contains a function name (the Assembler mnemonic) and a list of operands. The loop statements, introduced by a loop detection algorithm, have no direct correspondance in Assembler.

2.3.1 Loop Detection

One difference between Assembler and C is the way loop invariants are given. C annotations usually have loop invariants between the loop statement and the loop body, see Figure 2.2. Assembler has no native loop statement. The corresponding construct would be a label and a conditional jump to this label later in the code, see Figure 2.3. However, the C verifier does not allow invariants

```
    mov rax, 10;
LoopStart:
;^ invariant (rax >= 0)
    dec rax
    jnz LoopStart
```

Figure 2.3: Example for an Assembler loop with a simple loop invariant

```
    mov (rax, 10);
LoopStart:
    while (1)
    invariant (rax >= 0)
    {
        dec (rax);
        if ( rflags.zf == 0) break;
    }
```

Figure 2.4: The translation of the Assembler example from Figure 2.3

attached at labels. Thus, we introduce an artificial loop. The label is now not only a label, but also the place for a loop entry. Now, we can annotate the loop with the given invariant.

The loop detection is very simple. For every jump instruction it is checked, whether the label has appeared earlier in the code. If that is the case, we introduce a loop that starts at the label and extends up to the jump instruction. If the jump is conditional, then it is replaced by a break for the loop with the negation of the condition, see Figure 2.4. This simple loop detection was sufficient for all Assembler function that occur in the Microsoft Hypervisor code.

2.3.2 Memory Operands

Memory access in Assembler is done with the help of an address stored in a register. In C, this could be seen as a pointer with the address given by the value of the register. This memory is then read or written, the counterpart in C is a pointer dereference. A problem here is the pointer arithmetic that differs between Assembler and C. Some memory accesses do not just access the address given by a register, but add offsets given by a constant to this address. This does not seem to be very difficult at the first sight, but in reality this causes a lot of trouble. As an example, consider the memory operand $16[rax]$ in Assembler notation. Naively one may translate it to something like $*((uint64*)rax + 16)$, where $uint64$ would be the type for a 64-Bit number. However, this leads to the wrong address. In Assembler, the address would be the value stored in $rax + 16$. In C, the address would be the pointer stored in $rax + 8$ times 16

(where 8 is the size of a $uint64$ in bytes). Having some parenthesis around the computation would make the C program right: $*(uint64*)(rax+16)$. But then, the verification tool used was not able to detect a relation between specified memory locations in pre- and postconditions, and the accessed memory. In VCC, pointer arithmetic is only handled correctly if the addition is performed on pointers not on integer values. Therefore, rax has to be cast to a pointer before the addition can happen. This means, we have to adapt the constants in the computation. This again means that we have to analyze the syntax tree to detect the width of memory access and then divide by this number. In our example, $16[rax]$ would translate to $*((uint64*)rax+2)$. Although it does no longer look like the same address, it is correct now. Such casts are often introduced automatically in the transformer.

2.4 Pretty Printer

The pretty printer is a recursive walker that produces a string for every element in the syntax tree. For every node in the tree it outputs the corresponding C code. For example, for a loop statement it outputs the while header and calls itself recursively on the loop body. Similarly it handles the other constructs like labels, jumps, breaks and instructions.

An instruction in Assembler consists of an opcode and a number of optional operands. For our C representation, all instructions are translated to function calls. The number of parameters depends on the instruction. The instructions do not have a return parameter, they are functions from a global processor state into a new global processor state. To make the printer as simple as possible, we do not analyse the instructions themselves, but treat them as opaque strings. Illegal instructions are not checked, but cannot slip through and do any harm. First of all, an illegal instruction would be translated into a call to an undeclared function in the C code. This means that VCC complains about an undefined function. But even this error message should not occur, because the preprocessor would detect them before the translation.

In Assembler, there are also prefixes. Prefixes are used to switch certain processor paths. The only prefix that is used in the Microsoft Hypervisor code base is the **lock** prefix, that disables parallel updates to the same memory address. In a single threaded environment, this does not change the meaning of the instruction. Only caching algorithms are affected by this prefix. For concurrent programs, this prefix only restricts the behavior further. A locked instruction means that there is no interruption of the execution and no parallel instruction can access the same memory until this instruction has finished. This means, the possible schedulings of threads are limited by the prefix. The C

verifier proves the program correct for arbitrary schedulings. This means that if the C verifier can prove the program without the lock prefix correct, the program with the lock prefix is also correct.

After the syntax translation, there is still no semantics defined for the translated Assembler instructions. Even the registers are not defined in this step. Those are part of the processor model, described in Chapter 3. The processor model has to be included in C style to the translation to feed it into a C compiler or verification tool. Therefore, the processor model is implemented like a simulator, simulating the "Assembler" instructions (they are now C function calls) on global variables (representing the original processor registers).

Chapter 3

Abstract State for x86 Processor

Assembler is a very low-level language. It is close to the machine instruction set. To give its semantics, it is therefore necessary to understand the processor in detail. Nevertheless, the machine model does not have to include all details for this purpose. For example most of the cache architecture does not show up in our model. The caches should not be visible to the programmer, it only affects the speed of the machine and speed is not handled by our tool. On the other hand, it is not enough to simply represent the visible and well known registers of the processor. Many instructions also change the internal state of the processor and the results of the next computations depend on those changes. This means, we also have to represent the internal registers of the processor. In this chapter, we will give an overview about the Intel x86 Architecture as well as the model used for it.

3.1 The x86 Processor Architecture

The x86 has been extended over many years and is very complex concerning the number of instructions and the number of internal registers as well. A short overview about the history is given in Appendix A. With every generation of processors, new instructions were introduced. Nowadays, some of the old instructions are not used anymore because there are newer instructions that can simulate multiple of the old instructions. Old and unused instructions are not taken out of the instruction set because of compatibility with old programs. This means that the instruction set is unusually huge compared to other architectures.

For our processor model, a processor configuration has to be defined. The

processor configuration consists of a memory and a processor state, namely a number of registers. The registers may be changed directly by the instructions, but they can also change indirectly by side effects of instructions. Moreover, the values of the registers can have many different meanings: from a hardware point of view they are bit vectors, but instructions also use them polymorphically as numbers and as pointers for memory addresses (although from a hardware point of view, nobody would call them "pointer"). Processor instructions can change the state, in particular one or more registers or memory locations. For the verification it is essential to have a representation of the registers and the memory locations. The instruction's behavior can be described by the changes on that representation.

3.2 Concrete Processor State

The first part of the processor model is the memory. There are lots of different design decisions in the memory model, and x86 architecture has many different access modes. First of all, memory can be accessed with 8-, 16-, 32- or 64-Bit width. This means, the processor only reads the given bits out of the memory and writes them into a register. The addressing of the memory can be done in different modes again. We will present two different versions of *segmentation* and the *flat* memory model.

Segmentation, general The processor architecture supports a so called *segmented* memory. Segmentation means that the memory is not accessed as one big piece of memory, but is divided into chunks. Those chunks are called *segments*. In those segments, the memory can then be accessed with a so called *offset*, relative to the segment start. With this trick, 64-Bit memory can be accessed with two 32-Bit registers: one gives the segment (such to say the higher 32-Bits) and one gives the offset (such to say the lower 32-Bits).

For a better understanding, we will give an example for addressing pixels in a picture. If you want to refer to a single pixel in the picture (for example the color), you have different options. One possibility is to count the pixels, such that each pixel has a unique number. This would correspond to a direct address. But usually, referring to a pixel would be done with a combination of column and row numbers. You can give a row (this would correspond to the segment in our memory model), and in this single row we need a column to find the concrete pixel (this corresponds to the offset in our memory model). Transcoding the two different addressings into each other can be done by simple algorithms.

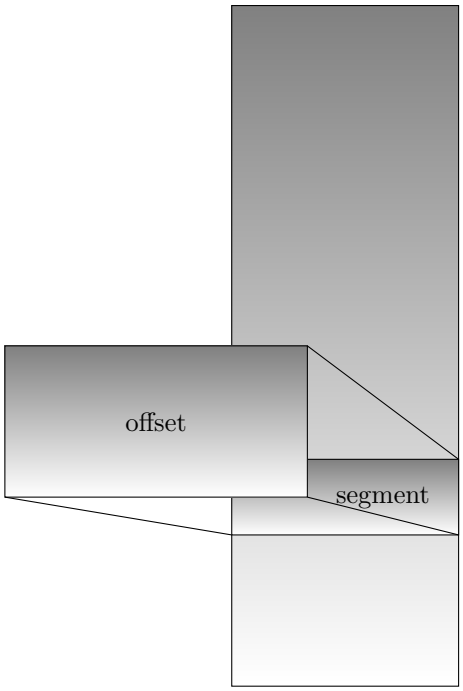

Figure 3.1: Segmentation of memory

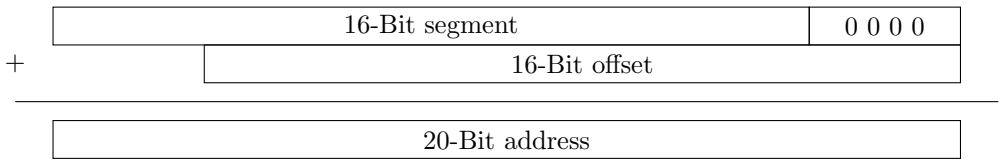

Figure 3.2: Intel Segmentation in Real Mode

Segmentation, special x86 mode The oldest addressing mode in x86 is segmented, it is called "Real Mode". In oldest days of the architecture, only 16-Bit registers were available. To be able to address more than 64k bytes of memory (in this case 1024k), a trick was used. The address of the desired byte was given by a pair of registers, where one is a so called *segment* and one is the so called *offset*. With two 16-Bit registers, one could address 32-Bits if they are combined directly into a number. But Intel used another design: they multiplied the segment by 16 (adding four 0 bits) and then added the offset register as shown in Figure 3.2 on the facing page. This leads to something more than 20-Bits, because there is an overlapping part of the two registers. Original processors had an overflow, so the highest possible numbers were wrapped again (modulo 2^{20}). It is easy to compute a linear address from a segment/offset pair: $address := 16 * segment + offset$. In the other direction, one possibility is $segment := address[19 : 16]0^{12}$ and $offset := address[15 : 0]$. Note that computing segment/offset pairs is not unique, because the two numbers do overlap in the address translation. The suggested formula is only one of the possibilities.

Flat memory model There is also a *flat* memory model, where all memory is accessed as one piece of memory directly. This is the only memory access mode we are handling here, because the other modes are not used in the Microsoft Hypervisor (and its verification is our main goal). In this mode, memory access can be seen as a 64-Bit pointer access in C notation. Note that a *64-Bit* pointer does not mean a pointer to a 64-Bit number, but refers to the bits of the memory address. The memory address corresponds to a pointer value in C and an access to the memory address corresponds to a pointer dereference in C. Thus we can use a C verification tool if it supports 64-Bit wide pointers (most verification tools do not care about the pointer size). The flat model is the easiest access mode, because no complicated computation has to be performed before getting the address.

Processor Registers The more voluminous parts are the registers of the processor. The complete processor model consists of about 160 registers. We will only give an overview, thus only mention the most important ones at this place. The general purpose registers are shown in Table 3.1 on the next page, all other registers are shown in Table 3.2 on page 33.

There exist 16 general purpose registers (GPR). They are used for all kinds of arithmetic, bitwise and memory access operations. When the processor architecture was introduced, only 8 of those registers did exist. The registers were extended to 32-Bit and later on to 64-Bit. The names of them are derived from

64-Bit	32-Bit	16-Bit	8-Bit
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RDI	EDI	DI	DIL
RSI	ESI	SI	SIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8	R8D	R8W	R8L
R9	R9D	R9W	R9L
R10	R10D	R10W	R10L
R11	R11D	R11W	R11L
R12	R12D	R12W	R12L
R13	R13D	R13W	R13L
R14	R14D	R14W	R14L
R15	R15D	R15W	R15L

Table 3.1: General Purpose Registers of the x86 Architecture

the old 16- and 32-Bit registers of the x86 architecture. All registers can be accessed as 64-, 32-, 16-, or 8-Bit values. The names and availability can be seen in Table 3.1. We implemented only the 64-Bit versions of the registers. All changes on the smaller portions of the registers are described with respect to the 64-Bit registers. Normal behavior for x86 processors is a zero extension, even if one would expect sign extension. Note that in this case, a negative 32-Bit number will result in a positive 64-Bit number. This is part of the Intel architecture and also the instruction semantics in our model.

Beside those general purpose registers, there exist a lot of additional registers. First and most important is the $RFLAGS$ register that is a bit vector where every bit represents an event of the last computation. For example, the bit 6 of the $RFLAGS$ register is the zero flag (abbreviated ZF). It is set if the result of the last computation was equal to 0. Conditional jumps check the flag register to decide whether to jump. In Assembler it is not possible to conditionally jump depending directly on the value of some general purpose register. Instead one has to use an additional instruction that changes the flag regsiter.

There are 6 segment registers (CS, DS, SS, ES, FS, and GS). They are used in segmentation mode where an address does not consist of just a flat number but of a pair of segment and offset. Those two values are then combined to compute the linear address. In our model, they do not have a meaning because we are working in flat mode. Nevertheless, they have to be modeled because the processor state can be saved or loaded to or from memory. The same is true for floating point registers (FP), registers of the multimedia extension (MMX),

name	size	bit-wise access
RFLAGS	64	yes
CS	64	no
DS	64	no
SS	64	no
ES	64	no
FS	64	no
GS	64	no
FP0-7	80	no
MMX0-7	64	no
XMM0-15	128	no
GDTR	80	no
IDTR	80	no
CR0-8	64	yes
DR0-7	64	no

Table 3.2: Special Purpose Registers and their size in the processor

and registers of the streaming single-instruction multiple-data extensions (SSE) called XMM. In the Microsoft Hypervisor they are not used except to save the processor state to memory.

Some registers describe the descriptor state of the guest operating systems. The global descriptor table ($GDTR$), local descriptor table ($LDTR$), and interrupt descriptor table ($IDTR$) as well as the task register (TR) do exist but again the values are not interesting for the Microsoft Hypervisor verification itself. They are only used if memory is exchanged between two different virtual address spaces.

Some other registers exist in the non-virtualization mode. There are 5 control registers ($CR0$, $CR2$, $CR3$, $CR4$, and $CR8$) and 6 debug registers ($DR0$, $DR1$, $DR2$, $DR3$, $DR6$, and $DR7$). They are used for example to control the debugging mode of the processor. In debugging mode, only one step of the guest operating system is executed before returning to the hypervisor. Such modes can be activated by special bits in the registers. Some instructions do use them to toggle behavior.

3.2.1 Hardware Virtualization

Under a normal operating system, different programs are executed side by side. Multiple programs are separated from each other by memory management. Server processes are often targets of attackers from networks. If one such server process is hacked, the attacker can reach other processes by executing operating system functions. To address this attack scenario, important server processes are executed on different computer systems. If an attacker is able to

hack one of the processes, it cannot affect the other systems. But with this separation, another problem occurs. The computers are not very balanced anymore: some computers are "overloaded" and cannot handle all requests as fast as they should, while others are "underloaded" so they would have computing power for other purposes.

Hardware Virtualization was invented to use computer hardware more efficiently. Virtualization allows running different operating systems on just one physical machine. This leads to a more efficient machine, because the processes that need processing power can now get the power from other processes that do not need it. Having the processes in different operating systems can have different reasons. The security reason was already mentioned. Some programs run better on special operating systems; so different server processes should not only be executed on different instances of one operating system, but also of different operating systems. To get these operating systems running in parallel, there is one program called *hypervisor*. This hypervisor manages all operating systems as so called *guests*. All guests have their own virtual memory environment and do not see that they are scheduled. Nowadays, operating systems do run different user programs at the same time and all programs do have their own virtual memory. The difference to a hypervisor is that user programs usually do not have shared access to hardware components except if the access happens through a hardware abstraction layer of the operating system or a special driver. Operating systems access hardware directly and thus cannot easily work with another operating system accessing the same hardware at the same time. The necessary scheduling is part of the hypervisor. The general idea behind a hypervisor is inherited from a micro kernel. Micro kernels only have a small number of hardware drivers and mainly schedule other processes, running with limited rights. Hardware access in micro kernels is done by processes. That is also the idea behind a hypervisor.

Writing such hypervisors can be done in pure software, running guests with the rights of a normal program. In this case, there are restrictions, e.g., the operating system cannot access hardware directly and it is executed with limited rights. Operating systems are not written to be executed in such a way, so the virtualization software has to manage all the differences in the background without the operating system noticing it. The virtualization software has to be updated to keep operating systems running with the newest updates and even then there are sometimes incompatibilities. Another method is to get hardware support. In this case, a more privileged mode than the one for a normal operating system is introduced. A guest operating system can now run in the original system layer. The hypervisor has even more privileges, for example on the memory allocation. The guest operating system does not see that it is

less privileged now. Memory access is not done like in the original system layer, but with an additional translation and an additional page table that a guest does not see or know of. The memory allocation is now done with an additional (invisible) pagetable, using so called *nested page tables*. The hypervisor can set wakeup events, such as invalid instructions (changing to the virtualization mode), interrupts and so on. Many additional administrative data structures are introduced in the software (hypervisor) and in hardware (processor) to hide the differences of the execution.

There are two concurrent implementations of hardware virtualization by AMD and Intel. They differ not only in the names of the instructions but also in the behavior and the processor state. At the moment, we provide only a model for the AMD virtualization. But the Intel architecture should be representable in a very similar way. We implemented the AMD version because it is older and thus the implementation in the hypervisor is expected to be more mature.

3.3 Abstract Processor Model

For the verification, we create an abstract model of the physical (concrete) processor. The concrete model consists of a set of bitvectors, called registers. We created a model for this that represents this concrete processor state in a programming language. This model can then be used for verification of the assembler programs. In our concrete scenario, we want to use an existing C verification tool, namely VCC that is used for the C parts of the Microsoft Hypervisor. The C model of the processor will be called *abstract* processor state. It consists of a set of variables containing numeric values. The real processor instructions change the concrete processor state, while the transition function presented in Section 3.4 will change the abstract processor state.

The physical registers in the concrete processor state have names, arbitrarily chosen by the developers. The concrete instructions refers with those names to the physical registers. We mostly use the same names to access our abstract state. Thus the semantics of the instructions can be expressed in a very similar fashion to the descriptions of the processor instructions.

During the work, two different models were implemented. The first implementation was simple, because it was very close to the concrete state. For every register there is a global variable with the same name. This allows for the specification to constrain the values of registers in the assembler code. In the translated version this specification refers to the corresponding variables in the C model. Also the translated C version of the code refers to these global variables and looks very similar to the original Assembler version.

In the first implementation, only arithmetic registers were modeled com-

```
    uint64              R[16];
2   SegReg              SR[6];
    uint64              kernelGSbase;
    SegReg              TR;
    SegReg              LDTR;
    DescTableReg        GDTR;
7   DescTableReg        IDTR;
    CtlReg0             CR0;
    uint64              CR2;
    CtlReg3             CR3;
    CtlReg4             CR4;
12  CtlReg8             CR8;
    FlagsReg            RFLAGS;
    bool                GIF;
    EFEReg              EFER;
    ActivityState       activity_state ;
17  bool                interrupt_shadow;
    uint08              CPL;
    uint64              DR[8];
    FPReg               FPR[8];
    uint128             XMM[16];
22  uint64              RIP;
    uint64              oldRIP;
    Prefixes            prefixes ;
    Rex                 rex;
    Opcode              opcode;
27  ModRM               modrm;
    Sib                 sib ;
    uint32              disp;
    uint64              imm;
    STAReg              STAR;
32  uint64              LSTAR;
    uint64              CSTAR;
    SFMASKReg           SFMASK;
    SysEnterCSReg       SYSENTER_CS;
    SysEnterESPReg      SYSENTER_ESP;
37  SysEnterEIPReg      SYSENTER_EIP;
    VmCtlReg            VM_CR;
    IgnNeReg            IGNNE;
    SmmCtlReg           SMM_CTL;
    VmHSavePaReg        VM_HSAVE_PA;
42  uint64              SVM_KEY;
    VirtualizationMode  vtmode;
    uint64              VMCB_ADDR;
    VmcbCA              VMCB_CA;
    CtlReg0             hCR0;
47  CtlReg3             hCR3;
    CtlReg4             hCR4;
    PatReg              hPAT;
    EFEReg              hEFER;
    PatReg              PAT;
52  MTRRcapReg          MTRRcap;
    MTRRdefTypeReg      MTRRdefType;
    MTRRphysBaseReg     MTRRphysBase[8];
    MTRRphysMaskReg     MTRRphysMask[8];
    MTRRfixReg          MTRRfix64K_00000;
57  MTRRfixReg          MTRRfix16K_80000;
    MTRRfixReg          MTRRfix16K_A0000;
    MTRRfixReg          MTRRfix4K_C0000;
    MTRRfixReg          MTRRfix4K_C8000;
    MTRRfixReg          MTRRfix4K_D0000;
62  MTRRfixReg          MTRRfix4K_D8000;
    MTRRfixReg          MTRRfix4K_E0000;
    MTRRfixReg          MTRRfix4K_E8000;
    MTRRfixReg          MTRRfix4K_F0000;
    MTRRfixReg          MTRRfix4K_F8000;
67  IORRBaseReg         IORRBase0;
    IORRMaskReg         IORRMask0;
    IORRBaseReg         IORRBase1;
    IORRMaskReg         IORRMask1;
    TopMemReg           TOP_MEM1;
72  TopMemReg           TOP_MEM2;
    SysCfgReg           SYSCFG;
    ApicBaseReg         APIC_BASE;
    uint64              TSC;
    uint64              TSC_AUX;
77  CPUID_00000000Reg   CPUID_00000000;
    CPUID_00000001Reg   CPUID_00000001;
    CPUID_80000000Reg   CPUID_80000000;
    CPUID_80000001Reg   CPUID_80000001;
```

Figure 3.3: Abstract processor state

pletely. Many auxiliary registers were modeled only by a variable storing the value. However, their effect to other instructions was not modeled. For example, in our first model, a processor could claim via its CPUID registers that it does not support virtualization although it executes virtualization functions. The values of the registers were chosen nondeterministicly.

Because the first processor model introduces a single global variable for each register, it can only support one processor. Nowadays, most of the existing computers have multiple processors or cores in them. This means that multiple threads can run at the same time. To enable verification for multi-processor architectures, a processor must be implemented as a datatype. This datatype can be instantiated for the different threads, and thus a multi-processor system can be simulated. Note that in a multi-processor system no data structures have to be shared between the single processors.

In our second model, the processor state is a huge data structure. The mapping from the concrete state to the abstract state is now done with the help of a partial function. This function takes as input a register name of the concrete state and returns the location of the corresponding register in the abstract state. This partial function is part of the automatic translator that was implemented in this work.

Note that the "Abstract" Processor Model is not more abstract in the sense that it stores less information than the Concrete Processor State. Instead, all information is kept, but it is represented not in hardware but in software. The word is mainly used to make clear what state we are referring to and was chosen because in the definition of abstraction it is referred to as a subset, not a strict subset.

For the rest of this work, we use the notation of the first model. This is easier to read, because the instance of the data structure does not appear. We use the register names from the concrete state as the names for the abstract processor state. In the real implementation, we have different instances and the additional overhead to select the current instance is handled by the automatic translation.

3.4 Instruction Semantics

Assembler functions consist of a sequence of Assembler instructions. An instruction takes as input a processor state and generates a new processor state. In instruction manuals of Intel and AMD, the changes are described relative to the concrete processor state. In our verification approach, instruction effects are described by C code that manipulates the abstract processor model from Section 3.3. For each Assembler instruction we define a C function. This function may

	type name	abbreviation in mnemonics
8 Bit	byte	b
16 Bit	word	w
32 Bit	double word	d
64 Bit	quad word	q

Table 3.3: Bit width and their abbreviation

take more parameters than the Assembler instruction to make some information explicit (for example the segment registers for memory access). This is more convenient for the instruction specification that is later used in the verification process.

Some instructions behave very different, depending on the type (memory or register) of their arguments. In the original description from Intel, the specification has many case distinctions. We avoid them by introducing several functions by coding the type of the destination operand into the name of the function. The relevant type information is already known by the parser. For example, the parser can distinguish between memory access and register access, so it produces not a call to a "mov" function, but to a "mov_mem" or a "mov_reg" function. The specifications of the functions are simpler if the type of the destination is known. In contrast, the source operand can be evaluated completely and can be given as a numeric value to the function. Thus its type does not need to be encoded in the function name. We also distinguish the width of the access, as shown in Table 3.3. A move into a 32-Bit register will then have the instruction name "mov_reg_d" in our translation.

In principle, the complete semantics could also be given by the automatic translation. Instead of introducing a call to a function, the translator could also inline the code that manipulates the abstract processor state. Our approach makes it easier to implement a new processor model (either a more precise model or for another architecture), because the complete model resides in an external file that does not interfere with the tool sources.

The semantics of the instructions, or in other word the changes made by the instructions, represent the transition function of the processor. In the hardware processor, this function takes a concrete processor state and returns a new concrete processor state. In our abstract model, this function takes as input an abstract processor state and returns a new abstract processor state. The abstract processor state is given by a set of registers as shown in Table 3.1 on page 32 and Table 3.2 on page 33 as C variables (see also the complete C definition in Figure 3.3 on page 36). The semantics of the instruction is given as a C function, for example *mov_mem_q(x,y)* (see Figure 3.4 on page 43).

Bit	name of the flag	abbreviation used
0	Carry Flag	CF
2	Parity Flag	PF
4	Auxiliary Carry Flag	AF
6	Zero Flag	ZF
7	Sign Flag	SF
8	Trap Flag	TF
9	Interrupt Enable Flag	IF
10	Direction Flag	DF
11	Overflow Flag	OF
12+13	IO Privilege Level	IOPL
14	Nested Task	NT
16	Resume Flag	RF
17	Virtual-8086 Mode	VM
18	Alignment Check	AC
19	Virtual Interrupt Flag	VIF
20	Virtual Interrupt Pending	VIP
21	ID Flag	ID

Table 3.4: Flags and their corresponding bits

The translation replaces the Assembler instruction by function calls to those C functions.

3.4.1 Flags

The register called *flags* is a bitvector, where every bit has a special meaning (see Table 3.4). For a complete description, the interested reader can look them up in the Intel architecture manuals. For simplicity, we will only explain those bits that are used in the presented instructions.

The bits of the *flags* register are set by many instructions, according to the result of the computation. In Assembler, they are mainly used for conditional jump instructions, which may depend only on the flags. The most important flags for the Microsoft Hypervisor code are the Zero flag (abbreviated ZF) and the Sign flag (abbreviated SF). The Zero flag is represented by bit 6 in the 64-Bit flag register, the Sign flag is represented by bit 7. If the result of a computation is zero, the zero flag is set, and if it is less than zero, the sign flag is set, otherwise the flags are cleared. The Zero flag computation can be represented with the following pseudo code:

if $a = 0$ **then**
  $ZF \leftarrow 1$
else
  $ZF \leftarrow 0$

end if

where ZF represents the Zero flag in the flag register and a is the result of the computation. The flags are important for example to handle loops, where a register is used to count down to 0. After every decreasing step, the Zero flag is set according to the decreasing computation. With a conditional jump, the code execution can continue, depending on the result. The jump instructions are described in Section 3.4.7.

The Sign flag is more complicated. In our abstract processor state the registers are defined as unsigned integers. This means there is no real sign. Hardware registers are bitvectors that are interpreted in different ways. For example the bitvector could be interpreted as a signed number or as an unsigned number, but also as an address in the memory. To represent negative binary numbers, the bitvector is interpreted as a two's complement number. In this notation, negative numbers have a "1" in the most significant bit (in this case the Bit 64). So, the Sign flag is originally not representing the sign of the number but the most significant bit and is also set if the result of an unsigned computation is very large. We check that the most significant bit is set by checking the (unsigned) register is at least 2^{63}. Note that we will not use it in the rest of the work for readability reasons, but the simplified test for a number less than 0. In the implementation, the test is done with the most significant bit.

In principle, we could define a single function that changes all flags according to the result. However, some of the processor instructions do not change all flags, but only parts of them; others set the flags differently, e.g., the sign flag differs for 32-bit computations. For example *logical and* only changes Zero, Sign, and Parity flag (set if the number of set bits in the result is even) according to the result; Overflow (usually set if the most-significant (sign) bit of the result differed from the signs of both source operands) and Carry flag (usually set if the last integer addition or subtraction resulted in a carry out of the most-significant bit position of the result) are always set to 0, while Auxiliary flag (set if the last binary-coded decimal (BCD) operation resulted in a carry) is undefined. The flags have to be changed after every instruction according to the instruction manuals from Intel or AMD. Therefore, we introduce auxiliary functions for each flag that sets it according to the result. These functions are called by the functions implementing the instructions as described in the instruction manuals, or the flags are set to fixed values if the instruction requires this.

3.4.2 Instruction Pointer (IP)

The instruction pointer (short: IP) points to the next instruction that will be executed. It can be changed because of one of three cases:

- an instruction has been executed, so the IP is increased to point to the next instruction,
- a jump has been performed and the IP now points to another program location, or
- a call has been performed (respectively a return at the end of the call) and the execution jumps to another function.

In our implementation, the IP is not part of the abstract semantics. This is not because we forgot about it, but we found out that we do not need it. A description of the reasons for neglecting the IP is presented in the next paragraphs.

First of all, if we translate the Assembler program into C code, we do not have addresses. So the IP would not make any sense in this case. To make it point to something, we would have to translate sizes of the instructions into labels and use those labels instead of addresses. So all lines of the translated code would start with a label. For the execution of the next instruction, we do not need the IP, because in C semantics, the next instruction is "loaded" automatically, without a pointer explicitly pointing to it. The more interesting parts are the branching instructions.

If a *jump* instruction is performed, the processor has to modify the IP register, such that the execution goes on at another point in the program. Usually, programs are not written in pure Assembler language, but in a so called *Macro Assembler*. This Macro Assembler allows not only direct jumps to an address given by a number, but also to labels that are defined somewhere else in the program. Direct jumps are very complicated, because instructions in Intel Assembler do not have fixed length. If, for example, a register access is changing from 16-Bit to 32-Bit width, the size of the instructions byte code changes because a prefix is dropped (it was needed for 16-Bit access). Direct jumps are not used in the complete Hypervisor code, and we have not seen them in other sources either. Instead, developers do use labels from the Macro Assembler language. This fact is first of all very interesting and second very helpful for our translation. In the C language, a construct with labels and jumps is also provided, namely the *goto* statement. We can translate each Macro Assembler label into a C label and each Assembler *jump* into a C *goto*. Doing so, we do not have to access the IP register.

For *call* and *return* instructions, not only the IP register is changed. They also use a special stack frame that is written by the *call* and restored by the *return*. In this stack frame, there are values for the back address to continue execution there, but also for the stack pointer and other information. Usually,

this stack frame is not accessed directly, except by viruses and schedulers. Using the C calling mechanism, the same information is only given implicitly. Having a return address to continue execution is also done by the semantics of C *return* statement. IP does not have to be changed explicitly there; it is changed by the C instructions implicitly.

Altogether, we decided not to model the IP at all. If somebody wants to access the stack frame, we would need to introduce a stack frame, which would be difficult from C code. If a direct jump outside the current function is used, our implementation would give an error at the moment, and this is not easy to circumvent, because it is a design decision.

3.4.3 Mov Access

A *Mov* access instruction assigns a value to a register or memory address. It is named after the mnemonic of the simplest instruction of the group. The value can come from a memory access, from a register, an immediate constant, etc. The destination can be a register or a memory region. It does not change the flag registers. Note that the instructions have some restrictions on the combinations of operands that are possible. For example, it is not possible to read from a debug register and write to another debug register. We do not care about the restrictions, because we use a preprocessor before our tool that already checks for those combinations.

An instruction $mov\, rax, rbx$ for example will copy the value from rbx into register rax:

$rax \leftarrow rbx$

Memory access is translated by the syntax translation (before this step) to pointer analysis in C style. No adaptations have to be introduced at this point. We denote memory access by writing the memory address in square brackets "[" and "]" as it is done in Assembler. This corresponds to a pointer access in C, where the memory address is given by a variable of a pointer type and the access to the cell is done by dereferencing with a "*" before the address Here is an example *mov*-instruction that will copy the value stored in register rbx into the memory cell refered to by rax ($mov\, [rax], rbx$):

$[rax] \leftarrow rbx$

The specification and the C implementation for a memory write access is given in Figure 3.4 on the facing page. The actual writing is implemented in an extra function because other instruction implementations reuse this function.

From here on, we will also refer to the parameters of an instruction with letters a, b and so on. So the general *mov* instruction would be:

$a \leftarrow b$

```
void mov_mem_q(SRIndex sreg, uint64* offset, uint64 data)
    maintains(thread_local(offset))
    maintains(mutable(offset))
    writes(offset)
    ensures(*offset == data)
{
        *offset = data;
}
```

Figure 3.4: Implementation of a 64-bit *mov* instruction writing to memory

Another variant of the *mov* instruction is *movnti* that is "storing a double word using non-temporal hint" (see instruction manuals). This means it writes 32- or 64-Bits to memory without caching the memory line (this is the non-temporal hint). The memory is not even in cache after the write process, so a read from that memory location would lead to a cache miss afterwards. This is a technical detail, mainly for performance reasons. Given that we are only interested in functional correctness, not in performance, we can translate the instruction in the same way as the ordinary *mov*.

The instruction "load effective address" *lea* computes the address of a memory cell without dereferencing it. The address of a memory cell allows for limited arithmetic, e.g. adding a constant to a base address given by a register. That is the reason why it is even used today, because it can make fast computations of simple arithmetic expressions (like addition or multiplication with a small constant) without overwriting the input. The instruction's parameters look like parameters of a *mov* instruction. The only difference is that the pointer arithmetic is done but the dereferencing is not. A difficulty for our implementation of the instruction is that the pointer dereference is already introduced during the syntax translation pass. Therefore, we had to change the syntax translation: the *lea* instruction has to be detected during the parsing process to find out whether the dereferencing should be left out. The *lea* instruction is only applicable if the second parameter is a memory address.

As last instructions of this section, we would like to mention *movaps* (Move Aligned Packed Single-Precision Floating Point Values) and *movdqa* (Move Aligned Double Quadword) that move a 128-Bit data structure to a XMM register. The values of the XMM registers are not used in the Microsoft Hypervisor, the difference between a packed single-precision floating point value and a double quadword can thus be neglected. In Hypervisor, the instructions are used to save XMM registers to the memory or load them back again.

3.4.4 Stack operations

The stack is a special memory region with a special pointer to it (the register *rsp*). Although the stack is not visible in high-level languages like C, it is always available. The compiler usually creates stack frames, containing addresses where the execution is to be continued after the end of the function, parameters of functions, and much additional information. In contrast to those automatically managed data areas in high-level languages, the stack has to be managed by hand in Assembler. If more variables than the 16 general purpose registers are needed, some of the results have to be saved somewhere temporarily. This is usually done by pushing them onto the stack, although there exist some other ways. The alternatives are mainly used by high-level compilers and only very rarely in Assembler code. If the results are needed later on, the program can pop them again from the stack. Interesting is that pushing does not only put a value in the memory, but also sets the stack pointer to the next memory location. The same is true for the pop operation: it reads the value from the stack and sets the stack pointer to the last memory location. This means the stack is implementing a LiFo (Last in First out) queue.

While the *push-pop* pair is writing or reading a normal register, *pushfq-popfq* are doing the same for the flag register. Thus the flag status can be saved and later on restored.

The implementation looks like this:

push

$rsp \leftarrow rsp - 8$
$[rsp] \leftarrow a$

pop

$a \leftarrow [rsp]$
$rsp \leftarrow rsp + 8$

pushf

$rsp \leftarrow rsp - 8$
$[rsp] \leftarrow rflags.raw$

popf

$rflags.raw \leftarrow [rsp]$
$rsp \leftarrow rsp + 8$

where the *raw* field refers to all flag bits as bitvector. Note that the *popf* instruction is simplified here. The processor manual specifies that not all flags

are read from memory and some reserved bits are ignored. Our implementation also takes care of this. Note that the stack grows downwards while most normal data structures would grow upwards.

3.4.5 Arithmetic Operations

So far, only linear arithmetic is modeled. But multiplication and division could be implemented in a very similar fashion.

C language implements linear arithmetic almost in the same way as machine instructions; an interesting point is the overflow handling of VCC. VCC will normally check for overflows and report an error if it cannot prove their absence. To remove this error the programmer has to give a precondition on the input values that restricts the range to prevent overflows. Therefore, our tool does not have to check for overflows; they cannot occur in correct programs. The overflow bit is cleared by all arithmetic operations. Other flags have to change according to the result of the operation. A real C implementation with its specification is given in Figure 3.5. The pseudo code representation of the *add* instruction looks like this:

$a \leftarrow a + b$
if $a = 0$ **then**
  $rflags.zf \leftarrow 1$
else
  $rflags.zf \leftarrow 0$
end if
if $a < 0$ **then**
  $rflags.sf \leftarrow 1$
else
  $rflags.sf \leftarrow 0$
end if

and in the same way the *sub* instruction:

$a \leftarrow a - b$
if $a = 0$ **then**
  $rflags.zf \leftarrow 1$
else
  $rflags.zf \leftarrow 0$
end if
if $a < 0$ **then**
  $rflags.sf \leftarrow 1$
else
  $rflags.sf \leftarrow 0$

```
void add_reg_q(uint64 &reg, uint64 data)
writes( rflags , reg)
ensures(reg == old(reg) + data)
ensures(rflags.CF == carry_add_q(old(reg),data))
ensures(rflags.PF == parity_q(unchecked(old(reg)+data)))
ensures(rflags.AF == (uint64)((old(reg) & 15) + (data & 15)) < (data & 15))
ensures( rflags.ZF == (uint64)(unchecked(old(reg)+data) == 0))
ensures( rflags.SF == (uint64)(unchecked(old(reg)+data) >> 63))
ensures(rflags.OF == overflow_add_q(old(reg),data))
ensures(rflags.TF == old(rflags.TF))
ensures( rflags.IF == old(rflags.IF))
ensures(rflags.DF == old(rflags.DF))
ensures(rflags.IOPL == old(rflags.IOPL))
ensures(rflags.NT == old(rflags.NT))
ensures(rflags.RF == old(rflags.RF))
ensures(rflags.VM == old(rflags.VM))
ensures(rflags.AC == old(rflags.AC))
ensures(rflags.VIF == old(rflags.VIF))
ensures(rflags.VIP == old(rflags.VIP))
ensures( rflags.ID == old(rflags.ID))
{
        uint64 oldr = reg;
        uint64 r = oldr + data;
        reg = r;
        rflags.CF = carry_add_q(oldr, data);
        rflags.AF = (uint64)((oldr & 15) + (data & 15)) < (data & 15);
        rflags.PF = parity_q(r);
        rflags.ZF = (uint64)(r == 0);
        rflags.SF = (uint64)(r >>63);
        rflags.OF = overflow_add_q(oldr, data);
}
```

Figure 3.5: Specification and implementation of an *add* Assembler instruction

end if

Note that we do not have any optimization at this position. If there are a number of additions one after another, the flags will not be used except for the last instruction. But instead of optimizing them out of the code, VCC or Z3 will detect the information to be irrelevant.

Increment and decrement are special cases of addition and subtraction, where the second parameter is set to "1". Thus, changes on the flags are equal to the changes of addition and subtraction.

The compare instruction *cmp* is a special instruction that is not obviously an arithmetic operation. It sets the flags according to a subtraction of the parameters like *sub*, but does not save the result in the register. Thus, only the flags are set according to the result and subsequent conditional execution can take place. Here is the implementation of *cmp*:

if $a = b$ **then**
  $rflags.zf \leftarrow 1$
else
  $rflags.zf \leftarrow 0$
end if
if $a < b$ **then**
  $rflags.sf \leftarrow 1$
else
  $rflags.sf \leftarrow 0$
end if

The compare and exchange *cmpxchg* instruction is more complicated to explain. It compares the first parameter to the value in the rax register (this is a fixed, implicit parameter). If the two values are equal, the value of rax is set to the value of the first parameter. Otherwise the value of the first parameter is set to the value of the second parameter. Here comes the pseudo code of the implementation:

if $rax = a$ **then**
  $rflags.zf \leftarrow 1$
else
  $rflags.zf \leftarrow 0$
end if
if $rflags.zf = 1$ **then**
  $rax \leftarrow a$
else
  $a \leftarrow b$
end if

Note that *cmpxchg* does not change the Sign flag, in contrast to *cmp*.

3.4.6 Logical Operations

Although they are called "logical operations" by Intel, the listed operations are bitwise operations. Bitwise operations handle the register, memory location or constant as bitvector. The instructions also affect the Zero and Sign flags, their computations will not be mentioned anymore henceforth. The implementation of the instructions is straight forward because bitwise operations also exist in the C language.

For the bitwise operations (*xor*, *or*, *and*, *test*, *not*) the Sign and the Zero flags are set according to the result. The operator $\circ$ would refer to one of the logical operations *xor*, *or*, *and*:

$a \leftarrow a \circ b$

if $a = 0$ **then**
  $rflags.zf \leftarrow 1$
else
  $rflags.zf \leftarrow 0$
end if
if $a < 0$ **then**
  $rflags.sf \leftarrow 1$
else
  $rflags.sf \leftarrow 0$
end if

The logical test instruction *test* sets the flags register like the *and* instruction, but it does not write back the result. The bitwise negation *not* switches all bits from 0 to 1, and from 1 to 0 respectively. Flags are not changed at all by the *not* operation.

3.4.7 Labels, Jumps and Invariants

Jumps exist in different varieties, because they usually contain conditionals in their name. As described in Section 3.4.2, the IP register is not modeled. Instead of using direct jumps with an address, labels in the Macro Assembler context are used. A jump to a label *LABEL* can then be translated like the following code:

$LABEL:$
...
Goto $LABEL$

The conditional jumps express the condition in the names, for example conditional jump for the Zero flag is called *jz*, or for the negation *jnz*. While *jz* jumps if the Zero flag is set, *jnz* jumps if the Zero flag is not set. The corresponding code for a conditional jump to a label *LABEL*:

$LABEL:$
...
if *cond* **then**
Goto $LABEL$
end if

The condition is coded into the 2nd and 3rd letter of the Assembler instruction, as shown in Table 3.5 on the next page. The instruction name is built by a leading "j" plus the condition behind.

In Assembler, we do not have structured loops. Instead, loops are implemented by a label at the beginning of the loop and a conditional jump back to that label. For the verification of such loops, invariants have to be annotated.

short cut	description	*cond*
Z	zero	$RFLAGS.ZF = 1$
NZ	not zero	$RFLAGS.ZF = 0$
E	equal	$RFLAGS.ZF = 1$
NE	unequal	$RFLAGS.ZF = 0$
S	negative	$RFLAGS.SF = 1$
NS	zero or positive	$RFLAGS.SF = 0$

Table 3.5: Conditions for conditional jumps

They do not exist in ordinary Assembler code, so we introduced them artificially as special comments. Comments in Assembler start with semicolon (;), additional specification starts with ";^" plus a keyword. The keyword for invariants is "invariant" and takes as argument VCC expressions, so the simplest invariant would be:

```
; ^ invariant (true)
```

Besides the simplest invariant, you could give arbitrary complex ones. They can include quantifiers, the global state of the processor, and the memory.

Chapter 4

Pitfalls

In Section 4.1, we show how C and Assembler functions can be combined. From the calling conventions of C compilers, we get proof obligations for the Assembler functions, presented in Section 4.2. In Section 4.3 we present the techniques to verify multi-threaded programs. In Section 4.4 there is a description of general properties we verified on the Assembler code.

Together with the function specific postconditions and writes clauses, we get the complete specification of every single function.

4.1 Mixing Assembler and C Functions

Most of the programs are written in high-level languages like C or Java, not in Assembler. There are different reasons for that. One is the fact that in Assembler, lots of implicit programming techniques for other languages have to be implemented explicitly. For example the stack that is implicitly used in high-level languages, has to be implemented for Assembler programs explicitly. There is no variable creation like in high-level programming languages, instead, only a fixed number of registers are available. In high-level languages, the compiler puts the registers into the memory (either stack or heap) and loads them automaticaly into registers when they are used as variables. In Assembler, such a concept has to be implemented by hand. Because of such reasons, Assembler code is huge compared to the same program implemented in high-level languages. On the other hand, such Assembler code can be more optimized because concepts that are not used in a program do not have to be implemented.

In the high-level/low-level translation (that a C compiler does), lots of things have to be made explicit for the hardware execution. Besides the fact that variables do not exist, there is also the fact that in hardware execution registers can be accessed or types are not as strict as in high-level languages. Such a

```
extern int bar(int a)
  ensures (result == a);

int foo(int a)
  ensures (result == a)
{
        return bar(a);
}
```

Figure 4.1: C code of a function, calling another C function

```
;^ ensures (rax == rcx);
bar proc

        mov rax, rcx
        ret

bar endp
```

Figure 4.2: Short Assembler example, that returns the first function parameter

coding can be made in many different ways. To be able to use the same libraries for different compilers, there exist some so called "calling conventions". They tell the compiler developers where to put function parameters, how to handle the stack, and lots more.

Programs written in high-level languages have to be translated to a low-level program that can be executed on the hardware machine. The binary code of such an executable file can also be represented as an Assembler program, where for each numeral description a mnemonic is introduced. Such equivalent descriptions are also part of the processor description, where mainly the binary format is described, but with the help of the Assembler mnemonics. This means that in the processor description, both are documented: the hardware binary format and the Assembler format.

Sometimes one needs to program in Assembler for performance reason or for direct hardware access. The main idea is not to implement a whole program in Assembler code, but use it as a possibility to extend C programs. A combination of C and Assembler is therefore necessary.

Here is the code of a simple example for mixed source. In Figure 4.1, there is a C implementation of a simple function that calls an Assembler function given in an external file. In Figure 4.2, the Assembler implementation is shown. The specification for both functions is very simple in this case: they return the value of the parameter.

In Figure 4.3 on the following page, our translated version of the Assembler

```
void bar()
  ensures (rax == rcx);
{
        rax = rcx;
}
```

Figure 4.3: C translation of a short Assembler example, that returns the first function parameter

function is given. Note the difference in handling parameters and return value. The C function needs parameters declared in the function footprint, where an Assembler function expects the parameters to be in some registers. The return parameter in a C function has to be given a type, whereas in Assembler programs, return values are always in the *rax* register and are loosely typed (a register can represent different things). In this small example, the stack is not used by the Assembler function. Nevertheless, verification of those differences is essential for pervasive verification, if the Assembler function does not meet the calling conventions, the C function will be destroyed completely.

Verification of Assembler functions does not only mean the verification of the Assembler parts of the program, but also that the Assembler function uses the calling conventions correctly. The correct behavior is described in Section 4.2.

4.2 Calling Conventions for C Compilers

A Compiler performs a translation: it takes a high-level language (in this case C code) and produces a kind of binary file (in this case we assume Assembler). Function and procedure calls are translated directly, but the implicit assumptions of the language are made explicit as well. In the C source there are functions with parameter definitions, declarations of local variables, statements, etc. C source code never has expressions about the function stack or a method of transferring parameters, those are the implicit assumptions introduced by the compiler. For the interaction of Assembler and C functions the details have to be made explicit. Assembler functions have to access register and memory in the same way that a C compiler does for the translation of C functions.

Note that there exist two kinds of Assembler:

- Inline Assembler, where the Assembler code is written into a C file
- Pure Assembler, where the Assembler code is written into separate ASM files.

In our case, we are only talking about pure Assembler. This is because our main focus is on Microsoft Hypervisor verification. This is 64-Bit only code, and the 64-Bit C compiler by Microsoft cannot handle inline Assembler. Pure Assembler has the advantage that the interfaces between C and Assembler are defined in a clearer way. In inline Assembler, every time an inlined block is beginning or ending, the border between the two languages is crossed. The translation then has to introduce assertions, preventing the results from being inconsistent. This means, registers and variables have to be in sync, and the verification tool has to keep track of them. In the case of functions, only the pre- and postconditions have to be translated, so that the C function can later use them after they are verified against the Assembler function.

In inline Assembler, jumps from the Assembler part of a function into the C part of the function can be used. This again would introduce a crossing of the languages. Then, the addresses of the instructions have to be computed. And this computation can differ, depending on the optimizations of the undelying compiler. This is the worst case that can occur in code. Hopefully, most of the Assembler programs would not make use of this possibility, because this would lead to very complicated computations for the verification.

In the rest of the document, we will only describe Windows 64-Bit Calling Conventions. For Linux, they can differ, as well as for Windows 32-Bit.

4.2.1 Registers and Parameters

A C compiler has to store the C variables somewhere. It can put them into the stack, in registers, or the heap memory. For computations, the values are often loaded into registers. This is all hidden from the high-level language developer. If a C function would simply call an Assembler function, this Assembler function could destroy all the values the C code depends on. For example, if the stack pointer would point to a different location, nothing that was stored previously on the stack would be found anymore.

To avoid such a behavior, C compiler conventions tell the programmer which of the registers can be used without saving. All other registers have to be saved (mainly on the stack) and restored before returning. Thus, the calling function does not see the change in the register values. For the possibly changed registers, the calling function must not depend on the value of the registers. If the values should be used, the calling function has to save and restore them itself. In Figure 4.4 on the next page, the row *caller* shows the registers that the caller has to care about while the row *callee* shows the registers that the called function has to preserve. From this table, a number of general correctness criteria can be created. For all entries in row *callee*, a postcondition can be introduced:

	rax	rbx	rcx	rdx	rdi	rsi	rbp	rsp	r8 – r11	r12 – r15	xmm
caller	✓	✓	✓	✓					✓		
callee					✓	✓	✓	✓		✓	✓

Figure 4.4: responsibility for the correct use of registers

parameter	1.	2.	3.	4.	return
register	rcx	rdx	r8	r9	rax

Figure 4.5: parameter transfer via registers

$\forall r \in callee : old(r) = r$, where *old(r)* denotes the value of the register at the function entry. In particular, the *rsp* register (the stack pointer) has to point to the same value as it did before the function call. This means the stack size must not grow, because if it would grow, the stack pointer would decrease. This is a very important proof obligation because if it does not hold, the calling function would not be able to read anything from its stack, because the stack could have changed completely.

Another important part that is introduced by the compiler is the parameter transfer for functions. In C, parameters are given as a comma separated list of variable definitions. The return parameter is given as a type of the function and its value is set by the *return* statement. In Assembler, such a construct does not exist. Instead, special registers are used as parameters of functions. Figure 4.5 shows which registers are used for parameter transfer. The register values are filled by the C compiler or (if called by another Assembler function) by the Assembler programmer. The return value is read again from the *rax* register by the caller. If all calls are made from Assembler to Assembler, this is not a problem at all. If all calls are made from C to C, this is not interesting at all. If calls are mixed in one direction, at least one of the functions has to be changed. In our case, we wrap the Assembler functions into a C function. This wrapper function transfers parameters to registers and returns the return value in C style.

I will present this on our example function:

```
void bar()
{
        rax = rcx;
}
```

The wrapper function has to introduce a new function that hides the bar function from calls and knows about the calling conventions. We can implement it like this:

```
1 int bar_wrap(int a)
```

```
ensures (result == a) /* annotation of C function */
ensures (rdi == old(rdi))
ensures (rsi == old(rsi))
ensures (rbp == old(rbp))
ensures (rsp == old(rsp))
ensures (r12 == old(r12))
ensures (r13 == old(r13))
ensures (r14 == old(r14))
ensures (r15 == old(r15))
{
        rcx = a;    //1st parameter
        bar()       //function call
        return rax; //return value
}
```

The postcondition is extended by the proof obligations from calling conventions. It also makes sure to have a non-growing stack after each function call. If this wrapper function can be verified, it will also ensure that the contract of the assembler routine matches the contract given in the C program. We can now call the function *bar_wrap* like every ordinary C function with a parameter and a return value. Everything concerning registers is completely hidden from the C function call. Note that those wrapper functions do not have to be introduced by the developer but are created by the translation tool fully automatically.

4.2.2 Memory

There exist two types of memory: heap memory and stack memory. In C, heap memory is allocated (and deallocated) by explicit allocation functions. For verification purposes, those allocation functions are extended by counting the allocated memory. Its existence allows information about the validity of pointers. Stack memory in contrast is only implicitly used. The compiler allocates it and uses it. The stack is completely hidden in the C language.

In Assembler, no explicit allocation functions exist. But in most of the Assembler programs (at least the whole Microsoft Hypervisor code) do not access arbitrary memory regions. Usually, memory is allocated in a C program. Assembler functions get those memory portions as a parameter. The precondition of the Assembler function requires that the memory is allocated. This has to be ensured by the calling function, which usually itself requires that the memory is allocated, unless it allocated the memory itself. Thus the allocatedness is propagated through the call stack of the program from the place where the alloc function was called in C.

4.3 Verification of Multi-threaded Software

Multi-threaded software is handled by VCC with a special memory model. The different threads have separated memory regions except common data structures. This is also an assumption that is usually used by developers of the software.

Shared data structures are usually protected by locks. When the structure is locked, the current thread becomes the owner of the data and can assume that it is the only thread changing the memory. When it releases the data structure, the structure is shared again and can then be locked by other threads. If the first thread locks and accesses the data again, it cannot assume that the content is still the same, since another thread may have changed it in between. Therefore, VCC assumes that its value changed non-deterministically.

This overapproximation would not allow verification, because the software would not be able to share any knowledge about this data structure. To make it more precise, an additional annotation is introduced. A data structure is annotated by data structure invariants. Those invariants give additional knowledge, e.g., a field of the data structure can only have a special range. If a data structure is locked by a thread, this data structure invariant can be temporarily violated. If the data structure is unlocked again, VCC checks that the data structure invariant holds again. If all access functions are verified, the special knowledge of the invariants is known to be valid, whenever the data structure is locked. VCC still assumes non-deterministic changes on the data structure, however, it assumes the data structure invariants satisfied. Access data structures that are not protected by locks or other mechanisms is similar, however, VCC checks that the invariant holds after each atomic operation.

For multi-threaded Assembler programs, we use this mechanism of VCC. The functional specification of the Assembler routine contains also a *writes* clause. If this writes clause does not contain a memory region, the Assembler code must not write into it. However, Assembler has no notion of data structures. If the Assembler routine writes into a memory region that contains a C data structure, the data structure must be locked before the Assembler routine is called. The Assembler routine cannot lock the data itself because the data structure and the virtual lock are not visible in Assembler. So the C code calling the assembler structure must lock the memory. VCC verifies that only locked data structures are changed. In the C code that releases the lock, VCC also proves that the data structure invariant is satisfied. This proof must follow inductively from the functional specification of the Assembler routine. In practice, this turned out to be an effective way for the Hypervisor verification.

Besides the concurrent memory access, we also want to handle multi-core

processors for the multi-threaded software. Therefore, we create multiple instances of our processor data structure. An instance is created with every thread creation and is then passed through the C program into the Assembler routines. Thus the C specification and the processor states of different threads can be separated. The main process can even compare the different core states.

4.4 Implicit Correctness Criteria

So far we considered functional properties. These were given explicitly by the programmer as pre- and postcondition of the function. Additionally, there are a lot of implicit proof obligations that are checked for every function. All these specifications have to be checked in a formal, mathematical way against the concrete implementation.

4.4.1 Memory safety

Memory safety means that memory access is only performed on valid (i.e., previously allocated and not freed) memory. Therefore, the precondition of an Assembler function needs to include the validity of all memory locations that are accessed. If a function tries to access memory that cannot be proven valid, VCC reports an error. Similarly, we specify explicitly the set of memory locations being written to, and it is an error to write to a memory location not listed in the writes clause. These properties are enforced to be transitive, i.e., if function f calls g then the writes set of g needs to be contained within the writes set of f, and also the precondition of g needs to follow from the context at the call site (including preconditions of f). The transitivity of the writes clauses is checked by the tool, but it has to be annotated by the developer.

4.4.2 Arithmetic safety

Arithmetic safety means absence of overflows, unless otherwise stated. For operations that can overflow (like addition, multiplication or signed division) VCC automatically adds assertions that check if the result is in the proper range. When an overflow behavior is desired, the user can specify this explicitly.

4.4.3 Call safety

Call safety means that the stack is cleaned up after every function call and registers are saved before every function call. If f calls g and the postcondition of g does not guarantee that it restores values of registers, then f needs to save itself the registers it cares about. The registers are saved on the stack, therefore

it is important to know that g will not modify stack locations above the current stack pointer (stacks grow downwards on the x86 architecture), and that g does not change the stack pointer at return. This is expressed using the postcondition *ensures(rsp == old(rsp))*.

Call safety also means to proof the calling conventions. Note that the condition that the stack pointer is unchanged will also follow from the normal calling conventions. The calling conventions result in a number of proof obligations for the verification tool. They are inserted automatically, so no manual interaction with a developer is needed.

4.4.4 Interrupt safety

Interrupt safety means that the stack is cleaned up after processing the whole interrupt. We cannot verify interrupt handlers like regular functions, because some of their subroutines push some registers on the stack, while other subroutines pop them later again from the stack. Only the whole interrupt routine (including all called subroutines) would satisfy that the stack size did not change.

Chapter 5

Case Study

Our main focus was on the Microsoft Hypervisor verification. Our aim was to verify the original Assembler code by only adding specifications. In many other projects, the source code was adapted (e.g. optimizations undone) to enable the verification tools to handle the problems. The Hypervisor policy only allows code changes that fix bugs and explicitly forbids changing actual code only for verification purposes.

Unfortunately, we cannot present arbitrary snippets from the Microsoft Hypervisor for legal reasons. Therefore, we will show our approach with the help of two examples: one function that is also in the Intel Optimization Guide, and one hand-written function which is only similar to a function in the Hypervisor.

5.1 Verifying Optimized Assembler Code

In this section, we will consider a function that sets a whole memory page to zero. This function is heavily used on memory allocation to prevent data leaking between two guest operating systems. Therefore, the function is written in highly optimized Assembler. It can be found in the Intel Optimization Guide and does not fall under the non-disclosure agreement with Microsoft.

The function takes the address of a memory page and fills it with zeros. A memory page in the x86 architecture is a block of 4096 bytes. In the code we use $X86\_PAGE\_SIZE$ to refer to that size. An example C implementation of the function is given in Figure 5.1 on the next page. The input parameter *base* is the address of the memory page.

In line 2–5, we give a functional specification for the C implementation. We need the precondition in line 2 to prevent arithmetic overflows in the implementation. The precondition in line 3 states that the memory page is allocated and writeable. With the annotation in line 4 we state that the function changes the

```
void set_to_zero (uint64* base)
requires ((uint64)base + (uint64)X86_PAGE_SIZE < (uint64)-1)
requires (assembler_is_mutable_array(base,X86_PAGE_SIZE/8))
writes (array_range(old(base),X86_PAGE_SIZE/8))
ensures ( forall (uint32 i; (0 <= i && i < X86_PAGE_SIZE/8) ==>
                            old(base)[i] == 0)))
{
        int32 count = X86_PAGE_SIZE / 8;
        while (count >= 0)
        {
                *(base) = 0;
                base++;
                count--;
        }
}
```

Figure 5.1: Example C implementation

```
        mov edx, X86_PAGE_SIZE / 8
label:
        mov [rcx], 0
        add rcx, 8
        dec edx      ;sets the zero flag if edx is equal to 0
        jnz label
```

Figure 5.2: Example Assembler implementation

memory page. The postcondition in line 5 ensures that the page contains only zeros after the execution of the function. Since the function changes the value of base, we denote its value on the entry by $old(base)$. The implementation is straight-forward, except that it writes a uint64 (8 bytes) at a time.

In Figure 5.2, a simple Assembler translation of the C code is depicted. In the Windows 64-Bit compiler conventions, the first parameter (*base* in the C code) is always transferred via the *rcx* register. In the implementation, register *edx* is used as the variable *count*. The instructions *dec* changes the flag register which the instruction *jnz* uses to conditionally jump back to *label*. Thus the loop was translated to a goto-program.

With this first translation, we will present and explain some of the optimizations made on this Assembler function. The most important instruction is the memory access $mov\ [rcx], 0$. The instruction has a length of 7 bytes if encoded in machine code. An instruction that assigns a register (with value 0) to the memory would only have length 5 instead. To have the value 0 in the register, an additional instruction has to be inserted at the beginning. We choose *rax* as register and set it to zero with $mov\ rax, 0$. This instruction would again have length 7. Instead of this, we choose the possibility of logic. Applying *XOR* on

```
        xor rax, rax
        mov edx, X86_PAGE_SIZE / 64
label:
4       mov [rcx], rax
        mov [rcx + 8], rax
        mov [rcx + 16], rax
        mov [rcx + 24], rax
        mov [rcx + 32], rax
9       mov [rcx + 40], rax
        mov [rcx + 48], rax
        mov [rcx + 56], rax
        add rcx, 64
        dec edx    ;sets the zero flag if edx is equal to 0
14      jnz label
```

Figure 5.3: Example Assembler implementation after loop unrolling

a value with itself results always in a 0. Instead of $mov\ rax, 0$, we could use $xor\ rax, rax$. This instruction has only length 4.

Processor manufacturers usually advice developers to use loop unrolling if the number of loop iterations is constant. This means to copy the body of the loop several times. However, it would be unwise to copy the instruction 512 times (4096/8), since this would unnecessarily fill the instruction cache. The optimal number of copies is given very precisely in the case of AMD and Intel. The memory access should be to contiguous 64 Bytes, which is equal to the size of the cache line in the design. In the function presented, all memory access is made to 64 Bit (or 8 Bytes) chunks. The loop should be unrolled 8 times in this case, because 8 times 8 Bytes results in memory access to 64 Bytes. After the described optimizations, the function will look as shown in Figure 5.3. Note that the specification of the Assembler function did not change although the function looks different now. The fact that the whole memory region is reset will not change in any of the optimizations, thus the specification will not change in any step.

Next, we will introduce an optimization aiming to help the branch prediction and pipelining of the processor. The effect of the jnz instruction depends on the arithmetic *dec* operation. In a pipelined architecture, the jnz would run in parallel to the *dec* instruction, but the processor takes time to compute the result of the *dec*. Therefore, the branch prediction unit of the processor will guess the upcoming result and continue execution with this guess. When the result of the *dec* instruction is ready, it will compare the result with the guess. If the guess was right, execution is already going on in the right branch. Otherwise, the processor will flush the execution and has to start over again with the right branch. It is preferable to know the result instead of guessing. To have the

```
1       xor rax, rax
        mov edx, X86_PAGE_SIZE / 64
label:
        mov [rcx], rax
        mov [rcx + 8], rax
6       mov [rcx + 16], rax
        add rcx, 64
        mov [rcx + 24 − 64], rax
        mov [rcx + 32 − 64], rax
        dec edx   ;sets the zero flag if edx is equal to 0
11      mov [rcx + 40 − 64], rax
        mov [rcx + 48 − 64], rax
        mov [rcx + 56 − 64], rax
        jnz label
```

Figure 5.4: Example Assembler implementation according to the optimization guides

result of the arithmetic operation at the conditional jump, the *dec* instruction has to move up further in the code. This can be done because the assignment instructions do not affect the flags of the processor. Since *add* also changes the flags, this instruction must appear before the *dec*. To move the *add* further up, we need to adjust the constants in the memory accesses that come afterwards. According to the Intel manual, the dependency of the address in the memory access on the result of the *add* instruction will not result in extra penalties. The resulting code is shown in Figure 5.4.

The code shown in Figure 5.5 includes the loop invariant which is necessary to verify the function against its specification automatically. Note that one of the most difficult problems is finding the loop invariants. Most of the invariants are straight-forward. The invariant in line 16 is necessary to guide the theorem prover.

Our tool Vx86 generates the code in Figure 5.6 on page 64 from the Assembler code in Figure 5.5 on the facing page. The pre- and postconditions are coming from a C header file declaring the function. In the specification, the parameter *base* is replaced by *rcx* according to the compiler conventions. On an Intel Core2 machine, the verification of the example presented takes about 30 seconds. The bottleneck of the verification is not the translation of the Assembler instruction but the background theory which is sitting in the *assembler_is_mutable_array* predicate.

```
1   ;^ spec (uint64 offset = 0);
            xor rax, rax
            mov edx, X86_PAGE_SIZE / 64
    label:
    ;^      invariant (0 < rdx && rdx <= (uint64)(X64_PAGE_SIZE/64))
6   ;^      invariant (rax == 0)
    ;^      invariant (offset + 8*rdx == X64_PAGE_SIZE/8)
    ;^      invariant ((old(rcx))-64*rdx+X64_PAGE_SIZE == rcx)
    ;^      invariant (((uint64*)(old(rcx)))+offset == (uint64*)rcx)
    ;^      invariant (((uint64*)rcx) == (((uint64*)rcx)+0))
11  ;^      invariant (assembler_is_mutable_array((uint64*)(old(rcx)),
                            (X64_PAGE_SIZE/8)))
    ;^      invariant ((uint64*)(old(rcx)) <= (uint64*)rcx)
    ;^      invariant ((uint64*)rcx < (uint64*)(old(rcx))+X64_PAGE_SIZE/8)
    ;^      invariant (forall(uint32 i; (0 <= i && i < offset) ==>
16                                  old(rcx)[i] == 0)))
                    mov [rcx], rax
                    mov [rcx + 8], rax
                    mov [rcx + 16], rax
                    ;^ assert(((uint64*)rcx)+8 == (uint64*)(rcx+64));
21                  add rcx, 64
                    mov [rcx + 24 - 64], rax
                    mov [rcx + 32 - 64], rax
                    dec edx    ;sets the zero flag if edx is equal to 0
                    mov [rcx + 40 - 64], rax
26                  mov [rcx + 48 - 64], rax
                    mov [rcx + 56 - 64], rax
                    ;^ spec ({ offset += 8; })
                    jnz label
```

Figure 5.5: Annotated source of the example function

```
1  void set_to_zero ()
   requires (assembler_is_mutable_array((uint64*)rcx,x64_page_size/8))
   requires (rcx+(uint64)x64_page_size < (uint64)-1)
   writes (array_range((uint64*)(old(rcx)),x64_page_size/8), &core)
   ensures ( forall (uint32 i; (0 <= i && i < X86_PAGE_SIZE/8) ==>
6                              old(rcx)[i] == 0)))
   {
          spec (uint64 offset = 0);
          xor (rax,rax);
          mov (rdx,X64_PAGE_SIZE / 64);
11        while()

          invariant (0 < rdx && rdx <= (uint64)(X64_PAGE_SIZE/64))
          invariant (rax == 0)
          invariant (offset + 8*rdx == X64_PAGE_SIZE/8)
16        invariant ((old(rcx))-64*rdx+X64_PAGE_SIZE == rcx)
          invariant (((uint64*)(old(rcx)))+offset == (uint64*)rcx)
          invariant (((uint64*)rcx) == (((uint64*)rcx)+0))
          invariant (assembler_is_mutable_array((uint64*)(old(rcx)),
                                (X64_PAGE_SIZE/8)))
21        invariant ((uint64*)(old(rcx)) <= (uint64*)rcx)
          invariant ((uint64*)rcx < (uint64*)(old(rcx))+X64_PAGE_SIZE/8)
          invariant ( forall (uint32 i; (0 <= i && i < offset) ==>
                                        old(rcx)[i] == 0)))
          {
26                mov (*((uint64*)rcx),rax);
                  mov (*((uint64*)rcx + 1),rax);
                  mov (*((uint64*)rcx + 2),rax);
                  assert (((uint64*)rcx)+8 == (uint64*)(rcx+64));
                  add (rcx,64);
31                mov (*((uint64*)rcx + 3 - 8),rax);
                  mov (*((uint64*)rcx + 4 - 8),rax);
                  dec (rdx);
                  mov (*((uint64*)rcx + 5 - 8),rax);
                  mov (*((uint64*)rcx + 6 - 8),rax;
36                mov (*((uint64*)rcx + 7 - 8),rax);
                  spec ({ offset += 8; })
                  jnz (label);
          }
   }
```

Figure 5.6: C code translation of the Assembler code

```
struct S{
        uint64 dr0;
        uint64 dr1;
        uint64 dr2;
        uint64 dr3;
}

void set_dr(struct S* structure)
ensures(structure->dr0 == dr0)
ensures(structure->dr1 == dr1)
ensures(structure->dr2 == dr2)
ensures(structure->dr3 == dr3);
```

Figure 5.7: C header file containing a structure and a specification

```
        FIELD_DR0 equ 0
        FIELD_DR1 equ 8
        FIELD_DR2 equ 16
        FIELD_DR3 equ 24

set_dr proc
        mov [rcx + FIELD_DR0],dr0
        mov [rcx + FIELD_DR1],dr1
        mov [rcx + FIELD_DR2],dr2
        mov [rcx + FIELD_DR3],dr3
```

Figure 5.8: Field access from Assembler to a C data structure

5.2 Translation of C Specifications

In this section, we will describe how to translate a C specification into a valid Assembler specification. In the Hypervisor, there are functions that take as an argument a pointer to a C structure and are implemented in Assembler. However, there is no concept of structures in Assembler. On the other hand, Assembler uses processor registers which are not visible in C but the specification needs to mention them. We use the following simple example to demonstrate the problem and our solution.

We consider a function *set_dr* that copies the values of the debugging registers into a C data structure. The specification is given in a C file depicted in Figure 5.7. In line 9, the debugging register of the processor is referred to with *dr0*. We allow access to the data structure in normal C syntax.

The implementation is given in a separate Assembler file as shown in Figure 5.8. The Macro Assembler does not support structures. Therefore, we need to define constants that are used as offsets for memory accesses. This is the original code, written by an Assembler programmer.

```
#define FIELD_DR0 0
#define FIELD_DR1 8
#define FIELD_DR2 16
#define FIELD_DR3 24

void set_dr()
ensures(*((uint64*)rcx + 0) == dr0)
ensures(*((uint64*)rcx + 1) == dr1)
ensures(*((uint64*)rcx + 2) == dr2)
ensures(*((uint64*)rcx + 3) == dr3)
{
        mov (*((uint64*)rcx + FIELD_DR0/8),dr0);
        mov (*((uint64*)rcx + FIELD_DR1/8),dr1);
        mov (*((uint64*)rcx + FIELD_DR2/8),dr2);
        mov (*((uint64*)rcx + FIELD_DR3/8),dr3);
}
```

Figure 5.9: C translation of an Assembler function accessing C data structure fields

Our tool Vx86 takes the two files and produces the code depicted in Figure 5.9. The first part of the translation is the translation of the parameter transfer. This can be done by a renaming step, where all occurrences of the first parameter are replaced by *rcx* and in this case a cast to a pointer structure. Translating the data structure access is more complex and requires pointer arithmetic. We use the address of the structure as the base address and add an offset representing the dereferenced field. Vx86 uses the code from VCC to compute this offset. This approach ensures that the alignment of the fields is the same as used by the C compiler. The verification tool can find out whether the two constants (the one resulting from the C specification and the one in the Assembler code) are equal. This is a direct consequence of our approach.

5.3 Statistics for the Hypervisor Verification

On the code base of the Microsoft Hypervisor, we were able to prove the correctness of low-level Assembler functions. They were specified by C contracts and implemented by Assembler code. The contracts were used for the verification of the correctness of the C code base. The C contracts were automatically transferred to Assembler contracts and verified against the code base by the methodology presented in this work. We were also able to find errors in the code base, where the specification was not fulfilled by the code. They are now fixed in the current version of the product. The Macro Assembler consists of 5k lines of code, the Assembler consists of 15k lines of code after inlining the macros. The source code is separated into 21 files. The function length varied

File	Annotated ASM	Preproc ASM	Translated C
zero	7,821	16,781	18,040
crashdump	5,625	17,791	20,217
GuestContext	1,422	16,673	18,136
Trap	82,316	420,854	444,865

Figure 5.10: File sizes in bytes at different stages of the translation

Filename	Verification Time [s]
zero	1.45
crashdump	3.29
GuestContext	≤ 0.01
Trap	67.20

Figure 5.11: Verification times for various files of the Microsoft Hypervisor

from only 3 lines to 1400 lines of code. We had to handle nested loops up to a depth of 3. Loop invariants needed for the verification typically consisted of up to 10 conjuncts, where nested loops are exceeding this number.

We were able to verify 14k of the 15k lines of Assembler code with the presented method. The methodology is not suitable for switches between different virtual memory systems, which are present in the Microsoft Hypervisor at thread switches and guest switches. Such switches are only available in operating system code, so other programs would verify completely with this method.

Figure 5.10 presents the size in bytes of four different files in different processing stages. The column *Annotated ASM* denotes the size in bytes of the annotated Macro Assembler function, i.e. the files the user edits. The column *Preproc ASM* shows the file sizes after Assembler macro expansion. The column *Translated C* shows the file size after the translation to C. This is the file that includes the syntax translation. The processor model, consisting of the processor state definition and the instruction semantics, has a size of 131k bytes. On the translated C files, we call VCC for verification. The translation itself takes only mentionable times for very large files, i.e. the file *Trap* takes 2 seconds for the translation.

Figure 5.11 shows the verification times for four example files of the Microsoft Hypervisor. As long as the procedures do not grow too much in size and do not introduce too many control flow paths, we can verify substantial code. For instance, the procedure *ExceptionDispatch*, which is part of the *Trap* file, has approximately 300 instructions. Nevertheless, it verifies in approximately 3 seconds. This shows that our approach cannot be used only for short toy functions but also for long and complex Assembler implementations. Verification of

the code took under 10ms for smaller functions and up to 10 minutes for the largest function.

Chapter 6

Conclusion and Future Work

Vx86 is a verifier for proving the correctness of concurrent Intel x86 Assembler code with AMD virtualization extensions against their contracts. Our approach has been to (1) provide a C simulator for Intel x86, and (2) to translate the annotated Assembler code into C code for this simulator. Despite the fact that providing a C simulator seems to be a detour for verifying Assembler code, it has turned out that this still allows us to verify the Assembler portion of a complex industrial program, like Microsoft Hypervisor, in reasonable time.

In the process of developing Vx86 we have learned several characteristics of handwritten Assembler programs: they might have complex control flow, but they operate only on a few registers and the memory; the operations on registers are often low-level, in addition operations have many side effects. Recursive data structures, which typically need transitivity to describe effects on them, are rarely used in handwritten Assembler code. As a consequence, changes to registers and the memory can often easily be described by enumeration and quantification.

We have also learned that Assembler code is particularly well suited for automated verification:

- Providing an Assembler verifier is overdue – except for Assembler compilers there are no tools to help Assembler writers.
- Verifying Assembler code is beneficial – if Assembler code fails, systems typically crash.
- Writing Assembler contracts is feasible – the contracts often only mention a limited amount of objects. Furthermore, the contracts are often easier

to write than the highly optimized implementation.

- Discharging Assembler contracts is a sweet spot for Automated Theorem Provers (ATPs) – ATPs can deal well with high quantities of low level details, since they often have specialized decision procedures for them. They can also deal well with quantifiers; however they often cannot deal well with complex heap structures. Luckily user written Assembler programs do not use them.

- Verifying the calling conventions enables the safe use of compilers for mixed executables

- Executing C programs relies on the calling conventions for Assembler programs, and mixed executables are the standard case in operating systems.

Vx86 is of course not restricted to consume only Microsoft Hypervisor code, it can be used to verify other code bases as well. Providing such code bases is not easy, because most companies have difficult copyright situations. Abstraction from the real code base is not very helpful, because the interesting algorithms must remain intact for verification purposes. Furthermore, we think that the presented approach is a viable way to quickly provide verifiers for other processors.

Our work opens up multiple directions for future work. It could be interesting to investigate whether our approach would be able to handle automatically generated code. Optimized code could include irreducible control flow. Currently, we are not dealing with such code. It is not clear whether the used tools could work with such code. This could enable approaches like PCC and TAL to use the stronger proof system of our approach.

It would also be interesting to use different case studies. VCC is already able to deal with C compliant source code, so many source code projects could be used. For the Assembler code, this is more difficult because open source projects are usually written for the GNU compilers. GNU uses the AT&T notation while we are using the Intel notation. This would mean a change in the parser. The approach itself, containing the processor model as well as the instruction description, would stay the same as described in this thesis. The result would be interesting because of three reasons:

1. verification of projects would lead to better products

2. verification of normal programs would not introduce the difficult assumptions on the different virtual memory areas

3. verification of normal programs does not include the verification of different address spaces and so that our assumption in this direction would not be of importance.

Follow up work to the work presented in this thesis was presented in [10]. Our methodology was used there for the verification of PikeOS which is a microkernel with paravirtualization. Paravirtualization means that instead of an ordinary operating system used as guest system, the guest systems is developed especially for this application. The task of the hypervisor does not have to run in an additional privileged mode as in Microsoft Hypervisor.

In [10], our approach was extended to accept inline Assembler. Inline Assembler implies more transitions from C to Assembler than in our scenario with separated functions. The resulting approach uses partly manual work where we use a fully automatic approach. They also use the idea of a simulation for the processor, in their case the PowerPC architecture.

Bibliography

[1] Bart Jacobs 0002, Frank Piessens, Jan Smans, K. Rustan M. Leino, and Wolfram Schulte. A programming model for concurrent object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 31(1), 2008.

[2] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert Schirmer, and Artem Starostin. The verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2008.

[3] Gogul Balakrishnan, Radu Gruian, Thomas W. Reps, and Tim Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In Rastislav Bodík, editor, *CC*, volume 3443 of *Lecture Notes in Computer Science*, pages 250–254. Springer, 2005.

[4] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2004.

[5] Gogul Balakrishnan and Thomas W. Reps. Analyzing stripped device-driver executables. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 124–140. Springer, 2008.

[6] Gogul Balakrishnan, Thomas W. Reps, Nicholas Kidd, Akash Lal, Junghee Lim, David Melski, Radu Gruian, Suan Hsi Yong, Chi-Hua Chen, and Tim Teitelbaum. Model checking x86 executables with codesurfer/x86 and wpds++. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2005.

[7] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs 0002, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In de Boer et al. [25], pages 364–387.

[8] Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs 0002, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The spec# programming system: Challenges and directions. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer, 2005.

[9] Mike Barnett, K. Rustan M. Leino, and Wolfram Schule. The spec# programming system: An overview. In *CASSIS*, pages 49–49, 2004.

[10] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In *SAFECOMP*, pages 187–200, 2009.

[11] William R. Bevier, Warren A. Hunt Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.

[12] Robert S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 1992.

[13] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In de Boer et al. [25], pages 342–363.

[14] Juan Chen, Chris Hawblitzel, Frances Perry, Michael Emmi, Jeremy Condit, Derrick Coetzee, and Polyvios Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers. In Gupta and Amarasinghe [36], pages 183–192.

[15] Ernie Cohen, Eyad Alkassar, Vladimir Boyarinov, Markus Dahlweid, Ulan Degenbaev, Mark A. Hillebrand, Bruno Langenstein, Dirk Leinenbach, Michal Moskal, Steven Obua, Wolfgang J. Paul, Hristo Pentchev, Elena Petrova, Thomas Santen, Norbert Schirmer, Sabine Schmaltz, Wolfram Schulte, Andrey Shadrin, Stephan Tobies, Alexandra Tsyban, and Sergey Tverdyshev. Invariants, modularity, and rights. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2009.

[16] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer,

Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[17] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2010.

[18] Ernie Cohen, Michal Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for c. *Electr. Notes Theor. Comput. Sci.*, 254:85–103, 2009.

[19] Karl Crary and J. Gregory Morrisett. Type structure for low-level programming languages. In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *ICALP*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 1999.

[20] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. Vcc: Contract-based modular verification of concurrent c. In *ICSE Companion*, pages 429–430. IEEE, 2009.

[21] Matthias Daum, Jan Dörrenbächer, Mareike Schmidt, and Burkhart Wolff. A verification approach for system-level concurrent programs. In *VSTTE*, pages 161–176, 2008.

[22] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *J. Autom. Reasoning*, 42(2-4):349–388, 2009.

[23] Matthias Daum, Stefan Maus, Norbert Schirmer, and M. Nassim Seghir. Integration of a software model checker into Isabelle. In Geoff Sutcliff and Andrei Voronkov, editors, *LPAR*, volume 3835 of *Lecture Noten in Computer Science*, pages 381–395. Springer, 2005.

[24] Matthias Daum, Norbert Schirmer, and Mareike Schmidt. From operating-system correctness to pervasively verified applications. In Dominique Méry and Stephan Merz, editors, *IFM*, volume 6396 of *Lecture Notes in Computer Science*, pages 105–120. Springer, 2010.

[25] Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors. *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*. Springer, 2006.

[26] Leonardo de Moura and Nikolaj Bjøner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[27] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Pfenning [59], pages 183–198.

[28] Leonardo Mendonça de Moura and Nikolaj Bjørner. Proofs and refutations, and z3. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *LPAR Workshops*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[29] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In Pottier and Necula [61], pages 67–78.

[30] Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In Olivier Danvy and Benjamin C. Pierce, editors, *ICFP*, pages 254–267. ACM, 2005.

[31] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In Gupta and Amarasinghe [36], pages 170–182.

[32] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[33] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.

[34] Mauro Gargano, Mark A. Hillebrand, Dirk Leinenbach, and Wolfgang J. Paul. On the correctness of operating system kernels. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[35] Denis Gopan and Thomas W. Reps. Low-level library analysis and summarization. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 68–81. Springer, 2007.

[36] Rajiv Gupta and Saman P. Amarasinghe, editors. *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. ACM, 2008.

[37] Chris Hawblitzel, Heng Huang, Lea Wittie, and Juan Chen. A garbage-collecting typed assembly language. In Pottier and Necula [61], pages 41–52.

[38] Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 441–453. ACM, 2009.

[39] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. 2007.

[40] Laurie J. Hendren, editor. *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4959 of *Lecture Notes in Computer Science*. Springer, 2008.

[41] Gerwin Klein. The l4.verified project - next steps. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 2010.

[42] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[43] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP*, pages 207–220. ACM, 2009.

[44] Rafal Kolanski and Gerwin Klein. Formalising the l4 microkernel api. In Joachim Gudmundsson and C. Barry Jay, editors, *CATS*, volume 51 of *CRPIT*, pages 53–68. Australian Computer Society, 2006.

[45] Hermann Lehner and Peter Müller. Formal translation of bytecode into boogiepl. *Electr. Notes Theor. Comput. Sci.*, 190(1):35–50, 2007.

[46] Dirk Leinenbach, Wolfgang J. Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 2–12. IEEE Computer Society, 2005.

[47] K. Rustan M. Leino. Designing verification conditions for software. In Pfenning [59], page 345.

[48] K. Rustan M. Leino. Specifying and verifying software. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer 0002, editors, *ASE*, page 2. ACM, 2007.

[49] K. Rustan M. Leino. This is boogie 2. 2009.

[50] K. Rustan M. Leino and Angela Wallenburg. Class-local object invariants. In Gautam Shroff, Pankaj Jalote, and Sriram K. Rajamani, editors, *ISEC*, pages 57–66. ACM, 2008.

[51] Jochen Liedtke. On micro-kernel construction. In *SOSP*, pages 237–250, 1995.

[52] Junghee Lim and Thomas W. Reps. A system for generating static analyzers for machine instructions. In Hendren [40], pages 36–52.

[53] Chunxiao Lin, Andrew McCreight, Zhong Shao, Yiyun Chen, and Yu Guo. Foundational typed assembly language with certified garbage collection. In *TASE*, pages 326–338. IEEE Computer Society, 2007.

[54] Stefan Maus, Michal Moskal, and Wolfram Schulte. Vx86: x86 assembler simulated in c powered by automated theorem proving. In *AMAST*, pages 284–298, 2008.

[55] Oleg Mürk, Daniel Larsson, and Reiner Hähnle. KeY-C: A tool for verification of C programs. In Pfenning [59], pages 385–390.

[56] George C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[57] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 320–333. ACM, 2006.

[58] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2007.

[59] Frank Pfenning, editor. *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*. Springer, 2007.

[60] Erik Poll. Teaching program specification and verification using jml and esc/java2. In Jeremy Gibbons and José Nuno Oliveira, editors, *TFM*, volume 5846 of *Lecture Notes in Computer Science*, pages 92–104. Springer, 2009.

[61] François Pottier and George C. Necula, editors. *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*. ACM, 2007.

[62] Thomas W. Reps and Gogul Balakrishnan. Improved memory-access analysis for x86 executables. In Hendren [40], pages 16–35.

[63] Thomas W. Reps, Gogul Balakrishnan, Junghee Lim, and Tim Teitelbaum. A next-generation platform for analyzing executables. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2005.

[64] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[65] Aleksy Schubert and Jacek Chrzaszcz. Esc/java2 as a tool to ensure security in the source code of java applications. In Krzysztof Sacha, editor, *SET*, volume 227 of *IFIP*, pages 337–348. Springer, 2006.

[66] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hoffmann and Matthias Felleisen, editors, *POPL*, pages 97–108. ACM, 2007.

[67] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 99–110. ACM, 2010.

Appendix A

x86 Processors

In this chapter we will give a brief description of how the x86 architecture evolved. The chapter is thought to give a deeper insight into the processor complexity. Only Intel processors are described here, although AMD has its own processors from 486 on. The different processor architectures from the companies Intel and AMD met later again, only to invent two new architectures afterwards. The Core i7 architecture is not mentioned, because it was presented only in the latest parts of the work and is not supported especially by our approach at the moment. Most of the information presented here was obtained from *Intel®64 and IA-32 Architectures Software Developers Manual Volume 1: Basic Architecture*. Note that in the tables with processor descriptions, the abbreviation *P* stands for *Pentium*.

With every processor generation, some extensions were made to the architecture. Furthermore, all older inventions are supported by the newer implementations. There was only one exception to that, and this exception has never been used in mainstream computers but only in embedded systems, like printers. This processor was called 80186, it is usually not counted as part of the x86 architecture. Some of the older instructions do not have much meaning in 64–Bit native mode, but they are needed for some of the compatibility modes. Thus it is of some interest to see how the architecture grew.

A.1 8086/8088 (1978)

Name	Date	Max Clock	Transistors	External Size	Address Space	Caches
8086	1978	8M	29k	16Bit	1MB	none

When 8086 was invented, the processor was not a complete new design. In-

stead, it was based on the older 8–Bit processor, 8080. It was not backwards compatible, so 8080 programs could not be executed on 8086. The instructions were, however, very similar and programs could be ported easily to the new processor. Note that 8086 architecture was not one of the fastest 16–Bit implementations that existed at the end of 70s.

When 8086 was sold, some people demanded a somehow simpler version of the processor. Therefore, Intel invented the 8088 processor, this is a 16–Bit processor that has only 8–Bit access to the outside. Thus, the processor needs 2 memory accesses to transfer only one 16–Bit value: first the upper 8–Bits, then the lower 8–Bits. The 8088 processor had only half the speed to the outside world than the 8086.

The operation mode introduced by the 8086 processor is called *Real Mode* today. It is not only the oldest mode used, but it is also the mode the processor starts in. This fact comes from some old compatibility issues Intel introduced when they extended their architecture.

Memory access The processor 8086 has 16–Bit wide registers. Thus, only 64KB of memory could be accessed. To avoid such limitations, the idea of segmentation was invented. A memory access is not only made with one register, but with two. With two registers, one could simply put one register as higher 16–Bit and another as lower 16–Bit and thus address 32–Bit. That was not the version implemented at that time. Instead, one had 20–Bit memory addresses. To get such 20–Bit values, the registers were nested by 4 bits. This means, the *segment* register was shifted by 4 and added to the *offset* register. This enables 20–Bit values, if one ignores the fact that this can lead to some additional larger values. If the value exceeds the 20–Bit value, it is wrapped by just ignoring the highest bit.

Registers and external communication The 8086 has eight 16–Bit registers, including a stack pointer. The instruction pointer cannot be used directly in programs; it has to be manipulated by jump instructions. Four of those eight registers can also be accessed as two 8–Bit registers each, one with the higher 8–Bits and one with the lower ones. These registers are named AX, BX, CX, and DX, the corresponding 8–Bit registers are called AH and AL, BH and BL, and so on. The other 4 registers (BP, SP, DI, and SI) are 16–Bit only registers. The stack pointer SP points to the stack. $PUSH$ and POP operations access this stack with hardware support, growing down. The stack architecture has not changed until today.

Communication to the outside world (memory, DMA–controller, interrupt–controller, ...) is done via a bus with 16–Bit and a full speed of 8 MHz. There

is no computation or cache needed for this part of the processor.

A.2 80286 (1982)

Name	Date	Max Clock	Transistors	external Size	Address Space	Caches
80286	1982	12.5M	134k	16Bit	16MB	none

The architecture of 80286 was able to calculate most of the instructions in only half the time required by the 8086. The performance increase of 80286 was, perhaps, the largest ever in the x86 architecture. It also introduced the *Protected Mode*, where a hardware memory management unit (MMU) was enabled. In *Real Mode*, the 80286 behaves like a fast version of 8086. Extensions can only be used in the newly introduced mode. 80286 was invented to support multitasking operating systems. However, most of the time the "mainstream" computers run them in *Real Mode*.

Registers and external communication did not change compared to 8086. One interesting famous feature was introduced at that time: switching from real to protected mode was supported by an instruction, switching back was not. Switching back from protected mode to real mode had to be done via a hardware reset. Perhaps Intel did not think that one would ever come back to real mode. But in reality this happened very often. Note that the operating system used was *DOS* at that time and it did only work in real mode. Every protected mode application had to switch to real mode if an interrupt ocured. If you consider clock interrupts that occur with 14kHz frequency, those switches are everything but rare. The fastest possibility was to initiate a "soft" boot, which only resets the CPU itself without resetting any peripheral devices. This introduced the famous "Gate A20" problem that is used to switch back to real mode until today (although today, one really does not need to switch back).

In principle, old real mode applications should work within protected mode, if one would respect some limitations. In real world applications, those limitations were violated all the time, thus nearly no real mode application did work in protected mode. Intel did not tackle that problem, but had the hope that the world would switch to new applications and operating systems very soon.

Memory Access Again, 80286 has only 16–Bit wide registers. To access the full 24–Bit physical address space, Intel did not extend the old idea from 8086. Instead, a hardware memory management unit was introduced. The segment register was used as a pointer into a table. This table was used by the MMU to get the value for the computation of the physical address. The idea is to

concatenate two 16–Bit values (although not all 16–Bits were used).

A.3 80386 (1985)

Name	Date	Max Clock	Transistors	External Size	Address Space	Caches
i386DX	1985	20M	275k	32Bit	4GB	none

With the introduction of 80386, the 32–Bit world came to the x86 architecture. The x86 architecture with those 32–Bit extensions is called IA–32, like *I*ntel *A*rchitecture 32–Bit. From an architectural point of view, this was one of the largest and most important changes. The *real mode* and *protected mode* were again supported, as in 80286. The *protected mode* was extended to the *extended mode* in order to support a physical address space of 4GB.

Note that today, an additional running mode exists, called *virtual 8086 mode*, short *VM86*. This mode is a submode of protected mode, but it exhibits a behavior similar to the real mode. I.e., only 16–Bit registers are available, only 1 MB of memory can be addressed, etc. This was introduced to run old real mode applications in protected mode. This is important to run old programs under a new operating system and to avoid switching from protected mode to real mode.

Memory Access In extended mode, the processor knows two different modes. The first one is called *flat memory model*. In this mode, all addresses are given with a 32–Bit value, a pointer into the 32–Bit linear address space. The second one is called *segmented memory model*. The memory is divided into 16,383 segments, selected by a so called segment selector (this is only a part of a segment register), and in this segment part the address is computed with a 32–Bit offset. The MMU has to translate logical addresses (containing only a segment selector and an offset) to linear addresses for the processor.

Linear addresses do not mean physical addresses. Memory can also be swapped to a hard disk. Therefore, the memory is divided into so called *pages* that are 4,096 bytes each. Such pieces of the memory can be located either in physical memory or on a hard disk, for each access to a linear address; a so called *page fault* can occur if the memory is not located in the physical memory. Page faults have to be handled by software to load the content from hard disk to memory. To make room for such contents, it is likely that another page has to be swapped out from the physical memory to hard disk. There exist translation tables that show where those pages can be found. The *linear* address does not mean that all neighboring pages are contiguous in the physical address space.

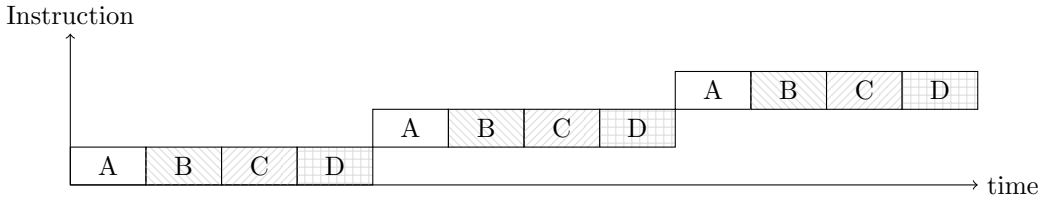

Figure A.1: non-pipelined processor

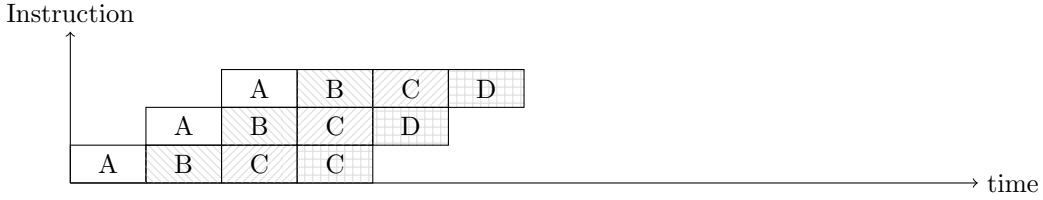

Figure A.2: pipelined processor

Only after the computations of the MMU that looks up the address translation, the location in physical memory can be seen.

Registers and External Communication Obviously, the register size was extended to 32–Bit. The new 32–Bit registers have names with an E in front of the old name, for example EAX. The old names are used to access the registers in their old state: 16–Bit or 8–Bit wide. Note that accessing 16–Bit or 8–Bit versions of a register only change those bits, all other bits are unchanged (as one would expect). For external communication, the buses were also extended by additional 16–Bits.

A.4 i486 (1989)

Name	Date	Max Clock	Transistors	External Size	Address Space	Caches
i486	1989	25M	1.2M	32Bit	4GB	L1: 8K

The Intel 486 processor brought changes that were less obvious. It was the first processor that had a floating point unit build into the processor (this was only available as a separate processor before). And it was the first pipelined processor in the x86 architecture.

In Figure A.1 and Figure A.2, one box labeled A, B, C, or D is representing one stage of the processor. One large box containing four of the smaller

boxes represents one complete instruction / In non–pipelined or fully sequential processors (see Figure A.1 on the preceding page), the processor executes one instruction. Only after the whole instruction is finished, the next one can be processed. This means that the memory (which holds the instruction) is waiting for work most of the time as well as most parts of the processor. In pipelined mode (see Figure A.2 on the previous page), the processor is divided into different stages. As an instruction progresses from one stage to another, the now empty stage can be refilled again with a new instruction. For example, in the second step, the first instruction has reached the stage called *B*, while the second instruction can start in stage *A*. This leads to a more effective execution in the processor.

A.5 Pentium (1993)

Name	Date	Max Clock	Transistors	External Size	Address Space	Caches
Pentium	1993	60M	3.1M	64Bit	4GB	L1: 16KB L2: 256 or 512K

The Pentium processor added a second execution pipeline to the 486 concept, which lead to superscalar performance. The instructions have to be filled into two pipelines, assuming that there are no dependencies of the instructions. Guessing which instructions are not coupled is one of the most important problems in this case. When the Pentium processor was released, no optimizations for that decoupling were implemented in compilers, and this did not lead to increasing performance in any case. The processor got more popular when it got higher clock speeds and the compilers had more optimizations built in. Pentium processors brought another important feature for future development: they got an APIC to support multiple processor systems. Although this was not very important for people at that time, it became more interesting for later processors.

External Communication E/ven for 486, the external periphery was not able to work at the same clock speed as the processor (for example memory was too slow). In Pentium processors, the problem became even worse. Intel decided to improve the outside performance by extending the external communication paths to 64–Bit. This does not mean that addresses were widened. Only the amount of data that was transferred at one step was doubled. It also introduced a so called “burst” mode. For normal mode, external communication is always made with a pair of address and data. In burst mode, only one address is used

for five consecutive data updates. This does lead to faster transfers because the addresses do not have to be processed each time.

A.6 P6 Family (1995-1999)

The P6 family is not just one single CPU but included different evolution steps. Altogether, they are based on a superscalar microarchitecture. Each of the pipelines of the processor has 12 stages. The main features consist of branch prediction, out–of–order execution, and speculative execution.

Branch prediction is performed whenever a branch in the execution can occur. From a high level point of view, an if–then–else construct could be seen as such a thing. A condition is checked and the execution path depends on the outcome of that check. The pipeline has to be filled efficiently, so the processor has to guess what to do next. The answer to that question is the branch prediction that tries to predict the correct path. Speculative execution can then be used to execute the path. If the outcome of the condition is known and leads to the opposite execution path, the pipeline has to be flushed and everything has to be deleted from all stages. Software that is optimized on the branch prediction of a processor is executed much more efficiently than arbitrary software.

If multiple pipelines exist, the execution of some instructions may depend on the result of instructions that are currently in some pipeline. This means the execution cannot be parallelized, and the new instruction has to wait until the input is available. If this occurs, out–of–order execution can be used. This means the order of the sequential program is changed and instead of the next instruction, the next instruction is executed. To execute exactly like a sequential processor, some unit has to rebuild the results again in the right order. /

A.6.1 Pentium Pro

Name	Date	Max Clock	Transistors	External Size	Address Space	Caches
P Pro	1995	200M	5.5M	64Bit	64GB	L1: 16KB L2: 256 or 512K

The Intel Pentium Pro processor is three-way superscalar. Three-way superscalar means that the processor can handle three instructions per clock cycle in average. Average means that if branch prediction is wrong, this value cannot be reached. An additional performance boost was reached by implementing a L2–cache on the processor die. The advantage of such caches depends on the

software that is executed.

Note that the address space was extended. It was not done by extending to full 64-Bit physical address space, but again by introducing segmentation. The processor could now have 16 segments with 4GB each. This extension is called *Physical Address Extension* (PAE) by Intel. The processor first chooses the segment. Then, it addresses the 4GB address space by normal access. For operating systems and programs, this means only a limited extension. Every program still has a memory limit of about 2GB, but more programs can use up their limit, because then something like a fast swapping would take place. A real 64-Bit extension was only introduced in the late Pentium IV era and the Core2 processors.

A.6.2 Pentium II

Name	Date	Max Clock	Transistors	External Size	Address Space	Caches
P II	1997	266M	7M	64Bit	64GB	L1: 32KB L2: 256 or 512K

The multimedia extensions were introduced to the P6–family by the Pentium II. Those extensions also needed some new registers, called MMX. Depending on the software this can be a big improvement for performance. They are called multimedia extensions because they were mainly used to improve programming of video compression and similar software. The effects of the MMX instructions are difficult to describe. Assuming that one wants to increase the values of 4 16–Bit registers by 1, you could have 4 instructions and then increase every register. But one could improve performance by putting the four registers into one register (4 time 16 means 64–Bit) and then have an instruction that adds a mask to the value. The result would be correct if overflows of the 4 segments would be ignored. The MMX instruction set consists of instructions, that are doing logical or arithmetic computations as described.

Another big step in the x86 history was the introduction of low power states. Although they were only beginnings on lowering the power consumption, they enabled later improvements. Those power saving states became more interesting with newer CPUs. When the first CPUs had a power consumption of around 1W, newer CPUs have power consumptions of up to 150W. This amounts to a lot of saving possibilities.

A.6.3 Pentium II Xeon

The Xeon family exists until today. They are modifications of the mainstream processors. An example for improvements is the cache performance. While the cache runs at half speed in Pentium II processors, in Pentium II Xeon the cache run with full processor speed.

A.6.4 Celeron

The Celeron family had a longer life time. Celeron versions exist of Pentium II, Pentium III and Pentium IV processors. It always represents a low budget version of the corresponding main stream CPU. It is usually limited by the cache size and clock speed. Main features of the processor stayed as in the corresponding main processor.

A.6.5 Pentium III

Name	Date	Max Clock	Transistors	External Size	Address Space	Caches
P III	1999	500M	8.2M	64Bit	64GB	L1: 32K L2: 512K
P III	1999	700M	28M	64Bit	64GB	L1: 32K L2: 256K

The Pentium III introduced a large extension to the multimedia unit of the architecture. It was extended to 128–Bits, and could either hold twice as many subtypes or twice as much data as before. Nevertheless, new instructions were introduced that were more general than the ones from the old MMX unit. Originally, Intel introduced the unit targeted at the multimedia software, the new instructions are also used for games and mathematical computations. Therefore it was important to have instructions for single–precision packed floating point numbers. The new unit was called *Streaming SIMD Extension*, where *SIMD* stands for *Single Instruction, Multiple Data*. The SSE extension was also optimized and extended in later days.

A.6.6 Pentium III Xeon

In contrast to the mainstream processors and like the Pentium II Xeon, the server processor had a full–speed cache. The normal version had it running only at half speed. This was only important for high performance computing.

A.7 Pentium IV (2000-2006)

Name	Date	Max Clock	Transistors	External Size	Address Space	Caches
P IV	2000	1.5G	42M	64Bit	64GB	12K OP L1: 8K L2: 256K
P IV	2004	3.4G	125M	64Bit	64GB	12K OP L1: 16K L2: 1M

Pentium IV is also the invention of the so called "netburst" architecture. The processor was build for a very long time and introduced a number of extensions to the architecture. One of the most important extensions was hyperthreading, which means that some pipelines can be accessed like a separate processor. This enables a dual processor system with only one physical processor. Note that this does only speed–up overall performance if multi threading is supported by the software, because less pipelines are able to work for a single thread.

The most important extension (for our purpose) introduced by Pentium IV was the 64–Bit extension and the introduction of hardware virtualization functionality. AMD was faster than Intel with both extensions, and Intel had to react on their processors. Note that neither 64–Bit nor virtualization were really used when those processors appeared.

The 64–Bit extension was made in a similar fashion to the 32–Bit extension: all registers got an R instead of the E, for example RAX. The 32–Bit version of the registers still exist in 64–Bit mode and can be accessed by the old name, EAX in this example. Note that in 64–Bit execution mode, the standard width for registers is 32–Bit. To have 64–Bit access, an additional prefix is needed. According to the Intel manual, virtualization is only available in 64–Bit mode.