Formal Specification of the x86 Instruction Set Architecture

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Ulan Degenbaev

Saarbrücken, Februar 2012

Tag des Kolloquiums:	6. Februar 2012
Dekan:	Prof. Dr. Holger Hermanns
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Sebastian Hack
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Dr. habil. Peter-Michael Seidel
Akademischer Mitarbeiter:	Dr. Art Tevs

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, im Februar 2012

Abstract

In this thesis we formally specify the x86 instruction set architecture (ISA) by developing an abstract machine that models the behaviour of a modern computer with multiple x86 processors. Our model enables reasoning about low-level system software by providing formal interpretation of thousand pages of the processor vendor documentation written in informal prose.

We show how to reduce the problem of ISA formalization to two simpler problems: memory model specification and instruction semantics specification. We solve the former problem by extending the classical Total Store Ordering memory model with caches, translation-lookaside buffers, memory fences, locks, and other features of the x86 processor.

In order to make instruction semantics specification readable and compact, we design a new domain-specific language. The language has intuitive syntax for defining registers and instructions, so that any programmer should be able to understand the specification. Although our language is external and not embedded into a formal proof system, the language is based on the same principles as embedded, monadic domain-specific languages. Thus, it is possible to translate specifications from our language to formal proof systems.

Zusammenfassung

In dieser Arbeit spezifizieren wir den x86-Befehlssatz durch die Definition einer abstrakten Maschine, die das Verhalten eines modernen Computers mit mehreren x86-Prozessoren modeliert. Unser Modell bietet eine formale Interpretation der Prozessorherstellerdokumentationen, die über Tausend Seiten von informellen Spezifikationen enthalten.

Wir zeigen, wie das Problem der Befehlssatz-Formalisierung in zwei einfachere Probleme zerlegt werden kann: Spezifikation von dem Speichermodell und Spezifikation von der Maschinenbefehlsemantik. Wir lösen das erste Problem durch die Erweiterung des klassischen “Total Store Ordering” Speichermodells mit Caches, Translation-Lookaside Buffers, Memory Fences und Locks.

Um die Maschinenbefehlsemantikspezifikation lesbar und kompakt zu machen, entwerfen wir ein neue domänenspezifische Sprache. Die Sprache hat intuitive Syntax zur Definition von Registern und Maschinenbefehlen, so dass jeder Programmierer in der Lage sein sollte, die Spezifikation zu verstehen. Obwohl unsere Sprache nicht in ein formales Beweissystem eingebettet ist, basiert sie auf den gleichen Grundsätzen wie eingebette monadische domänenspezifische Sprachen. So ist es möglich, Spezifikationen aus unserer Sprache in ein formales Beweissystem zu übertragen.

Acknowledgments

I would like to thank my supervisor Professor Wolfgang Paul for the guidance and encouragement.

I owe my graditute to many people in the Verisoft XT group for valuable suggestions, stimulating discussions, and friendly atmosphere. Special thanks to Christian Müller, Ernie Cohen, Eyad Alkassar, Mark A. Hillebrant, and Norbert Schirmer.

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Motivation

In the first Quarter of 2011, the x86 processors comprised about 99.9% of the personal computer market and 66.4% of the server market [Cor11a, Cor11b]. Even though these processors are ubiquitous, there is still no publicly available, rigorous description of how they execute instructions. Official vendor documentation is written in informal prose, that is often ambiguous or inconsistent. A programmer who wishes to write low-level software has to spend a vast amount of time interpreting the huge vendor documents, experimenting with the real hardware, and collecting bits of the low-level programming folklore. After finishing the software, the programmer is left with the only option to ensure software correctness: testing.

Two recent technological advances have highlighted the need for formal specification of the processor behaviour. The first is proliferation of multiprocessor computers. It is difficult to write correct concurrent programs when the programmer does not know how exactly the memory accesses from one processor are observed by another processor. Testing cannot catch subtle concurrency bugs which are triggered by a specific interleaving of memory accesses. This interleaving might occur once in every million executions of the program because the highly-optimized processors reorder memory accesses in practically unpredictable way. This means that the programmer has to prove the correctness of a concurrent program for all possible interleavings. Such proofs cannot exists without precise description of how a processor issues and reorders the memory accesses.

The second advance is virtualization. Cloud and web hosting providers have embraced this technology because it allows to run several independent operating systems on a single server. Each operating system has an illusion that it fully controls the server, but in reality there is a so-called hypervisor program running on the server and providing a virtual hardware environment for an operating system. Using the processor's virtualization capabilities, the hypervisor can choose which instructions of the operating system are to be executed natively by the processor and which instructions are to be emulated by the hypervisor. Having formal specification of the instructions would help to prove

that the hypervisor emulates them correctly. Even more important concern is security of the hypervisor. Since an operating system is free to run any code, including malicious or invalid code, the hypervisor has to ensure that such code can never escape the virtual environment. Malicious code might try to exploit obscure, poorly documented effects of an instruction. The hypervisor has to be programmed to handle such effects. Only with rigorous specification of the instructions, one can hope to prove the absence of loopholes in the hypervisor.

We could list a dozen of other reasons for why formal specification of the x86 instructions is a good thing. However, the two arguments above were our main motivation when we started formalizing the instructions as part of the Verisoft XT project on Microsoft® Hyper-V™ hypervisor verification in 2007. This thesis summarizes our efforts and answers the following question: "Is is possible to develop a useful formal specification of the modern x86 instruction set architecture?"

1.2 The Problem

Instruction set architecture (ISA) is an interface between a processor and software. In other words, an ISA is a high-level processor description that provides enough information for a programmer to write and reason about low-level software. Ideally an ISA would hide as much of processor implementation as possible. Thus, specifying an ISA is a fine art of balancing between being too loose and too strict. A too loose specification makes it impossible for a programmer to prove certain properties of software. A too strict specification precludes performance optimizations in the processor hardware. An ISA defines

- the processor registers and the meaning of each bit in a register;
- the memory model, i.e. how memory accesses are reordered;
- interrupts and interrupt handling;
- the data structures shared between the processor and software, such as descriptor tables, page tables, control blocks, etc.
- the instruction opcodes and operands;
- the instruction semantics, i.e. the effects of instruction execution.

Processor vendors prefer to specify the x86 architecture in informal prose as it requires considerably less effort than formal specification and it allows them to be vague in certain places in order to leave a room for future hardware optimizations [Int09,Adv07]. Our goal is to develop a formalism that would allow us to express the behaviour of modern x86 processor in readable, precise, and sound way.

1.3 Related Work

Extensive research has been done on memory models since Lamport first defined a sequentially consistent memory model in 1978 [Lam79]. A system of multiple processors

and of a shared memory is sequentially consistent if: *"the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."* Higham et. al. formalize this statement in two different ways: axiomatic and operational [HKV98]. For the former, they consider a collection P of processors, a collection J of memory cells, and a collection O of actions. An action is either a read action or a write action. For each action they know the processor performing it, the destination memory cell, and the data of the action's memory operation (data to be written or to be read). With such a setup, the memory consistency models can be defined in terms of partial orders satisfying specific constraints. For example, a computation with actions O is sequentially consistent if there exists a linearization $(O, <_L)$ such that $(O, \xrightarrow{\text{prog}}) \subseteq (O, <_L)$, where $\xrightarrow{\text{prog}}$ is a partial order over O that defines the program order, $<_L$ is a total order over O such that the data of each read action from cell x matches the data of the most recent (in terms of $<_L$) write to cell x or the initial value of cell x if there are no previous writes.

The operational way is to define an abstract machine as a nondeterministic transition system. Higham et. al. describe the following abstract machine that implements exactly a sequential consistent memory model:

- the machine has n processors $p_1, p_2 \dots p_n$ and a shared memory.
- each processor p_i is connected to the memory by two FIFO channels:
  - the $request_i$ channel is directed from the processor to the memory,
  - the $reply_i$ channel is directed from the memory to the processor.
- when processor p_i needs to read from the memory, it puts the corresponding read request in to the $request_i$ channel.
- when processor p_i needs to write to the memory, it puts the corresponding write request in to the $request_i$ channel.
- the memory nondeterministically chooses a nonempty request channel $request_i$ and serves the incoming request:
  - if the request is a read request, the memory puts the value of the requested memory cell into the $reply_i$ channel.
  - if the request is a write request, the memory updates the value of the requested memory cell.
- if the $reply_i$ channel is not empty, processor p_i reads the result from the channel.

Since we can order all memory accesses by the time they are served by the memory and this order agrees with the program order, the abstract machine is sequentially consistent. The reverse is also true, the machine admits all possible sequentially consistent executions.

Ideally, the memory model of an ISA would be defined using both axiomatic and operational approaches. The axiomatic specification is easier to work with in formal proofs, while the operational specification is more intuitive and is easier to understand. Real-world architectures, however, are not sequentially consistent as it would prevent certain performance optimizations like write buffers. There are many relaxed memory

models, in which the processors do not necessarily agree on the global order of the memory accesses [DSB98, AG96, HKV98, ANB+95].

We are not aware of any publicly available memory model that captures the effects of the complete memory system of a modern x86 processor, including write-combining buffers and caches. Vendor documents are particularly obscure in this aspect. They list rules that allow or forbid certain reorderings of memory accesses. Sarkar et. al. tried to formalize the rules for cacheable memory accesses that have the write-back caching policy [SSN+09]. They developed the x86-CC – a relaxed memory model with causal consistency. Later they discovered that the model was too strict, i.e. the model excluded some executions that may appear in real processors. Based on the experiments, they came up with a new, much simpler memory model, called x86-TSO [OSS09]. The model is similar to the Total Store Ordering (TSO) model of the SPARC architecture [SPA92]. Sarkar et. al. formalized the x86-TSO in HOL4 both operationally and axiomatically. In the operational model, they define the following abstract machine:

- the abstract machine has of multiple processors, a shared memory, and a global lock;
- a processor has registers, represented as a function from a register name to a value;
- a processor has a FIFO write buffer, represented as a list of address, value pairs;
- the shared memory is represented as a function from addresses to values;
- the global lock is either empty or contains the processor id, which is said to hold the lock;
- when a processor needs to write to the memory, it puts the address, value pair in to write buffer;
- when a processor needs to read from the memory, it checks whether the write buffer already has the required address;
  - if the address is in the buffer, the processor reads the corresponding value;
  - otherwise, the processor reads the value from the shared memory if it holds the lock or the lock is empty;
- a write operation at the front of a write buffer is applied to the shared memory if the corresponding processor holds the lock or the lock is empty;
- a processor reads and writes registers;
- when a processor needs to execute a memory fence instruction, it waits until its write buffer is empty;
- a processor acquires and releases the lock.

In the axiomatic model, they consider a set of events, where each event is an action (read, write, memory fence) augmented with the processor id and the instruction id. They define a valid execution in terms of partial orders on the event set and prove that the axiomatic model is equivalent to the operational model. The authors report that

the x86-TSO was validated using their own tool. The tool takes program fragments in an assembly-like syntax and runs the fragments multiple times in different threads checking that the outcome of each run agrees with the memory model.

Having reviewed the work on memory models, we proceed to discuss the work on instruction semantics specification. Groups in academia have given formal specifications for ARM, DLX, SPARC architectures [FF01, MP00, PPS+95]. There are two reasons why it is difficult to apply these methodologies to the x86 instruction specification. First, the x86 architecture is a complex instruction set computer (CISC) architecture, while the other architectures mentioned above are reduced instruction set computer (RISC) architectures. As the names imply, a RISC architecture has much simpler and more uniform instructions and registers than a CISC architecture. Even if one uses simple mathematical notation, the RISC instruction definitions tend to be compact and readable. However, such notation does not work for some huge x86 instructions that make dozens of different memory accesses. The definitions quickly become incomprehensible and error-prone.

The second reason is that there is a fundamental difference between instruction specification for a single-processor machine and for a multiprocessor machine. In the former case, the processor owns the memory[1], so the memory cannot change while the processor is executing an instruction. Thus, we can specify instruction semantics by defining two functions:

$$\begin{aligned} \mathit{fetch\text{-}and\text{-}decode} &\in \mathit{MachineState} \to (\mathit{MachineState}, \mathit{Instruction}), \\ \mathit{execute} &\in (\mathit{MachineState}, \mathit{Instruction}) \to \mathit{MachineState}. \end{aligned}$$

The first function takes the machine state (the processor registers and the memory) as an argument, and returns the new machine state together with an abstract representation of the decoded instruction. The second function executes the given instruction and returns the new machine state. Since the two function are ordinary mathematical functions, they can easily be expressed in any formal language.

In a multiprocessor machine, we cannot execute an instruction in a single step, because this would not interleave memory accesses of one processor with the accesses of another, and thus would make instructions atomic. This means that we have to specify each memory access of an instruction explicitly and then plug the accesses into a suitable memory model. Now we cannot just define a single $\mathit{execute}$ function as was the case for a single-processor machine. Instead, we have to turn the definition of the sequential $\mathit{execute}$ inside-out, revealing each place where the function reads the memory and replacing this place with a memory access request. Thus, the new $\mathit{execute}$ function does not have direct access to the memory, and it issues memory access requests:

$$\mathit{execute} \in (\mathit{Registers}, \mathit{Instruction}, \mathit{set}(\mathit{Reply})) \to (\mathit{Registers}, \mathit{set}(\mathit{Request})).$$

The function takes the current state of the registers, the current instruction, and a set of memory replies to the previous requests. The functions returns the new state of the registers and a set of new memory requests. Given such a function, it would be easy to plug the function into any operational memory model: apply the function, forward the requests to the memory, collect the replies, and then apply the function again, etc. The problem is that it is very difficult to explicitly define such a fine-grained function for

[1] assuming there are no devices

any non-trivial ISA. The functional programming community discovered that monads are good for solving this kind of problems [Wad92]. Monads allow one to represents computations as data. In monadic style programming, one first introduces primitive computations such as accessing the memory, writing a register, etc. Then one builds up more complex computations by combining the primitive computations with special combinator functions. Thus, instruction semantics can be represented as blocks of primitive computations glued together with combinator functions. By carefully defining the combinator functions and the primitive computations, one can get the required *execute* function, that can be plugged into the memory model.

Sarkar et. al. used monadic style to specify about 20 general-purpose instructions from the x86 architecture [SSN+09]. The primitive computations are: reading and writing a register (read_reg, write_reg), reading and writing 32-bit aligned memory (read_m32, write_m32), reading and writing the instruction pointer (read_eip, write_eip), reading and writing the flags register (read_eflags, write_eflags). Two computations can be combined either sequentially using the seqT combinator, or in parallel using the parT combinator. As an example, we list the definition of the POP instruction by Sarkar et. al. The POP instruction reads four bytes from the top of the stack, which is pointed to by the ESP register, and increments the ESP by four.

```
val x86_exec_pop_def = Define ‘
  x86_exec_pop ii rm =
    seqT (seqT (read_reg ii ESP) (\esp. addT esp (write_reg ii ESP (esp + 4w))))
         (\(old_esp,x). seqT (parT (ea_Xrm ii rm) (read_m32 ii old_esp))
                             (\(ea,w). write_ea ii ea w))‘;
```

Fox and Myreen used the same approach to fully specify the ARMv7 architecture [FM10]. The size of the model is about 6500 lines of HOL4 code. The authors validated the model by running thousands of tests.

Hunt is developing another x86 ISA specification using a domain-specific language embedded into the ACL2 theorem-prover [WAH10]. This specification is used for verification of processor components. Unfortunately the specification is not publicly available.

In 1990s several groups made effort on x86 formalization. Ramsey and Fernandez developed a toolkit for instruction format specification [RF95, RF97]. Using the toolkit they formally specified the instruction format of the Pentium processor. The toolkit employs a syntactic approach to define the instruction format and cannot specify semantics of complex instructions. Papers [RM99, HHD97] present general frameworks for ISA specification, which were successfully used to define simple instruction sets (MIPS, PowerPC). The frameworks are claimed to be powerful enough to specify x86 ISA, however, such specification is not available yet. Moreover, the frameworks are targeted at user level ISA and do not model such features as memory management, interrupts and exceptions, multiple processors.

Although not quite formal, the source code of x86 emulators can be very helpful for resolving certain ambiguities in the processor vendor documentation. Virtual Box, Bochs, and QEMU are open source emulators [Vir, Boc, QEM].

Our model is built upon previous work with Wolfgang Paul, Peter-Michael Seidel, Norbert Schirmer, and Ernie Cohen [DPS09, Deg07]. Christoph Baumann’s master thesis [Bau08] complements our work by modeling floating-point instructions.

1.4 Methodology

We are going to specify the instruction set operationally. Thus, our aim is to define a transition system that models the behaviour of an x86 multiprocessor machine. This problem can be divided into two smaller problems:

- define a memory model assuming abstract instruction semantics. In other words, we specify how memory accesses of one processor are reordered and observed by other processors without knowing how exactly the processor issues the accesses.
- define instruction semantics assuming an abstract memory model. This means that we specify how a single processor executes an instruction assuming that there is a memory system that can answer memory requests.

If we define a good interface between the two problems, then the problems can be attacked independently. Consider information flow between a processor and the memory system, when the processor is executing an instruction after fetching the instruction:

1. The processor makes computation based on the registers and the fetched instruction.
2. If the processor needs data from the memory, the processor issues one or more memory read requests.
3. The memory serves the requests and sends the replies back to the processor.
4. The processor makes computation based on the registers, the instruction, and the replies to the previous requests.
5. If the processor needs data from the memory, the processor issues one or more memory read requests.
6. The memory serves the requests and sends the replies back to the processor.
7. The previous three steps repeat until the processor needs no more data from the memory.
8. Once the processor has all the necessary data for the instruction, the processor actually executes the instruction, i.e. the processor computes the data to be written to the registers and to the memory.
9. The processor updates the registers, issues zero or more memory write requests, and completes the instruction.

In the previous section, we discussed a general execute function that models the processor computations on steps 1, 2, 4, 5, 8, 9:

$$execute \in (Registers, Instruction, set(Reply)) \to (Registers, set(Request)).$$

Note that this function is an interface between the processor and the memory system. The memory model can use this function as a black box without knowing how exactly it is defined. Thus, we already have the required interface, but we are going to adjust it in order to make it more suitable for the x86 architecture.

First we separate read requests from write requests. Thus, instead of a single $Request$ type, we have a $ReadReq$ type and a $WriteReq$ type. Now we can factor out steps 2, 5 into a separate function:

$$data\text{-}req \in (Registers, Instruction, set(Reply)) \to set(ReadReq).$$

We change the meaning of the execute function to denote only step 9:

$$execute \in (Registers, Instruction, set(Reply)) \to (Registers, set(WriteReq)).$$

In the x86 architecture, an instruction may fail and trigger an exception. For example, the division instruction generates an exception if the divisor is zero. Therefore, we need a function that checks whether the current computation fails or not, and returns information about the exception in case of failure.

$$fault \in (Registers, Instruction, set(Reply)) \to (Exception \cup \epsilon).$$

An instruction can also be intercepted by the hypervisor if it being executed in the virtual machine. Such intercept generates an exit from the virtual machine. We add a function that checks for an intercept and returns information about the intercept if it has occured:

$$vmexit \in (Registers, Instruction, set(Reply)) \to (Intercept \cup \epsilon).$$

If paging address translation is enabled, then the processor may request the translation-lookaside buffer (TLB) to translate a virtual address to a physical address. In our model, the TLB is a unit outside the processor that caches translations and walks page tables. We need to add another type of request and a function that computes the translation requests:

$$trans\text{-}req \in (Registers, Instruction, set(Reply)) \to set(TLBReq).$$

Assume that the $Reply$ type contains both replies from the memory and replies from the TLB. Besides the TLB, there are other external units such as caches and buffers. Some instructions change the state of the external units. We introduce a new kind of requests: commands to other units, and we change the execute function to return a set of commands:

$$execute \in (Registers, Instruction, set(Reply)) \to (Registers, set(WriteReq), set(Cmd)).$$

Now the $data\text{-}req, trans\text{-}req, fault, vmexit, execute$ functions completely cover all possible events that can occur during instruction execution and return all necessary information about the execution. This means that they are an interface between a processor and the rest of the machine[2]. Thus, we can use these functions to define transitions of the abstract machine. Afterwards, we will be able to plug in an implementation of the functions and obtain the complete ISA model.

Having fixed the interface, we can proceed to solving the two problems separately.

[2] In the next chapter we will extend the interface to cover other phases of the instruction processing cycle, such as fetch/decode, jump to an interrupt service routine, etc.

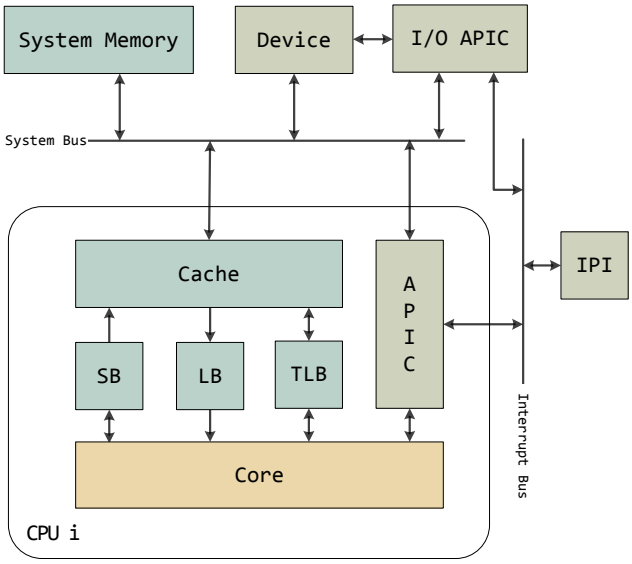

Figure 1.1: Abstract machine

1.4.1 Memory Model

The memory system consists of a shared physical memory, caches, store buffers, load buffers, and translation lookaside buffers. Figure 1.1 shows the memory system units together with other parts of the abstract machine. In the subsequent chapters we define transitions of the abstract machine. Each transition has a guard condition. Any transition with the satisfied guard condition can trigger nondeterministically. A triggered transition may change the state of one or more units of the abstract machine.

There is no need to justify why the abstract machine has the physical memory. However, other units of the memory system deserve a few words:

- a store buffer: it is a queue of stores. It delays stores on their way from the processor core to the caches/environment. Due to this delay, remote processors observe reordering of new instructions and loads ahead of old stores.
- a load buffer: we need it to model effects of out-of-order/speculative instruction execution. It nondeterministically prefetches instructions and data with the write-combining (WC) memory type. WC loads and instruction fetches can be reordered ahead of old instructions and loads. By prefetching, the load buffer introduces a negative delay, so that remote processors observe the required reordering.
- a translation-lookaside buffer: it traverses page tables and collects translations.
- a cache: it models data and code caches of real processors and implements the MOESI cache-coherence protocol. Effects of the caches become visible to software, when software accesses the same address with cachable memory type and uncacheable memory type.

Our memory model extends the total store ordering memory model with more relaxed instruction fetches and write-combining load/stores.

1.4.2 Instruction Semantics

For instruction specification we designed a new domain-specific language. The language has primitive operations such as read from memory, write to memory, get a translation from the TLB, write a register, stop execution with a failure. Using these primitives and standard language constucts (expressions, conditionals, functions) we can define more and more complex operations. Instruction semantics are then expressed in terms of these complex operations.

Here is an example of the stack pop operation that uses the *"read from logical address space"* operation:

```
action pop($n::Width, rsp::bits $sa)::(bits $n, bits $sa)
  call value = lread(stack, $n, SS, $sa, rsp)
  return (value, rsp + bits($sa, $n))
```

Based on the pop operation, we can specify the POP instruction:

```
opcode "8Fh␣/000b" reg_mem $v : call op1' = POP($v)
opcode "58h" reg $v           : call op1' = POP($v)
opcode "17h" ss 16            : call op1' = POP(16)
                              : when not $x64_mode
opcode "1Fh" ds 16            : call op1' = POP(16)
                              : when not $x64_mode
opcode "07h" es 16            : call op1' = POP(16)
                              : when not $x64_mode
opcode "0FA1h" fs 16          : call op1' = POP(16)
opcode "0FA9h" gs 16          : call op1' = POP(16)

action POP($n::{$v, 16})::bits $n
  call (value, new_rsp) = pop($v, RSP[$sa-1:0])
  write gpr($sa, new_rsp, RSP) to RSP
  return value[$n-1:0]
```

Notice that we specify both the semantics of the instruction and the information for decoding the opcode. From such specification it is possible to generate the definitions of the *data-req*, *trans-req*, *fault*, *vmexit*, *execute* functions by analysing the abstract syntax tree of the specification.

1.5 Scope of the model

Our model specifies 140 general-purpose and system programming instructions. As we had limited time, we omitted the following features:

- floating-point and multimedia instructions,
- debug facilities and alignment check exception,
- virtual-8086 mode and virtual interrupts,
- hardware task-switching,
- system management mode.

1.6 Outline

The thesis consists of two parts: Chapters 2-11 and Chapters 12-32. The first part describes the overall transition system of interacting units: cores, caches, load/store buffers, translation-lookaside buffers, and interrupt controllers. This part uses abstract functions that specify memory loads, memory stores, and register updates performed during instruction fetch, decode, and execution in a single core. These functions are later obtained from the the second part of the thesis, which defines a specification language and specifies instruction fetch, decode, execution in that language for a single core.

Chapter 2 introduces notation that we use to define the configuration and the relations of the transition system.

Chapter 5 gives high-level description of the transition system and defines the interface between the first part and the second part of the thesis.

Chapters 6-8 define configuration and transitions of the memory system, including caches, load and store buffers, and transition-lookaside buffers.

Chapter 9 describes the high-level configuration and transitions of a processor core. Because of interrupt handling and virtual machine intercepts, the instruction processing cycle of a core is quite sophisticated. This chapter defines several phases and shows how the core transitions from one phase to another as it processes an instruction. Many details are abstracted away and are filled in later in the second part of the thesis.

Chapter 10 describes the local advanced programmable interrupt controller (APIC), which accepts interrupts from devices and other APICs and forwards them to the processor core.

Chapter 11 introduces notation for the second part of the thesis. This notation is a domain specific language (DSL) for register and instruction specification.

Chapter 12 uses the DSL to specify the register of a processor core.

Chapter 13 describes architectural features of a processor core, such as operating modes, exceptions, segmentation, protection, etc.

Chapter 14 defines general instruction format and instruction fetch/decode in the DSL.

Chapters 15-18 and Appendix specify format and execution of concrete instructions.

Part I
ABSTRACT MACHINE

CHAPTER

TWO

NOTATION

A bitvector x of length n is denoted as $x \in \mathbb{B}^n$. We overload standard operators $+$, $-$, $*$, $/$ to allow arithmetic operations modulo 2^n on bitvectors $\mathbb{B}^n$. The i-th bit of x is denoted as $x[i]$, and a range of bits from the i-th bit to the j-th bit is $x[j:i]$. Thus, the second byte of x is $x[15:8]$. We denote concatenation of two bitvectors x and y as $x \circ y$. Concatenation of n copies of x is $x^k = \underbrace{x \circ \ldots \circ x}_{k \text{ times}}$.

We use functions $zxt_m(x)$ and $sxt_m(x)$ for zero- and sign-extention of x to m bits:

$$zxt_m(x) = 0^{m-n} \circ x,$$
$$sxt_m(x) = x[n-1]^{m-n} \circ x.$$

When it does not introduce ambiguity, we allow implicit zero-extension. For example, $x + t$ should be read as $x + zxt_n(t)$ if the width of t is smaller than n. Likewise, we overload numeric literals to denote bitvectors of different widths, for example, $8 \in \mathbb{B}^8$, $8 \in \mathbb{B}^{64}$.

A map m with a domain $\mathbb{B}^{64}$ and a range $\mathbb{B}^8$ is denoted as $m \in \mathbb{B}^{64} \to \mathbb{B}^8$. We can define a new map either using a lambda expression or by updating another map:

$$m = \lambda x \in \mathbb{B}^{64} : 0,$$
$$r \ = m \ \textbf{with} \ [1 \mapsto 8].$$

Using the **if** - **then** - **else** operator we can also define r as

$$r = \lambda x \in \mathbb{B}^{64} : \textbf{if} \ x = 1 \ \textbf{then} \ 8 \ \textbf{else} \ m[x].$$

We make a distinction between ordinary functions and functions that model memories and arrays, which we call maps. Ordinary functions can have maps as parameters. To emphasize the role of the maps, we write a map application as m[x] instead of more conventional $m(x)$.

One can think of a record as a tuple with named components. To define a record type we list all its components:

$$R \triangleq [a \in \mathbb{B}^{64}, b \in \mathbb{B}^8].$$

The symbol $\triangleq$ means 'define'. Thus, if $x \in R$, then $x.a \in \mathbb{B}^{64}$ and $x.b \in \mathbb{B}^8$. There are two operations on records: create and update. An example of the former is $x = R$ **with** $[a \mapsto 1, b \mapsto 9]$, and of the latter is $y = x$ **with** $[a \mapsto 2]$, which defines a record $y \in R$ such that $y.a = 2$ and $y.b = x.b$. When we need to update a component q of a nested record $t.p.r$, we use shortcut t **with** $[p.r.q \mapsto s]$ instead of t **with** $[p \mapsto t.p$ **with** $[r \mapsto t.p.r$ **with** $[q \mapsto s]]]$

A list l of n elements of type T is written as $l \in T^n$. We assume standard operations on lists: construction $l = x :: xs$, concatenation $l = a \circ b$, indexing the i-th element $l[i]$, indexing a range $l = l[0 \ldots n-1]$, explicit enumeration $l = [l[0], l[1], \ldots, l[n-1]]$.

2.1 Relations

The goal of this document is to define the transition relation of an abstract x86 machine. As this relation is quite complex, we decompose it into a union of smaller relations each of which represents a transition of a subset of the components of the abstract machine. We specify each transition in a box with three parts: 'label', 'guard', and 'effect'. The first part defines the label of the transition. The second part contains a set of conditions under which the transition can occur. Finally, the third part gives a set of equations between the current configuration and the next configuration.

The label can have a number of parameters. In this case, the box specifies a family of transitions, one for each possible value of the parameters. Each box has its own syntactic scope, where only the following names are visible:

- names of the label parameters;
- names of the abstract machine components (current configuration);
- primed names of the abstract machine components (next configuration);
- names of the functions;
- names of local variables introduced in the box.

As an example, consider a transition of the cache component of processor i:

label	$drop\text{-}line(i \in pid, pa \in \mathbb{B}^{pq})$
guard	$cache[i].state[pa] \in \{E, S\}$
effect	$cache'[i].state[pa] = I$

This transition specifies that a cache line in a clean state can be invalidated at any time. Note that all parts of the next configuration that are not mentioned in the 'effect' part are assumed to be unchanged. Let c, c' be configurations of the abstract machine and Δ be the transition relation, then the above box can be translated into the following expression:

$$(\exists i \in pid : \exists pa \in \mathbb{B}^{pq} : c.cache[i].state[pa] \in \{E, S\} \wedge \\ c' = c \textbf{ with } [cache[i].state[pa] \mapsto I]) \\ \Rightarrow (c, c') \in \Delta.$$

2.2 Functions

For each function we will give its signature: the function name, names and types of the parameters, and the type of the function result. We define a function by writing

its body as an expression. The expression is a mathematical expression extended with **let**−**in**, **choose**−**in** and **if**−**then**−**else** constructs. As an example, consider a function that calculates the maximum of three numbers:

$$max3(a \in \mathbb{N}, b \in \mathbb{N}, c \in N) \in \mathbb{N} \triangleq$$
$$\quad \textbf{let } t = (\ \textbf{if } a > b \textbf{ then } a \textbf{ else } b) \textbf{ in}$$
$$\quad\quad \textbf{if } t > c \textbf{ then } t \textbf{ else } c.$$

A function that calculates the remainder after division can be defined using the choice operator **choose**−**in**:

$$mod(a \in \mathbb{N}, b \in \mathbb{N}) \in \mathbb{N} \triangleq \textbf{choose } q, r \in N : a = b * q + r \wedge 0 \leq r < b \textbf{ in } r.$$

Although we do not use recursion, functions are not necessarily total. It may happen that there is no sensible result for the given values of parameters. In such cases we define the function domain. A predicate with the name $can\text{-}f$ defines the domain of a function with name f. Functions that do not have the corresponding $can\text{-}f$ predicate are total.

For each memory component we will define $read$ and $write$ functions. For brevity, we allow name overloading for these functions. Thus, instead of $read\text{-}cache(cache)$ and $read\text{-}env(env)$, we will write $read(cache)$ and $read(env)$.

2.3 Conventions for memory accesses

All memory accesses are 8-byte (quadword) aligned. If the processor needs to access only some part of a quadword, then the required bytes are selected with an 8-bit mask – one bit for each byte of the quadword. Using this we can simulate unaligned accesses.

We will use the following types of physical addresses:

- $\mathbb{B}^{pb}$ – physical byte address;
- $\mathbb{B}^{pq}$ – physical quadword address, $pq = pb - 3$;
- $\mathbb{B}^{pp}$ – physical page address, $pp = pb - 12$.

The physical address width pb is not fixed by the ISA, but it is less than 64. For AMD processors it is typically 52, and for Intel it is 36.

The virtual address width depends on the paging mode of the processor. We will use $\mathbb{B}^{vb}$ to represent a virtual byte address, and assume that if the actual virtual byte address has a smaller width, then it is zero-extended to vb bits. Thus, we have:

- $\mathbb{B}^{vb}$ – virtual byte address, $vb = 48$;
- $\mathbb{B}^{vq}$ – virtual quadword address, $vq = vb - 3$;
- $\mathbb{B}^{vp}$ – virtual page address, $vp = vb - 12$.

A physical memory access has $addr \in \mathbb{B}^{pq}$, $data \in \mathbb{B}^{64}$, and $mask \in \mathbb{B}^{8}$ components. We often need to combine the results of two memory accesses to the same address. Given two store accesses to the same address, we can combine them using the following function:

$$combine(old \in (\mathbb{B}^{64}, \mathbb{B}^{8}), new \in (\mathbb{B}^{64}, \mathbb{B}^{8})) \in (\mathbb{B}^{64}, \mathbb{B}^{8}).$$

Let $(data_3, mask_3) = combine((data_1, mask_1), (data_2, mask_2))$, then

$mask_3 = mask_1 \vee_8 mask_2$ and $\forall k \in \mathbb{N}_{64}$:

$$data_3[k] = \begin{cases} data_2[k] & \text{if } mask_2[k/8], \\ data_1[k] & \text{otherwise.} \end{cases}$$

CHAPTER

THREE

MODEL OVERVIEW

In this chapter we develop the ideas from section 1.4 and present a detailed interface between a processor core and the memory system. After that, we describe the memory system units and define the abstract machine configuration.

3.1 Instruction execution

The instructions are complex: some instructions make more than 50 memory accesses. After performing a few memory accesses an instruction might raise an exception, which discards changes to the processor registers and the memory system made by the instruction. Instructions usually make large case analyses because the x86 ISA is, actually, a combination of five ISAs, each of which is enabled by the corresponding operating mode: 64-bit mode, compatibility mode, protected mode, virtual 8086 mode, and real mode. This means that even in a single processor setting we need an advanced notation/language to specify semantics of the instructions. As we discussed in section 1.4, semantics of instruction execution for a single processor can be captured with the $execute$ function:

$$execute \in (Registers, Instruction, set(Reply)) \to (Registers, set(WriteReq), set(Cmd)).$$

The function takes as input:

- values of processor registers,
- a decoded instruction,
- a set of data loaded from the memory system,
- a set of paging translations from the TLB,

and produces as output:

- new values of processor registers,
- a set of memory stores,
- commands for other components of the processor, such as:
  - invalidate the cache,

- invalidate a cache line,
- flush the TLB,
- invalidate translation in the TLB.

A processor state in our abstract machine is modelled by a $Core$ component. This component contains processor registers and information about the current instruction. In order to simplify further definitions, we embed requests, replies, and commands in $Core$ component. Thus, the $execute$ becomes a state transformer:

$$execute \in Core \to Core.$$

The $Core$ is a record, which consists of the following components:

$$\begin{aligned} Core \triangleq [\,&\text{registers: } \ldots \text{ defined in section 9.1 and chapter 12} \\ &\text{instruction info: } \ldots \text{ defined in section 12.7} \\ &\text{commands:} \\ &invd \in CacheInvd, \\ &\textit{flush-line} \in CacheLineFlush, \\ &\textit{tlb-flush} \in TLBFlush, \\ &invlpg \in TLBFlushPage, \\ &\text{buffers:} \\ &\textit{mem-in} \in ReadReq \to ReadReply, \\ &\textit{mem-out} \in WriteReq \to \mathbb{B}, \\ &\textit{tlb-in} \in TLBReq \to TLBReply, \\ &\text{auxiliary components: } \ldots \text{defined in section 9.1}], \end{aligned}$$

Each command contains a flag called $valid$, which indicates whether the command is issued or not. Besides this flag, each command has information that is specific to the command, such as cache line address, page address, etc. We give detailed description of the commands in section 9.1.

More interesting is how we model requests and replies. A memory read request is a record with the following components:

$$ReadReq \triangleq [\, addr \in \mathbb{B}^{pq}, mask \in Mask, mt \in MemType, code \in \mathbb{B}]$$

where the $addr$ is a quadword aligned physical address, the $mask$ is a byte select mask ($Mask \triangleq \mathbb{B}^8$), which selects requested bytes in the quadword, the mt is the memory type (see section 5.2), and the $code$ is a flag that indicates whether the request is a code fetch request or an ordinary read request. For each served memory read request, the $\textit{mem-in}$ buffers stores the reply:

$$ReadReply \triangleq [\, ready \in \mathbb{B}, data \in Data],$$

where the $ready$ indicates whether the request was served or not, and the $data$ contains the result of the read access if the request was served ($Data \triangleq \mathbb{B}^{64}$).

As we discussed in section 1.4, we can get the set of issued memory read requests using the $data\text{-}req$ function:

$$data\text{-}req \in Core \to set(ReadReq).$$

Since any function $f \in X \to set(Y)$ can be turned into an equivalent predicate $p \in X \to Y \to \mathbb{B}$, we will use the following predicate to check whether a read request is issued or not:

$$data\text{-}req\text{-}execute(core \in Core, pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^{64}, mt \in MemType, code \in \mathbb{B}) \in \mathbb{B}.$$

The suffix $execute$ in the name of the predicate means that the predicate is valid only when the core is executing an instruction, i.e. the core is in the *execute* phase. There are several other phases such as *decode*, *vmexit*, etc. Section 9.2 describes the phases in detail.

The $mem\text{-}out$ buffer contains a set of issued memory write requests. A write request looks like a read request but it has an additional component: the data to be written.

$$WriteReq \triangleq [\, addr \in \mathbb{B}^{pq}, mask \in Mask, mt \in MemType, data \in Data].$$

The $tlb\text{-}in$ buffer is similar to the $mem\text{-}in$ buffer, but it stores a TLB reply for each served translation request. A translation request is a pair of a virtual page-aligned address and access rights:

$$\begin{aligned} TLBReq &\triangleq [\, va \in \mathbb{B}^{vp}, r \in Rights], \\ Rights &\triangleq [\, write \in \mathbb{B}, user \in \mathbb{B}, code \in \mathbb{B}], \end{aligned}$$

where the access rights indicate the type of the memory access for which the address translantion is being requested.

A TLB reply is a record with the following fields:

$$\begin{aligned} TLBReply \triangleq [\, &ready \in \mathbb{B}, fault \in \mathbb{B}, fault\text{-}code \in \mathbb{B}^8, \\ &ba \in \mathbb{B}^{pp}, mt \in MemType], \end{aligned}$$

where the $ready$ flag indicates whether the translation request was served or not, the $fault$ flag indicates whether the translation produced a page fault or not, the $fault\text{-}code$ contains the page fault code in case of a page fault, the ba is the translated page-aligned physical address, and the mt is the memory type translated address.

For each translation request, we can check whether it was issued or not using the following predicate:

$$trans\text{-}req\text{-}execute(core \in Core, va \in \mathbb{B}^{vp}, r \in Rights) \in \mathbb{B}.$$

The $execute$ function is partial, i.e. it might fail if

- more data from the memory is needed to complete the instruction execution;
- more translations from the TLB are needed to complete the instruction execution;

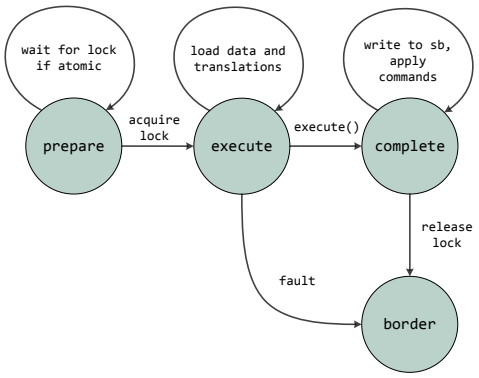

Figure 3.1: Instruction execution

- an exception occurs during the instruction execution;
- an virtual machine intercept occurs during the instruction execution;

We already described the predicates that check the former two conditions. The latter two conditions can be checked with the following predicates:

$$\textit{fault-on-execute}(\textit{core} \in \textit{Core}, e \in \textit{Exception}) \in \mathbb{B},$$
$$\textit{vmexit-on-execute}(\textit{core} \in \textit{Core}, x \in \textit{Intercept}) \in \mathbb{B},$$

where $\textit{Exception}$ and $\textit{Intercept}$ contain information describing the exception and the intercept (see section 9.1). Thus, $\textit{fault-on-execute}(\textit{core}, e)$ holds if and only if executing the current instruction raises the exception e, which is not intercepted. Likewise, $\textit{vmexit-on-execute}(\textit{core}, x)$ holds if and only if executing the current instruction triggers the intercept x. Based on these predicates, we can define the domain of the $\textit{execute}$ function:

$$\begin{aligned}&\textit{can-execute}(\textit{core} \in \textit{Core}) \in \mathbb{B} \triangleq \\ &\quad (\forall \textit{pa}, \textit{mask}, \textit{mt}, \textit{code} : \neg \textit{data-req-execute}(\textit{core}, \textit{pa}, \textit{mask}, \textit{mt}, \textit{code})) \\ &\quad \wedge (\forall \textit{va}, r : \neg \textit{trans-req-execute}(\textit{core}, \textit{va}, r)) \\ &\quad \wedge (\forall e : \neg \textit{fault-on-execute}(\textit{core}, e)) \\ &\quad \wedge (\forall x : \neg \textit{vmexit-on-execute}(\textit{core}, x)),\end{aligned}$$

so, $\textit{can-execute}(\textit{core})$ holds if and only if all necessary data are fetched into the buffers and the current instruction can be successfully executed.

To show how the predicates and the $\textit{execute}$ function fit together, we describe a simplified instruction execution scheme. Figure 3.1 illustrates the phases of instruction execution and transitions between the phases. Assuming that the instruction was fetched and decoded, the processor core executes it by making the following transitions:

1. Acquire the memory lock if the instruction must be executed atomically. When the processor acquires the memory lock, then other processors cannot access caches, physical memory, and memory-mapped devices.
2. (a) Collect translations from the TLB, using the $\textit{trans-req-execute}$ predicate to detect the requested translations.
   (b) Collect data from the memory, using the $\textit{data-req-execute}$ predicate to detect the requested data.

3. Check, using the *fault-on-execute* and *vmexit-on-execute* predicates, whether the instruction raises an exception or an intercept, and, in case of an exception or an intercept, record the exception information and abort the instruction execution.
4. Invoke the *execute* function (if the instruction can be executed successfully), update the registers, and enable:
   (a) transitions that copy the stores into the store buffer,
   (b) transitions that apply commands to other components.
   Note that applying commands to other components never fails (i.e. never raises an exception).
5. Release the memory lock if the lock was acquired.

Thus, the memory accesses of an atomic instruction are not interleaved with other memory accesses. In case of a non-atomic instruction, fine-grained transitions of type 2.a and transitions of the store buffer allow other processors to make steps between two consecutive accesses of the instruction.

We have defined an interface that allows us to abstract instruction execution. Besides executing instructions, the processor fetches/decodes instructions, jumps to interrupt service routines, and switches from guest mode to host mode. All these activities are similar to instruction execution: they load data and translations, update registers, produce stores, and even can raise exceptions or be intercepted. Therefore, we call them pseudo-instructions and specify them via the *decode*, *jisr*, and *vmexit* functions, which are analogs of the *execute* function. Let *xxxx* denote one of these functions, then we derive the following predicates from the definition of the *xxxx*:

$$
\begin{aligned}
&\textit{data-req-xxxx}(\textit{core} \in \textit{Core}, \textit{pa} \in \mathbb{B}^{pq}, \textit{mask} \in \mathbb{B}^{64}, \textit{mt} \in \textit{MemType}, \textit{code} \in \mathbb{B}) \in \mathbb{B},\\
&\textit{trans-req-xxxx}(\textit{core} \in \textit{Core}, \textit{va} \in \mathbb{B}^{vp}, r \in \textit{Rights}) \in \mathbb{B},\\
&\textit{fault-on-xxxx}(\textit{core} \in \textit{Core}, e \in \textit{Exception}) \in \mathbb{B},\\
&\textit{vmexit-on-xxxx}(\textit{core} \in \textit{Core}, x \in \textit{Intercept}) \in \mathbb{B},\\
&\textit{can-xxxx}(\textit{core} \in \textit{Core}) \in \mathbb{B}.
\end{aligned}
$$

Summarizing this section, we divide the original problem of x86 formalization into the following subproblems:

1. using a domain specific language (DSL), define instruction fetch/decode, execution, jump to interrupt service routine, switch to host mode for a single processor machine.
2. from the definitions in DSL derive formal definitions for *decode*, *execute*, *jisr*, *vmexit* and the corresponding predicates.
3. using functions from 2, define an abstract x86 machine as a transition system.

The first problem is solved partially in [Deg07], which specifies instructions in a functional language. The functional style, however, impairs readability because of the gap between formal specification and informal specification from the official manuals. Our DSL expresses specifications in an imperative style. This minimizes the gap and increases confidence in correctness of the specification. On the other hand, the DSL is simple enough (not turing-complete), which allows us to define simple formal semantics for the language and derive formal definitions for the functions listed in the second problem.

In the subsequent sections we solve the third problem.

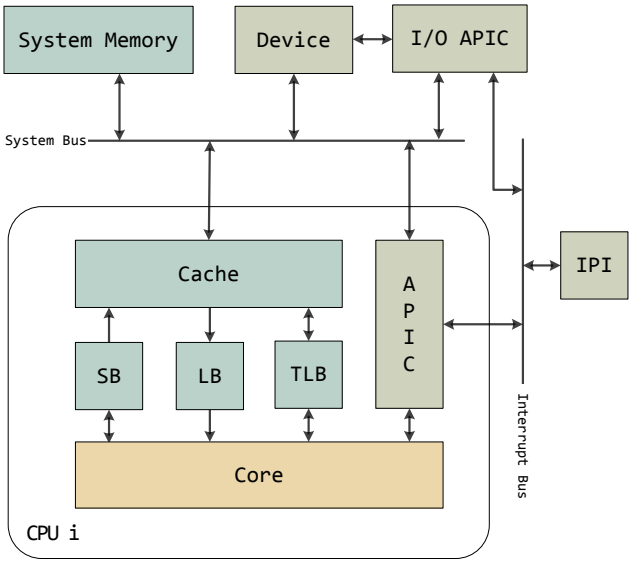

Figure 3.2: Abstract machine

3.2 Abstract x86 machine

The abstract x86 machine is a labeled transition system. Let pid denote the set of processor indices, and $dvid$ – the set of device indices. Then we define the configuration of the transition system as the following record:

$$\begin{aligned} AbsMachine \triangleq [\ & core \in pid \to Core, \\ & sb \in pid \to SB, \\ & lb \in pid \to LB, \\ & tlb \in pid \to TLB, \\ & lock \in Lock, \\ & cache \in pid \to Cache, \\ & ipi \in IPI, \\ & env \in Env], \end{aligned}$$

where the environment is the physical memory and memory mapped devices:

$$\begin{aligned} Env \triangleq [\ & memory \in \mathbb{B}^{pq} \to \mathbb{B}^{64}, \\ & apic \in pid \to APIC, \\ & ioapic \in IOAPIC, \\ & device \in dvid \to Device]. \end{aligned}$$

Figure 3.2 illustrates a conceptual structure of the abstract machine.

Although there are buses in the figure, we do not explicitly model them. Data transfer between two units occurs as a part of a transition that changes the state of both units. As we can express complex guard conditions and effects for transitions, there is no need for low-level handshake protocols, and, thus, units do not have *request*/*response*/*ack* fields.

Each processor has the following components:

- a processor core: it fetches, decodes, and executes instructions sequentially in the program order. After executing an instruction but before fetching the next instruction, the core checks for exceptions, intercepts, and collects interrupts from the local APIC. Section 9 describes configuration and transitions of the core.
- a store buffer: it is a queue of stores. It delays stores on their way from the processor core to the caches/environment. Due to this delay, remote processors observe reordering of new instructions and loads ahead of old stores. More details are in Section 6.
- a load buffer: we need it to model effects of out-of-order/speculative instruction execution. It nondeterministically prefetches instructions and WC data. WC loads and instruction fetches can be reordered ahead of old instructions and loads. By prefetching, the load buffer introduces a negative delay, so that remote processors observe the required reordering (Section 7).
- a translation-lookaside buffer: it traverses page tables and collects translations. More details are in Section 8.
- a cache: it models data and code caches of real processors and implements the MOESI cache-coherence protocol. More details are in Section 5.
- a local APIC: it accepts interrupts from devices and other processors, and forwards them to the local processor. It also allows to send interprocessor interrupts. More details are in Section 10.

Besides processors, the abstract machine has a physical memory, devices, an I/O APIC, an interprocessor interrupt (IPI) unit, and a memory lock. The physical memory is a map from quadword aligned addresses to quadwords. We do not model devices and the I/O APIC, so the $device$ and $ioapic$ components of the abstract machine are just placeholders to simplify future extensions. There is no IPI unit in a real processor, but we need it to specify how interprocessor interrupts are transfered between processors (Section 10.6).

In order to simplify definitions, we group units that are accessible via the system bus into a single component, which is called the environment (Section 4).

The $lock$ component of the abstract machine has type $Lock \triangleq pid \cup \{\epsilon\}$. When the lock is acquired ($lock \neq \epsilon$), only the processor whose index is equal to the $lock$, can access the caches and the environment. This allows us to execute instructions atomically. We will use the following functions to work with the lock:

$$busy(lock \in Lock, i \in pid) \in \mathbb{B} \triangleq (lock \neq i \wedge lock \neq \epsilon),$$
$$owns(lock \in Lock, i \in pid) \in \mathbb{B} \triangleq (lock = i),$$
$$can\text{-}acquire(lock \in Lock, i \in pid) \in \mathbb{B} \triangleq (lock = \epsilon),$$
$$acquire(lock \in Lock, i \in pid) \in Lock \triangleq i,$$
$$can\text{-}release(lock \in Lock, i \in pid) \in \mathbb{B} \triangleq (lock = i),$$
$$release(lock \in Lock, i \in pid) \in Lock \triangleq \epsilon.$$

CHAPTER

FOUR

ENVIRONMENT

The environment includes the physical memory, APICs, and devices. We access the environment with the following functions:

$$\textit{can-read}(\textit{env}, i, \textit{core}, \textit{lock}, \textit{pa}, \textit{mask}) \in \mathbb{B},$$
$$\textit{read}(\textit{env}, i, \textit{core}, \textit{pa}, \textit{mask}) \in (\textit{Env}, \mathbb{B}^{64}),$$
$$\textit{can-write}(\textit{env}, i, \textit{core}, \textit{lock}, \textit{pa}, \textit{mask}, \textit{data}) \in \mathbb{B},$$
$$\textit{write}(\textit{env}, i, \textit{core}, \textit{pa}, \textit{mask}, \textit{data}) \in \textit{Env},$$

where $\textit{env} \in \textit{Env}, i \in \textit{pid}, \textit{core} \in \textit{Core}, \textit{pa} \in \mathbb{B}^{pq}, \textit{mask} \in \mathbb{B}^8, \textit{data} \in \mathbb{B}^{64}$.

The $\textit{write}$ function processes the write access and returns a new environment. A read access can have side effects, therefore, the $\textit{read}$ function returns the read result and a new environment.

The functions simply forward the access to one of the $\textit{env}$ components. Which component gets the access depends on the address $\textit{pa}$ and on a so-called memory map. The memory map partitions the physical address space into regions. Each region is backed up either by an APIC, a device, or the physical memory. The memory map is implementation-dependent. We abstract it into predicates:

$$\textit{in-apic-range}(\textit{core} \in \textit{Core}, \textit{pa} \in \mathbb{B}^{pq}, \textit{mask} \in \mathbb{B}^8) \in \mathbb{B},$$
$$\textit{in-ioapic-range}(\textit{ioapic} \in \textit{IOAPIC}, \textit{pa} \in \mathbb{B}^{pq}, \textit{mask} \in \mathbb{B}^8) \in \mathbb{B},$$
$$\textit{in-device-range}(\textit{device} \in (\textit{dvid} \to \textit{Device}), \textit{pa} \in \mathbb{B}^{pq}, \textit{mask} \in \mathbb{B}^8) \in \mathbb{B}.$$

The first predicate is defined in Section 10, and we leave the remaining two predicates undefined since we do not model devices.

The $\textit{read}, \textit{write}$ functions have the following structure:

- check that the lock is not acquired by another processor;
- if the access is to a memory-mapped region, then forward the access to the corresponding device, using its $\textit{read}, \textit{write}$ functions;
- otherwise, the access is to the physical memory, use $\textit{env.memory}$ component.

In case the check fails or the corresponding device is not ready, the guard predicate $\textit{can-read}/\textit{can-write}$ forbids invoking the $\textit{read}/\textit{write}$ function. Formally, we define the functions and the predicates as follows:

$\mathit{can\text{-}read}(\mathit{env}, i, \mathit{core}, \mathit{lock}, \mathit{pa}, \mathit{mask}) \in \mathbb{B} \triangleq$
  if $\mathit{busy}(\mathit{lock}, i)$ **then** 0
  else if $\mathit{in\text{-}apic\text{-}range}(\mathit{core}, \mathit{pa}, \mathit{mask})$ **then**
    $\mathit{can\text{-}read}(\mathit{env}.\mathit{apic}[i], \mathit{core}, \mathit{pa}, \mathit{mask})$
  else if $\mathit{in\text{-}ioapic\text{-}range}(\mathit{env}.\mathit{ioapic}, \mathit{pa}, \mathit{mask})$ **then**
    $\mathit{can\text{-}read}(\mathit{env}.\mathit{ioapic}, \mathit{pa}, \mathit{mask})$
  else if $\mathit{in\text{-}device\text{-}range}(\mathit{env}.\mathit{device}, \mathit{pa}, \mathit{mask})$ **then**
    $\mathit{can\text{-}read}(\mathit{env}.\mathit{device}, \mathit{pa}, \mathit{mask})$
  else 1,

$\mathit{read}(\mathit{env}, i, \mathit{core}, \mathit{pa}, \mathit{mask}) \in (\mathit{Env}, \mathbb{B}^{64}) \triangleq$
  if $\mathit{in\text{-}apic\text{-}range}(\mathit{core}, \mathit{pa}, \mathit{mask})$ **then**
    let $(\mathit{apic}', \mathit{data}) = \mathit{read}(\mathit{env}.\mathit{apic}[i], \mathit{core}, \mathit{pa}, \mathit{mask})$ **in**
    $(\mathit{env}$ **with** $[\mathit{apic}[i] \mapsto \mathit{apic}'], \mathit{data})$
  else if $\mathit{in\text{-}ioapic\text{-}range}(\mathit{env}.\mathit{ioapic}, \mathit{pa}, \mathit{mask})$ **then**
    let $(\mathit{ioapic}', \mathit{data}) = \mathit{read}(\mathit{env}.\mathit{ioapic}, \mathit{pa}, \mathit{mask})$ **in**
    $(\mathit{env}$ **with** $[\mathit{ioapic} \mapsto \mathit{ioapic}'], \mathit{data})$
  else if $\mathit{in\text{-}device\text{-}range}(\mathit{env}.\mathit{device}, \mathit{pa}, \mathit{mask})$ **then**
    let $(\mathit{device}', \mathit{data}) = \mathit{read}(\mathit{env}.\mathit{device}, \mathit{pa}, \mathit{mask})$ **in**
    $(\mathit{env}$ **with** $[\mathit{device} \mapsto \mathit{device}'], \mathit{data})$
  else $(\mathit{env}, \mathit{env}.\mathit{memory}[\mathit{pa}])$,

$\mathit{can\text{-}write}(\mathit{env}, i, \mathit{core}, \mathit{lock}, \mathit{pa}, \mathit{mask}, \mathit{data}) \in \mathbb{B} \triangleq$
  if $\mathit{busy}(\mathit{lock}, i)$ **then** 0
  else if $\mathit{in\text{-}apic\text{-}range}(\mathit{core}, \mathit{pa}, \mathit{mask})$ **then**
    $\mathit{can\text{-}write}(\mathit{env}.\mathit{apic}[i], \mathit{core}, \mathit{pa}, \mathit{mask}, \mathit{data})$
  else if $\mathit{in\text{-}ioapic\text{-}range}(\mathit{env}.\mathit{ioapic}, \mathit{pa}, \mathit{mask})$ **then**
    $\mathit{can\text{-}write}(\mathit{env}.\mathit{ioapic}, \mathit{pa}, \mathit{mask}, \mathit{data})$
  else if $\mathit{in\text{-}device\text{-}range}(\mathit{env}.\mathit{ioapic}, \mathit{pa}, \mathit{mask})$ **then**
    $\mathit{can\text{-}write}(\mathit{env}.\mathit{device}, \mathit{pa}, \mathit{mask}, \mathit{data})$
  else 1,

$\mathit{write}(\mathit{env}, i, \mathit{core}, \mathit{pa}, \mathit{mask}, \mathit{data}) \in \mathit{Env} \triangleq$
  if $\mathit{in\text{-}apic\text{-}range}(\mathit{core}, \mathit{pa}, \mathit{mask})$ **then**
    let $\mathit{apic}' = \mathit{write}(\mathit{env}.\mathit{apic}[i], \mathit{core}, \mathit{pa}, \mathit{mask}, \mathit{data})$ **in**
    env **with** $[\mathit{apic}[i] \mapsto \mathit{apic}']$
  else if $\mathit{in\text{-}ioapic\text{-}range}(\mathit{env}.\mathit{ioapic}, \mathit{pa}, \mathit{mask})$ **then**
    let $\mathit{ioapic}' = \mathit{write}(\mathit{env}.\mathit{ioapic}, \mathit{pa}, \mathit{mask}, \mathit{data})$ **in**
    env **with** $[\mathit{ioapic} \mapsto \mathit{ioapic}']$
  else if $\mathit{in\text{-}device\text{-}range}(\mathit{env}.\mathit{device}, \mathit{pa}, \mathit{mask})$ **then**
    let $\mathit{device}' = \mathit{write}(\mathit{env}.\mathit{device}, \mathit{pa}, \mathit{mask}, \mathit{data})$ **in**
    env **with** $[\mathit{device} \mapsto \mathit{device}']$
  else let $x = \mathit{combine}((\mathit{env}.\mathit{memory}[\mathit{pa}], 1^8), (\mathit{data}, \mathit{mask}))$ **in**
    env **with** $[\mathit{memory}[\mathit{pa}] \mapsto x]$.

CHAPTER

FIVE

CACHE

A real processor has several caches: L1 data and instruction caches, L2 caches, and sometimes L3 caches. Hardware guarantees consistency between all these caches. Thus, we can model them as a single abstract cache. The size of a cache line is not fixed by the ISA, but it is a multiple of 8 bytes. For real caches, the size is typically 64 bytes. Transitions of our abstract cache are nondeterministic. This allows us to simulate any cache lines size using 8-byte cache lines.

5.1 MOESI protocol

Consistency between caches on different processors is maintained with the help of the MOESI cache coherence protocol [SS86]. Each line can be in one of the five states:

- Modified: only this cache contains the actual data, and it is responsible for the writeback. The data in the memory might be stale.
- Owned: this cache contains the actual data, and it is responsible for writeback. Other caches might contain the same data in the 'shared' state. The data in the memory might be stale.
- Exclusive: only this cache contains the actual data, and no writeback is required. The data in the memory is up-to-date.
- Shared: this cache contains the actual data, but it is not responsible for writeback. The data in the memory might be stale if another cache has the data in the 'owned' state.
- Invalid: this cache line does not contain the actual data. Other caches might contain the actual data. The data in the memory might be stale.

We define the abstract cache as a map from physical quadword addresses to line data and line states:

$$Cache \triangleq [state \in \mathbb{B}^{pq} \to MOESI, data \in \mathbb{B}^{pq} \to \mathbb{B}^{64}],$$

where $MOESI \triangleq \{M, O, E, S, I\}$.

Consider a cache line corresponding to the address pa in a cache i. The MOESI protocol specifies the following transitions for the cache line:

- if $cache[i].state[pa] = M$ then
  - on read request from the local processor:
    - forward $cache[i].data[pa]$ to the processor;
    - leave the line state unchanged.
  - on write request from the local processor:
    - write the data to $cache'[i].data[pa]$;
    - leave the line state unchanged.
  - on read probe from a remote cache j:
    - forward $cache[i].data[pa]$ to the remote cache;
    - change the line state to $cache'[i].state[pa] = O$;
    - the remote cache sets $cache'[j].state[pa] = S$.
  - on write probe from a remote cache j:
    - forward $cache[i].data[pa]$ to the remote cache;
    - change the line state to $cache'[i].state[pa] = I$;
    - the remote cache sets $cache'[j].state[pa] = M$.
  - at any time:
    - writeback $cache[i].data[pa]$ to the memory;
    - change the line state to $cache'[i].state[pa] = E$.
- if $cache[i].state[pa] = O$ then
  - on read request from the local processor:
    - forward $cache[i].data[pa]$ to the processor;
    - leave the line state unchanged.
  - on write request from the local processor:
    - issue write probe to other caches;
    - write data to $cache'[i].data[pa]$;
    - change the line state to $cache'[i].state[pa] = M$;
    - remote caches invalidate the corresponding line.
  - on read probe from a remote cache j:
    - forward $cache[i].data[pa]$ to the remote cache;
    - leave the line state unchanged;
    - the remote cache sets $cache'[j].state[pa] = S$.
  - on write probe from a remote processor j:
    - forward $cache[i].data[pa]$ to the remote cache;
    - change the line state to $cache'[i].state[pa] = I$;
    - the remote cache sets $cache'[j].state[pa] = M$.
  - at any time:
    - writeback $cache[i].data[pa]$ to the memory;
    - change the line state to $cache'[i].state[pa] = S$.
- if $cache[i].state[pa] = E$ then
  - on read request from the local processor:
    - forward $cache[i].data[pa]$ to the processor;
    - leave the line state unchanged.
  - on write request from the local processor:
    - write data to $cache'[i].data[pa]$;
    - change the line state to $cache'[i].state[pa] = M$.
  - on read probe from a remote cache j:

        * forward $cache[i].data[pa]$ to the remote cache (this is optional);
        * change the line state to $cache'[j].state[pa] = S$;
        * the remote cache sets $cache'[j].state[pa] = S$.
    - on write probe from a remote cache j:
        * forward $cache[i].data[pa]$ to the remote cache (this is optional);
        * change the line state to $cache'[i].state[pa] = I$;
        * the remote cache sets $cache'[j].state[pa] = M$.
    - at any time:
        * set the line state to $cache'[i].state[pa] = I$.
- if $cache[i].state[pa] = S$ then
    - on read request from the local processor:
        * forward $cache[i].data[pa]$ to the processor;
        * leave the line state unchanged.
    - on write request from the local processor:
        * issue write probe to other caches;
        * write data to $cache'[i].data[pa]$;
        * change the line state to $cache'[i].state[pa] = M$;
        * remote caches invalidate the corresponding line.
    - on read probe from a remote cache j:
        * forward $cache[i].data[pa]$ to the remote cache (this is optional);
        * leave the line state unchanged;
        * the remote cache sets $cache'[j].state[pa] = S$.
    - on write probe from a remote processor j:
        * forward $cache[i].data[pa]$ to the remote cache (this is optional);
        * change the line state to $cache'[i].state[pa] = I$;
        * the remote cache sets $cache'[j].state[pa] = M$.
    - at any time:
        * set the line state to $cache'[i].state[pa] = I$.
- if $cache[i].state[pa] = I$ then
    - on read request from the local processor:
        * issue read probe to other processors;
        * fetch the data from caches or from memory to $cache'[i].data[pa]$;
        * forward $cache'[i].data[pa]$ to the local processor;
        * if $\forall j \in pid : j \neq i \Rightarrow cache[j].state[pa] = I$ then change the line state to $cache'[i].state[pa] = E$;
        * if $\exists j \in pid : j \neq i \Rightarrow cache[j].state[pa] \neq I$ then change the line state to $cache'[i].state[pa] = S$.
    - on write request from the local processor:
        * issue write probe to other processors;
        * fetch the data from caches or from memory to $temp$;
        * combine the data from the processor with $temp$ into $cache'[i].data[pa]$;
        * change the line state to $cache'[i].state[pa] = M$;
        * remote caches invalidate the corresponding line.
    - on read probe from a remote processor j: do nothing.
    - on write probe from a remote processor j: do nothing.

In the subsequent section we formalize this protocol.

5.2 Memory types

Before we can describe how to integrate the MOESI protocol into our model, we need to discuss memory types. Each physical memory access has an associated memory type, which defines the caching policy and access reordering policy. The memory types have the following effects on the cache:

- Uncacheable: the cache is ignored, the access goes directly to the memory;
- Cache Disable: a cache hit invalidates the corresponding cache line (if the line is dirty it is written back), the access goes to the memory;
- Write-Combining: the cache is ignored, the access goes to the memory via the write-combining buffer;
- Write-Protect: reads are cacheable, writes are not, a write hit invalidates the line and updates the memory;
- Writethrough: reads are cacheable, a write hit updates the line and the memory, a write miss does not allocate a line;
- Writeback: reads and writes are cacheable;

Both virtual and physical addresses of memory accesses are used for the memory type computation. Each page table entry has a 3-bit field $pat\text{-}idx$, which is an index to the Page Attribute Table (PAT). The table is stored in the 64-bit PAT register and maps 3-bit indices into memory types. Thus, it is possible to control memory type of accesses on page granularity. After the translation, the physical address is matched against the memory type ranges, which are defined by the Memory Type Range Registers (MTRR). The registers map physical address ranges into memory types. Once both virtual and physical memory types are known, they are combined to produce the final memory type of the memory access. We abstract these computations into two functions:

$$pat\text{-}lookup(core \in Core, pat\text{-}idx \in \mathbb{B}^3) \in MemType,$$
$$mtrr(core \in Core, pa \in \mathbb{B}^{pq}, pat\text{-}mt \in MemType) \in MemType,$$

where $MemType \triangleq \{UC, CD, WC, WP, WT, WB\}$.

The first function looks up the memory type in the PAT register. The second function computes the MTRR memory type and combines it with the given PAT memory type. The functions are defined in section 12.8.

5.3 Cache model

The abstract cache maps a physical quadword aligned address to the line data and the line state:

$$Cache \triangleq [state \in \mathbb{B}^{pq} \to MOESI, data \in \mathbb{B}^{pq} \to \mathbb{B}^{64}].$$

In this section we define functions and transitions for the cache component of our abstract machine. These functions and transitions together specify a cache model that is, hopefully, sound with respect to the MOESI protocol and memory type semantics.

Other components of the abstract machine use the following interface to access the cache:

$$can\text{-}read(cache, env, core, i, pa, mask, mt) \in \mathbb{B},$$

$read(cache, env, core, i, pa, mask, mt) \in (Env, \mathbb{B}^{64})$,
$can\text{-}write(cache, env, core, i, pa, mask, data, mt) \in \mathbb{B}$,
$write(cache, env, core, i, pa, mask, data, mt) \in (Cache, Env)$,

where $cache \in (pid \to Cache), env \in Env, core \in Core, i \in pid, pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^8, data \in \mathbb{B}^{64}, mt \in MemType$.

Note that the $cache$ parameter is a map from processor indices to caches. This means that we need to know the states of other caches in order to access a single cache. For simplicity, we allow cache $read, write$ functions to work with uncacheable memory types and require them to forward such accesses to the environment.

A cache read access is handled as follows:

- if the access is cacheable, then
  - check that the line is valid in the local cache (read cache hit),
  - return the line data.
- if the access is of type 'Cache Disable', then
  - check that the line is invalid in the local cache,
  - forward access to the environment.
- if the access is of type 'Uncacheable', then forward access to the environment.

In case any check fails,the $can\text{-}read$ predicate does not hold.

Formalizing the above description, we get function definitions:

$$cacheable(mt \in MemType) \in \mathbb{B} \triangleq mt \in \{WB, WP, WT\},$$

$$\begin{aligned} &can\text{-}read(cache, env, core, i, pa, mask, mt) \in \mathbb{B} \triangleq \\ &\quad \textbf{if } cacheable(mt) \textbf{ then } cache[i].state[pa] \neq I \\ &\quad \textbf{else if } mt = CD \textbf{ then } cache[i].state[pa] = I \wedge can\text{-}read(env, core, i, pa, mask) \\ &\quad \textbf{else } can\text{-}read(env, core, i, pa, mask), \end{aligned}$$

$$\begin{aligned} &read(cache, env, core, i, pa, mask, mt) \in (Env, \mathbb{B}^{64}) \triangleq \\ &\quad \textbf{if } cacheable(mt) \textbf{ then } (env, cache[i].data[pa]) \\ &\quad \textbf{else } read(env, core, i, pa, mask). \end{aligned}$$

Handling a cache write access is more complicated because it depends on states of other caches, and updates the cache line data and state:

- if the access is of type 'Writeback', then
  - check that the line is valid in the local cache,
  - check that the line is invalid in all other caches,
  - write the access data into the line,
  - set the line state to 'Modified'.
- if the access is of type 'Writethrough', then
  - check that the line is invalid in all other caches,
  - write the access data into the line in case of write hit,
  - forward access to the environment.

  Note that in this case the line state is not updated.
- if the access is of type 'Write-Protect', then
  - check that the line is invalid in all other caches,
  - invalidate the line in the local cache,
  - forward access to the environment.

Note that mixing this access type with other types can lead to data loss as the cache line is invalidated without writing back to the memory.

- if the access is of type 'Cache Disable', then
  - check that the line is invalid in the local cache,
  - forward access to the environment.
- if the access is of type 'Uncacheable', then forward access to the environment.

The corresponding formal definitions give more details:

$$
\begin{array}{l}
\mathit{can\text{-}write}(\mathit{cache}, \mathit{env}, \mathit{core}, i, \mathit{pa}, \mathit{mask}, \mathit{data}, \mathit{mt}) \in \mathbb{B} \triangleq \\
\quad \textbf{if } \mathit{cacheable}(\mathit{mt}) \textbf{ then} \\
\qquad (\forall j \in \mathit{pid} : j \neq i \Rightarrow \mathit{cache}[j].\mathit{state}[\mathit{pa}] = I) \\
\qquad \wedge\ (\mathit{mt} = \mathit{WB} \Rightarrow \mathit{cache}[i].\mathit{state}[\mathit{pa}] \neq I) \\
\qquad \wedge\ (\mathit{mt} \in \{\mathit{WT}, \mathit{WP}\} \Rightarrow \mathit{can\text{-}write}(\mathit{env}, \mathit{core}, i, \mathit{pa}, \mathit{mask}, \mathit{data})) \\
\quad \textbf{else if } \mathit{mt} = \mathit{CD} \textbf{ then} \\
\qquad \mathit{cache}[i].\mathit{state}[\mathit{pa}] = I \wedge \mathit{can\text{-}write}(\mathit{env}, \mathit{core}, i, \mathit{pa}, \mathit{mask}, \mathit{data}) \\
\quad \textbf{else } \mathit{can\text{-}write}(\mathit{env}, \mathit{core}, i, \mathit{pa}, \mathit{mask}, \mathit{data}),
\end{array}
$$

$$
\begin{array}{l}
\mathit{write}(\mathit{cache}, \mathit{env}, \mathit{core}, i, \mathit{pa}, \mathit{mask}, \mathit{data}, \mathit{mt}) \in (\mathit{Cache}, \mathit{Env}) \triangleq \\
\quad \textbf{let } \mathit{env}' = \mathit{write}(\mathit{env}, \mathit{core}, i, \mathit{pa}, \mathit{mask}, \mathit{data}) \\
\qquad (\mathit{data}', \mathit{mask}') = \mathit{combine}((\mathit{cache}[i].\mathit{data}[\mathit{pa}], 1^8), (\mathit{data}, \mathit{mask})) \\
\qquad \mathit{cache}'_i = \mathit{cache}[i] \textbf{ with } [\mathit{data}[\mathit{pa}] \mapsto \mathit{data}'] \textbf{ in} \\
\quad \textbf{if } \mathit{cacheable}(\mathit{mt}) \textbf{ then} \\
\qquad \textbf{if } \mathit{mt} = \mathit{WB} \textbf{ then } (\mathit{cache}'_i \textbf{ with } [\mathit{state}[\mathit{pa}] \mapsto M], \mathit{env}) \\
\qquad \textbf{else if } \mathit{mt} = \mathit{WT} \textbf{ then } (\mathit{cache}'_i, \mathit{env}') \\
\qquad \textbf{else } (\mathit{cache}'_i \textbf{ with } [\mathit{state}[\mathit{pa}] \mapsto I], \mathit{env}') \\
\quad \textbf{else } (\mathit{cache}[i], \mathit{env}').
\end{array}
$$

Now we describe how cache fetches, shares, writes back and drops lines. These actions happen during cache transitions, which are labeled as follows:

$$
\begin{array}{l}
\mathit{fetch\text{-}line\text{-}from\text{-}env}(i \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mt} \in \mathit{MemType}, \mathit{code} \in \mathbb{B}), \\
\mathit{fetch\text{-}line\text{-}from\text{-}cache}(i \in \mathit{pid}, j \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mt} \in \mathit{MemType}, \mathit{code} \in \mathbb{B}), \\
\mathit{share\text{-}line}(i \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq}), \\
\mathit{writeback\text{-}line}(i \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq}), \\
\mathit{drop\text{-}line}(i \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq})
\end{array}
$$

Nondeterministic nature of the transitions allows us to simulate real caches regardless of the cache size, the line size, associativity, and eviction policy. We can even simulate effects of speculative/out-of-order instruction executions by allowing caches to speculatively fetch lines.

It is important that we do not allow caches to access uncacheable memory regions. A cache can fetch a line with the physical address pa and the memory type mt ($\mathit{cacheable}(\mathit{mt}) = 1$) if

- the page tables and the core memory typing registers specify the mt memory type for the address pa,
- or the store buffer needs to write to pa with the memory type mt.

If one of the condition holds, then the fetch is justified. More formally:

$$
\begin{array}{l}
\mathit{justified}(\mathit{core}, \mathit{sb}_i, \mathit{tlb}_i, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mt} \in \mathit{MemType}) \triangleq \\
\quad \mathit{reachable}(\mathit{core}, \mathit{tlb}_i, \mathit{pa}, \mathit{mt}, 0)
\end{array}
$$

$$\vee\ \mathit{reachable}(\mathit{core}, \mathit{tlb}_i, \mathit{pa}, \mathit{mt}, 1)$$
$$\vee\ \mathit{store\text{-}req}(\mathit{sb}_i, \mathit{pa}, \mathit{mt}).$$

The predicates *reachable* are defined in Section 9.5, and the predicate *store-req* is defined in Section 6.1.

A cache can fill in a line only when it is enabled:

$$\mathit{cache\text{-}enabled}(\mathit{core} \in \mathit{Core}) \triangleq \neg \mathit{core}.\mathit{CR0}.\mathit{CD}.$$

If other caches do not have a line, and the physical address corresponding to the line is cacheable, then the cache may load the line from the environment. In this case, the line goes to exclusive state, and the environment changes to reflect the read access:

label	$\mathit{fetch\text{-}line\text{-}from\text{-}env}(i \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mt} \in \mathit{MemType})$
guard	$\mathit{cache\text{-}enabled}(\mathit{core}[i])$, $\forall j \in \mathit{pid} : \mathit{cache}[j].\mathit{state}[\mathit{pa}] = I$, $\mathit{cacheable}(\mathit{mt})$, $\mathit{justified}(\mathit{core}[i], \mathit{sb}[i], \mathit{tlb}[i], \mathit{pa}, \mathit{mt})$, $\mathit{can\text{-}read}(\mathit{env}, \mathit{core}[i], \mathit{pa}, 1^8)$
effect	$\mathit{cache}'[i].\mathit{state}[\mathit{pa}] = E$, $(\mathit{env}', \mathit{data}) = \mathit{read}(\mathit{env}, \mathit{core}[i], \mathit{pa}, 1^8)$, $\mathit{cache}'[i].\mathit{data}[\mathit{pa}] = \mathit{data}$

A cache may fetch a line from some other cache, if the remote cache has the line in owned or shared state:

label	$\mathit{fetch\text{-}line\text{-}from\text{-}cache}(i \in \mathit{pid}, j \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mt} \in \mathit{MemType})$
guard	$\mathit{cache\text{-}enabled}(\mathit{core}[i])$, $\mathit{cache}[i].\mathit{state}[\mathit{pa}] = I$, $\mathit{cache}[j].\mathit{state}[\mathit{pa}] \in \{O, S\}$, $\mathit{cacheable}(\mathit{mt})$, $\mathit{justified}(\mathit{core}[i], \mathit{sb}[i], \mathit{tlb}[i], \mathit{pa}, \mathit{mt})$
effect	$\mathit{cache}'[i].\mathit{state}[\mathit{pa}] = S$, $\mathit{cache}'[i].\mathit{data}[\mathit{pa}] = \mathit{cache}[j].\mathit{data}[\mathit{pa}]$

A cache may share any line in exclusive or modified state:

label	$\mathit{share\text{-}line}(i \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq})$
guard	$\mathit{cache}[i].\mathit{state}[\mathit{pa}] \in \{E, M\}$
effect	$\mathit{cache}'[i].\mathit{state}[\mathit{pa}] = \begin{cases} S \text{ if } \mathit{cache}[i].\mathit{state}[\mathit{pa}] = E, \\ O \text{ otherwise} \end{cases}$

A cache may writeback any line in modified or owned state:

label	$\mathit{writeback\text{-}line}(i \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq})$
guard	$\mathit{cache}[i].\mathit{state}[\mathit{pa}] \in \{M, O\}$, $\mathit{can\text{-}write}(\mathit{env}, \mathit{core}[i], \mathit{pa}, 1^8, \mathit{cache}[i].\mathit{data}[\mathit{pa}])$
effect	$\mathit{cache}'[i].\mathit{state}[\mathit{pa}] = \begin{cases} E \text{ if } \mathit{cache}[i].\mathit{state}[\mathit{pa}] = M, \\ S \text{ otherwise,} \end{cases}$ $\mathit{env}' = \mathit{write}(\mathit{env}, \mathit{core}[i], \mathit{pa}, 1^8, \mathit{cache}[i].\mathit{data}[\mathit{pa}])$

A cache may drop any clean line:

label	$\mathit{drop\text{-}line}(i \in \mathit{pid}, \mathit{pa} \in \mathbb{B}^{pq})$
guard	$\mathit{cache}[i].\mathit{state}[\mathit{pa}] \in \{E, S\}$
effect	$\mathit{cache}'[i].\mathit{state}[\mathit{pa}] = I$

CHAPTER

SIX

STORE BUFFER

Each processor of our abstract machine has a store buffer, which models effects of the write and write-combining buffers in real processors. The store buffer is a queue of stores and store fences. Two adjacent stores in the buffer can be swapped. Moreover, two adjacent WC stores with to the same address may be combined into a single store.

In order to simplify store forwarding, we maintain two maps in the store buffer: $data$ and cnt:

$$SB \triangleq [buffer \in SBItem^*, data \in \mathbb{B}^{pq} \to \mathbb{B}^{64}, cnt \in \mathbb{B}^{pq} \to \mathbb{N}^8, uc \in \mathbb{N}],$$

where $SBItem = Store \cup \{SFENCE\}$ and

$$Store \triangleq [pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^8, data \in \mathbb{B}^{64}, mt \in MemType].$$

For each byte of every physical quadword we maintain the number of stores in the buffer that modify the byte:

$$\forall pa \in \mathbb{B}^{pq}, k \in \mathbb{N}_8 : \\ sb.cnt[pa][k] = |\{w \in Store \mid w \in sb.buffer \wedge w.pa = pa \wedge w.mask[k]\}|$$

Since loads cannot overpass an old UC store, we can issue a load access only if the store buffer does not contain an UC store. Thus, we need to maintain the number of UC store in the store buffer:

$$sb.uc = |\{w \in Store \mid w \in sb.buffer \wedge w.mt = UC\}|$$

6.1 Forwarding and writing

Given the counter, it is easy to detect a byte hit:

$$byte\text{-}hit(sb \in SB, pa \in \mathbb{B}^{pq}, k \in \mathbb{N}_8) \in \mathbb{B} \triangleq sb.cnt[pa][k] > 0.$$

The $data$ component of the store buffer maintains for each physical quadword the value of the latest store to that quadword. Store buffer forwarding can be defined as follows:

$$
\begin{array}{l}
\mathit{forward}(\mathit{sb} \in \mathit{SB}, \mathit{pa} \in \mathbb{B}^{pq}) \in (\mathbb{B}^{64}, \mathbb{B}^{8}) \triangleq \\
\quad \textbf{choose } \mathit{mask} \in \mathbb{B}^{8} : (\forall k \in \mathbb{N}_{8} : \mathit{mask}[i] = \mathit{byte\text{-}hit}(\mathit{sb}, \mathit{pa}, k)) \textbf{ in} \\
\quad (\mathit{sb}.\mathit{data}[\mathit{pa}], \mathit{mask}).
\end{array}
$$

The processor core adds stores and store fences into the store buffer using the write function:

$$
\begin{array}{l}
\mathit{write}(\mathit{sb} \in \mathit{SB}, \mathit{item} \in \mathit{SBItem}) \in \mathit{SB} \triangleq \\
\quad \textbf{if } \mathit{item} = \mathit{SFENCE} \textbf{ then } \mathit{sb} \textbf{ with } [\mathit{buffer} \mapsto \mathit{sb}.\mathit{buffer} \circ [\mathit{item}]] \\
\quad \textbf{else} \\
\quad\quad \textbf{let } \mathit{pa} = \mathit{item}.\mathit{pa} \\
\quad\quad\quad \mathit{mask} = \mathit{item}.\mathit{mask} \\
\quad\quad\quad \mathit{mt} = \mathit{item}.\mathit{mt} \\
\quad\quad\quad x = \mathit{combine}((\mathit{sb}.\mathit{data}[\mathit{pa}], 1^{8}), (\mathit{item}.\mathit{data}, \mathit{mask})) \\
\quad\quad\quad \mathit{data}' = \mathit{sb}.\mathit{data} \textbf{ with } [\mathit{pa} \mapsto x] \\
\quad\quad\quad \mathit{cnt}' = \mathit{sb}.\mathit{cnt} \textbf{ with } [\mathit{pa} \mapsto \mathit{inc}(\mathit{sb}.\mathit{cnt}[\mathit{pa}], \mathit{mask})] \\
\quad\quad\quad \mathit{uc}' = \textbf{if } \mathit{mt} = \mathit{UC} \textbf{ then } \mathit{sb}.\mathit{uc} + 1 \textbf{ else } \mathit{sb}.\mathit{uc} \textbf{ in} \\
\quad\quad \mathit{sb} \textbf{ with } [\mathit{buffer} \mapsto \mathit{sb}.\mathit{buffer} \circ [\mathit{item}], \mathit{data} \mapsto \mathit{data}', \mathit{cnt} \mapsto \mathit{cnt}', \mathit{uc} \mapsto \mathit{uc}'].
\end{array}
$$

A store fence is simply appended to the end of the queue. For a store access, we need to maintain invariants of the $\mathit{data}, \mathit{cnt}, \mathit{uc}$ components of the store buffer.

The function $\mathit{inc}(x \in \mathbb{N}^{8}, \mathit{mask} \in \mathbb{B}^{8}) \in \mathbb{N}^{8}$ increments the eight counters in x according to the given mask. Let $y = \mathit{inc}(x, \mathit{mask})$, then $\forall k \in \mathbb{N}_{8}$:

$$
y[k] = \begin{cases} x[k] + 1 & \text{if } \mathit{mask}[k], \\ x[k] & \text{otherwise.} \end{cases}
$$

Likewise, we define the dec function for decrementing the counters.

Using the $\mathit{store\text{-}req}$ function, the cache detects a write request from the store buffer:

$$
\begin{array}{l}
\mathit{store\text{-}req}(\mathit{sb} \in \mathit{SB}, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mt} \in \mathit{MemType}) \triangleq \\
\quad \mathit{length}(\mathit{sb}.\mathit{buffer}) > 0 \wedge \mathit{sb}.\mathit{buffer}[0].\mathit{pa} = \mathit{pa} \wedge \mathit{sb}.\mathit{buffer}[0].\mathit{mt} = \mathit{mt}.
\end{array}
$$

6.2 Transitions

The store buffer has four nondeterministic transitions:

$$
\begin{array}{l}
\mathit{reorder\text{-}stores}(i \in \mathit{pid}, j \in \mathbb{N}), \\
\mathit{drop\text{-}leading\text{-}sfence}(i \in \mathit{pid}), \\
\mathit{combine\text{-}stores}(i \in \mathit{pid}, j \in \mathbb{N}), \\
\mathit{commit\text{-}store}(i \in \mathit{pid}, x \in \mathit{Store}).
\end{array}
$$

Any two adjacent non-conflicting stores may be reordered if one of them has the WC memory type:

label	$reorder\text{-}stores(i \in pid, j \in \mathbb{N})$
guard	$j+1 < length(sb[i].buffer)$, $sb[i].buffer[j] \neq SFENCE$, $sb[i].buffer[j+1] \neq SFENCE$, $\neg conflict(sb.buffer[j], sb.buffer[j+1])$, $sb[i].buffer[j].mt = WC \vee sb[i].buffer[j+1].mt = WC$
effect	$sb'[i].buffer[j] = sb[i].buffer[j+1]$, $sb'[i].buffer[j+1] = sb[i].buffer[j]$

where the *conflict* predicate is defined as

$$conflict(x \in Store, y \in Store) \in \mathbb{B} \triangleq x.pa = y.pa \wedge (x.mask \wedge_8 y.mask) \neq 0.$$

A fence at the front of the queue may be dropped at any time:

label	$drop\text{-}leading\text{-}sfence(i \in pid)$
guard	$0 < length(sb[i].buffer)$, $SFENCE = sb[i].buffer[0]$
effect	$n = length(sb[i].buffer) - 1$, $\forall k \in N : k < n \Rightarrow sb'[i].buffer[k] = sb[i].buffer[k+1]$

Two adjacent WC stores to the same address may be combined. Note that after combining, we need to decrement byte counters corresponding to the overlapping bytes:

label	$combine\text{-}stores(i \in pid, j \in \mathbb{N})$
guard	$j+1 < length(sb[i].buffer)$, $sb[i].buffer[j] \neq SFENCE$, $sb[i].buffer[j+1] \neq SFENCE$, $sb[i].buffer[j].pa = sb[i].buffer[j+1].pa$, $sb[i].buffer[j].mt = WC \wedge sb[i].buffer[j+1].mt = WC$
effect	$x = sb[i].buffer[j]$, $y = sb[i].buffer[j+1]$, $mask' = x.mask \vee y.mask$, $data' = combine((x.data, x.mask), (y.data, y.mask))$, $x' = x$ **with** $[mask = mask', data = data']$, $n = length(sb[i].buffer) - 1$, $\forall k \in N : k < j \Rightarrow sb'[i].buffer[k] = sb[i].buffer[k]$, $\forall k \in N : j < k \wedge k < n \Rightarrow sb'[i].buffer[k] = sb[i].buffer[k+1]$, $sb'[i].buffer[j] = x$, $sb'[i].cnt[x.pa] = dec(sb.cnt[x.pa], x.mask \wedge_8 y.mask)$

A store at the front of the queue can be written to the cache/environment if the core is not in the process of copying stores from the *mem-out* to the store buffer. We require this condition to guarantee that all stores of an instruction are in the store buffer before the first store of the instruction hits the cache/environment. The core copies stores after it has completed the execution of an instruction or of a pseudo-instruction. In

such cases the $core.phase$ is $COMPLETE$ or $SERIALIZE$.

label	$commit\text{-}store(i \in pid, x \in Store)$
guard	$(core[i].phase \in \{COMPLETE, SERIALIZE\} \Rightarrow$ $\quad core[i].mem\text{-}out = empty\text{-}mem\text{-}out)$, $0 < length(sb[i].buffer)$, $x = sb[i].buffer[0]$, $can\text{-}write(cache, env, core, i, x.pa, x.mask, x.data, x.mt)$
effect	$(cache', env') = write(cache, env, core, i, x.pa, x.mask, x.data, x.mt)$, $n = length(sb[i].buffer) - 1$, $\forall k \in N : k < n \Rightarrow sb'[i].buffer[k] = sb[i].buffer[k+1]$, $sb'[i].cnt[x.pa] = dec(sb.cnt[x.pa], x.mask)$, $sb'[i].uc = \begin{cases} sb.uc - 1 & \text{if } x.mt = UC, \\ sb.uc & \text{otherwise} \end{cases}$

CHAPTER

SEVEN

LOAD BUFFERS

A processor has two kind of buffers for loaded data: persistent and temporary. A persistent buffer retains its data for more than one instruction. We use such buffers to model

- out-of-order loads from the write-combining memory,
- out-of-order instruction fetches.

We group persistent buffers in the $lb \in LB$ component of the abstract machine:

$$LB \triangleq [\, wc \in \mathbb{B}^{pq} \to (Data, Mask),$$
$$ib \in (\mathbb{B}^{pq}, MemType) \to (Data, Mask)]$$

where $Data \triangleq \mathbb{B}^{64}$ and $Mask \triangleq \mathbb{B}^{8}$.

There is one temporary load buffer – the $mem\text{-}in$ component of the processor core. The buffer contains data loaded by the current instruction or pseudo-instruction. Before execution of the (pseudo-)instruction, the buffer is empty. Figure 7.1 illustrates how the $mem\text{-}in$ collects data.

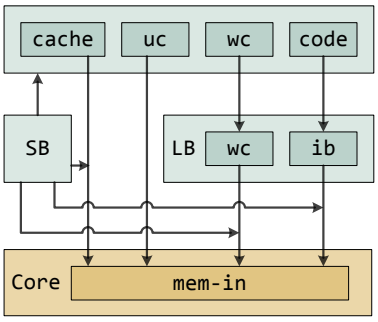

Figure 7.1: Load buffers

7.1 Loading code

When the core is in the instruction fetch/decode phase, the *load-code* transition fills the *mem-in* buffer with data from the instruction buffer combined with the result of store buffer forwarding.

label	$load\text{-}code(i \in pid, req \in ReadReq)$
guard	$core[i].phase = DECODE$
effect	$(x, m) = combine(lb[i].ib[req.pa, req.mt], forward(sb[i], req.pa))$, $core'[i].mem\text{-}in[req] = ReadReply$ **with** $\{ready \mapsto 1, data \mapsto x\}$

The instruction buffer may prefetch any instruction from cacheable and write-combining memory.

label	$prefetch\text{-}code\text{-}cacheable(i \in pid, pa \in \mathbb{B}^{pq}, mt \in MemType)$
guard	$cacheable(mt)$, $cache[i].state[pa] \neq I$, $reachable(core[i], tlb[i], pa, mt, 1)$,
effect	$data = cache[i].data[pa]$, $lb'[i].ib[pa, mt] = (data, 1^8)$,

The predicate *reachable* checks whether the physical address pa can be accessed with the memory type mt (Section 9.5).

label	$prefetch\text{-}code\text{-}wc(i \in pid, pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^8)$
guard	$can\text{-}read(env, core, i, pa, mask)$, $reachable(core[i], tlb[i], pa, WC, 1)$
effect	$(env', data) = read(env, core, i, pa, mask)$, $lb'[i].ib[pa, WC] = (data, mask)$

Instructions in uncacheable memory are fetched only on core request. We define the predicate *code-req* in section 9.5.

label	$fetch\text{-}code(i \in pid, pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^8, mt \in MemType)$
guard	$core[i].phase = DECODE$, $code\text{-}req(core[i], pa, mask, mt)$, $can\text{-}read(env, core, i, pa, mask, mt)$
effect	$(env', data) = read(env, core, i, pa, mask, mt)$, $lb'[i].ib[pa, mt] = (data, mask)$

7.2 Loading data

When the core is in a phase that requires data (execute, jump to interrupt service routine, switch to host mode), the following transition fills in the *mem-in* buffer with data from the cache:

label	$load\text{-}cacheable(i \in pid, req \in ReadReq)$
guard	$core[i].phase \in \{EXECUTE, VMEXIT, JISR1, JISR2\}$ $cacheable(req.mt)$, $cache[i].state[req.pa] \neq I$, $reachable(core[i], tlb[i], req.pa, req.mt, 0)$, $sb[i].uc = 0$
effect	$(x, m) = combine((cache[i].data[req.pa], req.mask), forward(sb[i], req.pa))$, $core'[i].mem\text{-}in[req] = ReadReply$ **with** $\{ready \mapsto 1, data \mapsto x\}$

Since loads cannot overpass old UC stores, the transition checks that the store buffer does not contain UC stores.

Uncachable data can be loaded only on core request $data\text{-}req$, which is defined in Section 9.5. Since an UC load cannot overpass an old store, so we check that there are no stores in the store buffer.

label	$load\text{-}in\text{-}order(i \in pid, req \in ReadReq)$
guard	$core[i].phase \in \{EXECUTE, VMEXIT, JISR1, JISR2\}$, $data\text{-}req(core[i], req) = mask$, $can\text{-}read(cache, env, core, i, req.pa, req.mask, req.mt)$, $sb[i].uc = 0$, $req.mt = UC \Rightarrow length(sb[i].buffer) = 0$
effect	$(cache', env', data) = read(cache, env, core, i, req.pa, req.idmask, req.mt)$, $(data', mask') = combine((data, mask), forward(sb[i], req.pa))$, $core'[i].mem\text{-}in[req] = ReadReply$ **with** $\{ready \mapsto 1, data \mapsto x\}$

We load WC data to the $mem\text{-}in$ from the wc load buffer.

label	$load\text{-}wc(i \in pid, req \in ReadReq)$
guard	$core[i].phase \in \{EXECUTE, VMEXIT, JISR1, JISR2\}$, $sb[i].uc = 0$ $req.mt = WC$
effect	$(x, m) = combine(lb[i].wc[req.pa], forward(sb[i], req.pa))$, $good\text{-}mask = (req.mask \wedge_8 m = req.mask)$, $core[i].mem\text{-}in[req] = \begin{cases} ReadReply \text{ with } \{ready \mapsto 1, data \mapsto x\} & \text{if } good\text{-}mask \\ core[i].mem\text{-}id[req] & \text{otherwise} \end{cases}$

The wc load buffer may prefetch data from the write-combining memory.

label	$prefetch\text{-}wc(i \in pid, pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^8)$
guard	$can\text{-}read(env, core, i, pa, mask)$, $reachable(core[i], tlb[i], pa, WC, 0)$, $sb[i].uc = 0$
effect	$(env', data) = read(env, core, i, pa, mask)$, $lb'[i].wc[pa] = (data, mask)$

7.3 Flushing

The temporary load buffer $mem\text{-}in$ is flushed before each instruction and pseudo-instruction. In other words, its value is set to

$$empty\text{-}mem\text{-}in \triangleq \lambda req \in ReadReq \to ReadReply(0,0).$$

The wc load buffer is flushed after the LFENCE instruction, before a locked and I/O instruction, and after a serializing event. The instruction buffer is flushed only after a serializing event:

$$empty\text{-}wc \triangleq \lambda pa \in B^{pq} \to (0,0),$$
$$empty\text{-}ib \triangleq \lambda(pa, mt) \in (\mathbb{B}^{pq}, mt \in MemType) \to (0,0),$$
$$\mathit{flush}\text{-}wc(lb \in LB) \in LB \triangleq lb \ \textbf{with}\ [wc \mapsto empty\text{-}wc],$$
$$\mathit{flush}\text{-}all(lb \in LB) \in LB \triangleq lb \ \textbf{with}\ [wc \mapsto empty\text{-}wc, ib \mapsto empty\text{-}ib].$$

CHAPTER

EIGHT

TRANSLATION-LOOKASIDE BUFFER

Real translation-lookaside buffer caches translations of virtual addresses. Our abstract TLB stores not only translations but states of page table traversal, which we call *walks*. Thus, a TLB is a set of walks:

$$TLB \triangleq [walks \in Walk \to \mathbb{B}]$$

In order to get a translation for a virtual page address[1] $vpfn$ and access rights r, we initiate a walk w with $w.vpfn = vpfn \wedge w.r = r$ at the root page table. Then we perform several *walk extensions* until we reach a leaf page table. The walk w contains all necessary information for walk extension, including

- $w.level$, the current page table level.
- $w.large$, the flag that indicates whether walk has reached a large page.
- $w.asid$, the address space identifier.
- $w.g$, the flag that indicates whether the walk is global or not. There are events that flush non-global walks (such as writing to the $CR3$ register).
- $w.ba$, the physical base address of the current page table (quadword aligned);
- $w.pat\text{-}idx$, the index into the page attribute table, which is used to calculate the memory type of the memory region where the page table lies.

Formally, we define a walk as a record:

$$\begin{aligned} Walk \triangleq [\, & vpfn \in \mathbb{B}^{vp}, r \in Rights, level \in \mathbb{N}, large \in \mathbb{B}, g \in \mathbb{B}, \\ & ba \in \mathbb{B}^{pq}, pat\text{-}idx \in \mathbb{B}^3] \end{aligned}$$

where $Rights \triangleq [write \in \mathbb{B}, user \in \mathbb{B}, code \in \mathbb{B}]$

[1] virtual page address is also called virtual page frame number

8.1 Page Tables

The number of levels in the page table hierarchy depends on the paging mode, and it can be 2, 3, or 4. Let us denote this number as dot (depth of translation). A virtual page address $x \in B^{vp}$ is split into dot bitvectors:

$$x[r_{dot} - 1 : 0] = x[r_{dot} - 1 : r_{dot-1}] \circ \ldots \circ x[r_1 - 1 : r_0],$$

The actual value of $(r_{dot}, \ldots, r_1, r_0)$ also depends on the paging mode, and it is

- (20, 10, 0) in legacy 32-bit mode without physical address extensions,
- (20, 18, 9, 0) in legacy 32-bit mode with physical address extensions,
- (36, 27, 18, 9, 0) in long mode.

A bitvector $x[r_j - 1 : r_{j-1}]$ is the index of a page table entry in the page table of level j. We abstract the index calculation into function $pte\text{-}idx$:

$$pte\text{-}idx(core \in Core, vpfn \in \mathbb{B}^{vp}, level \in \{2, 3, 4\}) \in Index.$$

Thus, $pte\text{-}idx(core, x, j) = x[r_j - 1 : r_{j-1}]$ and $Index = \bigcup_{j=1}^{dot} \mathbb{B}^{r_j - r_{j-1}}$. Entries in all page tables have the same size, which depends on the paging mode and is computed by $pte\text{-}size(core \in Core) \in \{4, 8\}$. A page table has size 4096 bytes[2], therefore, a page table entry size is equal to $min_{j=1}^{dot}(4096/2^{r_j - r_{j-1}})$.

Once we know the page table base address ba, the entry index idx, the entry size $size$, it is easy to compute the entry address $ba + idx * size$. However, there are some complications because the base address is quadword aligned, and we want the entry address to be quadword aligned. Clearly, if the entry size is smaller that eight bytes, then we need a byte select mask. We use function $pte\text{-}addr$ and $pte\text{-}mask$ to compute the entry address and the byte select mask:

$$pte\text{-}addr(core \in Core, ba \in \mathbb{B}^{pq}, idx \in Index) \in \mathbb{B}^{pq},$$
$$pte\text{-}mask(core \in Core, ba \in \mathbb{B}^{pq}, idx \in Index) \in \mathbb{B}^{8}.$$

Let $s = pte\text{-}size(core_i)$ and $t = ba \circ 0^3 + zxt_{pq+3}(idx) * s$, then

$$pte\text{-}addr(core, ba, idx) = t[pq + 3 : 3],$$
$$pte\text{-}mask(core, ba, idx) = \begin{cases} 1^8 & \text{if } s = 8, \\ 0^4 \circ 1^4 & \text{if } s = 4 \wedge t \bmod 8 = 0, \\ 1^4 \circ 0^4 & \text{otherwise.} \end{cases}$$

Now we can define functions for reading and writing page table entries. Let $PTE = \mathbb{B}^{64}$ be the set of all page table entries zero extended to 64 bits, then

$$\begin{aligned} &read\text{-}pte(cache, env, core, i, w \in Walk) \in (Env, PTE) \triangleq \\ &\quad \textbf{let } idx = pte\text{-}idx(w.vpfn, w.level) \\ &\qquad pa = pte\text{-}addr(w.ba, idx) \\ &\qquad mask = pte\text{-}mask(w.ba, idx) \\ &\qquad mt = mtrr(core[i], pa, pat\text{-}lookup(core, w.pat\text{-}idx)) \textbf{ in} \\ &\quad read(cache, env, core, i, pa, mask, mt), \end{aligned}$$

[2] except the top-level page table in legacy 32-bit mode with physical address extensions, which is 32 bytes long and has four 8

$$
\begin{array}{l}
\mathit{write\text{-}pte}(\mathit{cache}, \mathit{env}, \mathit{core}, i, w \in \mathit{Walk}, \mathit{pte} \in \mathit{PTE}) \in (\mathit{Cache}, \mathit{Env}) \triangleq \\
\quad \textbf{let } \mathit{idx} = \mathit{pte\text{-}idx}(w.\mathit{vpfn}, w.\mathit{level}) \\
\qquad \mathit{pa} = \mathit{pte\text{-}addr}(w.\mathit{ba}, \mathit{idx}) \\
\qquad \mathit{mask} = \mathit{pte\text{-}mask}(w.\mathit{ba}, \mathit{idx}) \\
\qquad \mathit{mt} = \mathit{mtrr}(\mathit{core}[i], \mathit{pa}, \mathit{pat\text{-}lookup}(\mathit{core}, w.\mathit{pat\text{-}idx})) \textbf{ in} \\
\quad \mathit{write}(\mathit{cache}, \mathit{env}, \mathit{core}, i, \mathit{pa}, \mathit{mask}, \mathit{pte}, \mathit{mt}).
\end{array}
$$

The corresponding guard predicates $\mathit{can\text{-}read\text{-}pte}$ and $\mathit{can\text{-}write\text{-}pte}$ are defined likewise (replace 'read' or 'write' with '$\mathit{can\text{-}read}$' or '$\mathit{can\text{-}write}$').

A page table entry has many fields. The exact layout of the fields depends on the paging mode. In order to simplify definitions, we introduce an abstract page table entry AbsPTE and assume that there is a function

$$\mathit{parse\text{-}pte}(\mathit{core} \in \mathit{Core}, \mathit{pte} \in \mathit{PTE}) \in \mathit{AbsPTE},$$

which takes the binary representation of a page table entry, and returns the abstract representation.

An abstract page table entry x has the following fields:

- $x.p$, the present flag,
- $x.r$, the entry access rights,
- $x.a$, the 'accessed' flag, which indicates whether the entry was accessed by the TLB or not,
- $x.d$, the 'dirty' flag, which indicates whether the entry was used to translate a virtual address for writing,
- $x.g$, the 'global' flag, which indicates whether a translation that uses this entry is global or not,
- $x.\mathit{large}$, the large page indicator, when this flag is set, then the page table entry specifies a large page.
- $x.\mathit{ba}$, the base address of the next level page table (quadword aligned),
- $x.\mathit{pat\text{-}idx}$, the index into the page attribute table, which is used to calculate the memory type of the memory region where the next level page table lies,
- $x.\mathit{valid}$, the flag that indicates whether the binary representation of the entry is valid, i.e. reserved fields are set to correct values.

Thus, AbsPTE is a record

$$
\begin{array}{l}
\mathit{AbsPTE} \triangleq [\, p \in \mathbb{B}, r \in \mathit{Rights}, a \in \mathbb{B}, d \in \mathbb{B}, g \in \mathbb{B}, \mathit{large} \in \mathbb{B}, \\
\qquad\qquad \mathit{ba} \in \mathbb{B}^{pq}, \mathit{pat\text{-}idx} \in \mathbb{B}^3, \mathit{valid} \in \mathbb{B}].
\end{array}
$$

For setting 'accesses' and 'dirty' bits we will use functions

$$
\begin{array}{l}
\mathit{set\text{-}accessed}(\mathit{pte} \in \mathit{PTE}) \in \mathit{PTE}, \\
\mathit{set\text{-}dirty}(\mathit{pte} \in \mathit{PTE}) \in \mathit{PTE}.
\end{array}
$$

These functions as well as $\mathit{pte\text{-}idx}, \mathit{pte\text{-}size}, \mathit{pte\text{-}addr}, \mathit{pte\text{-}mask}, \mathit{parse\text{-}pte}$ are defined formally in section 13.6.

8.2 Creating and dropping walks

The TLB makes transitions only if paging is enabled:

$$\mathit{paging\text{-}enabled}(\mathit{core} \in \mathit{Core}) \triangleq \mathit{core}.\mathit{CR0}.\mathit{PG}.$$

A new walk starts the level equal to the depth of translation. The base address of the root page table is taken from the $CR3$ register. The $CR0, CR3$ registers are defined in section 12.2.

label	$\mathit{new\text{-}walk}(i \in \mathit{pid}, w \in \mathit{Walk})$
guard	$\mathit{paging\text{-}enabled}(\mathit{core}[i])$, $w.\mathit{level} = \mathit{dot}(\mathit{core}[i])$, $w.\mathit{large} = 0$, $w.\mathit{asi} = \mathit{current\text{-}asid}(\mathit{core}[i])$, $w.g = 0$, $w.\mathit{ba} = \mathit{core}[i].\mathit{CR3}.\mathit{base}[pq + 3 : 3]$, $w.\mathit{pat\text{-}idx} = 0 \circ (\mathit{core}[i].\mathit{CR3}.\mathit{PCD}) \circ (\mathit{core}[i].\mathit{CR3}.\mathit{PWT})$
effect	$\mathit{tlb}'[i].\mathit{walks}[w] = 1$

The $\mathit{current\text{-}asid}$ function is defined in section 13.5.

Any walk can be dropped at any moment.

label	$\mathit{drop\text{-}walk}(i \in \mathit{pid}, w \in \mathit{Walk})$
guard	$\mathit{paging\text{-}enabled}(\mathit{core}[i])$, $\mathit{tlb}[i].\mathit{walks}[w] = 1$
effect	$\mathit{tlb}'[i].\mathit{walks}[w] = 0$

8.3 Extending walks

A walk w that has reached the level zero is complete, and $w.\mathit{ba}$ contains the base address of the physical page corresponding to the virtual page $w.\mathit{vpfn}$. There might be complete walks with a level greater than zero. This happens when a walk reaches a large physical page.

$$\mathit{complete}(w \in \mathit{Walk}) \in \mathbb{B} \triangleq w.\mathit{level} = 0 \vee w.\mathit{large}.$$

To extend an incomplete walk, we calculate the page table entry index, fetch the corresponding page table entry, and replace some fields of the walk with the fields of the page table entry.

Consider a walk w and a page table entry pte that is going to be used for walk extension. It is possible that the page table entry is invalid (reserved bits have forbidden values), the next level page table is not present, or the walk access right do not match the page table entry access rights. Each of these conditions leads to a page fault.

$$\mathit{page\text{-}fault}(\mathit{core} \in \mathit{Core}, w \in \mathit{Walk}, \mathit{pte} \in \mathit{AbsPTE}) \in \mathbb{B} \triangleq \\ \neg \mathit{pte}.\mathit{valid} \vee \neg \mathit{pte}.p \vee \mathit{access\text{-}violation}(\mathit{core}, w.r, \mathit{pte}.r),$$

$$\begin{aligned}&\mathit{access\text{-}violation}(\mathit{core} \in \mathit{Core}, \mathit{request} \in \mathit{Rights}, \mathit{allow} \in \mathit{Rights}) \in \mathbb{B} \triangleq \\ &\quad \mathit{request}.\mathit{write} \wedge \neg \mathit{allow}.\mathit{write} \wedge (\mathit{request}.\mathit{user} \vee \mathit{core}.\mathit{CR0}.\mathit{WP}) \vee \\ &\quad \mathit{request}.\mathit{user} \wedge \neg \mathit{allow}.\mathit{user} \vee \\ &\quad \mathit{request}.\mathit{code} \wedge \neg \mathit{allow}.\mathit{code}.\end{aligned}$$

Since a real hardware TLB does not cache faulting translations, we cannot extend a walk if there is a page fault condition. We also make sure that the ‘accessed‘ and ‘dirty‘ bits of the page table entry are set approriately before extending:

$$\begin{aligned}&\mathit{can\text{-}extend}(\mathit{core} \in \mathit{Core}, w \in \mathit{Walk}, \mathit{pte} \in \mathit{AbsPTE}) \in \mathbb{B} \triangleq \\ &\quad \neg \mathit{complete}(w) \wedge \neg \mathit{page\text{-}fault}(\mathit{core}, w, \mathit{pte}) \wedge \mathit{pte}.a \\ &\quad \wedge (w.r.\mathit{write} \wedge (w.\mathit{level} = 1 \vee \mathit{pte}.\mathit{large}) \Rightarrow \mathit{pte}.d).\end{aligned}$$

The ‘accessed’ bit is checked for every access, while the ‘dirty’ bit is checked only for write accesses in a leaf page table entry.

Once the preconditions are met, the walk can be extended.

$$\begin{aligned}&\mathit{extend}(\mathit{core} \in \mathit{Core}, w \in \mathit{Walk}, \mathit{pte} \in \mathit{AbsPTE}) \in \mathit{Walk} \triangleq \\ &\quad w \ \mathbf{with}\, [\mathit{level} \mapsto w.\mathit{level} - 1, \\ &\qquad\quad \mathit{ba} \mapsto \mathit{pte}.\mathit{ba}, \\ &\qquad\quad \mathit{pat\text{-}idx} \mapsto \mathit{pte}.\mathit{pat\text{-}idx}, \\ &\qquad\quad g \mapsto \mathit{pte}.g, \\ &\qquad\quad \mathit{large} \mapsto \mathit{pte}.\mathit{large}].\end{aligned}$$

The level of the walk is decremented, the base address and memory type of the next level page table are copied from the page table entry.

The corresponding transition looks as follows:

label	$\mathit{extend\text{-}walk}(i \in \mathit{pid}, w \in \mathit{Walk})$
guard	$\mathit{paging\text{-}enabled}(\mathit{core}[i])$, $\mathit{tlb}[i].\mathit{walks}[w] = 1$, $\neg \mathit{complete}(w)$, $\mathit{can\text{-}read\text{-}pte}(\mathit{cache}, \mathit{env}, \mathit{core}, i, w)$
effect	$(\mathit{env}', \mathit{pte\text{-}raw}) = \mathit{read\text{-}pte}(\mathit{cache}, \mathit{env}, \mathit{core}, i, w)$ $\mathit{pte} = \mathit{parse\text{-}pte}(\mathit{core}[i], \mathit{pte\text{-}raw})$, $w' = \begin{cases} \mathit{extend}(\mathit{core}[i], w, \mathit{pte}) \text{ if } \mathit{can\text{-}extend}(\mathit{core}[i], w, \mathit{pte}), \\ w \text{ otherwise}, \end{cases}$ $\mathit{tlb}'[i].\mathit{walks}[w'] = 1$

The ‘accessed‘ and ‘dirty’ bits are set in a separate transition, which

1. fetches the page table entry;
2. checks that the entry is valid;
3. sets its ‘accessed’ bit;
4. sets its ‘dirty‘ bit if the entry is a leaf entry and the walk is for a write access;
5. writes the updated page table entry back.

label	$\mathit{set\text{-}accessed\text{-}dirty}(i \in \mathit{pid}, w \in \mathit{Walk})$
guard	$\mathit{paging\text{-}enabled}(\mathit{core}[i])$, $\mathit{tlb}[i].\mathit{walks}[w] = 1$, $\neg \mathit{complete}(w)$, $\mathit{can\text{-}read\text{-}pte}(\mathit{cache}, \mathit{env}, \mathit{core}, i, w)$
effect	$(\mathit{env\text{-}temp}, \mathit{pte\text{-}raw}) = \mathit{read\text{-}pte}(\mathit{cache}, \mathit{env}, \mathit{core}, i, w)$ $\mathit{pte} = \mathit{parse\text{-}pte}(\mathit{core}[i], \mathit{pte\text{-}raw})$, $\mathit{pte}' = \mathit{set\text{-}accessed}(\mathit{pte\text{-}raw})$, $\mathit{pte}'' = \begin{cases} \mathit{set\text{-}dirty}(\mathit{pte}') \text{ if } w.r.\mathit{write} \wedge (w.\mathit{level} = 1 \vee \mathit{pte}.\mathit{large}), \\ \mathit{pte}' \text{ otherwise}, \end{cases}$ $\mathit{ok1} = \neg \mathit{page\text{-}fault}(\mathit{core}[i], w, \mathit{pte})$, $\mathit{ok2} = \mathit{can\text{-}write\text{-}pte}(\mathit{cache}, \mathit{env\text{-}temp}, \mathit{core}, i, w, \mathit{pte}'')$, $\mathit{ok} = \mathit{ok1} \wedge \mathit{ok2}$, $(\mathit{cache}'[i], \mathit{env}') = \begin{cases} \mathit{write\text{-}pte}(\mathit{cache}, \mathit{env\text{-}temp}, \mathit{core}, i, w, \mathit{pte}'') \text{ if } \mathit{ok}, \\ \mathit{cache}[i], \mathit{env\text{-}temp} \text{ otherwise} \end{cases}$

For a complete walk w in a large page, we need to combine the offset in the large page (defined by the remaining parts of the $w.\mathit{vpfn}$) and the base address $w.\mathit{ba}$.

label	$\mathit{extend\text{-}large\text{-}walk}(i \in \mathit{pid}, w \in \mathit{Walk})$
guard	$\mathit{paging\text{-}enabled}(\mathit{core}[i])$, $\mathit{tlb}[i].\mathit{walks}[w] = 1$, $w.\mathit{large}, w.\mathit{level} > 0$
effect	$\mathit{idx} = \mathit{zxt}_{pq}(\mathit{pte\text{-}idx}(\mathit{core}[i], w.\mathit{vpfn}, w.\mathit{level})$, $\mathit{size} = \mathit{zxt}_{pq}(2^{\mathit{remaining\text{-}bits}(\mathit{core}[i], w.\mathit{level})} * 2^9)$, $w' = w \textbf{ with } [\mathit{level} \mapsto 0, \mathit{ba} \mapsto w.\mathit{ba} + \mathit{idx} * \mathit{size}]$ $\mathit{tlb}'[i].\mathit{walks}[w'] = 1$

where $\mathit{remaining\text{-}bits}(\mathit{core}, \mathit{level}) = r_{\mathit{level}-1}$.

8.4 Loading translations into the Core

The TLB might contain several complete walks for a single virtual address. Any of these walks can provide translation for the virtual address. We want to capture this nondeterministic choice in separate transitions that load translation into the $\mathit{tlb\text{-}in}$ buffer of the processor core.

$$\begin{aligned} \mathit{Core} \triangleq [\,&\ldots \\ &\mathit{tlb\text{-}in} \in \mathit{TLBReq} \to \mathit{TLBReply}, \\ &\ldots]. \end{aligned}$$

The TLBReq and $\mathit{TLBReply}$ types are defined in section 3.1.

The buffer is flushed before instruction fetch/decode, instruction execution, and pseudo-

instruction.

$$\mathit{empty\text{-}tlb\text{-}in} \triangleq \lambda \mathit{req} \in \mathit{TLBReq} \to \mathit{TLBReply}\ \mathbf{with}\ [\mathit{ready} \mapsto 0].$$

After the flush, two transitions, *successful-translation* and *faulting-translation*, fill in the buffer. The former transition selects a complete walk in the TLB and stores the corresponding transition in the buffer:

label	$\mathit{successful\text{-}translation}(i \in \mathit{pid}, w \in \mathit{Walk}, \mathit{req} \in \mathit{TLBReq})$
guard	$\mathit{paging\text{-}enabled}(\mathit{core}[i])$, $\mathit{core}[i].\mathit{phase} \in \{\mathit{DECODE}, \mathit{EXECUTE}, \mathit{VMEXIT}, \mathit{JISR1}, \mathit{JISR2}\}$, $\mathit{tlb}[i].\mathit{walks}[w] = 1$, $w.\mathit{level} = 0$, $\mathit{req}.\mathit{pfn} = w.\mathit{vpfn}$, $\mathit{req}.\mathit{asid} = w.\mathit{asid}$, $\mathit{req}.\mathit{rw} = w.r.\mathit{rw}$, $\mathit{req}.\mathit{us} = w.r.\mathit{us}$, $\mathit{req}.\mathit{exec} = w.r.\mathit{exec}$
effect	$x = w.\mathit{ba}[\mathit{pq} - 1 : 9]$, $t = \mathit{TLBReply}\ \mathbf{with}\ [\mathit{ready} \mapsto 1, \mathit{fault} \mapsto 0, \mathit{ba} \mapsto x, \mathit{pat\text{-}idx} \mapsto w.\mathit{pat\text{-}idx}]$, $\mathit{core}'[i].\mathit{tlb\text{-}in}[\mathit{req}] = t$

In case of a page fault, we need to compute the page fault code for the page fault handler:

$$\mathit{page\text{-}fault\text{-}code}(\mathit{present} \in \mathbb{B}, \mathit{rights} \in \mathit{Rights}) \in \mathbb{B}^8 \triangleq \\ \mathit{zxt}_8(\mathit{rights}.\mathit{code} \circ 0 \circ \mathit{rights}.\mathit{user} \circ \mathit{rights}.\mathit{write} \circ \mathit{present}).$$

Before loading a faulting translation to the core, we check whether the cause of the fault is still there by trying to extend the walk.

label	$faulting\text{-}translation(i \in pid, w \in Walk, req \in TLBReq)$
guard	$paging\text{-}enabled(core[i])$, $core[i].phase \in \{DECODE, EXECUTE, VMEXIT, JISR1, JISR2\}$, $tlb[i].walks[w] = 1$, $\neg complete(w)$, $can\text{-}read\text{-}pte(cache, env, core, i, w)$ $req.pfn = w.vpfn$, $req.asid = w.asid$, $req.rw = w.r.rw$, $req.us = w.r.us$, $req.exec = w.r.exec$,
effect	$(env', pte\text{-}raw) = read\text{-}pte(cache, env, core, i, w)$ $pte = parse\text{-}pte(core[i], pte\text{-}raw)$, $pf = page\text{-}fault(core[i], w, pte)$, $pf\text{-}code = page\text{-}fault\text{-}code(pte.p, w.r)$, $t = \begin{cases} TLBReply\ \mathbf{with}\ [ready \mapsto 1, fault \mapsto 1, fault\text{-}code \mapsto pf\text{-}code] & \text{if } pf, \\ core[i].tlb\text{-}view[req] & \text{otherwise} \end{cases}$ $core'[i].tlb\text{-}in[req] = t$

8.5 Flushing

There are three types of TLB flushes: a global flush, a local flush, and a flush of a specific translation.

A global flush removes all the walks from the TLB:

$$flush\text{-}global(tlb \in TLB) \in TLB \triangleq tlb\ \mathbf{with}\ [walks \mapsto (\lambda w \in Walk \to 0)].$$

A local flush removes all incomplete walks and walks that do not have the *global* bit set:

$$flush\text{-}local(tlb \in TLB) \in TLB \triangleq$$
$$tlb\ \mathbf{with}\ [walks \mapsto (\lambda w \in Walk \to w.g \wedge complete(w) \wedge tlb.walks[w])].$$

It is possible to flush the translations for a specific address space:

$$flush\text{-}global\text{-}with\text{-}asid(tlb \in TLB, asid \in \mathbb{B}^{32}) \in TLB \triangleq$$
$$tlb\ \mathbf{with}\ [walks \mapsto (\lambda w \in Walk \to w.asid = asid \wedge tlb.walks[w])].$$

$$flush\text{-}local\text{-}with\text{-}asid(tlb \in TLB, asid \in \mathbb{B}^{32}) \in TLB \triangleq$$
$$tlb\ \mathbf{with}\ [walks \mapsto (\lambda w \in Walk \to (w.asid \neq asid \vee w.g) \wedge complete(w) \wedge tlb.walks[w])].$$

A flush of a specific translation removes all incomplete walks and walks that correspond to the given virtual page address:

$$invlpg(tlb \in TLB, vpfn \in \mathbb{B}^{vp}, asid \in \mathbb{B}^{32}) \in TLB \triangleq$$
$$tlb\ \mathbf{with}\ [walks \mapsto (\lambda w \in Walk \to (w.asid \neq asid \vee w.vpfn \neq vpfn) \wedge$$
$$complete(w) \wedge tlb.walks[w])].$$

CHAPTER

NINE

CORE

9.1 Core configuration

The core configuration consists of architecture registers, the current instruction, an auxiliary state, commands to other units, and the view buffers.

Chapter 12 describes all architecture registers. Most of them are used inside the $execute$ function. We will need the following registers to define the core transitions:

- $CR0$: bits CD and PG in this register enable/disable caches and paging.
- $CR3$: the base address and the memory type of the root page table.
- $CR8$: bits $[3:0]$ of this register are aliased with bits $[7:4]$ of the APIC task priority register. We synchronize these bits before and after instruction execution.
- $RFLAGS$: bit IF in this register enables/disables maskable interrupts.
- GIF: enables/disables all interrupts.
- $SR[CS]$: the code segment register contains the base address and attributes of the code segment.
- RIP: the instruction pointer register stores an offset in the code segment. In the decode phase, it contains the address of the current instruction. In the execute phase, it contains the address of the next instruction, i.e. the address of the current instruction plus the length of the current instruction.
- $APIC\text{-}BASE$: the ABA component of this register is the physical base address of the memory-mapped local APIC page. The BSP bit indicates whether the current processor is a bootstrap processor or an application processor. The AE bit enables/disables the local APIC.

In the decode phase, the information about the current instruction is stored in the core components: the instruction opcode, the instruction prefixes, the immediate operand, the ModRM byte, the SIB byte, etc. This information is later used by the $execute$ function. For the top-level transitions, we will need to know the type of the instruction: atomic, I/O, serializing, fence, etc. The $prefix.lock \in \mathbb{B}$ component of the core indicates whether the instruction is atomic or not. Later we will introduce the predicates that check for other instruction types.

The auxiliary state of the core includes registers that are not part of the architecture, but are necessary for the instruction processing cycle:

- $phase$: the phase of the instruction processing cycle, we describe this register together with the core transitions in Section 9.2.
- $vtmode \in VtMode$: the virtualization mode, indicates whether the core is in guest mode or host mode: $VtMode \triangleq \{HOST, GUEST\}$.
- $fault \in Exception$: when a (pseudo-)instruction raises a fault exception, the information about the exception is saved in this register until the jump to the interrupt service routine.

  $$Exception \triangleq [vector \in \mathbb{B}^8, ecode \in \mathbb{B}^{16} \cup \{\epsilon\}, data \in \mathbb{B}^{64} \cup \{\epsilon\}],$$

  the $vector$ component specifies the type of the exception; some exceptions have an error code that is pushed into the stack before the jump to the interrupt service routine; a page fault exception stores the faulting address in the $data$ component, which is written to the $CR2$ register before the jump.
- $trap \in Exception$: this register stores a trap exception that was triggered by the instruction. The difference between a fault and a trap is that the trap does not abort the instruction, so the trap is handled after the instruction completes.
- $intercept \in Intercept$: this register stores an intercept that was triggered by the instruction in guest mode.

  $$Intercept \triangleq [code \in \mathbb{B}^{64}, info1 \in \mathbb{B}^{64}, info2 \in \mathbb{B}^{64}, intinfo \in \mathbb{B}^{64} \cup \{\epsilon\}],$$

  the $code$ component specifies the cause of the intercept and the 'info' components contain additional information specific to the cause.
- $intr\text{-}shadow \in \mathbb{B}$: the interrupt shadow indicator. After a POP SS and a MOV SS instructions all external interrupts and debug traps are inhibited until the next instruction is completed. It is necessary to allow to adjust the stack pointer in the next instruction and to keep the stack consistent. This register is set to 1 in case of successful execution of the POP SS and the MOV SS instructions. It is reset to 0 if an exception occurs or if the execution completes and the previous value of the interrupt shadow was 1. Therefore, if the POP SS is followed by another POP SS, only the first instruction inhibits interrupts.
- $jisr\text{-}event \in Event$: during a jump to an interrupt service routine, this register stores the cause of the jump. We use term 'event' to denote anything that can disrupt the fetch/decode/execute cycle (except intercepts). Interrupts and exceptions are events. In guest mode, events are either real or injected. All events except for an initialization (INIT) interrupt and a startup interprocessor interrupt (SIPI) require a jump to the interrupt service routine. We will discuss how events are collected and processed in Section 9.3. An event has the following fields:

  $$\begin{aligned} Event \triangleq [\ & type \in EventType, \\ & vector \in \mathbb{B}^8, \\ & ecode \in \mathbb{B}^{32} \cup \{\epsilon\}, \\ & data \in \mathbb{B}^{64} \cup \{\epsilon\}, \\ & injected \in \mathbb{B}], \end{aligned}$$

where

$$\begin{aligned} \mathit{EventType} \triangleq \{\ & \mathit{INIT}, \\ & \mathit{SIPI}, \\ & \mathit{NMI}, \\ & \mathit{INTR}, \mathit{VINTR}, \\ & \mathit{SPURIOUS}, \\ & \mathit{EXCP}, \mathit{SOFT\text{-}INT}\}. \end{aligned}$$

Depending on the event type, the fields of the event record have the following meaning:

- INIT: the vector, ecode, data fields are irrelevant because the processor starts initialization (similar to RESET).
- SIPI: the ecode, data fields are irrelevant because the processor jumps directly to the instruction whose address is calculated from the vector.
- non-maskable interrupt: the vector is assumed to be 2, and the ecode, data fields are irrelevant.
- maskable interrupt: the ecode, data fields are irrelevant.
- virtual maskable interrupt: the ecode, data fields are irrelevant.
- spurious interrupt: the ecode, data fields are irrelevant.
- exception: all fields are relevant and correspond to the fields of the $\mathit{Exception}$ record.
- software interrupt: the ecode, data fields are irrelevant.

Section 10 describes interrupts in more details.

An instruction may update the core registers, the memory, and the processor local units. We abstract instruction execution into the $\mathit{execute}$ function, which takes the core registers, TLB translations, memory loads and returns the new core registers, the memory stores, and commands to the processor local units. Thus, after the transition that invokes the $\mathit{execute}$ function, there are other transitions that carry out the commands to the processor local units, which are specified in the command registers:

- $\mathit{invd} \in \mathit{CacheInvd}$: when the valid flag of this register is set, the processor invalidates the cache. If the $\mathit{writeback}$ flag is set, then the cache writes back all modified lines into the main memory before invalidation.

$$\begin{aligned} \mathit{CacheInvd} \triangleq [\ & \mathit{valid} \in \mathbb{B}, \\ & \mathit{writeback} \in \mathbb{B}]. \end{aligned}$$

- $\mathit{flush\text{-}line} \in \mathit{CacheLineFlush}$: when the valid flag of this register is set, the processor waits until the cache writes back and evicts the line with the physical address specified by the addr field.

$$\begin{aligned} \mathit{CacheLineFlush} \triangleq [\ & \mathit{valid} \in \mathbb{B}, \\ & \mathit{addr} \in \mathbb{B}^{pa}]. \end{aligned}$$

- $\mathit{tlb\text{-}flush} \in \mathit{TLBFlush}$: when the valid flag of this register is set, then the processor flushed the translations in the TLB. The local, $\mathit{with\text{-}asid}$, asid fields of the register specify whether to flush local translations or all translation and whether to flush translation in the specified address space or to flush translation in all address

spaces.

$$\begin{aligned} TLBFlush \triangleq [\ & valid \in \mathbb{B}, \\ & local \in \mathbb{B}, \\ & with\text{-}asid \in \mathbb{B}, \\ & asid \in \mathbb{B}^{32}]. \end{aligned}$$

- $invlpg \in TLBFlushPage$: when the $valid$ flag of this register is set, then the processor invalidates all partial walks and walks corresponding to address specified in the $addr$ field within the specified address space.

$$\begin{aligned} TLBFlushPage \triangleq [\ & valid \in \mathbb{B}, \\ & addr \in \mathbb{B}^{va}, \\ & asid \in \mathbb{B}^{32}]. \end{aligned}$$

- $sync\text{-}tpr \in \mathbb{B}$: when this register is 1, the processor synchronizes the task priority register TPR of the APIC with the $CR8$ of the core.

Summarizing this section, we get the following definition of the core configuration:

$Core \triangleq [$ architecture registers:
$CR0 \in CtlReg0$,
$CR3 \in CtlReg3$,
$CR8 \in \mathbb{B}^{64}$,
$RFLAGS \in FlagsReg$,
$GIF \in \mathbb{B}$,
... the rest is defined in chapter 12
instruction info:
$prefix \in Prefix$,
... the rest is defined in section 12.7
the auxiliary state:
$phase \in Phase$,
$vtmode \in VtMode$,
$fault \in Exception$,
$trap \in Exception$,
$intercept \in Intercept$,
$intr\text{-}shadow \in \mathbb{B}$,
$jisr\text{-}event \in Event$,
commands:
$invd \in CacheInvd$,
$flush\text{-}line \in CacheLineFlush$,
$tlb\text{-}flush \in TLBFlush$,
$invlpg \in TLBFlushPage$,
$sync\text{-}tpr \in \mathbb{B}$,
buffers:
$mem\text{-}in \in ReadReq \to ReadReply$,
$mem\text{-}out \in WriteReq \to \mathbb{B}$,
$tlb\text{-}in \in TLBReq \to TLBReply]$.

9.2 Overview of transitions

As we are specifying an abstract machine, we can ignore performance issues and define the simplest possible processor core. The core does not have a pipeline and processes instructions sequentially in the program order.

The instruction processing cycle is quite complex, so we split it into phases as shown in Figures 9.1 and 9.2, which extend Figure 3.1. After executing an instruction but before decoding the next instruction, the core is in the *border* phase. In this phase the core checks for interrupts and exceptions. In case of an INIT interrupt, the core goes to the *init* phase. If there is another interrupt or an exception, then the core goes to the *jump to interrupt service routine (jisr)* phase.

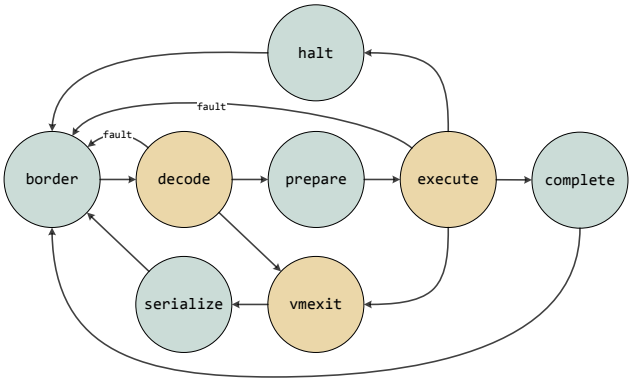

Figure 9.1: Core Transitions: instruction processing

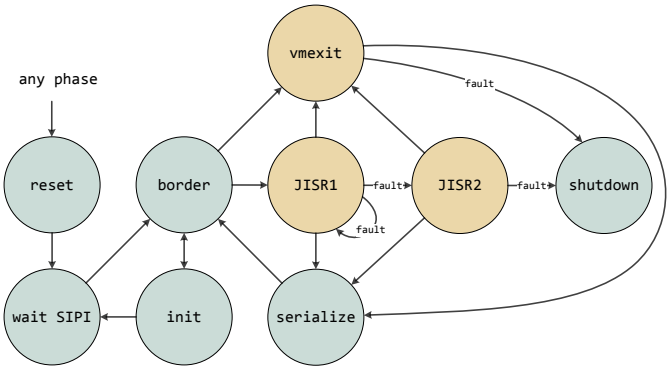

Figure 9.2: Core Transitions: interrupt processing

If there is no active event, then the core proceeds to the *fetch/decode* phase. Once the instruction is decoded, the core goes to the *prepare execution* phase, in which it tries to acquire the memory lock if the instruction has the lock prefix. After that the instruction is executed in the *execute* phase. Then the core gives the commands to processor local units in the *complete execution* phase. Finally, the core releases the memory lock (in case it has acquired it) and returns to the *border* phase.

The core might fail to decode or execute the instruction. This might happen either because of an exception or because of an guest intercept. In the former case, the exception is recorded in the $core.fault$, the execution is aborted, and the core returns to the *border* phase. In the latter case, the core moves to the *vmexit* phase, where it switches from guest mode to host mode.

An exception during the *jisr* phase may trigger a *double fault* exception, in this case the core proceeds to the *jump to double fault service routine (jisr2)* phase. An exception in this phase leads to the shutdown, i.e. the core switches to the *shutdown* phase and never leaves it.

Jumping to an interrupt service routine is a serializing event, which means that load/store buffers are flushed before fetching the next instruction. Therefore, the core goes to the *serialize* phase after a successful jump. In this phase, the core waits until the buffers are flushed.

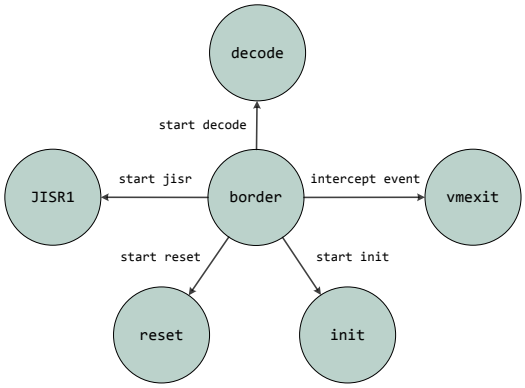

Figure 9.3: Transitions from the border phase

An intercept during a jump to a service routine leads to the *vmexit* phase. An exception in the *vmexit* phase results in the shutdown. Switching from guest mode to host mode is a serializing event.

A RESET signal puts the core into the *reset* phase from any other phase.

There are two idle phases: *halt* and *wait for a startup interprocessor interrupt (SIPI)*. A HLT instruction moves the core to the *halt* phase. The core remains in this phase until an external event occurs. The *wait for SIPI* phase is activated for non-bootstrap processors after the *reset* and the *init* phases. When a multi-processor machine is booting, one processor becomes the bootstrap processor, this processor initializes data structures and devices. Other processors wait for a startup interrupt from the bootstrap processor.

Thus, we have the following phases:

$$\begin{aligned} Phase \triangleq \{\ & BORDER, \\ & DECODE, \\ & PREPARE, EXECUTE, COMPLETE, \\ & VMEXIT, \\ & JISR1, JISR2, \\ & SERIALIZE, INIT, RESET, WAIT\text{-}SIPI, \\ & HALT, SHUTDOWN\}. \end{aligned}$$

In the subsequent sections, we discuss each phase in details.

9.3 Instruction border

In this phase the core collects interrupts from the local APIC and selects the highest priority event according to Table 9.1. Depending on the selected event, the following transitions are possible (Figure 9.3):

- if the event is empty, then start the decode phase.
- if the core is in guest mode and the event is intercepted, then start the vmexit phase.

priority	event
1	injected event
2	INIT
3	trap exception
4	NMI
5	interrupt
6	virtual interrupt
7	fault exception

Table 9.1: Event priority

- if the event is untercepted INIT, then start the init phase.
- otherwise, start the jump to the service routine phase.

We define the *top-event* function, which formalizes Table 9.1.

$$
\begin{aligned}
&\mathit{top\text{-}event}(\mathit{core} \in \mathit{Core}, \mathit{apic} \in \mathit{APIC}) \in (\mathit{Event} \cup \{\epsilon\}) \triangleq \\
&\quad \textbf{if } \mathit{injected\text{-}event}(\mathit{core}) \neq \epsilon \wedge \mathit{core.vtmode} = \mathit{GUEST\text{-}MODE} \\
&\qquad \textbf{then } \mathit{injected\text{-}event}(\mathit{core}) \\
&\quad \textbf{else if } \mathit{init\text{-}pending}(\mathit{apic}) \wedge \mathit{core.GIF} \\
&\qquad \textbf{then } \mathit{Event} \textbf{ with } [\mathit{type} \mapsto \mathit{INIT}, \mathit{injected} \mapsto 0] \\
&\quad \textbf{else if } \mathit{core.trap.valid} \wedge \mathit{core.GIF} \wedge \neg \mathit{core.intr\text{-}shadow} \\
&\qquad \textbf{then } \mathit{Event} \textbf{ with } [\mathit{type} \mapsto \mathit{EXCP}, \mathit{vector} \mapsto \mathit{core.trap.vector}, \\
&\qquad\qquad \mathit{ecode} \mapsto \epsilon, \mathit{injected} \mapsto 0] \\
&\quad \textbf{else if } \mathit{nmi\text{-}pending}(\mathit{apic}) \wedge \mathit{core.GIF} \\
&\qquad \textbf{then } \mathit{Event} \textbf{ with } [\mathit{type} \mapsto \mathit{NMI}, \mathit{vector} \mapsto 2, \mathit{ecode} \mapsto \epsilon, \mathit{injected} \mapsto 0] \\
&\quad \textbf{else if } \mathit{intr\text{-}pending}(\mathit{apic}) \wedge \mathit{core.GIF} \wedge \mathit{core.RFLAGS.IF} \wedge \\
&\qquad \neg \mathit{core.intr\text{-}shadow} \\
&\qquad \textbf{then } \mathit{Event} \textbf{ with } [\mathit{type} \mapsto \mathit{INTR}, \mathit{vector} \mapsto \mathit{top\text{-}pending\text{-}intr}(\mathit{apic}), \\
&\qquad\qquad \mathit{ecode} \mapsto \epsilon, \mathit{injected} \mapsto 0] \\
&\quad \textbf{else if } \mathit{vintr\text{-}pending}(\mathit{core}) \wedge \mathit{core.GIF} \wedge \mathit{core.RFLAGS.IF} \wedge \\
&\qquad \neg \mathit{core.intr\text{-}shadow} \wedge \mathit{core.vtmode} = \mathit{GUEST\text{-}MODE} \\
&\qquad \textbf{then } \mathit{Event} \textbf{ with } [\mathit{type} \mapsto \mathit{VINTR}, \mathit{vector} \mapsto \mathit{vintr\text{-}vector}(\mathit{core}), \\
&\qquad\qquad \mathit{ecode} \mapsto \epsilon, \mathit{injected} \mapsto 0] \\
&\quad \textbf{else if } \mathit{spurious\text{-}pending}(\mathit{apic}) \wedge \mathit{core.GIF} \wedge \mathit{core.RFLAGS.IF} \wedge \\
&\qquad \neg \mathit{core.intr\text{-}shadow} \\
&\qquad \textbf{then } \mathit{Event} \textbf{ with } [\mathit{type} \mapsto \mathit{SPURIOUS}, \mathit{vector} \mapsto (\mathit{apic.SVR.VEC}), \\
&\qquad\qquad \mathit{ecode} \mapsto \epsilon, \mathit{injected} \mapsto 0] \\
&\quad \textbf{else if } \mathit{core.fault.valid} \\
&\qquad \textbf{then } \mathit{Event} \textbf{ with } [\mathit{type} \mapsto \mathit{EXCP}, \mathit{vector} \mapsto \mathit{core.fault.vector}, \\
&\qquad\qquad \mathit{ecode} \mapsto \mathit{core.fault.ecode}, \mathit{injected} \mapsto 0] \\
&\quad \textbf{else } \epsilon.
\end{aligned}
$$

The functions *injected-event*, *vintr-pending*, and *vintr-vector* check the virtual machine control area in the core and return the injected event or the virtual interrupt (see section [Virtual Interrups and Injected Events]). Note that an injected interrupt is not the same as a virtual interrupt. A virtual interrupt has a lower priority and is affected by

the interrupt flags, while an injected interrupt is handled unconditionally.

The remaining functions check the APIC state and are defined in Section 10.

If there is no active event, then the core starts the decode phase. The view buffers need to be flushed because the core accesses memory in the decode phase.

label	$start\text{-}decode(i \in pid)$
guard	$core[i].phase = BORDER$, $top\text{-}event(core[i], apic[i]) = \epsilon$
effect	$core'[i].phase = DECODE$, $core'[i].tlb\text{-}in = empty\text{-}tlb\text{-}in$, $core'[i].mem\text{-}in = empty\text{-}mem\text{-}in$

If there is an event, then we check whether the event is intercepted or not using the predicate: $vmexit\text{-}on\text{-}event(core \in Core, e \in Event, x \in Intercept)$.

The predicate holds if and only if the core is in guest mode, the event intercept bit in the virtual machine control area is set and the x is an intercept of the event e.

label	$intercept\text{-}event(i \in pid, e \in Event, x \in Intercept)$
guard	$core[i].phase = BORDER$, $e = top\text{-}event(core[i], apic[i])$, $vmexit\text{-}on\text{-}event(core[i], e, x)$
effect	$core'[i].phase = VMEXIT$, $core'[i].intercept = x$, $core'[i].fault.valid = 0$, $core'[i].trap.valid = 0$, $core'[i].tlb\text{-}in = empty\text{-}tlb\text{-}in$, $core'[i].mem\text{-}in = empty\text{-}mem\text{-}in$

If the top event is an INIT signal which is not intercepted, then the core proceeds to the init phase.

label	$start\text{-}init(i \in pid, e \in Event)$
guard	$core[i].phase = BORDER$, $e = top\text{-}event(core[i], apic[i])$, $e.type = INIT$, $\neg\exists x \in Intercept : vmexit\text{-}on\text{-}event(core[i], e, x)$
effect	$core'[i].phase = INIT$

Finally, if the top event is an interrupt or an exception which is not intercepted, the core starts the jump to the interrupt service routine phase. The $jisr\text{-}event$ stores the event e.

label	$start\text{-}jisr(i \in pid, e \in Event)$
guard	$core[i].phase = BORDER,$ $e = top\text{-}event(core[i], apic[i]),$ $e.type \neq INIT \wedge e.type \neq RESET,$ $\neg\exists x \in Intercept : vmexit\text{-}on\text{-}event(core[i], e, x)$
effect	$core'[i].phase = JISR1,$ $core'[i].jisr\text{-}event = e,$ $core'[i].fault.valid = 0,$ $core'[i].trap.valid = 0,$ $core'[i].tlb\text{-}in = empty\text{-}tlb\text{-}in,$ $core'[i].mem\text{-}in = empty\text{-}mem\text{-}in$

9.4 RESET, INIT, HALT

Both the RESET and the INIT signals activate processor initialization. The difference between the signals is that after the INIT signal some parts of the processor do not change: the memory system, some core registers (FPU, multimedia, MTRR), and the local APIC.

We will define the values of the registers after RESET/INIT in chapter 12 , and in this section we use predicates:

$$reset\text{-}core(i \in pid, core \in Core) \in \mathbb{B},$$
$$init\text{-}core(i \in pid, core \in Core) \in \mathbb{B}.$$

The predicates hold if and only if the given $core$ is a valid core configuration after the initialization. We use predicates instead of functions that return a core configuration because some registers have undefined values. Thus, there can be many valid initial configurations. The processor index i is necessary to set the bootstrap bit BSP in the $APIC\text{-}BASE$ register of the core. Only one processor in the machine is the bootstrap processor, other processors are application processors. After initialization, an application processor remains in an idle state and waits for a startup interrupt. The index of the bootstrap processor is implementation dependent, so we abstract it by the $bootstrap(i \in pid) \in \mathbb{B}$ predicate, such that

$$\exists k \in pid : bootstrap(k) \wedge \forall j \in pid : j \neq k \Rightarrow \neg bootstrap(j).$$

Similarly, for initialization of the local APIC registers we use predicates:

$$reset\text{-}apic(i \in pid, apic \in APIC) \in \mathbb{B}.$$

In this case, the processor index i is used to set the $APIC\text{-}ID$ register.

The RESET signal invalidates all cache lines without writing them back:

$$reset\text{-}cache(cache \in Cache) \in \mathbb{B} \triangleq \forall pa \in \mathbb{B}^{pq} : cache.state[pa] = I.$$

The store buffer, the load buffer, and the TLB are all flushed after RESET. Stores in the store buffer are not written to the memory.

$$\begin{aligned}
&\mathit{reset\text{-}sb} \in \mathit{SB} \triangleq \mathit{SB}\ \mathbf{with}\,[\mathit{buffer} \mapsto [], \mathit{data} \mapsto (\lambda x \in \mathbb{B}^{pq} \to 0),\\
&\qquad\qquad\qquad\qquad \mathit{cnt} \mapsto (\lambda x \in \mathbb{B}^{pq} \to 0^8), \mathit{uc} \mapsto 0],\\
&\mathit{reset\text{-}lb} \in \mathit{LB} \triangleq \mathit{LB}\ \mathbf{with}\,[\mathit{wc} \mapsto (\lambda x \in \mathbb{B}^{pq} \to (0,0)),\\
&\qquad\qquad\qquad\qquad \mathit{ib} \mapsto (\lambda x \in \mathbb{B}^{pq} \to (0,0))],\\
&\mathit{reset\text{-}tlb} \in \mathit{TLB} \triangleq \mathit{TLB}\ \mathbf{with}\,[\mathit{walks} \mapsto (\lambda w \in \mathit{Walk} \to 0)].
\end{aligned}$$

Transition to the reset phase may happen at any moment:

label	$\mathit{start\text{-}reset}(i \in \mathit{pid}, e \in \mathit{Event})$
guard	
effect	$\mathit{core}'[i].\mathit{phase} = \mathit{RESET}$

Using the predicates, it is easy to define transitions that reset/init the processor,

label	$\mathit{reset}(i \in \mathit{pid}, \mathit{new\text{-}core} \in \mathit{Core}, \mathit{new\text{-}apic} \in \mathit{Apic}, \mathit{new\text{-}cache} \in \mathit{Cache})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{RESET}$, $\mathit{reset\text{-}core}(i, \mathit{new\text{-}core})$, $\mathit{reset\text{-}apic}(i, \mathit{new\text{-}apic})$, $\mathit{reset\text{-}cache}(\mathit{new\text{-}cache})$
effect	$\mathit{sb}'[i] = \mathit{reset\text{-}sb}$, $\mathit{lb}'[i] = \mathit{reset\text{-}lb}$, $\mathit{tlb}'[i] = \mathit{reset\text{-}tlb}$, $\mathit{core}'[i] = \begin{cases} \mathit{new\text{-}core}\ \mathbf{with}\,[\mathit{phase} \mapsto \mathit{BORDER}] & \text{if } \mathit{bootstrap}(i), \\ \mathit{new\text{-}core}\ \mathbf{with}\,[\mathit{phase} \mapsto \mathit{WAIT\text{-}SIPI}] & \text{otherwise}, \end{cases}$ $\mathit{env}'.\mathit{apic}[i] = \mathit{new\text{-}apic}$, $\mathit{env}'.\mathit{cache}[i] = \mathit{new\text{-}cache}$

label	$\mathit{init}(i \in \mathit{pid}, \mathit{new\text{-}core} \in \mathit{Core})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{INIT}$, $\mathit{init\text{-}core}(i, \mathit{new\text{-}core})$
effect	$\mathit{core}'[i] = \begin{cases} \mathit{new\text{-}core}\ \mathbf{with}\,[\mathit{phase} \mapsto \mathit{BORDER}] & \text{if } \mathit{bootstrap}(i), \\ \mathit{new\text{-}core}\ \mathbf{with}\,[\mathit{phase} \mapsto \mathit{WAIT\text{-}SIPI}] & \text{otherwise}, \end{cases}$ $\mathit{env}'.\mathit{apic}[i] = \mathit{init\text{-}delivered}(\mathit{env}.\mathit{apic}[i])$

After initialization, the bootstrap processor goes to the border phase, while an application processor remains idle until a startup IPI arrives. The startup IPI does not trigger a usual jump to an interrupt service routine. Instead, the vector of the interrupt directly specifies the jump target: the code segment base address is set to $\mathit{vector} \circ 0^{12}$ and the instruction offset is set to 0. As the core is in real mode after initialization, the code segment base address ba and the code segment selector $\mathit{selector}$ are related: $\mathit{ba} = \mathit{selector} \circ 0^4$.

label	*wake-up-from-wait-sipi*$(i \in pid)$
guard	$core[i].phase = WAIT\text{-}SIPI$, *sipi-pending*$(env.apic[i])$
effect	$core'[i].SR[CS].selector = sipi\text{-}vector(env.apic[i]) \circ 0^8$, $core'[i].SR[CS].ba = sipi\text{-}vector(env.apic[i]) \circ 0^{12}$, $core'[i].RIP = 0$, $core'[i].phase = BORDER$, $env'.apic[i] = sipi\text{-}delivered(env.apic[i])$

The code segment register is defined in section 12.3.

The functions *sipi-pending*, *sipi-vector*, *sipi-delivered* are defined in Section 10.2.

There is another idle phase – *halt*. The core goes to this phase after executing the HLT instruction, and remains in this phase until RESET, INIT, or an interrupt occurs. Official manuals do not specify what happens if the core enters the halt phase when the global interrupt flag GIF masks interrupts. We assume that in this case the core remains idle until a RESET signal.

label	*wake-up-from-halt*$(i \in pid)$
guard	$core[i].phase = HALT$, $core[i].GIF$, $(init\text{-}pending(env.apic[i]) \vee$ $nmi\text{-}pending(env.apic[i]) \vee$ $intr\text{-}pending(env.apic[i]) \wedge core[i].RFLAGS.IF)$
effect	$core'[i].phase = BORDER$

9.5 Memory accesses

In the subsequent sections we will describe the decode, the execute, the vmexit, and the jump to interrupt service routine phases. As we have discussed in Section 3.1, we will use functions $decode, execute, jisr, vmexit$ to define the transitions for these phases. Let $xxxx$ denote one of the functions, then we derive the following predicate from the definition of the $xxxx$:

$$data\text{-}req\text{-}xxxx(core \in Core, pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^{64}, mt \in MemType) \in \mathbb{B},$$

which holds until the load buffer transitions (Section 7) fetch enough data into the $mem\text{-}in$ buffer.

With the use of the $data\text{-}req\text{-}xxxx$ predicate, we can define $load\text{-}req$, $code\text{-}req$, and $data\text{-}req$ predicates. The core makes a load request when it needs to fetch an instruction or to load data:

$$load\text{-}req(core \in Core, pa \in \mathbb{B}^{pq}, mask \in Mask, mt \in MemType) \triangleq \\ code\text{-}req(core, pa, mask, mt) \vee data\text{-}req(core, pa, mask, mt).$$

We define an instruction fetch request as a data request of the $decode$ phase:

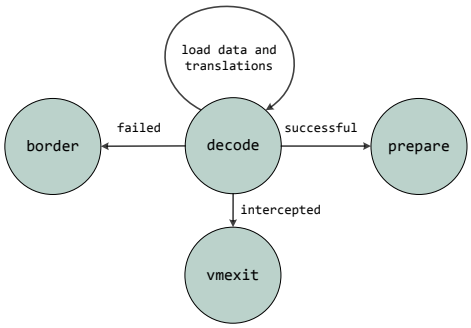

Figure 9.4: Transitions from the decode phase

$$\begin{array}{l}\mathit{code\text{-}req}(\mathit{core} \in \mathit{Core}, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mask} \in \mathit{Mask}, \mathit{mt} \in \mathit{MemType}) \triangleq \\ \quad \textbf{if } \mathit{core.phase} = \mathit{DECODE} \textbf{ then } \mathit{data\text{-}req\text{-}decode}(\mathit{core}, \mathit{pa}, \mathit{mask}, \mathit{mt}, 1) \\ \quad \textbf{else } 0.\end{array}$$

Similarly, we define a data load request:

$$\begin{array}{l}\mathit{data\text{-}req}(\mathit{core} \in \mathit{Core}, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mask} \in \mathit{Mask}, \mathit{mt} \in \mathit{MemType}) \triangleq \\ \quad \textbf{if } \mathit{core.phase} = \mathit{EXECUTE} \textbf{ then } \mathit{data\text{-}req\text{-}execute}(\mathit{core}, \mathit{pa}, \mathit{mask}, \mathit{mt}, 0) \\ \quad \textbf{else if } \mathit{core.phase} = \mathit{VMEXIT} \textbf{ then } \mathit{data\text{-}req\text{-}vmexit}(\mathit{core}, \mathit{pa}, \mathit{mask}, \mathit{mt}, 0) \\ \quad \textbf{else if } \mathit{core.phase} = \mathit{JISR1} \textbf{ then } \mathit{data\text{-}req\text{-}jisr}(\mathit{core}, \mathit{pa}, \mathit{mask}, \mathit{mt}, 0) \\ \quad \textbf{else if } \mathit{core.phase} = \mathit{JISR2} \textbf{ then } \mathit{data\text{-}req\text{-}jisr}(\mathit{core}, \mathit{pa}, \mathit{mask}, \mathit{mt}, 0) \\ \quad \textbf{else } 0.\end{array}$$

The load buffer transitions fetch cacheable and WC data nondeterministically because such loads do not have side effects. Uncacheable data, on the other hand, is loaded only on a core request. Thus, for any physical address, the transitions need to know the corresponding memory type. We define a predicate that checks whether the TLB has a walk that maps to the physical region which contains the given address and has the given the memory type:

$$\begin{array}{l}\mathit{reachable}(\mathit{core} \in \mathit{Core}, \mathit{tlb} \in \mathit{TLB}, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mt} \in \mathit{MemType}, \mathit{code} \in \mathbb{B}) \triangleq \\ \quad \textbf{if } \mathit{paging\text{-}enabled}(\mathit{core}) \textbf{ then} \\ \qquad \exists w \in \mathit{Walk} : \mathit{tlb.walks}[w] \wedge w.\mathit{ba} \leq \mathit{pa} < w.\mathit{ba} + 4096/8 \wedge \\ \qquad\qquad \mathit{mtrr}(\mathit{core}, \mathit{pa}, \mathit{pat\text{-}lookup}(\mathit{core}, w.\mathit{pat\text{-}idx})) = \mathit{mt} \wedge \\ \qquad\qquad w.r.\mathit{code} = \mathit{code} \\ \quad \textbf{else } \mathit{mtrr}(\mathit{core}, \mathit{pa}, \mathit{WB}) = \mathit{mt}.\end{array}$$

The $code$ parameter indicates whether the access is for the instruction fetch or for a data load. The function $pat\text{-}lookup$ looks up the memory type in the PAT register. The function $mtrr$ computes the memory type, using the memory type range registers, and combines it with the PAT memory type. The functions are defined in section 12.8.

9.6 Fetch and decode

In the decode phase, transitions described in Sections 7 and 8 fetch the instruction by filling in the $tlb\text{-}view$ and $mem\text{-}in$ buffers. After the instruction is fetched, three cases

are possible (Figure 9.4):

- decode succeeds. In this case, the *decode* function saves the information about the instruction in the core, increments the instruction pointer by the instruction length, and switches the core to the prepare execution phase.
- decode raises an exception. In this case, the information about the exception is saved in the *fault* field of the core, and the core returns to the border phase.
- decode is intercepted. In this case, the information about the intercept is saved in the *intercept* field of the core, and the core starts the vmexit phase.

Formalizing the cases, we get the corresponding three transitions:

label	$\mathit{successful\text{-}decode}(i \in \mathit{pid})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{DECODE}$, $\mathit{can\text{-}decode}(\mathit{core}[i])$
effect	$\mathit{core}'[i] = \mathit{decode}(\mathit{core}[i])$ **with** $[\mathit{phase} \mapsto \mathit{PREPARE}]$

Recall, that $\mathit{can\text{-}decode}(\mathit{core}[i])$ holds if and only if the instruction was fetched into the $\mathit{mem\text{-}in}$ buffer and the instruction can be successfully decoded.

label	$\mathit{failed\text{-}decode}(i \in \mathit{pid}, e \in \mathit{Exception})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{DECODE}$, $\mathit{fault\text{-}on\text{-}decode}(\mathit{core}[i], e)$
effect	$\mathit{core}'[i].\mathit{fault} = e$, $\mathit{core}'[i].\mathit{phase} = \mathit{BORDER}$

label	$\mathit{intercepted\text{-}decode}(i \in \mathit{pid}, x \in \mathit{Intercept})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{DECODE}$, $\mathit{vmexit\text{-}on\text{-}decode}(\mathit{core}[i], x)$
effect	$\mathit{core}'[i].\mathit{intercept} = x$, $\mathit{core}'[i].\mathit{phase} = \mathit{VMEXIT}$, $\mathit{core}'[i].\mathit{tlb\text{-}in} = \mathit{empty\text{-}tlb\text{-}in}$, $\mathit{core}'[i].\mathit{mem\text{-}in} = \mathit{empty\text{-}mem\text{-}in}$

In the latter case, we flush the view buffers because the core accesses the memory in the vmexit phase.

9.7 Execution

The instruction is executed in three phases (Figure 9.5):

1. the prepare execution phase takes care of acquiring the memory lock for atomic instructions, flushing load/store buffers for atomic, I/O, and serializing instructions, and flushing the view buffers;
2. the execute phase executes the instructions with the help of the *execute* function, this phase can fail either because of an exception or because of an intercept.
3. the complete execution phase moves stores from the *mem-out* buffer to the store buffer and carries out commands to other units.

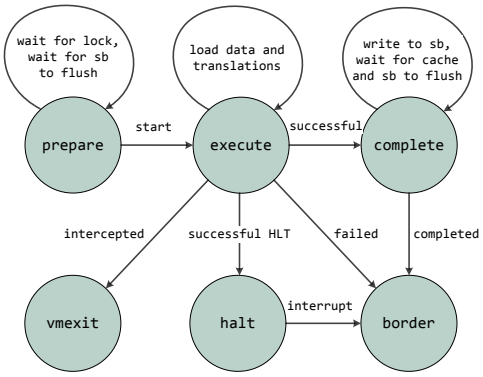

Figure 9.5: Instruction execution

In the prepare execution phase, the core waits until the memory lock becomes free if the current instruction is atomic. Since atomic, I/O, and serializing instructions guarantee that all stores of previous instructions are written to the memory before any memory access of the current instruction, the core waits until the store buffer becomes empty. Once the conditions are satisfied, the core acquires the memory lock for an atomic instruction, flushes the load buffer for atomic, I/O, and serializing instructions, flushes the view buffers, and proceeds to the execute phase. If we would not flush the load buffer, then a load access of an atomic, I/O, or serializing instruction could overpass a store access of an older instruction. The official manuals state that the lower four bits of the $CR8$ control register are aliased to the bits $[7:4]$ of the task priority register in the APIC. Therefore, we need to synchronize these bits before starting instruction execution.

label	$start\text{-}execution(i \in pid)$
guard	$core[i].phase = PREPARE,$ $core[i].prefix.lock \Rightarrow env.lock = \epsilon,$ $core[i].prefix.lock \Rightarrow length(sb[i].buffer) = 0,$ $io(core[i]) \Rightarrow length(sb[i].buffer) = 0,$ $serializing(core[i]) \Rightarrow length(sb[i].buffer) = 0$
effect	$core'[i].phase = EXECUTE,$ $core'[i].tlb\text{-}in = empty\text{-}tlb\text{-}in,$ $core'[i].mem\text{-}in = empty\text{-}mem\text{-}in,$ $core'[i].CR8[3:0] = env.apic[i].TPR[7:4],$ $x = core[i].prefix.lock \vee io(core[i]) \vee serializing(core[i]),$ $lb'[i] = \begin{cases} flush\text{-}wc(lb[i]) & \text{if } x, \\ lb[i] & \text{otherwise,} \end{cases}$ $env'.lock = \begin{cases} i \text{ if } core[i].prefix.lock, \\ env.lock \text{ otherwise} \end{cases}$

The predicates io and $serializing$ check whether the current instruction is an I/O instruction or a serializing instruction (see section [Opcode Table]).

In the execute phase, the core waits until load buffer and TLB transitions fill in the $mem\text{-}in$ and the $tlb\text{-}in$ buffers. Once there is enough data to execute the instruction, four cases are possible:

- instruction execution succeeds and the instruction is HLT;
- instruction execution succeeds and the instruction is not HLT;
- an exception is raised;
- the instruction is intercepted.

We define four transitions corresponding to these cases. The HLT instruction puts the core into an idle state, where the core waits for an external signal or an interrupt:

label	$successful\text{-}hlt\text{-}execution(i \in pid)$
guard	$core[i].phase = EXECUTE$, $can\text{-}execute(core[i])$, $hlt(core[i])$
effect	$core'[i] = execute(core[i])$ **with** $[phase \mapsto HALT]$

The hlt predicate compares the current instruction opcode with the opcode of the HLT instruction. Note that there is no need to release the memory lock in this case because the HLT instruction cannot have a lock prefix.

After executing a non-HLT instruction, the core proceeds to the complete execution phase. As the main complexity is hidden inside the $execute$ function, the transition is trivial, except for the interrupt shadow handling:

label	$successful\text{-}execution(i \in pid)$
guard	$core[i].phase = EXECUTE$, $can\text{-}execute(core[i])$, $\neg hlt(core[i])$
effect	$x = execute(core[i])$ **with** $[phase \mapsto COMPLETE]$, $y = x$ **with** $[intr\text{-}shadow \mapsto x.intr\text{-}shadow \wedge \neg core[i].intr\text{-}shadow]$, $core'[i] = y$

Interrupt shadow does not last more than one instruction. Thus, if there are two consecutive instructions with interrupt shadows, the shadow of the second instruction is removed.

In case of an exception, the core saves the exception information in the $fault$ field and returns to the border phase. The memory lock is released if it has been acquired before.

label	$failed\text{-}execution(i \in pid, e \in Exception)$
guard	$core[i].phase = EXECUTE$, $fault\text{-}on\text{-}execute(core[i], e)$
effect	$env'.lock = \begin{cases} \epsilon & \text{if } env.lock = i, \\ env.lock & \text{otherwise,} \end{cases}$ $core'[i].fault = e$, $core'[i].phase = BORDER$

If the instruction triggers an intercept in guest mode, then the core goes to the vmexit phase. The memory lock is released if it has been acquired before.

label	$\mathit{intercepted\text{-}execution}(i \in \mathit{pid}, x \in \mathit{Intercept})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{EXECUTE}$, $\mathit{vmexit\text{-}on\text{-}execute}(\mathit{core}[i], x)$
effect	$\mathit{env}'.\mathit{lock} = \begin{cases} \epsilon & \text{if } \mathit{env}.\mathit{lock} = i, \\ \mathit{env}.\mathit{lock} & \text{otherwise} \end{cases}$ $\mathit{core}'[i].\mathit{intercept} = x$, $\mathit{core}'[i].\mathit{phase} = \mathit{VMEXIT}$, $\mathit{core}'[i].\mathit{tlb\text{-}in} = \mathit{empty\text{-}tlb\text{-}in}$, $\mathit{core}'[i].\mathit{mem\text{-}in} = \mathit{empty\text{-}mem\text{-}in}$

In the complete execution phase, the core transfers stores from the $\mathit{mem\text{-}out}$ buffer to the store buffer. Since loads cannot overpass an old UC store, a UC store flushes the load buffer.

label	$\mathit{write\text{-}to\text{-}sb}(i \in \mathit{pid}, s \in \mathit{Store}, \mathit{req} \in \mathit{WriteReq})$
guard	$\mathit{core}[i].\mathit{phase} \in \{\mathit{COMPLETE}, \mathit{SERIALIZE}\}$, $s.\mathit{mask} \neq 0$, $\mathit{req}.\mathit{addr} = s.\mathit{pa}$, $\mathit{req}.\mathit{mt} = s.\mathit{mt}$, $\mathit{req}.\mathit{mask} = s.\mathit{mask}$, $\mathit{req}.\mathit{data} = s.\mathit{data}$, $\mathit{core}[i].\mathit{mem\text{-}out}[\mathit{req}]$
effect	$\mathit{sb}'[i] = \mathit{write}(\mathit{sb}[i], s)$, $\mathit{core}'[i].\mathit{mem\text{-}out}[\mathit{req}] = 0$, $\mathit{lb}'[i] = \begin{cases} \mathit{flush\text{-}wc}(\mathit{lb}[i]) & \text{if } s.\mathit{mt} = \mathit{UC}, \\ \mathit{lb}[i] & \text{otherwise} \end{cases}$

After all stores are transferred, the core completes the instruction execution by carrying out commands to other units. Atomic, I/O, serializing, and MFENCE instructions guarantee that all stores are written to the memory before a load or a store access of the next instruction. Therefore, the core waits until the store buffer is empty.

In case of the WBINVD instruction, the core waits until the cache drops all the lines. The CLFLUSH instruction writes back and invalidates a specific cache line using the $\mathit{wb\text{-}flush\text{-}line}$ command of the core, which makes the core wait until the line becomes invalid.

Once all the conditions are satisfied, the following transition takes place:

label	$complete\text{-}execution(i \in pid)$
guard	$core[i].phase = COMPLETE$, $core[i].mem\text{-}out = empty\text{-}mem\text{-}out$, $core[i].prefix.lock \Rightarrow length(sb[i].buffer) = 0$, $io(core[i]) \Rightarrow length(sb[i].buffer) = 0$, $serializing(core[i]) \Rightarrow length(sb[i].buffer) = 0$, $mfence(core[i]) \Rightarrow length(sb[i].buffer) = 0$, $core[i].invd.valid \wedge core[i].invd.wb \Rightarrow \forall pa \in \mathbb{B}^{pq} : cache[i].state[pa] = I$, $core[i].flush\text{-}line.valid \neq \epsilon \Rightarrow cache[i].state[core[i].flush\text{-}line.addr] = I$
effect	$cache'[i] = \begin{cases} invalidate(cache[i]) & \text{if } core[i].invd.valid \wedge \neg core[i].invd.wb, \\ cache[i] & \text{otherwise}, \end{cases}$ $x = core[i].prefix.lock \vee io(core[i]) \vee mfence(core[i]) \vee lfence(core[i])$, $lb'[i] = \begin{cases} flush\text{-}wc(lb[i]) & \text{if } x, \\ flush\text{-}all(lb[i]) & \text{if } serializing(core[i]), \\ lb[i] & \text{otherwise}, \end{cases}$ $flush\text{-}tlb = core[i].flush\text{-}tlb.valid$ $asid = current\text{-}asid(core[i])$ $with\text{-}asid = core[i].flush\text{-}tlb.with\text{-}asid$ $local = core[i].flush\text{-}tlb.local$ $invlpg\text{-}addr = core[i].invlpg.addr$ $tlb'[i] = \begin{cases} flush\text{-}global(tlb[i]) & \text{if } flush\text{-}tlb \wedge \neg local \wedge \neg with\text{-}asid, \\ flush\text{-}local(tlb[i]) & \text{if } flush\text{-}tlb \wedge local \wedge \neg with\text{-}asid, \\ flush\text{-}global\text{-}with\text{-}asid(tlb[i], asid) & \text{if } flush\text{-}tlb \wedge \neg local \wedge with\text{-}asid, \\ flush\text{-}local\text{-}with\text{-}asid(tlb[i], asid) & \text{if } flush\text{-}tlb \wedge local \wedge with\text{-}asid, \\ invlpg(tlb[i], invlpg\text{-}addr, asid) & \text{if } core[i].invlpg.valid, \\ tlb[i] & \text{otherwise}, \end{cases}$ $sb'[i] = \begin{cases} write(sb[i], SFENCE) & \text{if } sfence(core[i]), \\ sb[i] & \text{otherwise}, \end{cases}$ $env'.lock = \begin{cases} \epsilon & \text{if } env.lock = i, \\ env.lock & \text{otherwise}, \end{cases}$ $env'.apic[i].TPR[7:0] = \begin{cases} core[i].CR8[3:0] \circ 0^4 & \text{if } core[i].sync\text{-}tpr, \\ env.apic[i].TPR[7:0] & \text{otherwise}, \end{cases}$ $core'[i].phase = BORDER$

The predicates io, $wbinvd$, $invd$, $mfence$, $lfence$, $sfence$ check if the current opcode matches the opcode of the corresponding instruction (see section [Opcode Table]).

The cache is invalidated without writing the dirty lines back if the current instruction is INVD. Atomic, I/O, MFENCE, and LFENCE instructions flush the load buffer because loads cannot be reordered before these instructions. A serializing instruction flushes both the instruction and the data load buffers because it guarantees that all memory accesses of the instruction complete before the next instruction is fetched. The TLB flush commands update the TLB accordingly. SFENCE instruction enqueues a store

fence into the store buffer. The memory lock is released if it has been acquired before. In case $CR8$ was written, the bits $[7:4]$ of the TPR reflect it and the bits $[3:0]$ are cleared. After this transition, the instruction execution completes, and the core returns to the border phase.

9.8 VMEXIT

In the vmexit phase, the core switches from guest mode to host mode. Transitions from Section 7 load the host registers into the $mem\text{-}in$ buffer. If the switch succeeds then the core proceeds to the serialize phase, where the guest registers are copied from the $mem\text{-}out$ buffer to the store buffer and then to the memory. Official manuals specify that VMEXIT is a serializing event, however, it is unclear at which point serialization takes place. Either the load/store buffer are flushed before VMEXIT, or after it, or at both points. We have chosen to serialize afterwards as it makes the model simpler, but this might be unsound.

label	$\mathit{successful\text{-}vmexit}(i \in \mathit{pid})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{VMEXIT}$, $\mathit{can\text{-}vmexit}(\mathit{core}[i])$
effect	$\mathit{core}'[i] = \mathit{vmexit}(\mathit{core}[i])$ **with** $[\mathit{phase} \mapsto \mathit{SERIALIZE}]$

If the switch fails, then the core shuts down.

label	$\mathit{failed\text{-}vmexit}(i \in \mathit{pid}, e \in \mathit{Exception})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{VMEXIT}$, $\mathit{fault\text{-}on\text{-}vmexit}(\mathit{core}[i], e)$
effect	$\mathit{core}'[i].\mathit{phase} = \mathit{SHUTDOWN}$

9.9 Serializing

In the serialize phase, the core waits until all the stores in the $mem\text{-}out$ buffer are moved to the store buffer, and until all stores in the store buffer are committed to the memory. After that, the load buffer for instructions and WC data is flushed and the core returns to the border phase.

label	$\mathit{serialize}(i \in \mathit{pid})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{SERIALIZE}$, $\mathit{core}[i].\mathit{mem\text{-}out} = \mathit{empty\text{-}mem\text{-}out}$, $\mathit{length}(\mathit{sb}[i].\mathit{buffer}) = 0$
effect	$\mathit{core}'[i].\mathit{phase} = \mathit{BORDER}$ $\mathit{lb}'[i] = \mathit{flush\text{-}serializing}(\mathit{lb}[i])$

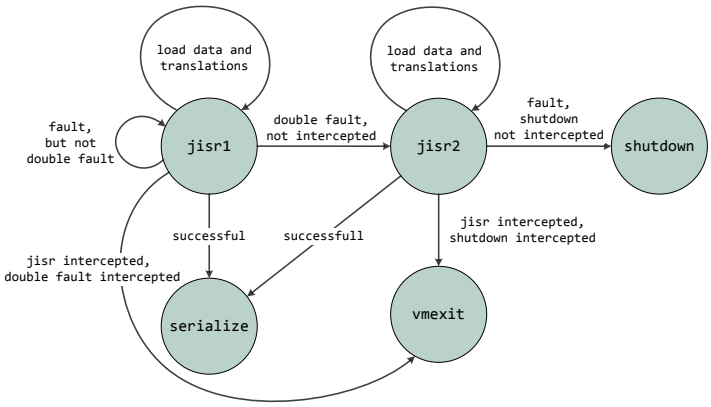

Figure 9.6: Jump to interrupt service routine

9.10 Jump to interrupt service routine

The function *jisr* performs the jump to the interrupt service routine for the *jisr-event*. The function looks up the jump target in the interrupt descriptor table using the event vector as an index. Since the jump can raise an exception, for which another jump must be performed, we have two phases: *JISR1* and *JISR2*. Each jump can succeed, fail with an exception, or be intercepted. As a result, we have the following cases (Figure 9.6):

- the first jump succeeds: the core completes the jump in the serialize phase.
- the first jump is intercepted: the core starts the vmexit phase.
- the first jump raises an exception: certain combinations of the first event and the raised exception trigger a double fault exception. Since the double fault exception might be intercepted, we have three cases:
  - a double fault exception is not triggered: the core starts a jump for the raised exception in the *JISR1* phase. Note that the new jump is considered as the first jump (i.e. the jump counter is reset).
  - a double fault exception is triggered but is not intercepted: the core starts a jump to the double fault exception handler in the *JISR2* phase.
  - a double fault exception is triggered and intercepted: the core starts the vmexit phase.
- the second jump succeeds: the core completes the jump in the serialize phase.
- the second jump is intercepted: the core starts the vmexit phase.
- the second jump raises an exception and the shutdown is not intercepted: the core shuts down.
- the second jump raises an exception and the shutdown is intercepted: the core starts the vmexit phase.

Transitions described in Sections 7 and 8 provide necessary data for a jump by filling in the *tlb-in* and *mem-in* buffers. If the first jump succeeds, then the core marks the event as delivered and proceeds to the serialize phase:

label	*successful-jisr1*$(i \in pid)$
guard	$core[i].phase = JISR1$, *can-jisr*$(core[i])$
effect	$c = jisr(core[i])$ **with** $[phase \mapsto SERIALIZE, jisr\text{-}event \mapsto \epsilon]$, $e = core[i].jisr\text{-}event$, $core'[i] = \begin{cases} vintr\text{-}delivered(c) & \text{if } e.type = VINTR, \\ injected\text{-}delivered(c) & \text{if } e.injected \wedge e.type \neq VINTR, \\ c & \text{otherwise}, \end{cases}$ $real = \neg e.injected \wedge e.type \neq VINTR$, $a = env.apic[i]$, $env'.apic[i] = \begin{cases} nmi\text{-}delivered(a) & \text{if } e.type = NMI \wedge real, \\ intr\text{-}delivered(a, e.vector) & \text{if } e.type = INTR \wedge real, \\ spurious\text{-}delivered(a) & \text{if } e.type = SPURIOUS, \\ a & \text{otherwise} \end{cases}$

Recall that the *jisr-event* register was defined in the beginning of this chapter and it stores information about the event that triggered the jump.

The functions $[nmi, intr, spurious]$-*delivered* update the local APIC to reflect the delivery of the interrupt (Sections 10). Similarly, the functions *vintr-delivered* and *injected-delivered* update the core to reflect the delivery of the virtual interrupt or of the injected event (see section [Injected Events and Virtual Interrups] of chapter [Virtualization]).

If the jump is intercepted, then the core starts the vmexit phase:

label	*intercepted-jisr1*$(i \in pid, x \in Intercept)$
guard	$core[i].phase = JISR1$, *vmexit-on-jisr*$(core[i], x)$
effect	$core'[i].intercept = x$, $core'[i].phase = VMEXIT$, $core'[i].jisr\text{-}event = \epsilon$, $core'[i].tlb\text{-}in = empty\text{-}tlb\text{-}in$, $core'[i].mem\text{-}in = empty\text{-}mem\text{-}in$

If the jump fails with an exception, we need to check for double fault conditions. A double fault occurs when a page fault is followed by another page fault or a so-called contributory exception. It also occurs if a contributory exception is followed by another contributory exception:

$$
\begin{aligned}
&double\text{-}fault\text{-}trigger(event1 \in Event, vector2 \in \mathbb{B}^8) \in \mathbb{B} \triangleq \\
&\quad event1.type = EXCP \\
&\quad \wedge ((event1.vector = PF \wedge vector2 = PF) \\
&\qquad \vee (event1.vector = PF \wedge contributory(vector2)) \\
&\qquad \vee (contributory(event1.vector) \wedge contributory(vector2))).
\end{aligned}
$$

The following exceptions are contributory exceptions: division by zero, task state segment exception, segment-not-present exception, stack segment exception, and general-protection exception:

$$contributory(vector \in \mathbb{B}^8) \in \mathbb{B} \triangleq vector \in \{DE, TS, NP, SS, GP\},$$

where $DE = 0$, $TS = 10$, $NP = 11$, $SS = 12$, $GP = 13$.

In case a double fault is triggered and is not intercepted, then the core sets *jisr-event* to the double fault exception, flushes the view buffers, and proceeds to the *JISR2* phase:

label	*double-fault-on-jisr*$(i \in pid, e \in Exception)$
guard	$core[i].phase = JISR1$, $fault\text{-}on\text{-}jisr(core[i], e)$, $double\text{-}fault\text{-}trigger(core[i].jisr\text{-}event, e.vector)$, $\neg\exists x \in Intercept : vmexit\text{-}on\text{-}double\text{-}fault(core[i], x)$
effect	$core'[i].jisr\text{-}event = Event$ **with** $[type \mapsto EXCP, vector \mapsto DF,$ $ecode \mapsto 0, injected \mapsto 0]$, $core'[i].phase = JISR2$, $core'[i].tlb\text{-}in = empty\text{-}tlb\text{-}in$, $core'[i].mem\text{-}in = empty\text{-}mem\text{-}in$

In case the double fault is intercepted, the core starts the vmexit phase:

label	*double-fault-intercepted*$(i \in pid, e \in Exception, x \in Intercept)$
guard	$core[i].phase = JISR1$, $fault\text{-}on\text{-}jisr(core[i], e)$, $double\text{-}fault\text{-}trigger(core[i].jisr\text{-}event, e.vector)$, $vmexit\text{-}on\text{-}double\text{-}fault(core[i], x)$
effect	$core'[i].intercept = x$, $core'[i].phase = VMEXIT$, $core'[i].jisr\text{-}event = \epsilon$, $core'[i].tlb\text{-}in = empty\text{-}tlb\text{-}in$, $core'[i].mem\text{-}in = empty\text{-}mem\text{-}in$

Otherwise, the core restarts the *JISR1* phase for the raised exception:

label	*failed-jisr1*$(i \in pid, e \in Exception)$
guard	$core[i].phase = JISR1$, $fault\text{-}on\text{-}jisr(core[i], e)$, $\neg double\text{-}fault\text{-}trigger(core[i].jisr\text{-}event, e.vector)$
effect	$core'[i].jisr\text{-}event = Event$ **with** $[type \mapsto EXCP, vector \mapsto e.vector,$ $ecode \mapsto e.ecode, injected \mapsto 0]$, $core'[i].phase = JISR1$, $core'[i].tlb\text{-}in = empty\text{-}tlb\text{-}in$, $core'[i].mem\text{-}in = empty\text{-}mem\text{-}in$

If the jump to the double fault handler succeeds, then the core completes the jump in the serialize phase:

label	$\mathit{successful\text{-}jisr2}(i \in \mathit{pid})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{JISR2}$, $\mathit{can\text{-}jisr}(\mathit{core}[i])$
effect	$\mathit{core}'[i] = \mathit{jisr}(\mathit{core}[i])$ **with** $[\mathit{phase} \mapsto \mathit{SERIALIZE}, \mathit{jisr\text{-}event} \mapsto \epsilon]$

If the jump is intercepted, then the core starts the vmexit phase:

label	$\mathit{intercepted\text{-}jisr2}(i \in \mathit{pid}, x \in \mathit{Intercept})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{JISR2}$, $\mathit{vmexit\text{-}on\text{-}jisr}(\mathit{core}[i], x)$
effect	$\mathit{core}'[i].\mathit{intercept} = x$, $\mathit{core}'[i].\mathit{phase} = \mathit{VMEXIT}$, $\mathit{core}'[i].\mathit{jisr\text{-}event} = \epsilon$, $\mathit{core}'[i].\mathit{tlb\text{-}in} = \mathit{empty\text{-}tlb\text{-}in}$, $\mathit{core}'[i].\mathit{mem\text{-}in} = \mathit{empty\text{-}mem\text{-}in}$

If the second jump raises an exception and the shutdown is not intercepted, then the core shuts down:

label	$\mathit{shutdown}(i \in \mathit{pid}, e \in \mathit{Exception})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{JISR2}$, $\mathit{fault\text{-}on\text{-}jisr}(\mathit{core}[i], e)$, $\neg\exists x \in \mathit{Intercept} : \mathit{vmexit\text{-}on\text{-}shutdown}(\mathit{core}[i], x)$
effect	$\mathit{core}'[i].\mathit{phase} = \mathit{SHUTDOWN}$

Otherwise, the core starts the vmexit phase:

label	$\mathit{shutdown\text{-}intercepted}(i \in \mathit{pid}, e \in \mathit{Exception}, x \in \mathit{Intercept})$
guard	$\mathit{core}[i].\mathit{phase} = \mathit{JISR2}$, $\mathit{fault\text{-}on\text{-}jisr}(\mathit{core}[i], e)$, $\mathit{vmexit\text{-}on\text{-}shutdown}(\mathit{core}[i], x)$
effect	$\mathit{core}'[i].\mathit{intercept} = x$, $\mathit{core}'[i].\mathit{phase} = \mathit{VMEXIT}$, $\mathit{core}'[i].\mathit{jisr\text{-}event} = \epsilon$, $\mathit{core}'[i].\mathit{tlb\text{-}in} = \mathit{empty\text{-}tlb\text{-}in}$, $\mathit{core}'[i].\mathit{mem\text{-}in} = \mathit{empty\text{-}mem\text{-}in}$

CHAPTER

TEN

LOCAL APIC

Each processor has a local advanced programmable interrupt controller (APIC), which collects interrupts and forwards them to the processor. The local APIC also allows the processor to send interprocessor interrupts.

The APIC registers control how interrupts are managed. The registers are memory-mapped, so the processor sets them up by writing to the APIC page. The $APIC\text{-}BASE$ register of the processor core contains the physical base address of the APIC page:

$$apic\text{-}base(core \in Core) \triangleq core.APIC\text{-}BASE.ABA \circ 0^{12}.$$

The AE bit in the $APIC\text{-}BASE$ register enables and disables the local APIC:

$$apic\text{-}enabled(core \in Core) \triangleq core.APIC\text{-}BASE.AE.$$

Read and write memory accesses are forwarded to the APIC only when they are in the APIC page and the APIC is enabled:

$$in\text{-}apic\text{-}range(core \in Core, pa \in \mathbb{B}^{pq}) \in \mathbb{B} \triangleq$$
$$apic\text{-}base(core) \leq pa \circ 0^3 < apic\text{-}base(core) + 2^{12} \wedge apic\text{-}enabled(core).$$

The APIC registers are 32 bits wide and are aligned on 16-byte boundaries. Unaligned accesses and accesses to non-existent registers do not affect the APIC state except for setting the 'illegal register address' bit in the error status register. A load access returns undefined data in such cases.

The following kinds of interrupts exist:

- a maskable interrupt (INTR): it has a $vector \in \mathbb{B}^8$ and a $trigger\text{-}mode \in \mathbb{B}$ attributes. The purpose of the vector is two-fold. First, it is used by the processor core as an index into the interrupt descriptor table to locate the interrupt handler. Second, the APIC uses the vector as a priority indicator. The APIC latches maskable interrupts in the interrupt request register (IRR) and sends them one by one to the processor core starting from the highest priority interrupt. The APIC is, in some sense, a priority queue for maskable interrupts.
  The trigger mode indicates how the the interrupt was signalled: by an active level of the interrupt line or by a falling/raising edge. In the former case, the in-

terrupt is level-triggered and the source device holds the line at the active level until it receives an end-of-interrupt (EOI) message from the APIC. In the latter case, the interrupt is edge-triggered and the device does not expect an EOI message. The APIC sends an EOI message only when the interrupt handler writes to the EOI register and the interrupt being serviced is level-triggered. The APIC stores the trigger mode for each pending maskable interrupt in the TMR register ($trigger\text{-}mode = 1$ means 'level-triggered').
The processor can mask maskable interrupts by clearing the $RFLAGS.IF$ bit or by setting a priority threshold in the task priority register (TPR).

- a non-maskable interrupt (NMI): its attributes are fixed, $vector = 2$ and $trigger\text{-}mode = 0$. When an NMI arrives, the APIC sends it directly to the processor core in case the core is not already handling another NMI. In order to keep track of pending NMI interrupts, we add two registers to the APIC configuration: $nmirr \in \mathbb{B}$ and $nmisr \in \mathbb{B}$. The latter register indicates whether the core is handling an NMI (an NMI is in service). The former register latches an NMI if another NMI is already in service.
- a startup interprocessor interrupt (SIPI): the $vector$ attribute specifies a jump target as described in Section 9.4. The trigger mode is irrelevant. An incoming SIPI is latched in the $sipirr \in \mathbb{B}$ register, and the $sipivec \in \mathbb{B}^8$ stores the vector.
- an initialization interrupt (INIT): the vector and trigger mode attributes are irrelevant. The core starts initialization upon receiving this interrupt. The $initrr \in \mathbb{B}$ register latches an incoming INIT.
- a system management interrupt (SMI): it switches the core to system management mode (SMM). Since we do not model SMM, we leave SMI out.
- an external interrupt (ExtInt): it is a legacy interrupt, we do not model it.

We divide the APIC registers into five categories: INTR, IPI, local vector table (LVT), miscellaneous, auxiliary. Table 10.1 lists the registers from the first four categories. The auxiliary registers are not present in the official manuals, but we need them to define the APIC transitions.

The APIC configuration is the following record:

$$\begin{aligned} APIC \triangleq [&\text{ registers from Table 10.1: } \ldots \\ &shadowESR \in ErrorStatusReg, \\ &initrr \in \mathbb{B}, \\ &nmirr \in \mathbb{B}, nmisr \in \mathbb{B}, \\ &sipirr \in \mathbb{B}, sipivec \in \mathbb{B}^8, \\ &spuriousrr \in \mathbb{B}, \\ &undef \in \mathbb{B}^{64}]. \end{aligned}$$

The subsequent sections describe each register category in details.

10.1 Maskable interrupts

The APIC stores information about maskable interrupts in three bitmaps. Each bitmap is 256 bits wide. Thus, for each interrupt vector, the APIC stores three bits. Due to the 32 bits limit on the register width, each bitmap is represented by eight registers:

Offset	Register	Description	Type
020h	$APIC\text{-}ID \in APIC\text{-}ID$	APIC ID	IPI
030h	$VERSION \in VERSION$	APIC Version	Misc
080h	$TPR \in PriorityReg$	Task Priority	INTR
090h	$APR \in PriorityReg$	Arbitration Priority	IPI
0A0h	$PPR \in PriorityReg$	Processor Priority	INTR
0B0h	$EOI \in \mathbb{B}^{32}$	End Of Interrupt	INTR
0C0h	$RRR \in \mathbb{B}^{32}$	Remote Read	IPI
0D0h	$LDR \in LDR$	Logical Destination	IPI
0E0h	$DFR \in DFR$	Destination Format	IPI
0F0h	$SVR \in SVR$	Spurious Interrupt	Misc
100h + 10h $* i$	$ISR[i] \in \mathbb{B}^{32}$	In-Service	INTR
180h + 10h $* i$	$TMR[i] \in \mathbb{B}^{32}$	Trigger Mode	INTR
200h + 10h $* i$	$IRR[i] \in \mathbb{B}^{32}$	Interrupt Request	INTR
280h	$ESR \in ESR$	Error Status	Misc
300h	$ICRL \in ICRL$	Interrupt Command Low	IPI
310h	$ICRH \in ICRH$	Interrupt Command High	IPI
320h	$TIMER \in LvtReg$	Timer	LVT
330h	$THERMAL \in LvtReg$	Thermal	LVT
340h	$PERF \in LvtReg$	Performance	LVT
350h	$LINT0 \in LvtReg$	Local Interrupt 0	LVT
360h	$LINT1 \in LvtReg$	Local Interrupt 1	LVT
370h	$ERROR \in LvtReg$	Internal Error	LVT
380h	$TICR \in \mathbb{B}^{32}$	Timer Initial Count	LVT
390h	$TCCR \in \mathbb{B}^{32}$	Timer Current Count	LVT
3E0h	$TDCR \in DivConfReg$	Timer Divide Configuration	LVT

Table 10.1: Local APIC Registers

- interrupt request registers: $IRR \in \mathbb{B}^3 \to \mathbb{B}^{32}$;
- in-service registers: $ISR \in \mathbb{B}^3 \to \mathbb{B}^{32}$;
- trigger mode registers: $TMR \in \mathbb{B}^3 \to \mathbb{B}^{32}$.

An accepted maskable interrupt with the vector v and the trigger-mode tm is stored in the IRR and the TMR:

$$
\begin{array}{l}
\mathit{accept\text{-}intr}(\mathit{apic} \in \mathit{APIC}, v \in \mathbb{B}^8, \mathit{tm} \in \mathbb{B}) \in \mathit{APIC} \triangleq \\
\quad \textbf{if } v < 32 \textbf{ then } \mathit{receive\text{-}illegal\text{-}vector}(\mathit{apic}) \\
\quad \textbf{else} \\
\quad\quad \textbf{let } i = v/32 \textbf{ in} \\
\quad\quad \textbf{let } j = v \bmod 32 \textbf{ in} \\
\quad\quad \mathit{update\text{-}APR}(\mathit{apic} \textbf{ with } [\mathit{IRR}[i][j] \mapsto 1, \mathit{TMR}[i][j] \mapsto \mathit{tm}]),
\end{array}
$$

where the *receive-illegal-vector* function records the error in the error status register, and the *update-APR* function updates the arbitration priority to reflect the new interrupt. We will discuss these functions later.

An incoming maskable interrupt is rejected if the corresponding bit in the IRR is already set:

$$
\begin{array}{l}
\mathit{can\text{-}accept\text{-}intr}(\mathit{apic} \in \mathit{APIC}, v \in \mathbb{B}^8, \mathit{tm} \in \mathbb{B}) \in \mathbb{B} \triangleq \\
\quad \neg \mathit{apic}.\mathit{IRR}[v/32][v \bmod 32] \wedge \mathit{apic}.\mathit{SVR}.\mathit{ASE}.
\end{array}
$$

The APIC software enable bit (ASE) of the SVR register enables/disables maskable interrtupts (Section 10.4).

When the core is ready to service a maskable interrupt (Section 9.3), the APIC checks whether there is a pending interrupt with the priority above the threshold:

$$
\begin{array}{l}
\mathit{intr\text{-}pending}(\mathit{apic} \in \mathit{APIC}) \in \mathbb{B} \triangleq \\
\quad \exists i \in \mathbb{B}^8 : \mathit{apic}.\mathit{IRR}[i/32][i \bmod 32] \neq 0 \wedge i[7:4] > \mathit{apic}.\mathit{PPR}.\mathit{priority}.
\end{array}
$$

For the threshold computation, the APIC has two priority registers:
$TPR, PPR \in \mathit{PriorityReg}$, where $\mathit{PriorityReg} = \mathbb{B}^{32}$ with the following abbreviations for all $x \in \mathit{PriorityReg}$:

$$
\begin{array}{ll}
x[3:0] & = x.\mathit{subpriority} \in \mathbb{B}^4, \\
x[7:4] & = x.\mathit{priority} \in \mathbb{B}^4, \\
x[31:8] & = x.\mathit{reserved} \in \mathbb{B}^{24}.
\end{array}
$$

Software controls the task priority register (TPR), which is aliased with the $CR8$ of the processor core. The processor priority register (PPR) is the maximum of the TPR and the priority of the top in-service interrupt:

$$
\begin{array}{l}
\mathit{update\text{-}PPR}(\mathit{apic} \in \mathit{APIC}) \in \mathit{APIC} \triangleq \\
\quad \textbf{let } v = \mathit{top\text{-}in\text{-}service\text{-}intr}(\mathit{apic}) \textbf{ in} \\
\quad \textbf{let } x = \mathit{max}(v[7:4] \circ 0^4, \mathit{apic}.\mathit{TPR}) \textbf{ in} \\
\quad \mathit{apic} \textbf{ with } [\mathit{PPR} \mapsto x],
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{top\text{-}in\text{-}service\text{-}intr}(\mathit{apic} \in \mathit{APIC}) \in \mathbb{B}^8 \triangleq \\
\quad \textbf{choose } i \in \mathbb{B}^8 : ((i = 0 \vee \mathit{apic}.\mathit{ISR}[i/32][i \bmod 32] \neq 0) \wedge \\
\quad\quad \forall j \in \mathbb{B}^8 : j > i \Rightarrow \mathit{apic}.\mathit{ISR}[j/32][j \bmod 32] = 0) \textbf{ in } i.
\end{array}
$$

Thus, if the priority of the top pending interrupt is above the threshold, then the APIC sends it to the core for handling.

$$\begin{array}{l}\mathit{top\text{-}pending\text{-}intr}(\mathit{apic} \in \mathit{APIC}) \in \mathbb{B}^8 \triangleq \\ \quad \textbf{choose } i \in \mathbb{B}^8 : ((i = 0 \vee \mathit{apic}.\mathit{IRR}[i/32][i \bmod 32] \neq 0) \wedge \\ \quad\quad \forall j \in \mathbb{B}^8 : j > i \Rightarrow \mathit{apic}.\mathit{IRR}[j/32][j \bmod 32] = 0) \textbf{ in } i.\end{array}$$

When the core completes the jump to the interrupt service routine, it notifies the APIC, and the APIC changes the state of the interrupt from 'pending' to 'in-service':

$$\begin{array}{l}\mathit{intr\text{-}delivered}(\mathit{apic} \in \mathit{APIC}, v \in \mathbb{B}^8) \in \mathit{APIC} \triangleq \\ \quad \textbf{let } i = v/32 \textbf{ in} \\ \quad \textbf{let } j = v \bmod 32 \textbf{ in} \\ \quad \mathit{update\text{-}PPR}(\mathit{apic} \textbf{ with } [\mathit{ISR}[i][j] \mapsto 1, \mathit{IRR}[i][j] \mapsto 0]).\end{array}$$

The interrupt service routine must write to the EOI register of the APIC before exiting. The act of writing causes the APIC to reset the highest bit in the ISR register and thus completes the interrupt handling:

$$\begin{array}{l}\mathit{write\text{-}EOI}(\mathit{apic} \in \mathit{APIC}, \mathit{data} \in \mathbb{B}^{32}) \in \mathit{APIC} \triangleq \\ \quad \textbf{let } v = \mathit{top\text{-}in\text{-}service\text{-}intr}(\mathit{apic}) \textbf{ in} \\ \quad \textbf{let } i = v/32 \textbf{ in} \\ \quad \textbf{let } j = v \bmod 32 \textbf{ in} \\ \quad \mathit{update\text{-}PPR}(\mathit{apic} \textbf{ with } [\mathit{ISR}[i][j] \mapsto 0]).\end{array}$$

Note that the APIC also sends an EOI message to the source device for level-triggered interrupts. However, we cannot express it as we do not model devices.

The $\mathit{IRR}, \mathit{ISR}, \mathit{TMR}, \mathit{PPR}$ registers are read-only, write accesses do not affect these registers. Software can update the TPR register either by writing to the corresponding APIC region or by writing to the $\mathit{CR8}$ core register. The APIC TPR register and the core $\mathit{CR8}$ register are synchronized before and after instruction execution.

$$\begin{array}{l}\mathit{write\text{-}TPR}(\mathit{apic} \in \mathit{APIC}, \mathit{data} \in \mathit{PriorityReg}) \in \mathit{APIC} \triangleq \\ \quad \textbf{let } x = \mathit{data} \textbf{ with } [\mathit{reserved} \mapsto 0] \textbf{ in} \\ \quad \mathit{update\text{-}APR}(\mathit{update\text{-}PPR}(\mathit{apic} \textbf{ with } [\mathit{TPR} \mapsto x])).\end{array}$$

As we do not have devices, we add a trivial transition that fires interrupts nondeterministically:

label	$\mathit{incoming\text{-}intr}(i \in \mathit{pid}, \mathit{vector} \in \mathbb{B}^8, \mathit{trigger\text{-}mode} \in \mathbb{B})$
guard	$\mathit{apic\text{-}enabled}(\mathit{core}[i]) \wedge \mathit{can\text{-}accept\text{-}intr}(\mathit{apic}[i], \mathit{vector}, \mathit{trigger\text{-}mode})$
effect	$\mathit{apic}'[i] = \mathit{accept\text{-}intr}(\mathit{apic}[i], \mathit{vector}, \mathit{trigger\text{-}mode})$

10.2 INIT, NMI, SIPI

The APIC does not have software visible registers that control management of INIT, NMI, and SIPI interrupts. Therefore, these interrupts are simply latched and forwarded to the core when the core is ready to process them.

The $\mathit{initrr} \in \mathbb{B}$ register stores the pending INIT interrupt. An incoming INIT is accepted if there is no pending INIT:

$$\mathit{can\text{-}accept\text{-}init}(\mathit{apic} \in \mathit{APIC}) \in \mathit{APIC} \triangleq \neg \mathit{apic}.\mathit{initrr},$$

$$\mathit{accept\text{-}init}(\mathit{apic} \in \mathit{APIC}) \in \mathit{APIC} \triangleq \mathit{apic}\ \mathbf{with}\ [\mathit{initrr} \mapsto 1\}.$$

Testing for a pending INIT is straightforward:

$$\mathit{init\text{-}pending}(\mathit{apic} \in \mathit{APIC}) \in \mathbb{B} \triangleq \mathit{apic}.\mathit{initrr}.$$

When the processor core finishes initialization, the initrr register is reset to 0:

$$\mathit{init\text{-}delivered}(\mathit{apic} \in \mathit{APIC}) \in \mathbb{B} \triangleq \mathit{apic}\ \mathbf{with}\ [\mathit{initrr} \mapsto 0].$$

Startup interrupts are processed likewise:

$$\mathit{can\text{-}accept\text{-}sipi}(\mathit{apic} \in \mathit{APIC}) \in \mathit{APIC} \triangleq \neg \mathit{apic}.\mathit{sipi},$$

$$\mathit{accept\text{-}sipi}(\mathit{apic} \in \mathit{APIC}, \mathit{vector} \in \mathbb{B}^8) \in \mathit{APIC} \triangleq \\ \quad \mathit{apic}\ \mathbf{with}\ [\mathit{sipirr} \mapsto 1, \mathit{sipi\text{-}vector} \mapsto \mathit{vector}\},$$

$$\mathit{sipi\text{-}pending}(\mathit{apic} \in \mathit{APIC}) \in \mathbb{B} \triangleq \mathit{apic}.\mathit{sipirr},$$

$$\mathit{sipi\text{-}delivered}(\mathit{apic} \in \mathit{APIC}) \in \mathit{APIC} \triangleq \mathit{apic}\ \mathbf{with}\ [\mathit{sipirr} \mapsto 0].$$

Non-maskable interrupts are slightly complicated because the APIC cannot forward an NMI to the core if the core is already handling another NMI. Therefore, we need two registers: $\mathit{nmirr} \in \mathbb{B}$ and $\mathit{nmisr} \in \mathbb{B}$:

$$\mathit{can\text{-}accept\text{-}nmi}(\mathit{apic} \in \mathit{APIC}) \in \mathbb{B} \triangleq \neg \mathit{apic}.\mathit{nmirr},$$

$$\mathit{accept\text{-}nmi}(\mathit{apic} \in \mathit{APIC}) \in \mathit{APIC} \triangleq \mathit{apic}\ \mathbf{with}\ [\mathit{nmirr} \mapsto 1],$$

$$\mathit{nmi\text{-}pending}(\mathit{apic} \in \mathit{APIC}) \in \mathbb{B} \triangleq \neg \mathit{apic}.\mathit{nmisr} \wedge \mathit{apic}.\mathit{nmirr},$$

$$\mathit{nmi\text{-}delivered}(\mathit{apic} \in \mathit{APIC}) \in \mathit{APIC} \triangleq \mathit{apic}\ \mathbf{with}\ [\mathit{nmisr} \mapsto 1, \mathit{nmirr} \mapsto 0],$$

$$\mathit{nmi\text{-}served}(\mathit{apic} \in \mathit{APIC}) \in \mathit{APIC} \triangleq \mathit{apic}\ \mathbf{with}\ [\mathit{nmisr} \mapsto 0].$$

Note that there is no end-of-interrupt register for non-maskable interrupts. The function $\mathit{nmi\text{-}served}$ is invoked when the core executes the return from interrupt service routine (IRET) instruction.

INIT and SIPI are interprocessor interrupts. However, NMI can be sent by a device. We model it with the following transition:

label	$\mathit{incoming\text{-}nmi}(i \in \mathit{pid})$
guard	$\mathit{apic\text{-}enabled}(\mathit{core}[i]) \wedge \mathit{can\text{-}accept\text{-}nmi}(\mathit{apic}[i])$
effect	$\mathit{apic}'[i] = \mathit{accept\text{-}nmi}(\mathit{apic}[i])$

10.3 Interprocessor interrupts

In order to send an interprocessor interrupt, software writes the information about the interrupt to the interrupt control registers $ICRH, ICRL$. The act of writing to the $ICRL$ initiates the IPI delivery. The highest byte in the $ICRH$ specifies the interrupt destination:

$$\begin{aligned} ICRH[23:0] &= ICRH.reserved \in \mathbb{B}^{24}, \\ ICRH[31:24] &= ICRH.DES \in \mathbb{B}^8 \qquad \text{(destination)}. \end{aligned}$$

There are two destination modes: physical and logical, specified by the DM bit in the $ICRL$. In physical mode, the interrupt destination is simply matched with the APIC ID, which is the highest byte in the $APIC\text{-}ID$ register:

$$\begin{aligned} APIC\text{-}ID[23:0] &= APIC\text{-}ID.reserved \in \mathbb{B}^{24}, \\ APIC\text{-}ID[31:24] &= APIC\text{-}ID.AID \in \mathbb{B}^8 \qquad \text{(APIC ID)}. \end{aligned}$$

Given the destination byte of the IPI, we check whether an APIC is a reciever using the following function:

$$\begin{aligned} &ipi\text{-}dest\text{-}physical(apic \in APIC, dest \in \mathbb{B}^8) \in \mathbb{B} \triangleq \\ &\quad apic.APIC\text{-}ID.AID = dest. \end{aligned}$$

Logical mode has two submodes: flat and cluster. The destination format register DFR of a receiving APIC selects the submode:

$$\begin{aligned} DFR[27:0] &= DFR.reserved \in \mathbb{B}^{28}, \\ DFR[31:28] &= DFR.model \in \mathbb{B}^4. \end{aligned}$$

The $DFR.model$ can be either $FLAT = 15$ or $CLUSTER = 0$. The official manuals do not specify what happens if the $model$ has other values. In flat submode, the IPI destination byte is used as a mask to select a group of recievers. In cluster submode, the higher nibble of the IPI destination byte specifies the cluster ID and the lower nibble is used as a mask to select a group of recievers within the cluster:

$$\begin{aligned} &ipi\text{-}dest\text{-}logical(apic \in APIC, dest \in \mathbb{B}^8) \in \mathbb{B} \triangleq \\ &\quad \textbf{let } x = apic.LDR.DLID \textbf{ in} \\ &\quad \textbf{if } apic.DFR.model = FLAT \textbf{ then } (x \wedge_8 dest) \neq 0 \\ &\quad \textbf{else } x[7:4] = dest[7:4] \wedge (x[3:0] \wedge_4 dest[3:0]) \neq 0, \end{aligned}$$

where the $DLID$ is the highest byte in the logical destination register LDR:

$$\begin{aligned} LDR[23:0] &= LDR.reserved \in \mathbb{B}^{24}, \\ LDR[31:24] &= LDR.DLID \in \mathbb{B}^8 \qquad \text{(destination logical ID)}. \end{aligned}$$

The $ICRL$ register is the main IPI register. It describes what kind of interrupt to trigger at the remote processor. It also selects the delivery mode and shows the delivery status.

Let x denote the $ICRL$, then

$$\begin{aligned}
x[7:0] &= x.VEC \in \mathbb{B}^8 && \text{(vector)},\\
x[10:8] &= x.MT \in \mathbb{B}^3 && \text{(message type)},\\
x[11] &= x.DM \in \mathbb{B} && \text{(destination mode)},\\
x[12] &= x.DS \in \mathbb{B} && \text{(delivery status)},\\
x[14:13] &= x.reserved[1:0] \in \mathbb{B}, &&\\
x[15] &= x.TGM \in \mathbb{B} && \text{(trigger mode)},\\
x[17:16] &= x.RRS \in \mathbb{B}^2 && \text{(remote read status)},\\
x[19:18] &= x.DSH \in \mathbb{B}^2 && \text{(destination shorthand)},\\
x[31:20] &= x.reserved[13:2] \in \mathbb{B}^{12}. &&
\end{aligned}$$

We have already discussed the destination mode except for the value encoding:

- $PHYSICAL = 0$,
- $LOGICAL = 1$.

The destination shorthand allows to broadcast interrupts ignoring the $ICRH.DES$:

- $DESTINATION = 0$: this value selects a regular delivery described above.
- $SELF = 1$: the IPI is sent only to the source APIC.
- $ALL = 2$: the IPI is sent to all APICs including the source.
- $ALL\text{-}BUT\text{-}SELF = 3$: the IPI is sent to all APICs excluding the source.

Formalizing this information, we derive a predicate that checks whether the i-th APIC is a receiver of the IPI, defined by copies $icrl_s, icrh_s$ of ICR registers of the sender s:

$$\begin{aligned}
&ipi\text{-}dest(i \in pid, core, apic, s \in pid, icrl_s, icrh_s) \in \mathbb{B} \triangleq\\
&\quad \textbf{if } \neg apic\text{-}enabled(core) \textbf{ then } 0\\
&\quad \textbf{else if } icrl_s.DSH = DESTINATION \textbf{ then}\\
&\qquad \textbf{if } icrh_s.DES = \text{0FFh} \textbf{ then } 1\\
&\qquad \textbf{else if } icrl_s.DM = LOGICAL \textbf{ then}\\
&\qquad\quad ipi\text{-}dest\text{-}logical(apic, icrh_s.DES)\\
&\qquad \textbf{else if } icrl_s.DM = PHYSICAL \textbf{ then}\\
&\qquad\quad ipi\text{-}dest\text{-}physical(apic, icrh_s.DES)\\
&\qquad \textbf{else } 0\\
&\quad \textbf{else if } icrl_s.DSH = SELF \textbf{ then } s = i\\
&\quad \textbf{else if } icrl_s.DSH = ALL\text{-}BUT\text{-}SELF \textbf{ then } s \neq i\\
&\quad \textbf{else } icrl_s.DSH = ALL.
\end{aligned}$$

The definition is straightforward, except for one special case: when the IPI destination is 0FFh, then the IPI is sent to all APICs.

The delivery status indicates whether the APIC is in the process of delivering the IPI ($DS = 1$) or idle ($DS = 0$). This bit is read-only for software. The APIC sets it to 1 after a write access to the $ICRL$ and resets it to 0 after delivering the IPI. Subsequent write accesses are stalled if $DS = 1$:

$$\begin{aligned}
&can\text{-}write\text{-}ICRL(apic \in APIC, data \in ICRegL) \in APIC \triangleq\\
&\quad \neg apic.ICRL.DS,\\
&\\
&write\text{-}ICRL(apic \in APIC, data \in ICRegL) \in APIC \triangleq\\
&\quad \textbf{let } x = data \textbf{ with } [DS \mapsto 1, reserved \mapsto 0] \textbf{ in}
\end{aligned}$$

$apic$ **with** $[ICRL \mapsto x]$.

The message type, the vector, and the trigger mode fields define the interrupt. There are the following message types:

- $MT\text{-}FIXED = 0$: a maskable interrupt.
- $MT\text{-}LOWEST\text{-}PRIORITY = 1$: a maskable interrupt and it is delivered to the APIC with the lowest value of the arbitration priority register. This register contains the maximum of the PPR, the TPR, and the priority of the top pending interrupt:

  $update\text{-}APR(apic \in APIC) \in APIC \triangleq$
  let $v = top\text{-}pending\text{-}intr(apic)$ **in**
  let $x = max(v[7:4] \circ 0^4, apic.PPR, apic.TPR)$ **in**
  $apic$ **with** $[APR \mapsto x]$.

- $MT\text{-}SMI = 2$: a system management interrupt.
- $MT\text{-}REMOTE\text{-}READ = 3$: a read request of the remote APIC register with the offset specified in the $ICRL.VEC$. As it seems to be a legacy feature, we do not support it in the current model.
- $MT\text{-}NMI = 4$: a non-maskable interrupt, it is assumed to be edge-triggered and the vector is assumed to be 2.
- $MT\text{-}INIT = 5$: an initialization request.
- $MT\text{-}SIPI = 6$: a startup interrupt.
- $MT\text{-}EXT\text{-}INT = 7$: an external interrupt.

Depending on whether the interrupt is a lowest-priority interrupt or not, we choose one of the two ways to deliver the IPI:

- lowest-priority: iterate over all APICs that can accept the IPI and select the one with the lowest ARP. Then deliver the IPI to the selected APIC. Note that the selection is not atomic, so by the time we deliver the IPI, the selected APIC might be unable to accept the IPI. The official manuals do not specify what happens in such cases.
- otherwise: iterate over all APICs and deliver the IPI to each APIC that can accept it.

We test whether the i-th APIC can accept the IPI, defined by copies $icrl_s, icrh_s$ of ICR registers of the sender s, as follows:

$can\text{-}accept\text{-}ipi(i, core, apic, s, icrl_s, icrh_s) \in \mathbb{B} \triangleq$
  if $\neg ipi\text{-}dest(i, core, apic, s, icrl_s, icrh_s)$ **then** 0
  else if $icrl_s.MT \in \{FIXED, LOWEST\text{-}PRIORITY\}$ **then**
    $can\text{-}accept\text{-}intr(apic, icrl_s.VEC, icrl_s.TGM)$
  else if $icrl_s.MT = NMI$ **then** $can\text{-}accept\text{-}nmi(apic)$
  else if $icrl_s.MT = INIT$ **then** $can\text{-}accept\text{-}init(apic)$
  else if $icrl_s.MT = SIPI$ **then** $can\text{-}accept\text{-}sipi(apic, icrl_s.VEC)$
  else 0.

To deliver the IPI, we use the $accept\text{-}xxxx$ function for the $xxxx$ message type:

$accept\text{-}ipi(i, core, apic, s, icrl, icrh) \in APIC \triangleq$
  if $icrl.MT \in \{FIXED, LOWEST\text{-}PRIORITY\}$ **then**
    $accept\text{-}intr(apic, icrl.VEC, icrl.TGM)$
  else if $icrl.MT = NMI$ **then** $accept\text{-}nmi(apic)$

else if $icrl.MT = INIT$ **then** $accept\text{-}init(apic)$
else if $icrl.MT = SIPI$ **then** $accept\text{-}sipi(apic, icrl.VEC)$
else $apic$.

Note that the IPI delivery is stateful, i.e. we need to keep track of the already processed APICs. For this reason, we add an auxiliary IPI component to the abstract machine and formalize the IPI delivery as transitions of this component. Section 10.6 describes the IPI component and the transitions in details.

10.4 Miscellaneous

The APIC version register is read-only, and the lowest byte contains the APIC version:

$$\begin{aligned} VERSION[7:0] &= VERSION.VER \in \mathbb{B}^8 \qquad \text{(apic version)}, \\ VERSION[31:8] &= VERSION.reserved \in \mathbb{B}^{24}. \end{aligned}$$

In real hardware, the APIC delivers a maskable interrupt to the processor in several steps, i.e. non-atomically. During this process, the task priority register may change and mask the interrupt. In such cases, the APIC delivers a spurious interrupt instead of the original interrupt. The SVR register specifies the vector of the spurious interrupt:

$$\begin{aligned} SVR[7:0] &= SVR.VEC \in \mathbb{B}^8 \qquad \text{(vector)}, \\ SVR[8] &= SVR.ASE \in \mathbb{B} \qquad \text{(APIC software enable)}, \\ SVR[31:9] &= SVR.reserved \in \mathbb{B}^{23}. \end{aligned}$$

We model spurious interrupts by allowing them to occur at any time nondeterministically. The $spuriousrr \in \mathbb{B}$ register indicates whether there is a pending spurious interrupt:

$$spurious\text{-}pending(apic \in APIC) \in \mathbb{B} \triangleq \neg apic.spuriousrr.$$

The register is reset to zero after the processor completes the jump to the service routine:

$$spurious\text{-}delivered(apic \in APIC) \in APIC \triangleq apic \textbf{ with } [spuriousrr \mapsto 0].$$

A spurious interrupt may occur at any time:

label	$incoming\text{-}spurious(i \in pid)$
guard	$apic\text{-}enabled(core[i]) \wedge \neg apic[i].spuriousrr$
effect	$apic'[i].spuriousrr = 1$

The ASE bit the of the SVR register has nothing to do with spurious interrupts. When the bit is cleared, the APIC stops accepting/delivering maskable interrupts and sets the mask bit in all LVT entries:

$$\begin{aligned} &write\text{-}SVR(apic \in APIC, data \in SpuriousVectorReg) \in APIC \triangleq \\ &\quad apic \textbf{ with } [SVR \mapsto (data \textbf{ with } [reserved \mapsto 0]), \\ &\qquad\qquad TIMER.M \mapsto apic.TIMER.M \vee \neg data.ASE, \\ &\qquad\qquad THERMAL.M \mapsto apic.THERMAL.M \vee \neg data.ASE, \end{aligned}$$

$$
\begin{aligned}
&PERF.M \mapsto apic.PERF.M \vee \neg data.ASE,\\
&LINT0.M \mapsto apic.LINT0.M \vee \neg data.ASE,\\
&LINT1.M \mapsto apic.LINT1.M \vee \neg data.ASE,\\
&ERROR.M \mapsto apic.ERROR.M \vee \neg data.ASE].
\end{aligned}
$$

Software can check the APIC status by reading the error status register ESR. The APIC records encountered errors in the shadow register $shadowESR$. On any write access to the ESR, the APIC copies errors from the shadows register to the ESR and resets the shadow register:

$$
\begin{aligned}
&write\text{-}ESR(apic \in APIC, data \in \mathbb{B}^{32}) \in APIC \triangleq\\
&\quad apic\ \mathbf{with}\ [ESR \mapsto apic.shadowESR, shadowESR \mapsto 0].
\end{aligned}
$$

Let x denote the ESR or the $shadowESR$, then:

$$
\begin{aligned}
x[1:0] &= x.reserved[1:0] \in \mathbb{B}^2,\\
x[2] &= x.SAE \in \mathbb{B} && \text{(sent accept error)},\\
x[3] &= x.RAE \in \mathbb{B} && \text{(receive accept error)},\\
x[4] &= x.reserved[2] \in \mathbb{B},\\
x[5] &= x.SIV \in \mathbb{B} && \text{(sent illegal vector error)},\\
x[6] &= x.RIV \in \mathbb{B} && \text{(receive illegal vector error)},\\
x[7] &= x.IRA \in \mathbb{B} && \text{(illegal register address error)},\\
x[31:8] &= x.reserved[26:3] \in \mathbb{B}^{24}.
\end{aligned}
$$

A ‘sent accept error’ occurs when an IPI sent by the APIC is not accepted by any APIC.

$$
\begin{aligned}
&sent\text{-}accept\text{-}error(apic \in APIC) \in APIC \triangleq\\
&\quad apic\ \mathbf{with}\ [shadowESR.SAE \mapsto 1].
\end{aligned}
$$

The precise meaning of the ‘receive accept error’ is not completely clear. The official manuals say that the APIC detects this error when it receives an IPI that is not accepted by any APIC, including itself.

$$
\begin{aligned}
&receive\text{-}accept\text{-}error(apic \in APIC) \in APIC \triangleq\\
&\quad apic\ \mathbf{with}\ [shadowESR.RAE \mapsto 1].
\end{aligned}
$$

A ‘sent illegal vector error’ occurs when the APIC sends an IPI with an illegal vector.

$$
\begin{aligned}
&sent\text{-}illegal\text{-}vector(apic \in APIC) \in APIC \triangleq\\
&\quad apic\ \mathbf{with}\ [shadowESR.SIV \mapsto 1].
\end{aligned}
$$

A ‘receive illegal vector error’ occurs when the APIC receives an IPI with an illegal vector.

$$
\begin{aligned}
&receive\text{-}illegal\text{-}vector(apic \in APIC) \in APIC \triangleq\\
&\quad apic\ \mathbf{with}\ [shadowESR.RIV \mapsto 1].
\end{aligned}
$$

An ‘illegal register address error’ occurs when the processor accesses the reserved area in the APIC page.

$$
\begin{aligned}
&illegal\text{-}register\text{-}address(apic \in APIC) \in APIC \triangleq\\
&\quad apic\ \mathbf{with}\ [shadowESR.IRA \mapsto 1].
\end{aligned}
$$

Besides the discussed registers, the APIC has a set of local vector table (LVT) registers. The registers define how to process interrupt requests from the processor local devices, such as the timer, the thermal sensor, etc. As we do not model devices, we exclude the registers from the current version of the document.

10.5 Register accesses

In this section we describe how the APIC handles read/write accesses to its memory-mapped page. If an access does not correspond to any register of the APIC, then an illegal register address error is recorded. A read access in such case returns an undefined result. Recall that the environment uses the *read* function of the APIC to carry out a read access (Section 4). Since the function needs to return an undefined result sometimes, we introduce a dummy APIC register $undef \in \mathbb{B}^{64}$, which is allowed to change nondeterministically. The *read* function uses the dummy register to return an undefined result.

label	$set\text{-}undefined(i \in pid, x \in \mathbb{B}^{64})$
guard	
effect	$apic'[i].undef = x$

The precondition for the *read* function is that the access is in the APIC page:

$$can\text{-}read(apic \in APIC, core \in Core, pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^8) \in \mathbb{B} \triangleq$$
$$\quad in\text{-}apic\text{-}range(core, pa),$$

where

$$in\text{-}apic\text{-}range(core \in Core, pa \in \mathbb{B}^{pq}) \in \mathbb{B} \triangleq$$
$$\quad apic\text{-}base(core) \leq pa \circ 0^3 < apic\text{-}base(core) + 2^{12} \wedge apic\text{-}enabled(core).$$

The *read* function checks that the access is aligned and then compares the access offset with the offset of each register. If the matching register is the EOI, then the function returns an undefined result because the EOI is a write-only register. For any other register, the function simply returns the register value zero-extended to 64 bits:

$$read(apic \in APIC, core \in Core, pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^8) \in (APIC, \mathbb{B}^{64}) \triangleq$$
$$\quad \textbf{let } offset = pa \circ 0^3 - apic\text{-}base(core)) \textbf{ in}$$
$$\quad \textbf{if } mask \neq 0^4 \circ 1^4 \textbf{ then } (illegal\text{-}register\text{-}address(apic), apic.undef)$$
$$\quad \textbf{else if } offset = APIC\text{-}ID\text{-}OFFSET \textbf{ then } (apic, zxt_{64}(apic.APIC\text{-}ID))$$
$$\quad \ldots$$
$$\quad \textbf{else if } offset = EOI\text{-}OFFSET \textbf{ then } (apic, apic.undef)$$
$$\quad \ldots$$
$$\quad \textbf{else } (illegal\text{-}register\text{-}address(apic), apic.undef).$$

A write access to the $ICRL$ register stalls if the previous IPI is being delivered:

$$can\text{-}write(apic \in APIC, core \in Core, pa \in \mathbb{B}^{pq}, mask \in \mathbb{B}^8, data \in \mathbb{B}^{64}) \in \mathbb{B} \triangleq$$
$$\quad \textbf{if } \neg in\text{-}apic\text{-}range(core, pa) \textbf{ then } 0$$
$$\quad \textbf{else let } offset = pa \circ 0^3 - apic\text{-}base(core)) \textbf{ in}$$
$$\quad\quad \textbf{if } offset = ICRL\text{-}OFFSET \textbf{ then } \neg apic.ICRL.DS$$
$$\quad\quad \textbf{else } 1.$$

Like the *read* function, the *write* function makes a big case distinction on the access offset. In case it finds a matching register *xxxx*, it invokes the corresponding

$$\mathit{write\text{-}xxxx}(\mathit{apic} \in \mathit{APIC}, \mathit{data} \in \mathbb{B}^{32}) \in \mathit{APIC}.$$

In previous sections, we defined *write-xxxx* functions for non-trivial cases. For other cases, we assume that the *write-xxxx* function simply updates the corresponding register with the given data preserving the read-only bits.

$$\begin{array}{l}
\mathit{write}(\mathit{apic} \in \mathit{APIC}, \mathit{core} \in \mathit{Core}, \mathit{pa} \in \mathbb{B}^{pq}, \mathit{mask} \in \mathbb{B}^{8}, \mathit{data} \in \mathbb{B}^{64}) \in \mathit{APIC} \triangleq \\
\quad \textbf{let } \mathit{offset} = \mathit{pa} \circ 0^3 - \mathit{apic\text{-}base}(\mathit{core})) \textbf{ in} \\
\quad \textbf{if } \mathit{mask} \neq 0^4 \circ 1^4 \textbf{ then} \\
\quad\quad \mathit{illegal\text{-}register\text{-}address}(\mathit{apic}) \\
\quad \textbf{else} \\
\quad\quad \textbf{if } \mathit{offset} = \mathit{APIC\text{-}ID\text{-}OFFSET} \textbf{ then} \\
\quad\quad\quad \mathit{write\text{-}APIC\text{-}ID}(\mathit{apic}, \mathit{data}[31:0]) \\
\quad\quad \ldots \\
\quad\quad \textbf{else } \mathit{illegal\text{-}register\text{-}address}(\mathit{apic}).
\end{array}$$

10.6 IPI Delivery

In this section we attempt to model the IPI delivery mechanism, which is not described precisely in the official manuals because it is implementation dependent. With the help of nondeterminism, we try to abstract from implementation details and yet give the meaning to the delivery status, the delivery errors, and the lowest-priority message. In order to keep the model simple, we omit some obscure features:

- remote register read: an APIC can send a register read request to another APIC via IPI. This seems to be a legacy feature.
- focus processor delivery: a processor can declare itself a focus processor for some interrupt vector. When an interrupt with that vector is broadcast in lowest-priority mode, the focus processor gets the interrupt regardless of its arbitration priority.
- receive accept error: an APIC detects this error when it receives an IPI that is not accepted by any APIC, including itself. To model this error, we could add another phase where all destination APICs get a notification that no other APIC has accepted the interrupt.

The $\mathit{ipi} \in \mathit{IPI}$ component of the abstract machine maintains the state of the IPI delivery:

$$\begin{array}{l}
\mathit{IPI} \triangleq [\, \mathit{phase} \in \mathit{IPI\text{-}Phase}, \\
\qquad \mathit{sender} \in \mathit{pid}, \\
\qquad \mathit{ICRL} \in \mathit{ICRegL}, \\
\qquad \mathit{ICRH} \in \mathit{ICRegH}, \\
\qquad \mathit{lowest} \in \mathit{pid} \cup \{\epsilon\}, \\
\qquad \mathit{delivered\text{-}any} \in \mathbb{B}, \\
\qquad \mathit{done} \in \mathit{pid} \to \mathbb{B}],
\end{array}$$

where

$$\mathit{IPI\text{-}Phase} \triangleq \{\mathit{IDLE}, \mathit{FIND\text{-}LOWEST}, \mathit{DELIVER\text{-}REGULAR}, \mathit{FINISH}\}.$$

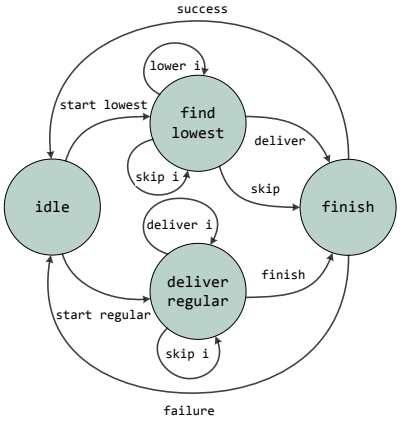

Figure 10.1: IPI Delivery

Figure 10.1 shows the phases and the transitions of the IPI delivery.

- in the *idle* phase we choose nondeterministically an APIC s that wants to send an IPI ($apic[s].ICRL.DS = 1$). We set $ipi.sender = s$ and save the ICR registers of the APIC in the $ipi.ICRL, ipi.ICRH$. In case the IPI is a lowest-priority interrupt, we proceed to the *find-lowest* phase. Otherwise, we proceed to the *deliver-regular* phase.
- in the *deliver-regular* phase, we iterate over all APICs and deliver the IPI to an APIC if it is in the destination set and can accept the IPI. After processing an APIC, we set the corresponding bit in the $ipi.done$, so that we do not deliver to the same APIC twice.
  When all APIC are processed, we proceed to the *finish* phase.
- in the *find-lowest* phase, we iterate over all APICs and maintain in the $ipi.lowest$ the index of the APIC with the lowest value of the APR among the already processed APICs that can accept the IPI. The $ipi.done$ maintains the set of the processed APICs.
  When all APIC are processed, we deliver the IPI to the APIC with index $ipi.lowest$ (if $ipi.lowest \neq \epsilon$) and proceed to the *finish* phase.
- the *finish* phase concludes the IPI delivery. In this phase we check if at least one APIC has accepted the IPI (indicated by the $ipi.delivered\text{-}any$). In case none of the APICs accepted the IPI, we record a 'sent accept error' in the error status register of the sending APIC.
  After that, we clear the delivery status bit of the sender and return to the *idle* phase.

In the next two subsections we describe each transition in details.

10.6.1 Regular IPI

The delivery of a regular interrupt starts with the initialization of the ipi fields. Note that we save the ICR registers of the sending APIC in the ipi fields. Otherwise, an update of the $ICRH$ register, while the IPI is being delivered, would make the IPI inconsistent.

label	$start(i \in pid)$
guard	$ipi.phase = IDLE$, $apic[i].ICRL.DS$, $apic[i].ICRL.MT \neq MT\text{-}LOWEST\text{-}PRIORITY$
effect	$ipi'.sender = i$, $ipi'.ICRL = apic[i].ICRL$, $ipi'.ICRH = apic[i].ICRH$, $ipi'.phase = DELIVER\text{-}REGULAR$, $ipi'.delivered\text{-}any = 0$, $ipi'.done = \lambda k \in pid \rightarrow 0$

In the *deliver-regular* phase, we choose nondeterministically an unprocessed APIC and check whether it can accept the IPI or not using the *can-accept-ipi* function, which internally checks whether the APIC is in the destination set or not (Section 10.3). In the former case, we deliver:

label	$deliver(j \in pid)$
guard	$ipi.phase = DELIVER\text{-}REGULAR$, $\neg ipi.done[j]$, $can\text{-}accept\text{-}ipi(j, core[j], apic[j], ipi.sender, ipi.ICRL, ipi.ICRH)$
effect	$ipi'.done[j] = 1$, $ipi'.delivered\text{-}any = 1$, $apic'[j] = accept\text{-}ipi(j, apic[j], core[j], ipi.sender, ipi.ICRL, ipi.ICRH)$

Otherwise, we mark the APIC as 'processed' and skip it:

label	$skip(j \in pid)$
guard	$ipi.phase = DELIVER\text{-}REGULAR$, $\neg ipi.done[j]$, $\neg can\text{-}accept\text{-}ipi(j, core[j], apic[j], ipi.sender, ipi.ICRL, ipi.ICRH)$
effect	$ipi'.done[j] = 1$

When all APICs are processed, we go to the *finish* phase:

label	$finish\text{-}regular(j \in pid)$
guard	$ipi.phase = DELIVER\text{-}REGULAR$, $\forall k \in pid : ipi.done[k]$
effect	$ipi'.phase = FINISH$

In the *finish* phase, we have two cases: either the IPI has been delivered to at least one APIC or not. In the latter case, we record the error in the error status register of the sender.

label	*success*
guard	$ipi.phase = FINISH$, $ipi.delivered\text{-}any$
effect	$ipi'.phase = IDLE$, $apic'[ipi.sender] = apic[ipi.sender]$ **with** $[ICRL.DS \mapsto 0]$

label	*failure*
guard	$ipi.phase = FINISH$, $\neg ipi.delivered\text{-}any$
effect	$ipi'.phase = IDLE$, $x = apic[ipi.sender]$ **with** $[ICRL.DS \mapsto 0]$, $apic'[ipi.sender] = sent\text{-}accept\text{-}error(x)$

10.6.2 Lowest-Priority IPI

For the delivery of a lowest-priority interrupt, we initialize the same fields as for the delivery of a regular interrupt. Additionally, we set the $ipi.lowest$ to denote an empty index :

label	$start\text{-}lowest(i \in pid)$
guard	$ipi.phase = IDLE$, $apic[i].ICRL.DS$, $apic[i].ICRL.MT = MT\text{-}LOWEST\text{-}PRIORITY$
effect	$ipi'.sender = i$, $ipi'.ICRL = apic[i].ICRL$, $ipi'.ICRH = apic[i].ICRH$, $ipi'.phase = FIND\text{-}LOWEST$, $ipi'.lowest = \epsilon$, $ipi'.delivered\text{-}any = 0$, $ipi'.done = \lambda k \in pid \rightarrow 0$

In the *find-lowest* phase, we choose nondeterministically an unprocessed APIC and check whether it can accept the IPI and its APR is lower than the APR of the lowest priority APIC seen so far. In the former case, we update the $ipi.lowest$:

label	$lower(j \in pid)$
guard	$ipi.phase = FIND\text{-}LOWEST$, $\neg ipi.done[j]$, $can\text{-}accept\text{-}ipi(j, core[j], apic[j], ipi.sender, ipi.ICRL, ipi.ICRH)$, $ipi.lowest = \epsilon \vee apic[j].APR < apic[ipi.lowest]$
effect	$ipi'.lowest = j$, $ipi'.done[j] = 1$

Otherwise, we skip the APIC:

label	$\mathit{not\text{-}lower}(j \in \mathit{pid})$
guard	$\mathit{ipi.phase} = \mathit{FIND\text{-}LOWEST}$, $\neg \mathit{ipi.done}[j]$, $(\neg \mathit{can\text{-}accept\text{-}ipi}(j, \mathit{core}[j], \mathit{apic}[j], \mathit{ipi.sender}, \mathit{ipi.ICRL}, \mathit{ipi.ICRH})$ $\vee \mathit{ipi.lowest} \neq \epsilon \wedge \mathit{apic}[j].\mathit{APR} \geq \mathit{apic}[\mathit{ipi.lowest}].\mathit{APR})$
effect	$\mathit{ipi}'.\mathit{done}[j] = 1$

When all APICs are processed, the $\mathit{ipi.lowest}$ field is either empty or contains the index of the APIC with the lowest APR. In case the APIC can accept the IPI, we deliver it.

label	$\mathit{deliver\text{-}lowest}(j \in \mathit{pid})$
guard	$\mathit{ipi.phase} = \mathit{FIND\text{-}LOWEST}$, $\forall k \in \mathit{pid} : \mathit{ipi.done}[k]$, $\mathit{ipi.lowest} = j$, $\mathit{can\text{-}accept\text{-}ipi}(j, \mathit{core}[j], \mathit{apic}[j], \mathit{ipi.sender}, \mathit{ipi.ICRL}, \mathit{ipi.ICRH})$
effect	$\mathit{ipi.phase} = \mathit{FINISH}$, $\mathit{ipi.delivered\text{-}any} = 1$, $\mathit{apic}'[j] = \mathit{accept\text{-}ipi}(j, \mathit{apic}[j], \mathit{core}[j], \mathit{ipi.sender}, \mathit{ipi.ICRL}, \mathit{ipi.ICRH})$

label	$\mathit{skip\text{-}lowest}(j \in \mathit{pid})$
guard	$\mathit{ipi.phase} = \mathit{FIND\text{-}LOWEST}$, $\forall k \in \mathit{pid} : \mathit{ipi.done}[k]$, $(\mathit{ipi.lowest} = \epsilon \vee \mathit{ipi.lowest} = j$ $\wedge \neg \mathit{can\text{-}accept\text{-}ipi}(j, \mathit{core}[j], \mathit{apic}[j], \mathit{ipi.sender}, \mathit{ipi.ICRL}, \mathit{ipi.ICRH})$
effect	$\mathit{ipi.phase} = \mathit{FINISH}$

Finally, we proceed to the *finish* phase.

Part II

INSIDE PROCESSOR CORE

CHAPTER

ELEVEN

DSL SYNTAX AND SEMANTICS

This chapter introduces a domain-specific language (DSL) for register and instruction specification. In order to specify a register we need to provide its name, its width, its fields, and its initial value. Instruction specification is more complex. First we need information that allows us to fetch and decode the instruction. This includes the opcode and the operands. After that we need to specify how the instruction is executed, i.e we need to specify how the instruction changes the registers and the memory system. Since the memory system of a multiprocessor machine is distributed, specification of the memory system requires more sophisticated language than specification of the registers. We want to make the DSL as simple as possible, therefore we abstract the memory system by request/response registers. Thus, specification of instruction execution is reduced to specification of how the registers change. Note that registers change only when instruction execution completes successfully. An instruction might fail because of an exception or an intercept. This means that our DSL has to be powerful enough to express computations over registers with possibility of failure.

We represent a computation as a sequence of statements. There are nine kinds of statements:

1. conditional statement:

   ```
   if expr then stmts1
   else stmts2
   ```

   This statement evaluates the boolean expression `expr`. In case the expression is true, the computation proceeds with statements `stmts1`. Otherwise, the computation proceeds with statements `stmts2`. We will formally cover expressions in subsequent sections. For now, it is sufficient to think of expressions as of usual mathematical expressions enhanced with bit-string operations. Expressions can contain register references. Each register reference evaluates to the value of the corresponding register before the whole computation started.

2. “let” statement:

   ```
   let ident = expr
   ```

This statement binds the identifier `ident` to the value of the expression `expr`. Thus, in subsequent statements the identifier `ident` evaluates to the expression `expr`.

3. “register write” statement:

```
write expr to regname
```

This statement evaluates the expression `expr` and writes the value to the register `regname`. In contrast to other languages, the write does not take place immediately but deferred until the end of the whole computation. In the following example, the `x` is bound to the value of the register `RAX` before the computation started. Thus, the `x` is not necessarily equal to `FFFFFFFFFFFFFFFFh`.

```
write FFFFFFFFFFFFFFFFh to RAX
let x = RAX
```

The reason for deferred writes is that we want to collect writes and apply them all at once without exposing intermediate processor state in the middle of the computation. An important consequence is that any register reference that appears in an expression evaluates to the register value before the computation started.

4. “register undefine” statement:

```
undef regname
```

This statement marks the register `regname` as undefined. Likewise to the register write statement, the effect of this statement is deferred until the end of the computation.

5. “fail” statement:

```
fail expr
```

This statement stops the computation and cancels all previous register writes. The value of the `expr` is returned as the result of the computation.

6. “call” statement:

```
call action_res = action_name(action_args)
```

Same sequences of statements appear in instruction definitions over and over again. The standard way to avoid such repetition is to introduce a way to group a sequence of statements under a name. In most languages such named groups of statements are called functions or procedures. In our DSL the word “function” refers to a named expression (like in many functional languages). To highlight the imperative nature of statements, we use word the “action” for a named group of statements. Later we will show how to define an action. The “call” statement allows to insert the specified action into the current computation. Actions can have arguments and can return a value. Thus, the “call” statements executes the specified action and binds its result to the specified identifier.

7. “return” statement:

```
return expr
```

This statement exits the current action and returns the value of the `expr` as the action result.

8. “chain” statement:

   ```
   chain action_name
   ```

   We use this statement only three times in the whole instruction set specification. The statement completes the current computation, applies all register writes, and then starts a new computation using the statements of the specified action. If the new computation fails, the register values are restored to the state before the old computation. There is another version of the statement, which restores the registers to the state after the old computation but before the new computation:

   ```
   commit and chain action_name
   ```

   The “chain” statement is necessary when the instruction has two contexts. For example, a far control transfer instruction performs some actions in the context of the old code segment, changes the code segment, and performs some action in the context of the new code segment. We could do without the “chain” statement, but that would lead to code duplication.

9. “assume” statement:

   ```
   assume expr
   ```

   This statement is a hint for the type checker and has no effect on the computation. When the type checker sees this statement, it can assume that the boolean expression `expr` holds.

We specify an instruction by providing the instruction opcode, the operands, and the statements, which describes how the instruction is executed. The subsequent sections describe the syntax and semantics of the DSL more formally and in more detail.

11.1 Source Code Structure

Source code of the instruction set specification is written in ASCII-encoded text files. The specification consists of a list of top-level definitions. Each definition starts in a separate line and spans one or more lines. The end of each definition is detected using indentation:

- let n be the indentation level of the first line of the definition. The indentation level is the number of characters from the beginning of the line to the first non-space character.
- The subsequent lines belong to the definition as long as their indentation level is greater then n.

Comments are treated as spaces. A comment starts with the character # and extends to the following newline character.

We present the DSL syntax using an extended BNF notation. Nonterminal symbols in production rules are written in *slanted font*, while terminal symbols are written in

bold font. We use the following notational conventions in the right hand side of production rules:

$alt_1 \mid alt_2$	choice between two alternatives
$(group)$	parenthesis for grouping
$many^*$	repeat zero or more times
$many^+$	repeat one or more times
$optional^?$	optional
$\lfloor block \rfloor$	indentation sensitive block that spans many lines

Using this notation we can express the fact that source code consists of a list of indentation sensitive definitions as follows.

$$specification \longrightarrow \lfloor definition \rfloor^*$$

A top-level definition is either a type related definition, a register definition, a function definition, an action definition, or an instruction definition. We will elobarate on these definitions in subsequent sections.

$$\begin{aligned} definition \longrightarrow\ & parameter \mid invariant \mid set \mid record \mid layout \mid \\ & union \mid register \mid function \mid action \mid instruction \end{aligned}$$

We conclude this section with description of the lexical structure. Identifiers are strings of alphanumeric, underscore (**_**), quote (**'**), and dollar (**$**) characters. An identifier can start with a dollar character and cannot start with a digit. There is a number of reserved keywords that cannot be used as identifiers.

$$\begin{aligned} id &\longrightarrow \texttt{\$}^? letter(letter\_ \mid digit)^* \text{ except } reserved \\ letter\_ &\longrightarrow \texttt{\_} \mid \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\ digit &\longrightarrow \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\ reserved &\longrightarrow \texttt{if} \mid \texttt{then} \mid \texttt{elif} \mid \texttt{else} \mid \texttt{let} \mid \texttt{in} \mid \texttt{when} \mid \texttt{array} \mid \texttt{of} \mid \texttt{alias} \mid \\ &\qquad \texttt{with} \mid \texttt{and} \mid \texttt{or} \mid \texttt{not} \mid \texttt{bit} \mid \texttt{bits} \mid \texttt{nat} \mid \texttt{int} \mid \texttt{zxt} \mid \texttt{sxt} \end{aligned}$$

There are two types of numeric literals: decimal literals and non-decimal literals. Literals of the former type evaluate to natural numbers, literals of the latter type evaluate to bit-strings.

$$\begin{aligned} lit &\longrightarrow decimal \mid bitstring \\ decimal &\longrightarrow digit^+ \\ bitstring &\longrightarrow hex^+\texttt{h} \mid oct^+\texttt{o} \mid bin^+\texttt{b} \\ hex &\longrightarrow digit \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \\ oct &\longrightarrow \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{7} \\ bin &\longrightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

The length of a non-decimal literal is meaningful because it encodes the width of the bit-string. Thus, `0001b` evaluates to a bit-string of width four, while `1b` evaluates to a

bit. Each character in a hexadecimal literal evaluates to four bits. Each character in an octal literal evaluates to three bits. Thus, `0Fh` evaluates to a byte, and `00o` evaluates to a bit-string of width six.

The following operators and special characters can appear in source code.

unop ⟶ **-** | **~** | **not**
binop ⟶ ***** | **/** | **%** | **+** | **-** | **<<** | **>>** | **++** | **&** | **^** | **|** |
< | **>** | **==** | **<>** | **<=** | **>=** | **and** | **or**
special ⟶ **(** | **)** | **[** | **]** | **{** | **}** | **"** | **=** | **,** | **.** | **:**

11.2 Types

Our specification language is typed, which means that registers, expressions, function arguments and results, action arguments and results, and instruction operands have types. Note that by assigning a type to each register we implictly define the set of possible configurations of the processor core.

We distinguish simple types and complex types. A complex type consists of named subcomponents, which can be accessed using the dot operator. For example, if an expression `x` has a complex type with a subcomponent 'y', then the expression 'x.y' refers to that subcomponent. There are three kinds of complex types: records, layouts, and unions. A record is just a tuple with named subcomponents, a layout and a union are bit-strings with named substrings. Each complex type must be be defined at the top-level of the specification, and must have a name, by which it is referred to in other parts of the specification. We describe the exact syntax of complex type definition at the end of this section.

A simple type does not have named subcomponents, thus the dot operator is not applicable to values of simple types. Simple types include bit-strings, integer and natural numbers, tuples, and explictly enumerated sets of bit-strings or numbers. In specification code we will use syntax x**::***type* to denote that the x has the type *type*, where x can be a register, an expression, a function argument or an action argument. The symbol *type* stands for either an identifier (the name of a complex type), or a bit, or a bit-string, or a natural number, or an integer number, or a tuple, or a finite set, or a range of bit-strings/numbers:

type ⟶ *id* | **bit** | **bits** *pexpr* | **nat** | **int** |
type-tuple | *set-expr* | *range-expr*
type-tuple ⟶ **(** $type_1$ **,** $type_2$ **,** ... **,** $type_n$ **)**
set-expr ⟶ **{** $pexpr_1$ **,** $pexpr_2$ **,** ... **,** $pexpr_n$ **}**
range-expr ⟶ **[** *pexpr* **..** *pexpr* **]**
pexpr ⟶ *expr* (limited to bit-strings and numbers)
expr ⟶ (defined in section 11.4)

The range expression is just a shorthand for enumerating a set, for example, the range `[1 .. 3]` is the same as a set `{1, 2, 3}`. The width of a bit-string and elements of sets and ranges are primitive expressions *pexpr*. A primitive expression is an expression on

bit-strings and numbers which can be evaluated without knowing the exact values of the registers.

An example of a primitive expression is a constant expression that contains only numeric literals and operations over them. For example, the type **bits** `(2+3)` is the same as the type **bits** `5` and is the set of bit-strings of width five. The following four types are equivalent: **bit**, **bits** `1`, `{0b, 1b}`, `[0b .. 1b]`.

Unfortunately, bit-strings of constant width are not sufficient for instruction specification. Many instructions have operands with variable widths that depend on operating mode. Therefore, we have to allow non-constant primitive expressions. However, it makes type checking very difficult. We solve this problem by giving hints for the type checker using the following convention: any identifier that starts with the dollar (**$**) sign must have finite (and small) number of possible values. Then we define a primitive expression as an expression that depends only on constants or on identifiers that start with the dollar sign. This allows the type checker to enumerate all possible values of a primitive expression by enumerating all possible values of the dollar identifiers. We call dollar identifiers type parameters, and they are defined at the top-level of the source code with the following syntax.

$$parameter \longrightarrow \textbf{parameter}\ \$id\ \textbf{::}\ type\ \ (\textbf{=}\ expr)^{?}$$

The definition starts with the keyword **parameter**, followed by a dollar identifier and a primitive type reference with finite number of possible values. The optional expression $expr$ binds the parameter to a register. In other words, the identifier id becomes an alias for the expression $expr$ when the expression is present.

An example of a unbound parameter is the physical address width, which is implementation dependent and is usually 36 bits (for Intel processors) or 52 bits (for AMD processors). Thus, we can define this parameter as **parameter** `$pa::{36, 52}`.

An example of a bound parameter is the protection enable bit of the `CR0` register. Since operating mode depends on this bit, operand sizes also depend on this bit. We cannot put the `CR0.PE` directly into a type expression, therefore, we introduce the following parameter: **parameter** `$PE::`**bit** `= CR0.PE`. The type checker can now enumerate the two values of the `$PE` without enumerating 2^{64} possible values of the `CR0`.

Some parameters are interdependent. For example, long mode cannot be activated without protection being enabled. We can express such relationships using invariants, which are defined at the top-level according to the following syntax.

$$invariant \longrightarrow \textbf{invariant}\ \ pexpr$$

Thus, the invariant for the `$LMA` (long mode activated) and the `$PE` can be defined as **invariant** `not $LMA` **or** `$PE`. Invariants allow the type checker to reduce the enumeration space.

A group of related constants can be defined at the top-level as elements of a named set. Definition of a named set starts with the keyword **set** followed by the set name and the elements, which are written either as an indented block or as a comma separated list.

$$
\begin{aligned}
\mathit{set} &\longrightarrow \texttt{set}\ \mathit{id}\ (\mathit{set\text{-}elems\text{-}block} \mid \mathit{set\text{-}elems\text{-}line}) \\
\mathit{set\text{-}elems\text{-}block} &\longrightarrow \lfloor \mathit{set\text{-}elem} \rfloor^{+} \\
\mathit{set\text{-}elems\text{-}line} &\longrightarrow \texttt{= \{}\ \mathit{set\text{-}elem}_1\ \texttt{,}\ \mathit{set\text{-}elem}_2\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{set\text{-}elem}_{n>0}\ \texttt{\}} \\
\mathit{set\text{-}elem} &\longrightarrow \mathit{id}\ (\texttt{=}\ \mathit{lit})^{?}
\end{aligned}
$$

For example, exception vectors are defined as follows.

```
set Vector = {xDE = 00h, xDB = 01h, xNMI = 02h, xBP = 03h, xOF = 04h,
              xBR = 05h, xUD = 06h, xNM = 07h, xDF = 08h, xTS = 00h,
              xNP = 01h, xSS = 02h, xGP = 03h, xPF = 04h, xMF = 10h,
              xAC = 11h, xMC = 12h, xXF = 13h, xSX = 1Eh}
```

Now the identifier `xPF` can be used in any expression and always evaluates to a byte with value `04h`.

A record is a tuple with named components. A record type can be defined at the top-level by writing the keyword **`record`** followed by the record name and the indented block of the record components. For each component, its name and its type have to specified.

$$
\begin{aligned}
\mathit{record} &\longrightarrow \texttt{record}\ \mathit{id}\ \lfloor \mathit{record\text{-}field} \rfloor^{+} \\
\mathit{record\text{-}field} &\longrightarrow \texttt{field}\ \mathit{id}\texttt{::}\mathit{type}
\end{aligned}
$$

A record `X` with an integer component `i` and a byte component `b` can be defined in the following way.

```
record X
  field i::int
  field b::bits 8
```

Many registers are bit-strings with named substrings (fields). For example, control registers have named bits. We call such bit-string layouts. In order to define a layout type, we write the keyword **`layout`** followed by the layout name and an indented block of named fields. Each field must have either a bit-string type or a layout type. For example, a 16-bit segment selector has the following layout:

```
layout Selector
  field RPL::bits 2
  field TI::bit
  field index::bits 13
```

This means that the two least significant bits of a selector are called `RPL`. The next bit is called `TI` and the 13 most significant bits are called `index`.

Register fields can be reserved, read-only, or ignored. The layout syntax allows to specify such attributes, as shown in the following example:

```
layout L
  field a::bit reserved and must be 0b
  field b::bit read only
  field c::bit read only and read as 1b
  field d::bit ignored
```

Let us assume that there is a register `x` with layout `L`. When software tries to load a four-bit value `v` into the `x`, the field attributes have the following effects:

- the `v.a` must be zero. In case this condition does not hold, an exception is raised.
- the `x.b` is unchanged.
- the `x.c` is unchanged and its values is always `1b`.
- the `v.d` is loaded into the `x.d`, but the processor never uses this bit, i.e. the bit is ignored by the processor and software can load any value.

Fields can have multiple attributes, and attributes can have conditions:

```
layout L2
  field a::bit reserved and must be 0b when $PE
  field b::bit read only when $PE
               ignored when not $PE
```

In this example, the field `a` is reserved only when protection is enabled. The field `b` is read-only when protection is enabled, and is ignored otherwise.

Specifying field attributes in layout definitions allows us to mechanically generate two useful functions: one that checks values of reserved fields and another that fixes the values of read-only fields. Let `X` be a name of a layout type. Then the first function is called `isX`, it takes a layout value and returns true if and only if all reserved fields of the value are set correctly. The second function is called `fixX`, it takes the old layout value and the new layout value. The function adjust the new layout value such that its read-only fields are copied from the old value.

Besides fields, layouts can have pseudo-fields, which we call abbreviations.

```
layout L3
  field a::bit
  field b::bit
  abbr  c::bit = a or b
```

In this example, the abbreviation `c` is a pseudo-field. Let `x` be a register that has layout `L3`, then writing `x.c` is the same as writing `x.a or x.b`. Thus, an abbreviation is merely syntactic sugar.

Summarizing the above examples, we can give formal syntax for layout definitions:

$$
\begin{aligned}
\textit{layout} &\longrightarrow \textbf{layout}\ \textit{id}\ \lfloor \textit{layout-field} \rfloor^{+}\ \lfloor \textit{layout-abbr} \rfloor^{+} \\
\textit{layout-field} &\longrightarrow \textbf{field}\ \textit{id}\textbf{::}\textit{type}\ \lfloor \textit{field-attr} \rfloor^{*} \\
\textit{field-attr} &\longrightarrow \textit{readonly} \mid \textit{ignored} \mid \textit{reserved} \\
\textit{readonly} &\longrightarrow \textbf{read only}\ (\textbf{and read as}\ \textit{pexpr})^{?}\ \lfloor \textbf{when}\ \textit{pexpr} \rfloor^{?} \\
\textit{ignored} &\longrightarrow \textbf{ignored}\ \ \lfloor \textbf{when}\ \textit{pexpr} \rfloor^{?} \\
\textit{reserved} &\longrightarrow \textbf{reserved and must be}\ \textit{pexpr}\ \lfloor \textbf{when}\ \textit{pexpr} \rfloor^{?} \\
\textit{layout-abbr} &\longrightarrow \textbf{abbr}\ \textit{id}\textbf{::}\textit{type}\ \textbf{=}\ \textit{expr}
\end{aligned}
$$

There are times when we have a bit-string but do not know its exact layout. All we know is a list of possible layouts that the bit-string might have. For such cases we use a union type. In order to define a union type, we write the keyword **union** followed by the name of the union and a list of layout types or union types. The widths of layouts and unions must be the same.

$$
\begin{aligned}
\textit{union} &\longrightarrow \textbf{union}\ \textit{id}\ \lfloor \textit{union-part} \rfloor^{+} \\
\textit{union-part} &\longrightarrow (\textbf{union} \mid \textbf{layout})\ \textit{id}\ \lfloor \textbf{when}\ \textit{pexpr} \rfloor^{?}
\end{aligned}
$$

An example of a union type is a user segment descriptor, which can be either a data segment descriptor or a code segment descriptor.

```
union UserSegment
  layout DataSegment
  layout CodeSegment
```

Only common fields of the `DataSegment` and the `CodeSegment` are accessible as fields of the `UserSegment`. Let `x` be a union `UserSegment`, then we can write `x.limit` because both the `DataSegment` and the `CodeSegment` have `limit`. The expression `x.limit` is evaluated as

```
if isDataSegment(x) then (x::DataSegment).limit
else (x::CodeSegment).limit
```

where `x::DataSegment` means convert `x` to `DataSegment`.

11.3 Registers

A register is defined at the top-level by writing the keyword **`register`** followed the register name and the register type. Register names and type names are in disjoint namespaces. Thus, assuming that there is a layout type `CR3`, we can define the `CR3` register as follows.

```
register CR3::CR3
```

There are registers that are grouped into an array. For example, general-purpose registers make up an array of 16 elements. We define a register array using the following syntax.

```
register R::array bits 4 of bits 64
```

Thus, `R` is an array of bit-strings of width 64 indexed by bit-strings of width 4. The type of an array index must be primitive. Arrays are not first-class objects, i.e. functions cannot take array arguments and cannot return arrays. The only possible operation over arrays is indexing, for example `R[0010b]`.

Official manuals refer to register array elements by names instead of indices. In order to maintain consistency with the manuals, we allow aliases for array elements. For example, the register `RCX` is an alias for `R[0001b]`. We specify this fact in the following way.

```
register RCX::bits 64 alias R[0001b]
```

Summarizing the above examples, we give formal syntax for register definition:

$$\begin{aligned} \textit{register} &\longrightarrow \textbf{register}\ \textit{id}\ \textbf{::}\ \textit{register-type}\ \ (\textbf{alias}\ \textit{id}\ \texttt{[}\ \textit{lit}\ \texttt{]}\ )^{?} \\ \textit{register-type} &\longrightarrow \textit{type} \mid\ \textbf{array}\ \textit{type}\ \textbf{of}\ \textit{type} \end{aligned}$$

Besides official registers, we will define a few “hidden” registers which are not described in the manuals but necessary for instruction specification. In order to distinguish the hidden registers from local variables and official registers, we will always start a hidden register name with the underscore (_) character.

11.4 Expressions

The following production rule shows the structure of expressions:

$$
\begin{aligned}
\mathit{expr} \longrightarrow\ & \mathit{lit} \mid \mathit{id} \mid \\
& \mathit{unop}\ \ \mathit{expr} \mid \mathit{expr}\ \ \mathit{binop}\ \ \mathit{expr} \mid \\
& \mathit{paren} \mid \mathit{tuple} \mid \mathit{apply} \mid \\
& \mathit{index} \mid \mathit{dot} \mid \mathit{update} \mid \mathit{offset} \mid \\
& \mathit{zxt} \mid \mathit{sxt} \mid \mathit{bits} \mid \mathit{int} \mid \mathit{nat} \\
& \mathit{if} \mid \mathit{let} \mid \mathit{annot}
\end{aligned}
$$

Thus, an expression is

- either a literal,
- or an identifier,
- or a unary operator applied to an expression; there are three unary operators:

Operator	Operand	Result	Meaning
`-`	**`bits`** `n`	**`bits`** `n`	negation
`~`	**`bits`** `n`	**`bits`** `n`	complement
`not`	**`bit`**	**`bit`**	complement

  Thus, *unop* ⟶ **`-`** | **`~`** | **`not`**. The negation operator is also defined for integer operands.

- or a binary operator applied to two expressions; there are 19 binary operators:

Operator	Left Operand	Right Operand	Result	Meaning
`*`	**bits** n	**bits** n	**bits** n	multiplication
`/`	**bits** n	**bits** n	**bits** n	division
`%`	**bits** n	**bits** n	**bits** n	modulo
`+`	**bits** n	**bits** n	**bits** n	addition
`-`	**bits** n	**bits** n	**bits** n	subtraction
`>>`	**bits** n	**nat**	**bits** n	shift to right
`<<`	**bits** n	**nat**	**bits** n	shift to left
`++`	**bits** n	**bits** m	**bits** (n+m)	concatenation
`&`	**bits** n	**bits** n	**bits** n	bitwise and
`^`	**bits** n	**bits** n	**bits** n	bitwise xor
`|`	**bits** n	**bits** n	**bits** n	bitwise or
`<`	**bits** n	**bits** n	**bit**	less than
`>`	**bits** n	**bits** n	**bit**	greater than
`==`	any	any	**bit**	equal
`<>`	any	any	**bit**	inequal
`<=`	**bits** n	**bits** n	**bit**	less or equal
`>=`	**bits** n	**bits** n	**bit**	greater or equal
and	**bit**	**bit**	**bit**	boolean and
or	**bit**	**bit**	**bit**	boolean or

Thus,

$$\begin{aligned} \mathit{binop} \longrightarrow\ & \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{+} \mid \texttt{-} \mid \texttt{<<} \mid \texttt{>>} \mid \texttt{++} \mid \texttt{\&} \mid \texttt{\^{}} \mid \texttt{|} \mid \\ & \texttt{<} \mid \texttt{>} \mid \texttt{==} \mid \texttt{<>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{and} \mid \texttt{or} \end{aligned}$$

The comparison (`<`, `<=`, `=>`, `>`) and arithmetic operators (`*`, `/`, `%`, `+`, `-`) are also defined for naturals and integers.

- or parenthesis: $\mathit{paren} \longrightarrow$ **(** expr **)** ,
- or a tuple: $\mathit{tuple} \longrightarrow$ **(** expr_1 **,** expr_2 **,** ... **,** $\mathit{expr}_{n>1}$ **)** ,
- or function application:

$$\begin{aligned} \mathit{apply} &\longrightarrow \mathit{id}\ \ \mathit{args} \\ \mathit{args} &\longrightarrow \texttt{(}\ \mathit{expr}_1\ \texttt{,}\ \mathit{expr}_2\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{expr}_{n>0}\ \texttt{)} \end{aligned}$$

- or element selection: $\mathit{index} \longrightarrow \mathit{expr}$ **[** $(\mathit{expr} \mid \mathit{expr}$ **:** $\mathit{expr})$ **]**. In expression `x[i]`, the `x` can be either a bit-string or a register array. In the former case, the `i` is a zero-based index of the selected bit. In the latter case, the `i` is an array index. The range selection `x[i:j]` is only applicable to bit-strings and the result is a substring of bits with indices from `j` to `i`. For example `10110b[3:1] == 011b`
- or subcomponent selection: $\mathit{dot} \longrightarrow \mathit{expr}$**.**$\mathit{id}$, the expression must have scoped type and the identifier must be the name of a subcomponent.

- or subcomponent update:

  $$update \longrightarrow expr \texttt{ with } \texttt{[} upd_1 \texttt{ , } upd_2 \texttt{ , } \ldots \texttt{ , } upd_{n>0} \texttt{]}$$
  $$upd \longrightarrow id \texttt{ = } expr$$

  Expression `x with [a = 1b]` is equal to the `x` with the value of the subcomponent `a` replaced by `1b`.

- or offset computation: $offset \longrightarrow \texttt{@}id\texttt{.}id$. The left identifier must be a name of a layout and the right identifier must be a name of a field of the layout. The expression `@X.a` returns the index of the least significant bit of the field `a` in the layout `X`.

- or zero-extension: $zxt \longrightarrow \texttt{zxt (} pexpr \texttt{ , } expr \texttt{)}$. The first argument must be a natural number. The second argument must be a bit-string. The expression `zxt(n, x)` is zero-extension of the `x` to `n` bits. The width of the `x` must not exceed `n`. Thus, `zxt(4, 10b) == 0010b`.

- or sign-extension: $sxt \longrightarrow \texttt{sxt (} pexpr \texttt{ , } expr \texttt{)}$. The arguments must satisfy the same conditions as the arguments of zero-extension. The expression `sxt(n, x)` is sign-extension of the `x` to `n` bits, where the sign is the most significant bit of the `x`. Thus, `sxt(4, 10b) == 1110b` and `sxt(4, 01b) == 0001b`.

- or conversion to a bit-string: $bits \longrightarrow \texttt{bits (} pexpr \texttt{ , } expr \texttt{)}$. The first arguments must a natural number. The second argument can be either an integer or a natural number. The expression `bits(n, a)` is the result of two's complement conversion of the number `a` to a bit-string of width `n`. Thus, `bits(4, 3) == 0011b` and `bits(4, -1) == 1111b`.

- or conversion to an integer: $int \longrightarrow \texttt{int (} expr \texttt{)}$. The argument must be a bit-string. The expression `int(x)` is the result of two's complement conversion of the bit-string `x` to an integer number. Thus, `int(1111b) == -1` and `int(0111b) == 7`.

- or conversion to a natural: $nat \longrightarrow \texttt{nat (} expr \texttt{)}$. The argument must be a bit-string. The expression `nat(x)` is the result of unsigned conversion of the bit-string `x` to a natural number. Thus, `nat(1111b) == 15`.

- or a conditional: $if \longrightarrow \texttt{if } expr \texttt{ then } expr \ (\texttt{elif } expr \texttt{ then } expr)^* \texttt{ else } expr$, where `elif` is a short version of `else if`.

- or "let" expression:

  $$let \longrightarrow \texttt{let } bind \texttt{ = } expr \texttt{ in } expr$$
  $$bind \longrightarrow id \mid \texttt{(} bind_1 \texttt{ , } bind_2 \texttt{ , } \ldots \texttt{ , } bind_{n>1} \texttt{)}$$

- or type annotation: $annot \longrightarrow expr\texttt{::}type$. The type of the expression `expr` must be compatible with the specified type. Integer and natural number are compatible. Bit-strings, layouts and unions are compatible if and only if they have the same width. The expression `x::T` is equal to the `x` converted to the type `T`.

11.5 Functions

Functions are named and parameterized expressions. Functions can be defined at the top-level using the following syntax.

$$
\begin{aligned}
\textit{function} &\longrightarrow \texttt{function}\ \textit{id}\ \ \textit{decl-args}^{?}\ \ (\texttt{::}\textit{type})^{?}\ \ (\texttt{when}\ \textit{pexpr})^{?}\ \ \texttt{=}\ \textit{expr} \\
\textit{decl-args} &\longrightarrow \texttt{(}\ \textit{arg}_1\ \texttt{,}\ \textit{arg}_2\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{arg}_{n>0}\ \texttt{)} \\
\textit{arg} &\longrightarrow \textit{id}\texttt{::}\textit{type}
\end{aligned}
$$

Function definition starts with the keyword **function**, which is followed by the function name. The argument list, the result type, and the condition are optional. If the result type is not specified, it is inferred from the type of the function body. It is possible to defined multiple functions with the same name as long as their conditions do not overlap. As an example, we define a few useful functions.

```
function max($a::[0..128], $b::[0..128])::[0..128]
  = if $a < $b then $b else $a
function min($a::[0..128], $b::[0..128])::[0..128]
  = if $a < $b then $a else $b
function zero($n::[1..128])::bits $n = bits($n, 0)
function one($n::[1..128])::bits $n = bits($n, 1)
function ones($n::[1..128])::bits $n = sxt($n, 1b)
```

11.6 Actions

Actions are named and parameterized lists of statements. Definition of an action is similar to definition of a function, except the keyword **action** is used and the action body is an indented block of statements. Another difference is that when the result type is not specified, the action returns nothing.

$$
\begin{aligned}
\textit{action} &\longrightarrow \texttt{action}\ \ \textit{decl-args}^{?}\ \ (\texttt{::}\textit{type})^{?}\ \ \lfloor\texttt{when}\ \textit{pexpr}\rfloor^{?}\ \ \textit{stmt}^{+} \\
\textit{stmt} &\longrightarrow \textit{if-stmt} \mid \lfloor\textit{let-stmt}\rfloor \mid \lfloor\textit{call-stmt}\rfloor \mid \lfloor\textit{return-stmt}\rfloor \mid \lfloor\textit{fail-stmt}\rfloor \mid \\
&\qquad \lfloor\textit{write-stmt}\rfloor \mid \lfloor\textit{undef-stmt}\rfloor \mid \lfloor\textit{chain-stmt}\rfloor \mid \lfloor\textit{assume-stmt}\rfloor
\end{aligned}
$$

We discussed the statements at the beginning of this chapter. Here we give formal syntax and highlight the omitted details.

The “let” statement can bind an identifier or a tuple of identifiers.

$$
\begin{aligned}
\textit{let-stmt} &\longrightarrow \texttt{let}\ \ \textit{bind}\ \ \texttt{=}\ \ \textit{expr} \\
\textit{bind} &\longrightarrow \textit{id} \mid \texttt{(}\ \textit{bind}_1\ \texttt{,}\ \textit{bind}_2\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{bind}_{n>1}\ \texttt{)}
\end{aligned}
$$

The conditional statement has optional **elif** and **else** parts.

$$
\begin{aligned}
\textit{if-stmt} \longrightarrow\ & \lfloor\texttt{if}\ \ \textit{expr}\ \ \texttt{then}\ \ \textit{stmt}^{+}\rfloor \\
& \lfloor\texttt{elif}\ \ \textit{expr}\ \ \texttt{then}\ \ \textit{stmt}^{+}\rfloor^{*} \\
& \lfloor\texttt{else}\ \ \textit{stmt}^{+}\rfloor^{?}
\end{aligned}
$$

Like the “let” statement, the “call” statement can bind an identifier or a tuple to the result of the action execution.

call-stmt ⟶ **`call`** (*bind* **`=`**)$^{?}$ *id* *args*$^{?}$

The expression parameter of the “return” statement is optional. When the expression is not given, the action returns nothing.

return-stmt ⟶ **`return`** *expr*$^{?}$

The “fail” statement has optional condition. When the condition is present and does not hold, the statement is skipped.

fail-stmt ⟶ **`fail`** *expr* (**`when`** *expr*)$^{?}$

The “write” and “undef” statements have optional conditions that act like the condition of the “fail” statement. The destination of the statements is not limited to a register name, but can be a part of a register. The *wexpr* symbol denotes an expression that selects a writeable part of a register, i.e. a part of a register that is reachable via a sequence of dot and index operators. For example, `SR[000b].desc[3:0]`.

write-stmt ⟶ **`write`** *expr* **`to`** *wexpr* (**`when`** *expr*)$^{?}$
undef-stmt ⟶ **`undef`** *wexpr* (**`when`** *expr*)$^{?}$

The “chain” and “assume” statements have the following syntax.

chain-stmt ⟶ (**`commit and`**)$^{?}$ **`chain`** *id*
assume-stmt ⟶ **`assume`** *pexpr*

11.7 Instructions

Instruction definition starts with the keyword **`instruction`**, which is followed by the instruction mnemonic. If the instruction is valid only in some operating modes, then this fact can be specified using the optional condition after the mnemonic. Afterwards, there is a list of opcodes, a list of attributes, and a semantic action, which specifies how to execute the instruction.

instruction ⟶ **`instruction`** *mnemonic* ⌊**`when`** *pexpr*⌋$^{?}$
⌊*opcode*⌋$^{+}$
⌊*attribute*⌋*
⌊*execute*⌋$^{?}$
mnemonic ⟶ **`"`**(any string except double quotes and newline)**`"`**

Instruction attributes specify whether the instruction has 64-bit default operand width or not, whether the instruction is serializing or not, whether the instruction can have a lock prefix or not.

attribute ⟶ **`default64`** | **`serializing`** | **`lockable`**

A semanic action is a statement in our domain-specific language:

$$execute \longrightarrow \textbf{execute}\ \ stmt$$

The statement can use special identifiers `op1, op2, op3`, `op1' ,op2'`, `op3'`, which represent the values of the operands before and after execution of the instruction. The widths of the operands are denoted as special identifiers `$n, $n1, $n2, $n3`, where `$n = $n1`. Chapter {instructions} explains semantic actions in more detail.

For each opcode we specify a bit-string pattern, an optional condition (if the opcode is valid only in some operating modes), a list of operands, a list of attributes, and an optional statement for execution. If the attributes/the statement are present, they override the attributes/the statement defined on the instruction level. For each operand we specify its type and its width.

$$\begin{aligned} opcode \longrightarrow\ & \textbf{opcode}\ \ pattern\ \ \lfloor \textbf{when}\ \ pexpr \rfloor^{?} \\ & \lfloor operand \rfloor^{*} \\ & \lfloor attribute \rfloor^{*} \\ & \lfloor execute \rfloor^{?} \\ operand \longrightarrow\ & \textbf{operand}\ \ expr\ \ pexpr \end{aligned}$$

During instruction fetch, the opcode pattern is matched against the fetched bytes. The pattern consists of one or more bit-strings followed by optional ModRM byte specifiers. The last bit-string before the ModRM byte specifiers is matched against the fetched opcode. All preceding bit-strings are matched against the fetches prefixes. The ModRM byte specifiers are matched against the fetched ModRM byte.

$$\begin{aligned} pattern &\longrightarrow \texttt{"}\ bitstring^{+}\ \ reg^{?}\ \ mod^{?}\ \ rm^{?}\ \texttt{"} \\ reg &\longrightarrow \texttt{/}\ bitstring \\ mod &\longrightarrow \textbf{mod}\ \ bitstring \\ rm &\longrightarrow \textbf{rm}\ \ bitstring \end{aligned}$$

The ModRM byte is described in chapter {instruction-fetch-and-decode}.

As an example, we give the definition of the `MOV` instruction (first two opcodes):

```
instruction "MOV"
  opcode "88h"
    operand reg_mem 8
    operand reg 8
    execute let op1' = op2
  opcode "89h"
    operand reg_mem $v
    operand reg $v
    execute let op1' = op2
```

Since putting each operand in a separate line takes too much space, in this document we will give condensed definitions:

```
instruction "MOV"
    opcode "88h" reg_mem 8, reg 8     : let op1' = op2
    opcode "89h" reg_mem $v, reg $v   : let op1' = op2
```

Chapter {instructions} describes instruction specification notation in more detail.

CHAPTER

TWELVE

REGISTERS

12.1 General-Purpose Registers

There are 16 general-purpose registers. Some instructions access a register partially, e.g. bits `[15:8]` may be updated, while the remaining bits remain intact. The architecture defines aliases for these registers and their parts. We use aliases only for the whole registers, and refer to the parts by ordinary bit selection.

```
register R::array bits 04 of bits 64
register RAX::bits 64 alias R[0000b]
register RCX::bits 64 alias R[0001b]
register RDX::bits 64 alias R[0010b]
register RBX::bits 64 alias R[0011b]
register RSP::bits 64 alias R[0100b]
register RBP::bits 64 alias R[0101b]
register RSI::bits 64 alias R[0110b]
register RDI::bits 64 alias R[0111b]
register R8::bits 64 alias R[1000b]
register R9::bits 64 alias R[1001b]
register R10::bits 64 alias R[1010b]
register R11::bits 64 alias R[1011b]
register R12::bits 64 alias R[1100b]
register R13::bits 64 alias R[1101b]
register R14::bits 64 alias R[1110b]
register R15::bits 64 alias R[1111b]
```

12.2 Control Registers

The control register array has 16 elements, however, only 5 of them are defined by the architecture. We use the following aliases for them: `CR0`, `CR2`, `CR3`, `CR4`, and `CR8`. Access to undefined registers generates an exception.

```
register CR::array bits 04 of bits 64
```

The CR0 register controls operating mode, memory and FPU features.

```
register CR0::CR0 alias CR[0000b]
layout CR0
  field PE::bit
  field MP::bit
  field EM::bit
  field TS::bit
  field ET::bit read only and read as 1b
  field NE::bit
  field rsv1::bits 10 reserved and must be bits(10, 0)
  field WP::bit
  field ign::bit ignored
  field AM::bit
  field rsv2::bits 10 reserved and must be bits(10, 0)
  field NW::bit ignored
  field CD::bit
  field PG::bit reserved and must be 0b when not $PE
  field rsv3::bits 32 reserved and must be bits(32, 0)
```

PE (protected mode enable) When the bit is set, the processor uses the segmentation-protection mechanism.

MP (monitor coprocessor) When the bit is set, executing the WAIT/FWAIT instruction after a task switch (i.e CR0.TS==1b) raises an xNM exception.

EM (emulate coprocessor) When the bit is set, executing an FPU instruction raises an xNM exception and executing a media instruction raises an xUD exception.

TS (task switched) The bit indicates whether a task switch was performed since the last time the bit was cleared. When the bit is set, executing any FPU or media instruction raises xNM exception.

ET (extension type) This legacy bit used to indicate whether FPU instructions are supported.

NE (numeric error) The bit controls how FPU errors are handled. When the bit is set, the processor uses internal handling mechanism. Otherwise, it forwards errors to an external device using FERR, IGNNE signals

WP (write protect) When the bit is set, all write accesses to write-protected pages produce an exception. When the bit is cleared, supervisor software (CPL==00b) can write to write-protected pages.

AM (alignment mask) When the bit is set and RFLAGS.AC==1b, then alignment checking is enabled, which means that unaligned memory accesses from user software (CPL==11b) raise an xAC exception.

NW (not writethrough) This legacy bit is ignored

CD (cache disable) When the bit is set, the caches are disabled.

PG (paging) When the bit is set, paging is enabled.

If a page fault occurs, then the faulting virtual address is stored in the CR2 register.

```
register CR2::bits 64 alias CR[0010b]
```

The CR3 register contains the base physical address of the top-level page table. The meaning of the some fields of this register depends on the $long_mode, $PAE, $pa parameters, which are defined in chapter 13.

```
register CR3::CR3 alias CR[0011b]
layout CR3
  field rsv1::bits 3 reserved and must be 000b
  field PWT::bit
  field PCD::bit
  field base0::bits 7
    reserved and must be 0000000b when $long_mode or not $PAE
  field base1::bits 20
  field base2::bits 20
    reserved and must be bits(20, 0) when $legacy_mode
  field rsv2::bits 12 reserved and must be 000h
  abbr base::bits $pa = if $long_mode then base2++base1++000h
                        elif $PAE then zxt($pa, base1)++base0++00000b
                        else zxt($pa, base1)++000h
```

PWT (page writethrough) When the bit is set, the top-level page table has write-through caching policy. Otherwise, the caching policy is writeback.

PCD (page cache disable) When the bit is set, the top-level page table is uncacheable. Otherwise, it is cacheable.

base0 the top-level page table base address (bits 11:5)

base1 the top-level page table base address (bits 31:12)

base2 the top-level page table base address (bits 51:32)

base page table base address

Bits in the CR4 register control model-specific features.

```
register CR4::CR4 alias CR[0100b]
layout CR4
  field VME::bit
  field PVI::bit
  field TSD::bit
  field DE::bit
  field PSE::bit ignored when $PAE
  field PAE::bit
  field MCE::bit
  field PGE::bit
  field PCE::bit
  field OSFXSR::bit
  field OSXMMEXCPT::bit
  field rsv::bits 53 reserved and must be bits(53, 0)
```

VME (virtual 8086-mode extensions) When the bit is set, the processor in virtual 8086 mode virtualizes the RFLAGS.IF using RFLAGS.VIF and RFLAGS.VIP and intercepts software interrupts using intercept bitmap in the task state segment. The RFLAGS register is defined later in this section.

PVI (protected mode virtual interrupts) When the bit is set, the processor in protected mode virtualizes the RFLAGS.IF using RFLAGS.VIF and RFLAGS.VIP.

TSD (time stamp disable) When the bit is set, only supervisor software (CPL==00b) can execute the RDTSC and RDTCP instructions. Otherwise, user software is also allowed to execute these instructions.

DE (debugging extensions) The bit enables I/O breakpoints and makes debug registers DR4 and DR5 reserved. When the bit is cleared, the registers DR4, DR5 are aliased with DR6, DR7.

PSE (page size extensions) The bit enables 4MB pages in legacy mode. When the bit is cleared, only 4KB pages are allowed. The bit is ignored when PAE is active.

PAE (physical address extensions) The bit controls the size of page table entries. When the bit is set, a page table entry is 8 bytes wide. Otherwise, it is 4 bytes wide. The bit also enables 2MB pages.

MCE (machine check enable) When the bit is set, a machine check error raises an xMC exception.

PGE (page-global enable) The bit enables global paging translations, which are not affected by local TLB flushes. The G bit in a leaf page table entry indicates whether a translation is global or local.

PCE (performance-monitoring counter enable) When the bit is set, only supervisor software (CPL==00b) can execute the RDPMC instruction. Otherwise, user software is also allowed to execute this instruction.

OSFXSR (operating system FXSAVE/FXRSTOR support) When the bit is set, software can use 128-bit media instructions and the FXSAVE/FXRSTOR instructions save and restore state for the FPU, 64-bit and 128-bit media instructions.

OSXMMEXCPT (operating system unmasked exception support) When the bit is set, unmasked 128-bit SIMD media errors produce an xXF exception. Otherwise, they produce an xUD exception.

The CR8 register is aliased with the TPR register of the local APIC.

```
register CR8::CR8 alias CR[1000b]
layout CR8
  field TPR::bits 32
  field rsv::bits 32 reserved and must be bits(32, 0)
```

TPR (task priority register alias) See the description of the TPR register.

The extended feature enable register EFER is a model-specific register. It controls operating mode and enables virtualization, fast system calls, and other features.

```
register EFER::EFER alias MSR[C0000080h]
layout EFER
  field SCE::bit
  field rsv1::bits 7 ignored and read as 0000000b
  field LME::bit
  field rsv2::bit reserved and must be 0b
  field LMA::bit read only
  field NXE::bit
  field SVME::bit
  field rsv3::bit reserved and must be 0b
  field FFXR::bit
  field rsv4::bits 49 reserved and must be bits(49, 0)
```

SCE (system call extensions) The bit enables the SYSCALL/SYSRET instructions.

LME (long mode enable) When the bit is set and `LMA==0`, the processor waits until paging is enabled (`CR0.PG==1`) and then activates long mode (sets `LMA`).

LMA (long mode activated) When the bit is set, the processor is executing in long mode.

NXE no-execute enable When the bit is set, instruction fetch from a data page produces a page fault. Otherwise, no checks are performed.

SVME (secure virtual machine enable) The bit enables virtualization instructions `VMRUN`, `VMLOAD`, `VMSAVE`, etc. When the bit is cleared, these instructions raise an `xUD` exception.

FFXR (fast `FXSAVE/FXRSTOR`) When the bit is set and `CPL==00b`, the `FXSAVE/FXRSTOR` instructions in 64-bit mode do not save and restore the `XMM` registers.

The `RFLAGS` register stores various flags and control bits. Flags `CF`, `PF`, `AF`, `ZF`, `SF`, `OF` are called status flags. Arithmetic-logical instructions update the status flags, according to the result of the operation. Conditional control transfers are performed based on the values of the status flags.

```
register RFLAGS::Flags
layout Flags
  field CF::bit
  field ro1::bit read only and read as 1b
  field PF::bit
  field ro2::bit read only and read as 0b
  field AF::bit
  field ro3::bit read only and read as 0b
  field ZF::bit
  field SF::bit
  field TF::bit
  field IF::bit
  field DF::bit
  field OF::bit
  field IOPL::bits 2
  field NT::bit
  field ro4::bit read only and read as 0b
  field RF::bit
  field VM::bit reserved and must be 0b when $long_mode
  field AC::bit
  field VIF::bit
  field VIP::bit
  field ID::bit
  field ro5::bits 42 read only and read as zero(42)
```

CF (carry flag) The carry bit of an unsigned operation.

PF (parity flag) The parity bit is set to one only when there are even number of set bits in the least-significant byte in the result of the last arithmetic-logic operation.

AF (auxiliary flag) The carry bit from the 4 least significant bits of the operands.

ZF (zero flag) The bit is set if the result of the last operation is zero.

SF (sign flag) The bit is set if the result of the last operation is negative.

TF (trap flag) When the bit is set, single step debug mode is enabled, which means that after each instruction execution, `xDB` debug exception is raised.

IF (interrupt flag) When the bit is cleared, the processor does not accept maskable interrupts for handling.

DF (direction flag) When the bit is set, string instructions decrement the `RDI/RSI` string pointers. Otherwise, the pointers are incremented.

OF (overflow flag) The overflow bit of a signed operation.

IOPL (I/O privilege level) The field defines the privilege level that is required to execute the I/O instructions. Software can execute these instructions only when `IOPL <= CPL`. Otherwise, `xGP` exception is raised.

NT (nested task) The bit indicates whether the current tast is nested in another task. The `IRET` instruction uses the bit to perform a correct return from the interrupt handler.

RF (resume flag) When the bit is set, instruction breakpoint does not raise `xDB` exception.

VM (virtual 8086 mode) The bit indicates that the processor is executing in virtual 8086 mode.

AC (alignment check) When the bit is set and `CR0.AM==1b`, then alignment checking is enabled, which means that unaligned memory accesses from user software (`CPL==11b`) raise `xAC` exception.

VIF (virtual interrupt) When `IF` virtualization is active, instructions modify this bit instead of the `IF` bit.

VIP (virtual interrupt pending) When the bit is set and `IF` virtualization is active, setting `VIF` generates `xGP` exception.

ID (CPUID present flag) The ability of software to modify this bit indicates that the processor supports the CPUID instruction. In other words, the `CPUID` instruction is supported if after negating this bit software reads the new value instead of the old value.

Clearing the global interrupt flag `GIF` masks all external events (including non-maskable interrupts).

```
register GIF::bit
```

After the `POP SS` and a `MOV SS` instructions all external interrupts and debug traps are inhibited until the next instruction is completed. It is necessary to allow to adjust the stack pointer in the next instruction and to keep the stack consistent. We use the hidden `_intr_shadow` register to indicate whether or not to inhibit interrupts after executing the current instruction.

```
register _intr_shadow::bit
```

12.3 Segment Registers

There are six segment registers: one code segment register (`CS`), one stack segment register (`SS`), and four data segment registers (`DS, ES, FS, GS`). Access to undefined registers in the `SR` array produces an `xUD` exception.

```
register SR::array bits 03 of SR
register ES::SR alias SR[000b]
register CS::SR alias SR[001b]
register SS::SR alias SR[010b]
register DS::SR alias SR[011b]
register FS::SR alias SR[100b]
register GS::SR alias SR[101b]
```

A segment register has a software visible part and a hidden part. The visible part is 16 bits wide and is called a selector. The selector is used as an index in the descriptor table to locate the segment descriptor, which is then unpacked and loaded into the hidden part of the segment register. The hidden part functions as a small cache, but it is not kept consistent with the descriptor table. This means that if the descriptor table is updated, the hidden part of the segment register is not reloaded automatically.

```
record SR
  field sel::Selector
  field attr::UserSegmentAttr
  field limit::bits 32
  field base::bits 64
```

sel This field contains, a selector which is used as an index in the descriptor table. See section 13.9 for the definition of selectors.

attr (attributes) This field is extracted from the descriptor and defines segment attributes, such as readable, writeable, etc. For more information, refer to the definition of `UserSegmentAttr` in section 13.7.

limit (segment limit) This field is extracted from the descriptor and defines either the upper or the lower limit of the segment, depending on `attr.expand_down`. For more information, refer to the definition of `UserSegment` in section 13.7.

base (segment linear base address) This field is extracted from the descriptor and defines the linear base address of the segment. In 64-bit mode, this field is assumed to be zero for all segment registers except `FS` and `GS`.

Software can access `FS.base` and `GS.base` in 64-bit mode via the `FSBase` and `GSBase` model-specific registers.

```
register FSBase::bits 64 alias MSR[C0000100h]
register GSBase::bits 64 alias MSR[C0000101h]
```

Another way to access `GS.base` in 64-bit mode is to use the `SWAPGS` instruction, which swaps `GS.base` with the `KernelGSBase` model-specific register.

```
register KernelGSBase::bits 64 alias MSR[C0000102h]
```

Software cannot directly modify the `CS` register, instead, it uses far control transfer instructions. Transfer to a different code segment may change the current privilege level, which is stored in the `CPL` register.

```
register CPL::bits 2
```

12.4 Descriptor Table Registers

A descriptor table register specifies the location of the descriptor table in the linear address space (the address space before paging translation). There are three descriptor table registers: the global descriptor table register, the local descriptor table register, and the interrupt descriptor table register.

```
register GDTR::DTR
register LDTR::LDTR
register IDTR::DTR
```

The `GDTR` and `LDTR` registers have two components: the table base address and the table limit. The limit is the offset of the last valid byte in the table.

```
record DTR
  field limit::bits 16
  field base::bits 64
```

The local descriptor table is defined by a descriptor in the global descriptor table. The `LDTR` registers stores the selector for the descriptor and the unpacked components of the descriptor.

```
record LDTR
  field sel::Selector
  field attr::LDTAttr
  field limit::bits 32
  field base::bits 64
```

12.5 Task Register

The task register `TR` specifies the current Task State Segment (TSS), which stores the task context, I/O permission bitmap, inner level stacks. Software can modify only `TR.sel`, other parts of the register are loaded by hardware from the global descriptor table.

```
register TR::TR
record TR
  field sel::Selector
  field attr::TSSAttr
  field limit::bits 32
  field base::bits 64
```

12.6 Virtualization Registers

When the secure virtual machine extensions are enabled (`EFER.SVME==1b`), the processor runs either in host mode or in guest mode. Since the architecture does not define any register to store the current virtualization mode, we add a hidden boolean register.

```
register _guest::bit
```

A set bit in the register means that the processor is running in guest mode. We introduce parameters for the SVME and _guest bits, so that we can use them in type expressions.

```
parameter $SVME::bit = EFER.SVME
parameter $GUEST::bit = _guest
function $HOST = not $GUEST
invariant not $GUEST or $SVME
```

Guest mode can be activated only by the VMRUN instruction, which saves the host registers in the memory area defined by the VM_HSAVE_PA register and loads the guest registers from the virtual machine control block (VMCB) defined by the RAX register. The VM_HSAVE_PA stores a page aligned physical address of the host state save area.

```
register VM_HSAVE_PA::bits 64 alias MSR[C0010116h]
```

Before loading the guest state, the VMRUN instruction must save somewhere the physical address of the VMCB. Otherwise, the processor would not know where to save the guest context on VMEXIT. The architecture does not specify such a register, therefore, we add a hidden register _vmcb_addr, which store a page aligned physical address of the VMCB.

```
register _vmcb_addr::bits 64
```

Along with the guest registers the VMRUN instruction loads control bits from the control area of the VMCB. We use a hidden register _vmcb_ca to store these control bits. For more information, refer to definition of VMCB_CA.

```
register _vmcb_ca::VMCB_CA
```

12.7 Instruction Registers

During instruction execution, the instruction pointer register RIP contains the address of the next instruction. Software can observe this register using the CALL instruction, which saves the RIP register in the stack. During instruction fetch and decode, the register contains the address of the current instruction.

```
register RIP::bits 64
```

Although it is not defined in the architecture, we add a hidden register _old_RIP, which stores the address of the current instruction during instruction execution. During instruction fetch and decode, _old_RIP == RIP. Instructions with the REP prefix a repeated by setting RIP to _old_RIP.

```
register _old_RIP::bits 64
```

The length of an instruction cannot exceed 15 bytes as stated in the official manuals. Instruction fetch raises an exception if the length exceeds 15 bytes.

```
register _instr_len::bits 4
```

The decoded components of the instruction are stored in the following hidden registers:

```
register _prefix::Prefix
register _opcode::Opcode
```

```
register _modrm::ModRM
register _sib::SIB
register _disp::bits 64
register _imm::bits 64
```

In case the instruction does not have some component, the corresponding register is filled with zeros. The `_disp` register stores the sign-extended displacement. The `_imm` register stores the zero-extended immediate operand. For more information on instruction format, refer to chapter [Instruction Fetch and Decode]

12.8 Memory Type Registers

As we defined in section 5.2, six memory types exist:

```
set MemType = {UC = 000b, WC = 001b, WP = 101b, WT = 100b,
               WB = 110b, CD}
```

All memory types except the 'cache-disable' (`CD`) type have 3-bit binary encodings. The `CD` memory type is assigned to a memory access with cacheable memory types when caching is disabled (`CR0.CD=1b`). Other memory types are assigned based on the linear address and the physical address of a memory access. We compute the memory type for the linear address, the memory type for the physical address, and then combine both memory types to get the memory type of the memory access.

Paging translation of a linear address produces a physical address and an index to the Page Attribute Table (PAT). The table consists of eight elements and is stored in the `PAT` register. Each element is a 3-bit binary encoding of a memory type.

```
register PAT::PAT alias MSR[00000277h]
layout PAT
  field memType0::bits 3
  field rsv0::bits 5 reserved and must be 00000b
  field memType1::bits 3
  field rsv1::bits 5 reserved and must be 00000b
  field memType2::bits 3
  field rsv2::bits 5 reserved and must be 00000b
  field memType3::bits 3
  field rsv3::bits 5 reserved and must be 00000b
  field memType4::bits 3
  field rsv4::bits 5 reserved and must be 00000b
  field memType5::bits 3
  field rsv5::bits 5 reserved and must be 00000b
  field memType6::bits 3
  field rsv6::bits 5 reserved and must be 00000b
  field memType7::bits 3
  field rsv7::bits 5 reserved and must be 00000b
```

Given an index to the PAT, the following function returns the corresponding memory type:

```
function pat_lookup(pat_idx::bits 3)::MemType
 = if pat_idx == 000b then PAT.memType0
   elif pat_idx == 001b then PAT.memType1
```

```
    elif pat_idx == 010b then PAT.memType2
    elif pat_idx == 011b then PAT.memType3
    elif pat_idx == 100b then PAT.memType4
    elif pat_idx == 101b then PAT.memType5
    elif pat_idx == 110b then PAT.memType6
    else PAT.memType7
```

The memory type of a physical address is computed based on two sets of memory range registers: variable-range registers and fixed-range registers. These registers map a range of physical addressed to a memory type. A range can have either a variable length or a fixed length.

Variable-range registers come in pairs. The first register in a pair specifies the base address of the range and the memory type. The second register in a pair specifies the length of the range via a binary mask:

```
layout MTRRphysBase
  field type::bits 3
  field rsv1::bits 9 reserved and must be 0
  field base::bits 40
  field rsv2::bits 12 reserved and must be 0
```

type is the memory type of the range.

base is the page-aligned physical base address of the range.

```
layout MTRRphysMask
  field rsv1::bits 11 reserved and must be 0
  field valid::bit
  field mask::bits 40
  field rsv2::bits 12 reserved and must be 0
```

valid indicates whether the range should be used for memory type computation or not.

mask is the page-aligned mask of the range. A page-aligned physical address `x` belongs to the range if masking the `x` produces the masked base address of the range: `(x & mask) == (base & mask)`.

Given a pair of variable-range registers, the default memory type and a physical address, the following function computes the memory type of the physical address:

```
function mtrr_var(b::MTRRphysBase,
                  m::MTRRphysMask,
                  def::MemType,
                  x::bits $pa)::MemType
 = if not m.valid then def
   elif (zxt(52,x)[52:12] & m.mask) == (b.base & m.mask) then
        combine_mt(b.type, def)
   else def
```

The function returns the default type if the range is invalid or the range does not cover the given physical address. Otherwise, the function combines the range memory type and the default memory type as follows:

```
function combine_mt(t1::MemType, t2::MemType)::MemType
 = if t1 == t2 then t1
   elif t1 == UC or t2 == UC then UC
   elif t1 == WT and t2 == WB or t1 == WB and t2 == WT then WT
   else undefined(3)
```

Eight pairs variable-range registers exist:

```
register MTRRphysBase0::MTRRphysBase alias MSR[00000200h]
register MTRRphysMask0::MTRRphysMask alias MSR[00000201h]
register MTRRphysBase1::MTRRphysBase alias MSR[00000202h]
register MTRRphysMask1::MTRRphysMask alias MSR[00000203h]
register MTRRphysBase2::MTRRphysBase alias MSR[00000204h]
register MTRRphysMask2::MTRRphysMask alias MSR[00000205h]
register MTRRphysBase3::MTRRphysBase alias MSR[00000206h]
register MTRRphysMask3::MTRRphysMask alias MSR[00000207h]
register MTRRphysBase4::MTRRphysBase alias MSR[00000208h]
register MTRRphysMask4::MTRRphysMask alias MSR[00000209h]
register MTRRphysBase5::MTRRphysBase alias MSR[0000020Ah]
register MTRRphysMask5::MTRRphysMask alias MSR[0000020Bh]
register MTRRphysBase6::MTRRphysBase alias MSR[0000020Ch]
register MTRRphysMask6::MTRRphysMask alias MSR[0000020Dh]
register MTRRphysBase7::MTRRphysBase alias MSR[0000020Eh]
register MTRRphysMask7::MTRRphysMask alias MSR[0000020Fh]
```

By chaining the `mttr_var` functions, we can take into accout all the variable-range registers:

```
function mtrr_vars(def_type::MemType, x::bits $pa)::MemType
 = let t0 = mtrr_var(MTRRphysBase0, MTRRphysMask0, def_type, x) in
   let t1 = mtrr_var(MTRRphysBase1, MTRRphysMask1, t0, x) in
   let t2 = mtrr_var(MTRRphysBase1, MTRRphysMask1, t1, x) in
   let t3 = mtrr_var(MTRRphysBase1, MTRRphysMask1, t2, x) in
   let t4 = mtrr_var(MTRRphysBase1, MTRRphysMask1, t3, x) in
   let t5 = mtrr_var(MTRRphysBase1, MTRRphysMask1, t4, x) in
   let t6 = mtrr_var(MTRRphysBase1, MTRRphysMask1, t5, x) in
   let t7 = mtrr_var(MTRRphysBase1, MTRRphysMask1, t6, x) in
   t7
```

Fixed-range memory type registers have the base addresses and the lengths of the ranges encoded in the register names with `MTRRfix[size]K_[base]` pattern:

```
register MTRRfix64K_00000::MTRRfix alias MSR[00000250h]
register MTRRfix16K_80000::MTRRfix alias MSR[00000258h]
register MTRRfix16K_A0000::MTRRfix alias MSR[00000259h]
register MTRRfix4K_C0000::MTRRfix alias MSR[00000268h]
register MTRRfix4K_C8000::MTRRfix alias MSR[00000269h]
register MTRRfix4K_D0000::MTRRfix alias MSR[0000026Ah]
register MTRRfix4K_D8000::MTRRfix alias MSR[0000026Bh]
register MTRRfix4K_E0000::MTRRfix alias MSR[0000026Ch]
register MTRRfix4K_E8000::MTRRfix alias MSR[0000026Dh]
register MTRRfix4K_F0000::MTRRfix alias MSR[0000026Eh]
register MTRRfix4K_F8000::MTRRfix alias MSR[0000026Fh]
```

Each register specifies the memory type for eight fixed consecutive ranges. The register name contains the base address of the first range. A fixed-range register is partitioned into eight bytes:

```
layout MTRRfix
  field range0::bits 8
  field range1::bits 8
  field range2::bits 8
  field range3::bits 8
  field range4::bits 8
  field range5::bits 8
  field range6::bits 8
  field range7::bits 8
```

The three least significant bits of each byte encode a memory type:

```
layout MTRRfixTypeExtended
  field type::bits 3
  field rsv::bits 5 reserved and must be 00000b
```

Given a physical address, the following function checks if the address is covered by any fixed range:

```
function mtrr_fix_cover(x::bits $pa)::bit
  = (x < zxt($pa, 1000000h))
```

Given a physical address that is covered by a fixed range, the following function returns the memory type of the covering fixed range:

```
function mtrr_fix(addr::bits $pa)::bit
  = let x = nat(addr) in
    if x<nat(80000h) then lookup(MTRRfix64K_00000, x/65536)
    elif x<nat(A0000h) then lookup(MTRRfix16K_80000, (x-nat(80000h))/16384)
    elif x<nat(C0000h) then lookup(MTRRfix16K_A0000, (x-nat(A0000h))/16384)
    elif x<nat(C8000h) then lookup(MTRRfix4K_C0000, (x-nat(C0000h))/4096)
    elif x<nat(D0000h) then lookup(MTRRfix4K_C8000, (x-nat(C8000h))/4096)
    elif x<nat(D8000h) then lookup(MTRRfix4K_D0000, (x-nat(D0000h))/4096)
    elif x<nat(E0000h) then lookup(MTRRfix4K_D8000, (x-nat(D8000h))/4096)
    elif x<nat(E8000h) then lookup(MTRRfix4K_E0000, (x-nat(E0000h))/4096)
    elif x<nat(F0000h) then lookup(MTRRfix4K_E8000, (x-nat(E8000h))/4096)
    elif x<nat(F8000h) then lookup(MTRRfix4K_F0000, (x-nat(F0000h))/4096)
    else lookup(MTRRfix4K_F8000, (x-nat(F8000h))/4096)
```

where the `lookup` function takes a fixed-range register and the range index as arguments and returns the memory type of the range in the register:

```
function lookup(r::MTRRfix, i::[0..7])::MemType
 = if i == 0 then r.range0[2:0]
   elif i == 1 then r.range1[2:0]
   elif i == 2 then r.range2[2:0]
   elif i == 3 then r.range3[2:0]
   elif i == 4 then r.range4[2:0]
   elif i == 5 then r.range5[2:0]
   elif i == 6 then r.range6[2:0]
   else r.range7[2:0]
```

If a physical is not covered by any memory type range, then it gets the default memory type, which specified by the following register:

MTRR default memory type register

```
register MTRRdefType::MTRRdefType alias MSR[000002FFh]
layout MTRRdefType
  field type::bits 3
  field rsv1::bits 7 reserved and must be 0
  field FIXE::bit
  field E::bit
  field rsv2::bits 52 reserved and must be 0
```

type the default memory type.
FIXE enables and disables fixed-range memory type registers.
E enables and disables all memory type registers.

Summarizing this section, we define a function that computes the memory type of a physical address and combines it with the memory type from the PAT. The function first checks if the MTRR are disabled, in which case the memory type of the physical address is set to the `UC`. Otherwise, the function computes the memory type using fixed and variable ranges. In case the physical address is covered by both a fixed range and a variable range, the fixed range takes precedence:

```
function mtrr(addr::bits $pa, pat_mt::MemType)::MemType
 = if not MTRRdefType.E then combine_mtrr_pat(UC, pat_mt)
   elif mtrr_fix_cover(addr) and MTRRdefType.FIXE then
        combine_mtrr_pat(mttr_fix(addr), pat_mt)
   else combine_mtrr_pat(mtrr_vars(addr, MTRRdefType.type), pat_mt)
```

The `combine_mtrr_pat` function combines the memory type of the physical address with the memory type of the linear address:

```
function combine_mtrr_pat(mtrr_mt::MemType, pat_mt::MemType)::MemType
 = if pat_mt == UC then UC
   elif pat_mt == WC then WC
   elif pat_mt == WP then
      if mtrr_mt == UC or mtrr_mt == WC or mtrr_mt == WT then UC
      else WP
   elif pat_mt == WT then
      if mtrr_mt == UC or mtrr_mt == WC or mtrr_mt == WP then UC
      else WT
   else mtrr_mt
```

12.9 Fast System Call

Before fast system call instructions were introduced, the only way for user processes (`CPL==11b`) to call system services (`CPL==00b`) was to perform a far control transfer via call or task gate. This mechanism is complex and slow, because the target code and stack segment descriptors and offsets have to be fetched from the descriptor table and the task state segment. The bit `EFER.SCE` enables fast system call instructions. We introduce a parameter alias for this bit.

```
parameter $SCE::bit = EFER.SCE
```

Fast system call instructions provide a shortcut: instead of fetching the descriptors and offsets, the instructions compute them using a set of model-specific registers. There are two pairs of fast system call instructions: SYSCALL/SYSRET and legacy SYSENTER/SYSEXIT. The former pair uses STAR, LSTAR, CSTAR, and SFMASK registers. The latter pair uses SYSENTER_CS, SYSENTER_ESP, and SYSENTER_EIP registers.

The system target address register is used by SYSCALL and SYSRET instructions.

```
register STAR::STAR alias MSR[C0000081h]
layout STAR
  field EIP::bits 32
  field SYSCALL_CS::Selector
  field SYSRET_CS::Selector
```

EIP is used by the SYSCALL instruction to set the RIP in legacy mode.

SYSCALL_CS is used by the SYSCALL instruction to set the CS/SS selectors.

SYSRET_CS is used by the SYSRET instruction to set the CS/SS selectors.

The LSTAR and CSTAR registers are used by the SYSCALL instruction to set the RIP register in long mode. In compatibility mode, RIP = CSTAR. In 64-bit mode, RIP = LSTAR.

```
register LSTAR::bits 64 alias MSR[C0000082h]
register CSTAR::bits 64 alias MSR[C0000083h]
```

The SYSCALL instruction clears bits in the flags registers RFLAGS using the mask SFMASK. A set bit in the SFMASK indicates that the corresponding bit in the RFLAGS must be cleared.

```
register SFMASK::SFMASK alias MSR[C0000084h]
layout SFMASK
  field mask::bits 32
  field rsv::bits 32 reserved and must be 00000000h
```

The SYSENTER_XXX registers are used in a similar way by the SYSENTER/SYSEXIT instructions. For detailed information, refer to specifcation of the instruction in chapter I.

```
register SYSENTER_CS::SYSENTER_CS alias MSR[00000174h]
layout SYSENTER_CS
  field CS::bits 16
  field rsv::bits 48 reserved and must be bits(48, 0)
register SYSENTER_ESP::SYSENTER_ESP alias MSR[00000175h]
layout SYSENTER_ESP
  field ESP::bits 32
  field rsv::bits 32 reserved and must be 00000000h
register SYSENTER_EIP::SYSENTER_EIP alias MSR[00000176h]
layout SYSENTER_EIP
  field EIP::bits 32
  field rsv::bits 32 reserved and must be 00000000h
```

12.10 APIC Base Address

The APIC base register defines a page in the physical address range, where the local APIC registers are mapped. Besides that, the register has a control bit that enables/disables the local APIC and a bit that indicates whether the current processor is the bootstrap processor or not.

```
register APIC_BASE::APIC_BASE alias MSR[0000001Bh]
layout APIC_BASE
  field rsv1::bits 8 reserved and must be 00h
  field BSP::bit read only
  field rsv2::bits 2 reserved and must be 00b
  field AE::bit
  field ABA::bits 40
  field rsv3::bits 12 reserved and must be 000h
  abbr base::bits 52 = ABA++000h
```

BSP (bootstrap processor) When the bit is set, the current processor is the bootstrap processor. Otherwise, the current processor is an application processor. The bootstrap processor is responsible for setting up all necessary system data structures and for waking up other application processors during the system boot.

AE (APIC enable) When the bit is set, the local APIC is enabled and accepts interrupts. Otherwise, the local APIC is disabled.

base (APIC base physical address)

12.11 Time-Stamp Counters

The time-stamp counter register is incremented on every processor cycle. Software can read the register using the `RDTSC/RDTSCP` instructions.

```
register TSC::bits 64 alias MSR[00000010h]
```

The time-stamp counter auxiliary register is available for system software, i.e. software can set the register to any value. The `RDTSCP` instruction loads the lower double word of the register into the `RCX[31:0]`.

```
register TSC_AUX::TSC_AUX alias MSR[C0000103h]
layout TSC_AUX
  field value::bits 32
  field rsv::bits 32 reserved and must be 00000000h
```

CHAPTER

THIRTEEN

ARCHITECTURE

13.1 Operating Modes

An operating mode is a combination of special control bits. We introduce parameter aliases for these bits so that we can refer to them in type expressions:

```
parameter $PE::bit = CR0.PE
parameter $PG::bit = CR0.PG
parameter $LMA::bit = EFER.LMA
parameter $L::bit = CS.attr.L
parameter $VM::bit = RFLAGS.VM
```

Not every combination is an operating mode though. The following invariants hold:

```
invariant not $VM or $PE
invariant not $VM or not $LMA
invariant not $LMA or $PG
invariant not $L or $LMA
```

There are 5 operating modes:

- Real mode (`$PE==0b`, other special control bits are reserved and must be zero) This is the initial operating mode, i.e the mode in which the processor starts running after receiving a `RESET` or an `INIT` signal. In this mode the processor operates similar to the intel–8086 processor:
  - protection is effectively disabled (`CPL = 00b`),
  - segmentation is simple (no descriptor tables, software can not change segment register attributes and limits),
  - paging is disabled,
  - interrupt handling is simple (no gate/task descriptors),
  - default address and operand widths are 16 bits,
  - 64-bit registers are not accessible

  - some system instructions are not recognized (generate xUD exception)
  - some system registers are not accessible Note that it is possible to switch to the protected mode (`PE = 1`), change segment register attributes and limits, and then switch back to the real mode. Software can access upto 4GB memory.

- Legacy protected mode (`$PE==1b`, `$VM==0b`, `$LMA==0b`, `$L` is reserved and must be zero) In this mode most of the processor features are enabled:
  - protection is enabled (the processor checks accesses to the descriptor tables and segments, I/O ports, system instructions),
  - segmentation is enabled (software can update segment registers by loading them from the descriptor tables or performing far control transfers),
  - hardware task switching is supported,
  - 32-bit paging is supported (software can enable paging by setting `$PG==1b`),
  - virtualization is supported,
  - interrupt handling goes through interrupt/trap/task gates,
  - 64-bit registers are not accessible,
  - most of the system instructions are recognized,
  - most of the system registers are accessible.

- Virtual 8086 mode (`$PE==1b`, `$VM==1b`, other control bits are reserved and must be zero) This mode simulates real mode. It was introduced to run legacy real mode applications in a protected mode operating system:
  - protection is enabled, the least privileged level is used (`CPL==11b`),
  - segmentation is similar to that in real mode,
  - 32-bit paging is supported,
  - interrupts are handled either as in real mode or in legacy protected mode, depending on `RFLAGS.IOPL`, `CR4.VME`, `TSS` interrupt redirection bitmap
  - default address and operand widths are 16 bits,
  - 64-bit registers are not accessible,
  - some system instructions are not recognized (generate `xUD` exception),
  - some system registers are not accessible.

- 64-bit mode (`$PE==1b`, `$LMA==1b`, `$L==1b`, `$VM` is reserved and must be zero) This mode extends legacy protected mode with 64-bit registers and 64-bit paging. It also simplifies the segmentation mechanism by ignoring the base address, limit, and some attribute fields of segment registers, thus setting a single flat logical address space. Among other changes to legacy protected mode are:
  - hardware task switching is not supported,
  - default address width is 64 bits,
  - default operand width is 32 bits,
  - RIP relative address mode is supported,

  - slight modifications to the far control transfer mechanism,
  - some instructions are redefined or not recognized.
- Compatibility mode (`$PE==1b`, `$LMA==1b`, `$L==0b`, `$VM` is reserved and must be zero) This mode simulates legacy protected mode. A 64-bit OS can run legacy protected mode applications in compatibility mode. Consequently, this mode combines features of the 64-bit mode and legacy protected mode. Paging, control transfer, interrupt handling are performed like in 64-bit mode. Segmentation, address and operand widths, available instructions are similar to that in legacy protected mode.

Some groups of operating modes have names.

- major mode: protected
  - sub-mode: legacy protected mode,
  - sub-mode: compatibility mode,
  - sub-mode: 64-bit mode,
- major mode: legacy
  - sub-mode: real mode,
  - sub-mode: legacy protected mode,
  - sub-mode: virtual 8068 mode,
- major mode: long mode
  - sub-mode: 64-bit mode,
  - sub-mode: compatibility mode.

```
function $legacy_mode = not $LMA
function $long_mode = $LMA
function $real_mode = not $PE
function $vm86_mode = $legacy_mode and $PE and $VM
function $protected_mode = $PE and not $VM
function $legacy_protected_mode = not $LMA and $protected_mode
function $compatibility_mode = $long_mode and not $L
```

Operating modes can also be grouped by register/address widths.

- major mode: x16 mode
  - sub-mode: real mode
  - sub-mode: virtual 8068 mode
- major mode: x32 mode
  - sub-mode: legacy protected mode
  - sub-mode: compatibility mode
- major mode: x64 mode

- sub-mode: 64-bit mode

```
function $x16_mode = $real_mode or $vm86_mode
function $x32_mode = $protected_mode and not $long_mode or $compatibility_mode
function $x64_mode = $long_mode and $L
```

13.2 Exceptions

Exceptions are internal events, that are generated by

- the processor, when it detects an error during instruction fetch and execution;
- software, using `INTO` and `INT3` instructions.

This distinguishes exceptions from interrupts, which are asynchronous events that are typically triggered by I/O devices. However, interrupts are recognized at instruction boundary. Depending on how the interrupted program is restarted, exceptions are divided into faults, traps, aborts.

- After handling a fault exception, the processor repeats execution of the instruction that caused the exception.
- After handling a trap exception, the processor continues execution from the instruction immediately after the trapping instruction.
- Program restart is not supported for abort exceptions.

Each exception has a byte assigned to it, which is called a vector. This number is used as an index in the interrupt descriptor table to determine the location of the interrupt handler. Exceptions have associated 16-bit error code, which we store in the `code` field. The field `data` is used only with page fault exception to store the faulting virtual address.

```
record Exception
  field valid::bit
  field vector::bits 8
  field code::bits 16
  field data::bits 64
```

The first 32 vectors are reserved for system exceptions and are used as shown in Table 13.2. Other vectors are available for software and external interrupts. We define a set of reserved vectors.

```
set Vector = {xDE = 00h, xDB = 01h, xNMI = 02h, xBP = 03h,
              xOF = 04h, xBR = 05h, xUD = 06h, xNM = 07h, xDF = 08h, xTS = 00h,
              xNP = 01h, xSS = 02h, xGP = 03h, xPF = 04h, xMF = 10h, xAC = 11h,
              xMC = 12h, xXF = 13h, xSX = 1Eh}
```

Actions raise an exception via the **fail** statement using the following helper function, that constructs an exception given the vector and error code.

```
function exception(v::Vector, c::bits 16)::Exception
    = Exception with [valid = 1b, vector = v, code = c, data = bits(64, 0)]
```

Vector	Description	Alias	Type	Source
0	Divide-by-Zero Error	xDE	Fault	Internal
1	Debug	xDB	Fault or Trap	Internal
2	Non-Maskable Interrupt	xNMI	N/A	External
3	Breakpoint	xBP	Trap	Software
4	Overflow	xOF	Trap	Software
5	Bound-Range	xBR	Fault	Internal
6	Invalid-Opcode	xUD	Fault	Internal
7	Device-Not-Available	xNM	Fault	Internal
8	Double-Fault	xDF	Fault	Internal
9	Reserved	N/A	N/A	N/A
10	Invalid-TSS	xTS	Fault	Internal
11	Segment-Not-Present	xNP	Fault	Internal
12	Stack-Segment	xSS	Fault	Internal
13	General-Protection	xGP	Fault	Internal
14	Page-Fault	xPF	Fault	Internal
15	Reserved	N/A	N/A	N/A
16	FP Exception Pending	xMF	Fault	Internal
17	Alignment-Check	xAC	Fault	Internal
18	Machine-Check	xMC	Fault	External
19	SIMD Floating-Point	xXF	Fault	Internal
20–31	Reserved	N/A	N/A	N/A

Table 13.1: Exception and Interrupts

Sometimes the vector of an exception depends on the origin of the faulting memory access. We define the following helper functions that select the correct vector.

```
function xGP_xTS(origin::Origin)::Vector
    = if origin == task then xTS
      else xGP
function xGP_xSS(origin::Origin)::Vector
    = if origin == stack then xSS
      else xGP
function xNP_xTS(origin::Origin)::Vector
    = if origin == task then xTS
      else xNP
function xSS_xTS(origin::Origin)::Vector
    = if origin == task then xTS
      else xSS
```

13.3 Address spaces

Software references memory using a logical address, which is a pair of a segment register and an offset. The offset width depends on the type of the memory access. An instructions is fetched using the `RIP[$la-1:0]` offset, where `$la` depends on the operating mode:

```
function $la
    = if $x64_mode then 64
      else if $x32_mode then 32
           else 16
```

The width of an stack offset depends on the operating mode and the `$B` bit of the stack descriptor:

```
function $sa
    = if $x64_mode then 64
      else if $B then 32
           else 16
```

Where `$B` is an alias for the `D/B` bit of the stack segment descriptor.

```
parameter $B::bit = SS.attr.DB
```

Calculation of the address width of a memory operand is a bit involved. We describe it in chapter 14. For now, we define a short alias `$oa`.

```
function $oa = $addr_width
```

Segmentation translation converts a logical address into a virtual (also referred to as linear) address. The width of a virtual address is computed as follows:

```
function $va::{32, 48}
    = if $long_mode then 48
      else 32
```

Paging translation converts a virtual address into a physical address. The width of a physical address is implementation dependent. However, it does not exceed 52 bits. We introduce a parameter for physical address width:

```
parameter $pa::[32..52]
```

13.4 Memory System Interface

We are going to describe a number of abstract actions that define an interface to the memory system. In section 9, we defined memory request/reply and command registers. The following actions issue requests to the memory system by reading and writing those registers:

- read from the physical memory: `pread(origin, width, addr, memtype);`
- write to the physical memory: `pwrite(origin, width, data, addr, memtyp);`
- read from an I/O port: `read_port(port_addr, width);`

- write to an I/O port: `write_port(port_addr, width, data);`
- invalidate the cache: `invalidate_cache(writeback);`
- flush a cache line: `flush_cache_line(addr);`
- read a translation from the TLB: `read_tlb(va, asid, rw, us, exe);`
- flush the TLB: `flush_tlb(local, with_asid, asid);`
- flush a translation: `invlpg(addr, with_asid, asid);`

These actions belong to the set of primitives of our domain-specific language along with the **`write register`**, **`fail`**, etc. statements. Therefore, we do not define these actions, but assume that they will be defined when our domain-specific language is integrated to an abstract machine. In the rest of this section we specify the arguments and results of the actions.

Actions for reading and writing the physical memory take as arguments the origin of the access, the width of the data, the address of the data, the data (in case of the write action), and the memory type of the access. The read action returns the requested data.

```
action pread(origin::Origin, $n::Width,
             addr::bits $pa, memtype::MemType)::bits $n

action pwrite(origin::Origin, $n::Width, data::bits $n,
              addr::bits $pa, memtype::MemType)
```

We distinguish the following origins of an access:

```
set Origin = {code, stack, data, fsgs, task, sys}
```

Memory access width is any multiple of `8` not exceeding `64`.

```
set Width = {w8 = 8, w16 = 16, w24 = 24, w32 = 32, w40 = 40,
             w48 = 48, w56 = 56, w64 = 64}
```

For reading and writing an I/O port, we need to specify the access width, the 16-bit port address, and the data (in case of a write access). The read action returns the requested data.

```
action read_port($n::{8, 16, 32}, port::bits 16)::bits $n
action write_port($n::{8, 16, 32}, data::bits $n, port::bits 16)
```

There are two types of cache invalidation: with writeback and without writeback. The former write any updated data back to the main memory. The latter discards all updates. Thus, the action for cache invalidation takes the writeback indicator as an argument.

```
action invalidate_cache(writeback::bit)
```

It is possible to flush a specific cache line, using the following action:

```
action flush_cache_line(addr::bits $pa)
```

If the cache line contains updates, it is written back to the main memory before the flush.

The following action takes a virtual address, an address space identifier, access right indicators (read/write, user/superuser, code/data) and returns a translation of the virtual address along with page fault indicator, page fault error code, and the memory type:

```
action read_tlb(addr::bits $va, asid::bits 32, rw::bit,
                us::bit, code::bit)::(bit, bits 32, bits $pa, MemType)
```

In the next section we show how to use this action to perform virtual read and virtual write accesses.

To flush the TLB we specify whether the flush affects all translation or only local translation, and whether the flush affects only translation with in the specific address space.

```
action flush_tlb(local::bit, with_asid::bit, asid::bits 32)
```

If the `with_asid` is false, then the action flushes all translations in all address spaces.

Based on the `flush_tlb` action, we define helper actions:

```
action flush_tlb_all
  call flush_tlb(0b, 0b, 00000000h)
action flush_tlb_local(asid::bits 32)
  call flush_tlb(1b, 1b, asid)
```

It is possible to flush all translation of a specific virtual address. The following action takes a virtual address and an address space identifier and flushes all translations of the virtual address within the specified address space.

```
action invlpg(addr::bits $va, with_asid::bit, asid::bits 32)
```

The name of the action comes from ‘Invalidate a Page’.

13.5 Reading and Writing the Virtual Memory

The following action translates a virtual address to the physical address using the `read_tlb` action. In case if translation is impossible, the action generates a page fault exception.

```
action va_to_pa(addr::bits $va, rw::bit, us::bit, code::bit)::(bits $pa, MemType)
  call (failed, ecode, paddr, memtype) = read_tlb(addr, current_asid, rw, us, exe)
  let page_fault = Exception with [valid = 1b, vector = xPF,
                                   code = ecode,
                                   data = zxt(64, addr)]
  fail page_fault when failed
  return (paddr, memtype)
```

The current address space identifier depends on virtualization mode. In host mode, the identifier is zero. In guest mode, it is read from the VMCB register.

```
function current_asid = if $GUEST then _vmcb_ca.GUEST_ASID
                        else 00000000h
```

Actions `vread/vwrite` perform memory read/write accesses using `va_to_pa` and `pread/pwrite` actions. If the virtual memory access does not cross a page boundary, then the action simply translates the virtual address and forwards the access to the physical memory. Otherwise, the access is split into two accesses at the page boundary.

```
action vread(origin::Origin, $n::Width, addr::bits $va, cpl::bits 2)::bits $n
  let to_next_page = nat(FFFh - addr[11:0]) + 1
  if to_next_page <= $n / 8 then
```

```
    call (physaddr, memtype) = va_to_pa(read, origin, addr, cpl)
    call res = pread(origin, $n, physaddr, memtype)
    return res
  else let $k = (($n / 8) - to_next_page) :: [1 .. $n/8-1]
      let $n0 = $k * 8
      let $n1 = $n - $n0
      let addr1 = (addr[$va-1:12] + bits ($va-12, 1)) ++ 000h
      call (physaddr0, memtype0) = va_to_pa(addr, 0b, cpl==00b, origin==code)
      call (physaddr1, memtype1) = va_to_pa(addr1, 0b, cpl==00b, origin==code)
      call res0 = pread(origin, $n0, physaddr0, memtype0)
      call res1 = pread(origin, $n1, physaddr1, memtype1)
      return res1 ++ res0
action vwrite(origin::Origin, $n::Width, data::bits $n, addr::bits $va, cpl::bits 2)
  let to_next_page = nat(FFFh - addr[11:0]) + 1
  if to_next_page <= $n / 8 then
    call (physaddr, memtype) = va_to_pa(write, origin, addr, cpl)
    call pwrite(origin, $n, data, physaddr, memtype)
  else let $k = (($n / 8) - to_next_page) :: [1 .. $n/8-1]
      let $n0 = $k * 8
      let $n1 = $n - $n0
      let addr1 = (addr[$va-1:12] + one($va-12)) ++ 000h
      call (physaddr0, memtype0) = va_to_pa(addr, 1b, cpl==00b, 0b)
      call (physaddr1, memtype1) = va_to_pa(addr1, 1b, cpl==00b, 0b)
      call pwrite(origin, $n0, data[$n0-1:0], physaddr0, memtype0)
      call pwrite(origin, $n1, data[$n-1:$n0], physaddr1, memtype1)
```

13.6 Page Tables

Three paging modes exist:

1. paging mode in long mode, with the following properties:
   - four page table levels;
   - 512 page table entries in each page table;
   - 64-bit wide virtual addresses;
   - 64-bit wide page table entries.
2. paging mode in legacy mode with page addresses extenstions (PAE) enabled, with the following properties:
   - three page table levels;
   - 8 page table entries in the top-level page tables, and 512 page table entries in other page tables;
   - 64-bit wide virtual addresses;
   - 64-bit wide page table entries.
3. paging mode in lagacy mode with PAE disabled, with the following properties:
   - two page table levels;

- 1024 page table entries in each page table;
- 32-bit wide virtual addresses;
- 32-bit wide page table entries.

The current paging mode can be detected by examining two control bits: `EFER.LMA` and `CR4.PAE`. The former bit is aliased to `$long_mode`, and the latter bit is aliased to `$PAE`. Note that `$long_mode` implies `$PAE` because long mode cannot be activated without enabling page address extenstions.

The 12 least significant bits in a virtual address specify a byte offset in the page. The remaining bits are partitioned into groups, such that each group specifies the index in the page table at the corresponding level. The following layouts show how the bits are grouped in all three paging modes:

```
layout VirtualAddress
  field idx0::bits 12
  field idx1::bits 9
  field idx2::bits 9
  field idx3::bits 9
  field idx4::bits 9
  field rsv::bits 16 ignored
layout VirtualAddressLegacyPAE
  field idx0::bits 12
  field idx1::bits 9
  field idx2::bits 9
  field idx3::bits 2
layout VirtualAddressLegacy
  field idx0::bits 12
  field idx1::bits 10
  field idx2::bits 10
```

We compute the number of page table level using the following function:

```
function $max_level
  = if $long_mode then 4
    elif $PAE then 3
    else 2
```

The size of a page table entry is 32 bits in legacy mode without PAE, and 64 bits in other modes (recall that `$long_mode` implies `$PAE`):

```
function $pte_size = if $PAE then 64
                     else 32
```

As we have define in section 8, an abtract page table entry has the following fields:

```
record AbsPTE
  field p::bit
  field r::Rights
  field a::bit
  field d::bit
  field g::bit
  field large::bit
  field ba::bits 64
  field pat_idx::bits 3
  field valid::bit
```

p the present flag.

r the entry access rights.

a the ‘accessed’ flag indicates whether the entry was accessed by the TLB or not.

d the ‘dirty’ flag indicates whether the entry was used to translate a virtual address for a write access.

g the ‘global’ flag, which indicates whether a translation that uses this entry is global or not.

large the large page indicator, when this flag is set, then the page table entry specifies a large page.

ba the base address of the next level page table or the page.

pat_idx the index into the page attribute table, which is used to calculate the memory type of the memory region where the next level page table lies.

valid the flag that indicates whether the binary representation of the entry is valid, i.e. reserved fields are set to correct values.

The layout of a concrete page table entry depends on paging mode and on the page table level. A 64-bit leaf page table entry has the following layout in long mode and in legacy mode with PAE:

```
layout PTE1
  field P::bit
  field RW::bit
  field US::bit
  field PWT::bit
  field PCD::bit
  field A::bit
  field D::bit
  field PAT::bit
  field G::bit
  field AVL::bits 3 ignored
  field PFN::bits 40
  field ign::bits 11 ignored
  field NX::bit
```

P the present flag.

RW the read-write flag, when this flag is set, writing to the page is allowed.

US the user-system flag, when this flag is set, user accesses to the page are allowed.

PWT the page-write-through flag, it is the `pat_idx[0]` bit of the abstract page table entry.

PCD the page-cache-disable flag, it is the `pat_idx[1]` bit of the abstract page table entry.

A the ‘accessed’ flag indicates whether the entry was accessed by the TLB or not.

D the ‘dirty’ flag indicates whether the entry was used to translate a virtual address for a write access.

PAT the `pat_idx[2]` bit of the abstract page table entry.

G the ‘global’ flag, which indicates whether a translation that uses this entry is global or not.

PFN the page-frame number of the page.

NX the not-execute flag, when the flag is cleared, code fetches from the page are allowed.

We list the layouts of page table entries in other paging modes in Appendix S. They has the same fields, but the fields are in other position.

Knowing the layouts of the page table entries, it is straightforward to define a function that converts from the conrete page table entries to abstrac page table entries by checking the page table level, the paging mode and copying the bits:

```
function parse_pte($level::[1..$max_level], x::bits $pte_size)::AbsPTE
  = if $level == 1 then parse_pte1(x)
    elif $level == 2 then parse_pte2(x)
    elif $level == 3 then parse_pte3(x)
    else parse_pte4(x)
```

Functions `parse_pte*` are defined in Appendix S.

13.7 Segment Descriptors

When software writes a new selector to a segment register, the processor uses the new selector as an index into the descriptor table to look up the corresponding user segment descriptor. If software has enough privileges to access the descriptor, then it is loaded into the hidden part of the segment register. The user segment descriptor defines the virtual base address, the limit, readability, and other attributes of the segment. Besides user segment descriptors, a descriptor table contains system segment descriptors and control transfer gate descriptors. Thus, we define the `Descriptor` type as a union:

```
union Descriptor
  union Segment
  layout CallGate
  layout IntrGate
  layout TrapGate
  layout TaskGate
union Segment
  union UserSegment
  union SystemSegment
```

Each descriptor is `64` bits wide. The type of a descriptor `x` can be determined from bits `x[44:40]`. When `x[44]==1b`, the `x` is a user segment descriptor. A user segment descriptor defines either a data segment or a code segment.

```
union UserSegment
  layout DataSegment
  layout CodeSegment
```

The layouts of a data segment descriptor and a code segment descriptor are similar and differ in a few attribute bits.

```
layout DataSegment
  field limit1::bits 16 ignored when $x64_mode
  field base1::bits 24
  field A::bit ignored when $x64_mode
  field W::bit ignored when $x64_mode
  field E::bit ignored when $x64_mode
  field code::bit reserved and must be 0b
```

```
field S::bit reserved and must be 1b
field DPL::bits 2
field P::bit
field limit2::bits 4 ignored when $x64_mode
field AVL::bit ignored
field rsv::bit reserved and must be 0b
field DB::bit ignored when $x64_mode
field G::bit ignored when $x64_mode
field base2::bits 8
abbr readable::bit = 1b
abbr writeable::bit = W
abbr executable::bit = 0b
abbr expand_down::bit = E
abbr conforming::bit = 0b
abbr base::bits 32 = base2++base1
abbr limit::bits 32 = if G then limit2++limit1++000h
                      else 000h++limit2++limit1
```

DPL The segment descriptor privilege level is used by the protection mechanism when software tries to load the descriptor into a segment register. See chapter 13.10 for more information.

P The present bit indicates whether the segment defined by the descriptor is in the memory or not. An attempt to access a non-present segment generates an exception.

AVL This bit is available for software. In other words, software can set the bit to any value.

DB For stack segments, this bit controls the stack address width. For expand-down segments, this bit defines the upper limit of the segment. When the bit is set, the stack address width is `32` and the upper limit is `FFFFFFFFh`. Otherwise, the stack address width is `16`, and the upper limit is `FFFFh`. In 64-bit mode, this bit is ignored and the stack address width is `64`.

G The granularity bit control the limit scaling factor. When the bit is set, the factor is `4096`. Otherwise, the factor is `1`.

readable All data segments are readable.

writeable When the bit is set, the data segment is writeable. Otherwise, it is read-only. In 64-bit mode, all data segments are writeable

executable All data segments are not executable. Attempt to load a data segment descriptor into the code segment register raises an exception.

expand_down When the bit is set, the data segment is expand-down. The `limit` field of an expand-down segment defines the smallest valid offset (in other words, the `limit` is a lower bound of the segment). In 64-bit mode, no data segment is expand-down.

conforming All data segments are not conforming.

base Segmentation mechanism adds the segment base address to an offset to get a virtual address. Note that the base address is 32 bits wide. In 64-bit mode, the base address is zero-extended to 64 bits for the `FS/GS` registers. For other registers in 64-bit mode, the base address is assumed to be zero.

limit When the granularity bit is set, the segment limit is scaled by 4096. For expand-down segments the limit defines the lower bound (the smallest valid offset), and for expand-up segments the limit defines the upper bound (the largest valid offset). In 64-bit mode, no limit checks are performed.

```
layout CodeSegment
  field limit1::bits 16 ignored when $x64_mode
  field base1::bits 24 ignored when $x64_mode
  field A::bit ignored when $x64_mode
  field R::bit ignored when $x64_mode
  field C::bit
  field code::bit reserved and must be 1b
  field S::bit reserved and must be 1b
  field DPL::bits 2
  field P::bit
  field limit2::bits 4 ignored when $x64_mode
  field AVL::bit ignored
  field L::bit
  field D::bit
  field G::bit ignored when $x64_mode
  field base2::bits 8
  abbr readable::bit = R
  abbr writeable::bit = 0b
  abbr executable::bit = 1b
  abbr expand_down::bit = 0b
  abbr conforming::bit = C
  abbr base::bits 32 = base2++base1
  abbr limit::bits 32 = if G then limit2++limit1++000h
                        else 000h++limit2++limit1
```

readable When the bit is set, read from the segment are allowed. In 64-bit mode, all code segments are readable.

writeable No code segment is writeable. An attempt to write to a code segment generates an exception.

executable All code segments are executable.

expand_down All code segments are expand-up.

conforming When the bit is set, the descriptor can be accessed from a lower privileged software. Control transfer to a conforming code segment does not change the current privilege level.

The following function extracts attributes from the given descriptor:

```
function desc_attr(desc::Descriptor)::Attr = 0h ++ desc[55:52] ++ desc[47:40]
```

The following two functions construct trivial data segment descriptors with valid attributes:

```
function null_stack_desc::DataSegment
  = DataSegment with [W = 1b]
function null_data_desc::DataSegment
  = DataSegment
```

There are two types of system segments: a local descriptor table segment and a task state segment.

```
union SystemSegment
  layout LDT
  layout TSS
```

Software cannot directly access a system segment descriptor. Therefore, a system descriptor does not have `readable`, `writeable`, `executable` attributes. All system segments are expand-up and non-conforming. In contrast to user segments, a system segment in 64-bit mode has a 64-bit base address. Since 64-bit system descriptor cannot accommodate a 64-bit base address, each system descriptor is followed by a dummy descriptor in a descriptor table. The dummy descriptor stores the upper 32 bits of the base address. The type bits of the dummy descriptor must all be zeros.

```
layout UpperDescriptor
  field upper::bits 32
  field ign1::bits 8 ignored
  field rsv1::bits 5 reserved and must be 00000b
  field ign2::bits 19
```

The type bits of an LDT descriptor must be `00010b`. Other components of an LDT descriptor have the same meaning as the corresponding components of a user segment.

```
layout LDT
  field limit1::bits 16
  field base1::bits 24
  field type::bits 4 reserved and must be 0010b
  field S::bit reserved and must be 0b
  field DPL::bits 2
  field P::bit
  field limit2::bits 4
  field AVL::bit ignored
  field ign::bits 2 ignored
  field G::bit
  field base2::bits 8
  abbr base::bits 32 = base2++base1
  abbr limit::bits 32 = if G then limit2++limit1++000h
                        else 000h++limit2++limit1
```

Chapter [Task State Segment] describes the structure of a task state segment, which stores the processor context between task switches and stack pointers for interrupt handlers. The type bits `44`, `42`, and `40` of a TSS descriptor must be `0b`, `0b`, and `1b` respectively. A task segment can be either busy or available. The `busy` bit of a TSS descriptor indicates whether the task segment is busy or not. If a task segment is busy, then there exists a call sequence from that task segment to the current task segment referred by the `TR` register. The current task cannot call a busy task. Thus, the `busy` bit prevents recursive task calls.

In legacy mode, the `size` bit of a TSS descriptor indicates whether the task segment is `16`-bit task segment or `32`-bit task segment. When the bit is zero, the task segment is `16`-bit, otherwise, it is `32`-bit. These two types of task segments differ in the set of stored registers. In long mode, there are only `64`-bit registers, and the `size` bit must be one.

The remaining components of a TSS descriptor have the same meaning as the corresponding components of an LDT descriptor.

```
layout TSS
  field limit1::bits 16
  field base1::bits 24
  field rsv1::bit reserved and must be 1b
  field busy::bit
  field rsv2::bit reserved and must be 0b
  field size::bit reserved and must be 1b when $long_mode
  field S::bit reserved and must be 0b
  field DPL::bits 2
  field P::bit
  field limit2::bits 4
  field AVL::bit ignored
  field ign::bits 2 ignored
  field G::bit
  field base2::bits 8
  abbr base::bits 32 = base2++base1
  abbr limit::bits 32 = if G then limit2++limit1++000h
                        else 000h++limit2++limit1
```

13.8 Gate Descriptors

A gate descriptor is used by control transfer instructions. It specifies the logical address (selector and offset) of the target instruction and the minimal privilege level required to perform the control transfer. There are four types of control transfer gates:

- call gate — a descriptor in the LDT or the GDT, which contains the address the target instruction.
- trap gate — a descriptor in the IDT, which contains the address of the first instruction of the exception handler.
- interrupt gate — a descriptor in the IDT, which contains the address of the first instruction of the interrupt handler. An interrupt gate is identical to a trap gate. The only difference between them is that control transfer through interrupt gate sets the `IF` flag to zero.
- task gate — a descriptor in the LDT or the GDT, which contains the selector of an available TSS. A control transfer through a task gate leads to a task switch. In long mode task gates are not supported.

The layouts of call, interrupt, and trap gates are similar. The layouts of interrupt and trap gates are identical, except for the values of the type bits. We define `Gate` and `IntrOrTrapGate` unions, so that we can work with multiple gate types simultaneously.

```
union Gate
  layout CallGate
  layout IntrGate
  layout TrapGate
```

```
union IntrOrTrapGate
  layout IntrGate
  layout TrapGate
```

The type bits `44`, `42`, `41`, `40` of a call gate descriptor must be respectively `0b`, `1b`, `0b`, `0b`. A call gate descriptor has the following layout:

```
layout CallGate
  field offset1::bits 16
  field sel::Selector
  field params_ign::bits 5 ignored and read as 00000b when $long_mode
  field ign::bits 3 ignored
  field rsv::bits 3 reserved and must be 100b
  field wide::bit reserved and must be 1b when $long_mode
  field S::bit reserved and must be 0b
  field DPL::bits 2
  field P::bit
  field offset2::bits 16
  abbr offset::bits 32 = offset2++offset1
  abbr params::bits 5 = if $long_mode then 00000b
                        else params_ign
```

`sel` (selector) This field contains the selector of the target code segment.

`wide` In legacy mode, this bit controls the size of the gate. When the bit is set, the size is `32` bits. Otherwise, the size is `16` bits. In long mode, the size is always `64` bits, and this bit must be `1b`. The gate size affects the operand size of stack push/pop operations, which are used for saving the return address, parameters, and stack pointer. More specifically, the operand size is equal to the gate size.

`offset` This field specifies the offset of the target instruction in the target code segment, defined by the `sel` selector. In long mode, the gate descriptor is followed by a dummy descriptor in a descriptor table. The dummy descriptor provides the upper `32` bits of the `64`-bit offset (See definition of the `UpperDescriptor`).

`params` This field is meaningful only for call gates in legacy mode, and shows how many parameters should be copied during a control transfer that requires a stack switch.

Formalizing the description of the `wide` bit, we define the following function:

```
function gate_size(gate::Gate)::{16, 32, 64}
  = if not gate.wide then 16
    elif $legacy_mode then 32
    else 64
```

An interrupt gate descriptor has the same fields as a call gate descriptor except for the `param` field. An interrupt handler does not have parameters. Instead of the `param` field, an interrupt gate descriptor has the `ist` field.

```
layout IntrGate
  field offset1::bits 16
  field sel::Selector
  field ist_ign::bits 3 ignored and read as 000b when not $long_mode
  field ign::bits 5 ignored
```

```
  field rsv::bits 3 reserved and must be 110b
  field wide::bit reserved and must be 1b when $long_mode
  field S::bit reserved and must be 0b
  field DPL::bits 2
  field P::bit
  field offset2::bits 16
  abbr offset::bits 32 = offset2++offset1
  abbr ist::bits 3 = if $long_mode then ist_ign
                     else 000b
```

ist (interrupt stack table) In long mode, this field contains an index into the interrupt stack table in the current TSS. The entry in the table contains a stack pointer that is used for the stack switch during control transfer.

A trap gate descriptor is almost identical to the interrupt gate descriptor. The only difference is the values of the type bits.

```
layout TrapGate
  field offset1::bits 16
  field sel::Selector
  field ist_ign::bits 3 ignored and read as 000b when not $long_mode
  field ign::bits 5 ignored
  field rsv::bits 3 reserved and must be 111b
  field wide::bit reserved and must be 1b when $long_mode
  field S::bit reserved and must be 0b
  field DPL::bits 2
  field P::bit
  field offset2::bits 16
  abbr offset::bits 32 = offset2++offset1
  abbr ist::bits 3 = if $long_mode then ist_ign
                     else 000b
```

The `sel` of a task gate descriptor contains the selector of a TSS descriptor. The type bits `[44:40]` must be `00010b`.

```
layout TaskGate
  field ign1::bits 16 ignored
  field sel::Selector
  field ign2::bits 8 ignored
  field type::bits 4 reserved and must be 0010b
  field S::bit reserved and must be 0b
  field DPL::bits 2
  field P::bit
  field ign3::bits 16 ignored
```

13.9 Descriptor Tables

There are three descriptor tables: the global descriptor table, the local descriptor table, and the interrupt descriptor table. The GDT can contain descriptors of any type except interrupt and trap gate descriptors. The LDT can contain only user segment descriptors

and call, task gate descriptors. The IDT can contain only interrupt, trap, and task gate descriptors.

Descriptors in the GDT and the LDT are accessed using selectors. A selector is a bit string of width 16 bits, and has three fields.

```
layout Selector
  field RPL::bits 2
  field TI::bit
  field index::bits 13
```

RPL (requestor privilege level) This field is used in protection checks. See chapter 13.10 for more information.

TI (table index) This bit specifies the descriptor table. If the bit is set, then the descriptor is in the LDT. Otherwise, the descriptor is in the GDT.

index (descriptor index) This field contains a zero-based index of the descriptor in the descriptor table. Since the descriptor size is `8` bytes. The virtual address of the descriptor is equal to `base + index*8`, where the `base` is the virtual base address of the table. Note that the fields `RPL` and `TI` have total width `3`. Therefore, if `x` is the value of the whole selector, then the address of the descriptor can be computed as `base + x & FFF8h`.

Given the `TI` bit of a selector, the function `gdt_or_ldt` returns the base address and the limit of the chosen descriptor table. In case the LDT must be returned, but the `LDTR` points to a non-present descriptor, the function returns a dummy base address and a zero limit.

```
function gdt_or_ldt(local::bit)::(bits $va, nat)
  = if local then
      if LDTR.attr.P then (LDTR.base[$va-1:0], nat(LDTR.limit))
      else (zero($va), 0)
    else (GDTR.base[$va-1:0], nat(GDTR.limit))
```

Fetching a descriptor is a frequently used action. It can occur during control transfer, segment register update, exit from guest mode, etc. We define the `read_desc` action, which given a selector fetches the corresponding descriptor from the descriptor table. The action returns a bit which indicates whether the fetch was successful or not.

```
action read_desc(sel::Selector)::(Descriptor, bit)
  let (dt_base, dt_limit) = gdt_or_ldt(sel.TI)
  let offset = zxt($va, sel & FFF8h)
  if nat(offset) + 7 <= dt_limit then
    call entry = vread(sys, 64, dt_base + offset, 00b)
    return (entry, 1b)
  else return (zero(64), 0b)
```

The `vread` action is defined in section 13.5. It reads from the virtual memory. The first argument is the origin of the access, which in this case indicates that the virtual memory access is requested by the hardware system, not by software.

In 64-bit mode, system descriptors have the second part which contains the upper 32 bits of the base address. Given the selector of a system descriptor, the action `read_upper_desc` fetches the second part of the descriptor. Similarly to the `read_desc`, this action returns a success/fail bit.

```
action read_upper_desc(sel::Selector)::(UpperDescriptor, bit)
  let (dt_base, dt_limit) = gdt_or_ldt(sel.TI)
  let offset = zxt($va, sel & FFF8h) + bits($va, 8)
  if nat(offset) + 7 <= dt_limit then
    call entry = vread(sys, 64, dt_base + offset, 00b)
    return (entry, isUpperDescriptor(entry))
  else return (zero(64), 0b)
```

The following action fetches the upper part of a LDT/TSS descriptor in 64-bit mode using the `read_upper_desc`. In 32-bit mode, the action does not fetch anything and returns a dummy zero descriptor. This allows to uniformly compute the 64-bit base address of a LDT/TSS descriptor.

```
action read_upper_ldt_tss(sel::Selector)::(UpperDescriptor, bit)
  if $x64_mode then
    call (entry, valid) = read_upper_desc(sel)
    return (entry, valid)
  else return (zero(64), 1b)
```

The following action is similar to the `read_upper_ldt_tss`, but it fetches the upper part of a gate descriptor.

```
action read_upper_gate(sel::Selector)::(UpperDescriptor, bit)
  if $long_mode then
    call (entry, valid) = read_upper_desc(sel)
    return (entry, valid)
  else return (zero(64), 1b)
```

Given a selector and a descriptor value, the `write_desc` action writes the value into the descriptor table entry defined by the selector. The action returns a bit which indicates whether the write was successful or not.

```
action write_desc(sel::Selector, desc::Descriptor)::bit
  let (dt_base, dt_limit) = gdt_or_ldt(sel.TI)
  let offset = zxt($va, sel & FFF8h)
  if nat(offset) + 7 > dt_limit then return 0b
  else call vwrite(sys, 64, desc, dt_base + offset, 00b)
       return 1b
```

In contrast to the GDT/LDT, the IDT is accessed using a vector byte (instead of a selector). In 16-bit mode, the IDT does not contain descriptors, but contains 32-bit entries. Each entry is a pair of 16-bit selector and 16-bit offset.

```
action read_idt(vector::bits 8)::(bits 16, bits 16) when $x16_mode
  let offset = zxt($va, vector) << 2
  fail exception(xGP, 0000h) when nat(offset) + 3 > nat(IDTR.limit)
  call entry = vread(sys, 32, IDTR.base[$va-1:0] + offset, CPL)
  return (entry[31:16], entry[15:0])
```

In legacy protected mode, the descriptors in the IDT do not have the upper part. Therefore, the `read_idt` action returns a dummy zero descriptor instead of fetching the upper part.

```
action read_idt(vector::bits 8)::(Descriptor, UpperDescriptor)
  when $legacy_protected_mode
```

```
  let offset = zxt($va, vector) << 3
  let ecode = zxt(14, vector) ++ 10b
  fail exception(xGP, ecode) when nat(offset) + 7 > nat(IDTR.limit)
  call entry = vread(sys, 64, IDTR.base[$va-1:0] + offset, CPL)
  fail exception(xGP, ecode) when not (entry::Descriptor).P
  return (entry, zero(64))
```

In long mode, the `read_idt` action fetched both the descriptor and the upper part.

```
action read_idt(vector::bits 8)::(Descriptor, UpperDescriptor) when $long_mode
  let offset = zxt($va, vector) << 4
  let ecode = zxt(14, vector) ++ 10b
  fail exception(xGP, ecode) when nat(offset) + 15 > nat(IDTR.limit)
  call entry0 = vread(sys, 64, IDTR.base[$va-1:0] + offset, CPL)
  call entry1 = vread(sys, 64, IDTR.base[$va-1:0] + offset + bits($va, 8), CPL)
  fail exception(xGP, ecode) when not (entry0::Descriptor).P
  fail exception(xGP, ecode) when not isUpperDescriptor(entry1)
  return (entry0, entry1)
```

13.10 Protection

Each code segment in the system is associated with one of the four privilege levels. The 2-bit `CPL` register stores the current privilege level. System software runs at `CPL = 00b`, user applications run at `CPL = 11b`. Privilege levels `01b` and `10b` are usually assigned to device drivers.

The processor performs privilege checks (if protection is enabled, `CR0.PE = 1b`) on any access to the system registers and instructions, descriptor tables, page tables and pages, and input/output ports. The goal of these checks is to guarantee that user applications can not get an unauthorized access to system resources. In particular, a user application should not be able to escalate its privilege level.

System instructions raise an exception when `CPL > 00b`. The system registers are protected from user software because they can be accessed only via the system instructions.

During paging translation of a virtual address `va`, the processor checks the `SU` bits in all the page table entries that are used for translation of the `va`. If `CPL > 00b` and at least one of the `SU` bits is cleared, the processor raises a page fault. Thus, it is possible to protect regions of virtual memory on 4K page granularity.

Input/output ports can be accessed only via the `IN/INS/OUT/OUTS` instructions. These instructions raise an exception when `CPL > RFLAGS.IOPL`. Thus, I/O ports can be protected from user software by setting `RFLAGS.IOPL` appropriately.

In the rest of this section, we describe how segments and descriptors are protected. Descriptor privilege checks involve three elements:

- `CPL` — the current privilege level, which specifies the privilege level of the code that is currently running on the processor.
- `sel.RPL` — the requestor privilege level, which is set by software when it accesses a descriptor. Since this field is controlled by software, the privilege level checks

involving this field do not protect from malicious software, because software can set this field to the highest privilege level. This field is useful only for guarding from software bugs.

- `desc.DPL` — the descriptor privilege level, which protects the descriptor from user software accesses.

For a system segment descriptor `desc` that is selected using selector `sel`, the processor performs the following check:

```
function can_access_desc(sel::Selector, desc::Descriptor)::bit
  = CPL <= desc.DPL and sel.RPL <= desc.DPL
```

When software tries to load a user segment descriptor into the `DS/ES/FS/GS` register, the processor makes the following checks:

```
function can_access_data(sel::Selector, desc::UserSegment)::bit
  = $x64_mode or desc.conforming or CPL <= desc.DPL and sel.RPL <= desc.DPL
```

Thus, there are no checks in 64-bit mode or if the descriptor is a conforming code segment. Otherwise, the `CPL` and the `RPL` must not exceed the `DPL`.

Write to the stack segment register `SS` succeeds only if the `RPL`, the `CPL`, and the `DPL` are all equal. In 64-bit mode, the `DPL` is not checked. Since far control transfer instructions might write to the `SS`, we define a more general `can_access_stack` function, which accepts the new `CPL` and the new 64-bit mode indicator.

```
function can_access_stack(sel::Selector, desc::DataSegment,
                          new_cpl::bits 2, new_x64_mode::bit)::bit
  = sel.RPL == new_cpl and (new_x64_mode or new_cpl == desc.DPL)
```

The code segment register `CS` can be updated only in far control transfer instructions, task and guest switches. Given the new `CPL` and the target code segment selector and descriptor, the following function checks whether the `CS` update is allowed or not.

```
function can_access_code(sel::Selector, desc::CodeSegment, new_cpl::bits 2)::bit
  = if desc.conforming then desc.DPL <= CPL
    else desc.DPL == new_cpl and sel.RPL <= new_cpl
```

When the target code segment is conforming the `CPL` does not change, otherwise, the new `CPL` equals the `DPL`.

```
function code_cpl(desc::CodeSegment)::bits 2
  = if desc.conforming then CPL
    else desc.DPL
```

An access to a gate descriptor is allowed only if the `CPL` does not exceed the gate `DPL`.

```
function can_access_gate(desc::Gate)::bit = CPL <= desc.DPL
```

A far return from procedure pops the target code segment selector from the stack. The `RPL` of the selector becomes the new `CPL` when the instruction completes. The processor ensures that the new `CPL` (denoted by the `sel.RPL`) is less privileged that the old `CPL`.

```
function can_ret(sel::Selector)::bit = CPL <= sel.RPL
```

Normally, the following conditions hold:

- `CPL == CS.sel.RPL`
- `CPL == SS.sel.RPL`
- `CPL == CS.attr.DPL if CS.attr.conforming`
- `CPL >= CS.attr.DPL if not CS.attr.conforming`

However, in real mode the `CPL` is always `00b`, and in virtual–8086 mode the `CPL` is always `11b`. System software can violate the equations by providing invalid data to some control transfer instructions (for example, `SYSRET`).

13.11 Privilege Level Change

A `CPL` change can occur only at the following places:

- far procedure call via a call gate (`CALL` instruction):
  1. the processor checks that the `CPL` is sufficient to access the gate descriptor (`CPL==gate_sel.DPL` and `gate_sel.RPL==gate_desc.DPL`);
  2. using the `gate_desc.sel` selector field, the processor fetches the target code segment descriptor `cs_desc`;
  3. if the target code segment is conforming, then no `CPL` change occurs and `cs_desc.DPL <= CPL` must hold;
  4. otherwise, the `cs_desc.DPL` specifies the new `CPL` and `cs_desc.DPL==CPL` must hold;
  5. the processor starts executing instruction at offset `gate_desc.offset` in the target code segment; Note that, if the descriptor tables are write-protected from user applications (either using segmentation-protection or paging-protection), then the target code at `(gate_desc.sel, gate_desc.offset)` is set by system software. Therefore, the privilege level change is safe.
- far return from the procedure (`RETF` instruction):
  1. the processor loads the target instruction address `(cs_sel, cs_offset)` from the stack;
  2. `CPL>=cs_sel.RPL` must hold, and the new `CPL` is set to the `cs_sel.RPL`
  3. the processor fetches the target code segment from the descriptor table using `cs_sel` and it is unclear whether any checks on `cs_desc.DPL` are performed;
  4. the processor starts executing instruction at offset `cs_offset` in the target code segment; Note that the control is transfered to the same or less privileged code, therefore no privilege level escalation can occur.
- fast system call (`SYSENTER` and `SYSCALL` instructions):
  1. the processor computes the target code and stack segments from the `SYSENTER_CS` and the `STAR` registers;

2. the target instruction offset is obtained from either the SYSENTER_EIP, the STAR, the CSTAR, or the LSTAR registers, depending on operating mode and the instruction;
3. the new CPL is set to 00b;
4. the processor starts executing the target instruction; Note that the SYSENTER_CS, the SYSENTER_EIP, the STAR, the CSTAR, and the LSTAR registers can only be written using the WRMSR instruction when CPL==0. Therefore, the privilege level change is safe.

- fast system return (SYSEXIT and SYSRET instructions): these instructions set the new CPL to 11b, thus, no privilege level escalation can occur.
- control transfer to an interrupt handler via an interrupt or a trap gate: the processor actions are similar to those in the far procedure call via a call gate. The gate descriptors are fetched from the IDT. Thus, the privilege change is safe as long as the IDT is write-protected from user application.
- return from an interrupt handler IRET if the nested flag is off (RFLAGS.NT==0b): the processor actions are similar to those in the case of RETF control transfer.
- task switch (only in legacy mode) initiated by:
  - far JMP or CALL via an available TSS descriptor,
  - far JMP or CALL via a task gate,
  - control transfer to an interrupt handler via a task gate,
  - far return from an interrupt handler if the nested flag is on (RFLAGS.NT==1b). The new CPL and the target instruction address are read from the target task state segment. Therefore, task state segments must be write-protected from user applications.
- guest enter VMRUN: this instruction can be executed only when CPL==00b.
- guest exit vmexit: the target instruction address is read from the host state area (VM_HSAVE_PA), which must be write-protected from user applications.

These checks are formalized in the definitions of the corresponding instructions.

13.12 Segmentation Translation

A logical address is a pair of a segment register and an offset into the segment. Segmentation translation converts a logical address into a virtual (linear) address by adding the segment base address to the offset. The lread action shows how logical read is performed. It takes the source of the read access, the data width, the segment register, the width of the offset, and the offset. The action first checks that the offset is within the segment and the access is valid using the check_logical action. After that, the action computes the virtual address as a sum of the segment base and the offset. Once the virtual address is known, the action performs a virtual read access using the vread action.

```
action lread(origin::Origin, $n::Width,
             sr::SR, $k::{16, 32, 64}, offs::bits $k)::bits $n
  call check_logical(read, origin, $n, sr, $k, offs)
  let addr = zxt($va, effective_base(origin, sr.base)) + zxt($va, offs)
  call res = vread(origin, $n, addr, CPL)
  return res
```

In 64-bit mode the segment base address for CS/DS/ES/SS is assumed to be zero. Therefore, we use the following function that checks the source of the access and returns the correct base address.

```
function effective_base(origin::Origin,
                        base::bits 64)::bits $va
  = if $x64_mode or origin == fsgs then base[$va-1:0]
    else zero($va)
```

Similarly to the lread, we define the lwrite.

```
action lwrite(origin::Origin, $n::Width, data::bits $n,
              sr::SR, $k::{16, 32, 64}, offs::bits $k)
  call check_logical(write, origin, $n, sr, $k, offs)
  let addr = zxt($va, effective_base(origin, sr.base)) + zxt($va, offs)
  call vwrite(origin, $n, data, addr, CPL)
```

Checks performed by the processor for a logical memory access depend on operating mode. In 64-bit mode the checks are simplified, while in
other modes the processor makes the following checks:

```
action check_logical(rw::RW, origin::Origin, $n::Width, sr::SR,
                     $k::{16, 32, 64}, offset::bits $k) when not $x64_mode
  fail exception(xGP, sr.sel & FFF8h) when not sr.attr.P
  fail exception(xGP, 0000h) when rw == write and not sr.attr.writeable
  fail exception(xGP, 0000h) when rw == read and not sr.attr.readable
  fail exception(xAC, 0000h)
    when not aligned(offset[2:0], $n) and CR0.AM and RFLAGS.AC and CPL == 11b
  fail exception(xGP_xSS(origin), 0000h)
    when not check_limit(sr.attr, nat(sr.limit), nat(offset))
  fail exception(xGP_xSS(origin), 0000h)
    when not check_limit(sr.attr, nat(sr.limit), nat(offset) + $n/8-1)
```

An attempt to access a non-present segment, to write to a read-only segment, or to read from read-protected code segment raises an exception.

Access alignment checks are enabled for user software at CPL==11b when the AM/AC flags are set. Given the lowest 3 bits of the offset and the access width, the following function check whether the
access is aligned or not.

```
function aligned(addr::bits 3, $n::Width)::bit
  = if $n == 16 and addr[0] then 0b
    elif $n == 32 and addr[1:0] <> 00b then 0b
    elif $n == 64 and addr[2:0] <> 000b then 0b
    else 1b
```

In order to check whether the offset is inside the segment, we check if the segment is an expand-down segment or not. For an expand-down segment, the segment limit

specifies the lower bound of valid offsets. The upper bound is either FFFFFFFFh or FFFFh depending on the DB attribute of the segment. For an expand-up segment, the limit specifies the upper bound of valid offset.

```
function check_limit(attr::UserSegmentAttr,
                     limit::nat, addr::nat)::bit when not $x64_mode
  = if attr.expand_down then
      if (attr::DataAttr).DB then nat(FFFFFFFFh) >= addr and addr > limit
      else 0000FFFFh::nat >= addr and addr >limit
    else addr <= limit
```

In 64-bit mode, the processor does not check for readability/writeability, and, instead of checking that the offset is in the segment, the processor ensures that the offset is canonical. A 64-bit offset is canonical if bits [63:$va] are either all zeros or all ones.

```
function canonical(addr::bits 64)::bit
  = addr[63:$va] == zxt(64-$va, 0b) or addr[63:$va] == sxt(64-$va, 1b)
action check_logical(rw::RW, origin::Origin, $n::Width, sr::SR,
                     $k::{16, 32, 64}, offset::bits $k) when $x64_mode
  fail exception(xGP, sr.sel & FFF8h) when not sr.attr.P and origin <> stack
  fail exception(xAC, 0000h)
    when not aligned(offset[2:0], $n) and CR0.AM and RFLAGS.AC and CPL == 11b
  fail exception(xGP_xSS(origin), 0000h) when not canonical(zxt(64, offset))
```

Note that the SS segment register is allowed to be null.

Control transfer instructions use the following action to check the validity of the new instruction pointer. The action takes the new code segment descriptor and the new instruction pointer. If the new operating mode is 64-bit mode, then the pointer must be canonical. Otherwise, the pointer must be in the segment bounds.

```
action check_rip(desc::CodeSegment, rip::bits 64)
  if $long_mode and desc.L  then
    fail exception(xGP, 0000h) when not canonical(rip)
  else assume not $long_mode
       let attr = desc_attr(desc)
       let limit = desc.limit
       fail exception(xGP, 0000h)
         when not check_limit(attr, nat(limit), nat(rip))
```

13.13 Segment Register Access

In this section we define actions that read from and write to the user segment registers. These segment registers are stored in the SR register array in the following way:

```
SR[000b] = SR[iES] = ES
SR[001b] = SR[iCS] = CS
SR[010b] = SR[iSS] = SS
SR[011b] = SR[iDS] = DS
SR[100b] = SR[iFS] = FS
SR[101b] = SR[iGS] = GS
```

A segment register has four components: the 16-bit selector, the 16-bit attributes, the 32-bit limit, and the 64-bit base address. Software cannot access the latter three components directly. Instead, it can only read and write the first component. On each segment register write, the processor fetches the latter three components from the descriptor table using the selector as an index.

Thus, the action for reading a segment register is very simple: it returns the selector.

```
action read_sr(idx::bits 3)::bits 16
  return SR[idx].sel
```

Writing a segment register is more involved, as we need to update the hidden components too. We describe writes in 16-bit mode and writes in 32/64-bit mode separately. In 16-bit mode the attributes and the limit do not change, and the bases address is simply the selector multiplied by 16. The `write_sr` action has three arguments: the origin of the write, the new value of the selector, and the segment register index. The first argument is not used in 16-bit mode.

```
action write_sr(origin::Origin, sel::Selector, idx::SRIndex) when $x16_mode
  write sel to SR[idx].sel
  write zxt(64, sel) << 4 to SR[idx].base
```

In 32/64-bit mode, there two cases: the selector is null or not. It is possible to write a null selector to the data segment registers `ES`, `DS`, `GS`, `FS` without raising an exception. The exception will be raised later if software attempts to read these registers.

In case the selector is not null, the action performs the following steps:

1. Fetch the descriptor from the descriptor table using the selector as an index.
2. Check that the descriptor is a valid user segment descriptor.
3. Parse the descriptor components and write them to the corresponding segment register components.
4. In case of the `FS`, `GS` segment registers, update the `FSBase`, `GSBase` model specific registers, which store the base address component.

```
action write_sr(origin::Origin, sel::Selector, idx::SRIndex) when not $x16_mode
  if (sel & FFF8h) == 0000h then
    fail exception(xGP_xTS(origin), sel & FFF8h) when idx == iSS or idx == iCS
    write sel to SR[idx].sel
    write 0000h to SR[idx].attr
    write 00000000h to SR[idx].limit
    write zero(64) to SR[idx].base
  else call (desc_temp, desc_valid) = read_desc(sel)
       fail exception(xGP, sel & FFF8h) when not desc_valid
       let desc = desc_temp::UserSegment
       call check_segment(origin, sel, desc, idx)
       write sel to SR[idx].sel
       write desc_attr(desc) to SR[idx].attr
       write desc.limit to SR[idx].limit
       write zxt(64, desc.base) to SR[idx].base
       write zxt(64, desc.base) to FSBase when idx == iFS
       write zxt(64, desc.base) to GSBase when idx == iFS
```

The `read_desc` action is defined in section 13.9. The action `check_segment` makes sure that the given descriptor is present, valid, and can be written to the given segment register. The validity checks are different for the code segment register, stack segment register, and data segment registers. Therefore, this action makes case distinction on the segment register index, and invokes the appropriate action.

```
action check_segment(origin::Origin, sel::Selector,
                     desc::UserSegment, idx::SRIndex)
  if idx == iCS then call check_code(origin, sel, desc)
  elif idx == iSS then call check_stack(origin, sel, desc, CPL, $x64_mode)
  else call check_data(origin, sel, desc)
```

The code segment register is only written to in control transfer and virtualization instructions. The descriptor must be present, must define a code segment, and must have the reserved bits set to zero.

```
action check_code(origin::Origin, sel::Selector, desc::CodeSegment)
  fail exception(xNP_xTS(origin), sel & FFF8h) when not desc.P
  fail exception(xGP_xTS(origin), sel & FFF8h) when not isCodeSegment(desc)
  fail exception(xGP_xTS(origin), sel & FFF8h) when desc.L and $legacy_mode
  fail exception(xGP_xTS(origin), sel & FFF8h)
    when desc.D and desc.L and $long_mode
```

A present user segment descriptor can be written to a data segment register if the current privilege level is sufficient to access it. The privilege level checks are performed in the `can_access_data` function, which is defined in section 13.10. In 32-bit mode, the user segment must be readable.

```
action check_data(origin::Origin, sel::Selector, desc::UserSegment)
  fail exception(xNP_xTS(origin), sel & FFF8h) when not desc.P
  fail exception(xGP_xTS(origin), sel & FFF8h) when not isUserSegment(desc)
  fail exception(xGP_xTS(origin), sel & FFF8h)
    when not desc.readable and not $x64_mode
  fail exception(xGP_xTS(origin), sel & FFF8h)
    when not can_access_data(sel, desc)
```

A present data segment descriptor can be written to the stack segment register if the current privilege level is sufficient to access it. The privilege level checks are performed in the `can_access_stack` function, which is defined in section 13.10. In 32-bit mode, the segment must be writable.

```
action check_stack(origin::Origin, sel::Selector, desc::DataSegment,
                   new_cpl::bits 2, new_x64_mode::bit)
  fail exception(xSS_xTS(origin), sel & FFF8h) when not desc.P
  fail exception(xGP_xTS(origin), sel & FFF8h) when not isDataSegment(desc)
  fail exception(xGP_xTS(origin), sel & FFF8h)
    when not desc.writable and not new_x64_mode
  let accessible = can_access_stack(sel, desc, new_cpl, new_x64_mode)
  fail exception(xGP_xTS(origin), sel & FFF8h) when not accessible
```

13.14 Task State Segment

The task register TR contains a reference to the current task state segment. The following action performs a read from the TSS, given the data width and the offset into the segment.

```
action read_tss($n::Width, offset::bits $va)::bits $n
  fail exception(xTS, TR.sel & FFF8h) when not TR.attr.P
  fail exception(xTS, TR.sel & FFF8h) when nat(offset) + $n - 1 > nat(TR.limit)
  call val = vread(task, $n, TR.base[$va-1:0] + offset, 00b)
  return val
```

The long mode does not support hardware task switch. Therefore, the TSS is used only for storing the stack pointers and the I/O permission bitmap.

```
layout TaskState
  field rsv1::bits 32
  field RSP0::bits 64
  field RSP1::bits 64
  field RSP2::bits 64
  field rsv2::bits 64
  field IST1::bits 64
  field IST2::bits 64
  field IST3::bits 64
  field IST4::bits 64
  field IST5::bits 64
  field IST6::bits 64
  field IST7::bits 64
  field rsv3::bits 64
  field rsv4::bits 16
  field IOPB_BA::bits 16
```

There are two types of stack pointer in the TSS: privilege level stack pointers (RSPi) and interrupt stack pointers (ISTi). These stack pointers define the new stacks for control transfer instructions and for interrupt handlers. The ISTi pointers are available only in long mode. The ist field of an interrupt/trap gate selects one of the seven pointers (when ist > 000b). The RSP0, RSP1, RSP2 pointers are used when the CPL changes to 00b, 01b, 10b during far control transfer.

Given the new CPL and the ist field of a gate (or 000b if the gate does not have the ist field), the following action fetches the pointer to the new stack.

```
action tss_rsp(new_cpl::bits 2, ist::bits 3)::bits 64
  if ist > 000b and $long_mode then
    let addr = (zxt($va, ist)<< 3) + bits($va, 28)
    call res = read_tss(64, addr)
    return res
  else
    let addr = (zxt($va, new_cpl) << 3) + bits($va, 4)
    call res = read_tss(64, addr)
    return res
```

The I/O intercept bitmap contains a bit for each I/O port. When user software attempts to read or write a port and the corresponding bit in the bitmap is set, an exception is

raised. The IOPB_BA field of the TSS specifies the offset of the first byte of the bitmap in the TSS.

```
action tss_iopb_base::bits $va
  call res = read_tss(16, bits($va, 102))
  return zxt($va, res)
```

Given the I/O access width $n and the port address port, the following action determines whether the I/O permission bitmap allows the access or not. Depending on the access width, the action must check 1, 2, or 4 bits in the bitmap. The offset of the byte that contains the first bit is equal to port[15:3]. The last bit is either in the same byte or in the next byte. Therefore, the action fetches two bytes from the bitmap. In case the bytes are beyond the segment limits, the access is not permitted.

```
action tss_io_intercepted($n::{8, 16, 32}, port::bits 16)::bit
  let offset = zxt($va, port[15:3])
  let $bit_idx = nat(port[2:0])::[0..7]
  call base = tss_iopb_base
  let addr = base + offset
  if nat(addr) + 1 > nat(TR.limit) then return 1b
  else call mask = read_tss(16, addr)
       return mask[$n/8+$bit_idx-1:$bit_idx] <> zero($n/8)
```

In legacy mode, the TSS contains the task context registers, the stack pointers, the stack selectors, and the I/O permission bitmap. The bitmap has the same meaning as in the long mode TSS. The stack pointers and the stack selectors correspond to the RSPi stack pointers in the long mode TSS. The stack selectors are necessary because a stack segment in legacy mode has a non-zero base address. Note that there are no interrupt stack pointers (ISTi).

The task context registers are the general purporse registers, the segment registers, and the CR3, LDT registers.

```
layout LegacyTaskState
  field link::bits 16
  field rsv1::bits 16
  field ESP0::bits 32
  field SS0::bits 16
  field rsv2::bits 16
  field ESP1::bits 32
  field SS1::bits 16
  field rsv3::bits 16
  field ESP2::bits 32
  field SS2::bits 16
  field rsv4::bits 16
  field CR3::bits 32
  field EIP::bits 32
  field EFLAGS::bits 32
  field EAX::bits 32
  field ECX::bits 32
  field EDX::bits 32
  field EBX::bits 32
  field ESP::bits 32
  field EBP::bits 32
  field ESI::bits 32
```

```
  field EDI::bits 32
  field ES::bits 16
  field rsv5::bits 16
  field CS::bits 16
  field rsv6::bits 16
  field SS::bits 16
  field rsv7::bits 16
  field DS::bits 16
  field rsv8::bits 16
  field FS::bits 16
  field rsv9::bits 16
  field GS::bits 16
  field rsv10::bits 16
  field LDT::bits 16
  field rsv11::bits 16
  field trap::bit
  field rsv12::bits 15
  field IOPB_BA::bits 16
```

The following action shows how to fetch the new stack corresponding to the new `CPL` from the legacy TSS.

```
action tss_esp_ss(new_cpl::bits 2)::(bits 32, bits 16)
  let addr = (zxt($va, new_cpl) << 3) + bits($va, 4)
  call esp = read_tss(32, addr)
  call ss  = read_tss(16, addr + bits($va, 4))
  return (esp, ss)
```

CHAPTER

FOURTEEN

INSTRUCTION FETCH AND DECODE

14.1 Instruction Format

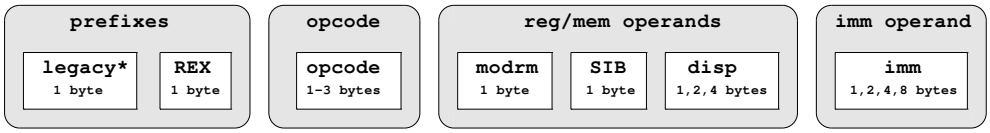

Figure 14.1: Instruction Format

An instruction has four parts: optional prefixes, the opcode, optional register/memory operand references, and an optional immediate operand. The length of each part is not fixed, but the total length cannot exceed 15 bytes. In the following sections we discuss each part in detail.

14.2 Opcode

The opcode of an instruction is preceded by zero or more prefixes, and is followed by the operands. The opcodes have length from 1 to 3 bytes, and the opcode encoding is prefix-free, which means that any valid opcode of length i bytes is not a prefix of a valid i+1-byte opcode. It is achieved by using escape sequences: 1-byte opcode 0Fh is not valid and is used as an escape to 2-byte opcodes; similarly, 2-byte opcodes 0F38h, 0F3Ah are escapes to 3-byte opcodes.

```
layout Opcode
  field byte1::bits 8
  field byte2::bits 8
  field byte3::bits 8
```

byte1 the actual opcode for 1-byte instructions, or `0Fh` escape code for 2- and 3-byte instructions.

byte2 `00h` for 1-byte instructions, the actual opcode for 2-byte instructions, or `38h/3Ah` for 3-byte instructions.

byte3 `00h` for 1- and 2-byte instructions, or byte 3 of the actual opcode for 3-byte instructions.

Some opcodes are shared between several instructions. In such cases, either the prefixes or the ModRM byte help to disambiguate.

14.3 Prefixes

The opcode defines the semantics of the instruction, and prefixes modify the default behavior of the instruction. Any prefix has length one byte and precedes the opcode. An instruction may have an arbitrary number of prefixes, but instructions with length greater than 15 bytes are considered to be invalid. An important fact about prefix values is that they do not collide with the first-byte values of a valid opcode. So, during instruction fetch, it is easy to determine when the prefixes stop and the opcode starts.

There are 12 types of prefixes. All except the REX are called legacy. A legacy prefix can appear several times in the same instruction. A REX prefix can appear at most one time and must immediately precede the opcode.

For storing the fetched prefixes we use the following record. Note that the record has one bit for each legacy prefix. The bit indicates whether the corresponding prefix was fetched or not. Thus, multiple occurrences of the same legacy prefix have the same effect as a single occurrence. The name of a bit-field coincides with the value of the corresponding legacy prefix byte.

```
record Prefix
  field x26::bit
  field x2E::bit
  field x36::bit
  field x3E::bit
  field x64::bit
  field x65::bit
  field x66::bit
  field x67::bit
  field xF0::bit
  field xF2::bit
  field xF3::bit
  field rex::REX
```

x26 (ES segment override prefix)

x2E (CS segment override prefix)

x36 (SS segment override prefix)

x3E (DS segment override prefix)

x64 (FS segment override prefix)

x65 (GS segment override prefix) Instructions, that reference memory, implicitly use one of the segment registers as part of a logical address, and a segment override prefix allows to explicitly specify another segment register. Segment override prefixes can not be used with stack instructions (`PUSH`, `POP`, etc). The use of the prefixes with branch instructions is reserved for branch hints.

x66 (operand-size override prefix) This prefix selects the non-default operand size. See the definition of the `oper_width`.

x67 (address-size override prefix) This prefix selects the non-default address size. See the definition of the `addr_width`.

xF0 (LOCK prefix) Execution of an instruction usually takes several steps, such as fetching operands from the memory, calculating the result, writing the result to the memory. In a multiprocessor environment, it is important to make sure that some instructions execute atomically, i.e. no other processor reads/writes the memory region that is accessed by the instruction. The lock prefix guarantees such atomic execution. It can be used only with the following instructions: `ADC`, `ADD`, `AND`, `BTC`, `BTR`, `BTS`, `CMPXCHG`, `CMPXCHG8B`, `DEC`, `INC`, `NEG`, `NOT`, `OR`, `SBB`, `SUB`, `XADD`, `XCHG`, and `XOR`.

xF2 (REPNE/REPNZ prefix)

xF3 (REP/REPE/REPZ prefix) An instruction with a repeat prefix is executed multiple times until either the `RCX` register becomes zero or the `RFLAGS.ZF` stops to satisfy the condition specified in the name of the prefix. In pseudocode, `REPxx instr` is equivalent to the following loop:

```
while RCX > 0
  instr
  decrement RCX
  if not condition then exit loop
```

where the `condition` depends on the type of the prefix:

- REPNE/REPNZ: `condition = not RFLAGS.ZF`,
- REPE/REPZ: `condition = RFLAGS.ZF`,
- REP: `condition = 1b`.

Whether the value `F3h` is interpreted as the REP prefix or the REPE/REPZ prefix depends on the instruction. Instructions `INS`, `OUTS`, `LODS`, `MOVS`, `STOS` allow only the REP prefix. Instructions `CMPS` and `SCAS` allow only REPNE/REPNZ/REPE/REPZ prefixes. Other instructions do not support repeat prefixes. However, media instructions can be used with the `F2h/F3h` prefixes. In such cases, the prefixes do not repeat the instruction, but disambiguate the opcode, which is mapped to multiple instructions.

rex (REX prefix) This prefix is valid only in 64-bit mode, and can have 16 possible values: `40h..4Fh`. In other modes these byte values are opcodes of the `INC` and `DEC` instructions. Since `00h` is an invalid REX prefix, we set the `rex` field to this value when the instruction has no REX prefix.

The number of general-purpose, XMM, control, and debug registers is doubled in 64-bit mode, and to access these additional registers an instruction needs one more bit in each field specifying a register. The lower nibble of the REX prefix provides these bits.

```
layout REX
  field B::bit
  field X::bit
  field R::bit
  field W::bit
  field rsv::bits 4 reserved and must be 4h
```

B Base, extends `ModRM.rm` and `SIB.base`.

X indeX, extends `SIB.index`.

R Reg, extends `ModRM.reg`.

W Width, selects the 64-bit operand width.

We start the instruction fetch with the empty set of prefixes:

```
function empty_prefix::Prefix
  = Prefix with [x26 = 0b, x2E = 0b, x36 = 0b, x3E = 0b,
           x64 = 0b, x65 = 0b, x66 = 0b, x67 = 0b,
           xF0 = 0b, xF2 = 0b, xF3 = 0b, rex = 00h]
```

For each fetched byte, we check whether the byte is a prefix or not using the following function.

```
function is_prefix(b::bits 8)::bit
  = isREX(b) or b == 26h or b == 2Eh
    or b == 36h or b == 3Eh or b == 64h
    or b == 65h or b == 66h or b == 67h
    or b == F0h or b == F2h or b == F3h
```

In case the byte `b` is a prefix, we update the current set of prefixes `p`, using the following function.

```
function save_prefix(b::bits 8, p::Prefix)::Prefix
  = if b == 26h then p with [x26 = 1b]
    elif b == 2Eh then p with [x2E = 1b]
    elif b == 36h then p with [x36 = 1b]
    elif b == 3Eh then p with [x3E = 1b]
    elif b == 64h then p with [x64 = 1b]
    elif b == 65h then p with [x65 = 1b]
    elif b == 66h then p with [x66 = 1b]
    elif b == 67h then p with [x67 = 1b]
    elif b == F0h then p with [xF0 = 1b]
    elif b == F2h then p with [xF2 = 1b]
    elif b == F3h then p with [xF3 = 1b]
    else p with [rex = b]
```

The following function checks whether a REX prefix was fetched or not.

```
function rex_present(prefix::Prefix)::bit
  = isREX(prefix.rex) and $x64_mode
```

Once all the prefixes are fetched, they are stored in the `_prefix` register. We define the following parameter aliases for the operand and address size override prefixes, because we will refer to them in type expressions.

```
parameter $prefix66::bit = _prefix.x66
parameter $prefix67::bit = _prefix.x67
parameter $REX_W::bit = _prefix.REX.W
invariant not $REX_W or $LMA
```

14.4 ModRM byte

The Mode-Register-Memory byte contains information about the register/memory operands. For some opcodes that are mapped to multiple instructions, the ModRM byte is used as a secondary opcode. The byte has three parts.

```
layout ModRM
  field rm::bits 3
  field reg::bits 3
  field mod::bits 2
```

rm (register/memory) Depending on the value of the `mod` field, this field either contains the index of a register or specifies the addressing mode for the memory operand.

reg (register) This field either contains the index of a register or is a secondary operand.

mod (mode) When this field is `11b`, then the `rm` field specifies a register. Otherwise, this field together with the `rm` field defines the addressing mode for the memory operand.

The presence of the ModRM byte in the instruction depends on the opcode. Thus, after fetching the opcode, we can determine whether we need to fetch the ModRM byte or not. In case the ModRM byte is fetched, it is stored in the `_modrm` register.

We will refer to the `mod` field in the context of type expressions, therefore, we define a parameter alias.

```
parameter $mod::bits 2 = _modrm.mod
```

14.5 SIB byte

The Scale-Index-Base byte is used for memory operand address computation.

```
layout SIB
  field base::bits 3
  field index::bits 3
  field scale::bits 2
```

base specifies the base register.

index specifies the index register.

scale specifies the scale factor applied to the index.

After fetching the ModRM byte, we can determine whether there is an SIB byte or not. The SIB is present when there is a memory operand and the `rm` field specifies the SIB addressing mode and the address width is not 16 bits. More formally, the following conditions must hold if the SIB byte is present:

- The memory operand is present when the `mod` field of the ModRM byte is less the three.
- The SIB addressing mode is active when the `rm` field of the ModRM byte is equal to four.
- The address width is not equal to 16 bits if
  - either the default address width is not 16 bits and there is not width override prefix,
  - or the default address width is 16 bits and there is a width override prefix.

```
function has_sib(prefix::Prefix, opcode::Opcode, modrm::ModRM)::bit
  = modrm.mod < 11b and modrm.rm == 100b
    and has_memop(prefix, opcode, modrm)
    and ($def_addr_width == 16 and prefix.x67
       or $def_addr_width > 16 and (prefix.rex.W or not prefix.x67))
```

The function `has_memop` is defined in section 14.8. The function `$def_addr_width` is defined in section 14.10.

Generally, the SIB byte defines the memory address as the following expression:

```
R[base] + (R[index] << scale).
```

However, there are special cases. See the definition of the `sib` function for details.

14.6 Displacement

A displacement is a constant that is added to the linear combination of register to obtain the memory operand address. The number of displacement byte in the instruction depends on the addressing mode (`mod, rm, base`) and the operand address width (`$oa`). The following function computes the displacement width.

```
function disp_width::{0, 8, 16} when $oa == 16
  = if _modrm.mod == 00b and _modrm.rm == 110b then 16
    elif _modrm.mod == 01b then 8
    elif _modrm.mod == 10b then 16
    else 0
function disp_width::{0, 8, 16, 32} when $oa > 16
  = if _modrm.mod == 00b and _modrm.rm == 101b then 32
    elif _modrm.mod == 00b and _modrm.rm == 100b and _sib.base == 101b then 32
    elif _modrm.mod == 01b then 8
    elif _modrm.mod == 10b then 32
    else 0
```

The fetched displacement is sign-extended and stored in the `_disp` register. The following function retrieves the displacement from the `_disp` register.

```
function disp::bits $oa
  = if disp_width == 0 then (zero($oa))
    else _disp[$oa-1:0]
```

14.7 Immediate Operand

An immediate operand is an operand that is embedded into the instruction. After fetching the opcode and possibly the ModRM byte, we can determine the presence and the size of the immediate operand. In the next section we will define the `imm_width` function. The fetched immediate operand is zero-extended and stored in the `_imm` register.

14.8 Opcode Table

In the appendix chapters we will describe instructions. For each instruction we list opcodes and operands. Based on this information, it is possible to construct the opcode table, which maps opcodes to operand types, operand widths, instruction attributes, and actions. The following functions, that query the opcode table, can be derived from the opcode table. Therefore, we do not explicitly define them in this document.

After fetching the prefixes and the opcode, we use the `has_modrm` function to check whether the ModRM byte is present or not. In case the prefixes or the opcode are invalid, the function returns `0b`.

```
function has_modrm(prefix::Prefix, opc::Opcode)::bit
  = implicit_definition
```

The prefixes, the opcode, and the ModRM byte (if present) uniquely determine the instruction. The following function check whether the instruction is valid or not. In case the instruction does not have the ModRM byte, the `modrm` argument should be `00h`.

```
function valid_instr(prefix::Prefix, opc::Opcode, modrm::ModRM)::bit
  = implicit_definition
```

In the definition of each instruction we give a list of attributes. We then use the following function to get the attributes of an instruction.

```
function instr_attr(prefix::Prefix, opc::Opcode, modrm::ModRM)::InstrAttr
  = implicit_definition
```

There are three instruction attributes.

```
record InstrAttr
  field serializing::bit
  field default64::bit
  field lockable::bit
```

`serializing` Serializing instructions enforce strong memory ordering. All reads and writes of the serializing instruction and of the previous instructions finish before the first memory access of the next instruction.

`default64` In 64-bit mode, the default operand width of the most instructions is 32 bits. However, instructions with the `default64` attribute have the 64-bit default operand width.

`lockable` Instructions with this attribute are executed atomically when they have the LOCK prefix.

Based on this function, we can define predicates that check the type of the current instruction stored in the `_opcode` register:

```
function serializing()::bit
  = instr_attr(_prefix, _opcode, _modrm).serializing
function default64()::bit
  = instr_attr(_prefix, _opcode, _modrm).default64
function lockable()::bit
  = instr_attr(_prefix, _opcode, _modrm).lockable
```

We also use several predicate that check whether the current instruction is a memory fence, a store fence, a load fence, or an io instruction.

```
function mfence()::bit
  = (_opcode == 000FAEF0h)
function sfence()::bit
  = (_opcode == 000FAEF8h)
function lfence()::bit
  = (_opcode == 000FAEE8h)
function io()::bit
  = (_opcode == 000000E4h or
     _opcode == 000000E5h or
     _opcode == 000000ECh or
     _opcode == 000000EDh or
     _opcode == 000000E6h or
     _opcode == 000000E7h or
     _opcode == 000000EEh or
     _opcode == 000000EFh or
     _opcode == 0000006Ch or
     _opcode == 0000006Dh or
     _opcode == 0000006Eh or
     _opcode == 0000006Fh
    )
```

An instruction has up to three operands. In the instruction definition, we specify the type and the width of each operand. The following functions are mechanically generated.

```
function op1type(prefix::Prefix, opc::Opcode, modrm::ModRM)::OpType
  = generated_type
function op2type(prefix::Prefix, opc::Opcode, modrm::ModRM)::OpType
  = generated_type
function op3type(prefix::Prefix, opc::Opcode, modrm::ModRM)::OpType
  = generated_type
function op1width(prefix::Prefix, opc::Opcode, modrm::ModRM)::OpWidth
  = generated_width
function op2width(prefix::Prefix, opc::Opcode, modrm::ModRM)::OpWidth
  = generated_width
function op3width(prefix::Prefix, opc::Opcode, modrm::ModRM)::OpWidth
  = generated_width
```

The functions `op1type, op2type, op3type` return `none` in case the instruction does not have the corresponding operand. The `OpType` and the `OpWidth` are sets of operand types and operand widths.

```
set OpType = {reg, reg_rm, reg_mem, mem, imm, imm2, mem_ptr,
              mem_pair, imm_ptr, es_rdi, ds_rsi, creg, sreg, dreg, mmx, mmx_rm,
              mmx_mem, xmm, xmm_rm, xmm_mem, moffset, rel_off, rax, rcx, rdx,
              rbx, rsp, rbp, rsi, rdi, rax_r8, rcx_r9, rdx_r10, rbx_r11, rsp_r12,
              rbp_r13, rsi_r14, rdi_r15, rflags, es, cs, ss, ds, fs, gs, cr8,
              const_0, const_1, none}
set OpWidth = {_0 = 0, _8 = 8, _16 = 16, _32 = 32, _48 = 48,
               _64 = 64, _80 = 80, _128 = 128}
```

Section 14.13 describes the meaning of each operand type. We conclude this section by defining a helper function that checks whether the instruction has a memory operand or not.

```
function has_memop(prefix::Prefix, opcode::Opcode, modrm::ModRM)::bit
  = is_memop(op1type(prefix, opcode, modrm))
    or is_memop(op2type(prefix, opcode, modrm))
    or is_memop(op3type(prefix, opcode, modrm))
function is_memop(op::OpType)::bit
  = op == reg_mem or op == mem or op == mem_ptr or op == mmx_mem or op == xmm_mem
```

14.9 Instruction Fetch

The instruction is fetched in two stages. The first stage fetches

- the prefixes,
- the opcode,
- the ModRM byte (if present),
- the SIB byte (if present).

The fetched bytes are stored in the `_prefix`, `_opcode`, `_modrm`, and `_sib` registers. The second stage uses these registers to determine the size of the displacement and the size of the immediate operand. Afterwards, the displacement and the immediate operand are fetched (if present), and the `RIP` is set to the next instruction. Thus, the `fetch` action has the following definition.

```
action fetch
  call fetch1
  chain fetch2
```

The **`chain`** statement calls the specified action with the updated registers. This means that all register writes made by the `fetch1` are visible in the `fetch2`. The complicated business with two stages is necessary because the operand address size and the operand width depend on prefixes and on the `default64` attribute of the instruction. There are two solutions to this problem:

- either provide the prefixes as explicit arguments to the `$addr_width, $oper_width` functions.
- or save the prefixes into a register so that the functions access them implicitly.

We have chosen the second approach because it simplifies many definitions.

Besides what is described above, the `fetch1` action checks that the instruction is valid and writes the instruction attributes and the intermediate instruction length into the `_instr_attr` and the `_instr_len` registers.

```
action fetch1
  call (len1, byte1, prefix) = fetch_prefixes
  call (len2, opc) = fetch_opcode(len1, byte1)
  call (len3, modrm) = fetch_modrm(len2, prefix, opc)
  fail exception(xUD, 0000h) when not valid_instr(prefix, opc, modrm)
  write prefix to _prefix
  write opc to _opcode
  write modrm to _modrm
  write instr_attr(prefix, opc, modrm) to _instr_attr
  write bits(4, len3) to _instr_len
```

The `fetch2` action saves the displacement and the immediate operand in the `_disp` and the `_imm` registers. It also updates the `_instr_len`, the `_old_RIP`, and the `RIP` registers.

```
action fetch2
  call (len1, disp) = fetch_disp
  call (len2, imm) = fetch_imm(len1)
  write disp to _disp
  write imm to _imm
  write bits(4, len2) to _instr_len
  write RIP to _old_RIP
  write RIP + bits(64, len2) to RIP
```

The prefixes are fetched byte by byte until either the instruction length exceeds 15 bytes or the fetched byte is the first byte of the opcode. In pseudocode the `fetch_prefixes` action could be defined as follows

```
byte := read byte at RIP
length := 1
prefix := empty_prefix
while length < 15 and (byte is a prefix)
  prefix := save_prefix(byte, prefix)
  byte := read byte at RIP + length
  length := length + 1
return (length, byte, prefix)
```

Since our DSL does not have loop constructs, we manually unroll the loop.

```
action fetch_prefixes::(nat, bits 8, Prefix)
  call byte00 = lread(code, 8, CS, $la, RIP[$la-1:0])
  call (len01, byte01, prefix01) = fetch_prefix(1, byte00, empty_prefix)
  call (len02, byte02, prefix02) = fetch_prefix(len01, byte01, prefix01)
  call (len03, byte03, prefix03) = fetch_prefix(len02, byte02, prefix02)
  call (len04, byte04, prefix04) = fetch_prefix(len03, byte03, prefix03)
  call (len05, byte05, prefix05) = fetch_prefix(len04, byte04, prefix04)
  call (len06, byte06, prefix06) = fetch_prefix(len05, byte05, prefix05)
  call (len07, byte07, prefix07) = fetch_prefix(len06, byte06, prefix06)
  call (len08, byte08, prefix08) = fetch_prefix(len07, byte07, prefix07)
  call (len09, byte09, prefix09) = fetch_prefix(len08, byte08, prefix08)
  call (len10, byte10, prefix10) = fetch_prefix(len09, byte09, prefix09)
```

```
  call (len11, byte11, prefix11) = fetch_prefix(len10, byte10, prefix10)
  call (len12, byte12, prefix12) = fetch_prefix(len11, byte11, prefix11)
  call (len13, byte13, prefix13) = fetch_prefix(len12, byte12, prefix12)
  call (len14, byte14, prefix14) = fetch_prefix(len13, byte13, prefix13)
  fail exception(xUD, 0000h) when is_prefix(byte14)
  return (len14, byte14, prefix14)
```

The `fetch_prefix` action is called 14 times. As arguments the `fetch_prefix` action expects the number of fetched bytes, the last fetched byte and the accumulated prefixes. In case the action detects that the last fetched byte is the first byte of the opcode, the it does nothing and returns the arguments intact. Otherwise, it adds the last fetched byte into prefixes and fetches another byte.

```
action fetch_prefix(len::nat, last::bits 8, prefix::Prefix)::(nat, bits 8, Prefix)
  if is_prefix(last) and not rex_present(prefix) then
    let offset = RIP[$la-1:0] + bits($la, len)
    call next = lread(code, 8, CS, $la, offset)
    return (len+1, next, save_prefix(last, prefix))
  else return (len, last, prefix)
```

The first byte of the opcode was already fetched either by the `fetch_prefixes`. If this byte is an escape byte, then the `fetch_opcode` action fetches the second byte of the opcode. If the second byte is also an escape byte, then the third byte is fetched. In any case, the action zero-extends the opcode and returns a 3-byte opcode.

```
action fetch_opcode(len::nat, last::bits 8)::(nat, Opcode)
  if last <> 0Fh then return (len, zxt(24, last))
  else fail exception(xUD, 0000h) when len == 15
     call byte2 = lread(code, 8, CS, $la, RIP[$la-1:0] + bits($la, len))
     if byte2 <> 38h and byte2 <> 3Ah then return (len+1, zxt(24, byte2++last))
     else fail exception(xUD, 0000h) when len+1 == 15
      call byte3 = lread(code, 8, CS, $la, RIP[$la-1:0] + bits($la, len+1))
      return (len+2, byte3++byte2++last)
```

The presence of the ModRM byte is detected by the `has_modrm` function that looks up the opcode table. The following action returns the ModRM byte in case it is present, or returns `00h` otherwise.

```
action fetch_modrm(len::nat, prefix::Prefix, opc::Opcode)::(nat, ModRM)
  if not has_modrm(prefix, opc) then return (len, 00h)
  else fail exception(xUD, 0000h) when len == 15
       call modrm = lread(code, 8, CS, $la, RIP[$la-1:0] + bits($la, len))
       return (len+1, modrm)
```

The presence of the SIB byte is detected by the `has_sib` function that we defined in section 14.5. The following action returns the SIB byte in case it is present, or returns `00h` otherwise.

```
action fetch_sib(len::nat, prefix::Prefix, opc::Opcode, modrm::ModRM)::(nat, SIB)
  if not has_sib(prefix, opc, modrm) then return (len, 00h)
  else fail exception(xUD, 0000h) when len == 15
       call sib = lread(code, 8, CS, $la, RIP[$la-1:0] + bits($la, len))
       return (len+1, sib)
```

Recall that the `fetch_disp` and the `fetch_imm` actions are invoke during the second stage of the instruction fetch. This means that the operand address width is well-defined and we can use the `disp_width` function (which depends on the operand address width). We defined the `disp_width` function in section 14.6. The following action fetches the displacement.

```
action fetch_disp::(nat, bits 64)
  let len = nat(_instr_len)
  if disp_width == 0 then return (len, zero(64))
  else let $n = disp_width::{8, 16, 32}
       fail exception(xUD, 0000h) when len + $n/8 > 15
       call disp = lread(code, $n, iCS, $la, RIP[$la-1:0] + bits($la, len))
       return (len+$n/8, sxt(64, disp))
```

We determine the width of the immediate operand by examining all three operands and checking whether any of the is an immediate operand or not.

```
function imm_width::OpWidth
  = imm1_width + imm2_width + imm3_width
function imm1_width::OpWidth
  = if is_immop(op1type(_prefix, _opcode, _modrm)) then
      op1width(_prefix, _opcode, _modrm)
    else 0
function imm2_width::OpWidth
  = if is_immop(op2type(_prefix, _opcode, _modrm)) then
      op2width(_prefix, _opcode, _modrm)
    else 0
function imm3_width::OpWidth
  = if is_immop(op3type(_prefix, _opcode, _modrm)) then
      op3width(_prefix, _opcode, _modrm)
    else 0
function is_immop(op::OpType)::bit
  = op == imm or op == imm2 or op == imm_ptr or op == moffset or op == rel_off
```

The following action fetches the immediate operand.

```
action fetch_imm(len::nat)::(nat, bits 64)
  let $n = imm_width::OpWidth
  if $n == 0 then return (len, zero(64))
  else fail exception(xUD, 0000h) when len + $n/8 > 15
       call imm = lread(code, $n, CS, $la, RIP[$la-1:0] + bits($la, len))
       return (len+$n/8, zxt(64, imm))
```

14.10 Operand Width

When defining an instruction, we specify for each operand its width. The width is either a constant or an expression that uses one of the following functions:

- `$v` — effective operand width,
- `$vw` — the same as `$v` for register operands and 16 bits for memory operands,
- `$z` — the same as `$v` but capped to 32 bits,

- `$qd` — quad word in 64-bit mode, and double word in other modes.

```
function $v = $oper_width
function $vw = if $mod == 11b then $v
               else 16
function $z = if $oper_width == 16 then 16
              else 32
function $qd = if $x64_mode then 64
               else 32
```

The effective operand width is computed as follows. If the `W` bit of the REX prefix is set then the effective width is 64 bits. Otherwise, the width depends on the presence of the `66h` prefix. If the prefix is not present, then the effective width is equal to the default operand width. Otherwise, the effective width is equal to the non-default operand width.

```
function $oper_width
   = if $x64_mode and $REX_W then 64
     else if not $prefix66 then $def_oper_width
          else $flip($def_oper_width)
```

Where the `$flip` function selects the non-default width, given the default width.

```
function $flip(x::{16, 32})
   = if x == 16 then 32
     else 16
```

In 64-bit mode the default operand width depends on the `default64` attribute of the current instruction. In other modes, the width depends on the `D/B` bit of the current code segment descriptor.

```
function $def_oper_width
   = if not $x64_mode and not $D then 16
     else if not $x64_mode and $D then 32
          else if not $default64 then 32
               else 64
```

Where the `$D` is an alias for the `D/B` bit of the code segment descriptor, and the `$default64` is an alias for the `default64` attribute of the current instruction.

```
parameter $D::bit = CS.attr.DB
invariant not ($L and $D)
parameter $default64::bit = _instr_attr.default64
```

14.11 Memory Operand Address Width

The effective address width of the memory operand depends on the presence of the `67h` prefix. If the prefix is not present, then the effective width is equal to the default address width. Otherwise, the effective width is equal to the non-default address width.

```
function $addr_width
    = if not $prefix67 then $def_addr_width
      else $flip($def_addr_width)
```

We will use a short alias `$oa` (operand address) instead of `$addr_width`.

The default address in 64-bit mode is always 64 bits. In other modes, the default address width depends on the `D/B` bit of the code segment descriptor.

```
function $def_addr_width
    = if not $x64_mode and not $D then 16
      else if not $x64_mode and $D then 32
           else 64
```

14.12 Memory Operand Address

The logical address of the memory operand consists of two parts: the segment register and the offset. The index of the segment register is computed by the `segment` function, the offset is computed by the `ea` function (effective address). Both computations depend on the addressing mode specified by the ModRM byte.

There are two sets of addressing modes. Legacy addressing modes are enabled when the operand address width is 16 bits (`$oa==16`). Legacy addressing modes do not use the SIB byte, and the `rm` field of the ModRM byte selects a linear combination of the `RBX`, `RBP`, `RSI`, `RDI` registers and the displacement. Note that the `disp` function returns zero when there is no displacement.

```
function ea::bits 16 when $oa == 16
  = if _modrm.mod == 00b and _modrm.rm == 110b then disp
    elif _modrm.rm == 000b then RBX[15:0] + RSI[15:0] + disp
    elif _modrm.rm == 001b then RBX[15:0] + RDI[15:0] + disp
    elif _modrm.rm == 010b then RBP[15:0] + RSI[15:0] + disp
    elif _modrm.rm == 011b then RBP[15:0] + RDI[15:0] + disp
    elif _modrm.rm == 100b then RSI[15:0] + disp
    elif _modrm.rm == 101b then RDI[15:0] + disp
    elif _modrm.rm == 110b then RBP[15:0] + disp
    else RBX[15:0] + disp
```

When the operand address width is 32 or 64 bits, the effective address is computed as follows.

```
function ea::bits $oa when $oa > 16
  = if $x64_mode and _modrm.mod==00b and _modrm.rm==101b then RIP[$oa-1:0] + disp
    elif _modrm.rm <> 100b and (_modrm.mod <> 00b or _modrm.rm <> 101b) then
      R[_prefix.rex.B++_modrm.rm][$oa-1:0] + disp
    else sib + disp
```

Thus, the effective address is the displacement added to either the instruction pointer, or a general-purpose register, or a combination of the general-purpose registers specified by the SIB byte. The SIB combination is `R[base] + (R[index] << scale)`. However, there is special case when the index is `0100`.

```
function sib::bits $oa when $oa > 16
  = if _prefix.rex.X == 0b and SIB.index == 100b then ebase
    else ebase + (R[_prefix.rex.X ++ SIB.index][$oa-1:0] << nat(SIB.scale))
```

The base component of the SIB combination is either zero or a general-purpose register selected by the `base` field of the SIB byte.

```
function ebase::bits $oa when $oa > 16
  = if _modrm.mod == 00b and SIB.base == 101b then (zero($oa))
    else R[_prefix.rex.B ++ SIB.base][$oa-1:0]
```

For segment computation, we take into account the segment override prefixes. The `segment` function takes the default segment register index and returnes the effective index after applying the segment override prefixes.

```
function segment(def::bits 3)::bits 3
  = if _prefix.x26 and not $x64_mode then iES
    elif _prefix.x2E and not $x64_mode then iCS
    elif _prefix.x36 and not $x64_mode then iSS
    elif _prefix.x64 then iFS
    elif _prefix.x65 then iGS
    else def
```

The default segment register depends on the operand type. When the operand type is `es_rdi/ds_rsi` the default segment register is the `ES/DS` segment. Otherwise, it either the data segment `DS` or the stack segment `SS`. If any register except `RBP` and `RSP` was used as the base register for effective address computation, then the default segment register is the `DS`, otherwise it is the `SS`. The following function computes the default segment when the operand type is not `es_rdi/ds_rsi`.

```
function def_segment::bits 3 when $oa == 16
  = if _modrm.mod <> 00b and _modrm.rm == 110b then iSS
    else iDS
function def_segment::bits 3 when $oa > 16
  = if _prefix.rex.B == 0b then
      if _modrm.mod <> 00b and _modrm.rm == 101b then iSS
      elif _modrm.rm == 100b and _sib.base == 100b then iSS
      elif _modrm.rm == 100b and _sib.base == 101b and _modrm.mod <> 00b then iSS
      else iDS
    else iDS
```

14.13 Operand Decode

In this section we describe two actions that read and write instruction operands:

```
action read_op($w::OpWidth, t::OpType)::bits $w
action write_op($w::OpWidth, value::bits $w, t::OpType)
```

Given the operand width and type, the first action fetches the operand. The second action writes the new value to the given operand. The definition of both actions is rather long because we need to make case discrimination on operand type, and there are many operand types. We put the full definition of the action into Appendix R. In this section we describe operand types and define helper actions for reading and writing each operand type.

Table 14.1 shows the meaning of each operand. There are eight groups of operand types:

1. general-purpose registers,

Type	Meaning
reg	R[_prefix.rex.R++_modrm.reg]
reg_rm	R[_prefix.rex.B++_modrm.rm] **and** _modrm.mod==11b
reg_mem	R[_prefix.rex.B++_modrm.rm] **if** _modrm.mod==11b
reg_mem	memory at (segment, ea) **if** _modrm.mod<>11b
mem	memory at (segment, ea) **and** _modrm.mod<>11b
imm	immediate operand in the instruction
imm2	second immediate operand in the instruction
mem_ptr	(sel, offset) pair in memory at (segment, ea)
mem_pair	pair in memory at (segment, ea)
imm_ptr	(sel, offset) pair in the instruction
es_rdi	memory at (ES, RDI)
ds_rsi	memory at (segment, RSI)
creg	CR[_prefix.reg.R++_modrm.reg]
sreg	SR[_modrm.reg]
dreg	DR[_modrm.reg]
mmx	FPR[_modrm.reg]
mmx_rm	FPR[_modrm.rm] **and** _modrm.mod==11b
mmx_mem	FPR[_modrm.rm] **if** _modrm.mod==11b
mmx_mem	memory at (segment, ea) **if** _modrm.mod<>11b
xmm	XMM[_prefix.rex.R++_modrm.reg]
xmm_rm	XMM[_prefix.rex.B++_modrm.rm] **and** _modrm.mod==11b
xmm_mem	XMM[_prefix.rex.B++_modrm.rm] **if** _modrm.mod==11b
xmm_mem	memory at (segment, ea) **if** _modrm.mod<>11b
moffset	memory at (DS, _imm)
rel_off	relative offset in the instruction
rax..r15	R[0++i]
rax_r8..rdi_r15	R[_prefix.rex.B++i]
rflags	RFLAGS
cs..gs	SR[i]
cr8	CR[1000b]
const_0	constant operand with value 0
const_1	constant operand with value 1

Table 14.1: Operand Decode

2. segment register,
3. control registers,
4. debug registers,
5. floating-point and multimedia registers,

6. memory operands,

7. immediate operands,

8. constants: `0` and `1`.

Constant and immediate operand are read-only. Memory operands are read and written using logical memory access actions `lread` and `lwrite`. For each type of register operand, we define separate read/write actions which are then invoked in the `read_op`/`write_op` actions.

The following operand types specify general-purpose registers:

```
reg, reg_rm, reg_mem (when _modrm.mod==11b),
rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi,
rax_r8, rcx_r9, rdx_r10, rbx_r11, rsp_r12, rbp_r13, rsi_r14, rdi_r15.
```

For the first three types, the REX and the ModRM provide the register index. For the remaining types, the index is computed as follows.

```
function encode_gpr(t::OpType)::bits 4
  = if   t == rax then 0000b
    elif t == rcx then 0001b
    elif t == rdx then 0010b
    elif t == rbx then 0011b
    elif t == rsp then 0100b
    elif t == rbp then 0101b
    elif t == rsi then 0110b
    elif t == rdi then 0111b
    elif t == rax_r8  then _prefix.rex.B ++ 000b
    elif t == rcx_r9  then _prefix.rex.B ++ 001b
    elif t == rdx_r10 then _prefix.rex.B ++ 010b
    elif t == rbx_r11 then _prefix.rex.B ++ 011b
    elif t == rsp_r12 then _prefix.rex.B ++ 100b
    elif t == rbp_r13 then _prefix.rex.B ++ 101b
    elif t == rsi_r14 then _prefix.rex.B ++ 110b
    else _prefix.rex.B ++ 111b
```

One would expect that reading `$w` bits from the register with the index `idx` is simply `R[idx][$w-1:0]`. However, the architecture allows to access the second byte of the first eight registers when there is no REX prefix.

```
action read_gpr($w::{8, 16, 32, 64}, idx::bits 4)::bits $w
  if $w > 8 or rex_present(_prefix) or idx < 0100b then return R[idx][$w-1:0]
  else assume $w == 8
       return R[idx & 0011b][15:8]
```

Writing a general-purpose register is further complicated with the fact that 32-bit writes are zero-extended to 64-bit writes.

```
action write_gpr($w::{8, 16, 32, 64}, val::bits $w, idx::bits 4)
  if $w > 8 or rex_present(_prefix) or idx < 0100b then
    write val to R[idx][$w-1:0]
    if $w == 32 then write zero(32) to R[idx][63:32]
  else assume $w == 8
       write val to R[idx & 0011b][15:8]
```

When we want to write a register by its name rather than by its index, we use the following pattern:

```
write gpr($w, x, RSP) to RSP
```

where the function `gpr` takes care of zero-extending 32-bit values.

```
function gpr($w::{8, 16, 32, 64}, new_val::bits $w, old_val::bits 64)::bits 64
  = if $w >= 32 then zxt(64, new_val)
    else old_val[63:$w]++new_val[$w-1:0]
```

The operand types `sreg`, `es`, `cs`, `ss`, `ds`, and `fs` refer to the segment registers. The index of the segment register for the `sreg` operand type is equal to `_modrm.reg`. For the other types, the index is computed as follows.

```
function encode_sr(t::OpType)::bits 3
  = if t == es then 000b
    elif t == cs then 001b
    elif t == ss then 010b
    elif t == ds then 011b
    elif t == fs then 100b
    else 101b
```

Writing segment registers is complicated because it involves fetching a descriptor from the descriptor. We define actions `read_sr` and `write_sr` in chapter L.

The operand types `creg` and `cr8` refer to the control registers. The register index for the `creg` type is equal to `_prefix.rex.R++_modrm.reg`. For the `cr8` type the register index is `1000b`. We define actions `read_cr` and `write_cr` in chapter N.

The operand types `xmm*`, `mmx*`, `dreg` refer to media registers and debug registers. As we do not model media and debug features, we give dummy definitions for reading and writing these registers.

```
action write_dr($w::{32, 64}, val::bits $w, idx::bits 3)
  write zxt(64, val) to DR[idx]
action read_dr($w::{32, 64}, idx::bits 3)::bits $w
  return DR[idx][$w-1:0]
action write_mmx($w::{8, 16, 32, 64}, val::bits $w, idx::bits 3)
  write val to FPR[idx][$w-1:0]
action read_mmx($w::{8, 16, 32, 64}, idx::bits 3)::bits $w
  return FPR[idx][$w-1:0]
action write_xmm($w::{8, 16, 32, 64, 128}, val::bits $w, idx::bits 4)
  write val to XMM[idx][$w-1:0]
action read_xmm($w::{8, 16, 32, 64, 128}, idx::bits 4)::bits $w
  return XMM[idx][$w-1:0]
```

We already discussed address computation for memory operands in previous sections. For reading and writing the logical memory we need to provide the origin of the access as one of the arguments. Since we are reading/writing the memory operand, the origin is either `data` or `fsgs` depending on the chosen segment register.

```
function segment_origin(x::SRIndex)::Origin
  = if x == iFS or x == iGS then fsgs
    else data
```

Note that even if the memory operand is in the code or the stack segment, the origin is `data`, because the memory operand is data.

CHAPTER

FIFTEEN

STACK AND STACK OPERATIONS

The current stack is defined by two registers: `SS` and `RSP`. The `SS` register specifies the stack segment. The `RSP` register points to the top of the stack, which is the offset of the last pushed data. The stack grows downwards, i.e. a push operation decrements the `RSP`, and a pop operation increments the `RSP`. The width of the stack pointer depends on operating mode, and we denote it as `$sa` (stack address width, see section 13.3).

We define a push operation as an action that

- takes the data width in bits, the data, and the current stack pointer,
- decrements the stack pointer by the data width,
- writes the given data into the stack segment at the new stack pointer.

```
action push($n::Width, val::bits $n, rsp::bits $sa)::bits $sa
  let next_rsp = rsp - bits($sa, $n)
  call lwrite(stack, $n, val, SS, $sa, next_rsp)
  return next_rsp
```

A pop operation does the opposite of a push operation:

- it takes the data width in bits and the current stack pointer,
- it reads the stack segment at the current stack pointer,
- increments the stack pointer by the data width,
- returns the read data and the new stack pointer.

```
action pop($n::Width, rsp::bits $sa)::(bits $n, bits $sa)
  call val = lread(stack, $n, SS, $sa, rsp)
  return (val, rsp + bits($sa, $n))
```

For the sake of convenience, we define two more actions that perform push/pop conditionally depending of the given flag:

```
action push_if(cond::bit, $n::Width, val::bits $n, rsp::bits $sa)::bits $sa
  if cond then call next_rsp = push($n, val, rsp)
               return next_rsp
  else return rsp
action pop_if(cond::bit, $n::Width, rsp::bits $sa)::(bits $n, bits $sa)
  if cond then call (val, next_rsp) = pop($n, rsp)
               return (val, next_rsp)
  else return (zero($n), rsp)
```

15.1 Inner Stack

In legacy mode, the task segment stores a stack segment selector and a stack pointer for each system privilege level. So, there are three 48-bit strings in the task segment. The upper 16-bits of a bit string contain the stack selector, the remaining bits contain the stack pointer. In long mode, a stack is specified only by the 64-bit stack pointer and the stack selector is assumed to be null. The task segment stores three stacks corresponding to the privilege levels. But in addition to it, the task segment also stores stacks in the interrupt-stack-table (IST). These stacks are used to during control transfer to an interrupt handler if the `ist` field of the gate descriptor is not zero. Taking all this into account, we define the `inner_stack` function that returns the new stack corresponding to the specified `cpl` and `ist` values. In long mode, if `ist` is not zero then the stack is read from IST, otherwise the stack corresponding to the specified privilege level is returned. The stack is returned as a pair of stack segment register and stack pointer. Therefore, in legacy mode, the `inner_stack` function fetches the descriptor corresponding to the stack selector and performs validity checks.

```
action inner_stack(ist::bits 3,
                   new_cpl::bits 2)::(bits 16, DataSegment, bits 64) when $long_mode
  if ist == 000b and new_cpl == CPL then
    return (SS.sel, desc_from_sr(SS), RSP & FFFFFFFFFFFFFFF0h)
  else call rsp = tss_rsp(new_cpl, ist)
       return (zxt(16, new_cpl), null_stack_desc, rsp)
```

The `null_stack_desc` function returns a dummy data descriptor with attributes that specify a present, writable data segment. The function is defined in section 13.7. The `tss_rsp` action reads the stack pointer from the IST and is defined in section 13.14.

The `desc_from_sr` function takes a segment register and reconstructs a data segment descriptor from the segment register components. It unscales the limit and copies the attribute bits.

```
function desc_from_sr(sr::SR)::DataSegment
  = let attr = sr.attr::DataAttr in
    let limit = (if attr.G then sr.limit[31:12]
                 else sr.limit[19:0]) in
    DataSegment with [limit1 = limit[15:0], base1 = sr.base[23:0],
                      A = attr.A, W = attr.W, E = attr.E,
                      DPL = attr.DPL, P = attr.P,
                      limit2 = limit[19:16], AVL = attr.AVL,
                      DB = attr.DB, G = attr.G,
                      base2 = sr.base[31:24]]
```

In legacy mode, the `inner_stack` action reads both the stack selector and the stack pointer from the task state segment and fetches the stack descriptor from the descriptor table.

```
action inner_stack(ist::bits 3,
                   new_cpl::bits 2)::(bits 16, DataSegment, bits 64) when $legacy_mode
  call (esp, sel) = tss_esp_ss(new_cpl)
  call (desc, desc_valid) = read_desc(sel)
  fail exception(xTS, sel & FFF8h) when not desc_valid
  call check_stack(task, sel, desc, new_cpl, 0b)
  return (sel, desc, zxt(64, esp))
```

The `tss_esp_ss` action is defined in section 13.14. This action reads the stack selector and the stack pointer corresponding to the given privilege level from the TSS.

CHAPTER

SIXTEEN

FAR CONTROL TRANSFER

This chapter describes the mechanism for control transfer between two code segments. Such control transfer is called *far control transfer*, and it occurs during a far jump, a far procedure call, and an interrupt handler call. In all three cases, control transfer follows the same general algorithm, although each case has its own peculiarities:

1. The jump and procedure call instructions take the target of control transfer as a pair of `(selector, offset)`, where the `selector` defines the target code segment and the `offset` defines the offset of the target instruction in the target code segment. In case of an interrupt handler call, the address of the handler is computed based on the interrupt `vector`, which defines the target entry in the interrupt descriptor table.
2. In case of an interrupt handler/procedure call, the return address is saved in the stack.
3. A descriptor `desc` is fetched from the descriptor table using the selector or the `vector` as an index.
4. Privilege level checks are performed to ensure that the current code is allowed to access the `desc`.
5. The rest of control transfer depends on the type of the `desc` descriptor.
    - if `desc` is a code segment then it is the target code segment (this case cannot happen during an interrupt handler call):
        - load the `selector` and `desc` into the `CS` register,
        - load the `offset` into the `RIP` register.
    - if `desc` is a call-gate descriptor then the target code segment selector and offset are stored in `desc` (this case cannot happen during interrupt handler call):
        - fetch the target code segment descriptor `cs_desc` from the descriptor table using `desc.sel`,
        - check that the current code is allowed to access the fetched descriptor,

    - switch the stacks and copy parameters if necessary,
    - load `desc.sel` and `cs_desc` into the `CS` register,
    - load `desc.offset` into the `RIP` register,
    - update the `CPL` according to the `cs_desc.DPL`.
  - if `desc` is a task segment descriptor, then perform a task switch by loading register states from the task segment.
  - if `desc` is a task gate descriptor, then the selector of the target task segment is stored in `desc.sel`:
    - fetch the task segment descriptor `tss_desc` from the descriptor table using `desc.sel`.
    - check that the current code is allowed to access the fetched descriptor.
    - perform a task switch to the task defined by `tss_desc`.
  - if `desc` is a trap gate descriptor then perform all the steps from the call-gate descriptor case. Additionally, save the flags register into the stack and if there is an error code associated with `vector` then push the error code into stack. This case can happen only during an interrupt handler call.
  - if `desc` is an interrupt gate descriptor then disable maskable interrupt and perform the same steps as in the trap gate descriptor case. This case can happen only during an interrupt handler call.

It is possible to return from an interrupt handler or a procedure using the `IRET` and the `RETF` instructions, which perform control transfer to the return address loaded from the stack. In the subsequent sections we formally define

- far jump,
- far procedure call,
- interrupt handler call,
- return from a procedure,
- return from an interrupt handler.

16.1 Far Jump

In 16-bit mode, the far jump instruction follows a simplified scheme. It computes the base address of the target code segment directly from the given selector. The current privilege level and the attributes of the code segment remain unchanged:

```
action jmp_far(sel::Selector, offset::bits $z) when $x16_mode
  fail exception(xGP, 0000h) when nat(offset) > nat(CS.limit)
  write sel to CS.sel
  write zxt(64, sel ++ 0h) to CS.base
  write zxt(64, offset) to RIP
```

In protected mode, the far jump instruction fetches a descriptor from the descriptor table using the given selector as an index, asserts that the descriptor is valid and present, and makes a case distinction based on the descriptor type.

```
action jmp_far(sel::Selector, offset::bits $z) when not $x16_mode
  call (desc, desc_valid) = read_desc(sel)
  fail exception(xGP, sel & FFF8h) when not desc_valid
  fail exception(xNP, sel & FFF8h) when not desc.P
  if isCodeSegment(desc) then call jmp_code_segment(sel, desc, zxt(64, offset))
  elif isCallGate(desc) then call jmp_via_call_gate(sel, desc)
  elif isTSS(desc) then call task_switch(sel, desc)
  elif isTaskGate(desc) then call task_switch_gate(sel, desc)
  else fail exception(xGP, sel & FFF8h)
```

The `read_desc` action is defined in section 13.9. It returns a (descriptor, success indicator) pair. The descriptor type-check predicates (`isCodeSegment`, etc) are defined implicitly based on the reserved bits of the corresponding layouts specified in sections 13.7, 13.8. The task switch actions `task_switch` and `task_switch_gate` are defined in section 16.5. In the rest of this section, we discuss the two remaining actions: `jmp_code_segment` and `jmp_via_call_gate`.

The `jmp_code_segment` action performs a direct jump to the given target code segment and the offset. Before changing the `CS` and the `RIP` registers, the action makes sure that

- the target code segment is accessible at the current privilege level,
- the target code segment is a valid code segment,
- the target offset is within the target code segment.

```
action jmp_code_segment(sel::Selector, desc::CodeSegment, rip::bits 64)
  fail exception(xGP, sel & FFF8h) when not can_access_code(sel, desc, CPL)
  call check_code(sys, sel, desc)
  call check_rip(desc, rip)
  write (sel with [RPL = CS.sel.RPL]) to CS.sel
  write desc_attr(desc) to CS.attr
  write zxt(64, desc.base) to CS.base
  write desc.limit to CS.limit
  write rip to RIP
```

The `can_access_code` function is defined in section 13.10. The `check_code` and `check_rip` actions are defined in section 13.12. If any these checks fails, the processor raises an exception. Note that the current privilege level `CPL` does not change and the request privilege level of the new code segment selector `CS.sel.RPL` is inherited from the old code segment selector.

The `jmp_via_call_gate` action reads the target code segment selector and the offset from the given call-gate descriptor, fetches the target code segment descriptor, and then performs a direct jump to the target code segment.

```
action jmp_via_call_gate(sel::Selector, gate::CallGate) when not $x16_mode
  fail exception(xGP, sel & FFF8h) when not can_access_gate(gate)
  call (upper, upper_valid) = read_upper_gate(sel)
  fail exception(xGP, sel & FFF8h) when not upper_valid
  let cs_sel = gate.sel with [RPL = 00b]
  let rip = upper[31:0] ++ gate.offset
  call (cs_desc, cs_desc_valid) = read_desc(cs_sel)
  fail exception(xGP, cs_sel & FFF8h) when not cs_desc_valid
  call jmp_code_segment(cs_sel, cs_desc, rip)
```

The can_access_gate function is defined in section 13.10. In long mode the upper 32 bits of the target offset are stored in the descriptor table entry next to the call-gate descriptor. The action read_upper_gate takes care of checking the mode and fetching the upper bits of the target offset. This action is defined in section 13.9.

16.2 Far Procedure Call

Procedures are implemented using the CALL and the RET instructions. The CALL instruction pushes the current instruction address in the stack and transfers control to the first instruction of the procedure. The last instruction of the procedure must be a RET instruction, which reads the return address from the stack, and transfers the control back to the code that called the procedure.

In 16-bit mode, a far procedure call is similar to a far jump. The only difference is that a procedure call saves the current code segment selector and the current instruction pointer into the stack using the push action defined in chapter G.

```
action call_far(sel::Selector, offset::bits $z) when $x16_mode
  fail exception(xGP, 0000h) when nat(offset) > nat(CS.limit)
  call rsp1 = push($v, zxt($v, CS.sel), RSP[$sa-1:0])
  call rsp2 = push($v, RIP[$v-1:0], rsp1)
  write gpr($sa, rsp2, RSP) to RSP
  write sel to CS.sel
  write zxt(64, sel ++ 0h) to CS.base
  write zxt(64, offset) to RIP
```

In protected mode, the far call instruction fetches a descriptor from the descriptor table using the given selector as an index, asserts that the descriptor is valid and present, and makes a case distinction based on the descriptor type.

```
action call_far(sel::Selector, offset::bits $z) when not $x16_mode
  call (desc, desc_valid) = read_desc(sel)
  fail exception(xGP, sel & FFF8h) when not desc_valid
  fail exception(xNP, sel & FFF8h) when not desc.P
  if isCodeSegment(desc) then call call_code_segment(sel, desc, zxt(64, offset))
  elif isCallGate(desc) then call call_via_call_gate(sel, desc)
  elif isTSS(desc) then call task_switch(sel, desc)
  elif isTaskGate(desc) then call task_switch_gate(sel, desc)
  else fail exception(xGP, sel & FFF8h)
```

Note that the definition of the call_far action is almost the same as the definition of the jmp_far action except for invocations of the call_code_segment and call_via_call_gate actions. Before we define these actions formally, let us describe the high-level algorithm of these actions. Given the target code segment selector, descriptor, and the target offset, the call_code_segment action makes a direct procedure call as follows:

1. Check that the code segment descriptor is accessible at the current privilege level.
2. Check that the code segment descriptor defines a valid code segment and the target offset is within the code segment.
3. Push the current code segment selector CS.sel into the stack.

4. Push the current instruction pointer `RIP` into the stack.

5. Load the target code segment selector and descriptor into the `CS` register.

6. Load the target offset into the `RIP` register.

Notice that the current privilege level `CPL` remains unchanged throughout the direct procedure call. When the procedure call is performed via a call-gate, the `CPL` can change into a more privileged level. The change of the `CPL` triggers a stack switch. A new stack is used in order to protect the higher privileged target code from the lower privileged current code. Given the selector and the descriptor of a call-gate, the `call_via_call_gate` action makes the following steps:

1. Check that the call-gate descriptor is accessible at the current privilege level.

2. Read the target code segment selector and the lower 32 bits of the target offset from the call-gate descriptor.

3. If the operating mode is long mode then fetch the upper 32 bits of the target offset from the descriptor table entry immediately next to the given call-gate descriptor.

4. Fetch the target code segment descriptor from the descriptor table using the target code segment selector.

5. Check that the fetched code segment descriptor is valid and present.

6. Compute the new privilege level from `DPL` field of the code segment descriptor.

7. If the new privilege level is the same as the old privilege level then perform a direct procedure call to the target code segment in the same way as in the `call_code_segment` action.

8. If the new privilege level is more privileged than the old privilege level then

    1. Load the new stack segment selector and the new stack pointer from the current task state segment.
    2. Fetch the new stack segment descriptor from the descriptor table and check that it is valid and present.
    3. Push the old stack segment selector and old stack pointer into the new stack.
    4. Copy parameters from the old stack segment to the new stack segment. The number of parameters to be copied is given in the `params` field of the call-gate descriptor.
    5. Push the current code segment selector `CS.sel` into the new stack.
    6. Push the current instruction pointer `RIP` into the new stack.
    7. Load the target code segment selector and descriptor into the `CS` register.
    8. Load the target offset into the `RIP` register.
    9. Load the new privilege level into the `CPL` register.

Control transfer to an interrupt handler makes the same steps with the exception that it additionally pushes the flags register and the error code into the stack. As we do not want to formally specify the shared steps twice, we extract these steps into actions `call_with_old_stack` and `call_with_new_stack`, which perform direct call to the target code segment in the old stack and the new stack respectively. Before proceeding with the `call_code_segment` action, we need to explain the `call_with_old_stack` action.

The `call_with_old_stack` action makes all the steps from the algorithm that was described for the `call_code_segment` action. Additionally, the `call_with_old_stack` action can also push the flags and the error code into the stack depending on the given call options. Call options is a record the fields of which enable/disable some steps in the `call_with_old_stack` and the `call_with_new_stack` actions.

```
record CallOptions
  field cpl::bits 2
  field ist::bits 3
  field params::bits 5
  field flags::bit
  field ecode::bit
  field ecodeval::bits 16
```

`cpl` is the new privilege level.

`ist` is the index into the interrupt stack table.

`params` is the number of parameters to be copied between the old and the new stack.

`flags` indicates whether to push the flags into the stack or not.

`ecode` indicates whether to push the error code into the stack or not.

`ecodeval` is the value of the error code.

The `call_with_old_stack` action takes five arguments: the target code segment selector and descriptor, the target offset, the width of the push operation, and the call options.

```
action call_with_old_stack(sel::Selector, desc::CodeSegment,
                           rip::bits 64, $n::{16, 32, 64}, options::CallOptions)
  fail exception(xGP, sel & FFF8h) when not can_access_code(sel, desc, CPL)
  call check_code(sys, sel, desc)
  call check_rip(desc, rip)
  call rsp1 = push_if(options.flags, $n, RFLAGS[$n-1:0], RSP[$sa-1:0])
  call rsp2 = push($n, zxt($n, CS.sel), rsp1)
  call rsp3 = push($n, RIP[$n-1:0], rsp2)
  call rsp4 = push_if(options.ecode, $n, zxt($n, options.ecodeval), rsp3)
  write gpr($sa, rsp4, RSP) to RSP
  write (sel with [RPL = CS.sel.RPL]) to CS.sel
  write desc_attr(desc) to CS.attr
  write zxt(64, desc.base) to CS.base
  write desc.limit to CS.limit
  write rip to RIP
```

The `can_access_code` function is defined in section 13.10. The `check_code` and `check_rip` actions are defined in section 13.12. The `push` and `push_if` actions are defined in chapter G. Note that the current privilege level `CPL` does not change and the request privilege level of the new code segment selector `CS.sel.RPL` is inherited from the old code segment selector.

The `call_code_segment` action simply invokes the `call_with_old_stack` action with the correct call options.

```
action call_code_segment(sel::Selector, desc::CodeSegment, offset::bits 64)
  let options = CallOptions with [flags=0b, ecode=0b]
  call call_with_old_stack(sel, desc, offset, $v, options)
```

Assuming the existence of the `call_with_new_stack` action, we can define the `call_via_call_gate` action following the algorithm described above:

```
action call_via_call_gate(sel::Selector, gate::CallGate)
  fail exception(xGP, sel & FFF8h) when not can_access_gate(gate)
  call (upper, upper_valid) = read_upper_gate(sel)
  fail exception(xGP, sel & FFF8h) when not upper_valid
  let cs_sel = gate.sel with [RPL = 00b]
  let rip = upper[31:0] ++ gate.offset
  call (cs_desc, cs_desc_valid) = read_desc(cs_sel)
  fail exception(xGP, cs_sel & FFF8h) when not cs_desc_valid
  let new_cpl = code_cpl(cs_desc)
  let $gs = gate_size(gate)::{16, 32, 64}
  if new_cpl == CPL then
    let options = CallOptions with [flags=0b, ecode=0b]
    call call_with_old_stack(cs_sel, cs_desc, rip, $gs, options)
  else
    let options = CallOptions with [cpl = new_cpl, ist = 000b,
                                    params = gate.params,
                                    flags=0b, ecode=0b]
    call call_with_new_stack(cs_sel, cs_desc, rip, $gs, options)
```

The `can_access_gate` function is defined in section 13.10. The action `read_upper_gate` fetches the upper bits of the target offset. This action is defined in section 13.9. The function `code_cpl` computes the new privilege level based on the target code segment descriptor, and it is defined in section 13.10.

In the rest of this section we define the `call_with_new_stack` action. This action is the most difficult action in the whole instruction set because it operates on two stacks: the old stack and the new stack. Before we dive into details, let us discuss the high-level steps of the action. The action takes five arguments: the target code segment selector and descriptor, the target offset, the width of the push operation, and the call options. The algorithm for the action looks like this:

1. Check that
   - the target code segment descriptor is accessible at the new privilege level.
   - the target code segment descriptor defines a valid code segment.
   - the target offset is within the target code segment limits.
2. Load the new stack segment selector and pointer from the task state segment.
3. Fetch the new stack segment descriptor from the descriptor table and checks that it is valid and present.
4. Load all the parameters from the old stack and saves them in a temporary array.
5. Save the old stack segment selector and the old stack pointer in temporary registers.

6. Save the old code segment selector and the old instruction pointer in temporary registers.
7. Load the new stack segment selector and descriptor into the SS register.
8. Load the new stack pointer into the RSP register.
9. Load the target code segment selector and descriptor into the CS register.
10. Load the target offset into the RIP register.
11. Load the new privilege level into the CPL register.
12. Use the **chain** statement in order to apply all register writes. As a result of this statement, the SS and the RSP registers point to the new stack in the subsequent steps of the algorithm.
13. Push the old stack selector from the temporary register into the new stack.
14. Push the old stack pointer from the temporary register into the new stack.
15. Push the parameters from the temporary array into the new stack.
16. Push the flags register into the new stack if the call options require it.
17. Push the old code selector from the temporary register into the new stack.
18. Push the old instruction pointer from the temporary register into the new stack.
19. Push the error code into the new stack if the call options require it.

We need temporary registers for steps 5, 6, 7. For storing the parameters we use an array of 32 elements because the number of parameters cannot exceed 32:

```
register _params::array bits 5 of bits 64
```

As the number of parameters can be less than 32, we also need to store the counter:

```
register _params_cnt::bits 5
```

For storing the selector, pointers, and options we use the following register:

```
register _call_context::CallContext
record CallContext
  field ss_sel::bits 16
  field rsp::bits 64
  field cs_sel::bits 16
  field rip::bits 64
  field gate_size::bits 3
  field options::CallOptions
```

Now we can give the definition of the call_with_new_stack action.

```
action call_with_new_stack(sel::Selector, desc::CodeSegment,
                           rip::bits 64, $n::{16, 32, 64}, options::CallOptions)
  fail exception(xGP, sel & FFF8h) when not can_access_code(sel, desc, options.cpl)
  call check_code(sys, sel, desc)
  call check_rip(desc, rip)
```

```
  call (ss_sel, ss_desc, rsp) = inner_stack(options.ist, options.cpl)
  call load_params($n, options.params)
  call save_context($n, options)
  write ss_sel to SS.sel
  write desc_attr(ss_desc) to SS.attr
  write zxt(64, ss_desc.base) to SS.base
  write rsp to RSP
  write (sel with [RPL = options.cpl]) to CS.sel
  write desc_attr(desc) to CS.attr
  write zxt(64, desc.base) to CS.base
  write desc.limit to CS.limit
  write rip to RIP
  write options.cpl to CPL
  chain push_in_new_stack
```

The `inner_stack` action is defined in chapter G. Given the interrupt stack table index `ist` and the new privilege level `cpl`, the action loads the new stack selector, descriptor, and pointer.

The `load_params` action will be defined later in this section. Given the width of a parameter and the number of the parameters, the action loads the parameters from the old stack into the `_params` array and saves the number of the parameters in the `_params_cnt` register.

The `save_context` action stores the options, the gate size, and the old values of the stack and code registers into the `_call_context` register.

```
action save_context($n::{16, 32, 64}, options::CallOptions)
  write options to _call_context.options
  write bits(3, $n/16) to _call_context.gate_size
  write SS.sel to _call_context.ss_sel
  write RSP to _call_context.rsp
  write CS.sel to _call_context.cs_sel
  write RIP to _call_context.rip
```

The `push_in_new_stack` action performs steps 13–20 from the algorithm. Since this action is invoked using the **`chain`** statement, all register writes are already applied when the action starts executing. Thus, the `SS` and the `RIP` registers define the new stack, and the `_call_context` register fields contain values written by the `save_context` action.

```
action push_in_new_stack
  let ctx = _call_context
  let opt = ctx.options
  let $n = (nat(ctx.gate_size) * 16)::{16, 32, 64}
  call rsp1 = push($n, zxt($n, ctx.ss_sel), RSP[$sa-1:0])
  call rsp2 = push($n, ctx.rsp[$n-1:0], rsp1)
  call rsp3 = push_params($n, rsp2)
  call rsp4 = push_if(opt.flags, $n, RFLAGS[$n-1:0], rsp3)
  call rsp5 = push($n, zxt($n, ctx.cs_sel), rsp4)
  call rsp6 = push($n, ctx.rip[$n-1:0], rsp5)
  call rsp7 = push_if(opt.ecode, $n, zxt($n, opt.ecodeval), rsp6)
  write gpr($sa, rsp7, RSP) to RSP
```

In order to complete the description of the far procedure call mechanism, we need to define how the parameters are copied from the old stack into the new stack. This

happens in two steps:

1. Load the parameters from the old stack into the `_params` array using the `load_params` action.
2. Push the parameters from the `_params` array into the new stack using the `push_params` action.

The `load_params` action is an unrolled loop, where each iteration of the loop loads a single parameter from the stack. Let the `$n` denote the width of each parameter and the `cnt` denote the number of parameters, then the `load_params` action has the following pseudocode:

```
_params_cnt = cnt
for idx from 11110b downto 00000b do
  if idx < param_cnt then
    _params[idx] = pop $n bytes from the stack
```

By putting the body of the loop into the `load_param` action and unrolling the loop, we get the definition of the `load_params` action (we do not show iterations 28–1 for brevity):

```
action load_params($n::{16, 32, 64}, cnt::bits 5)
  let rsp30 = RSP[$sa-1:0]
  let idx30 = 11110b
  write cnt to _params_cnt
  call (idx29, rsp29) = load_param($n, cnt, idx30, rsp30)
  call (idx28, rsp28) = load_param($n, cnt, idx29, rsp29)
  call (idx0, rsp0) = load_param($n, cnt, idx1, rsp1)
  call (idx_, rsp_) = load_param($n, cnt, idx0, rsp0)
```

The `load_param` action takes four arguments: the width of each parameter, the number of the parameters, the index of the current parameter, the stack pointer. If the parameter index is smaller then the number of the parameters then the action loads the parameter from the stack. The action returns a decremented index and the new stack pointer.

```
action load_param($n::{16, 32, 64}, cnt::bits 5,
                  idx::bits 5, rsp::bits $sa)::(bits 5, bits $sa)
  if idx >= cnt then return (idx - 00001b, rsp)
  else call (val, new_rsp) = pop($n, rsp)
       write zxt(64, val) to _params[idx]
       return (idx - 00001b, new_rsp)
```

The `push_params` action does exact opposite of the `load_params` action as shown in the following pseudocode:

```
for idx from 00000b to 11110b do
  if idx < param_cnt then
    push _params[idx] into the stack
```

By unrolling the loop and putting the loop body into the `push_param` action, we get the definition of the `push_params` action (we do not show iterations 2–30 for brevity):

```
action push_params($n::{16, 32, 64}, rsp::bits $sa)::bits $sa
  let rsp0 = rsp
```

```
  let idx0 = 00000b
  let cnt = _params_cnt
  call (idx1, rsp1) = push_param($n, cnt, idx0, rsp0)
  call (idx2, rsp2) = push_param($n, cnt, idx1, rsp1)
  call (idx31, rsp31) = push_param($n, cnt, idx30, rsp30)
  return rsp31
```

Similarly to the `load_param` action, the `push_param` action checks if the parameter index is smaller that the number of the parameters, and then pushes the parameter into the stack.

```
action push_param($n::{16, 32, 64}, cnt::bits 5,
                  idx::bits 5, rsp::bits $sa)::(bits 5, bits $sa)
  if idx >= cnt then return (idx + 00001b, rsp)
  else call new_rsp = push($n, _params[idx][$n-1:0], rsp)
       return (idx + 00001b, new_rsp)
```

16.3 Control Transfer to an Interrupt Handler

In this section we describe the control transfer mechanism to an interrupt handler. The `jisr` action is defined as follows:

```
action jisr
  if jisr_event.ecode == epsilon then
    call icall(jisr_event.vector, 0b, 0000h)
  else
    call icall(jisr_event.vector, 1b, jisr_event.ecode)
```

where the `icall` action calls the interrupt handler corresponding to the given vector. The second argument of the `icall` indicates whether the interrupt has an associated error code. The third argument contains the actual value of the error code.

In real mode an interrupt descriptor table entry is a pair of a code segment selector and an offset within the code segment. The `icall` action fetches the entry corresponding to the given vector and computes the address of the handler using simplified the segmentation mechanism of real mode. Before performing the control transfer, the action pushes the flags, the old code segment selector, and the old instruction pointer into the stack. Besides that, the action disables maskable interrupt and clears debug flags.

```
action icall(vector::bits 8, ecode::bit, ecodeval::bits 16) when $x16_mode
  call (sel, ip) = read_idt(vector)
  fail exception(xGP, 0000h) when nat(ip) > nat(CS.limit)
  call rsp1 = push(16, RFLAGS[15:0], RSP[$sa-1:0])
  call rsp2 = push(16, CS.sel, rsp1)
  call rsp3 = push(16, RIP[15:0], rsp2)
  let adjucted_flags = RFLAGS with [TF = 0b, IF = 0b, AC = 0b, RF = 0b]
  write adjucted_flags to RFLAGS
  write gpr($sa, rsp3, RSP) to RSP
  write sel to CS.sel
  write zxt(64, sel ++ 0h) to CS.base
  write zxt(64, ip) to RIP
```

The `read_idt` action is defined in section 13.9. The `push` action is defined in chapter G.

In protected mode, an interrupt table entry is either a trap gate, or an interrupt gate, or a task gate. In this mode, the `read_idt` action returns the gate descriptor and the upper 32 bits of the handler's offset (the lower 32 bits are stored inside the gate descriptor). Depending on the gate type, the `icall` action invokes either the `icall_via_gate` action or the `task_switch_gate` action. The latter action is defined in chapter 16.5.

```
action icall(vector::bits 8, ecode::bit, ecodeval::bits 16) when not $x16_mode
  call (desc, upper) = read_idt(vector)
  if isIntrGate(desc) or isTrapGate(desc) then
    call icall_via_gate(desc, upper, vector, ecode, ecodeval)
  elif isTaskGate(desc) then
    call task_switch_gate(zxt(14, vector) ++ 10b, desc)
  else fail exception(xGP, zxt(14, vector) ++ 10b)
```

The `icall_via_gate` action is very similar to the `call_via_call_gate` action defined in the previous section. The action takes the gate descriptor, the upper 32 bits of the target offset, the vector, the error code indicator, and the error code value. The action makes the following steps:

1. Check that the given gate is accessible at the current privilege level.
2. From the gate descriptor read the target code segment selector and the lower 32 bits of the target offset.
3. Compute the full target offset by combining the upper bits with the lower bits.
4. Fetch the target code segment descriptor from the descriptor table using the target selector.
5. Check that the fetched descriptor is valid and present.
6. Clear debug flags and the interrupt flag (if the gate is an interrupt gate).
7. Compute the new privilege level based on the target code segment descriptor.
8. If the new privilege level is the same as the current privilege level then make a direct call to the handler with the old stack.
9. Otherwise, make a direct call to the handler with the new stack.

The described algorithm can be translated into a formal definition as follows:

```
action icall_via_gate(gate::IntrOrTrapGate, upper::bits 32,
                      vector::bits 8, ecode::bit, ecodeval::bits 16)
  fail exception(xGP, sel) when not can_access_gate(gate)
  let rip = upper[31:0] ++ gate.offset
  let cs_sel = gate.sel with [RPL = 00b]
  call (cs_desc, cs_valid) = read_desc(cs_sel)
  fail exception(xGP, cs_sel & FFF8h) when not cs_valid
  fail exception(xGP, cs_sel & FFF8h) when not cs_desc.P
  write 0b to RFLAGS.TF
  write 0b to RFLAGS.NT
  write 0b to RFLAGS.RF
  write 0b to RFLAGS.VM
  write 0b to RFLAGS.IF when isIntrGate(gate)
```

```
let $gs = gate_size(gate)::{16, 32, 64}
let new_cpl = code_cpl(cs_desc)
if $legacy_mode and new_cpl == CPL then
  let options = CallOptions with [flags=1b, ecode=ecode, ecodeval=ecodeval]
  call call_with_old_stack(cs_sel, cs_desc, rip, $gs, options)
else
  let options = CallOptions with [cpl = new_cpl, ist = gate.ist,
                                  params = 00000b, flags=1b,
                                  ecode = ecode, ecodeval = ecodeval]
  call call_with_new_stack(cs_sel, cs_desc, rip, $gs, options)
```

The `can_access_gate` function is defined in section 13.10. The action `read_upper_gate` fetches the upper bits of the target offset. This action is defined in section 13.9. The function `code_cpl` computes the new privilege level based on the target code segment descriptor, and it is defined in section 13.10. The `call_with_old_stack` and the `call_with_new_stack` stack actions are defined in the previous section.

16.4 Far Return

Far return control transfer occurs when a procedure exits using the `RETF` instruction or when an interrupt handler exits using the `IRET` instruction. These instructions read the return address from the stack and load it into the `RIP` and `CS` registers. The `IRET` instruction also loads the flags from the stack. In case the stack was switched during control transfer to the procedure or the interrupt handler, the return instructions read the old stack pointers from the current stack and switch the stacks back.

We extract the common steps from the `RETF` and `IRET` instructions into the `ret_far` action, which takes two arguments: `skip` and `intr`. The first argument specifies how many bytes must be popped from the stack in order to remove the parameters of the procedure. The second argument indicates whether the action is called from the `IRET` or from the `RETF`.

In 16-bit mode, the `ret_far` action makes the following steps:

1. Pop the return address (code segment selector, instruction pointer) from the stack.
2. In case of a return from an interrupt handler, pop the flags from the stack.
3. Pop and discard the parameters from the stack, i.e. pop the `skip` number of bytes from the stack.
4. Adjust and write the popped flags into the `RFLAGS` register.
5. Compute the base address of the return code segment as `selector * 16`.
6. Write the return code segment selector and base address to the `CS` register.
7. Write the return instruction pointer to the `RIP` register.
8. Write the new stack pointer to the `RSP` register.

By formalizing these steps we get the following definition:

```
action ret_far(skip::bits 16, intr::bit) when $x16_mode
  call (rip, cs_sel, flags, rsp) = pop_rip_cs_flags(skip, intr, RSP[$sa-1:0])
  write adjust_flags_ret(RFLAGS[63:$v]++flags) to RFLAGS when intr
  fail exception(xGP, 0000h) when nat(rip) > nat(CS.limit)
  call write_rip_cs(zxt(64, rip), cs_sel, CS.attr,
                    zxt(64, cs_sel ++ 0h), CS.limit)
  write gpr($sa, rsp, RSP) to RSP
```

The action `pop_rip_cs_flags` pops the instruction pointer, pops the code segment selector, conditionally pops the flags, and skips over the parameters in the stack.

```
action pop_rip_cs_flags(skip::bits 16, intr::bit, rsp::bits $sa)
                       ::(bits $v, Selector, bits $v, bits $sa)
  call (rip, rsp1) = pop($v, RSP[$sa-1:0])
  call (x, rsp2) = pop($v, rsp1)
  let cs_sel = x[15:0]::Selector
  call (flags, rsp3) = pop_if(intr, $v, rsp2)
  let rsp4 = rsp3 + zxt($sa, skip)
  return (rip, cs_sel, flags, rsp4)
```

Before writing the popped flags into the `RFLAGS` register, we adjust them:

- Copy all read-only bits from the `RFLAGS`,
- Clear the debug resume flag `RF`,
- Copy the virtual 8086 indicator flags `VM` from the `RFLAGS`,
- Copy the virtual interrupt flags `VIP` and `VIF` from the `RFLAGS` if the current privilege level indicates user software or the current operating mode is real mode.
- Copy the IO privilege level from the `RFLAGS` if the current privilege level indicates user software.
- Copy the interrupt flag `IF` from the `RFLAGS` if the IO privilege level is higher (numerically less) than the current privilege level.

The function `adjust_flags_ret` carries out all the discussed adjustments:

```
function adjust_flags_ret(f::Flags)::Flags
  = let f1 = fixFlags(f, RFLAGS) in
    let f2 = f1 with [RF = 0b, VM = RFLAGS.VM] in
    adjust_IF_IOPL(adjust_VIF_VIP(f2))
function adjust_VIF_VIP(f::Flags)::Flags
  = if 00b == CPL and not $real_mode then f
    else f with [VIP = RFLAGS.VIP, VIF = RFLAGS.VIF]
function adjust_IF_IOPL(f::Flags)::Flags
  = if RFLAGS.IOPL < CPL then f with [IF = RFLAGS.IF, IOPL = RFLAGS.IOPL]
    elif 00b < CPL then f with [IOPL = RFLAGS.IOPL]
    else f
```

Given the new flags and the old flags, the function `fixFlags` returns combined flags such that all read-only flags are taken from the old flags and all remaining flags are taken from the new flags. This function is defined implicitly based on the read-only attributes of the `Flags` layout defined in section 12.2.

The action `write_rip_cs` updates the `RIP` and `CS` register with the given values:

```
action write_rip_cs(rip::bits 64, sel::Selector, attr::CodeAttr,
                    base::bits 64, limit::bits 32)
    write rip to RIP
    write sel to CS.sel
    write attr to CS.attr
    write base to CS.base
    write limit to CS.limit
```

In 32- and 64-bit modes, the `ret_far` action is more complicated then in 16-bit mode. The first difference is that we need to fetch the return code segment descriptor from the descriptor table using the return code segment selector in the stack. We must ensure that the descriptor is valid, present, and defines a code segment. Besides that, we check that the return instruction pointer is within the return code segment and that the return code is not more privileged than the current code (the privilege level of the return code is specified by the `RPL` field of the return selector).

If the control transfer to the current procedure or the interrupt handler involved a stack switch, the return action has to switch the stacks back. This is done by popping the old stack selector and pointer from the current stack, and then writing them in to the `SS` and `RSP` registers.

Summarizing all these differences, we get a definition of the `ret_far` action:

```
action ret_far(skip::bits 16, intr::bit) when not $x16_mode
  call (rip, cs_sel, flags, rsp) = pop_rip_cs_flags(skip, intr, RSP[$sa-1:0])
  write adjust_flags_ret(RFLAGS[63:$v]++flags) to RFLAGS when intr
  call (temp_desc, cs_valid) = read_desc(cs_sel)
  let cs_desc = temp_desc::CodeSegment
  fail exception(xGP, cs_sel & FFF8h) when not cs_valid
  fail exception(xNP, cs_sel & FFF8h) when not cs_desc.P
  fail exception(xGP, cs_sel & FFF8h) when not can_ret(cs_sel)
  let new_cpl = cs_sel.RPL
  call check_code(sys, sel, cs_desc)
  call check_rip(cs_desc, zxt(64, rip))
  call write_rip_cs(zxt(64, rip), cs_sel, desc_attr(cs_desc),
                    cs_desc.base, cs_desc.limit)
  if CPL == new_cpl and not ($long_mode and intr) then
    write gpr($sa, rsp, RSP) to RSP
  else call (new_rsp, rsp1) = pop($v, rsp)
       call (x, rsp2) = pop($v, rsp1)
       let ss_sel = x[15:0]::Selector
       call ss_desc = ret_stack(ss_sel, new_cpl, $long_mode and cs_desc.L)
       write ss_sel to SS.sel
       write desc_attr(ss_desc) to SS.attr
       write zxt(64, ss_desc.base) to SS.base
       write ss_desc.limit to SS.limit
       write gpr($v, new_rsp + zxt($v, skip), RSP) to RSP
```

The `can_ret` function is defined in section 13.10. The `check_code` and `check_rip` actions are defined in section 13.12. The action `ret_stack` fetches the stack segment descriptor using the stack segment selector, the new privilege level, and the new operating mode indicator. The architecture allows null stack segments in system 64-bit mode. Therefore, the `ret_stack` action checks whether the stack selector is null or not. If yes, it returns a null stack descriptor as defined in section [Segment Descriptors] Otherwise,

the action fetches the stack descriptor from the descriptor table and checks that the descriptor is valid using the `check_stack` action defined in section 13.13.

```
action ret_stack(sel::Selector, new_cpl::bits 2, new_x64_mode::bit)::DataSegment
  if sel & FFF8h == 0000h then
    fail exception(xGP, 0000h) when not new_x64_mode or new_cpl == 11b
    fail exception(xGP, 0000h) when sel.RPL <> new_cpl
    return null_stack_desc
  else call (desc, desc_valid) = read_desc(sel)
    fail exception(xGP, sel & FFF8h) when not desc_valid
    call check_stack(sys, sel, desc, new_cpl, new_x64_mode)
    return desc
```

16.5 Task Switch

Hardware task switch is a legacy operation, all modern operation systems implement software task switch. In long mode hardware task switch is not supported at all. We leave out formalization of hardware task switch as future work.

Assuming that the action `task_switch` performs hardware task switch, we can define the action `task_switch_gate` which performs task switch via a given task gate.

```
action task_switch(sel::Selector, desc::TSS)
  fail todo
action task_switch_gate(sel::Selector, gate::TaskGate)
  fail exception(xGP, sel & FFF8h) when not can_access_gate(gate)
  let tss_sel = gate.sel
  call (temp_desc, tss_valid) = read_desc(tss_sel)
  let tss_desc = temp_desc::TSS
  fail exception(xGP, tss_sel & FFF8h) when not tss_valid
  fail exception(xNP, tss_sel & FFF8h) when not tss_desc.P
  fail exception(xGP, tss_sel & FFF8h) when not isTSS(tss_desc)
  fail exception(xGP, tss_sel & FFF8h) when tss_desc.busy
  call task_switch(tss_sel, tss_desc)
```

The action makes sure that the current code can access the given task gate descriptor, and then fetches the task segment descriptor from the global descriptor table using the selector field of the task gate. The task segment descriptor must be valid, present, and available (not busy). If all these conditions hold, the action dispatches task switch to the task segment.

CHAPTER

SEVENTEEN

VIRTUALIZATION

Besides operating modes, a processor with virtualization support has virtualization modes, which are the host mode and the guest mode. We refer to software running in host mode as a host, and to software running in guest mode as a guest. The goal of virtualization is to execute the guest in a virtual machine with the ISA of the real machine. The guest has full control of the virtual machine, in particular it can operate at the system privilege level (`CPL=00b`). However, the guest does not control the real machine. The host with the hardware assistance monitors the guest execution and stops it as soon as the guest makes a sensitive action, which is an action that would break virtualization. After that, the host emulates the sensitive action for the guest and continues the guest execution. In this chapter we describe the virtualization mechanism provided by hardware.

The current virtualization mode is indicated by the `_guest` register. When the register is set to one, then the processor is running in guest mode. Otherwise, the processor is running in host mode. Any combination of an operating mode and a virtualization mode is valid.

In order to switch to guest mode, the processor executes the `VMRUN` instruction. This instruction has one operand: the physical address of the virtual machine control block (VMCB), which stores the state of the guest registers, injected interrupts, and guest exit conditions (also known as intercept conditions). The `VMRUN` instruction makes the following steps:

1. Save the host registers into the special memory region, the physical address of which is specified by the `VM_HSAVE_PA` register.
2. Load the guest registers from the state save area of the VMCB.
3. Load injected interrupts and exit conditions from the control area of the VMCB into the `_vmcb_ca` register.
4. Run the instruction processing cycle until any of the exit conditions triggers.
5. Save the guest registers to the VMCB SSA.

6. Write the exit code, exit information into the control area of the VMCB.

7. Load the host registers from the VM_HSAVE_PA memory region.

Since the host instruction pointer RIP is saved in step 1 and restored in step 7, the processor starts executing the instruction immediately next to the VMRUN instruction when it returns from guest mode to host mode. Therefore, the host code that executes a guest in a virtual machine is usually implemented as a loop around the VMRUN instruction:

```
initialize the VMCB at address x
repeat:
  save extended host state
  load extended guest state
  VMRUN x
  save extended guest state
  load extended host state
  check the exit code and fix the problem
  enable and process interrupts
  inject interrupts/exceptions to guest if necessary
  goto repeat
```

In this code the extended guest/host state denotes all important registers that are not saved and restored automatically by the VMRUN. The extended guest state can be saved and restored using the VMSAVE and the VMLOAD instructions. There is no hardware assistance for the extended host state, therefore it has to be saved and restored by software explicitly.

Notice that the host controls guest execution by configuring the fields of the VMCB, which is a 4K region in the physical memory space. The VMCB consists of two parts: the control area (CA) and the state save area (SSA).

```
record VMCB
  field CA::VMCB_CA
  field SSA::VMCB_SSA
```

The CA starts at beginning of the VMCB, and the SSA starts at the offset 400h. Since the size of the CA is much smaller than 400h, there is a region of reserved space between the end of the CA and the start of the SSA. The purpose of the SSA is to store the values of the guest registers. The CA stores guest exit conditions (intercepts), guest exit code, injected events, guest TLB tag, the physical base address of the root nested page table. In the following two sections we give detailed descriptions of the SSA and the CA.

17.1 Guest State Save Area — VMCB SSA

The guest state save area stores a large subset of the processor registers. These registers are saved and restored in the VMRUN, the VMLOAD and the VMSAVE instructions. The registers that are not included in the SSA have to be saved and restored by the host explicitly.

The VMRUN instruction saves and restores the following registers:

- the ES, CS, DS, SS segment registers including all the hidden state (the attributes, the limit, the base address),

- the GDTR and the IDTR table descriptor registers,
- the CR0, CR3, CR4 control registers,
- the EFER register,
- the RFLAGS register,
- the DR6, DR7 debug registers,
- the CPL current privilege level register.
- the RSP stack pointer register,
- the RAX general-purpose register,
- the RIP instruction pointer register,
- the G_PAT guest page attribute table register if nested paging is enabled.

The remaining registers in the SSA are saved and restored by the VMSAVE/VMLOAD pair of instructions. These register are:

- the FS, GS segment registers, including all the hidden state (the attributes, the limit, and the base address),
- the LDTR local descriptor table register, including all the hidden state,
- the TR task segment register, including all the hidden state,
- the LSTAR, CSTAR, SFMASK fast control transfer registers,
- the KernelGSBase register,
- the SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP legacy fast control transfer registers,
- the CR2 page-fault address register,
- the branch optimization and debug registers that we do not include in our model: DBGCTL, BR_FROM, BR_TO, LASTEXCPFROM, LASTEXCPTO.

We conclude this section by listing the definition of the VMCB_SSA. Because of many reserved fields the definition is quite large.

```
layout VMCB_SSA
  field ES_sel::bits 16
  field ES_attr::bits 16
  field ES_limit::bits 32
  field ES_base::bits 64
  field CS_sel::bits 16
  field CS_attr::bits 16
  field CS_limit::bits 32
  field CS_base::bits 64
  field SS_sel::bits 16
  field SS_attr::bits 16
  field SS_limit::bits 32
  field SS_base::bits 64
```

```
field DS_sel::bits 16
field DS_attr::bits 16
field DS_limit::bits 32
field DS_base::bits 64
field FS_sel::bits 16
field FS_attr::bits 16
field FS_limit::bits 32
field FS_base::bits 64
field GS_sel::bits 16
field GS_attr::bits 16
field GS_limit::bits 32
field GS_base::bits 64
field GDTR_rsv1::bits 32
field GDTR_limit::bits 16
field GDTR_rsv2::bits 16 ignored
field GDTR_base::bits 64
field LDTR_sel::bits 16
field LDTR_attr::bits 16
field LDTR_limit::bits 32
field LDTR_base::bits 64
field IDTR_rsv1::bits 32 ignored
field IDTR_limit::bits 16
field IDTR_rsv2::bits 16 ignored
field IDTR_base::bits 64
field TR_sel::bits 16
field TR_attr::bits 16
field TR_limit::bits 32
field TR_base::bits 64
field rsv1::bits 64 ignored
field rsv2::bits 64 ignored
field rsv3::bits 64 ignored
field rsv4::bits 64 ignored
field rsv5::bits 64 ignored
field rsv6::bits 24 ignored
field CPL::bits 8
field rsv7::bits 32 ignored
field EFER::bits 64
field rsv8::bits 64 ignored
field rsv9::bits 64 ignored
field rsv10::bits 64 ignored
field rsv11::bits 64 ignored
field rsv12::bits 64 ignored
field rsv13::bits 64 ignored
field rsv14::bits 64 ignored
field rsv15::bits 64 ignored
field rsv16::bits 64 ignored
field rsv17::bits 64 ignored
field rsv18::bits 64 ignored
field rsv19::bits 64 ignored
field rsv20::bits 64 ignored
field rsv21::bits 64 ignored
field CR4::bits 64
field CR3::bits 64
```

```
field CR0::bits 64
field DR7::bits 64
field DR6::bits 64
field RFLAGS::bits 64
field RIP::bits 64
field rsv22::bits 64 ignored
field rsv23::bits 64 ignored
field rsv24::bits 64 ignored
field rsv25::bits 64 ignored
field rsv26::bits 64 ignored
field rsv27::bits 64 ignored
field rsv28::bits 64 ignored
field rsv29::bits 64 ignored
field rsv30::bits 64 ignored
field rsv31::bits 64 ignored
field rsv32::bits 64 ignored
field RSP::bits 64
field rsv33::bits 64 ignored
field rsv34::bits 64 ignored
field rsv35::bits 64 ignored
field RAX::bits 64
field STAR::bits 64
field LSTAR::bits 64
field CSTAR::bits 64
field SFMASK::bits 64
field KernelGSBase::bits 64
field SYSENTER_CS::bits 64
field SYSENTER_ESP::bits 64
field SYSENTER_EIP::bits 64
field CR2::bits 64
field rsv36::bits 64 ignored
field rsv37::bits 64 ignored
field rsv38::bits 64 ignored
field rsv39::bits 64 ignored
field G_PAT::bits 64
field DBGCTL::bits 64
field BR_FROM::bits 64
field BR_TO::bits 64
field LASTEXCPFROM::bits 64
field LASTEXCPTO::bits 64
```

17.2 Guest Control Area — VMCB CA

By writing to the guest control area, the host can

- enable or disable guest exits (intercepts) on various predefined conditions,
- specify the guest TLB tag, also known as the guest address space identifier (ASID),
- request a full TLB flush during the switch to the guest,

- specify the physical address of the nested root page table in case the nested paging is enabled,
- inject virtual interrupts and exceptions into the guest.

After a guest exit, the host can examine the guest control area and learn

- the guest exit code, which specifies the condition that triggered the intercept,
- the guest exit information, which helps to handle the intercept.

We will give the formal definition of the guest control area at the end of this section. Before that we need to discuss each of the above mentioned aspects. We start with the intercept conditions.

There is a predefined set of sensitive actions which could potentially break guest virtualization. The host can enable intercept of each of those actions by setting the associated bit in the guest control area. The set of sensitive actions includes:

- access to the control registers, the flags register, the debug registers, the descriptor table registers, the task register, and the model-specific registers;
- access to the input/output ports;
- execution of the system instructions;
- exceptions and interrupts;
- the INIT signal and processor shutdown.

The set of the input/output ports and the set of the model-specific registers are quite large. Therefore, the intercept enable bits for the ports and the model-specific registers are located in bitmaps outside of the control area, which only stores the physical base addresses of the bitmaps. For all other actions, the intercept enable bits are stored directly in the control area. These bits are grouped into the following fields according to the intercept type:

```
field READ_CR::bits 16
field WRITE_CR::bits 16
field READ_DR::bits 16
field WRITE_DR::bits 16
field EXCP::bits 32
field intercept::Intercepts
```

The `i`-th bit of the `READ_CR` field enables interception of reads from the `i`-th control register `CR[i]`. When the bit is set, a read from the control register triggers guest exit. Otherwise, the read access succeeds inside the guest. The bits of the `WRITE_CR`, `READ_DR`, `WRITE_DR` have similar meaning, which can be deduced from the field names.

The `EXCP` field enables intercepts of the exceptions. Each exception has an associated vector, which can have a value in range `[0..31]`. When the `i`-th bit of the `EXCP` field is set, then a raised exception with the vector `i` triggers guest exit. When the bit is cleared, the exception is handled within the guest.

The `intercept` field is 64 bits wide and controls the all the remaining intercepts. We first give the definition and then discuss each bit of this field.

```
layout Intercepts
  field INTR::bit
  field NMI::bit
  field SMI::bit
  field INIT::bit
  field VINTR::bit
  field WRITE_CR0::bit
  field READ_IDTR::bit
  field READ_GDTR::bit
  field READ_LDTR::bit
  field READ_TR::bit
  field WRITE_IDTR::bit
  field WRITE_GDTR::bit
  field WRITE_LDTR::bit
  field WRITE_TR::bit
  field RDTSC::bit
  field RDPMC::bit
  field PUSHF::bit
  field POPF::bit
  field CPUID::bit
  field RSM::bit
  field IRET::bit
  field INT::bit
  field INVD::bit
  field PAUSE::bit
  field HLT::bit
  field INVLPG::bit
  field INVLPGA::bit
  field IO_PROT::bit
  field MSR_PROT::bit
  field TASK_SWITCH::bit
  field FERR_FREEZE::bit
  field SHUTDOWN::bit
  field VMRUN::bit
  field VMMCALL::bit
  field VMLOAD::bit
  field VMSAVE::bit
  field STGI::bit
  field CLGI::bit
  field SKINIT::bit
  field RDTSCP::bit
  field ICEBP::bit
  field WBINVD::bit
  field MONITOR::bit
  field MWAIT::bit
  field ARMED_MWAIT::bit
  field rsv1::bits 19 ignored
```

INTR enables intercepts of the physical maskable interrupt.

NMI enables intercepts of the non-maskable interrupt.

SMI enables intercepts of the system-management interrupt.

INIT enables intercepts of INIT signal.

VINTR enables intercepts of the virtual maskable interrupts.

WRITE_CR0 enables intercepts of the CR0 writes that change bits other than CR0.TS or CR0.MP.

READ_IDTR enables intercepts of reads from the IDTR.

READ_GDTR enables intercepts of reads from the GDTR.

READ_LDTR enables intercepts of reads from the LDTR.

READ_TR enables intercepts of reads from the TR.

WRITE_IDTR enables intercepts of writes to the IDTR.

WRITE_GDTR enables intercepts of writes to the GDTR.

WRITE_LDTR enables intercepts of writes to the LDTR.

WRITE_TR enables intercepts of writes to the TR.

RDTSC enables intercepts of the RDTSC instruction.

RDPMC enables intercepts of the RDPMC instruction.

PUSHF enables intercepts of the PUSHF instruction.

POPF enables intercepts of the POPF instruction.

CPUID enables intercepts of the CPUID instruction.

RSM enables intercepts of the RSM instruction.

IRET enables intercepts of the IRET instruction.

INT enables intercepts of the INT instruction.

INVD enables intercepts of the INVD instruction.

PAUSE enables intercepts of the PAUSEinstruction.

HLT enables intercepts of the HLT instruction.

INVLPG enables intercepts of the INVLPG instruction.

INVLPGA enables intercepts of the INVLPGA instruction.

IO_PROT enables intercepts of accesses to the I/O ports selected by the bitmap, the physical address of which is stored in the IOPM_BASE_PA field.

MSR_PROT enables intercepts of accesses to the model-specific registers selected by the bitmap, the physical address of which is stored in the MSRPM_BASE_PA field.

TASK_SWITCH enables intercepts of task switches.

FERR_FREEZE enables intercepts of processor freezing during legacy FERR handling.

SHUTDOWN enables intercepts of shutdown events.

VMRUN enables intercepts of the VMRUN instruction.

VMMCALL enables intercepts of the VMCALL instruction.

VMLOAD enables intercepts of the VMLOAD instruction.

VMSAVE enables intercepts of the VMSAVE instruction.

STGI enables intercepts of the STGI instruction.

CLGI enables intercepts of the CLGI instruction.

SKINIT enables intercepts of the SKINIT instruction.

RDTSCP enables intercepts of the RDTSCP instruction.

ICEBP enables intercepts of the ICEBP instruction.

WBINVD enables intercepts of the WBINVD instruction.

MONITOR enables intercepts of the MONITOR instruction.

MWAIT enables intercepts of the MWAIT instruction unconditionally.

ARMED_MWAIT enables intercepts of the MWAIT instruction if Monitor hardware is armed.

As we mentioned before, intercepts of I/O port accesses and of MSR accesses are enabled in external bitmaps. The `IOPM_BASE_PA` and the `MSRPM_BASE_PA` fields of the control area contain page-aligned physical addresses of the bitmaps. For each port/register the bitmaps store exactly two bits: the lower bit controls intercepts of read accesses, and the higher bit controls interrupts of write accesses. For detailed description of bit offset computations in the bitmaps, refer to the definitions of the `check_msr_intercept` and `check_io_intercept` functions in section N.2 and in section K.

So far we specified how to enable intercepts of the sensitive guest actions. When an action is intercepted, the processor exits guest mode and writes the exit code and exit information into the `EXITCODE`, `EXITINFO1`, `EXITINFO2`, `EXITINTINFO` fields of the control area. The exit code uniquely identifies the action, and the exit information describes the action parameters so that the host could emulate the action. For the list of the exit code refer to section P.4. Layout of the exit information fields is intercept specific. In other words, the meaning of the exit information depends on the sensitive action that triggered the intercept. When formally defining a sensitive action, we always write a statement that checks for intercept conditions. If the conditions hold, we use the **`fail`** statement of our specification language to stop the action and signal the intercept. As an argument for the **`fail`** statement, we give a value of the following type:

```
record Intercept
  field valid::bit
  field exitcode::ExitCode
  field exitinfo1::bits 64
  field exitinfo2::bits 64
  field exitintinfo::EventInfo
```

Thus, one can learn the meaning of the exit information fields by reading the specification of the sensitive actions and examining the values in the `exitinfo1`, `exitinfo2`, and `exitintinfo` fields. Most of the intercepts have no associated information or have only the `exitinfo1`. The following helper functions construct a value of the `Intercept` type:

```
function intercept(ecode::ExitCode)::Intercept
  = Intercept with [valid = 1, exitcode = ecode]
function intercept1(ecode::ExitCode, info1::bits 64)::Intercept
  = Intercept with [valid = 1, exitcode = ecode, exitinfo1 = info1]
```

We conclude this section with the formal definition of the control area:

```
layout VMCB_CA
  field READ_CR::bits 16
  field WRITE_CR::bits 16
  field READ_DR::bits 16
  field WRITE_DR::bits 16
  field EXCP::bits 32
  field intercept::Intercepts
  field rsv1::bits 32 ignored
  field rsv2::bits 64 ignored
  field rsv3::bits 64 ignored
  field rsv4::bits 64 ignored
  field rsv5::bits 64 ignored
  field rsv6::bits 64 ignored
  field IOPM_BASE_PA::bits 64
  field MSRPM_BASE_PA::bits 64
```

```
field TSC_OFFSET::bits 64
field GUEST_ASID::bits 32
field TLB_CONTROL::bits 8
field rsv7::bits 24
field V_INTR::VirtualIntr
field INTERRUPT_SHADOW::bit
field rsv8::bits 63 ignored
field EXITCODE::bits 64
field EXITINFO1::bits 64
field EXITINFO2::bits 64
field EXITINTINFO::EventInfo
field NESTED_PAGING::bits 64
field rsv11::bits 64 ignored
field rsv12::bits 64 ignored
field EVENTINJ::EventInfo
field N_CR3::bits 64
field LBR_VE::bits 64
```

READ_CR the i-th bit of this field enables intercept of reads from the CR[i].

WRITE_CR the i-th bit of this field enables intercept of writes to the CR[i].

READ_DR the i-th bit of this field enables intercept of reads from the DR[i].

WRITE_DR the i-th bit of this field enables intercept of writes to the DR[i].

EXCP the i-th bit of this field enables intercept of the exception with the vector i.

intercept each bit of this field enables intercept of some event or some instruction.

IOPM_BASE_PA contains the page-aligned physical base address of the I/O permission bitmap.

MSRPM_BASE_PA contains the page-aligned physical base address of the model-specific permission bitmap.

TSC_OFFSET contains the timestamp counter offset, which is added to the result of the RDTSC and RDTSCP instruction in guest mode.

GUEST_ASID contains the address space identifier that is used as a tag in the TLB during guest execution.

TLB_CONTROL if this field is 01h, then the TLB is flushed before executing the first instruction of the guest.

V_INTR controls virtual interrupts, we will discuss it in the next section.

INTERRUPT_SHADOW if this bit is set then the guest is in interrupt shadow, which means that the interrupts are not recognized until after the first guest instruction successfully completes.

EXITCODE on guest exit the processor writes the exit code into this field.

EXITINFO1 contains complementary information for the exit code.

EXITINFO2 contains complementary information for the exit code.

EXITINTINFO if the guest exit is triggered during control transfer to an interrupt handler, this field contains information about the interrupt.

NESTED_PAGING when this field is 1 then nested page tables are used to translate guest physical address to system physical addresses.

EVENTINJ contains an interrupt or an exception that is to be enjected in the guest before execution its first instruction.

N_CR3 contains the base address of the root nested page table.

LBR_VE enables last-branch virtualization (we do not model last branch).

17.3 Injected Events and Virtual Interrupts

The host can inject events into the guest by writing to the EVENTINJ field of the guest control area. This field has the following layout:

```
layout EventInfo
  field VECTOR::bits 8
  field TYPE::bits 3
  field EV::bit
  field rsv::bits 19 ignored
  field V::bit
  field ERRORCODE::bits 32
```

VECTOR specifies the vector of the injected exception or maskable interrupt.

TYPE is the type of the injected event: maskable interrupt, non-maskable interrupt, exception, or software interrupt.

EV when this bit is set, the injected event has an associated error code, which is stored in the ERRORCODE field and will be given to the handler.

V when this bit is set, the injected event is valid and will be delivered to the guest before executing its first instruction.

ERRORCODE contains the error code associated with the injected event.

The TYPE field of the injected event can take the following values:

```
set InjEventType::bits 3 = {EVENT_INTR = 000b, EVENT_NMI = 010b,
                            EVENT_EXCEPTION = 011b, EVENT_SOFT_INT = 100b}
```

The first value indicates that the injected event is an ordinary maskable interrupt. The second value means that the injected event is a non-maskable interrupt. In this case, the VECTOR field is ignored because non-maskable interrupts always have the vector equal to 2. By writing the third value, the host can inject exceptions. The fourth value indicates that the injected event is to be delivered to the guest as if the guest executed the INT instruction, which generates a software interrupt.

Injected events are delivered in the guest unconditionally, regardless of the interrupt mask flag RFLAGS.IF. Injected events have priority before other events. There is another mechanism, called virtual interrupts, which allows to inject maskable interrupts that behave more like ordinary maskable interrupts. The host can inject virtual interrupts by writing to the V_INTR field of the guest control area. The field has the following layout:

```
layout VirtualIntr
  field V_TPR::bits 8
  field V_IRQ::bit
  field rsv7::bits 7
  field V_INTR_PRIO::bits 4
  field V_IGN_TPR::bit
  field rsv8::bits 3
  field V_INTR_MASKING::bit
  field rsv9::bits 7
  field V_INTR_VECTOR::bits 8
  field rsv10::bits 24
```

V_TPR is the virtual task priority level (TPR) of the guest. When the guest is running and the V_INTR_MASKING bit is set, the V_TPR plays the same role for virtual interrupts as the APIC TPR register does for real interrupts. In other words, the all virtual interrupts that have the priority lower than the priority specified by the V_TPR are masked.

V_IRQ indicates that the virtual interrupt is active. The processor clears this bit before control transfer to the handler of the virtual interrupt.

V_INTR_PRIO is the priority of the virtual interrupt, the value in this field gets compared with the V_TPR if the V_INTR_MASKING is set, or with the real TPR otherwise.

V_IGN_TPR when this bit is set, the processor does not check the priority of the virtual interrupt.

V_INTR_MASKING enables virtual TPR and virtual RFLAGS.IF for virtual interrupts. When this bit is set, the processor uses the real TPR and the host value of the RFLAGS.IF for processing real interrupts. The virtual TPR and the guest value of the RFLAGS.IF are used only for virtual interrupts. When the bit is cleared, the processor uses the real TPR and the guest value of the RFLAGS.IF for all interrupts.

V_INTR_VECTOR is the vector of the virtual interrupt.

In chapter 9 we used the a few functions and actions that check the state of the injected and virtual interrupt and mark them as delivered. We are going to define the function and action here.

The following function returns the injected event:

```
function injected_event::Event
  = Event with [ valid = _vmcb_ca.EVENTINJ.V,
                 type = convert_type(_vmcb_ca.EVENTINJ.TYPE),
                 vector = _vmcb_ca.EVENTINJ.VECTOR,
                 ecode_valid = _vmcb_ca.EVENTINJ.EV,
                 ecode = _vmcb_ca.EVENTINJ.ERRORCODE,
                 data_valid = 0b,
                 injected = 1b
               ]
```

Where the Event is a type for general events, and was defined in chapter 9 as follows:

```
record Event
  field valid::bit
  field vector::bits 8
  field type::EventType
  field ecode_valid::bit
  field ecode::bits 32
  field data_valid::bit
  field data::bits 64
  field injected::bit
```

The EventType is a set of all possible events, including injected events:

```
set EventType = {INIT, SIPI, NMI, INTR, VINTR, SPURIOUS, EXCP,
                 SOFT_INT}
```

The convert_type function maps the injected event types into general event types:

```
function convert_type(inj_type::InjEventType)::EventType
 = if inj_type == EVENT_INTR then INTR
   elif inj_type == EVENT_NMI then NMI
   elif inj_type == EVENT_EXCEPTION then EXCP
   else SOFT_INT
```

The following function checks whether a virtual interrupt is pending or not:

```
function vintr_pending::bit
 = if _vmcb_ca.VINTR.V_IRQ then
     if _vmcb_ca.VINTR.V_INTR_MASKING then
       (_vmcb_ca.VINTR.V_INTR_PRIO > _vmcb_ca.VINTR.V_TPR[7:4])
     else
       (_vmcb_ca.VINTR.V_INTR_PRIO > CR8[3:0])
   else 0b
```

The following function returns the vector of the virtual interrupt:

```
function vintr_vector::bits 8
 = _vmcb_ca.VINTR.V_INTR_VECTOR
```

The following two actions mark the injected event and the virtual interrupt as delivered, making them inactive:

```
action vintr_delivered
  write 0b to _vmcb_ca.VINTR.V_IRQ
action injected_delivered
  write 0b to _vmcb_ca.EVENTINJ.V
```

17.4 Host State Save Area

Before switching to guest mode, the VMRUN instruction saves a subset of the host registers in host state save area . This area is a 4K page at the physical address that is stored in the VM_HSAVE_PA register. After switching back to host mode, the VMRUN restores the registers from this area. The layout of the area is implementation dependent and is left unspecified by the official manuals. As we need some layout in order to define the VMRUN instruction, we will use the following layout which contains a minimal subset of the host registers:

```
layout HOST_SSA
  field ES_sel::bits 16
  field ES_attr::bits 16
  field ES_limit::bits 32
  field ES_base::bits 64
  field CS_sel::bits 16
  field CS_attr::bits 16
  field CS_limit::bits 32
  field CS_base::bits 64
  field SS_sel::bits 16
  field SS_attr::bits 16
  field SS_limit::bits 32
  field SS_base::bits 64
  field DS_sel::bits 16
```

```
field DS_attr::bits 16
field DS_limit::bits 32
field DS_base::bits 64
field rsv::bits 32 ignored
field GDTR_limit::bits 16
field IDTR_limit::bits 16
field GDTR_base::bits 64
field IDTR_base::bits 64
field EFER::EFER
field CR4::CR4
field CR3::CR3
field CR0::CR0
field RFLAGS::Flags
field RIP::bits 64
field RSP::bits 64
field RAX::bits 64
field PAT::bits 64
```

From the names of the fields, one can easily deduce which registers are saved and restored. We define the `VMRUN`, `VMLOAD`, and `VMSAVE` in chapter P.

CHAPTER

EIGHTEEN

INSTRUCTIONS

In the remainder of the document we formally specify instructions in our domain-specific language. We group the instructions as follows:

Move instructions:

```
CMOVcc, CMPXCHG, CMPXCHG8B-CMPXCHG16B,
MOV, MOVSX, MOVSXD, MOVZX, SETcc, XCHG.
```

Arithmetic instructions:

```
ADC, ADD, CMP, DEC, DIV, IDIV, IMUL,
INC, MUL, NEG, SBB, SUB, XADD.
```

Logic instructions:

```
AND, NOT, OR, TEST, XOR.
```

Bit-string instructions:

```
BSF, BSR, BSWAP, BT, BTC, BTR, BTS,
CBW-CWDE-CDQE, CWD-CDQ-CQO, RCL, RCR,
ROL, ROR, SAL-SHL, SAR, SHLD, SHR, SHRD,
```

BCD instructions:

```
AAA, AAD, AAM, AAS, DAA, DAS.
```

Flag instructions:

```
CLC, CLD, CLI, CMC, LAHF, SAHF,
STC, STD, STI.
```

Stack instructions:

```
ENTER, LEAVE, POP, POPA, POPF,
PUSH, PUSHA, PUSHF.
```

Near control transfer instructions:

```
CALL, JCXZ, JMP, Jcc, LOOPcc, RET.
```

Far control transfer instructions:

```
SYSCALL, SYSRET, SYSENTER, SYSEXIT,
JMP, CALL, INT, INT3, INT0, RETF, IRET.
```

String instructions:

```
CMPS, LODS, MOVS, SCAS, STOS.
```

Input/Output instructions:

```
IN, INS, OUT, OUTS.
```

Segmentation instructions:

```
LDS, LES, LFS, LGDT, LGS, LIDT, LLDT,
LSS, LTR, SGDT, SIDT, SLDT, STR, SWAPGS.
```

Protection instructions:

```
ARPL, LAR, LSL, VERR, VERW.
```

CR and MSR access instructions:

```
LMSW, MOV(CRn), RDMSR, RDTSC, RDTSCP, SMSW, WRMSR.
```

Memory-management instructions:

```
CLFLUSH, INVD, INVLPG, INVLPGA,
LFENCE, MFENCE, SFENCE, WBINVD.
```

Virtualization instructions:

```
VMRUN, VMMCALL, VMLOAD, VMSAVE, CLGI, STGI.
```

Miscellaneous instructions:

```
BOUND, HLT, LEA, XLAT, PAUSE.
```

Each group has a separate chapter in the Appendix. Specification of an instruction in a group starts with a citation from the official manuals. Then we give a list of opcodes that decode into the instruction. Each opcode has a list of operands. For each operand we specify its width and type. Finally, we specify a semantic action that describes how the instructions executes. A semantic action is a statement in our domain-specific language, the statement can read from special variables `op1`, `op2`, `op3`, which represent the values of the operands. When the statement declares primes versions of the special variables `op1'` ,`op2'`, `op3'`, this means that the instruction writes to the associated operands.

As an example, consider the specification of the `MOVZX` instruction:

Instruction MOVZX

Copies the value in a register or memory location (second operand) into a register (first operand), zero- extending the value to fit in the destination register. The operand-size attribute determines the size of the zero-extended value.

[cited from AMD volume 3]

```
opcode "0FB6h" reg $v, reg_mem 8           : let op1' = zxt($v, op2)
opcode "0FB7h" reg max($v, 16), reg_mem 16 : let op1' = zxt(max($v, 16), op2)
```

The semantic action `let op1' = zxt($v, op2)` means the instruction reads the value of the second operand, zero-extends it to the instruction operand width `$v` and writes the new value to the first operand. Thus, the explicit specification of instruction execution has a `read_op` action before the semantic action, and a `write_op` action after the semantic action:

```
let $n1 = op1width(_prefix, _opcode, _modrm)::[1..128]
let $n2 = op2width(_prefix, _opcode, _modrm)::[1..128]
let $n = $n1
let t1 = op1type(_prefix, _opcode, _modrm)
let t2 = op2type(_prefix, _opcode, _modrm)
op2 = read_op($n2, t2)
let op1' = zxt($v, op2)
write_op($n1, t1, op1')
```

where the `op1width`, `op2width`, `op1type`, `op2type` functions return the types and widths of the operands.

More complex instructions have semantic actions that invoke other actions, that are defined in the main part of the document or are defined immediately after the instruction.

CHAPTER

NINETEEN

CONCLUSION

We developed an operational model for the x86 instruction set architecture of a modern multiprocessor machine. The model is a nondeterministic abstract machine that executes instructions on multiple processors. We specified the abstract machine in two parts: first we specified the memory model, and then we specified instruction fetch, decode, and execution. Instruction semantics was defined in a domain specific language locally for a single core. By transforming the single-core instruction specification and plugging it into the transition system, we obtained a multiprocessor x86 model.

19.1 Validating the model

Our model might contain the following types of errors:

1. the memory model forbids memory access reordering that can occur in real hardware;
2. the memory model allows memory access reordering that can never occur in real hardware;
3. sequential instruction execution is incorrect, i.e. register and memory contents after instruction execution in the abstract machine with a single processor do not agree with those in real hardware with a single processor.

We could validate the absence of errors of the first type by running multiprocessor test programs on real computer and checking whether any forbidden reordering has actually occurred. Additionally, we could construct a simplified hardware that implements our model or to make the model executable. This would at least show that the model is not inconsistent.

In order to make the model executable we need to schedule the transitions of the abstract machine in such a way that it makes progress in executing instructions. Recall that each transition is parameterized with arguments and has a guard condition. So we

need to find a method to fill-in the arguments of each transition such that the guard condition of the transition is satisfied and the transition effects contribute to the progress. For many transitions it is obvious what arguments satisfy the guard conditions. For the remaining transitions we can use the following scheme:

- iterate over each processor core i in the abstract machine:
  - if core i is not in the $decode$, $execute$, $vmexit$, $JISR1$, $JISR2$ phase then try to trigger all possible transitions in the phase. The transitions are simple and it is easy to choose arguments that satisfy the guard conditions.
  - otherwise, core i needs to execute one the actions defined in our domain specific language: interpret the action statements. One of the following outcomes can occur:
    - the action needs to fetch a translation for a virtual address from the TLB:
      1. if the corresponding complete walk is not in the TLB, then initiate a walk for the requested virtual address and extend the walk until it is complete.
      2. return the result to the core, by writing to its $tlb\text{-}in$ buffer.
    - the action needs to fetch code or data at a physical address from the memory:
      1. make cache transitions for that physical address to fill in the line if the access memory type is cacheable.
      2. make load buffer transitions to fetch the code or data from the cache/memory.
      3. return the result to the core, by writing to its $mem\text{-}in$ buffer.
    - the action raises an exception or an intercept: make core transitions that change phase to $JISR1/2$ or $vmexit$.
    - the action completes: update the registers and save the memory write accesses and make core transition to the next phase.
  - make a store buffer transition if possible. In case the store buffer tries to commit a write access to the cache, and the cache does not have the corresponding line, make cache transition to fill-in the line.
  - make an APIC transition of possible
- iterate over all pair of processor cores i, j in the abstract machine and make an IPI transition for that pair if possible.

Thus, the above scheme makes round-robin transition scheduling for the processor cores, the store buffers, the APICs, and the IPI. The load buffers, the TLBs, and the caches make transitions depending on processor core requests.

Errors of the second type are not as critical as error of the first type. They can be detected when one tries to prove correctness of programs which rely on the fact that certain memory access reorderings are not possible. If our model allows 'impossible' memory access reorderings, then one will not be able to prove such programs.

Errors of the third type can be detected by test programs on the executable model and comparing the outcome with the expected outcome.

Part III
APPENDIX

APPENDIX

A

MOVE INSTRUCTIONS

Instruction MOV

Copies an immediate value or the value in a general-purpose register, or memory location (second operand) to a general-purpose register, or memory location. The source and destination must be the same size (byte, word, doubleword, or quadword) and cannot both be memory locations. In opcodes A0 through A3, the memory offsets (called moffsets) are address sized. In 64-bit mode, memory offsets default to 64 bits. Opcodes A0-A3, in 64-bit mode, are the only cases that support a 64-bit offset value. (In all other cases, offsets and displacements are a maximum of 32 bits.) The B8 through BF (B8 +rq) opcodes, in 64-bit mode, are the only cases that support a 64-bit immediate value (in all other cases, immediate values are a maximum of 32 bits). It is possible to move a null segment selector value (0000–0003h) into the DS, ES, FS, or GS register. This action does not cause a general protection fault, but a subsequent reference to such a segment does cause a #GP exception. When the MOV instruction is used to load the SS register, the processor blocks external interrupts until after the execution of the following instruction. This action allows the following instruction to be a MOV instruction to load a stack pointer into the ESP register (MOV ESP,val) before an interrupt occurs. However, the LSS instruction provides a more efficient method of loading SS and ESP. *[cited from AMD volume 3]*

```
opcode "88h" reg_mem 8, reg 8     : let op1' = op2
opcode "89h" reg_mem $v, reg $v   : let op1' = op2
opcode "8Ah" reg 8, reg_mem 8     : let op1' = op2
opcode "8Bh" reg $v, reg_mem $v   : let op1' = op2
opcode "8Ch" reg_mem $vw, sreg 16 : let op1' = zxt($vw, op2)
opcode "8Eh" sreg 16, reg_mem 16  : let op1' = op2
opcode "A0h" rax 8, moffset 8     : let op1' = op2
opcode "A1h" rax $v, moffset $v   : let op1' = op2
opcode "A2h" moffset 8, rax 8     : let op1' = op2
opcode "A3h" moffset $v, rax $v   : let op1' = op2
opcode "B0h" reg 8, imm 8         : let op1' = op2
opcode "B8h" reg $v, imm $v       : let op1' = op2
opcode "C6h" reg_mem 8, imm 8     : let op1' = op2
```

```
opcode "C7h" reg_mem $v, imm $z   : let op1' = sxt($v, op2)
```

The real work happens in the `read_op` and the `write_op` actions that are invoked before and after the execute statement of this instruction. These action are defined in section 14.13.

Instruction MOVSX

Copies the value in a register or memory location (second operand) into a register (first operand), extending the most significant bit of an 8-bit or 16-bit value into all higher bits in a 16-bit, 32-bit, or 64-bit register.

[cited from AMD volume 3]

```
opcode "0FBEh" reg $v, reg_mem 8           : let op1' = sxt($v, op2)
opcode "0FBFh" reg max($v, 16), reg_mem 16 : let op1' = sxt(max($v, 16), op2)
```

Instruction MOVZX

Copies the value in a register or memory location (second operand) into a register (first operand), zero- extending the value to fit in the destination register. The operand-size attribute determines the size of the zero-extended value.

[cited from AMD volume 3]

```
opcode "0FB6h" reg $v, reg_mem 8           : let op1' = zxt($v, op2)
opcode "0FB7h" reg max($v, 16), reg_mem 16 : let op1' = zxt(max($v, 16), op2)
```

Instruction MOVSXD (`when $x64_mode`)

Copies the 32-bit value in a register or memory location (second operand) into a 64-bit register (first operand), extending the most significant bit of the 32-bit value into all higher bits of the 64-bit register. This instruction requires the REX prefix 64-bit operand size bit (REX.W) to be set to 1 to sign-extend a 32-bit source operand to a 64-bit result. Without the REX operand-size prefix, the operand size will be 32 bits, the default for 64-bit mode, and the source is zero-extended into a 64-bit register. With a 16- bit operand size, only 16 bits are copied, without modifying the upper 48 bits in the destination. This instruction is available only in 64-bit mode. In legacy or compatibility mode this opcode is interpreted as ARPL.

[cited from AMD volume 3]

```
opcode "63h" reg $v, reg_mem $z : let op1' = sxt($v, op2)
```

Instruction CMOVcc (`when $CPUID_80000001_CMOV`)

Conditionally moves a 16-bit, 32-bit, or 64-bit value in memory or a general-purpose register (second operand) into a register (first operand), depending upon the settings of condition flags in the rFLAGS register. If the condition is not satisfied, the instruction has no effect. The mnemonics of CMOVcc instructions denote the condition that must be satisfied. Most assemblers provide instruction mnemonics with A (above) and B (below) tags to supply the semantics for manipulating unsigned integers. Those with G (greater than) and L (less than) tags deal with signed integers. Many opcodes may be represented by synonymous mnemonics. For example, the CMOVL instruction is synonymous with the CMOVNGE instruction and denote the instruction with the opcode 0F 4C. Support for CMOVcc instructions depends on the processor implementation. To determine whether a processor can perform CMOVcc instructions, use the CPUID

instruction to determine whether EDX bit 15 of CPUID function 0000_0001h or function 8000_0001h is set to 1. *[cited from AMD volume 3]*

```
opcode "0F40h" reg $v, reg_mem $v
opcode "0F41h" reg $v, reg_mem $v
opcode "0F42h" reg $v, reg_mem $v
opcode "0F43h" reg $v, reg_mem $v
opcode "0F44h" reg $v, reg_mem $v
opcode "0F45h" reg $v, reg_mem $v
opcode "0F46h" reg $v, reg_mem $v
opcode "0F47h" reg $v, reg_mem $v
opcode "0F48h" reg $v, reg_mem $v
opcode "0F49h" reg $v, reg_mem $v
opcode "0F4Ah" reg $v, reg_mem $v
opcode "0F4Ah" reg $v, reg_mem $v
opcode "0F4Ch" reg $v, reg_mem $v
opcode "0F4Dh" reg $v, reg_mem $v
opcode "0F4Eh" reg $v, reg_mem $v
opcode "0F4Fh" reg $v, reg_mem $v
call CMOVcc
action CMOVcc
  if cc(Opcode[3:0]) then
    call op1 = read_op($v, reg)
    call write_op($v, op1, reg_mem)
```

The cc function maps the lower 4 bits of the opcode to a condition on flags, in the following way:

```
function cc(code::bits 4)::bit =
   if code == 0h then RFLAGS.OF
   else if code == 1h then not RFLAGS.OF
   else if code == 2h then RFLAGS.CF
   else if code == 3h then not RFLAGS.CF
   else if code == 4h then RFLAGS.ZF
   else if code == 5h then not RFLAGS.ZF
   else if code == 6h then RFLAGS.CF or RFLAGS.ZF
   else if code == 7h then not (RFLAGS.CF or RFLAGS.ZF)
   else if code == 8h then RFLAGS.SF
   else if code == 9h then not RFLAGS.SF
   else if code == Ah then RFLAGS.PF
   else if code == Bh then not RFLAGS.PF
   else if code == Ch then RFLAGS.OF <> RFLAGS.SF
   else if code == Dh then RFLAGS.OF == RFLAGS.SF
   else if code == Eh then RFLAGS.ZF or RFLAGS.OF <> RFLAGS.SF
   else not RFLAGS.ZF and RFLAGS.OF == RFLAGS.SF
```

Instruction SETcc

Checks the status flags in the rFLAGS register and, if the flags meet the condition specified in the mnemonic (cc), sets the value in the specified 8-bit memory location or register to 1. If the flags do not meet the specified condition, SETcc clears the memory location or register to 0. Mnemonics with the A (above) and B (below) tags are intended for use when performing unsigned integer comparisons; those with G (greater) and L (less) tags are intended for use with signed integer comparisons. Software typically uses the SETcc instructions to set logical indicators. Like the CMOVcc instructions

(page 91), the SETcc instructions can replace two instructions - a conditional jump and a move. Replacing conditional jumps with conditional sets can help avoid branch-prediction penalties that may result from conditional jumps. If the logical value “true” (logical one) is represented in a high-level language as an integer with all bits set to 1, software can accomplish such representation by first executing the opposite SETcc instruction - for example, the opposite of SETZ is SETNZ - and then decrementing the result. A ModR/M byte is used to identify the operand. The reg field in the ModR/M byte is unused. *[cited from AMD volume 3]*

```
opcode "0F90h" reg_mem 8
opcode "0F91h" reg_mem 8
opcode "0F92h" reg_mem 8
opcode "0F93h" reg_mem 8
opcode "0F94h" reg_mem 8
opcode "0F95h" reg_mem 8
opcode "0F96h" reg_mem 8
opcode "0F97h" reg_mem 8
opcode "0F98h" reg_mem 8
opcode "0F99h" reg_mem 8
opcode "0F9Ah" reg_mem 8
opcode "0F9Bh" reg_mem 8
opcode "0F9Ch" reg_mem 8
opcode "0F9Dh" reg_mem 8
opcode "0F9Eh" reg_mem 8
opcode "0F9Fh" reg_mem 8
call op1' = SETcc
action SETcc::bits 8
  if cc(Opcode[3:0]) then return 01h
  else return 00h
```

Instruction XCHG

Exchanges the contents of the two operands. The operands can be two general-purpose registers or a register and a memory location. If either operand references memory, the processor locks automatically, whether or not the LOCK prefix is used and independently of the value of IOPL. For details about the LOCK prefix, see “Lock Prefix” on page 8. The x86 architecture commonly uses the XCHG EAX, EAX instruction (opcode 90h) as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h as a true NOP only if it would exchange rAX with itself. Without this special handling, the instruction would zero-extend the upper 32 bits of RAX, and thus it would not be a true no-operation. Opcode 90h can still be used to exchange rAX and r8 if the appropriate REX prefix is used. This special handling does not apply to the two-byte ModRM form of the XCHG instruction. *[cited from AMD volume 3]*

```
opcode "86h" reg_mem 8, reg 8
opcode "87h" reg_mem $v, reg $v
opcode "90h" rax $v, rax_r8 $v
opcode "91h" rax $v, rcx_r9 $v
opcode "92h" rax $v, rdx_r10 $v
opcode "93h" rax $v, rbx_r11 $v
opcode "92h" rax $v, rsp_r12 $v
opcode "92h" rax $v, rbp_r13 $v
opcode "92h" rax $v, rsi_r14 $v
opcode "92h" rax $v, rdi_r15 $v
```

```
let (op1', op2') = (op2, op1)
```

Instruction CMPXCHG

Compares the value in the AL, AX, EAX, or RAX register with the value in a register or a memory location (first operand). If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the rFLAGS register to 1. Otherwise, it copies the value in the first operand to the AL, AX, EAX, or RAX register and clears the ZF flag to 0. The OF, SF, AF, PF, and CF flags are set to reflect the results of the compare. When the first operand is a memory operand, CMPXCHG always does a read-modify-write on the memory operand. If the compared operands were unequal, CMPXCHG writes the same value to the memory operand that was read. *[cited from AMD volume 3]*

```
opcode "0FB0h" reg_mem 8, reg 8
opcode "0FB1h" reg_mem $v, reg $v
call op1' = CMPXCHG($n, op1, op2)
```

Given the two operands, the `CMPXCHG` action compares the `RAX` register with the first operand using the `sub` action that subtracts the first operand from the `RAX` and sets the zero flags appropriately. If the result of subtraction is zero, then the `CMPXCHG` action returns the second operand. Otherwise, the action writes the first operand to the `RAX` and returns it. The `sub` action is defined in section B.2.

```
action CMPXCHG($n::{8, $v}, op1::bits $n, op2::bits $n)::bits $n
  let x = RAX[$n-1:0]
  call r = sub($n, x, op1, 0b)
  if r == zero($n) then return op2
  else write gpr($n, op1, RAX) to RAX
       return op1
```

Instruction CMPXCHG8B-CMPXCHG16B

Compares the value in the rDX:rAX registers with a 64-bit or 128-bit value in the specified memory location. If the values are equal, the instruction copies the value in the rCX:rBX registers to the memory location and sets the zero flag (ZF) of the rFLAGS register to 1. Otherwise, it copies the value in memory to the rDX:rAX registers and clears ZF to 0. If the effective operand size is 16-bit or 32-bit, the CMPXCHG8B instruction is used. This instruction uses the EDX:EAX and ECX:EBX register operands and a 64-bit memory operand. If the effective operand size is 64-bit, the CMPXCHG16B instruction is used; this instruction uses rdx:RAX and rcx:RBX register operands and a 128-bit memory operand. The CMPXCHG8B and CMPXCHG16B instructions always do a read-modify-write on the memory operand. If the compared operands were unequal, the instructions write the same value to the memory operand that was read. The CMPXCHG8B and CMPXCHG16B instructions support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 8. Support for the CMPXCHG8B and CMPXCHG16B instructions depends on the processor implementation. To find out if a processor can execute the CMPXCHG8B instruction, use the CPUID instruction to determine whether EDX bit 8 of CPUID function 0000_0001h or function 8000_0001h is set to 1. To find out if a processor can execute the CMPXCHG16B instruction, use the CPUID instruction to determine whether ECX bit 13 of CPUID function 0000_0001h is set to 1. *[cited from AMD volume 3]*

```
opcode "0FC7h" mem_pair 2*$qd : call op1' = CMPXCHG8B_16B(op1)
```

```
action CMPXCHG8B_16B(a::bits (2*$qd))::bits (2*$qd)
  let b = RDX[$qd-1:0] ++ RAX[$qd-1:0]
  if a == b then
    write 1b to RFLAGS.ZF
    return RCX[$qd-1:0] ++ RBX[$qd-1:0]
  else
    write 0b to RFLAGS.ZF
    write gpr($qd, a[$qd-1:0], RAX) to RAX
    write gpr($qd, a[2*$qd-1:$qd], RDX) to RDX
    return a
```

APPENDIX

B

ARITHMETIC INSTRUCTIONS

B.1 Addition

Instruction ADC

Adds the carry flag (CF), the value in a register or memory location (first operand), and an immediate value or the value in a register or memory location (second operand), and stores the result in the first operand location. The instruction cannot add two memory operands. The CF flag indicates a pending carry from a previous addition operation. The instruction sign-extends an immediate value to the length of the destination register or memory location. This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a carry in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result. Use the ADC instruction after an ADD instruction as part of a multibyte or multiword addition. *[cited from AMD volume 3]*

```
opcode "10h" reg_mem 8, reg 8
opcode "11h" reg_mem $v, reg $v
opcode "12h" reg 8, reg_mem 8
opcode "13h" reg $v, reg_mem $v
opcode "14h" rax 8, imm 8
opcode "15h" rax $v, imm $z
opcode "80h /010b" reg_mem 8, imm 8
opcode "81h /010b" reg_mem $v, imm $z
opcode "83h /010b" reg_mem $v, imm 8
call op1' = add($n, op1, sxt($n, op2), RFLAGS.CF)
```

Given the operand width, the operands and the carry bit, the `add` action updates the status flags and returns the result of the addition.

```
action add($n::{4, 8, 16, 32, 64}, a::bits $n, b::bits $n, carry::bit)::bits $n
  let res = a + b + zxt($n, carry)
  write carry_add($n, a, b, carry) to RFLAGS.CF
  write parity($n, res) to RFLAGS.PF
  write carry_add(4, a[3:0], b[3:0], carry) to RFLAGS.AF
```

```
  write (res == zero($n)) to RFLAGS.ZF
  write res[$n - 1] to RFLAGS.SF
  write overflow_add($n, a, b, carry) to RFLAGS.OF
  return res
```

We compute the carry bit of the addition by adding the operands zero-extended to `($n+1)` bits and looking-up the most-significant bit of the result.

```
function carry_add($n::{4, 8, 16, 32, 64},
                   a::bits $n, b::bits $n, carry::bit)::bit
  = let x = (0b++a) + (0b++b) + zxt($n+1, carry) in x[$n]
```

The overflow flag indicates that the sign-bit of the result is not equal to the sign-bit of the first operand.

```
function overflow_add($n::{4, 8, 16, 32, 64},
                      a::bits $n, b::bits $n, carry::bit)::bit
  = let x = a + b + zxt($n, carry) in x[$n-1] <> a[$n-1]
```

The parity flag is `1` if and only if the least significant bit of the result has even number of the set bits. The result of xoring the `8` least-significant bits is exactly the negation of the parity bit. Therefore, we can compute the parity bit by negating (the same as xoring with `1`) the result of the xor.

```
function parity(a::bits 08)::bit
  = a[0] ^ a[1] ^ a[2] ^ a[3] ^ a[4] ^ a[5] ^ a[6] ^ a[7] ^ 1
```

Instruction ADD

Adds the value in a register or memory location (first operand) and an immediate value or the value in a register or memory location (second operand), and stores the return in the first operand location. The instruction cannot add two memory operands. The instruction sign-extends an immediate value to the length of the destination register or memory operand. This instruction evaluates the return for both signed and unsigned data types and sets the OF and CF flags to indicate a carry in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result. *[cited from AMD volume 3]*

```
opcode "00h" reg_mem 8, reg 8
opcode "01h" reg_mem $v, reg $v
opcode "02h" reg 8, reg_mem 8
opcode "03h" reg $v, reg_mem $v
opcode "04h" rax 8, imm 8
opcode "05h" rax $v, imm $z
opcode "80h /000b" reg_mem 8, imm 8
opcode "81h /000b" reg_mem $v, imm $z
opcode "83h /000b" reg_mem $v, imm 8
call op1' = add($n, op1, sxt($n, op2), 0b)
```

Instruction XADD

Exchanges the contents of a register (second operand) with the contents of a register or memory location (first operand), computes the sum of the two values, and stores the result in the first operand location. *[cited from AMD volume 3]*

```
opcode "0FC0h" reg_mem 8, reg 8
```

```
opcode "0FC1h" reg_mem $v, reg $v
call (op1', op2') = xadd($n, op1, op2)
action xadd($n::{8, $v}, a::bits $n, b::bits $n)::(bits $n, bits $n)
  call sum = add($n, a, b, 0b)
  return (sum, a)
```

Instruction INC

Adds 1 to the specified register or memory location. The CF flag is not affected, even if the operand is incremented to 0000. *[cited from AMD volume 3]*

```
opcode "FEh /000b" reg_mem 8
opcode "FFh /000b" reg_mem $v
opcode "40h" rax $v          : when not $x64_mode
opcode "41h" rcx $v          : when not $x64_mode
opcode "42h" rdx $v          : when not $x64_mode
opcode "43h" RBX $v          : when not $x64_mode
opcode "44h" RSP $v          : when not $x64_mode
opcode "45h" RBP $v          : when not $x64_mode
opcode "46h" RSI $v          : when not $x64_mode
opcode "47h" RDI $v          : when not $x64_mode
call op1' = inc($n, op1)
action inc($n::{4, 8, 16, 32, 64}, a::bits $n)::bits $n
  let b = one($n)
  let res = a + b
  write parity($n, res) to RFLAGS.PF
  write carry_add(4, a[3:0], b[3:0], 0b) to RFLAGS.AF
  write (res == zero($n)) to RFLAGS.ZF
  write res[$n - 1] to RFLAGS.SF
  write overflow_add($n, a, b, 0b) to RFLAGS.OF
  return res
```

B.2 Subtraction

Instruction SBB

Subtracts an immediate value or the value in a register or a memory location (second operand) from a register or a memory location (first operand), and stores the result in the first operand location. If the carry flag (CF) is 8, the instruction subtracts 1 from the result. Otherwise, it operates like SUB. The SBB instruction sign-extends immediate value operands to the length of the first operand size. This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a borrow in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result. This instruction is useful for multibyte (multiword) numbers because it takes into account the borrow from a previous SUB instruction. *[cited from AMD volume 3]*

```
opcode "18h" reg_mem 8, reg 8
opcode "19h" reg_mem $v, reg $v
opcode "1Ah" reg 8, reg_mem 8
opcode "1Bh" reg $v, reg_mem $v
opcode "1Ch" rax 8, imm 8
opcode "1Dh" rax $v, imm $z
```

```
opcode "80h /011b" reg_mem 8, imm 8
opcode "81h /011b" reg_mem $v, imm $z
opcode "83h /011b" reg_mem $v, imm 8
call op1' = sub($n, op1, sxt($n, op2), RFLAGS.CF)
```

Given the operand width, the operands and the carry bit, the `sub` action updates the status flags and returns the result of the subtraction.

```
action sub($n::{4, 8, 16, 32, 64, 128},
           a::bits $n, b::bits $n, borrow::bit)::bits $n
  let res = a - b - zxt($n, borrow)
  write carry_sub($n, a, b, borrow) to RFLAGS.CF
  write parity($n, res) to RFLAGS.PF
  write carry_sub(4, a[3:0], b[3:0], borrow) to RFLAGS.AF
  write (res == zero($n)) to RFLAGS.ZF
  write res[$n - 1] to RFLAGS.SF
  write overflow_sub($n, a, b, borrow) to RFLAGS.OF
  return res
```

The `parity` function is defined in section B.1.

The carry flag of subtraction of two `$n` bit operands can be computed by performing subtracting the operands sign-extended to the `($n+1)` bits and comparing the most-significant bit of the result with the most-significant bit of the first operand.

```
function carry_sub($n::{4, 8, 16, 32, 64, 128},
                   a::bits $n, b::bits $n, borrow::bit)::bit
  = let x = (a[n-1]++a) - (b[n-1]++b) - zxt($n+1, borrow) in x[$n] <> a[$n]
```

The overflow flag indicates that the sign-bit of the result is not equalt to the sign-bit of the first bit.

```
function overflow_sub($n::{4, 8, 16, 32, 64, 128},
                      a::bits $n, b::bits $n, borrow::bit)::bit
  = let x = a - b - zxt($n, borrow) in x[$n-1] <> a[$n-1]
```

Instruction SUB

Subtracts an immediate value or the value in a register or memory location (second operand) from a register or a memory location (first operand) and stores the result in the first operand location. An immediate value is sign-extended to the length of the first operand. This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a borrow in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result. *[cited from AMD volume 3]*

```
opcode "28h" reg_mem 8, reg 8
opcode "29h" reg_mem $v, reg $v
opcode "2Ah" reg 8, reg_mem 8
opcode "2Bh" reg $v, reg_mem $v
opcode "2Ch" rax 8, imm 8
opcode "2Dh" rax $v, imm $z
opcode "80h /101b" reg_mem 8, imm 8
opcode "81h /101b" reg_mem $v, imm $z
opcode "83h /101b" reg_mem $v, imm 8
call op1' = sub($n, op1, sxt($n, op2), 0b)
```

Instruction DEC

Subtracts 1 from the specified register or memory location. The CF flag is not affected. *[cited from AMD volume 3]*

```
opcode "FEh /001b" reg_mem 8
opcode "FFh /001b" reg_mem $v
opcode "48h" rax $v          : when not $x64_mode
opcode "49h" rcx $v          : when not $x64_mode
opcode "4Ah" rdx $v          : when not $x64_mode
opcode "4Bh" RBX $v          : when not $x64_mode
opcode "4Ch" RSP $v          : when not $x64_mode
opcode "4Dh" RBP $v          : when not $x64_mode
opcode "4Eh" RSI $v          : when not $x64_mode
opcode "4Fh" RDI $v          : when not $x64_mode
call op1' = dec($n, op1)
```

The `dec` action is the same as the `sub` action except it does not update the `CF` flag and the second operand is one.

```
action dec($n::{4, 8, 16, 32, 64}, a::bits $n)::bits $n
  let b = one($n)
  let res = a - b
  write parity($n, res) to RFLAGS.PF
  write carry_sub(4, a[3:0], b[3:0], 0b) to RFLAGS.AF
  write (res == zero($n)) to RFLAGS.ZF
  write res[$n - 1] to RFLAGS.SF
  write overflow_sub($n, a, b, 0b) to RFLAGS.OF
  return res
```

Instruction NEG

Performs the two's complement negation of the value in the specified register or memory location by subtracting the value from 0. Use this instruction only on signed integer numbers. If the value is 0, the instruction clears the CF flag to 0; otherwise, it sets CF to 1. The OF, SF, ZF, AF, and PF flag settings depend on the result of the operation. *[cited from AMD volume 3]*

```
opcode "F6h /011b" reg_mem 8
opcode "F7h /011b" reg_mem $v
call op1' = sub($n, zero($n), op1, 0b)
```

B.3 Comparison

Instruction CMP

Compares the contents of a register or memory location (first operand) with an immediate value or the contents of a register or memory location (second operand), and sets or clears the status flags in the rFLAGS register to reflect the results. To perform the comparison, the instruction subtracts the second operand from the first operand and sets the status flags in the same manner as the SUB instruction, but does not alter the first operand. If the second operand is an immediate value, the instruction sign- extends the value to the length of the first operand. Use the CMP instruction to set the condition codes for a subsequent conditional jump (Jcc), conditional move (CMOVcc),

or conditional SETcc instruction. Appendix E, “Instruction Effects on RFLAGS,” shows how instructions affect the rFLAGS status flags. *[cited from AMD volume 3]*

```
opcode "38h" reg_mem 8, reg 8
opcode "39h" reg_mem $v, reg $v
opcode "3Ah" reg 8, reg_mem 8
opcode "3Bh" reg $v, reg_mem $v
opcode "3Ch" rax 8, imm 8
opcode "3Dh" rax $v, imm $z
opcode "80h /111b" reg_mem 8, imm 8
opcode "81h /111b" reg_mem $v, imm $z
opcode "83h /111b" reg_mem $v, imm 8
call sub($n, op1, sxt($n, op2), 0b)
```

B.4 Multiplication

Instruction MUL

Multiplies the unsigned byte, word, doubleword, or quadword value in the specified register or memory location by the value in AL, AX, EAX, or RAX and stores the result in AX, DX:AX, EDX:EAX, or rdx:RAX (depending on the operand size). It puts the high-order bits of the product in AH, DX, EDX, or rdx. If the upper half of the product is non-zero, the instruction sets the carry flag (CF) and overflow flag (OF) both to 1. Otherwise, it clears CF and OF to 0. The other arithmetic flags (SF, ZF, AF, PF) are undefined. *[cited from AMD volume 3]*

```
opcode "F6h /100b" reg_mem 8
opcode "F7h /100b" reg_mem $v
call mul_rax($n, op1, 0b)
```

The `mul_rax` gets the result from the `mul` action and writes it into the `RAX` register or the `RDX`, `RAX` register pair depending on the width of the operands.

```
action mul_rax($n::{8, 16, 32, 64}, a::bits $n, sign::bit)
  call res = mul($n, a, RAX[$n-1:0], sign)
  if $n == 8 then write res to RAX[2*$n-1:0]
  else write gpr($n, res[$n-1:0], RAX) to RAX
       write gpr($n, res[2*$n-1:$n], RDX) to RDX
```

The `mul` action takes the operand width, the operands, and the sign indicator. It updates the status flags and returns the double-width result of the multiplication.

```
action mul($n::{8, 16, 32, 64}, a::bits $n, b::bits $n, sign::bit)::bits (2*$n)
  undef RFLAGS.PF
  undef RFLAGS.AF
  undef RFLAGS.SF
  undef RFLAGS.ZF
  if sign then
    let res = sxt(2*$n, a) * sxt(2*$n, b)
    let overflow = res[2*$n-1:0] <> sxt(2*$n, res[$n-1:0])
    write overflow to RFLAGS.CF
    write overflow to RFLAGS.OF
    return res
```

```
else
  let res = zxt(2*$n, a) * zxt(2*$n, b)
  let overflow = res[2*$n-1:$n] <> zero($n)
  write overflow to RFLAGS.CF
  write overflow to RFLAGS.OF
  return res
```

Instruction IMUL

Multiplies two signed operands. The number of operands determines the form of the instruction. If a single operand is specified, the instruction multiplies the value in the specified general-purpose register or memory location by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and stores the product in AX, DX:AX, EDX:EAX, or rdx:RAX, respectively. If two operands are specified, the instruction multiplies the value in a general-purpose register (first operand) by an immediate value or the value in a general-purpose register or memory location (second operand) and stores the product in the first operand location. If three operands are specified, the instruction multiplies the value in a general-purpose register or memory location (second operand), by an immediate value (third operand) and stores the product in a register (first operand). The IMUL instruction sign-extends an immediate operand to the length of the other register/memory operand. The CF and OF flags are set if, due to integer overflow, the double-width multiplication result cannot be represented in the half-width destination register. Otherwise the CF and OF flags are cleared. *[cited from AMD volume 3]*

```
opcode "F6h /101b" reg_mem 8            : call mul_rax(8, op1, 1b)
opcode "F7h /101b" reg_mem $v           : call mul_rax($v, op1, 1b)
opcode "0FAFh" reg $v, reg_mem $v       : call op1' = IMUL($v, op1, op2)
opcode "6Bh" reg $v, reg_mem $v, imm 8  : call op1' = IMUL($v, op2, sxt($v, op3))
opcode "69h" reg $v, reg_mem $v, imm $z : call op1' = IMUL($v, op2, sxt($v, op3))
```

The `IMUL` action simply truncates the result of the `mul` action.

```
action IMUL($n::{8, 16, 32, 64}, a::bits $n, b::bits $n)::bits $n
  call res = mul($n, a, b, 1b)
  return res[$n-1:0]
```

B.5 Division

Instruction DIV

Divides the unsigned value in a register by the unsigned value in the specified register or memory location. The register to be divided depends on the size of the divisor. When dividing a word, the dividend is in the AX register. The instruction stores the quotient in the AL register and the remainder in the AH register. When dividing a doubleword, quadword, or double quadword, the most-significant word of the dividend is in the rDX register and the least-significant word is in the rAX register. After the division, the instruction stores the quotient in the rAX register and the remainder in the rDX register. The instruction truncates non-integral results towards 0 and the remainder is always less than the divisor. An overflow generates a #DE (divide error) exception, rather than setting the CF flag. Division by zero generates a divide-by-zero exception. *[cited from AMD volume 3]*

```
opcode "F6h /110b" reg_mem 8
opcode "F7h /110b" reg_mem $v
call div_rax($n, op1)
```

The `div_rax` action reconstructs the dividend from the `RAX` register or the `RDX`, `RAX` register pair depending on the operand width. After that, the action invokes the `div` action, which performs unsigned division and returns the quotient and the remainder.

```
action div_rax($n::{8, 16, 32, 64}, divisor::bits $n)::bits $n
  let dividend = if $n == 8 then RAX[15:0]
                 else RDX[$n-1:0]++RAX[$n-1:0]
  call (quo, rem) = div($n, dividend, divisor)
  write quo to RAX[$n-1:0]
  write rem to RDX[$n-1:0]
```

The `div` action takes the width of the divisor, the dividend, and the divisor and performs unsigned division. The width of the dividend is twice the width of the divisor. The action returns the quotient and the remainder. In case the divisor is zero or the quotient is too large, the action raises an exception.

```
action div($n::{8, 16, 32, 64}, a::bits (2*$n), b::bits $n)::(bits $n, bits $n)
  fail exception(xDE, 0000h) when b == zero($n)
  let quo = a / zxt(2*$n, b)
  let rem = a % zxt(2*$n, b)
  fail exception(xDE, 0000h) when quo[2*$n-1:$n] <> zero($n)
  return (quo[$n-1:0], rem[$n-1:0])
```

Instruction IDIV

Divides the signed value in a register by the signed value in the specified register or memory location. The register to be divided depends on the size of the divisor. When dividing a word, the dividend is in the AX register. The instruction stores the quotient in the AL register and the remainder in the AH register. When dividing a doubleword, quadword, or double quadword, the most-significant word of the dividend is in the rDX register and the least-significant word is in the rAX register. After the division, the instruction stores the quotient in the rAX register and the remainder in the rDX register. The instruction truncates non-integral results towards 0. The sign of the remainder is always the same as the sign of the dividend, and the absolute value of the remainder is less than the absolute value of the divisor. An overflow generates a #DE (divide error) exception, rather than setting the OF flag. To avoid overflow problems, precede this instruction with a CBW, CWD, CDQ, or CQO instruction to sign-extend the dividend. *[cited from AMD volume 3]*

```
opcode "F6h /111b" reg_mem 8
opcode "F7h /111b" reg_mem $v
call idiv_rax($n, op1)
```

The `idiv_rax` action reconstructs the dividend from the `RAX` register or the `RDX`, `RAX` register pair depending on the operand width. After that, the action invokes the `idiv` action, which performs signed division and returns the quotient and the remainder.

```
action idiv_rax($n::{8, 16, 32, 64}, divisor::bits $n)::bits $n
  let dividend = if $n == 8 then RAX[15:0]
                 else RDX[$n-1:0]++RAX[$n-1:0]
  call (quo, rem) = idiv($n, dividend, divisor)
```

```
  write quo to RAX[$n-1:0]
  write rem to RDX[$n-1:0]
```

The `idiv` action takes the width of the divisor, the dividend and the divisor and performs signed division. Since the division operator in our specification language performs only unsigned division, the `idiv` action makes division of the absolute values of the operands and then adjust the sign of the result.

```
action idiv($n::{8, 16, 32, 64}, a::bits (2*$n), b::bits $n)::(bits $n, bits $n)
  fail exception(xDE, 0000h) when b == zero($n)
  let c = sxt(2*$n+1, a)
  let d = sxt(2*$n+1, b)
  let (quo, rem) = if   not c[2*$n] and not d[2*$n] then (c/d, c%d)
                   elif     c[2*$n] and not d[2*$n] then (-((-c)/d), -((-c)%d))
                   elif     c[2*$n] and     d[2*$n] then ((-c)/(-d), -(-c)%(-d))
                   else (-(c/(-d)), c%(-d))
  fail exception(xDE, 0000h) when quo[2*$n:0] <> sxt(2*$n+1, quo[$n-1:0])
  return (quo[$n-1:0], rem[$n-1:0])
```

APPENDIX

C

LOGIC INSTRUCTIONS

Instruction AND

Performs a bitwise AND operation on the value in a register or memory location (first operand) and an immediate value or the value in a register or memory location (second operand), and stores the result in the first operand location. The instruction cannot AND two memory operands. The instruction sets each bit of the result to 1 if the corresponding bit of both operands is set; otherwise, it clears the bit to 0. *[cited from AMD volume 3]*

```
opcode "20h" reg_mem 8, reg 8
opcode "21h" reg_mem $v, reg $v
opcode "22h" reg 8, reg_mem 8
opcode "23h" reg $v, reg_mem $v
opcode "24h" rax 8, imm 8
opcode "25h" rax $v, imm $z
opcode "80h /100b" reg_mem 8, imm 8
opcode "81h /100b" reg_mem $v, imm $z
opcode "83h /100b" reg_mem $v, imm 8
call op1' = AND($n, op1, zxt($n, op2))
action AND($n::{8, $v}, a::bits $n, b::bits $n)::bits $n
  let r = a & b
  call flags_after_logic_op($n, r)
  return r
```

After any logic instruction specified in this section the flags are set as follows:

```
action flags_after_logic_op($n::{8, 16, 32, 64}, res::bits $n)
  write 0b to RFLAGS.CF
  write parity($n, res) to RFLAGS.PF
  undef RFLAGS.AF
  write res == zero($n) to RFLAGS.ZF
  write res[$n-1] to RFLAGS.SF
  write 0b to RFLAGS.OF
```

The `parity` function is defined in section B.1.

Instruction TEST

Performs a bit-wise logical AND on the value in a register or memory location (first operand) with an immediate value or the value in a register (second operand) and sets the flags in the rFLAGS register based on the result. While the AND instruction changes the contents of the destination and the flag bits, the TEST instruction changes only the flag bits. *[cited from AMD volume 3]*

```
opcode "84h" reg_mem 8, reg 8
opcode "85h" reg_mem $v, reg $v
opcode "A8h" rax 8, imm 8
opcode "A9h" rax $v, imm $z
opcode "F6h /000b" reg_mem 8, imm 8
opcode "F7h /000b" reg_mem $v, imm $z
call AND($n, op1, zxt($n, op2))
```

Instruction OR

Performs a logical OR on the bits in a register, memory location, or immediate value (second operand) and a register or memory location (first operand) and stores the result in the first operand location. The two operands cannot both be memory locations. *[cited from AMD volume 3]*

```
opcode "08h" reg_mem 8, reg 8
opcode "09h" reg_mem $v, reg $v
opcode "0Ah" reg 8, reg_mem 8
opcode "0Bh" reg $v, reg_mem $v
opcode "0Ch" rax 8, imm 8
opcode "0Dh" rax $v, imm $z
opcode "80h /001b" reg_mem 8, imm 8
opcode "81h /001b" reg_mem $v, imm $z
opcode "83h /001b" reg_mem $v, imm 8
call op1' = OR($n, op1, zxt($n, op2))
action OR($n::{8, $v}, a::bits $n, b::bits $n)::bits $n
  let r = a | b
  call flags_after_logic_op($n, r)
  return r
```

Instruction XOR

Performs a bitwise exclusive OR operation on both operands and stores the result in the first operand location. The first operand can be a register or memory location. The second operand can be an immediate value, a register, or a memory location. XOR-ing a register with itself clears the register. *[cited from AMD volume 3]*

```
opcode "30h" reg_mem 8, reg 8
opcode "31h" reg_mem $v, reg $v
opcode "32h" reg 8, reg_mem 8
opcode "33h" reg $v, reg_mem $v
opcode "34h" rax 8, imm 8
opcode "35h" rax $v, imm $z
opcode "80h /110b" reg_mem 8, imm 8
opcode "81h /110b" reg_mem $v, imm $z
opcode "83h /110b" reg_mem $v, imm 8
call op1' = XOR($n, op1, zxt($n, op2))
action XOR($n::{8, $v}, a::bits $n, b::bits $n)::bits $n
```

```
    let r = a ^ b
    call flags_after_logic_op($n, r)
    return r
```

Instruction NOT

Performs the one's complement negation of the value in the specified register or memory location by inverting each bit of the value. *[cited from AMD volume 3]*

```
opcode "F6h /010b" reg_mem 8
opcode "F7h /010b" reg_mem $v
let op1' = ~op1
```

Notice that the `NOT` instruction does not change any status flag.

APPENDIX

D

BIT STRING INSTRUCTIONS

D.1 Bit Test and Set

Instruction BT

Copies a bit, specified by a bit index in a register or 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register. If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the operand size) of the bit index to select a bit in the register. If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. If the bit index is in a register, the instruction selects a bit position relative to the bit base When the instruction attempts to copy a bit from memory, it accesses 2, 4, or 8 bytes starting from the specified memory address for 16-bit, 32-bit, or 64-bit operand sizes, respectively, using the following formula: Effective Address + (NumBytesi * (BitOffset DIV NumBitsi*8))*[cited from AMD volume 3]*

```
opcode "0FA3h" reg_mem $v, reg $v      : call BT(int(op2))
opcode "0FBAh /100b" reg_mem $v, imm 8 : call BT(nat(op2)%$v)
action BT(idx::int)
  call x = read_bit_container(idx)
  let $k = (idx % $v)::[0..$v-1]
  write x[$k] to RFLAGS.CF
```

The `read_bit_container` action splits the bit-string operand of the instruction into chunks of width `$v`. The action returns the chunk that contains the bit with the given index `idx`.

```
action read_bit_container(idx::int)::(bits $v)
  if _modrm.mod == 11b then
    call x = read_op($v, reg_rm)
    return x
  else
    let offset = (idx / $v) * ($v / 8)
    let s = segment(iDS)
    let origin = segment_origin(s)
```

```
    call x = lread(origin, $v, SR[s], $oa, ea + bits($oa, offset))
    return x
```

The _modrm byte defines whether the bit-string is in a register or in the memory. In the former case, we simply return the register value using the read_op action, which is defined in section 14.13. In the latter case, we compute the offset of the chunk that contains the specified bit and perform a logical read lread. Functions that compute the segment and the effective address of the memory operand are defined in section 14.12. The logical read action is defined in section 13.12.

Instruction BTC

Copies a bit, specified by a bit index in a register or 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register, and then complements (toggles) the bit in the bit string. If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the operand size) of the bit index to select a bit in the register. If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. If the bit index is in a register, the instruction selects a bit position relative to the bit base When the instruction attempts to copy a bit from memory, it accesses 2, 4, or 8 bytes starting from the specified memory address for 16-bit, 32-bit, or 64-bit operand sizes, respectively, using the following formula: Effective Address + (NumBytesi * (BitOffset DIV NumBitsi*8))*[cited from AMD volume 3]*

```
opcode "0FBBh" reg_mem $v, reg $v      : call BTC(int(op2))
opcode "0FBAh /111b" reg_mem $v, imm 8 : call BTC(nat(op2)%$v)
action BTC(idx::int)
  call x = read_bit_container(idx)
  let $k = (idx % $v)::[0..$v-1]
  write x[$k] to RFLAGS.CF
  call write_bit_container(idx, x ^ (zxt($v, 1b) << $k))
```

The write_bit_container action splits the bit-string operand of the instruction into chunks of width $v. The action writes the given chunk val into the chunk that contains the bit with the given index idx. Refer to the description of the read_bit_container for more explanation.

```
action write_bit_container(idx::int, val::bits $v)
  if ModRM.mod == 11b then
    call write_op($v, val, reg_rm)
  else
    let offset = (idx / $v)*($v / 8)
    let s = segment(iDS)
    let origin = segment_origin(s)
    call lwrite(origin, $v, val, SR[s], $oa, ea + bits($oa, offset))
```

Instruction BTR

Copies a bit, specified by a bit index in a register or 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register, and then clears the bit in the bit string to 0. If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the operand size) of the bit index to select a bit in the register. If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base

of the bit string. If the bit index is in a register, the instruction selects a bit position relative to the bit base When the instruction attempts to copy a bit from memory, it accesses 2, 4, or 8 bytes starting from the specified memory address for 16-bit, 32-bit, or 64-bit operand sizes, respectively, using the following formula: Effective Address + (NumBytesi * (BitOffset DIV NumBitsi*8))*[cited from AMD volume 3]*

```
opcode "0FB3h" reg_mem $v, reg $v      : call BTR(int(op2))
opcode "0FBAh /110b" reg_mem $v, imm 8 : call BTR(nat(op2)%$v)
action BTR(idx::int)
  call x = read_bit_container(idx)
  let $k = (idx % $v)::[0..$v-1]
  write x[$k] to RFLAGS.CF
  call write_bit_container(idx, x ^ (zxt($v, x[$k]) << $k))
```

Instruction BTS

Copies a bit, specified by bit index in a register or 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register, and then sets the bit in the bit string to 1. If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the operand size) of the bit index to select a bit in the register. If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. If the bit index is in a register, the instruction selects a bit position relative to the bit base When the instruction attempts to copy a bit from memory, it accesses 2, 4, or 8 bytes starting from the specified memory address for 16-bit, 32-bit, or 64-bit operand sizes, respectively, using the following formula: Effective Address + (NumBytesi * (BitOffset DIV NumBitsi*8))*[cited from AMD volume 3]*

```
opcode "0FABh" reg_mem $v, reg $v      : call BTS(int(op2))
opcode "0FBAh /101b" reg_mem $v, imm 8 : call BTS(nat(op2)%$v)
action BTS(idx::int)
  call x = read_bit_container(idx)
  let $k = (idx % $v)::[0..$v-1]
  write x[$k] to RFLAGS.CF
  call write_bit_container(idx, x | (zxt($v, 1b) << $k))
```

D.2 Bit Search

Instruction BSF

Searches the value in a register or a memory location (second operand) for the least-significant set bit. If a set bit is found, the instruction clears the zero flag (ZF) and stores the index of the least-significant set bit in a destination register (first operand). If the second operand contains 0, the instruction sets ZF to 1 and does not change the contents of the destination register. The bit index is an unsigned offset from bit 0 of the searched value. *[cited from AMD volume 3]*

```
opcode "0FBCh" reg $v, reg_mem $v : call BSF(op2)
action BSF(x::bits $v)
  if x == zero($v) then write 1b to RFLAGS.ZF
  else let idx = tzcnt64(zxt(64, x))
       call write_op($v, bits($v, idx), reg)
       write 0b to RFLAGS.ZF
```

The `tzcnt64` function returns the number of the trailing zeros in the given 64-bit wide bit-string. Notice that the number of trailing zero coincides with the index of the least-significant set bit if there exists a set bit. The function is defined in terms of the `tzcnt32` function which counts the number of trailing zeros in a 32-bit wide bit-string. We apply the same pattern several times, until we get to the `tzcnt04` function, which explicitly counts the number of trailing zeros.

```
function tzcnt64(x::bits 64)::[0..64]
  = if x[31:0] <> zero(32) then tzcnt32(x[31:0])
    else 32 + tzcnt32(x[63:32])
function tzcnt32(x::bits 32)::[0..32]
  = if x[15:0] <> zero(16) then tzcnt16(x[15:0])
    else 16 + tzcnt16(x[31:16])
function tzcnt16(x::bits 16)::[0..16]
  = if x[7:0] <> zero(08) then tzcnt08(x[7:0])
    else 8 + tzcnt08(x[15:8])
function tzcnt08(x::bits 8)::[0..8]
  = if x[3:0] <> zero(04) then tzcnt04(x[3:0])
    else 4 + tzcnt04(x[7:4])
function tzcnt04(x::bits 4)::[0..4]
  = if x[0] then 0
    elif x[1] then 1
    elif x[2] then 2
    elif x[3] then 3
    else 4
```

Instruction BSR

Searches the value in a register or a memory location (second operand) for the most-significant set bit. If a set bit is found, the instruction clears the zero flag (ZF) and stores the index of the most-significant set bit in a destination register (first operand). If the second operand contains 0, the instruction sets ZF to 1 and does not change the contents of the destination register. The bit index is an unsigned offset from bit 0 of the searched value. *[cited from AMD volume 3]*

```
opcode "0FBDh" reg $v, reg_mem $v : call BSR(op2)
action BSR(x::bits $v)
  if x == zero($v) then write 1b to RFLAGS.ZF
  else let idx = 64-lzcnt64(zxt(64, x))
       call write_op($v, bits($v, idx), reg)
       write 0b to RFLAGS.ZF
```

The `lzcnt64` function counts the number of the leading zeros. Refer to the description of the `tzcnt64` function for more explanation.

```
function lzcnt64(x::bits 64)::[0..64]
  = if x[63:32] <> zero(32) then lzcnt32(x[63:32])
    else 32 + lzcnt32(x[31:0])
function lzcnt32(x::bits 32)::[0..32]
  = if x[31:16] <> zero(16) then lzcnt16(x[31:16])
    else 16 + lzcnt16(x[15:0])
function lzcnt16(x::bits 16)::[0..16]
  = if x[15:8] <> zero(8) then lzcnt08(x[15:8])
    else 8 + lzcnt08(x[7:0])
function lzcnt08(x::bits 8)::[0..8]
```

```
  = if x[7:4] <> zero(4) then lzcnt04(x[7:4])
    else 4 + lzcnt04(x[3:0])
function lzcnt04(x::bits 4)::[0..4]
  = if x[3] then 0
    elif x[2] then 1
    elif x[1] then 2
    elif x[0] then 3
    else 4
```

D.3 Bit String Conversions

Instruction BSWAP

Reverses the byte order of the specified register. This action converts the contents of the register from little endian to big endian or vice versa. In a doubleword, bits 7–0 are exchanged with bits 31–24, and bits 15–8 are exchanged with bits 23–16. In a quadword, bits 7–0 are exchanged with bits 63–56, bits 15–8 with bits 55–48, bits 23–16 with bits 47–40, and bits 31–24 with bits 39–32. A subsequent use of the BSWAP instruction with the same operand restores the original value of the operand. *[cited from AMD volume 3]*

```
opcode "0FC8h" reg $v : call op1' = BSWAP(op1)
action BSWAP(x::bits $v)::bits $v
  if $v == 16 then return $undefined[$v-1:0]
  elif $v == 32 then return x[7:0] ++ x[15:8] ++ x[23:16] ++ x[31:24]
  else let hi = x[7:0] ++ x[15:8] ++ x[23:16] ++ x[31:24]
       let lo = x[39:32] ++ x[47:40] ++ x[55:48] ++ x[63:56]
       return hi ++ lo
```

Instruction CBW-CWDE-CDQE

Copies the sign bit in the AL or eAX register to the upper bits of the rAX register. The effect of this instruction is to convert a signed byte, word, or doubleword in the AL or eAX register into a signed word, doubleword, or double quadword in the rAX register. This action helps avoid overflow problems in signed number arithmetic. *[cited from AMD volume 3]*

```
opcode "98h" : write gpr($v, sxt($v, RAX[$v/2-1:0]), RAX) to RAX
```

The `gpr` function is defined in section 14.13.

Instruction CWD-CDQ-CQO

Copies the sign bit in the rAX register to all bits of the rDX register. The effect of this instruction is to convert a signed word, doubleword, or quadword in the rAX register into a signed doubleword, quadword, or double-quadword in the rDX:rAX registers. This action helps avoid overflow problems in signed number arithmetic. *[cited from AMD volume 3]*

```
opcode "99h" : write gpr($v, sxt($v, RAX[$v-1]), RDX) to RDX
```

The `gpr` function is defined in section 14.13.

D.4 Shifts

Instruction SAL-SHL

Shifts the bits of a register or memory location (first operand) to the left through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. For each bit shift, the SAL instruction clears the least-significant bit to 0. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand. The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. The effect of this instruction is multiplication by powers of two. For 1-bit shifts, the instruction sets the OF flag to the exclusive OR of the CF bit (after the shift) and the most significant bit of the result. When the shift count is greater than 1, the OF flag is undefined. If the shift count is 0, no flags are modified. *[cited from AMD volume 3]*

```
opcode "D0h /100b" reg_mem 8, const_1 8
opcode "D2h /100b" reg_mem 8, rcx 8
opcode "C0h /100b" reg_mem 8, imm 8
opcode "D1h /100b" reg_mem $v, const_1 8
opcode "D3h /100b" reg_mem $v, rcx 8
opcode "C1h /100b" reg_mem $v, imm 8
call op1' = SHL($n, op1, op2)
```

The `SHL` action takes the operand width, the operand, and the shift amount. It computes the real shift amount as `$k` by masking the given shift amount. In case the `$k` is zero, the action does nothing. Otherwise, it computes the result and the status flags.

```
action SHL($n::{8, $v}, op1::bits $n, shift::bits 8)::bits $n
  let $k = if $n == 64 then nat(shift[5:0])::[0..63]
           else nat(shift[4:0])::[0..31]
  if $k == 0 then return op1
  else let res = op1 << $k
       write op1[$n - $k] to RFLAGS.CF when $k <= $n
       write 0b to RFLAGS.CF when $k > $n
       write parity($n, res) to RFLAGS.PF
       undef RFLAGS.AF
       write res == zero($n) to RFLAGS.ZF
       write res[$n - 1] to RFLAGS.SF
       write op1[$n - 1] ^ res[$n - 1] to RFLAGS.OF when $k == 1
       undef RFLAGS.OF when $k > 1
       return res
```

Instruction SHLD

Shifts the bits of a register or memory location (first operand) to the left by the number of bit positions in an unsigned immediate value or the CL register (third operand), and shifts in a bit pattern (second operand) from the right. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand. The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. If the masked count is greater than the operand size, the result in the destination register is undefined. If the shift

count is 0, no flags are modified. If the count is 1 and the sign of the operand being shifted changes, the instruction sets the OF flag to 1. If the count is greater than 1, OF is undefined. *[cited from AMD volume 3]*

```
opcode "0FA4h" reg_mem $v, reg $v, imm 8
opcode "0FA5h" reg_mem $v, reg $v, rcx 8
call op1' = SHLD($n, op1, op2, op3)
```

The `SHLD` action takes the operand width, the operand, the fill-in pattern, and the shift amount. It computes the real shift amount as `$k` by masking the given shift amount. In case the `$k` is zero, the action does nothing. In case the `$k` exceeds the operand width, the action returns an undefined result. Otherwise, it computes the result and the status flags.

```
action SHLD($n::{8, $v}, op1::bits $n, pattern::bits $n, shift::bits 8)::bits $n
  let $k = if $n == 64 then nat(shift[5:0])::[0..63]
           else nat(shift[4:0])::[0..31]
  if $k == 0 then return op1
  elif $k > $n then undef RFLAGS.CF
                    undef RFLAGS.PF
                    undef RFLAGS.AF
                    undef RFLAGS.ZF
                    undef RFLAGS.SF
                    undef RFLAGS.OF
                    return $undefined[$n-1:0]
  else let res = if $k == $n then pattern
                 else op1[$n-$k-1:0]++pattern[$k-1:0]
       write op1[$n - $k] to RFLAGS.CF
       write parity($n, res) to RFLAGS.PF
       undef RFLAGS.AF
       write res == zero($n) to RFLAGS.ZF
       write res[$n - 1] to RFLAGS.SF
       write op1[$n - 1] ^ res[$n - 1] to RFLAGS.OF when $k == 1
       undef RFLAGS.OF when $k > 1
       return res
```

Instruction SAR

Shifts the bits of a register or memory location (first operand) to the right through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand. The SAR instruction does not change the sign bit of the target operand. For each bit shift, it copies the sign bit to the next bit, preserving the sign of the result. The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. For 1-bit shifts, the instruction clears the OF flag to 0. When the shift count is greater than 1, the OF flag is undefined. If the shift count is 0, no flags are modified. Although the SAR instruction effectively divides the operand by a power of 2, the behavior is different from the IDIV instruction. For example, shifting –11 (FFFFFFF5h) by two bits to the right (that is, divide –11 by 4), gives a result of FFFFFFFDh, or –3, whereas the IDIV instruction for dividing –11 by 4 gives a result of –2. This is because the IDIV instruction rounds

off the quotient to zero, whereas the SAR instruction rounds off the remainder to zero for positive dividends and to negative infinity for negative dividends. So, for positive operands, SAR behaves like the corresponding IDIV instruction. For negative operands, it gives the same result if and only if all the shifted-out bits are zeroes; otherwise, the result is smaller by 1. *[cited from AMD volume 3]*

```
opcode "D0h /111b" reg_mem 8, const_1 8
opcode "D2h /111b" reg_mem 8, rcx 8
opcode "C0h /111b" reg_mem 8, imm 8
opcode "D1h /111b" reg_mem $v, const_1 8
opcode "D3h /111b" reg_mem $v, rcx 8
opcode "C1h /111b" reg_mem $v, imm 8
call op1' = SAHR($n, op1, sxt(64, op1[$n-1]), op2)
```

Since there are two right shifts: arithmetic shift, which fills in the sign-bit, and logic shift, which fills in zero, we define a general shift action `SAHR`. The action takes as parameters operand width, the operand, the fill-in pattern, and the shift amount. Notice that the fill-in pattern has either all bits cleared or all bits set. The action computes the real shift amount `$k` by masking the given shift amount. If the real shift is zero, the action does nothing. Otherwise, the action computes the result and the status flags.

```
action SAHR($n::{8, $v}, op1::bits $n, pattern::bits 64, shift::bits 8)::bits $n
  let $k = if $n == 64 then nat(shift[5:0])::[0..63]
           else nat(shift[4:0])::[0..31]
  if $k == 0 then return op1
  else let res = if $k >= $n then pattern[$n-1:0]
                 else pattern[$k-1:0] ++ op1[$n-1:$k]
       write op1[$k-1] to RFLAGS.CF when $k <= $n
       write pattern[0] to RFLAGS.CF when $k > $n
       write parity($n, res) to RFLAGS.PF
       undef RFLAGS.AF
       write res == zero($n) to RFLAGS.ZF
       write res[$n - 1] to RFLAGS.SF
       write 0b to RFLAGS.OF when $k == 1
       undef RFLAGS.OF when $k > 1
       return res
```

Instruction SHR

Shifts the bits of a register or memory location (first operand) to the right through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand. For each bit shift, the instruction clears the most-significant bit to 0. The effect of this instruction is unsigned division by powers of two. The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. For 8-bit shifts, the instruction sets the OF flag to the most-significant bit of the original value. If the count is greater than 1, the OF flag is undefined. If the shift count is 0, no flags are modified. *[cited from AMD volume 3]*

```
opcode "D0h /101b" reg_mem 8, const_1 8
opcode "D2h /101b" reg_mem 8, rcx 8
```

```
opcode "C0h /101b" reg_mem 8, imm 8
opcode "D1h /101b" reg_mem $v, const_1 8
opcode "D3h /101b" reg_mem $v, rcx 8
opcode "C1h /101b" reg_mem $v, imm 8
call op1' = SAHR($n, op1, zero(64), op2)
```

For the definition of the SAHR action refer to the description of the SAR instruction.

Instruction SHRD

Shifts the bits of a register or memory location (first operand) to the right by the number of bit positions in an unsigned immediate value or the CL register (third operand), and shifts in a bit pattern (second operand) from the left. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand. The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. If the masked count is greater than the operand size, the result in the destination register is undefined. If the shift count is 0, no flags are modified. If the count is 8 and the sign of the value being shifted changes, the instruction sets the OF flag to 1. If the count is greater than 1, the OF flag is undefined. *[cited from AMD volume 3]*

```
opcode "0FACh" reg_mem $v, reg $v, imm 8
opcode "0FADh" reg_mem $v, reg $v, rcx 8
call op1' = SHRD($n, op1, op2, op3)
```

The SHRD action takes the operand width, the operand, the fill-in pattern, and the shift amount. It is similar to the SAHR action that we defined for the SAR and the SHR instructions, but it returns undefined result if the real shift amount exceeds the operand width.

```
action SHRD($n::{8, $v}, op1::bits $n, pattern::bits $n, shift::bits 8)::bits $n
  let $k = if $n == 64 then nat(shift[5:0])::[0..63]
           else nat(shift[4:0])::[0..31]
  if $k == 0 then return op1
  elif $k > $n then undef RFLAGS.CF
                    undef RFLAGS.PF
                    undef RFLAGS.AF
                    undef RFLAGS.ZF
                    undef RFLAGS.SF
                    undef RFLAGS.OF
                    return $undefined[$n-1:0]
  else let res = if $k == $n then pattern
                 else pattern[$k-1:0] ++ op1[$n-1:$k]
       write op1[$k-1] to RFLAGS.CF
       write parity($n, res) to RFLAGS.PF
       undef RFLAGS.AF
       write res == zero($n) to RFLAGS.ZF
       write res[$n - 1] to RFLAGS.SF
       write 0b to RFLAGS.OF when $k == 1
       undef RFLAGS.OF when $k > 1
       return res
```

D.5 Rotations

Instruction ROL

Rotates the bits of a register or memory location (first operand) to the left (toward the more significant bit positions) by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated out left are rotated back in at the right end (lsb) of the first operand location. The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, it masks the upper two bits of the count, providing a count in the range of 0 to 63. After completing the rotation, the instruction sets the CF flag to the last bit rotated out (the lsb of the result). For 1-bit rotates, the instruction sets the OF flag to the exclusive OR of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected. *[cited from AMD volume 3]*

```
opcode "D0h /000b" reg_mem 8, const_1 8
opcode "D2h /000b" reg_mem 8, rcx 8
opcode "C0h /000b" reg_mem 8, imm 8
opcode "D1h /000b" reg_mem $v, const_1 8
opcode "D3h /000b" reg_mem $v, rcx 8
opcode "C1h /000b" reg_mem $v, imm 8
call op1' = ROL($n, op1, op2)
action ROL($n::{8, $v}, op1::bits $n, shift::bits 8)::bits $n
  let $k = if $n == 64 then nat(shift[5:0])::[0..63]
           else nat(shift[4:0])::[0..31]
  if $k == 0 then return op1
  else let $m = $k % $n
       let res = if $m == 0 then op1 else op1[$n-$m-1:0] ++ op1[$n-1:$n-$m]
       write res[0] to RFLAGS.CF
       write res[$n - 1] ^ op1[$n - 1] to RFLAGS.OF when $k == 1
       undef RFLAGS.OF when $k > 1
       return res
```

Instruction ROR

Rotates the bits of a register or memory location (first operand) to the right (toward the less significant bit positions) by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated out right are rotated back in at the left end (the most significant bit) of the first operand location. The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. After completing the rotation, the instruction sets the CF flag to the last bit rotated out (the most significant bit of the result). For 1-bit rotates, the instruction sets the OF flag to the exclusive OR of the two most significant bits of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected. *[cited from AMD volume 3]*

```
opcode "D0h /001b" reg_mem 8, const_1 8
opcode "D2h /001b" reg_mem 8, rcx 8
opcode "C0h /001b" reg_mem 8, imm 8
```

```
opcode "D1h /001b" reg_mem $v, const_1 8
opcode "D3h /001b" reg_mem $v, rcx 8
opcode "C1h /001b" reg_mem $v, imm 8
call op1' = ROR($n, op1, op2)
action ROR($n::{8, $v}, op1::bits $n, shift::bits 8)::bits $n
  let $k = if $n == 64 then nat(shift[5:0])::[0..63]
           else nat(shift[4:0])::[0..31]
  if $k == 0 then return op1
  else let $m = $k % $n
       let res = if $m == 0 then op1 else op1[$m-1:0] ++ op1[$n-1:$m]
       write res[$n-1] to RFLAGS.CF
       write res[$n-1] ^ op1[$n-1] to RFLAGS.OF when $k == 1
       undef RFLAGS.OF when $k > 1
       return res
```

Instruction RCL

Rotates the bits of a register or memory location (first operand) to the left (more significant bit positions) and through the carry flag by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated through the carry flag are rotated back in at the right end (lsb) of the first operand location. The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. For 1-bit rotates, the instruction sets the OF flag to the exclusive OR of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected. *[cited from AMD volume 3]*

```
opcode "D0h /010b" reg_mem 8, const_1 8
opcode "D2h /010b" reg_mem 8, rcx 8
opcode "C0h /010b" reg_mem 8, imm 8
opcode "D1h /010b" reg_mem $v, const_1 8
opcode "D3h /010b" reg_mem $v, rcx 8
opcode "C1h /010b" reg_mem $v, imm 8
call op1' = RCL($n, op1, op2)
action RCL($n::{8, $v}, op1::bits $n, shift::bits 8)::bits $n
  let $k = if $n == 64 then nat(shift[5:0])::[0..63]
           else nat(shift[4:0])::[0..31]
  if $k == 0 then return op1
  else let $m = $k % ($n+1)
       let x = RFLAGS.CF ++ op1
       let res = if $m == 0 then x else x[$n-$m:0] ++ x[$n:$n+1-$m]
       write res[$n] to RFLAGS.CF
       write res[$n - 1] ^ op1[$n - 1] to RFLAGS.OF when $k == 1
       undef RFLAGS.OF when $k > 1
       return res[$n-1:0]
```

Instruction RCR

Rotates the bits of a register or memory location (first operand) to the right (toward the less significant bit positions) and through the carry flag by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated through the carry flag are rotated back in at the left end (msb) of the first operand lo-

cation. The processor masks the upper three bits in the count operand, thus restricting the count to a number between 0 and 31. When the destination is 64 bits wide, the processor masks the upper two bits of the count, providing a count in the range of 0 to 63. For 1-bit rotates, the instruction sets the OF flag to the exclusive OR of the CF flag (before the rotate) and the most significant bit of the original value. When the rotate count is greater than 1, the OF flag is undefined. When the rotate count is 0, no flags are affected. *[cited from AMD volume 3]*

```
opcode "D0h /011b" reg_mem 8, const_1 8
opcode "D2h /011b" reg_mem 8, rcx 8
opcode "C0h /011b" reg_mem 8, imm 8
opcode "D1h /011b" reg_mem $v, const_1 8
opcode "D3h /011b" reg_mem $v, rcx 8
opcode "C1h /011b" reg_mem $v, imm 8
call op1' = RCR($n, op1, op2)
action RCR($n::{8, $v}, op1::bits $n, shift::bits 8)::bits $n
  let $k = if $n == 64 then nat(shift[5:0])::[0..63]
           else nat(shift[4:0])::[0..31]
  if $k == 0 then return op1
  else let $m = $k % ($n+1)
       let x = RFLAGS.CF ++ op1
       let res = if $m == 0 then x else x[$m-1:0] ++ x[$n:$m]
       write res[$n] to RFLAGS.CF
       write res[$n-1] ^ op1[$n-1] to RFLAGS.OF when $k == 1
       undef RFLAGS.OF when $k > 1
       return res[$n-1:0]
```

APPENDIX

E

INSTRUCTIONS FOR BINARY CODED DECIMALS

Instruction AAA (**when** not $x64_mode)

Adjusts the value in the AL register to an unpacked BCD value. Use the AAA instruction after using the ADD instruction to add two unpacked BCD numbers. If the value in the lower nibble of AL is greater than 9 or the AF flag is set to 1, the instruction increments the AH register, adds 6 to the AL register, and sets the CF and AF flags to 1. Otherwise, it does not change the AH register and clears the CF and AF flags to 0. In either case, AAA clears bits 7–4 of the AL register, leaving the correct decimal digit in bits 3–0. *[cited from AMD volume 3]*

```
opcode "37h" : call AAA
action AAA
  if RAX[3:0] > 9h or RFLAGS.AF == 1b then
     write RAX[15:8] + 01h to RAX[15:8]
     write (RAX[7:0] + 06h) & 0Fh to RAX[7:0]
     write 1b to RFLAGS.CF
     write 1b to RFLAGS.AF
  else write RAX[7:0] & 0Fh to RAX[7:0]
       write 0b to RFLAGS.CF
       write 0b to RFLAGS.AF
```

Instruction AAS (**when** not $x64_mode)

Adjusts the value in the AL register to an unpacked BCD value. Use the AAS instruction after using the SUB instruction to subtract two unpacked BCD numbers. If the value in AL is greater than 9 or the AF flag is set to 1, the instruction decrements the value in AH, subtracts 6 from the AL register, and sets the CF and AF flags to 1. Otherwise, it clears the CF and AF flags and the AH register is unchanged. In either case, the instruction clears bits 7–4 of the AL register, leaving the correct decimal digit in bits 3–0. *[cited from AMD volume 3]*

```
opcode "3Fh" : call AAS
action AAS
```

```
  if RAX[3:0] > 9h or RFLAGS.AF == 1b then
     write RAX[15:8] - 01h to RAX[15:8]
     write (RAX[7:0] - 06h) & 0Fh to RAX[7:0]
     write 1b to RFLAGS.CF
     write 1b to RFLAGS.AF
  else write RAX[7:0] & 0Fh to RAX[7:0]
       write 0b to RFLAGS.CF
       write 0b to RFLAGS.AF
```

Instruction AAD (`when not $x64_mode`)

Converts two unpacked BCD digits in the AL (least significant) and AH (most significant) registers to a single binary value in the AL register using the following formula: AL = ((10d * AH) + (AL)) After the conversion, AH is cleared to 00h. In most modern assemblers, the AAD instruction adjusts from base–10 values. However, by coding the instruction directly in binary, it can adjust from any base specified by the immediate byte value (ib) suffixed onto the D5h opcode. For example, code D508h for octal, D50Ah for decimal, and D50Ch for duodecimal (base 12). *[cited from AMD volume 3]*

```
opcode "D5h" imm 8 : call AAD(op1)
action AAD(base::bits 8)
  let res = RAX[15:8] * base + RAX[7:0]
  write zxt(16, res) to RAX[15:0]
  write parity(8, res) to RFLAGS.PF
  write res == 00h to RFLAGS.ZF
  write res[7] to RFLAGS.SF
  undef RFLAGS.CF
  undef RFLAGS.AF
  undef RFLAGS.OF
```

Instruction AAM (`when not $x64_mode`)

Converts the value in the AL register from binary to two unpacked BCD digits in the AH (most significant) and AL (least significant) registers using the following formula: AH = (AL/10d) AL = (AL mod 10d) In most modern assemblers, the AAM instruction adjusts to base–10 values. However, by coding the instruction directly in binary, it can adjust to any base specified by the immediate byte value (ib) suffixed onto the D4h opcode. For example, code D408h for octal, D40Ah for decimal, and D40Ch for duodecimal (base 12). *[cited from AMD volume 3]*

```
opcode "D4h" imm 8 : call AAM(op1)
action AAM(base::bits 8)
  let high = RAX[7:0] / base
  let low = RAX[7:0] % base
  let res = high ++ low
  write res to RAX[15:0]
  write parity(16, res) to RFLAGS.PF
  write res == 0000h to RFLAGS.ZF
  write res[15] to RFLAGS.SF
  undef RFLAGS.CF
  undef RFLAGS.AF
  undef RFLAGS.OF
```

Instruction DAA (`when not $x64_mode`)

Adjusts the value in the AL register into a packed BCD result and sets the CF and AF flags in the rFLAGS register to indicate a decimal carry out of either nibble of AL. Use this instruction to adjust the result of a byte ADD instruction that performed the binary addition of one 2-digit packed BCD values to another. The instruction performs the adjustment by adding 06h to AL if the lower nibble is greater than 9 or if AF = 1. Then 60h is added to AL if the original AL was greater than 99h or if CF = 1. If the lower nibble of AL was adjusted, the AF flag is set to 1. Otherwise AF is not modified. If the upper nibble of AL was adjusted, the CF flag is set to 1. Otherwise, CF is not modified. SF, ZF, and PF are set according to the final value of AL. *[cited from AMD volume 3]*

```
opcode "27h" : call DAA
action DAA
  let al1 = if RAX[3:0] > 9h or RFLAGS.AF then RAX[7:0] + 06h else RAX[7:0]
  let al2 = if RAX[7:0] > 99h or RFLAGS.CF then al1 + 60h else al1
  write 1b to RFLAGS.AF when RAX[3:0] > 9h or RFLAGS.AF
  write 1b to RFLAGS.CF when RAX[7:0] > 99h or RFLAGS.CF
  write al2 to RAX[7:0]
  write parity(8, al2) to RFLAGS.PF
  write al2 == 00h to RFLAGS.ZF
  write al2[7] to RFLAGS.SF
```

Instruction DAS (`when not $x64_mode`)

Adjusts the value in the AL register into a packed BCD result and sets the CF and AF flags in the rFLAGS register to indicate a decimal borrow. Use this instruction to adjust the result of a byte SUB instruction that performed a binary subtraction of one 2-digit, packed BCD value from another. This instruction performs the adjustment by subtracting 06h from AL if the lower nibble is greater than 9 or if AF = 1. Then 60h is subtracted from AL if the original AL was greater than 99h or if CF = 1. If the adjustment changes the lower nibble of AL, the AF flag is set to 1; otherwise AF is not modified. If the adjustment results in a borrow for either nibble of AL, the CF flag is set to 1; otherwise CF is not modified. The SF, ZF, and PF flags are set according to the final value of AL. *[cited from AMD volume 3]*

```
opcode "2Fh" : call DAS
action DAS
  let al1 = if RAX[3:0] > 9h or RFLAGS.AF then RAX[7:0] - 06h else RAX[7:0]
  let al2 = if RAX[7:0] > 99h or RFLAGS.CF then al1 - 60h else al1
  write 1b to RFLAGS.AF when RAX[3:0] > 9h or RFLAGS.AF
  write 1b to RFLAGS.CF when RAX[7:0] > 99h or RFLAGS.CF
  write al2 to RAX[7:0]
  write parity(8, al2) to RFLAGS.PF
  write al2 == 00h to RFLAGS.ZF
  write al2[7] to RFLAGS.SF
```

APPENDIX

F

FLAG INSTRUCTIONS

Instruction STI

Sets the interrupt flag (IF) in the rFLAGS register to 1, thereby allowing external interrupts received on the INTR input. Interrupts received on the non-maskable interrupt (NMI) input are not affected by this instruction. In real mode, this instruction sets IF to 1. In protected mode and virtual–8086-mode, this instruction is IOPL-sensitive. If the CPL is less than or equal to the rFLAGS.IOPL field, the instruction sets IF to 1. In protected mode, if IOPL < 3, CPL = 3, and protected mode virtual interrupts are enabled (CR4.PVI = 1), then the instruction instead sets rFLAGS.VIF to 1. If none of these conditions apply, the processor raises a general protection exception (#GP). For more information, see “Protected Mode Virtual Interrupts” in Volume 2. In virtual–8086 mode, if IOPL < 3 and the virtual–8086-mode extensions are enabled (CR4.VME = 1), the STI instruction instead sets the virtual interrupt flag (rFLAGS.VIF) to 1. If STI sets the IF flag and IF was initially clear, then interrupts are not enabled until after the instruction following STI. *[cited from AMD volume 3]*

```
opcode "FBh" : call STI
action STI
  if CPL <= RFLAGS.IOPL then
    write 1b to _intr_shadow when RFLAGS.IF == 0b
    write 0b to RFLAGS.IF
  elif CPL == 11b and RFLAGS.IOPL < 11b
          and ($vm86_mode and CR4.VME or $protected_mode and CR4.PVI) then
    write 0b to RFLAGS.VIF
  else fail exception(xGP, 0000h)
```

Instruction CLI

Clears the interrupt flag (IF) in the rFLAGS register to zero, thereby masking external interrupts received on the INTR input. Interrupts received on the non-maskable interrupt (NMI) input are not affected by this instruction. In real mode, this instruction clears IF to 0. In protected mode and virtual–8086-mode, this instruction is IOPL-sensitive. If the CPL is less than or equal to the rFLAGS.IOPL field, the instruction clears IF to 0. In protected mode, if IOPL < 3, CPL = 3, and protected mode virtual

interrupts are enabled (CR4.PVI = 1), then the instruction instead clears rFLAGS.VIF to 0. If none of these conditions apply, the processor raises a general-purpose exception (#GP). In virtual–8086 mode, if IOPL < 3 and the virtual–8086-mode extensions are enabled (CR4.VME = 1), the CLI instruction clears the virtual interrupt flag (rFLAGS.VIF) to 0 instead. *[cited from AMD volume 3]*

```
opcode "FAh" : call CLI
action CLI
  if CPL <= RFLAGS.IOPL then write 0b to RFLAGS.IF
  elif CPL == 11b and RFLAGS.IOPL < 11b
       and ($vm86_mode and CR4.VME or $protected_mode and CR4.PVI) then
    write 0b to RFLAGS.VIF
  else fail exception(xGP, 0000h)
```

Instruction STC

Sets the carry flag (CF) in the rFLAGS register to one. *[cited from AMD volume 3]*

```
opcode "F9h" : write 1b to RFLAGS.CF
```

Instruction CLC

Clears the carry flag (CF) in the rFLAGS register to zero. *[cited from AMD volume 3]*

```
opcode "F8h" : write 0b to RFLAGS.CF
```

Instruction STD

Set the direction flag (DF) in the rFLAGS register to 1. If the DF flag is 0, each iteration of a string instruction increments the data pointer (index registers rSI or rDI). If the DF flag is 1, the string instruction decrements the pointer. Use the CLD instruction before a string instruction to make the data pointer increment. *[cited from AMD volume 3]*

```
opcode "FDh" : write 1b to RFLAGS.DF
```

Instruction CLD

Clears the direction flag (DF) in the rFLAGS register to zero. If the DF flag is 0, each iteration of a string instruction increments the data pointer (index registers rSI or rDI). If the DF flag is 1, the string instruction decrements the pointer. Use the CLD instruction before a string instruction to make the data pointer increment. *[cited from AMD volume 3]*

```
opcode "FCh" : write 0b to RFLAGS.DF
```

Instruction CMC

Complements (toggles) the carry flag (CF) bit of the rFLAGS register. *[cited from AMD volume 3]*

```
opcode "F5h" : write (not RFLAGS.CF) to RFLAGS.CF
```

Instruction LAHF (`when not $x64_mode or $CPUID_80000001_LAHF_SAHF`)

Loads the lower 8 bits of the rFLAGS register, including sign flag (SF), zero flag (ZF), auxiliary carry flag (AF), parity flag (PF), and carry flag (CF), into the AH register. The instruction sets the reserved bits 1, 3, and 5 of the rFLAGS register to 1, 0, and 0, respectively, in the AH register. The LAHF instruction can only be executed in 64-bit mode if supported by the processor implementation. Check the status of ECX bit 0

returned by CPUID function 8000_0001h to verify that the processor supports LAHF in 64-bit mode. *[cited from AMD volume 3]*

```
opcode "9Fh" : call LAHF
action LAHF
  let x = fixFlags(RFLAGS, RFLAGS)
  write x[7:0] to RAX[15:8]
```

Instruction SAHF (**when** not $x64_mode **or** $CPUID_80000001_LAHF_SAHF)

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). The instruction ignores bits 1, 3, and 5 of register AH; it sets those bits in the EFLAGS register to 1, 0, and 0, respectively. The SAHF instruction can only be executed in 64-bit mode if supported by the processor implementation. Check the status of ECX bit 0 returned by CPUID function 8000_0001h to verify that the processor supports SAHF in 64-bit mode. *[cited from AMD volume 3]*

```
opcode "9Eh" : call SAHF
action SAHF
  let x = fixFlags(RFLAGS[63:8] ++ RAX[15:8], RFLAGS)
  write x[7:0] to RFLAGS[7:0]
```

APPENDIX

G

STACK INSTRUCTIONS

Instruction PUSH

Decrements the stack pointer and then copies the specified immediate value or the value in the specified register or memory location to the top of the stack (the memory location pointed to by SS:rSP). The operand-size attribute determines the number of bytes pushed to the stack. The stack-size attribute determines whether SP, ESP, or RSP is the stack pointer. The address-size attribute is used only to locate the memory operand when pushing a memory operand to the stack. If the instruction pushes the stack pointer (rSP), the resulting value on the stack is that of rSP before execution of the instruction. *[cited from AMD volume 3]*

```
opcode "FFh /110b" reg_mem $v : call PUSH(op1)
opcode "50h" reg $v           : call PUSH(op1)
opcode "6Ah" imm 8            : call PUSH(sxt($v, op1))
opcode "68h" imm $z           : call PUSH(sxt($v, op1))
opcode "0Eh" cs 16            : call PUSH(zxt($v, CS.sel))
                              : when not $x64_mode
opcode "16h" ss 16            : call PUSH(zxt($v, SS.sel))
                              : when not $x64_mode
opcode "1Eh" ds 16            : call PUSH(zxt($v, DS.sel))
                              : when not $x64_mode
opcode "06h" es 16            : call PUSH(zxt($v, ES.sel))
                              : when not $x64_mode
opcode "0FA0h" fs 16          : call PUSH(zxt($v, FS.sel))
opcode "0FA8h" gs 16          : call PUSH(zxt($v, GS.sel))
```

The `PUSH` action uses the `push` action from chapter 15.

```
action PUSH(val::bits $v)
  call rsp_new = push($v, val, RSP[$sa-1:0])
  write gpr($sa, rsp_new, RSP) to RSP
```

Instruction POP

Copies the value pointed to by the stack pointer (SS:rSP) to the specified register or memory location and then increments the rSP by 2 for a 16-bit pop, 4 for a 32-bit

pop, or 8 for a 64-bit pop. The operand-size attribute determines the amount by which the stack pointer is incremented (2, 4 or 8 bytes). The stack-size attribute determines whether SP, ESP, or RSP is incremented. It is possible to pop a null segment selector value (0000–0003h) into the DS, ES, FS, or GS register. This action does not cause a general protection fault, but a subsequent reference to such a segment does cause a #GP exception. For more information about segment selectors, see “Segment Selectors and Registers” on page 67. In 64-bit mode, the POP operand size defaults to 64 bits and there is no prefix available to encode a 32- bit operand size. Using POP DS, POP ES, or POP SS instruction in 64-bit mode generates an invalid- opcode exception. *[cited from AMD volume 3]*

```
opcode "8Fh /000b" reg_mem $v : call op1’ = POP($v)
opcode "58h" reg $v           : call op1’ = POP($v)
opcode "17h" ss 16            : call op1’ = POP(16)
                              : when not $x64_mode
opcode "1Fh" ds 16            : call op1’ = POP(16)
                              : when not $x64_mode
opcode "07h" es 16            : call op1’ = POP(16)
                              : when not $x64_mode
opcode "0FA1h" fs 16          : call op1’ = POP(16)
opcode "0FA9h" gs 16          : call op1’ = POP(16)
```

The `POP` action uses the `pop` action from chapter 15.

```
action POP($n::{$v, 16})::bits $n
  call (val, new_rsp) = pop($v, RSP[$sa-1:0])
  write gpr($sa, new_rsp, RSP) to RSP
  return val[$n-1:0]
```

Instruction POPA (`when not $x64_mode`)

Pops words or doublewords from the stack into the general-purpose registers in the following order: eDI, eSI, eBP, eSP (image is popped and discarded), eBX, eDX, eCX, and eAX. The instruction increments the stack pointer by 16 or 32, depending on the operand size *[cited from AMD volume 3]*

```
opcode "61h" : call POPA
action POPA
  call (rdi, rsp1) = pop($v, RSP[$sa-1:0])
  call (rsi, rsp2) = pop($v, rsp1)
  call (rbp, rsp3) = pop($v, rsp2)
  call (rsp, rsp4) = pop($v, rsp3)
  call (rbx, rsp5) = pop($v, rsp4)
  call (rdx, rsp6) = pop($v, rsp5)
  call (rcx, rsp7) = pop($v, rsp6)
  call (rax, rsp8) = pop($v, rsp7)
  write gpr($v, rdi, RDI) to RDI
  write gpr($v, rsi, RSI) to RSI
  write gpr($v, rbp, RBP) to RBP
  write gpr($sa, rsp8, RSP) to RSP
  write gpr($v, rbx, RBX) to RBX
  write gpr($v, rdx, RDX) to RDX
  write gpr($v, rcx, RCX) to RCX
  write gpr($v, rax, RAX) to RAX
```

Instruction PUSHA (`when not $x64_mode`)

Pushes the contents of the eAX, eCX, eDX, eBX, eSP (original value), eBP, eSI, and eDI general- purpose registers onto the stack in that order. This instruction decrements the stack pointer by 16 or 32 depending on operand size. *[cited from AMD volume 3]*

```
opcode "60h" : call PUSHA
action PUSHA
  call rsp1 = push($v, RAX[$v-1:0], RSP[$sa-1:0])
  call rsp2 = push($v, RCX[$v-1:0], rsp1)
  call rsp3 = push($v, RDX[$v-1:0], rsp2)
  call rsp4 = push($v, RBX[$v-1:0], rsp3)
  call rsp5 = push($v, RSP[$v-1:0], rsp4)
  call rsp6 = push($v, RBP[$v-1:0], rsp5)
  call rsp7 = push($v, RSI[$v-1:0], rsp6)
  call rsp8 = push($v, RDI[$v-1:0], rsp7)
  write gpr($sa, rsp8, RSP) to RSP
```

Instruction PUSHF

Decrements the rSP register and copies the rFLAGS register (except for the VM and RF flags) onto the stack. The instruction clears the VM and RF flags in the rFLAGS image before putting it on the stack. The instruction pushes 2, 4, or 8 bytes, depending on the operand size. In 64-bit mode, this instruction defaults to a 64-bit operand size and there is no prefix available to encode a 32-bit operand size. In virtual–8086 mode, if system software has set the IOPL field to a value less than 3, a general- protection exception occurs if application software attempts to execute PUSHFx or POPFx while VME is not enabled or the operand size is not 16-bit. *[cited from AMD volume 3]*

```
opcode "9Ch" : call PUSHF
action PUSHF
  fail intercept(VMEXIT_PUSHF) when _vmcb_ca.intercept.PUSHF
  let flags = RFLAGS with [RF = 0b, VM = 0b]
  call new_rsp = push($v, flags[$v-1:0], RSP[$sa-1:0])
  write gpr($sa, new_rsp, RSP) to RSP
```

Instruction POPF

Pops a word, doubleword, or quadword from the stack into the rFLAGS register and then increments the stack pointer by 2, 4, or 8, depending on the operand size. In protected or real mode, all the non-reserved flags in the rFLAGS register can be modified, except the VIP, VIF, and VM flags, which are unchanged. In protected mode, at a privilege level greater than 0 the IOPL is also unchanged. The instruction alters the interrupt flag (IF) only when the CPL is less than or equal to the IOPL. In virtual–8086 mode, if IOPL field is less than 3, attempting to execute a POPFx or PUSHFx instruction while VME is not enabled, or the operand size is not 16-bit, generates a #GP exception. *[cited from AMD volume 3]*

```
opcode "9Dh" : call POPF
action POPF
  fail intercept(VMEXIT_POPF) when _vmcb_ca.intercept.POPF
  if $vm86_mode and RFLAGS.IOPL < 11b then
    fail exception(xGP, 0000h) when not CR4.VME or $v <> 16
  call (flags, new_rsp) = pop($v, RSP[$sa-1:0])
  let adjusted_flags = adjust_flags_POPF(RFLAGS[63:$v] ++ flags)
```

```
  write adjusted_flags[$v-1:0] to RFLAGS[$v-1:0]
  write gpr($sa, new_rsp, RSP) to RSP
```

Adjusting flags for the `POPF` instruction means retaining the old values of the `VIP`, `VIF`, and `VM` flags and adjusting the `IF` flag and the `IOPL` I/O privilege level using the `adjust_IF_IOPL` function from section 16.4.

```
function adjust_flags_POPF(f::Flags)::Flags
  = let f1 = fixFlags(f, RFLAGS) in
    let f2 = f1 with [VIP = RFLAGS.VIP, VIF = RFLAGS.VIF, VM = RFLAGS.VM] in
    adjust_IF_IOPL(f2)
```

Instruction ENTER

Creates a stack frame for a procedure. The first operand specifies the size of the stack frame allocated by the instruction. The second operand specifies the nesting level (0 to 31—the value is automatically masked to 5 bits). For nesting levels of 1 or greater, the processor copies earlier stack frame pointers before adjusting the stack pointer. This action provides a called procedure with access points to other nested stack frames. The 32-bit enter N, 0 (a nesting level of 0) instruction is equivalent to the following 32-bit instruction sequence: push ebp ; save current EBP mov ebp, esp ; set stack frame pointer value sub esp, N ; allocate space for local variables The ENTER and LEAVE instructions provide support for block structured languages. The LEAVE instruction releases the stack frame on returning from a procedure. In 64-bit mode, the operand size of ENTER defaults to 64 bits, and there is no prefix available for encoding a 32-bit operand size. *[cited from AMD volume 3]*

```
opcode "C8h" imm 16, imm2 8 : call ENTER(op1, op2)
action ENTER(space::bits 16, level::bits 8)
  let k = level[4:0]
  call rsp1 = push($v, RBP[$v-1:0], RSP[$sa-1:0])
  let rbp = gpr($sa, rsp1, RSP)
  call rsp2 = copy_stack_pointers(k, RBP[$sa-1:0], rbp)
  let rsp3 = rsp2 - zxt($sa, space)
  call check_logical(write, stack, $v, SS, $sa, rsp3)
  write gpr($sa, rsp3, RSP) to RSP
  write rbp to RBP
```

The most involved part of the `ENTER` instruction is copying the stack pointers. We use the `copy_stack_pointer` action, which takes the number of stack pointers to copy, the old base pointer `rbp` and the new stack pointer `rsp`. Let `k` be the number of paramters, then the action is equivalent to the following pseudocode:

```
for i from 1 to 31 do
  if i < k then
    x = read logical memory at rbp - i*($v/8)
    push x
```

As our specification language does not have loops, we express the loop body as a seperate action `copy_stack_pointer` and unroll the loop in the `copy_stack_pointers` action.

The `copy_stack_pointer` action takes `(k, rpb, i, rsp)`, copies the `i`-th stack pointer, and returns the new values for the `(i, rsp)`.

```
action copy_stack_pointer(k::bits 5, rbp::bits $sa,
```

```
                          i::bits 5, rsp ::bits $sa)::(bits 5, bits $sa)
  if i >= k then return (i, rsp)
  else call x = lread(stack, $v, SS, $sa, rbp - bits($sa, nat(i) * $v/8))
       call new_rsp = push($v, x, rsp)
       return (i+00001b, new_rsp)
```

The `copy_stack_pointers` action simply calls the `copy_stack_pointer` action 31 times.

```
action copy_stack_pointers(k::bits 5, rbp::bits $sa, rsp::bits 64)::bits $sa
  if k == 00000b then return rsp[$sa-1:0]
  else let i1 = 00001b
       let rsp1 = rsp[$sa-1:0]
       call (i2, rsp2) = copy_stack_pointer(k, rbp, i1, rsp1)
       call (i3, rsp3) = copy_stack_pointer(k, rbp, i2, rsp2)
       call (i4, rsp4) = copy_stack_pointer(k, rbp, i3, rsp3)
       call (i5, rsp5) = copy_stack_pointer(k, rbp, i4, rsp4)
       call (i6, rsp6) = copy_stack_pointer(k, rbp, i5, rsp5)
       call (i7, rsp7) = copy_stack_pointer(k, rbp, i6, rsp6)
       call (i8, rsp8) = copy_stack_pointer(k, rbp, i7, rsp7)
       call (i9, rsp9) = copy_stack_pointer(k, rbp, i8, rsp8)
       call (i10, rsp10) = copy_stack_pointer(k, rbp, i9, rsp9)
       call (i11, rsp11) = copy_stack_pointer(k, rbp, i10, rsp10)
       call (i12, rsp12) = copy_stack_pointer(k, rbp, i11, rsp11)
       call (i13, rsp13) = copy_stack_pointer(k, rbp, i12, rsp12)
       call (i14, rsp14) = copy_stack_pointer(k, rbp, i13, rsp13)
       call (i15, rsp15) = copy_stack_pointer(k, rbp, i14, rsp14)
       call (i16, rsp16) = copy_stack_pointer(k, rbp, i15, rsp15)
       call (i17, rsp17) = copy_stack_pointer(k, rbp, i16, rsp16)
       call (i18, rsp18) = copy_stack_pointer(k, rbp, i17, rsp17)
       call (i19, rsp19) = copy_stack_pointer(k, rbp, i18, rsp18)
       call (i20, rsp20) = copy_stack_pointer(k, rbp, i19, rsp19)
       call (i21, rsp21) = copy_stack_pointer(k, rbp, i20, rsp20)
       call (i22, rsp22) = copy_stack_pointer(k, rbp, i21, rsp21)
       call (i23, rsp23) = copy_stack_pointer(k, rbp, i22, rsp22)
       call (i24, rsp24) = copy_stack_pointer(k, rbp, i23, rsp23)
       call (i25, rsp25) = copy_stack_pointer(k, rbp, i24, rsp24)
       call (i26, rsp26) = copy_stack_pointer(k, rbp, i25, rsp25)
       call (i27, rsp27) = copy_stack_pointer(k, rbp, i26, rsp26)
       call (i28, rsp28) = copy_stack_pointer(k, rbp, i27, rsp27)
       call (i29, rsp29) = copy_stack_pointer(k, rbp, i28, rsp28)
       call (i30, rsp30) = copy_stack_pointer(k, rbp, i29, rsp29)
       call (i31, rsp31) = copy_stack_pointer(k, rbp, i30, rsp30)
       call rsp32 = push($v, rsp[$v-1:0], rsp31)
       return rsp32
```

Instruction LEAVE

Releases a stack frame created by a previous ENTER instruction. To release the frame, it copies the frame pointer (in the rBP register) to the stack pointer register (rSP), and then pops the old frame pointer from the stack into the rBP register, thus restoring the stack frame of the calling procedure. The 32-bit LEAVE instruction is equivalent to the following 32-bit operation: MOV ESP,EBP POP EBP *[cited from AMD volume 3]*

```
opcode "C9h" : call LEAVE
action LEAVE
```

```
let rsp = gpr($v, RBP[$v-1:0], RSP)
call (rbp, new_rsp) = pop($v, rsp[$sa-1:0])
write gpr($v, rbp, RBP) to RBP
write gpr($sa, new_rsp, rsp) to RSP
```

APPENDIX

H

NEAR CONTROL TRANSFER INSTRUCTIONS

Instruction JMP (Near)

Unconditionally transfers control to a new address without saving the current rIP value. This form of the instruction jumps to an address in the current code segment and is called a near jump. The target operand can specify a register, a memory location, or a label. If the JMP target is specified in a register or memory location, then a 16-, 32-, or 64-bit rIP is read from the operand, depending on operand size. This rIP is zero-extended to 64 bits. If the JMP target is specified by a displacement in the instruction, the signed displacement is added to the rIP (of the following instruction), and the result is truncated to 16, 32, or 64 bits depending on operand size. The signed displacement can be 8 bits, 16 bits, or 32 bits, depending on the opcode and the operand size. For near jumps in 64-bit mode, the operand size defaults to 64 bits. The E9 opcode results in RIP = RIP + 32-bit signed displacement, and the FF /4 opcode results in RIP = 64-bit offset from register or memory. No prefix is available to encode a 32-bit operand size in 64-bit mode. *[cited from AMD volume 3]*

```
opcode "EBh" imm 8            : call JMP(RIP[$v-1:0]+sxt($v, op1))
opcode "E9h" imm $z           : call JMP(RIP[$v-1:0]+sxt($v, op1))
opcode "FFh /100b" reg_mem $v : call JMP(op1)
action JMP(rip::bits $v)
  fail exception(xGP, 0000h) when not $x64_mode and nat(rip) > nat(CS.limit)
  fail exception(xGP, 0000h) when $x64_mode and not canonical(zxt(64, rip))
  write zxt(64, rip) to RIP
```

Instruction JCXZ

Checks the contents of the count register (rCX) and, if 0, jumps to the target instruction located at the specified 8-bit relative offset. Otherwise, execution continues with the instruction following the JrCXZ instruction. The size of the count register (CX, ECX, or rcx) depends on the address-size attribute of the JrCXZ instruction. Therefore, JrcxZ can only be executed in 64-bit mode and JCXZ cannot be executed in 64-bit mode. If the

jump is taken, the signed displacement is added to the rIP (of the following instruction) and the result is truncated to 16, 32, or 64 bits, depending on operand size. *[cited from AMD volume 3]*

```
opcode "E3h" imm 8 : call JCXZ(RIP[$v-1:0]+sxt($v, op1))
action JCXZ(rip::bits $v)
  if RCX[$oa-1:0] == zero($oa) then
    call JMP(rip)
```

Instruction Jcc

Checks the status flags in the rFLAGS register and, if the flags meet the condition specified by the condition code in the mnemonic (cc), jumps to the target instruction located at the specified relative offset. Otherwise, execution continues with the instruction following the Jcc instruction. Unlike the unconditional jump (JMP), conditional jump instructions have only two forms—short and near conditional jumps. Different opcodes correspond to different forms of one instruction. For example, the JO instruction (jump if overflow) has opcode 0Fh 80h for its near form and 70h for its short form, but the mnemonic is the same for both forms. The only difference is that the near form has a 16- or 32-bit relative displacement, while the short form always has an 8-bit relative displacement. Mnemonics are provided to deal with the programming semantics of both signed and unsigned numbers. Instructions tagged A (above) and B (below) are intended for use in unsigned integer code; those tagged G (greater) and L (less) are intended for use in signed integer code. If the jump is taken, the signed displacement is added to the rIP (of the following instruction) and the result is truncated to 16, 32, or 64 bits, depending on operand size. In 64-bit mode, the operand size defaults to 64 bits. The processor sign-extends the 8-bit or 32-bit displacement value to 64 bits before adding it to the RIP. *[cited from AMD volume 3]*

```
opcode "70h" imm 8
opcode "0F80h" imm $z
opcode "71h" imm 8
opcode "0F81h" imm $z
opcode "72h" imm 8
opcode "0F82h" imm $z
opcode "73h" imm 8
opcode "0F83h" imm $z
opcode "74h" imm 8
opcode "0F84h" imm $z
opcode "75h" imm 8
opcode "0F85h" imm $z
opcode "76h" imm 8
opcode "0F86h" imm $z
opcode "77h" imm 8
opcode "0F87h" imm $z
opcode "78h" imm 8
opcode "0F88h" imm $z
opcode "79h" imm 8
opcode "0F89h" imm $z
opcode "7Ah" imm 8
opcode "0F8Ah" imm $z
opcode "7Bh" imm 8
opcode "0F8Bh" imm $z
opcode "7Ch" imm 8
```

```
opcode "0F8Ch" imm $z
opcode "7Dh" imm 8
opcode "0F8Dh" imm $z
opcode "7Eh" imm 8
opcode "0F8Eh" imm $z
opcode "7Fh" imm 8
opcode "0F8Fh" imm $z
call Jcc(RIP[$v-1:0]+sxt($v, op1))
action Jcc(rip::bits $v)
  if cc(Opcode[3:0]) then
    call JMP(rip)
```

The `cc` function is defined in chapter A, and it maps the lower 4 bits of the opcode to a condition on flags.

Instruction LOOPcc

Decrements the count register (rCX) by 1, then, if rCX is not 0 and the ZF flag meets the condition specified by the mnemonic, it jumps to the target instruction specified by the signed 8-bit relative offset. Otherwise, it continues with the next instruction after the LOOPcc instruction. The size of the count register used (CX, ECX, or rcx) depends on the address-size attribute of the LOOPcc instruction. The LOOP instruction ignores the state of the ZF flag. The LOOPE and LOOPZ instructions jump if rCX is not 0 and the ZF flag is set to 1. In other words, the instruction exits the loop (falls through to the next instruction) if rCX becomes 0 or ZF = 0. The LOOPNE and LOOPNZ instructions jump if rCX is not 0 and ZF flag is cleared to 0. In other words, the instruction exits the loop if rCX becomes 0 or ZF = 1. The LOOPcc instruction does not change the state of the ZF flag. Typically, the loop contains a compare instruction to set or clear the ZF flag. If the jump is taken, the signed displacement is added to the rIP (of the following instruction) and the result is truncated to 16, 32, or 64 bits, depending on operand size. In 64-bit mode, the operand size defaults to 64 bits without the need for a REX prefix, and the processor sign-extends the 8-bit offset before adding it to the RIP. *[cited from AMD volume 3]*

```
opcode "E0h" imm 8
opcode "E1h" imm 8
opcode "E2h" imm 8
call LOOPcc(RIP[$v-1:0]+sxt($v, op1))
action LOOPcc(rip::bits $v)
  let rcx = RCX[$oa-1:0] - one($oa)
  write gpr($oa, rcx, RCX) to RCX
  if rcx <> zero($oa) and lcc(Opcode[1:0]) then
    call JMP(rip)
```

The `lcc` function maps the lower two bits of the opcode to a condition on the zero flag:

```
function lcc(code::bits 2)::bit
  = if code == 00b then not RFLAGS.ZF
    elif code == 01b then RFLAGS.ZF
    else 1b
```

Instruction CALL (Near)

Pushes the offset of the next instruction onto the stack and branches to the target address, which contains the first instruction of the called procedure. The target operand

can specify a register, a memory location, or a label. A procedure accessed by a near CALL is located in the same code segment as the CALL instruction. If the CALL target is specified by a register or memory location, then a 16-, 32-, or 64-bit rIP is read from the operand, depending on the operand size. A 16- or 32-bit rIP is zero-extended to 64 bits. If the CALL target is specified by a displacement, the signed displacement is added to the rIP (of the following instruction), and the result is truncated to 16, 32, or 64 bits, depending on the operand size. The signed displacement is 16 or 32 bits, depending on the operand size. In all cases, the rIP of the instruction after the CALL is pushed on the stack, and the size of the stack push (16, 32, or 64 bits) depends on the operand size of the CALL instruction. For near calls in 64-bit mode, the operand size defaults to 64 bits. The E8 opcode results in RIP = RIP + 32-bit signed displacement and the FF /2 opcode results in RIP = 64-bit offset from register or memory. No prefix is available to encode a 32-bit operand size in 64-bit mode. At the end of the called procedure, RET is used to return control to the instruction following the original CALL. When RET is executed, the rIP is popped off the stack, which returns control to the instruction after the CALL. *[cited from AMD volume 3]*

```
opcode "E8h" imm $z           : call CALL(RIP[$v-1:0]+sxt($v, op1))
opcode "FFh /010b" reg_mem $v : call CALL(op1)
action CALL(rip::bits $v)
  fail exception(xGP, 0000h) when not $x64_mode and nat(rip) > nat(CS.limit)
  fail exception(xGP, 0000h) when $x64_mode and not canonical(zxt(64, rip))
  call rsp = push($v, RIP[$v-1:0], RSP[$sa-1:0])
  write gpr($sa, rsp, RSP) to RSP
  write zxt(64, rip) to RIP
```

The `push` action is defined in chapter 15.

Instruction RET

Returns from a procedure previously entered by a CALL near instruction. This form of the RET instruction returns to a calling procedure within the current code segment. This instruction pops the rIP from the stack, with the size of the pop determined by the operand size. The new rIP is then zero-extended to 64 bits. The RET instruction can accept an immediate value operand that it adds to the rSP after it pops the target rIP. This action skips over any parameters previously passed back to the subroutine that are no longer needed. In 64-bit mode, the operand size defaults to 64 bits (eight bytes) without the need for a REX prefix. No prefix is available to encode a 32-bit operand size in 64-bit mode. *[cited from AMD volume 3]*

```
opcode "C3h"        : call RET(0000h)
opcode "C2h" imm 16 : call RET(op1)
action RET(pop_bytes::bits 16)
  call (rip, rsp) = pop($v, RSP[$sa-1:0])
  fail exception(xGP, 0000h) when not $x64_mode and nat(rip) > nat(CS.limit)
  fail exception(xGP, 0000h) when $x64_mode and not canonical(zxt(64, rip))
  write gpr($sa, rsp + zxt($sa, pop_bytes), RSP) to RSP
  write zxt(64, rip) to RIP
```

APPENDIX

I

FAR CONTROL TRANSFER INSTRUCTIONS

I.1 Fast System Call Instructions

There are two pairs of fast system call instructions: `SYSCALL/SYSRET` and `SYSENTER/SYSEXIT`. The latter pair is valid only in legacy 32-bit mode.

Both pairs of the instructions avoid costly descriptor table look-ups by loading the target code/stack segment components and instruction/stack pointers from the special registers. The segment attributes that are loaded into the code and stack segment register are predefined as follows:

```
function stackAttr = DataAttr with [A = 1b, W = 1b, E = 0b, DPL = 00b,
                                    P = 1b, DB = 1b, G = 1b]
function codeAttr32 = CodeAttr with [A = 1b, R = 1b, C = 0b, DPL = 00b,
                                     P = 1b, L = 0b, D = 0b, G = 1b]
function codeAttr64 = CodeAttr with [C = 0b, DPL = 00b, P = 1b, L = 1b, D = 0b]
```

Thus, the target stack segment is a present writeable data segment, which expands up. In legacy mode, the target code segment is present, readable, non-conforming, and 32-bit. In long mode, the target code segment is present, readable, non-conforming, and 64-bit.

Instruction SYSCALL (`when $SCE`)

In legacy x86 mode, when SYSCALL is executed, the EIP of the instruction following the SYSCALL is copied into the ECX register. Bits 31:0 of the SYSCALL/SYSRET target address register (STAR) are copied into the EIP register. (The STAR register is model-specific register C000_0081h.) New selectors are loaded, without permission checking, as follows:

- Bits 47:32 of the STAR register specify the selector that is copied into the CS register.

- Bits 47:32 of the STAR register + 8 specify the selector that is copied into the SS register.
- The CS_base and the SS_base are both forced to zero.
- The CS_limit and the SS_limit are both forced to 4 Gbyte.
- The CS segment attributes are set to execute/read 32-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by ESP.

When long mode is activated, the behavior of the SYSCALL instruction depends on whether the calling software is in 64-bit mode or compatibility mode. In 64-bit mode, SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from LSTAR bits 63:0. (The LSTAR register is model-specific register C000_0082h.) In compatibility mode, SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from CSTAR bits 63:0. (The CSTAR register is model-specific register C000_0083h.) New selectors are loaded, without permission checking (see above), as follows:

- Bits 47:32 of the STAR register specify the selector that is copied into the CS register.
- Bits 47:32 of the STAR register + 8 specify the selector that is copied into the SS register.
- The CS_base and the SS_base are both forced to zero.
- The CS_limit and the SS_limit are both forced to 4 Gbyte.
- The CS segment attributes are set to execute/read 64-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 64-bit stack referenced by RSP. *[cited from AMD volume 3]*

```
opcode "0F05h" : call SYSCALL
action SYSCALL when $legacy_mode
  write RIP[31:0] to RCX[31:0]
  write zxt(64, STAR.EIP) to RIP
  write 00b to CPL
  let sel = STAR.SYSCALL_CS with [RPL = 00b]
  write codeAttr32  to CS.attr
  write zero(64) to CS.base
  write FFFFFFFFh to CS.limit
  write sel + 0008h to SS.sel
  write stackAttr to SS.attr
  write zero(64) to SS.base
  write FFFFFFFFh to SS.limit
  write RFLAGS with [IF = 0b, RF = 0b, VM = 0b] to RFLAGS

action SYSCALL when $long_mode
  write RIP to RCX
  if $x64_mode then
```

```
  write LSTAR to RIP
else
  write CSTAR to RIP
write 00b to CPL
write RFLAGS to R11
write (RFLAGS with [RF = 0b]) & ~SFMASK to RFLAGS
let sel = STAR.SYSCALL_CS with [RPL = 00b]
write sel to CS.sel
write codeAttr64  to CS.attr
write zero(64) to CS.base
write FFFFFFFFh to CS.limit
write sel + 0008h to SS.sel
write stackAttr to SS.attr
write zero(64) to SS.base
write FFFFFFFFh to SS.limit
```

Instruction SYSRET (`when $SCE`)

When a system procedure performs a SYSRET back to application software, the CS selector is updated from bits 63:50 of the STAR register (STAR.SYSRET_CS) as follows:

- If the return is to 32-bit mode (legacy or compatibility), CS is updated with the value of STAR.SYSRET_CS.
- If the return is to 64-bit mode, CS is updated with the value of STAR.SYSRET_CS + 16.

In both cases, the CPL is forced to 3, effectively ignoring STAR bits 49:48. The SS selector is updated to point to the next descriptor-table entry after the CS descriptor (STAR.SYSRET_CS + 8), and its RPL is not forced to 3.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 RET instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS base value is forced to 0.
- The CS limit value is forced to 4 Gbytes.

The CS segment attributes are set to execute-read 32 bits or 64 bits (see below). The SS segment base, limit, and attributes are not modified. When SYSCALLed system software is running in 64-bit mode, it has been entered from either 64-bit mode or compatibility mode. The corresponding SYSRET needs to know the mode to which it must return. Executing SYSRET in non–64-bit mode or with a 16- or 32-bit operand size returns to 32-bit mode with a 32-bit stack pointer. Executing SYSRET in 64-bit mode with a 64-bit operand size returns to 64-bit mode with a 64-bit stack pointer. The instruction pointer is updated with the return address based on the operating mode in which SYSRET is executed:

- If returning to 64-bit mode, SYSRET loads RIP with the value of RCX.
- If returning to 32-bit mode, SYSRET loads EIP with the value of ECX.

How SYSRET handles RFLAGS depends on the processor's operating mode:

- If executed in 64-bit mode, SYSRET loads the lower–32 RFLAGS bits from R11[31:0] and clears the upper 32 RFLAGS bits.
- If executed in legacy mode or compatibility mode, SYSRET sets EFLAGS.IF. *[cited from AMD volume 3]*

```
opcode "0F07h" : call SYSRET
action SYSRET when not $x64_mode
  fail exception(xGP, 0000h) when CPL <> 00b
  write zxt(64, RCX[31:0]) to RIP
  write 11b to CPL
  write RFLAGS with [IF = 1b] to RFLAGS
  let sel = STAR.SYSRET_CS with [RPL = 00b]
  write sel to CS.sel
  write codeAttr32  to CS.attr
  write zero(64) to CS.base
  write FFFFFFFFh to CS.limit
  write sel + 0008h to SS.sel
  write zero(64) to SS.base
  write FFFFFFFFh to SS.limit
action SYSRET when $x64_mode
  fail exception(xGP, 0000h) when CPL <> 00b
  write 11b to CPL
  write R11 to RFLAGS
  let sel = STAR.SYSRET_CS with [RPL = 00b]
  write sel + 0008h to SS.sel
  write zero(64) to CS.base
  write FFFFFFFFh to CS.limit
  if $v == 64 then
    write RCX to RIP
    write sel + 0010h to CS.sel
    write codeAttr64 to CS.attr
  else
    write zxt(64, RCX[31:0]) to RIP
    write sel to CS.sel
    write codeAttr32 to CS.attr
```

Instruction SYSENTER (`when $legacy_mode`)

Three model-specific registers (MSRs) are used to specify the target address and stack pointers for the SYSENTER instruction, as well as the CS and SS selectors of the called and returned procedures:

- MSR_SYSENTER_CS: Contains the CS selector of the called procedure. The SS selector is set to
- MSR_SYSENTER_CS + 8.
- MSR_SYSENTER_ESP: Contains the called procedure's stack pointer.
- MSR_SYSENTER_EIP: Contains the offset into the CS of the called procedure.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 CALL instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS and SS base values are forced to 0.
- The CS and SS limit values are forced to 4 Gbytes.
- The CS segment attributes are set to execute/read 32-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by ESP.

System software must create corresponding descriptor-table entries referenced by the new CS and SS selectors that match the values described above. The return EIP and application stack are not saved by this instruction. System software must explicitly save that information. *[cited from AMD volume 3]*

```
opcode "0F34h" : call SYSENTER
action SYSENTER when $legacy_mode
  fail exception(xGP, 0000h) when $x16_mode
  let sel = SYSENTER_CS.CS
  fail exception(xGP, 0000h) when sel & FFFEh == 0000h
  write sel to CS.sel
  write codeAttr32  to CS.attr
  write zero(64) to CS.base
  write FFFFFFFFh to CS.limit
  write sel + 0008h to SS.sel
  write stackAttr to SS.attr
  write zero(64) to SS.base
  write FFFFFFFFh to SS.limit
  write zxt(64, SYSENTER_EIP.EIP) to RIP
  write zxt(64, SYSENTER_ESP.ESP) to RSP
  write 00b to CPL
  write RFLAGS with [VM = 0b, IF = 0b] to RFLAGS
```

Instruction SYSEXIT (`when $legacy_mode`)

When a system procedure performs a SYSEXIT back to application software, the CS selector is updated to point to the second descriptor entry after the SYSENTER CS value (MSR SYSENTER_CS+16). The SS selector is updated to point to the third descriptor entry after the SYSENTER CS value (MSR SYSENTER_CS+24). The CPL is forced to 3, as are the descriptor privilege levels. The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 RET instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS and SS base values are forced to 0.
- The CS and SS limit values are forced to 4 Gbytes.
- The CS segment attributes are set to 32-bit read/execute at CPL 3.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by ESP.

System software must create corresponding descriptor-table entries referenced by the new CS and SS selectors that match the values described above. The following additional actions result from executing SYSEXIT:

- EIP is loaded from EDX.
- ESP is loaded from ECX.

System software must explicitly load the return address and application software-stack pointer into the EDX and ECX registers prior to executing SYSEXIT. *[cited from AMD volume 3]*

```
opcode "0F35h" : call SYSEXIT
action SYSEXIT
  fail exception(xGP, 0000h) when $x16_mode
  fail exception(xGP, 0000h) when CPL <> 00b
  let sel = SYSENTER_CS.CS
  fail exception(xGP, 0000h) when sel & FFFEh == 0000h
  write sel+0010h to CS.sel
  write codeAttr32  to CS.attr
  write zero(64) to CS.base
  write FFFFFFFFh to CS.limit
  write sel + 0018h to SS.sel
  write stackAttr to SS.attr
  write zero(64) to SS.base
  write FFFFFFFFh to SS.limit
  write zxt(64, RDX) to RIP
  write zxt(64, RCX) to RSP
  write 11b to CPL
  write RFLAGS with [RF = 0b] to RFLAGS
```

I.2 Far JMP and CALL instructions

Instruction JMP (FAR)

Unconditionally transfers control to a new address without saving the current CS:rIP values. This form of the instruction jumps to an address outside the current code segment and is called a far jump. The operand specifies a target selector and offset. The target operand can be specified by the instruction directly, by containing the far pointer in the jmp far opcode itself, or indirectly, by referencing a far pointer in memory. In 64-bit mode, only indirect far jumps are allowed, executing a direct far jmp (opcode EA) will generate an undefined opcode exception. For both direct and indirect far calls, if the JMP (Far) operand-size is 16 bits, the instruction’s operand is a 16-bit selector followed by a 16-bit offset. If the operand-size is 32 or 64 bits, the operand is a 16-bit selector followed by a 32-bit offset. In all modes, the target selector used by the instruction can be a code selector. Additionally, the target selector can also be a call gate in protected mode, or a task gate or TSS selector in legacy protected mode. - Target is a code segment - Control is transferred to the target CS:rIP. In this case, the target offset can only be a 16 or 32 bit value, depending on operand-size, and is zero-extended to 64 bits. No CPL change is allowed. - Target is a call gate - The call gate specifies the actual

target code segment and offset, and control is transferred to the target CS:rIP. When jumping through a call gate, the size of the target rIP is 16, 32, or 64 bits, depending on the size of the call gate. If the target rIP is less than 64 bits, it is zero- extended to 64 bits. In long mode, only 64-bit call gates are allowed, and they must point to 64-bit code segments. No CPL change is allowed. - Target is a task gate or a TSS - If the mode is legacy protected mode, then a task switch occurs. *[cited from AMD volume 3]*

```
opcode "EAh" imm_ptr $z+16      : when not $x64_mode
opcode "FFh /101b" mem_ptr $z+16
call jmp_far(op1[$z+15:$z], op1[$z-1:0])
```

Refer to section 16.1 for the definition of the `jmp_far` action.

Instruction CALL (Far)

Pushes procedure linking information onto the stack and branches to the target address, which contains the first instruction of the called procedure. The operand specifies a target selector and offset. The instruction can specify the target directly, by including the far pointer in the CALL (Far) opcode itself, or indirectly, by referencing a far pointer in memory. In 64-bit mode, only indirect far calls are allowed, executing a direct far call (opcode 9A) generates an undefined opcode exception. For both direct and indirect far calls, if the CALL (Far) operand-size is 16 bits, the instruction's operand is a 16- bit selector followed by a 16-bit offset. If the operand-size is 32 or 64 bits, the operand is a 16-bit selector followed by a 32-bit offset. The target selector used by the instruction can be a code selector in all modes. Additionally, the target selector can reference a call gate in protected mode, or a task gate or TSS selector in legacy protected mode. - Target is a code selector - The CS:rIP of the next instruction is pushed to the stack, using operand- size stack pushes. Then code is executed from the target CS:rIP. In this case, the target offset can only be a 16- or 32-bit value, depending on operand-size, and is zero-extended to 64 bits. No CPL change is allowed. - Target is a call gate - The call gate specifies the actual target code segment and offset. Call gates allow calls to the same or more privileged code. If the target segment is at the same CPL as the current code segment, the CS:rIP of the next instruction is pushed to the stack. If the CALL (Far) changes privilege level, then a stack-switch occurs, using an inner-level stack pointer from the TSS. The CS:rIP of the next instruction is pushed to the new stack. If the mode is legacy mode and the param-count field in the call gate is non-zero, then up to 31 operands are copied from the caller's stack to the new stack. Finally, the caller's SS:rSP is pushed to the new stack. When calling through a call gate, the stack pushes are 16-, 32-, or 64-bits, depending on the size of the call gate. The size of the target rIP is also 16, 32, or 64 bits, depending on the size of the call gate. If the target rIP is less than 64 bits, it is zero-extended to 64 bits. Long mode only allows 64- bit call gates that must point to 64-bit code segments. - Target is a task gate or a TSS - If the mode is legacy protected mode, then a task switch occurs. *[cited from AMD volume 3]*

```
opcode "9Ah" imm_ptr $z+16      : when not $x64_mode
opcode "FFh /011b" mem_ptr $z+16
call call_far(op1[$z+15:$z], op1[$z-1:0])
```

Refer to section 16.2 for the definition of the `call_far` action.

I.3 Software Interrupt Instructions

Instruction INT

Transfers execution to the interrupt handler specified by an 8-bit unsigned immediate value. This value is an interrupt vector number (00h to FFh), which the processor uses as an index into the interrupt- descriptor table (IDT). *[cited from AMD volume 3]*

```
opcode "CDh" imm 8 : call INT(op1)
action INT(op1::bits 8)
  fail intercept(VMEXIT_SWINT) when _vmcb_ca.intercept.INT
  call icall(op1, 0b, 0000h)
```

Refer to section 16.3 for the definition of the `icall` action.

Instruction INT3 (`when not $x64_mode`)

Calls the debug exception handler. This instruction maps to a 1-byte opcode (CC) that raises a #BP exception. The INT 3 instruction is normally used by debug software to set instruction breakpoints by replacing the first byte of the instruction opcode bytes with the INT 3 opcode. This one-byte INT 3 instruction behaves differently from the two-byte INT 3 instruction (opcode CD 03) in two ways: - The #BP exception is handled without any IOPL checking in virtual x86 mode. (IOPL mismatches will not trigger an exception.) - In VME mode, the #BP exception is not redirected via the interrupt redirection table. (Instead, it is handled by a protected mode handler.) *[cited from AMD volume 3]*

```
opcode "CCh" : call INT3
action INT3
  fail intercept(VMEXIT_EXCP0 + zxt(64, xBP)) when _vmcb_ca.EXCP[nat(xBP)]
  call icall(xBP, 0b, 0000h)
```

Refer to section 16.3 for the definition of the `icall` action.

Instruction INTO (`when not $x64_mode`)

Checks the overflow flag (OF) in the rFLAGS register and calls the overflow exception (#OF) handler if the OF flag is set to 1. This instruction has no effect if the OF flag is cleared to 0. The INTO instruction detects overflow in signed number addition. *[cited from AMD volume 3]*

```
opcode "CEh" : call INTO
action INTO
  fail intercept(VMEXIT_EXCP0 + zxt(64, xOF)) when _vmcb_ca.EXCP[nat(xOF)]
  call icall(xOF, 0b, 0000h)
```

Refer to section 16.3 for the definition of the `icall` action.

I.4 Return Instructions

Instruction RETF

Returns from a procedure previously entered by a CALL Far instruction. This form of the RET instruction returns to a calling procedure in a different segment than the current code segment. It can return to the same CPL or to a less privileged CPL. RET

Far pops a target CS and rIP from the stack. If the new code segment is less privileged than the current code segment, the stack pointer is incremented by the number of bytes indicated by the immediate operand, if present; then a new SS and rSP are also popped from the stack. The final value of rSP is incremented by the number of bytes indicated by the immediate operand, if present. This action skips over the parameters (previously passed to the subroutine) that are no longer needed. All stack pops are determined by the operand size. If necessary, the target rIP is zero-extended to 64 bits before assuming program control. If the CPL changes, the data segment selectors are set to NULL for any of the data segments (DS, ES, FS, GS) not accessible at the new CPL. *[cited from AMD volume 3]*

```
opcode "CBh"        : call ret_far(0000h, 0b)
opcode "CAh" imm 16 : call ret_far(op1, 0b)
```

Refer to section 16.4 for the definition of the `ret_far` action.

Instruction IRET

Returns program control from an exception or interrupt handler to a program or procedure previously interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions also perform a return from a nested task. All flags, CS, and rIP are restored to the values they had before the interrupt so that execution may continue at the next instruction following the interrupt or exception. In 64-bit mode or if the CPL changes, SS and RSP are also restored. *[cited from AMD volume 3]*

```
opcode "CFh" : call IRET
```

If the nested flag `NT` is set then the `IRET` action performs a return from the nested task, otherwise, it performs a far return from procedure.

```
action IRET
  fail intercept(VMEXIT_IRET) when _vmcb_ca.intercept.IRET
  if not RFLAGS.NT then call ret_far(0000h, 1b)
  else fail exception(xGP, 0000h) when $long_mode
    call tss_sel = read_tss(16, zero($va))
    call (temp_desc, tss_valid) = read_desc(tss_sel)
    let tss_desc = temp_desc::TSS
    fail exception(xTS, tss_sel & FFF8h) when not tss_valid
    fail exception(xTS, tss_sel & FFF8h) when not tss_desc.P
    fail exception(xTS, tss_sel & FFF8h) when not isTSS(tss_desc)
    fail exception(xTS, tss_sel & FFF8h) when tss_desc.busy
    call task_switch(tss_sel, tss_desc)
```

Refer to section 16.4 for the definition of the `ret_far` action. For discussion of task switch refer to section 16.5.

APPENDIX

J

STRING INSTRUCTIONS

String instructions `LODS`, `MOVS`, `STOS`, `CMPS`, `SCAS` work with strings that are located at logical addresses (`DS`, `RSI`) and (`ES`, `RDI`). After performing an operation over the current elements of the strings, they increment or decrement `RSI` and/or `RDI` depending on the direction flags. If a string instruction has a repetition prefix REP (`xF3`), REPZ (`xF3`), REPNZ (`xF2`) then the `RCX` register is decremented. If the new `RCX` is not zero and zero flag satisfies the condition specified by the prefix, then the instruction is repeated. Notice that the values of REP and REPZ prefixes coincide. However, there is no ambiguity because instruction that can have a REP prefix cannot have REPZ/REPNZ prefixes and vice versa.

An instruction with a REP prefix executes according to the following scheme (pseudocode):

```
while (RCX > 0) do
  process an element of the string
  adjust RSI and/or RDI
  decrement RCX
```

An instruction with a REPZ/REPNZ prefix does the same, but alse checks the value of the zero flag before the next iteration:

```
while (RCX > 0) do
  process an element of the string
  adjust RSI and/or RDI
  decrement RCX
  if ZF does not match prefix condition then exit loop
```

Recall that the `_old_RIP` register stores the pointer to the current instruction and the `RIP` register stores the pointer to the next instruction. Thus, we can repeat the current instruction by setting the `RIP` to the `_old_RIP`. This allows us to model the `while` loops from the pseudocode.

The following action checks for a REP prefix. If the prefix is present, then the action decrements the `RCX` register and adjusts the `RIP` to repeat the instruction if the new `RCX` is not zero.

```
action repeat
  if _prefix.xF3 then
    let rcx = RCX[$oa-1:0] - one($oa)
    write gpr($oa, rcx, RCX) to RCX
    write _old_RIP to RIP when rcx <> zero($oa)
```

The `repeat_with_zf` action does the same but also checks whether the new zero flag matches the prefix condition:

```
action repeat_with_zf(zf::bit)
  if _prefix.xF2 or _prefix.xF3 then
    let rcx = RCX[$oa-1:0] - one($oa)
    write gpr($oa, rcx, RCX) to RCX
    if rcx <> zero($oa) then
      if _prefix.xF2 and not zf or _prefix.xF3 and zf then
        write _old_RIP to RIP
```

The `RSI/RDI` registers are incremented if the direction flag `DF` is cleared and are decremented otherwise:

```
action adjust_rsi($n::{8, 16, 32, 64})
  if RFLAGS.DF then
    write gpr($oa, RSI[$oa-1:0] - bits($oa, $n/8), RSI) to RSI
  else
    write gpr($oa, RSI[$oa-1:0] + bits($oa, $n/8), RSI) to RSI
action adjust_rdi($n::{8, 16, 32, 64})
  if RFLAGS.DF then
    write gpr($oa, RDI[$oa-1:0] - bits($oa, $n/8), RDI) to RDI
  else
    write gpr($oa, RDI[$oa-1:0] + bits($oa, $n/8), RDI) to RDI
```

We will use these action in the definitions of the string instructions.

Instruction LODS

Copies the byte, word, doubleword, or quadword in the memory location pointed to by the DS:rSI registers to the AL, AX, EAX, or RAX register, depending on the size of the operand, and then increments or decrements the rSI register according to the state of the DF flag in the rFLAGS register. If the DF flag is 0, the instruction increments rSI; otherwise, it decrements rSI. It increments or decrements rSI by 1, 2, 4, or 8, depending on the number of bytes being loaded. *[cited from AMD volume 3]*

```
opcode "ACh" ds_rsi 8
opcode "ADh" ds_rsi $v
call LODS($n)
action LODS($n::{8, $v})
  if not _prefix.xF3 or RCX[$oa-1:0] <> zero($oa) then
    call val = read_op($n, ds_rsi)
    write gpr($n, val, RAX) to RAX
    call adjust_rsi($n)
    call repeat
```

Instruction STOS

Copies a byte, word, doubleword, or quadword from the AL, AX, EAX, or RAX registers to the memory location pointed to by ES:rDI and increments or decrements the rDI

register according to the state of the DF flag in the rFLAGS register. If the DF flag is 0, the instruction increments the pointer; otherwise, it decrements the pointer. It increments or decrements the pointer by 1, 2, 4, or 8, depending on the size of the value being copied. *[cited from AMD volume 3]*

```
opcode "AAh" es_rdi 8
opcode "ABh" es_rdi $v
call STOS($n)
action STOS($n::{8, $v})
  if not _prefix.xF3 or RCX[$oa-1:0] <> zero($oa) then
    call write_op($n, RAX[$n-1:0], es_rdi)
    call adjust_rdi($n)
    call repeat
```

Instruction MOVS

Moves a byte, word, doubleword, or quadword from the memory location pointed to by DS:rSI to the memory location pointed to by ES:rDI, and then increments or decrements the rSI and rDI registers according to the state of the DF flag in the rFLAGS register. If the DF flag is 0, the instruction increments both pointers; otherwise, it decrements them. It increments or decrements the pointers by 1, 2, 4, or 8, depending on the size of the operands. *[cited from AMD volume 3]*

```
opcode "A4h" es_rdi 8, ds_rsi 8
opcode "A5h" es_rdi $v, ds_rsi $v
call MOVS($n)
action MOVS($n::{8, $v})
  if not _prefix.xF3 or RCX[$oa-1:0] <> zero($oa) then
    call val = read_op($n, ds_rsi)
    call write_op($n, val, es_rdi)
    call adjust_rsi($n)
    call adjust_rdi($n)
    call repeat
```

Instruction SCAS

Compares the AL, AX, EAX, or RAX register with the byte, word, doubleword, or quadword pointed to by ES:rDI, sets the status flags in the rFLAGS register according to the results, and then increments or decrements the rDI register according to the state of the DF flag in the rFLAGS register. If the DF flag is 0, the instruction increments the rDI register; otherwise, it decrements it. The instruction increments or decrements the rDI register by 1, 2, 4, or 8, depending on the size of the operands. *[cited from AMD volume 3]*

```
opcode "AEh" es_rdi 8
opcode "AFh" es_rdi $v
call SCAS($n)
action SCAS($n::{8, $v})
  if not _prefix.xF3 and not _prefix.xF2 or RCX[$oa-1:0] <> zero($oa) then
    call val = read_op($n, es_rdi)
    call sub($n, RAX[$n-1:0], val, 0b)
    call adjust_rdi($n)
    call repeat_with_zf(RAX[$n-1:0] == val)
```

Instruction CMPS

Compares the bytes, words, doublewords, or quadwords pointed to by the rSI and rDI registers, sets or clears the status flags of the rFLAGS register to reflect the results, and then increments or decrements the rSI and rDI registers according to the state of the DF flag in the rFLAGS register. To perform the comparison, the instruction subtracts the second operand from the first operand and sets the status flags in the same manner as the SUB instruction, but does not alter the first operand. The two operands must be the same size. If the DF flag is 0, the instruction increments rSI and rDI; otherwise, it decrements the pointers. It increments or decrements the pointers by 1, 2, 4, or 8, depending on the size of the operands. *[cited from AMD volume 3]*

```
opcode "A6h" es_rdi 8, ds_rsi 8
opcode "A7h" es_rdi $v, ds_rsi $v
call CMPS($n)
action CMPS($n::{8, $v})
  if not _prefix.xF3 and not _prefix.xF2 or RCX[$oa-1:0] <> zero($oa) then
    call op1 = read_op($n, es_rdi)
    call op2 = read_op($n, ds_rsi)
    call sub($n, op1, op2, 0b)
    call adjust_rdi($n)
    call adjust_rsi($n)
    call repeat_with_zf(op1 == op2)
```

APPENDIX

K

INPUT/OUTPUT INSTRUCTIONS

Instructions `IN`, `OUT`, `INS`, `OUTS` read or write I/O ports using the `read_port`, `write_port` actions defined in section 13.4. Each I/O port has the width of one byte. The instructions can access several adjacent ports at the same time as if the I/O port space was a memory space.

These instructions have complex interception logic. There two ways to intercept an I/O instruction: via the permission bitmap in the task state segment and via the permission bitmap in the guest control area. The former kind of intercept raises an exception, the latter kind of intercept triggers guest exit.

The following action checks whether the given instruction is intercepted or not. The instruction is specified by the access width, the port index, and two indicators: `string` and `input`. The `string` flag indicates whether the instruction is a string instruction (`INS`, `OUTS`) or not. The `input` flag indicates whether the instruction is an input instruction (`IN`, `INS`) or an output instruction (`OUT`, `OUTS`).

```
action check_io_intercept($n::{8, 16, 32}, port::bits 16, string::bit, input::bit)
  if CPL > RFLAGS.IOPL or $vm86_mode then
     call tss_intercepted = tss_io_intercepted($n, port)
     fail exception(xGP, 0000h) when tss_intercepted
  call intercepted = io_intercepted($n, port)
  let exitinfo = io_exitinfo(port, _prefix.xF2 or _prefix.xF3, string, input)
  fail intercept1(VMEXIT_IOIO, exitinfo) when intercepted
```

The `tss_io_intercepted` action looks up the permission bitmap in the task segment and is defined in section 13.14.

The `io_intercepted` action looks up the permission bitmap in guest control area if the processor is running in guest mode. The physical address of the start of the bitmap is stored in the `_vmcb_ca.IOPM_BASE_PA` field. The action reads a bit for each accessed byte. If any of the bits is set, then the instruction is intercepted.

```
action io_intercepted($n::{8, 16, 32}, port::bits 16)::bit
  if not _guest or not _vmcb_ca.intercept.IO_PROT then return 0b
  else let offset = zxt($pa, port[15:3])
       let $bit_idx = nat(port[2:0])::[0..7]
```

```
    let addr = _vmcb_ca.IOPM_BASE_PA[$pa-1:0] + offset
    call mask = pread(sys, $n+8, addr, WB)
    return mask[$n+$bit_idx-1:$bit_idx] <> zero($n)
```

If the instruction is intercepted then the processor performs guest exits and writes the exit information into the guest control area. The exit information specifies the instruction type (input/output, string/not string), the presence of a repeat prefix, the operand width, the address width, and the port index:

```
layout IOExitInfo
  field input::bit
  field rsv1::bit reserved and must be 0b
  field string::bit
  field repeat::bit
  field op_width16::bit
  field op_width32::bit
  field op_width64::bit
  field addr_width16::bit
  field addr_width32::bit
  field addr_width64::bit
  field rsv2::bits 6 reserved and must be 000000b
  field port::bits 16
  field rsv3::bits 32 reserved and must be zero(32)
```

We will use the following function for constructing the I/O exit information:

```
function io_exitinfo(port::bits 16, repeat::bit, string::bit, input::bit)::bits 64
  = (IOExitInfo) with [ input        = input,
                        string       = string,
                        repeat       = repeat,
                        op_width16   = $v == 16,
                        op_width32   = $v == 32,
                        op_width64   = $v == 64,
                        addr_width16 = $oa == 16,
                        addr_width32 = $oa == 32,
                        addr_width64 = $oa == 64,
                        port         = port ]
```

Instruction IN

Transfers a byte, word, or doubleword from an I/O port (second operand) to the AL, AX or EAX register (first operand). The port address can be an 8-bit immediate value (00h to FFh) or contained in the DX register (0000h to FFFFh). If the CPL is higher than IOPL, or the mode is virtual mode, IN checks the I/O permission bitmap in the TSS before allowing access to the I/O port. *[cited from AMD volume 3]*

```
opcode "E4h" rax 8, imm 8
opcode "E5h" rax $z, imm 8
opcode "ECh" rax 8, rdx 16
opcode "EDh" rax $z, rdx 16
call op1' = IN($n, zxt(16, op2))
action IN($n::{8, 16, 32}, port::bits 16)::bits $n
  call check_io_intercept($n, port, 0b, 1b)
  call res = read_port($n, port)
  return res
```

Instruction OUT

Copies the value from the AL, AX, or EAX register (second operand) to an I/O port (first operand). The port address can be a byte-immediate value (00h to FFh) or the value in the DX register (0000h to FFFFh). The source register used determines the size of the port (8, 16, or 32 bits). *[cited from AMD volume 3]*

```
opcode "E6h" imm 8, rax 8
opcode "E7h" imm 8, rax $z
opcode "EEh" rdx 16, rax 8
opcode "EFh" rdx 16, rax $z
call OUT(zxt(16, op1), $n2, op2)
action OUT(port::bits 16, $n::{8, 16, 32}, val::bits $n)
  call check_io_intercept($n, port, 0b, 0b)
  call write_port($n, val, port)
```

Instruction INS

Transfers data from the I/O port specified in the DX register to an input buffer specified in the rDI register and increments or decrements the rDI register according to the setting of the DF flag in the rFLAGS register. If the DF flag is 0, the instruction increments rDI by 1, 2, or 4, depending on the number of bytes read. If the DF flag is 1, it decrements the pointer by 1, 2, or 4. *[cited from AMD volume 3]*

```
opcode "6Ch" es_rdi 8, rdx 16
opcode "6Dh" es_rdi $z, rdx 16
call INS($n, op2)
action INS($n::{8, 16, 32}, port::bits 16)
  if not _prefix.xF3 or RCX[$oa-1:0] <> zero($oa) then
    call check_io_intercept($n, port, 0b, 1b)
    call val = read_port($n, port)
    call write_op($n, val, es_rdi)
    call adjust_rdi($n)
    call repeat
```

Instruction OUTS

Copies data from the memory location pointed to by DS:rSI to the I/O port address (0000h to FFFFh) specified in the DX register, and then increments or decrements the rSI register according to the setting of the DF flag in the rFLAGS register. If the DF flag is 0, the instruction increments rSI; otherwise, it decrements rSI. It increments or decrements the pointer by 1, 2, or 4, depending on the size of the value being copied. *[cited from AMD volume 3]*

```
opcode "6Eh" rdx 16, ds_rsi 8
opcode "6Fh" rdx 16, ds_rsi $z
call OUTS($n2, op1)
action OUTS($n::{8, 16, 32}, port::bits 16)
  if not _prefix.xF3 or RCX[$oa-1:0] <> zero($oa) then
    call val = read_op($n, ds_rsi)
    call check_io_intercept($n, port, 0b, 0b)
    call write_port($n, val, port)
    call adjust_rsi($n)
    call repeat
```

APPENDIX

L

SEGMENTATION INSTRUCTIONS

L.1 Load SR and GPR from Memory

Instructions `LDS`, `LES`, `LSS`, `LFS`, `LGS` load "a far pointer from a memory location (second operand) into a segment register (mnemonic) and general-purpose register (first operand). The instruction stores the 16-bit segment selector of the pointer into the segment register and the 16-bit or 32-bit offset portion into the general-purpose register. The operand-size attribute determines whether the pointer is 32-bit or 48-bit." [cited from AMD volume 3].

All these instructions use the following action that takes a segment register index and a pair of (selector, offset), fetches the segment descriptor using the selector, loads the selector and the segment descriptor into the segment register, and returns the offset, which is later on is loaded into the GPR.

```
action LxS(idx::SRIndex, sel_offset::bits ($z+16))::bits $z
  let sel = sel_offset[$z+15:$z]
  let offset = sel_offset[$z-1:0]
  call write_sr(sys, sel, idx)
  return offset
```

Instruction LDS (`when not $x64_mode`) *[cited from AMD volume 3]*

```
opcode "C5h" reg $z, mem_ptr $z+16 : call op1' = LxS(iDS, op2)
```

Instruction LES (`when not $x64_mode`) *[cited from AMD volume 3]*

```
opcode "C4h" reg $z, mem_ptr $z+16 : call op1' = LxS(iES, op2)
```

Instruction LSS *[cited from AMD volume 3]*

```
opcode "0FB4h" reg $z, mem_ptr $z+16 : call op1' = LxS(iSS, op2)
```

Instruction LFS *[cited from AMD volume 3]*

```
opcode "0FB4h" reg $z, mem_ptr $z+16 : call op1' = LxS(iFS, op2)
```

Instruction LGS *[cited from AMD volume 3]*

```
opcode "0FB5h" reg $z, mem_ptr $z+16 : call op1' = LxS(iGS, op2)
```

L.2 SWAPGS

Instruction SWAPGS (`when $x64_mode`)

The SWAPGS instruction only exchanges the base-address value located in the KernelGSBase model- specific register (MSR address C000_0102h) with the base-address value located in the hidden-portion of the GS selector register (GS.base). This allows the system-kernel software to access kernel data structures by using the GS segment-override prefix during memory references. *[cited from AMD volume 3]*

```
opcode "0F01h /111b mod 11b rm 000b" : call SWAPGS
action SWAPGS
  fail exception(xGP, 0000h) when CPL <> 00b
  write KernelGSBase to GS.base
  write GS.base to KernelGSBase
```

L.3 Task Register Access

Instruction LTR (`when $protected_mode`)

Loads the specified segment selector into the visible portion of the task register ('TR'). The processor uses the selector to locate the descriptor for the TSS in the global descriptor table. It then loads this descriptor into the hidden portion of 'TR'. The TSS descriptor in the GDT is marked busy, but no task switch is made. *[cited from AMD volume 3]*

```
opcode "0F00h /011b" reg_mem 16 : call LTR(op1)
```

The `LTR` action performs the following steps:

- Check that the processor is running at the most privileged level.
- Check for an intercept.
- Check that the new selector is not null.
- Fetch the task segment descriptor from the GDT.
- Fetch the upper descriptor in case the task segment descriptor has it.
- Check that the task segment descriptor is valid.
- Mark the new task state segment as busy.
- Write the new selector and descriptor into the `TR` register.

```
action LTR(sel::Selector)
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_TR_WRITE) when _vmcb_ca.intercept.WRITE_TR
  fail exception(xGP, sel & FFF8h) when sel.TI
  fail exception(xGP, sel & FFF8h) when (sel & FFF8h) == 0000h
  call (desc_temp, desc_valid) = read_desc(sel)
  fail exception(xGP, sel & FFF8h) when not desc_valid
  call (upper, upper_valid) = read_upper_ldt_tss(sel)
  fail exception(xGP, sel & FFF8h) when not upper_valid
  let desc = desc_temp::TSS
  let base = upper[31:0] ++ desc.base
  call check_tss(sys, sel, desc, upper)
  call mark_tss_busy(sel, desc)
  write sel to TR.sel
  write desc_attr(desc) to TR.attr
  write base to TR.base
  write desc.limit to TR.limit
```

The `read_desc`, `read_upper_ldt_tss` actions are defined in section 13.9. The `check_tss` action makes sure that the given descriptor is present, accessible, and defines a valid tast state segment.

```
action check_tss(origin::Origin, sel::Selector,
                 desc::TSS, upper::UpperDescriptor)
  fail exception(xNP_xTS(origin), sel & FFF8h) when not desc.P
  fail exception(xGP_xTS(origin), sel & FFF8h) when not isTSS(desc)
  let base = upper[31:0] ++ desc.base
  fail exception(xGP_xTS(origin), sel & FFF8h)
    when $x64_mode and not canonical(base)
  fail exception(xGP_xTS(origin), sel & FFF8h)
    when not can_access_desc(sel, desc)
```

Marking the TSS busy means writing the TSS descriptor with the `busy` bit set into the GDT.

```
action mark_tss_busy(sel::Selector, desc::TSS)
  call write_desc(sel, desc with [busy = 1b])
```

The `write_desc` action is defined in section 13.9.

Instruction STR (`when $protected_mode`)

Stores the task register (‘TR’) selector to a register or memory destination operand. *[cited from AMD volume 3]*

```
opcode "0F00h /001b" reg_mem $vw : call op1' = STR
action STR::bits $vw
  fail intercept(VMEXIT_TR_READ) when _vmcb_ca.intercept.READ_TR
  return zxt($vw, TR.sel)
```

L.4 Descriptor Table Register Access

Instruction LLDT (`when $protected_mode`)

Loads the specified segment selector into the visible portion of the local descriptor table (LDT). The processor uses the selector to locate the descriptor for the LDT in the global descriptor table. It then loads this descriptor into the hidden portion of the LDTR. *[cited from AMD volume 3]*

```
opcode "0F00h /010b" reg_mem 16 : call LLDT(op1)
```

The `LLDT` action works similarly to the `LTR` action described in the previous section, but it is possible to load a null selector to the `LDTR`.

```
action LLDT(sel::Selector)
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_LDTR_WRITE) when _vmcb_ca.intercept.WRITE_LDTR
  fail exception(xGP, sel & FFF8h) when sel.TI
  if (sel & FFF8h) == 0000h then
    write sel to LDTR.sel
    write 0000h to LDTR.attr
  else call (desc_temp, desc_valid) = read_desc(sel)
       fail exception(xGP, sel & FFF8h) when not desc_valid
       call (upper, upper_valid) = read_upper_ldt_tss(sel)
       fail exception(xGP, sel & FFF8h) when not upper_valid
       let desc = desc_temp::LDT
       let base = upper[31:0] ++ desc.base
       fail exception(xGP, sel & FFF8h) when not canonical(base) and $x64_mode
       call check_ldt(sys, sel, desc, upper)
       write sel to LDTR.sel
       write desc_attr(desc) to LDTR.attr
       write base to LDTR.base
       write desc.limit to LDTR.limit
```

When a system segment descriptor is loaded into the `LDTR` register, the processor ensures that the descriptor is a present LDT descriptor and that the current code can access the descriptor.

```
action check_ldt(origin::Origin, sel::Selector,
                 desc::LDT, upper::UpperDescriptor)
  fail exception(xNP_xTS(origin), sel & FFF8h) when not desc.P
  fail exception(xGP_xTS(origin), sel & FFF8h) when not isLDT(desc)
  let base = upper[31:0] ++ desc.base
  fail exception(xGP_xTS(origin), sel & FFF8h)
    when $x64_mode and not canonical(base)
  fail exception(xGP_xTS(origin), sel & FFF8h)
    when not can_access_desc(sel, desc)
```

Instruction SLDT (`when $protected_mode`)

Stores the local descriptor table (LDT) selector to a register or memory destination operand. *[cited from AMD volume 3]*

```
opcode "0F00h /000b" reg_mem $vw : call op1' = SLDT
action SLDT::bits $vw
  fail intercept(VMEXIT_LDTR_READ) when _vmcb_ca.intercept.READ_LDTR
  return zxt($vw, LDTR.sel)
```

Instruction LGDT

Loads the pseudo-descriptor specified by the source operand into the global descriptor table register (‘GDTR’). The pseudo-descriptor is a memory location containing the GDTR base and limit. In legacy and compatibility mode, the pseudo-descriptor is 6 bytes; in 64-bit mode, it is 10 bytes. *[cited from AMD volume 3]*

```
opcode "0F01h /010b" mem_ptr 16+$qd : call LGDT(op1)
action LGDT(base_limit::bits ($qd+16))
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_GDTR_WRITE) when _vmcb_ca.intercept.WRITE_GDTR
  let limit = base_limit[15:0]
  let base = base_limit[$qd+15:16]
  fail exception(xGP, 0000h) when $x64_mode and not canonical(zxt(64, base))
  let adjusted_base = if $x64_mode or $v <> 16 then base
                      else zxt($qd, base[23:0])
  write zxt(64, adjusted_base) to GDTR.base
  write limit to GDTR.limit
```

Instruction SGDT

Stores the global descriptor table register (GDTR) into the destination operand. In legacy and compatibility mode, the destination operand is 6 bytes; in 64-bit mode, it is 10 bytes. In all modes, operand-size prefixes are ignored. *[cited from AMD volume 3]*

```
opcode "0F01h /000b" mem_ptr 16+$qd : call SGDT
action SGDT::bits ($qd+16)
  fail intercept(VMEXIT_GDTR_READ) when _vmcb_ca.intercept.READ_GDTR
  return GDTR.base[$qd-1:0] ++ GDTR.limit
```

Instruction LIDT

Loads the pseudo-descriptor specified by the source operand into the interrupt descriptor table register (IDTR). The pseudo-descriptor is a memory location containing the IDTR base and limit. In legacy and compatibility mode, the pseudo-descriptor is six bytes; in 64-bit mode, it is 10 bytes. *[cited from AMD volume 3]*

```
opcode "0F01h /011b" mem_ptr 16+$qd : call LIDT(op1)
action LIDT(base_limit::bits ($qd+16))
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_IDTR_WRITE) when _vmcb_ca.intercept.WRITE_IDTR
  let limit = base_limit[15:0]
  let base = base_limit[$qd+15:16]
  fail exception(xGP, 0000h) when $x64_mode and not canonical(zxt(64, base))
  let adjusted_base = if $x64_mode or $v <> 16 then base
                      else zxt($qd, base[23:0])
  write zxt(64, adjusted_base) to IDTR.base
  write limit to IDTR.limit
```

Instruction SIDT

Stores the interrupt descriptor table register (IDTR) in the destination operand. In legacy and compatibility mode, the destination operand is 6 bytes; in 64-bit mode it is 10 bytes. In all modes, operand-size prefixes are ignored. *[cited from AMD volume 3]*

```
opcode "0F01h /001b" mem_ptr 16+$qd : call op1' = SGDT
action SIDT::bits ($qd+16)
  fail intercept(VMEXIT_IDTR_READ) when _vmcb_ca.intercept.READ_IDTR
  return IDTR.base[$qd-1:0] ++ IDTR.limit
```

APPENDIX

M

PROTECTION INSTRUCTIONS

Instruction ARPL (`when $protected_mode`)

Compares the requestor privilege level (RPL) fields of two segment selectors in the source and destination operands of the instruction. If the RPL field of the destination operand is less than the RPL field of the segment selector in the source register, then the zero flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the destination operand remains unchanged and the zero flag is cleared. *[cited from AMD volume 3]*

```
opcode "63h" reg_mem 16, reg 16 : call op1' = ARPL(op1, op2)
action ARPL(op1::Selector, op2::Selector)::Selector
  if op1.RPL < op2.RPL then
    write 1b to RFLAGS.ZF
    return op1 with [RPL = op2.RPL]
  else write 0b to RFLAGS.ZF
       return op1
```

Instruction LAR (`when $protected_mode`)

Loads the access rights from the segment descriptor specified by a 16-bit source register or memory operand into a specified 16-bit, 32-bit, or 64-bit general-purpose register and sets the zero (ZF) flag in the rFLAGS register if successful. LAR clears the zero flag if the descriptor is invalid for any reason. *[cited from AMD volume 3]*

```
opcode "0F02h" reg $v, reg_mem 16 : call op1' = LAR(op1, op2)
function can_access(sel::Selector, desc::Descriptor)::bit
  = if (sel & FFF8h) == 0000h or not desc.P then 0b
    elif isUserSegment(desc) then can_access_data(sel, desc)
    elif isSystemSegment(desc) then can_access_desc(sel, desc)
    else 0b
action LAR(op1::bits $v, sel::Selector)::bits $v
  call (desc, valid) = read_desc(sel)
  if valid and can_access(sel, desc) then
    write 1b to RFLAGS.ZF
    return zxt($v, desc.DPL)
  else write 0b to RFLAGS.ZF
```

```
        return op1
```

Instruction LSL (`when $protected_mode`)

Loads the segment limit from the segment descriptor specified by a 16-bit source register or memory operand into a specified 16-bit, 32-bit, or 64-bit general-purpose register and sets the zero (ZF) flag in the rFLAGS register if successful. LSL clears the zero flag if the descriptor is invalid for any reason. *[cited from AMD volume 3]*

```
opcode "0F03h" reg $v, reg_mem 16 : call op1' = LSL(op1, op2)
action LSL(op1::bits $v, sel::bits 16)::bits $v
  call (desc, valid) = read_desc(sel)
  if valid and can_access(sel, desc) and isSegment(desc) then
    write 1b to RFLAGS.ZF
    return zxt(64, (desc::Segment).limit)[$v-1:0]
  else write 0b to RFLAGS.ZF
       return op1
```

Instruction VERR (`when $protected_mode`)

Verifies whether a code or data segment specified by the segment selector in the 16-bit register or memory operand is readable from the current privilege level. The zero flag (ZF) is set to 1 if the specified segment is readable. Otherwise, ZF is cleared. *[cited from AMD volume 3]*

```
opcode "0F00h /010b" reg_mem 16 : call VERR(op1)
action VERR(sel::bits 16)::bits 64
  call (desc, valid) = read_desc(sel)
  let accessable = valid and can_access(sel, desc)
  let readable = isUserSegment(desc) and (desc::UserSegment).readable
  write (accessable and readable) to RFLAGS.ZF
```

Instruction VERW (`when $protected_mode`)

Verifies whether a data segment specified by the segment selector in the 16-bit register or memory operand is writable from the current privilege level. The zero flag (ZF) is set to 1 if the specified segment is writable. Otherwise, ZF is cleared. *[cited from AMD volume 3]*

```
opcode "0F00h /101b" reg_mem 16 : call VERW(op1)
action VERW(sel::bits 16)::bits 64
  call (desc, valid) = read_desc(sel)
  let accessable = valid and can_access(sel, desc)
  let writable = isDataSegment(desc) and (desc::DataSegment).writable
  write (accessable and writable) to RFLAGS.ZF
```

APPENDIX

N

CR AND MSR ACCESS INSTRUCTIONS

N.1 Control Register Access

Instruction MOV(CRn)

Moves the contents of a 32-bit or 64-bit general-purpose register to a control register. In 64-bit mode, the operand size is fixed at 64 bits without the need for a REX prefix. In non–64-bit mode, the operand size is fixed at 32 bits and the upper 32 bits of the destination are forced to 0. CR0 maintains the state of various control bits. CR2 and CR3 are used for page translation. CR4 holds various feature enable bits. CR8 is used to prioritize external interrupts. CR1, CR5, CR6, CR7, and CR9 through CR15 are all reserved and raise an undefined opcode exception (#UD) if referenced. CR8 can be read and written in 64-bit mode, using a REX prefix. CR8 can be read and written in all modes using a LOCK prefix instead of a REX prefix to specify the additional opcode bit. To verify whether the LOCK prefix can be used in this way, check the status of ECX bit 4 returned by CPUID function 8000_0001h. *[cited from AMD volume 3]*

```
opcode "0F20h" reg $qd, creg $qd
opcode "F0h 0F20h" reg $qd, creg8 $qd
opcode "0F22h" creg $qd, reg $qd
opcode "F0h 0F22h" creg8 $qd, reg $qd
let op1' = op2
```

For reading and writing the `creg` operand type, we use the `read_cr` and the `write_cr` actions. Reading is simple, given the operand width and the register index:

1. make sure that `CPL == 00b`,
2. check for the control register read intercept,
3. make sure that the index specifies an existing register.

4. return the appropiate element from the `CR` array.

```
action read_cr($w::{32, 64}, idx::bits 4)::bits $w
  fail exception(xGP, 0000h) when CPL <> 00b
  let $idx = nat(idx)::[0..15]
  fail intercept(VMEXIT_READ_CR0 + zxt(64, idx)) when _vmcb_ca.READ_CR[$idx]
  fail exception(xUD, 0000h) when idx <> 0h and idx <> 2h and idx <> 3h
                                  and idx <> 4h and idx <> 8h
  return CR[idx][$w-1:0]
```

Writing is more complex, because we need to check that the reserved bits in the new register value are set correctly and in some cases flush the TLB and the store buffer.

The `write_cr` action makes privilege level and intercept checks, and then invokes the specific `write_cr0`, `write_cr2`, etc. action.

```
action write_cr($w::{32, 64}, val::bits $w, idx::bits 4)
  fail exception(xGP, 0000h) when CPL <> 00b
  let $idx = nat(idx)::[0..15]
  fail intercept(VMEXIT_WRITE_CR0 + zxt(64, idx)) when _vmcb_ca.WRITE_CR[$idx]
  if idx == 0h then
    call write_cr0($w, val)
  elif idx == 2h then
    call write_cr2($w, val)
  elif idx == 3h then
    call write_cr3($w, val)
  elif idx == 4h then
    call write_cr4($w, val)
  elif idx == 8h then
    call write_cr8($w, val)
  else fail exception(xUD, 0000h)
```

Given the new `CR0` register value, the `write_cr0` action copies the read-only bits from the old value (using function `fixCR0`) and checks that all reserved bits have the correct value (using function `isCR0`). These functions are defined implicitly based on the layout of the `CR0` register. When the `MP` bit or the `TS` bit changes, an intercept might be raised depending on the `WRITE_CR0` bit in the guest control area. When the `PG` changes, all local entries in the TLB are flushed. When the `PG` changes from `0b` to `1b` and the long mode is enabled, then the long mode is activated.

```
action write_cr0($w::{32, 64}, val::bits $w)
    let cr0 = fixCR0(CR0[63:$w] ++ val, CR0)
    fail exception(xGP, 0000h) when not isCR0(cr0)
    let written = (cr0 with [MP = 0b, TS = 0b]) <> (CR0 with [MP = 0b, TS = 0b])
    fail intercept(VMEXIT_CR0_SEL_WRITE)
      when _vmcb_ca.intercept.WRITE_CR0 and written
    call flush_tlb_local(current_asid) when cr0.PG <> CR0.PG
    write 1b to EFER.LMA when EFER.LME and cr0.PG and not CR0.PG
    write cr0 to CR0
```

A write to the `CR2` simply updates the register without any side-effects.

```
action write_cr2($w::{32, 64}, val::bits $w)
    write val to CR2[$w-1:0]
```

A write to the CR3 flushes all local entries in the TLB.

```
action write_cr3($w::{32, 64}, val::bits $w)
    let cr3 = fixCR3(CR3[63:$w] ++ val, CR3)
    fail exception(xGP, 0000h) when not isCR3(cr3)
    call flush_tlb_local(current_asid)
    write cr3 to CR3
```

A write to the CR4 register flushes all entries in the TLB if the page-mode related bits are changed.

```
action write_cr4($w::{32, 64}, val::bits $w)
    let cr4 = fixCR4(CR4[63:$w] ++ val, CR4)
    fail exception(xGP, 0000h) when not isCR4(cr4)
    call flush_tlb_global(current_asid)
      when cr4.PAE <> CR4.PAE or cr4.PSE <> CR4.PSE or cr4.PGE <> CR4.PGE
    write cr4 to CR4
```

A write to the CR8 updates the _vmcb_ca.V_INTR.V_TPR in guest mode if virtual interrupt masking is enabled.

```
action write_cr8($w::{32, 64}, val::bits $w)
    let cr8 = fixCR8(CR8[63:$w] ++ val, CR8)
    fail exception(xGP, 0000h) when not isCR8(cr8)
    write cr8 to CR8
    if _guest and _vmcb_ca.V_INTR.V_INTR_MASKING then
      write cr8[3:0] ++ 0000b to _vmcb_ca.V_INTR.V_TPR
```

Instruction SMSW

Stores the lower bits of the machine status word (CR0). The target can be a 16-, 32-, or 64-bit register or a 16-bit memory operand. *[cited from AMD volume 3]*

```
opcode "0F01h /100b" reg_mem $vw : call op1' = SMSW
action SMSW::bits $vw
  fail intercept(VMEXIT_READ_CR0) when _vmcb_ca.READ_CR[0]
  return CR0[$vw-1:0]
```

Instruction LMSW

Loads the lower four bits of the 16-bit register or memory operand into bits 3–0 of the machine status word in register CR0. Only the protection enabled (PE), monitor coprocessor (MP), emulation (EM), and task switched (TS) bits of CR0 are modified. Additionally, LMSW can set CR0.PE, but cannot clear it. *[cited from AMD volume 3]*

```
opcode "0F01h /110b" reg_mem 16 : call LMSW(op1)
```

The LMSW action is similar to the write_cr0 action.

```
action LMSW(val::bits 16)
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_WRITE_CR0) when _vmcb_ca.WRITE_CR[0]
  let x = zxt(64, val)::CR0
  let cr0 = CR0 with [TS = x.TS, EM = x.EM, MP = x.MP, PE = (x.PE | CR0.PE)]
  let written = (cr0 with [MP = 0b, TS = 0b]) <> (CR0 with [MP = 0b, TS = 0b])
  fail intercept(VMEXIT_CR0_SEL_WRITE)
    when _vmcb_ca.intercept.WRITE_CR0 and written
  write cr0 to CR0
```

N.2 Model Specific Register Access

Instruction RDMSR

Loads the contents of a 64-bit model-specific register (MSR) specified in the ECX register into registers EDX:EAX. The EDX register receives the high-order 32 bits and the EAX register receives the low order bits. The RDMSR instruction ignores operand size; ECX always holds the MSR number, and EDX:EAX holds the data. If a model-specific register has fewer than 64 bits, the unimplemented bit positions loaded into the destination registers are undefined. This instruction must be executed at a privilege level of 0 or a general protection exception (#GP) will be raised. This exception is also generated if a reserved or unimplemented model-specific register is specified in ECX. *[cited from AMD volume 3]*

```
opcode "0F32h" : call RDMSR
action RDMSR
  fail exception(xGP, 0000h) when CPL <> 00b
  let idx = RCX[31:0]
  call check_msr_intercept(rd, idx)
  call val = read_msr(idx)
  write val[63:32] to RDX[31:0]
  write val[31:0] to RCX[63:32]
```

In guest mode, the `check_msr_intercept` action looks up the intercept bit in the MSR permission bitmap located at the `_vmcb_ca.MSRPM_BASE_PA`. When the bit is set, it raises an intercept.

```
action check_msr_intercept(rw::RW, idx::bits 32)
  if _guest and _vmcb_ca.intercept.MSR_PROT then
    let x = if rw == read then zero(64) else one(64)
    let $bit_idx = nat(idx[4:0]++x[0])::[0..63]
    let base = _vmcb_ca.MSRPM_BASE_PA[$pa-1:12] ++ 000h
    if idx < 00002000h then
      let addr = base + zxt($pa, idx[12:5]++000b)
      call map = pread(sys, 64, addr, WB)
      fail intercept1(VMEXIT_MSR, x) when map[$bit_idx]
    elif C0000000h <= idx and idx < C0002000h then
      let addr = base + zxt($pa, idx[12:5]++000b) + zxt($pa, 800h)
      call map = pread(sys, 64, addr, WB)
      fail intercept1(VMEXIT_MSR, x) when map[$bit_idx]
    elif C0002000h <= idx and idx < C0004000h then
      let addr = base + zxt($pa, idx[12:5]++000b) + zxt($pa, 1000h)
      call map = pread(sys, 64, addr, WB)
      fail intercept1(VMEXIT_MSR, x) when map[$bit_idx]
    else fail intercept1(VMEXIT_MSR, x)
```

As there are too many model-specific registers, we will not list the full `read_msr` action. Instead, we just define a scheme based on two registers and assume that other registers are added according to the scheme.

```
action read_msr(idx::bits 32)::bits 64
  if idx == iAPIC_BASE then return APIC_BASE
  elif idx == iEFER then return EFER
  else fail exception(xGP, 0000h)
```

Instruction WRMSR

Writes data to 64-bit model-specific registers (MSRs). These registers are widely used in performance- monitoring and debugging applications, as well as testability and program execution tracing. This instruction writes the contents of the EDX:EAX register pair into a 64-bit model-specific register specified in the ECX register. The 32 bits in the EDX register are mapped into the high-order bits of the model-specific register and the 32 bits in EAX form the low-order 32 bits. This instruction must be executed at a privilege level of 0 or a general protection fault #GP(0) will be raised. This exception is also generated if an attempt is made to specify a reserved or unimplemented model-specific register in ECX. *[cited from AMD volume 3]*

```
opcode "0F30h" : call WRMSR
action WRMSR
  fail exception(xGP, 0000h) when CPL <> 00b
  let idx = RCX[31:0]
  call check_msr_intercept(wr, idx)
  let val = RDX[31:0] ++ RAX[31:0]
  call write_msr(val, idx)
```

Refer to the description of the `RDMSR` instruction for the `check_msr_intercept` action. As we did for the `read_msr` action, we do not list the full `write_msr` action. Instead, we define a scheme based on two registers, and assume that other register are added according to the scheme.

```
action write_msr(val::bits 64, idx::bits 32)
  if idx == iAPIC_BASE then
    let x = fixAPIC_BASE(val, APIC_BASE)
    fail exception(xGP, 0000h) when not isAPIC_BASE(x)
    write x to APIC_BASE
  elif idx == iEFER then
    let x = fixEFER(val, EFER)
    fail exception(xGP, 0000h) when not isEFER(x)
    write x to EFER
  else fail exception(xGP, 0000h)
```

Instruction RDTSC

Loads the value of the processor's 64-bit time-stamp counter into registers EDX:EAX. The time-stamp counter (TSC) is contained in a 64-bit model-specific register (MSR). The processor sets the counter to 0 upon reset and increments the counter every clock cycle. INIT does not modify the TSC. The high-order 32 bits are loaded into EDX, and the low-order 32 bits are loaded into the EAX register. This instruction ignores operand size. When the time-stamp disable flag (TSD) in CR4 is set to 1, the RDTSC instruction can only be used at privilege level 0. If the TSD flag is 0, this instruction can be used at any privilege level. This instruction is not serializing. Therefore, there is no guarantee that all instructions have completed at the time the time-stamp counter is read. The behavior of the RDTSC instruction is implementation dependent. The TSC counts at a constant rate, but may be affected by power management events (such as frequency changes), depending on the processor implementation. If CPUID 8000_0007.edx[8] = 1, then the TSC rate is ensured to be invariant across all P-States, C-States, and stop-grant transitions (such as STPCLK Throttling); therefore, the TSC is suitable for use as a source of time. Consult the BIOS and kernel developer's guide for your AMD

processor implementation for information concerning the effect of power management on the TSC. *[cited from AMD volume 3]*

```
opcode "0F31h" : call RDTSC
action RDTSC
  fail exception(xGP, 0000h) when CPL <> 00b and CR4.TSD
  fail intercept(VMEXIT_RDTSC) when _vmcb_ca.intercept.RDTSC
  write TSC >> 32 to RDX
  write TSC & 00000000FFFFFFFFh to RAX
```

Instruction RDTSCP

Loads the value of the processor's 64-bit time-stamp counter into registers EDX:EAX, and loads the value of TSC_AUX into ECX. This instruction ignores operand size. The time-stamp counter is contained in a 64-bit model-specific register (MSR). The processor sets the counter to 0 upon reset and increments the counter every clock cycle. INIT does not modify the TSC. The high-order 32 bits are loaded into EDX, and the low-order 32 bits are loaded into the EAX register. The TSC_AUX value is contained in the low-order 32 bits of the TSC_AUX register (MSR address C000_0103h). This MSR is initialized by privileged software to any meaningful value, such as a processor ID, that software wants to associate with the returned TSC value. When the time-stamp disable flag (TSD) in CR4 is set to 1, the RDTSCP instruction can only be used at privilege level 0. If the TSD flag is 0, this instruction can be used at any privilege level. Unlike the RDTSC instruction, RDTSCP forces all older instructions to retire before reading the time- stamp counter. The behavior of the RDTSCP instruction is implementation dependent. The TSC counts at a constant rate, but may be affected by power management events (such as frequency changes), depending on the processor implementation. If CPUID 8000_0007.edx[8] = 1, then the TSC rate is ensured to be invariant across all P-States, C-States, and stop-grant transitions (such as STPCLK Throttling); therefore, the TSC is suitable for use as a source of time. Consult the BIOS and kernel developer's guide for your AMD processor implementation for information concerning the effect of power management on the TSC. *[cited from AMD volume 3]*

```
opcode "0F01h /111b mod 11b rm 001b" : call RDTSCP
action RDTSCP
  fail exception(xGP, 0000h) when CPL <> 00b and CR4.TSD
  fail intercept(VMEXIT_RDTSC) when _vmcb_ca.intercept.RDTSCP
  write TSC >> 32 to RDX
  write TSC & 00000000FFFFFFFFh to RAX
  write TSC_AUX & 00000000FFFFFFFFh to RCX
```

APPENDIX

O

MEMORY MANAGEMENT INSTRUCTIONS

O.1 TLB Invalidation

Instruction INVLPG

Invalidates the TLB entry that would be used for the 1-byte memory operand. This instruction invalidates the TLB entry, regardless of the G (Global) bit setting in the associated PDE or PTE entry and regardless of the page size (4 Kbytes, 2 Mbytes, or 4 Mbytes). It may invalidate any number of additional TLB entries, in addition to the targeted entry. *[cited from AMD volume 3]*

```
opcode "0F01h /111b" mem 8 : call INVLPG
action INVLPG
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_INVLPG) when _vmcb_ca.intercept.INVLPG
  let s = segment(iDS)
  let origin = segment_origin(s)
  let addr = zxt($va, effective_base(origin, SR[s].base)) + zxt($va, ea)
  call invlpg(addr, current_asid)
```

The `invlpg` action takes a virtual address and a TLB tag. It flushes all TLB entries that translate the given virtual address with the given tag. This action and the `current_asid` function are defined in section 13.4.

Instruction INVLPGA (`when $SVME`)

Invalidates the TLB mapping for a given virtual page and a given current_asid. The virtual address is specified in the implicit register operand rAX. The portion of RAX used to form the address is determined by the effective address size. The current_asid is taken from ECX. The INVLPGA instruction may invalidate any number of additional TLB entries, in addition to the targeted entry. The INVLPGA instruction is a serializing instruction and a privileged instruction. The current privilege level must be 0 to execute this instruction. *[cited from AMD volume 3]*

```
opcode "0F01h /011b mod 11b rm 111b" : call INVLPGA
action INVLPGA
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_INVLPGA) when _vmcb_ca.intercept.INVLPGA
  call invlpg(zxt($va, RAX[$oa-1:0]), RCX[31:0])
```

O.2 Memory Fences

All memory fences are modelled at the store buffer level (see Part I), therefore they are no-operations in out domain-specific language.

Instruction MFENCE

Acts as a barrier to force strong memory ordering (serialization) between load and store instructions preceding the MFENCE, and load and store instructions that follow the MFENCE. The processor may perform loads out of program order with respect to non-conflicting stores for certain memory types. The MFENCE instruction guarantees that the system completes all previous memory accesses before executing subsequent accesses. The MFENCE instruction is weakly-ordered with respect to data and instruction prefetches. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an MFENCE. *[cited from AMD volume 3]*

```
opcode "0FAEF0h" : call NOP
```

Instruction SFENCE

Acts as a barrier to force strong memory ordering (serialization) between store instructions preceding the SFENCE and store instructions that follow the SFENCE. Stores to differing memory types, or within the WC memory type, may become visible out of program order; the SFENCE instruction ensures that the system completes all previous stores in such a way that they are globally visible before executing subsequent stores. This includes emptying the store buffer and all write-combining buffers. The SFENCE instruction is weakly-ordered with respect to load instructions, data and instruction prefetches, and the LFENCE instruction. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an SFENCE. *[cited from AMD volume 3]*

```
opcode "0FAEF8h" : call NOP
```

Instruction LFENCE

Acts as a barrier to force strong memory ordering (serialization) between load instructions preceding the LFENCE and load instructions that follow the LFENCE. Loads from differing memory types may be performed out of order, in particular between WC/WC+ and other memory types. The LFENCE instruction assures that the system completes all previous loads before executing subsequent loads. The LFENCE instruction is weakly-ordered with respect to store instructions, data and instruction prefetches, and the SFENCE instruction. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an LFENCE. *[cited from AMD volume 3]*

```
opcode "0FAEE8h" : call NOP
```

O.3 Cache Invalidation

Instruction WBINVD

Writes all modified cache lines in the internal caches back to main memory and invalidates (flushes) internal caches. It then causes external caches to write back modified data to main memory; the external caches are subsequently invalidated. After invalidating internal caches, the processor proceeds immediately with the execution of the next instruction without waiting for external hardware to invalidate its caches. *[cited from AMD volume 3]*

```
opcode "0F09h" : call WBINVD
action WBINVD
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_WBINVD) when _vmcb_ca.intercept.WBINVD
  call invalidate_cache(1b)
```

The `invalidate_cache` action takes a flag that indicates whether to write back a dirty line before invalidation or not. The action is defined in section 13.4.

Instruction INVD

Invalidates internal caches (data cache, instruction cache, and on-chip L2 cache) and triggers a bus cycle that causes external caches to invalidate themselves as well. No data is written back to main memory from invalidating internal caches. After invalidating internal caches, the processor proceeds immediately with the execution of the next instruction without waiting for external hardware to invalidate its caches. This is a privileged instruction. The current privilege level (CPL) of a procedure invalidating the processor's internal caches must be 0. *[cited from AMD volume 3]*

```
opcode "0F08h" : call INVD
action INVD
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_INVD) when _vmcb_ca.intercept.INVD
  call invalidate_cache(0b)
```

Instruction CLFLUSH

Flushes the cache line specified by the mem8 linear-address. The instruction checks all levels of the cache hierarchy—internal caches and external caches—and invalidates the cache line in every cache in which it is found. If a cache contains a dirty copy of the cache line (that is, the cache line is in the modified or owned MOESI state), the line is written back to memory before it is invalidated. The instruction sets the cache-line MOESI state to invalid. The instruction also checks the physical address corresponding to the linear-address operand against the processor's write-combining buffers. If the write-combining buffer holds data intended for that physical address, the instruction writes the entire contents of the buffer to memory. This occurs even though the data is not cached in the cache hierarchy. In a multiprocessor system, the instruction checks the write-combining buffers only on the processor that executed the CLFLUSH instruction. *[cited from AMD volume 3]*

```
opcode "0FAEh /111b" mem 8 : call CLFLUSH
action CLFLUSH
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_INVD) when _vmcb_ca.intercept.INVD
```

```
let s = segment(iDS)
let origin = segment_origin(s)
call check_logical(read, origin, 8, SR[s], $oa, ea)
let addr = zxt($va, effective_base(origin, SR[s].base)) + zxt($va, ea)
call (paddr, memtype) = va_to_pa(read, origin, addr, CPL)
call flush_cache_line(paddr)
```

The `flush_cache_line` action is defined in section 13.4.

APPENDIX

P

VIRTUALIZATION INSTRUCTIONS

P.1 Run Guest

Instruction VMRUN

Starts execution of a guest instruction stream. The physical address of the virtual machine control block (VMCB) describing the guest is taken from the rAX register (the portion of RAX used to form the address is determined by the effective address size). VMRUN saves a subset of host processor state to the host state-save area specified by the physical address in the VM_HSAVE_PA MSR. VMRUN then loads guest processor state (and control information) from the VMCB at the physical address specified in rAX. The processor then executes guest instructions until one of several intercept events (specified in the VMCB) is triggered. When an intercept event occurs, the processor stores a snapshot of the guest state back into the VMCB, reloads the host state, and continues execution of host code at the instruction following the VMRUN instruction. *[cited from AMD volume 3]*

```
opcode "0F01h /011b mod 11b rm 000b" rax $oa : when $SVME
                      : call VMRUN(zxt(64, op1)[$pa-1:0])
action VMRUN(addr::bits $pa)
  fail exception(xGP, 0000h) when CPL <> 00b
  fail exception(xGP, 0000h) when addr[11:0] <> 000h
  fail intercept(VMEXIT_VMRUN) when $GUEST and _vmcb_ca.intercept.VMRUN
  call save_host(addr)
  call load_guest(addr)
  commit and chain check_guest_state
```

We will define the `save_host` and the `load_guest` actions later. The `check_guest_state` action goes over subset of the guest registers and checks whether their values are valid or not. After that the action carries out the TLB flush request, enables interrupts, and copies the interrupt shadow indicator from the guest control area to the `_intr_shadow`

register.

```
action check_guest_state
  fail intercept(VMEXIT_INVALID) when not check_registers
  fail intercept(VMEXIT_INVALID) when not _vmcb.intercept.VMRUN
  fail intercept(VMEXIT_INVALID) when _vmcb.intercept.GUEST_ASID == zero(32)
  if _vmcb_ca.TLB_CONTROL == 01h then
    call flush_tlb_all
  write 1b to GIF
  write _vmcb_ca.INTERRUPT_SHADOW to _intr_shadow
```

The `check_registers` function returns true if and only if the control registers, the `EFER` register, and the `CS` register specify a valid operating mode, and have reserved bits set to correct values.

```
function check_registers
  = if EFER.LME and CR0.PG and (not CR4.PAE or not CR0.PE)then 0b
    else if CS.attr.L and CS.attr.D then 0b
    else isCR0(CR0)
         and isCR3(CR3)
         and isCR4(CR4)
         and isEFER(EFER)
```

The `save_host` and the `load_guest` are straightforward: for each register in the host/guest state they issue a physical memory write/read. The only noteworthy thing is computation of register offsets. We use the `@` operator that returns the field offset in a layout measured in bits. To convert the bits into byte we use the following function:

```
function as_pa(offset::nat)::bits $pa = bits($pa, (offset / 8))
```

For discussion of the host and guest save state are refer to chapter 17.

```
action save_host(addr::bits $pa)
  call pwrite(sys, 16, ES.sel, addr + as_pa(@HOST_SSA.ES_sel), WB)
  call pwrite(sys, 16, ES.attr, addr + as_pa(@HOST_SSA.ES_attr), WB)
  call pwrite(sys, 32, ES.limit, addr + as_pa(@HOST_SSA.ES_limit), WB)
  call pwrite(sys, 64, ES.base, addr + as_pa(@HOST_SSA.ES_base), WB)
  call pwrite(sys, 16, CS.sel, addr + as_pa(@HOST_SSA.CS_sel), WB)
  call pwrite(sys, 16, CS.attr, addr + as_pa(@HOST_SSA.CS_attr), WB)
  call pwrite(sys, 32, CS.limit, addr + as_pa(@HOST_SSA.CS_limit), WB)
  call pwrite(sys, 64, CS.base, addr + as_pa(@HOST_SSA.CS_base), WB)
  call pwrite(sys, 16, SS.sel, addr + as_pa(@HOST_SSA.SS_sel), WB)
  call pwrite(sys, 16, SS.attr, addr + as_pa(@HOST_SSA.SS_attr), WB)
  call pwrite(sys, 32, SS.limit, addr + as_pa(@HOST_SSA.SS_limit), WB)
  call pwrite(sys, 64, SS.base, addr + as_pa(@HOST_SSA.SS_base), WB)
  call pwrite(sys, 16, DS.sel, addr + as_pa(@HOST_SSA.DS_sel), WB)
  call pwrite(sys, 16, DS.attr, addr + as_pa(@HOST_SSA.DS_attr), WB)
  call pwrite(sys, 32, DS.limit, addr + as_pa(@HOST_SSA.DS_limit), WB)
  call pwrite(sys, 64, DS.base, addr + as_pa(@HOST_SSA.DS_base), WB)
  call pwrite(sys, 16, GDTR.limit, addr + as_pa(@HOST_SSA.GDTR_limit), WB)
  call pwrite(sys, 64, GDTR.base, addr + as_pa(@HOST_SSA.GDTR_base), WB)
  call pwrite(sys, 16, IDTR.limit, addr + as_pa(@HOST_SSA.IDTR_limit), WB)
  call pwrite(sys, 64, IDTR.base, addr + as_pa(@HOST_SSA.IDTR_base), WB)
  call pwrite(sys, 64, EFER, addr + as_pa(@HOST_SSA.EFER), WB)
  call pwrite(sys, 64, CR4, addr + as_pa(@HOST_SSA.CR4), WB)
```

```
  call pwrite(sys, 64, CR3, addr + as_pa(@HOST_SSA.CR3), WB)
  call pwrite(sys, 64, CR0, addr + as_pa(@HOST_SSA.CR0), WB)
  call pwrite(sys, 64, RFLAGS, addr + as_pa(@HOST_SSA.RFLAGS), WB)
  call pwrite(sys, 64, RIP, addr + as_pa(@HOST_SSA.RIP), WB)
  call pwrite(sys, 64, RSP, addr + as_pa(@HOST_SSA.RSP), WB)
  call pwrite(sys, 64, RAX, addr + as_pa(@HOST_SSA.RAX), WB)
  call pwrite(sys, 64, PAT, addr + as_pa(@HOST_SSA.PAT), WB)

action load_guest(addr::bits $pa)
  call rcr = pread(sys, 16, addr + as_pa(@VMCB_CA.READ_CR), WB)
  call wcr = pread(sys, 16, addr + as_pa(@VMCB_CA.WRITE_CR), WB)
  call rdr = pread(sys, 16, addr + as_pa(@VMCB_CA.READ_DR), WB)
  call wdr = pread(sys, 16, addr + as_pa(@VMCB_CA.WRITE_DR), WB)
  call excp = pread(sys, 32, addr + as_pa(@VMCB_CA.EXCP), WB)
  call icpt = pread(sys, 64, addr + as_pa(@VMCB_CA.intercept), WB)
  call io = pread(sys, 64, addr + as_pa(@VMCB_CA.IOPM_BASE_PA), WB)
  call msr = pread(sys, 64, addr + as_pa(@VMCB_CA.MSRPM_BASE_PA), WB)
  call tsc = pread(sys, 64, addr + as_pa(@VMCB_CA.TSC_OFFSET), WB)
  call asid = pread(sys, 64, addr + as_pa(@VMCB_CA.GUEST_ASID), WB)
  call tlb = pread(sys, 8, addr + as_pa(@VMCB_CA.TLB_CONTROL), WB)
  call vintr = pread(sys, 64, addr + as_pa(@VMCB_CA.V_INTR), WB)
  call shw = pread(sys, 8, addr + as_pa(@VMCB_CA.INTERRUPT_SHADOW), WB)
  call np = pread(sys, 64, addr + as_pa(@VMCB_CA.NESTED_PAGING), WB)
  call inj = pread(sys, 64, addr + as_pa(@VMCB_CA.EVENTINJ), WB)
  call ncr3 = pread(sys, 64, addr + as_pa(@VMCB_CA.N_CR3), WB)
  call lbr = pread(sys, 64, addr + as_pa(@VMCB_CA.LBR_VE), WB)
  let ssa = addr + zxt($pa, 400h)
  call es_sel = pread(sys, 16, ssa + as_pa(@VMCB_SSA.ES_sel), WB)
  call es_attr = pread(sys, 16, ssa + as_pa(@VMCB_SSA.ES_attr), WB)
  call es_lim = pread(sys, 32, ssa + as_pa(@VMCB_SSA.ES_limit), WB)
  call es_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.ES_base), WB)
  call cs_sel = pread(sys, 16, ssa + as_pa(@VMCB_SSA.CS_sel), WB)
  call cs_attr = pread(sys, 16, ssa + as_pa(@VMCB_SSA.CS_attr), WB)
  call cs_lim = pread(sys, 32, ssa + as_pa(@VMCB_SSA.CS_limit), WB)
  call cs_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.CS_base), WB)
  call ss_sel = pread(sys, 16, ssa + as_pa(@VMCB_SSA.SS_sel), WB)
  call ss_attr = pread(sys, 16, ssa + as_pa(@VMCB_SSA.SS_attr), WB)
  call ss_lim = pread(sys, 32, ssa + as_pa(@VMCB_SSA.SS_limit), WB)
  call ss_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.SS_base), WB)
  call ds_sel = pread(sys, 16, ssa + as_pa(@VMCB_SSA.DS_sel), WB)
  call ds_attr = pread(sys, 16, ssa + as_pa(@VMCB_SSA.DS_attr), WB)
  call ds_lim = pread(sys, 32, ssa + as_pa(@VMCB_SSA.DS_limit), WB)
  call ds_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.DS_base), WB)
  call gdtr_lim = pread(sys, 16, ssa + as_pa(@VMCB_SSA.GDTR_limit), WB)
  call gdtr_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.GDTR_base), WB)
  call idtr_lim = pread(sys, 16, ssa + as_pa(@VMCB_SSA.IDTR_limit), WB)
  call idtr_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.IDTR_base), WB)
  call cpl = pread(sys, 8, ssa + as_pa(@VMCB_SSA.CPL), WB)
  call efer = pread(sys, 64, ssa + as_pa(@VMCB_SSA.EFER), WB)
  call cr4 = pread(sys, 64, ssa + as_pa(@VMCB_SSA.CR4), WB)
  call cr3 = pread(sys, 64, ssa + as_pa(@VMCB_SSA.CR3), WB)
  call cr0 = pread(sys, 64, ssa + as_pa(@VMCB_SSA.CR0), WB)
  call dr7 = pread(sys, 64, ssa + as_pa(@VMCB_SSA.DR7), WB)
```

```
call dr6 = pread(sys, 64, ssa + as_pa(@VMCB_SSA.DR6), WB)
call flags = pread(sys, 64, ssa + as_pa(@VMCB_SSA.RFLAGS), WB)
call rip = pread(sys, 64, ssa + as_pa(@VMCB_SSA.RIP), WB)
call rsp = pread(sys, 64, ssa + as_pa(@VMCB_SSA.RSP), WB)
call rax = pread(sys, 64, ssa + as_pa(@VMCB_SSA.RAX), WB)
call pat = pread(sys, 64, ssa + as_pa(@VMCB_SSA.G_PAT), WB)
write rcr to _vmcb_ca.READ_CR
write wcr to _vmcb_ca.WRITE_CR
write rdr to _vmcb_ca.READ_DR
write wdr to _vmcb_ca.WRITE_DR
write excp to _vmcb_ca.EXCP
write icpt to _vmcb_ca.intercept
write io to _vmcb_ca.IOPM_BASE_PA
write msr to _vmcb_ca.MSRPM_BASE_PA
write tsc to _vmcb_ca.TSC_OFFSET
write asid[31:0] to _vmcb_ca.GUEST_ASID
write tlb to _vmcb_ca.TLB_CONTROL
write vintr to _vmcb_ca.V_INTR
write shw[0] to _vmcb_ca.INTERRUPT_SHADOW
write np to _vmcb_ca.NESTED_PAGING
write inj to _vmcb_ca.EVENTINJ
write ncr3 to _vmcb_ca.N_CR3
write lbr to _vmcb_ca.LBR_VE
write es_sel to ES.sel
write es_attr to ES.attr
write es_lim to ES.limit
write es_base to ES.base
write cs_sel to CS.sel
write cs_attr to CS.attr
write cs_lim to CS.limit
write cs_base to CS.base
write ss_sel to SS.sel
write ss_attr to SS.attr
write ss_lim to SS.limit
write ss_base to SS.base
write ds_sel to DS.sel
write ds_attr to DS.attr
write ds_lim to DS.limit
write ds_base to DS.base
write gdtr_lim to GDTR.limit
write gdtr_base to GDTR.base
write idtr_lim to IDTR.limit
write idtr_base to IDTR.base
write cpl[1:0] to CPL
write efer to EFER
write cr4 to CR4
write cr3 to CR3
write cr0 to CR0
write dr7 to DR7
write dr6 to DR6
write flags to RFLAGS
write rip to RIP
write rsp to RSP
```

```
  write rax to RAX
  write pat to PAT
  write 1b to _guest
```

P.2 Exit Guest

Instruction VMMCALL

Provides a mechanism for a guest to explicitly communicate with the VMM by generating a #VMEXIT. A non-intercepted VMMCALL unconditionally raises a #UD exception. VMMCALL is not restricted to either protected mode or CPL zero. *[cited from AMD volume 3]*

```
opcode "0F01h /011b mod 11b rm 001b" : when $SVME and $GUEST : call VMMCALL
action VMMCALL
  fail exception(xUD, 0000h) when not _vmcb_ca.intercept.VMMCALL
  fail intercept(VMEXIT_VMMCALL)
action VMEXIT(i::Intercept)
   call save_guest( _vmcb_addr[$pa-1:0], i)
   call load_host(VM_HSAVE_PA[$pa-1:0])
   write 0b to GIF
   write 00b to CPL
   write 0b to _guest
   commit and chain check_host_state
```

The `check_host_state` action makes sure that a subset of the host registers have valid values using the `check_registers` function defined in the previous section.

```
action check_host_state
  fail exception(xGP, 0000h) when not check_registers
action save_guest(addr::bits $pa, i::Intercept)
  call pwrite(sys, 64, _vmcb_ca.V_INTR, addr + as_pa(@VMCB_CA.V_INTR), WB)
  call pwrite(sys, 64, i.exitcode, addr + as_pa(@VMCB_CA.EXITCODE), WB)
  call pwrite(sys, 64, i.exitinfo1, addr + as_pa(@VMCB_CA.EXITINFO1), WB)
  call pwrite(sys, 64, i.exitinfo2, addr + as_pa(@VMCB_CA.EXITINFO2), WB)
  call pwrite(sys, 64, i.exitintinfo, addr + as_pa(@VMCB_CA.EXITINTINFO), WB)
  let ssa = addr + zxt($pa, 400h)
  call pwrite(sys, 16, ES.sel, ssa + as_pa(@VMCB_SSA.ES_sel), WB)
  call pwrite(sys, 16, ES.attr, ssa + as_pa(@VMCB_SSA.ES_attr), WB)
  call pwrite(sys, 32, ES.limit, ssa + as_pa(@VMCB_SSA.ES_limit), WB)
  call pwrite(sys, 64, ES.base, ssa + as_pa(@VMCB_SSA.ES_base), WB)
  call pwrite(sys, 16, CS.sel, ssa + as_pa(@VMCB_SSA.CS_sel), WB)
  call pwrite(sys, 16, CS.attr, ssa + as_pa(@VMCB_SSA.CS_attr), WB)
  call pwrite(sys, 32, CS.limit, ssa + as_pa(@VMCB_SSA.CS_limit), WB)
  call pwrite(sys, 64, CS.base, ssa + as_pa(@VMCB_SSA.CS_base), WB)
  call pwrite(sys, 16, SS.sel, ssa + as_pa(@VMCB_SSA.SS_sel), WB)
  call pwrite(sys, 16, SS.attr, ssa + as_pa(@VMCB_SSA.SS_attr), WB)
  call pwrite(sys, 32, SS.limit, ssa + as_pa(@VMCB_SSA.SS_limit), WB)
  call pwrite(sys, 64, SS.base, ssa + as_pa(@VMCB_SSA.SS_base), WB)
  call pwrite(sys, 16, DS.sel, ssa + as_pa(@VMCB_SSA.DS_sel), WB)
  call pwrite(sys, 16, DS.attr, ssa + as_pa(@VMCB_SSA.DS_attr), WB)
  call pwrite(sys, 32, DS.limit, ssa + as_pa(@VMCB_SSA.DS_limit), WB)
  call pwrite(sys, 64, DS.base, ssa + as_pa(@VMCB_SSA.DS_base), WB)
```

```
  call pwrite(sys, 16, GDTR.limit, ssa + as_pa(@VMCB_SSA.GDTR_limit), WB)
  call pwrite(sys, 64, GDTR.base, ssa + as_pa(@VMCB_SSA.GDTR_base), WB)
  call pwrite(sys, 16, IDTR.limit, ssa + as_pa(@VMCB_SSA.IDTR_limit), WB)
  call pwrite(sys, 64, IDTR.base, ssa + as_pa(@VMCB_SSA.IDTR_base), WB)
  call pwrite(sys, 8, zxt(8, CPL), ssa + as_pa(@VMCB_SSA.CPL), WB)
  call pwrite(sys, 64, EFER, ssa + as_pa(@VMCB_SSA.EFER), WB)
  call pwrite(sys, 64, CR0, ssa + as_pa(@VMCB_SSA.CR0), WB)
  call pwrite(sys, 64, CR3, ssa + as_pa(@VMCB_SSA.CR3), WB)
  call pwrite(sys, 64, CR4, ssa + as_pa(@VMCB_SSA.CR4), WB)
  call pwrite(sys, 64, DR6, ssa + as_pa(@VMCB_SSA.DR6), WB)
  call pwrite(sys, 64, DR7, ssa + as_pa(@VMCB_SSA.DR7), WB)
  call pwrite(sys, 64, RFLAGS, ssa + as_pa(@VMCB_SSA.RFLAGS), WB)
  call pwrite(sys, 64, RIP, ssa + as_pa(@VMCB_SSA.RIP), WB)
  call pwrite(sys, 64, RSP, ssa + as_pa(@VMCB_SSA.RSP), WB)
  call pwrite(sys, 64, RAX, ssa + as_pa(@VMCB_SSA.RAX), WB)
  call pwrite(sys, 64, PAT, ssa + as_pa(@VMCB_SSA.G_PAT), WB)

action load_host(addr::bits $pa)
  call es_sel = pread(sys, 16, addr + as_pa(@HOST_SSA.ES_sel), WB)
  call es_attr = pread(sys, 16, ssa + as_pa(@HOST_SSA.ES_attr), WB)
  call es_lim = pread(sys, 32, ssa + as_pa(@HOST_SSA.ES_limit), WB)
  call es_base = pread(sys, 64, ssa + as_pa(@HOST_SSA.ES_base), WB)
  call cs_sel = pread(sys, 16, addr + as_pa(@HOST_SSA.CS_sel), WB)
  call cs_attr = pread(sys, 16, ssa + as_pa(@HOST_SSA.CS_attr), WB)
  call cs_lim = pread(sys, 32, ssa + as_pa(@HOST_SSA.CS_limit), WB)
  call cs_base = pread(sys, 64, ssa + as_pa(@HOST_SSA.CS_base), WB)
  call ss_sel = pread(sys, 16, addr + as_pa(@HOST_SSA.SS_sel), WB)
  call ss_attr = pread(sys, 16, ssa + as_pa(@HOST_SSA.SS_attr), WB)
  call ss_lim = pread(sys, 32, ssa + as_pa(@HOST_SSA.SS_limit), WB)
  call ss_base = pread(sys, 64, ssa + as_pa(@HOST_SSA.SS_base), WB)
  call ds_sel = pread(sys, 16, addr + as_pa(@HOST_SSA.DS_sel), WB)
  call ds_attr = pread(sys, 16, ssa + as_pa(@HOST_SSA.DS_attr), WB)
  call ds_lim = pread(sys, 32, ssa + as_pa(@HOST_SSA.DS_limit), WB)
  call ds_base = pread(sys, 64, ssa + as_pa(@HOST_SSA.DS_base), WB)
  call gdtr_limit = pread(sys, 16, addr + as_pa(@HOST_SSA.GDTR_limit), WB)
  call gdtr_base = pread(sys, 64, addr + as_pa(@HOST_SSA.GDTR_base), WB)
  call idtr_limit = pread(sys, 16, addr + as_pa(@HOST_SSA.IDTR_limit), WB)
  call idtr_base = pread(sys, 64, addr + as_pa(@HOST_SSA.IDTR_base), WB)
  call efer = pread(sys, 64, addr + as_pa(@HOST_SSA.EFER), WB)
  call cr4 = pread(sys, 64, addr + as_pa(@HOST_SSA.CR4), WB)
  call cr3 = pread(sys, 64, addr + as_pa(@HOST_SSA.CR3), WB)
  call cr0 = pread(sys, 64, addr + as_pa(@HOST_SSA.CR0), WB)
  call flags = pread(sys, 64, addr + as_pa(@HOST_SSA.RFLAGS), WB)
  call rip = pread(sys, 64, addr + as_pa(@HOST_SSA.RIP), WB)
  call rsp = pread(sys, 64, addr + as_pa(@HOST_SSA.RSP), WB)
  write es_sel to ES.sel
  write es_attr to ES.attr
  write es_lim to ES.limit
  write es_base to ES.base
  write cs_sel to CS.sel
  write cs_attr to CS.attr
  write cs_lim to CS.limit
  write cs_base to CS.base
```

```
  write ss_sel to SS.sel
  write ss_attr to SS.attr
  write ss_lim to SS.limit
  write ss_base to SS.base
  write ds_sel to DS.sel
  write ds_attr to DS.attr
  write ds_lim to DS.limit
  write ds_base to DS.base
  write gdtr_limit to GDTR.limit
  write idtr_limit to IDTR.limit
  write gdtr_base to GDTR.base
  write idtr_base to IDTR.base
  write efer to EFER
  write cr4 to CR4
  write cr3 to CR3
  write (cr0::CR0) with [PE = 1b] to CR0
  write (flags::Flags) with [VM = 0b] to RFLAGS
  write rip to RIP
  write rsp to RSP
```

P.3 Save and Restore Guest Extended State

Instruction VMLOAD

Loads a subset of processor state from the VMCB specified by the physical address in the rAX register. The portion of RAX used to form the address is determined by the effective address size. The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state. *[cited from AMD volume 3]*

```
opcode "0F01h /011b mod 11b rm 010b" rax $oa : when $SVME
                     : call VMLOAD(zxt(64, op1)[$pa-1:0])
action VMLOAD(addr::bits $pa)
  let ssa = addr + zxt($pa, 400h)
  call load_guest_ext(ssa)
action load_guest_ext(ssa::bits $pa)
  call fs_sel = pread(sys, 16, ssa + as_pa(@VMCB_SSA.FS_sel), WB)
  call fs_attr = pread(sys, 16, ssa + as_pa(@VMCB_SSA.FS_attr), WB)
  call fs_lim = pread(sys, 32, ssa + as_pa(@VMCB_SSA.FS_limit), WB)
  call fs_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.FS_base), WB)
  call gs_sel = pread(sys, 16, ssa + as_pa(@VMCB_SSA.GS_sel), WB)
  call gs_attr = pread(sys, 16, ssa + as_pa(@VMCB_SSA.GS_attr), WB)
  call gs_lim = pread(sys, 32, ssa + as_pa(@VMCB_SSA.GS_limit), WB)
  call gs_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.GS_base), WB)
  call tr_sel = pread(sys, 16, ssa + as_pa(@VMCB_SSA.TR_sel), WB)
  call tr_attr = pread(sys, 16, ssa + as_pa(@VMCB_SSA.TR_attr), WB)
  call tr_lim = pread(sys, 32, ssa + as_pa(@VMCB_SSA.TR_limit), WB)
  call tr_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.TR_base), WB)
  call ldtr_sel = pread(sys, 16, ssa + as_pa(@VMCB_SSA.LDTR_sel), WB)
  call ldtr_attr = pread(sys, 16, ssa + as_pa(@VMCB_SSA.LDTR_attr), WB)
  call ldtr_lim = pread(sys, 32, ssa + as_pa(@VMCB_SSA.LDTR_limit), WB)
```

```
  call ldtr_base = pread(sys, 64, ssa + as_pa(@VMCB_SSA.LDTR_base), WB)
  call kgsb = pread(sys, 64, ssa + as_pa(@VMCB_SSA.KernelGSBase), WB)
  call star = pread(sys, 64, ssa + as_pa(@VMCB_SSA.STAR), WB)
  call lstar = pread(sys, 64, ssa + as_pa(@VMCB_SSA.LSTAR), WB)
  call cstar = pread(sys, 64, ssa + as_pa(@VMCB_SSA.CSTAR), WB)
  call sfmask = pread(sys, 64, ssa + as_pa(@VMCB_SSA.SFMASK), WB)
  call syscs = pread(sys, 64, ssa + as_pa(@VMCB_SSA.SYSENTER_CS), WB)
  call sysesp = pread(sys, 64, ssa + as_pa(@VMCB_SSA.SYSENTER_ESP), WB)
  call syseip = pread(sys, 64, ssa + as_pa(@VMCB_SSA.SYSENTER_EIP), WB)
  write fs_sel to FS.sel
  write fs_attr to FS.attr
  write fs_lim to FS.limit
  write fs_base to FS.base
  write gs_sel to GS.sel
  write gs_attr to GS.attr
  write gs_lim to GS.limit
  write gs_base to GS.base
  write tr_sel to TR.sel
  write tr_attr to TR.attr
  write tr_lim to TR.limit
  write tr_base to TR.base
  write ldtr_sel to LDTR.sel
  write ldtr_attr to LDTR.attr
  write ldtr_lim to LDTR.limit
  write ldtr_base to LDTR.base
  write kgsb to KernelGSBase
  write star to STAR
  write lstar to LSTAR
  write cstar to CSTAR
  write sfmask to SFMASK
  write syscs to SYSENTER_CS
  write sysesp to SYSENTER_ESP
  write syseip to SYSENTER_EIP
```

Instruction VMSAVE

Stores a subset of the processor state into the VMCB specified by the physical address in the rAX register (the portion of RAX used to form the address is determined by the effective address size). The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state. *[cited from AMD volume 3]*

```
opcode "0F01h /011b mod 11b rm 011b" rax $oa : when $SVME
                     : call VMSAVE(zxt(64, op1)[$pa-1:0])
action VMSAVE(addr::bits $pa)
  let ssa = addr + zxt($pa, 400h)
  call save_guest_ext(ssa)
action save_guest_ext(ssa::bits $pa)
  call pwrite(sys, 16, FS.sel, ssa + as_pa(@VMCB_SSA.FS_sel), WB)
  call pwrite(sys, 16, FS.attr, ssa + as_pa(@VMCB_SSA.FS_attr), WB)
  call pwrite(sys, 32, FS.limit, ssa + as_pa(@VMCB_SSA.FS_limit), WB)
  call pwrite(sys, 64, FS.base, ssa + as_pa(@VMCB_SSA.FS_base), WB)
  call pwrite(sys, 16, GS.sel, ssa + as_pa(@VMCB_SSA.GS_sel), WB)
```

```
  call pwrite(sys, 16, GS.attr, ssa + as_pa(@VMCB_SSA.GS_attr), WB)
  call pwrite(sys, 32, GS.limit, ssa + as_pa(@VMCB_SSA.GS_limit), WB)
  call pwrite(sys, 64, GS.base, ssa + as_pa(@VMCB_SSA.GS_base), WB)
  call pwrite(sys, 16, TR.sel, ssa + as_pa(@VMCB_SSA.TR_sel), WB)
  call pwrite(sys, 16, TR.attr, ssa + as_pa(@VMCB_SSA.TR_attr), WB)
  call pwrite(sys, 32, TR.limit, ssa + as_pa(@VMCB_SSA.TR_limit), WB)
  call pwrite(sys, 64, TR.base, ssa + as_pa(@VMCB_SSA.TR_base), WB)
  call pwrite(sys, 16, LDTR.sel, ssa + as_pa(@VMCB_SSA.LDTR_sel), WB)
  call pwrite(sys, 16, LDTR.attr, ssa + as_pa(@VMCB_SSA.LDTR_attr), WB)
  call pwrite(sys, 32, LDTR.limit, ssa + as_pa(@VMCB_SSA.LDTR_limit), WB)
  call pwrite(sys, 64, LDTR.base, ssa + as_pa(@VMCB_SSA.LDTR_base), WB)
  call pwrite(sys, 64, KernelGSBase, ssa + as_pa(@VMCB_SSA.KernelGSBase), WB)
  call pwrite(sys, 64, STAR, ssa + as_pa(@VMCB_SSA.STAR), WB)
  call pwrite(sys, 64, LSTAR, ssa + as_pa(@VMCB_SSA.LSTAR), WB)
  call pwrite(sys, 64, CSTAR, ssa + as_pa(@VMCB_SSA.CSTAR), WB)
  call pwrite(sys, 64, SFMASK, ssa + as_pa(@VMCB_SSA.SFMASK), WB)
  call pwrite(sys, 64, SYSENTER_CS, ssa + as_pa(@VMCB_SSA.SYSENTER_CS), WB)
  call pwrite(sys, 64, SYSENTER_ESP, ssa + as_pa(@VMCB_SSA.SYSENTER_ESP), WB)
  call pwrite(sys, 64, SYSENTER_EIP, ssa + as_pa(@VMCB_SSA.SYSENTER_EIP), WB)
```

Instruction CLGI (`when $SVME`)

Clears the global interrupt flag (GIF). While GIF is zero, all external interrupts are disabled. *[cited from AMD volume 3]*

```
opcode "0F01h /011b mod 11b rm 101b" : call CLGI
action CLGI
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_CLGI) when $GUEST and _vmcb_ca.intercept.CLGI
  write 0b to GIF
```

Instruction STGI (`when $SVME`)

Sets the global interrupt flag (GIF) to 1. While GIF is zero, all external interrupts are disabled. *[cited from AMD volume 3]*

```
opcode "0F01h /011b mod 11b rm 100b" : call STGI
action STGI
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_STGI) when $GUEST and _vmcb_ca.intercept.STGI
  write 1b to GIF
```

P.4 Exit Codes

As we mentioned in chapter 17, each intercept is associated with an exit code, which is provided to the host via the `EXITCODE` field of the guest control area. There are the following exit codes:

```
set ExitCode = {VMEXIT_READ_CR0 = 0000000000000000h,
                VMEXIT_READ_CR1 = 0000000000000001h,
                VMEXIT_READ_CR2 = 0000000000000002h,
                VMEXIT_READ_CR3 = 0000000000000003h,
                VMEXIT_READ_CR4 = 0000000000000004h,
                VMEXIT_READ_CR5 = 0000000000000005h,
```

```
VMEXIT_READ_CR6 = 0000000000000006h,
VMEXIT_READ_CR7 = 0000000000000007h,
VMEXIT_READ_CR8 = 0000000000000008h,
VMEXIT_READ_CR9 = 0000000000000009h,
VMEXIT_READ_CR10 = 000000000000000Ah,
VMEXIT_READ_CR11 = 000000000000000Bh,
VMEXIT_READ_CR12 = 000000000000000Ch,
VMEXIT_READ_CR13 = 000000000000000Dh,
VMEXIT_READ_CR14 = 000000000000000Eh,
VMEXIT_READ_CR15 = 000000000000000Fh,
VMEXIT_WRITE_CR0 = 0000000000000010h,
VMEXIT_WRITE_CR1 = 0000000000000011h,
VMEXIT_WRITE_CR2 = 0000000000000012h,
VMEXIT_WRITE_CR3 = 0000000000000013h,
VMEXIT_WRITE_CR4 = 0000000000000014h,
VMEXIT_WRITE_CR5 = 0000000000000015h,
VMEXIT_WRITE_CR6 = 0000000000000016h,
VMEXIT_WRITE_CR7 = 0000000000000017h,
VMEXIT_WRITE_CR8 = 0000000000000018h,
VMEXIT_WRITE_CR9 = 0000000000000019h,
VMEXIT_WRITE_CR10 = 000000000000001Ah,
VMEXIT_WRITE_CR11 = 000000000000001Bh,
VMEXIT_WRITE_CR12 = 000000000000001Ch,
VMEXIT_WRITE_CR13 = 000000000000001Dh,
VMEXIT_WRITE_CR14 = 000000000000001Eh,
VMEXIT_WRITE_CR15 = 000000000000001Fh,
VMEXIT_READ_DR0 = 0000000000000020h,
VMEXIT_READ_DR1 = 0000000000000021h,
VMEXIT_READ_DR2 = 0000000000000022h,
VMEXIT_READ_DR3 = 0000000000000023h,
VMEXIT_READ_DR4 = 0000000000000024h,
VMEXIT_READ_DR5 = 0000000000000025h,
VMEXIT_READ_DR6 = 0000000000000026h,
VMEXIT_READ_DR7 = 0000000000000027h,
VMEXIT_READ_DR8 = 0000000000000028h,
VMEXIT_READ_DR9 = 0000000000000029h,
VMEXIT_READ_DR10 = 000000000000002Ah,
VMEXIT_READ_DR11 = 000000000000002Bh,
VMEXIT_READ_DR12 = 000000000000002Ch,
VMEXIT_READ_DR13 = 000000000000002Dh,
VMEXIT_READ_DR14 = 000000000000002Eh,
VMEXIT_READ_DR15 = 000000000000002Fh,
VMEXIT_WRITE_DR0 = 0000000000000030h,
VMEXIT_WRITE_DR1 = 0000000000000031h,
VMEXIT_WRITE_DR2 = 0000000000000032h,
VMEXIT_WRITE_DR3 = 0000000000000033h,
VMEXIT_WRITE_DR4 = 0000000000000034h,
VMEXIT_WRITE_DR5 = 0000000000000035h,
VMEXIT_WRITE_DR6 = 0000000000000036h,
VMEXIT_WRITE_DR7 = 0000000000000037h,
VMEXIT_WRITE_DR8 = 0000000000000038h,
VMEXIT_WRITE_DR9 = 0000000000000039h,
VMEXIT_WRITE_DR10 = 000000000000003Ah,
```

```
VMEXIT_WRITE_DR11 = 000000000000003Bh,
VMEXIT_WRITE_DR12 = 000000000000003Ch,
VMEXIT_WRITE_DR13 = 000000000000003Dh,
VMEXIT_WRITE_DR14 = 000000000000003Eh,
VMEXIT_WRITE_DR15 = 000000000000003Fh,
VMEXIT_EXCP0 = 0000000000000040h,
VMEXIT_EXCP1 = 0000000000000041h,
VMEXIT_EXCP2 = 0000000000000042h,
VMEXIT_EXCP3 = 0000000000000043h,
VMEXIT_EXCP4 = 0000000000000044h,
VMEXIT_EXCP5 = 0000000000000045h,
VMEXIT_EXCP6 = 0000000000000046h,
VMEXIT_EXCP7 = 0000000000000047h,
VMEXIT_EXCP8 = 0000000000000048h,
VMEXIT_EXCP9 = 0000000000000049h,
VMEXIT_EXCP10 = 000000000000004Ah,
VMEXIT_EXCP11 = 000000000000004Bh,
VMEXIT_EXCP12 = 000000000000004Ch,
VMEXIT_EXCP13 = 000000000000004Dh,
VMEXIT_EXCP14 = 000000000000004Eh,
VMEXIT_EXCP15 = 000000000000004Fh,
VMEXIT_EXCP16 = 0000000000000050h,
VMEXIT_EXCP17 = 0000000000000051h,
VMEXIT_EXCP18 = 0000000000000052h,
VMEXIT_EXCP19 = 0000000000000053h,
VMEXIT_EXCP20 = 0000000000000054h,
VMEXIT_EXCP21 = 0000000000000055h,
VMEXIT_EXCP22 = 0000000000000056h,
VMEXIT_EXCP23 = 0000000000000057h,
VMEXIT_EXCP24 = 0000000000000058h,
VMEXIT_EXCP25 = 0000000000000059h,
VMEXIT_EXCP26 = 000000000000005Ah,
VMEXIT_EXCP27 = 000000000000005Bh,
VMEXIT_EXCP28 = 000000000000005Ch,
VMEXIT_EXCP29 = 000000000000005Dh,
VMEXIT_EXCP30 = 000000000000005Eh,
VMEXIT_EXCP31 = 000000000000005Fh,
VMEXIT_INTR = 0000000000000060h,
VMEXIT_NMI = 0000000000000061h,
VMEXIT_SMI = 0000000000000062h,
VMEXIT_INIT = 0000000000000063h,
VMEXIT_VINTR = 0000000000000064h,
VMEXIT_CR0_SEL_WRITE = 0000000000000065h,
VMEXIT_IDTR_READ = 0000000000000066h,
VMEXIT_GDTR_READ = 0000000000000067h,
VMEXIT_LDTR_READ = 0000000000000068h,
VMEXIT_TR_READ = 0000000000000069h,
VMEXIT_IDTR_WRITE = 000000000000006Ah,
VMEXIT_GDTR_WRITE = 000000000000006Bh,
VMEXIT_LDTR_WRITE = 000000000000006Ch,
VMEXIT_TR_WRITE = 000000000000006Dh,
VMEXIT_RDTSC = 000000000000006Eh,
VMEXIT_RDPMC = 000000000000006Fh,
```

```
VMEXIT_PUSHF = 0000000000000070h,
VMEXIT_POPF = 0000000000000071h,
VMEXIT_CPUID = 0000000000000072h,
VMEXIT_RSM = 0000000000000073h,
VMEXIT_IRET = 0000000000000074h,
VMEXIT_SWINT = 0000000000000075h,
VMEXIT_INVD = 0000000000000076h,
VMEXIT_PAUSE = 0000000000000077h,
VMEXIT_HLT = 0000000000000078h,
VMEXIT_INVLPG = 0000000000000079h,
VMEXIT_INVLPGA = 000000000000007Ah,
VMEXIT_IOIO = 000000000000007Bh,
VMEXIT_MSR = 000000000000007Ch,
VMEXIT_TASK_SWITCH = 000000000000007Dh,
VMEXIT_FERR_FREEZE = 000000000000007Eh,
VMEXIT_SHUTDOWN = 000000000000007Fh,
VMEXIT_VMRUN = 0000000000000080h,
VMEXIT_VMMCALL = 0000000000000081h,
VMEXIT_VMLOAD = 0000000000000082h,
VMEXIT_VMSAVE = 0000000000000083h,
VMEXIT_STGI = 0000000000000084h,
VMEXIT_CLGI = 0000000000000085h,
VMEXIT_SKINIT = 0000000000000086h,
VMEXIT_RDTSCP = 0000000000000087h,
VMEXIT_WBINVD = 0000000000000089h,
VMEXIT_INVALID = FFFFFFFFFFFFFFFFh}
```

APPENDIX

Q

MISCELLANEOUS INSTRUCTIONS

Instruction BOUND (**when** not $x64_mode)

Checks whether an array index (first operand) is within the bounds of an array (second operand). The array index is a signed integer in the specified register. If the operand-size attribute is 16, the array operand is a memory location containing a pair of signed word-integers; if the operand-size attribute is 32, the array operand is a pair of signed doubleword-integers. The first word or doubleword specifies the lower bound of the array and the second word or doubleword specifies the upper bound. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound. If the index is not within the specified bounds, the processor generates a BOUND range- exceeded exception (#BR). *[cited from AMD volume 3]*

```
opcode "64h" reg $v, mem_pair 2*$v : call BOUND(op1, op2)
action BOUND(idx::bits $v, bounds::bits (2*$v)) when not $x64_mode
  let low = bounds[$v-1:0]
  let upp = bounds[2*$v-1:$v]
  fail exception(xBR, 0000h) when int(idx) < int(low) or int(idx) > int(upp)
```

Instruction LEA

Computes the effective address of a memory location (second operand) and stores it in a general- purpose register (first operand). *[cited from AMD volume 3]*

```
opcode "8Dh" reg $v, mem 8 : let op1' = zxt(64, ea)[$v-1:0]
```

The ea function returns the effective address of the first operand and is defined in section 14.12.

Instruction HLT

Causes the microprocessor to halt instruction execution and enter the HALT state. Entering the HALT state puts the processor in low-power mode. Execution resumes when an unmasked hardware interrupt (INTR), non-maskable interrupt (NMI), system management interrupt (SMI), RESET, or INIT occurs. If an INTR, NMI, or SMI is used to

resume execution after a HLT instruction, the saved instruction pointer points to the instruction following the HLT instruction. Before executing a HLT instruction, hardware interrupts should be enabled. If rFLAGS.IF = 0, the system will remain in a HALT state until an NMI, SMI, RESET, or INIT occurs. *[cited from AMD volume 3]*

```
opcode "F4h" : call HLT
```

On the level of our domain-specific language, the `HLT` instruction is a no-operation. It gets processed later on in the instruction processing cycle as was described in chapter 9.

```
action HLT
  fail exception(xGP, 0000h) when CPL <> 00b
  fail intercept(VMEXIT_HLT) when _vmcb_ca.intercept.HLT
  call NOP
action NOP
  return
```

Instruction XLAT

Uses the unsigned integer in the AL register as an offset into a table and copies the contents of the table entry at that location to the AL register. The instruction uses seg:[rBX] as the base address of the table. The value of seg defaults to the DS segment, but may be overridden by a segment prefix. This instruction writes AL without changing RAX[63:8]. This instruction ignores operand size. The single-operand form of the XLAT instruction uses the operand to document the segment and address size attribute, but it uses the base address specified by the rBX register. *[cited from AMD volume 3]*

```
opcode "D7h" mem 8 : call XLAT
action XLAT
  let s = segment(iDS)
  let origin = segment_origin(s)
  call x = lread(origin, 8, SR[s], $oa, RBX[$oa-1:0] + zxt($oa, RAX[7:0]))
  write x to RAX[7:0]
```

Instruction PAUSE

Improves the performance of spin loops, by providing a hint to the processor that the current code is in a spin loop. The processor may use this to optimize power consumption while in the spin loop. Architecturally, this instruction behaves like a NOP instruction. Processors that do not support PAUSE treat this opcode as a NOP instruction. *[cited from AMD volume 3]*

```
opcode "F390h" : call NOP
```

APPENDIX

R

OPERAND READ AND WRITE

This section defines the `read_op` and the `write_op` actions, which are described in section 14.13.

```
action read_op($w::{8, 16, 32, 48, 64, 128}, t::OpType)::bits $w
  if t == reg or t == reg_mem and _modrm.mod == 11b then
    assume ($w == 8 or $w == 16 or $w == 32 or $w == 64)
    call x = read_gpr($w, _prefix.rex.R++_modrm.reg)
    return x
  elif t == reg_rm then
    assume ($w == 8 or $w == 16 or $w == 32 or $w == 64)
    call x = read_gpr($w, _prefix.rex.B++_modrm.rm)
    return x
  elif t == mem or t == mem_ptr then
    fail exception(xUD, 0000h) when _modrm.mod == 11b
    assume $w < 128
    let s = segment(def_segment)
    call x = lread(segment_origin(s), $w, SR[s], $oa, ea)
    return x
  elif t == mem_pair then
    fail exception(xUD, 0000h) when _modrm.mod == 11b
    assume $w == 64 or $w == 128
    let s = segment(def_segment)
    let addr0 = ea
    let addr1 = addr0 + bits($oa, $w/16)
    call x0 = lread(segment_origin(s), $w/2, SR[s], $oa, addr0)
    call x1 = lread(segment_origin(s), $w/2, SR[s], $oa, addr1)
    return x1++x0
  elif _modrm.mod <> 11b and (t == reg_mem or t == mmx_mem or t == xmm_mem) then
    assume $w <= 64
    let s = segment(def_segment)
    call x = lread(segment_origin(s), $w, SR[s], $oa, ea)
    return x
  elif t == imm or t == rel_off or t == imm_ptr then
    assume $w <= 32
```

```
    return _imm[$w-1:0]
  elif t == imm2 then
    assume $w <= 16
    return _imm[$w+15:16]
  elif t == moffset then
    assume $w <= 64
    let s = segment(iDS)
    call x = lread(segment_origin(s), $w, SR[s], $oa, _imm[$oa-1:0])
    return x
  elif t == es_rdi then
    assume $w <= 64
    let s = segment(iES)
    call x = lread(segment_origin(s), $w, SR[s], $oa, RDI[$oa-1:0])
    return x
  elif t == ds_rsi then
    assume $w <= 64
    let s = segment(iDS)
    call x = lread(segment_origin(s), $w, SR[s], $oa, RSI[$oa-1:0])
    return x
  elif t == creg then
    let idx = _prefix.rex.R++_modrm.reg
    fail exception(xUD, 0000h) when idx == 0001b or (idx > 0100b and idx <> 1000b)
    assume $w == 32 or $w == 64
    call x = read_cr($w, idx)
    return x
  elif t == sreg then
    let idx = _modrm.reg
    fail exception(xUD, 0000h) when idx > iFS
    assume $w == 16
    call x = read_sr(_modrm.reg)
    return x
  elif t == dreg then
    assume $w == 32 or $w == 64
    call x = read_dr($w, _modrm.reg)
    return x
  elif t == mmx or t == mmx_mem and _modrm.mod == 11b then
    assume ($w == 8 or $w == 16 or $w == 32 or $w == 64)
    call x = read_mmx($w, _modrm.reg)
    return x
  elif t == xmm or t == xmm_mem and _modrm.mod == 11b then
    assume ($w == 8 or $w == 16 or $w == 32 or $w == 64 or $w == 128)
    call x = read_xmm($w, _prefix.rex.R++_modrm.reg)
    return x
  elif t == rax or t == rcx or t == rdx or t == rbx
    or t == rsp or t == rbp or t == rsi or t == rdi
    or t == rax_r8 or t == rcx_r9 or t == rdx_r10
    or t == rbx_r11 or t == rbx_r11 or t == rsp_r12
    or t == rbp_r13 or t == rsi_r14 or t == rdi_r15 then
    assume ($w == 8 or $w == 16 or $w == 32 or $w == 64)
    call x = read_gpr($w, encode_gpr(t))
    return x
  elif t == rflags then
    assume ($w == 8 or $w == 16 or $w == 32 or $w == 64)
```

```
    return RFLAGS[$w-1:0]
  elif t == es or t == cs or t == ss
    or t == ds or t == fs or t == gs then
    assume $w == 16
    call x = read_sr(encode_sr(t))
   return x
  elif t == cr8 then
    assume $w == 32 or $w == 64
    call x = read_cr($w, 8h)
    return x
  elif t == const_0 then return (zero($w))
  elif t == const_1 then return (one($w))
  else fail bug
action write_op($w::{8, 16, 32, 48, 64, 128}, val::bits $w, t::OpType)
  if t == reg or t == reg_mem and _modrm.mod == 11b then
    assume ($w == 8 or $w == 16 or $w == 32 or $w == 64)
    call write_gpr($w, val, _prefix.rex.R++_modrm.reg)
  elif t == reg_rm then
    assume ($w == 8 or $w == 16 or $w == 32 or $w == 64)
    call write_gpr($w, val, _prefix.rex.B++_modrm.rm)
  elif t == mem or t == mem_ptr then
    assume $w <= 64
    let s = segment(def_segment)
    call lwrite(segment_origin(s), $w, val, SR[s], $oa, ea)
  elif _modrm.mod <> 11b and (t == reg_mem or t == mmx_mem or t == xmm_mem) then
    assume $w <= 64
    let s = segment(def_segment)
    call lwrite(segment_origin(s), $w, val, SR[s], $oa, ea)
  elif t == mem_pair then
    assume $w == 64 or $w == 128
    let s = segment(def_segment)
    let addr0 = ea
    let addr1 = addr0 + bits($oa, $w/16)
    call lwrite(segment_origin(s), $w/2, val[$w/2-1:0], SR[s], $oa, addr0)
    call lwrite(segment_origin(s), $w/2, val[$w-1:$w/2], SR[s], $oa, addr1)
  elif t == moffset then
    assume $w <= 64
    let s = segment(iDS)
    call lwrite(segment_origin(s), $w, val, SR[s], $oa, _imm[$oa-1:0])
  elif t == es_rdi then
    assume $w <= 64
    let s = segment(iES)
    call lwrite(segment_origin(s), $w, val, SR[s], $oa, RDI[$oa-1:0])
  elif t == ds_rsi then
    assume $w <= 64
    let s = segment(iDS)
    call lwrite(segment_origin(s), $w, val, SR[s], $oa, RSI[$oa-1:0])
  elif t == creg then
    let idx = _prefix.rex.R++_modrm.reg
    fail exception(xUD, 0000h) when idx == 0001b or (idx > 0100b and idx <> 1000b)
    assume $w == 32 or $w == 64
    call write_cr($w, val, idx)
  elif t == sreg then
```

```
  let idx = _modrm.reg
  fail exception(xUD, 0000h) when idx == iCS
  fail exception(xUD, 0000h) when idx > iFS
  assume $w == 16
  call write_sr(segment_origin(idx), val, idx)
elif t == dreg then
  assume $w == 32 or $w == 64
  call write_dr($w, val, _modrm.reg)
elif t == mmx or t == mmx_mem and _modrm.mod == 11b then
  assume $w == 8 or $w == 16 or $w == 32 or $w == 64
  call write_mmx($w, val, _modrm.reg)
elif t == xmm or t == xmm_mem and _modrm.mod == 11b then
  assume $w == 8 or $w == 16 or $w == 32 or $w == 64 or $w == 128
  call write_xmm($w, val, _prefix.rex.R++_modrm.reg)
elif t == rax or t == rcx or t == rdx or t == rbx
  or t == rsp or t == rbp or t == rsi or t == rdi
  or t == rax_r8 or t == rcx_r9 or t == rdx_r10
  or t == rbx_r11 or t == rbx_r11 or t == rsp_r12
  or t == rbp_r13 or t == rsi_r14 or t == rdi_r15 then
  assume ($w == 8 or $w == 16 or $w == 32 or $w == 64)
  call write_gpr($w, val, encode_gpr(t))
elif t == rflags then
  assume $w == 9 or $w == 16 or $w == 32 or $w == 64
  write val to RFLAGS[$w-1:0]
elif t == es or t == cs or t == ss
  or t == ds or t == fs or t == gs then
  assume $w == 16
  let idx = encode_sr(t)
  call write_sr(segment_origin(idx), val, idx)
elif t == cr8 then
  assume $w == 32 or $w == 64
  call write_cr($w, val, 8h)
else fail bug
```

APPENDIX

S

PAGE TABLE ENTRIES

```
layout PTELarge
  field P::bit
  field RW::bit
  field US::bit
  field PWT::bit
  field PCD::bit
  field A::bit
  field D::bit
  field PS::bit reserved and must be 1b
  field G::bit
  field AVL::bits 3 ignored
  field PAT::bit
  field rsv::bits 8 reserved and must be 00h
  field PFN::bits 31
  field AVL2::bits 11
  field NX::bit
layout PTE2
  field P::bit
  field RW::bit
  field US::bit
  field PWT::bit
  field PCD::bit
  field A::bit
  field D::bit ignored
  field PS::bit
  field G::bit ignored
  field AVL::bits 3
  field PFN::bits 40
  field ign::bits 11 ignored
  field NX::bit
layout PTE3
  field P::bit
  field RW::bit reserved and must be 0b when not $long_mode
  field US::bit reserved and must be 0b when not $long_mode
```

```
    field PWT::bit
    field PCD::bit
    field A::bit reserved and must be 0b when not $long_mode
    field D::bit
      reserved and must be 0b when not $long_mode
      ignored when $long_mode
    field PS::bit
    field G::bit reserved and must be 0b
    field AVL::bits 3
    field PFN::bits 40
    field ign::bits 11 ignored
    field NX::bit reserved and must be 0b when not $long_mode
  layout PTE4
    field P::bit
    field RW::bit
    field US::bit
    field PWT::bit
    field PCD::bit
    field A::bit
    field D::bit ignored
    field PS::bit reserved and must be 0b
    field G::bit reserved and must be 0b
    field AVL::bits 3
    field PFN::bits 40
    field ign::bits 11 ignored
    field NX::bit
  layout LegacyPTE1
    field P::bit
    field RW::bit
    field US::bit
    field PWT::bit
    field PCD::bit
    field A::bit
    field D::bit
    field PAT::bit
    field G::bit
    field AVL::bits 3
    field PFN::bits 20
  layout LegacyPTE2
    field P::bit
    field RW::bit
    field US::bit
    field PWT::bit
    field PCD::bit
    field A::bit
    field D::bit ignored
    field PS::bit
    field G::bit ignored
    field AVL::bits 3
    field PFN::bits 20
  layout LegacyPTELarge
    field P::bit
    field RW::bit
```

```
  field US::bit
  field PWT::bit
  field PCD::bit
  field A::bit
  field D::bit
  field PS::bit reserved and must be 1b
  field G::bit
  field AVL::bits 3
  field PAT::bit
  field PFN2::bits 8
  field rsv::bit reserved and must be 0b
  field PFN1::bits 10
function parse_pte1(x::PTE1)::AbsPTE when $PAE
  = AbsPTE with [valid = isPTE1(x), p = x.P, a = x.A, d = x.D, g = x.G,
                 r = Rights with [write = x.RW, user = x.US, code=not x.NX],
                 large = 0b, ba = x.PFN ++ bits(12, 0),
                 pat_idx = x.PAT ++ x.PCD ++ x.PWT]
function parse_large_pte(x::PTELarge, is1Gb::bit)::AbsPTE when $PAE
  = AbsPTE with [valid = isPTELarge(x) and not is1Gb or x.PFN[8:0],
                 p = x.P, a = x.A, d = x.D, g = x.G,
                 r = Rights with [write = x.RW, user = x.US, code=not x.NX],
                 large = 1b, ba = x.PFN ++ bits(21, 0),
                 pat_idx = x.PAT ++ x.PCD ++ x.PWT]
function parse_pte2(x::PTE2)::AbsPTE when $PAE
  = if x.PS then parse_large_pte(x, 0b)
    else AbsPTE with [valid = isPTE2(x), p = x.P, a = x.A, d = 0b, g = 0b,
                      r = Rights with [write = x.RW, user = x.US, code=not x.NX],
                      large = 0b, ba = x.PFN ++ bits(12, 0),
                      pat_idx = 0b ++ x.PCD ++ x.PWT]
function parse_pte3(x::PTE3)::AbsPTE when $PAE
  = if x.PS then parse_large_pte(x, 1b)
    else AbsPTE with [valid = isPTE3(x), p = x.P, a = x.A, d = 0b, g = 0b,
                      r = Rights with [write = x.RW, user = x.US, code=not x.NX],
                      large = 0b, ba = x.PFN ++ bits(12, 0),
                      pat_idx = 0b ++ x.PCD ++ x.PWT]
function parse_pte4(x::PTE4)::AbsPTE when $PAE
  = AbsPTE with [valid = isPTE4(x), p = x.P, a = x.A, d = 0b, g = 0b,
                 r = Rights with [write = x.RW, user = x.US, code=not x.NX],
                 large = 0b, ba = x.PFN ++ bits(12, 0),
                 pat_idx = 0b ++ x.PCD ++ x.PWT]
function parse_pte1(x::LegacyPTE1)::AbsPTE when not $PAE
  = AbsPTE with [valid = isLegacyPTE1(x), p = x.P, a = x.A, d = x.D, g = x.G,
                 r = Rights with [write = x.RW, user = x.US, code=1b],
                 large = 0b, ba = bits(32, 0) ++ x.PFN ++ bits(12, 0),
                 pat_idx = x.PAT ++ x.PCD ++ x.PWT]
function parse_large_pte(x::LegacyPTELarge)::AbsPTE when not $PAE
  = AbsPTE with [valid = isLegacyPTELarge(x), p = x.P, a = x.A, d = x.D, g = x.G,
                 r = Rights with [write = x.RW, user = x.US, code=1b],
                 large = 1b, ba = bits(32, 0) ++ x.PFN2 ++ x.PFN1 ++ bits(22, 0),
                 pat_idx = x.PAT ++ x.PCD ++ x.PWT]
function parse_pte2(x::LegacyPTE2)::AbsPTE when not $PAE
  = if x.PS then parse_large_pte(x)
    else AbsPTE with [valid = isLegacyPTE2(x), p = x.P, a = x.A, d = 0b, g = 0b,
```

```
r = Rights with [write = x.RW, user = x.US, code=1b],
large = 0b, ba = bits(32, 0) ++ x.PFN ++ bits(12, 0),
pat_idx = x.PAT ++ x.PCD ++ x.PWT]
```

BIBLIOGRAPHY

[Adv07] Advanced Micro Devices (AMD), Inc. *AMD64 Architecture Programmer's Manual: Volumes 1–3*, September 2007.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29:66–76, December 1996.

[ANB+95] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, pages 37–49, 1995.

[Bau08] Christoph Baumann. Formal specification of the x87 floating-point instruction set. Master's thesis, Saarland University, Germany, 2008.

[Boc] Bochs ia-32 emulator project. urlhttp://bochs.sourceforge.net.

[Cor11a] International Data Corporation. Worldwide pc microprocessor unit shipments. Press Release, May 2011. `http://www.idc.com/getdoc.jsp?containerId=prUS22814611`.

[Cor11b] International Data Corporation. Worldwide server market revenue. Press Release, May 2011. `http://www.idc.com/getdoc.jsp?containerId=prUS22841411`.

[Deg07] Ulan Degenbaev. Formalization of parts of the x86-64 Instruction Set Architecture. Master's thesis, Saarland University, Germany, 2007.

[DPS09] Ulan Degenbaev, Wolfgang J. Paul, and Norbert Schirmer. *Pervasive Theory of Memory*, pages 74–98. Springer-Verlag, Berlin, Heidelberg, 2009.

[DSB98] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 320–328, New York, NY, USA, 1998. ACM.

[FF01] Anthony C. J. Fox and Anthony Fox. A hol specification of the arm instruction set architecture, 2001.

[FM10] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. pages 243–258, 2010.

[HHD97] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: an instruction set description language for retargetability. In *Proceedings of the 34th annual Design Automation Conference*, DAC '97, pages 299–302, New York, NY, USA, 1997. ACM.

[HKV98] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Weak memory consistency models. part i: Definitions and comparisons. Technical report, Department of Computer Science, The University of Calgary, 1998.

[Int09] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volumes 1–3b*. Intel Corporation, Santa Clara, CA, USA, March 2009.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28:690–691, September 1979.

[MP00] S.M. Mueller and W.J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[PPS+95] Woosang Park, Woosang Park, Alexandre Santoro, Alexandre Santoro, Alexandre Santoro, David Luckham, David Luckham, and David Luckham. Sparc-v9 architecture specification with rapide, 1995.

[QEM] Qemu processor emulator project. urlhttp://qemu.org.

[RF95] Norman Ramsey and Mary F. Fernandez. The new jersey machine-code toolkit. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 24–24, Berkeley, CA, USA, 1995. USENIX Association.

[RF97] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst.*, 19:492–524, May 1997.

[RM99] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *Proceedings of the 12th International Conference on VLSI Design - 'VLSI for the Information Appliance'*, VLSID '99, pages 132–, Washington, DC, USA, 1999. IEEE Computer Society.

[SPA92] SPARC International, Inc. *The SPARC Architecture Manual, V. 8. SPARC*. 1992.

[SS86] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 414–423, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[SSN+09] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In *Symposium on Principles of Programming Languages*, pages 379–391, 2009.

[Vir] Virtualbox x86 virtualization project. urlhttp://www.virtualbox.org.

[Wad92] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[WAH10] Jr Warren A. Hunt. Verifying via nano microprocessor components. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*, pages 3–10, October 2010.