Technical Report: MIPS-86 – A Multi-Core MIPS ISA Specification (November 4, 2013)

Sabine Schmaltz

November 4, 2013

Abstract

This report provides a simple multi-core MIPS model we call MIPS-86. It aims at providing an overall specification for the reverse-engineered hardware models provided in [Pau12]. We take the simple sequential MIPS processor model from [Pau12] and extend it with a memory and device model that resembles the one of modern x86 architectures. The model has the following features: Sequential processor core abstraction with atomic execute-and fetch transitions, memory-management units (MMUs) with translation-lookaside buffers (TLBs), store buffers (SBs), processor-local advanced programmable interrupt controllers (local APICs), I/O APICs, and devices.

Open issues:

- adaption of this specification to fit to the pipelined hardware implementation. E.g. by introduction of pipelining abstractions (e.g. delayed PC, delayed visibility of multiplication results in HI, LO)
- addition of $emode$ special purpose register

Changes:

- Oct 21, 2013: Removed dirty bit from page table entries.
- Nov 1, 2012: Added missing requirement regarding the $running$ flag on the **tlb-create** transition.

Contents

1 Formal Model of MIPS-86

This report provides a simple multi-core MIPS model we call MIPS-86. It aims at providing an overall specification for the reverse-engineered hardware models provided in [Pau12]. Essentially, we take the simple sequential MIPS processor model from [Pau12] and extend it with a memory and device model that resembles the one of modern x86 architectures. The model has the following features:

- Sequential processor core abstraction with atomic execute-and fetch transitions

  In order to justify modeling instruction execution by an atomic transition that combines fetching the instruction from memory and executing it, the absence of self-modifying code is a prerequisite. When instructions being fetched cannot be changed by the execution of other cores, these fetch cycles can be reordered to occur right before the corresponding execute cycle. This, in turn, means that the semantics of fetch and execute steps can be combined into single atomic steps.

- Memory-management unit (MMU) with translation-lookaside buffer (TLB)

  The memory-management unit considered performs a 2-level translation from virtual to physical addresses, caching partial and complete translations (which are called *walks*) in a *translation lookaside buffer* (TLB). The *page table origin*, i.e. the address of the first-level page table, is taken from the special purpose register pto. In order to allow an update of page-tables to be performed in a consistent manner, the machine is extended by two instructions: A *flush*-operation that empties the TLB of all walks, and an *invalidate-page*-operation that removes all walks to a certain virtual page address from the TLB.

- Store buffer (SB)

  In order to argue about store-buffer reduction, we provide a processor model with store-buffer. The store-buffer we consider is simple in the sense that it does not reorder or combine accesses but instead simply acts as a first-in-first-out queue for memory write operations to physical addresses. We provide two serializing instructions: A *fence*-operation that simply drains the store-buffer, and a *compare-and-swap*-operation that atomically updates a memory word on a conditional basis while also draining the store-buffer.

- Processor-local advanced programmable interrupt controller (local APIC)

  In order to have a similar boot mechanism as the x86-architecture, we imitate the inter-processor-interrupt (IPI) mechanism of the x86-architecture. We extend our

MIPS model by a strongly simplified local APIC for each processor. The local APIC provides interrupt signals to the processor and acts as a device for sending inter-processor-interrupts between processors. Local APIC ports are mapped into a processor's memory space by means of memory-mapped I/O.

- I/O APIC

  The I/O APIC is a component that is connected to the devices of the system and to the local APICs of the processors of the system. It provides the means to configure distribution of device interrupts to the processors of the multi-core system, i.e. whether a given device interrupt is masked and which processor will receive the interrupt. We do not consider edge-triggered interrupts, however, we do model the end-of-interrupt (EOI) protocol between local APIC and I/O APIC: After sending an interrupt signal to a local APIC, the I/O APIC will not sample a raised device interrupt again until the corresponding EOI message has been received from the local APIC.

- Devices

  We use a generic framework along the lines of the Verisoft device model [HIP05]: Device configurations are required to have a certain structure which can be instantiated, e.g. certain device transitions that specify side-effects associated with reading or writing device ports must be provided. Every device consists of ports and an interrupt line it may raise as well as some internal state that may be instantiated freely. Devices may interact with an external environment by receiving inputs and providing outputs.

In the following, we proceed by providing tables that give an overview over the instruction-set-architecture of MIPS-86, followed by operational semantics of the non-deterministic MIPS-86 model.

2 Instruction-Set-Architecture Overview and Tables

The instruction-set-architecture of MIPS-86 provides three different types of instructions: I-type instructions, J-type instructions and R-type instructions. I-type instructions are instructions that operate with two registers and a so-called *immediate constant*, J-type instructions are absolute jumps, and R-type instructions rely on three register operands.

2.1 Instruction Layout

The instruction-layout of MIPS-86 depends on the type of instruction. In the subsequent definition of the *MIPS-86* instruction layout, rs, rt and rd specify registers of the MIPS-86 machine.

2.1.1 I-Type Instruction Layout

Bits	31 . . . 26	25 . . . 21	20 . . . 16	15 . . . 0
Field Name	opcode	rs	rt	immediate constant imm

2.1.2 R-Type Instruction Layout

Bits	31 . . . 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . . . 6	5 . . . 0
Field Name	opcode	rs	rt	rd	shift amount sa	function code fun

2.1.3 J-Type Instruction Layout

Bits	31 . . . 26	25 . . . 0
Field Name	opcode	instruction index $iindex$

2.1.4 Effect of Instructions

A quick overview of available instructions is given in tables 1 (for I-type), 2 (for J-type), and 3 (for R-type). Note that these tables – while giving a general idea what is available and what it approximately does – are not comprehensive. In particular, note that for all instructions whose mnemonic ends with "u", register values are interpreted as binary numbers whereas in all other cases they are interpreted as two's-complement numbers. Note also that MIPS-86 is still incomplete in the sense that in order to accommodate the distributed cache model of the hardware construction, the architecture needs to be extended to allow proper management of cache-bypassing memory access (e.g. to devices). The abstract model provided here is one where caches are already abstracted into a view of a single coherent memory. The exact semantics of all instructions present in the provided instruction-set architecture tables is given later in the transition function of the MIPS-86 processor core.

2.2 Coprocessor Instructions and Special-Purpose Registers

Note that in contrast to most MIPS-architectures, in MIPS-86 coprocessor-instructions are provided as R-type instructions. Coprocessor instructions in MIPS-86 deal with moving data between special-purpose register file and general-purpose register file and exception return. The available special purpose registers of MIPS-86 are listed in table 5.

2.3 Interrupts

Traditionally, hardware architectures provide an *interrupt* mechanism that allows the processor to react to events that require immediate attention. When an *interrupt signal* is raised, the hardware construction reacts by transferring control to an interrupt handler – on the level of hardware, this basically means that the program counter is set to the specific address where the hardware expects the interrupt handler code to be placed by the programmer and that information about the nature of the interrupt is provided in special registers. Since interrupts are mainly supposed to be handled by an operating system instead of by user processes, such a *jump to interrupt service routine* (JISR) step also tends to involve switching the processor to *system mode*.

Interrupts come in two major flavors: There are *internal interrupts* that are triggered by executing instructions, e.g. an overflow occuring in an arithmetic operation, a system-call instruction being executed (which is supposed to have the semantics of

Table 1: I-Type Instructions of MIPS-86.

opcode	Mnemonic	Assembler-Syntax	d	Effect
Data Transfer				
100 000	lb	lb $rt\ rs\ imm$	1	rt = sxt(m[rs+imm])
100 001	lh	lh $rt\ rs\ imm$	2	rt = sxt(m[rs+imm])
100 011	lw	lw $rt\ rs\ imm$	4	rt = m[rs+imm]
100 100	lbu	lbu $rt\ rs\ imm$	1	rt = 0^{24}m[rs+imm]
100 101	lhu	lhu $rt\ rs\ imm$	2	rt = 0^{16}m[rs+imm]
101 000	sb	sb $rt\ rs\ imm$	1	m[rs+imm] = rt[7:0]
101 001	sh	sh $rt\ rs\ imm$	2	m[rs+imm] = rt[15:0]
101 011	sw	sw $rt\ rs\ imm$	4	m[rs+imm] = rt
Arithmetic, Logical Operation, Test-and-Set				
001 000	addi	addi $rt\ rs\ imm$		rt = rs + sxt(imm)
001 001	addiu	addiu $rt\ rs\ imm$		rt = rs + sxt(imm)
001 010	slti	slti $rt\ rs\ imm$		rt = (rs $<$ sxt(imm) ? 1 : 0)
001 011	sltiu	sltiu $rt\ rs\ imm$		rt = (rs $<$ sxt(imm) ? 1 : 0)
001 100	andi	andi $rt\ rs\ imm$		rt = rs $\wedge$ zxt(imm)
001 101	ori	ori $rt\ rs\ imm$		rt = rs $\vee$ zxt(imm)
001 110	xori	xori $rt\ rs\ imm$		rt = rs $\oplus$ zxt(imm)
001 111	lui	lui $rt\ imm$		rt = imm0^{16}

opcode	rt	Mnemonic	Assembler-Syntax	Effect
Branch				
000 001	00000	bltz	bltz $rs\ imm$	pc = pc + (rs $<$ 0 ? imm00 : 4)
000 001	00001	bgez	bgez $rs\ imm$	pc = pc + (rs $\geq$ 0 ? imm00 : 4)
000 100		beq	beq $rs\ rt\ imm$	pc = pc + (rs $=$ rt ? imm00 : 4)
000 101		bne	bne $rs\ rt\ imm$	pc = pc + (rs $\neq$ rt ? imm00 : 4)
000 110	00000	blez	blez $rs\ imm$	pc = pc + (rs $\leq$ 0 ? imm00 : 4)
000 111	00000	bgtz	bgtz $rs\ imm$	pc = pc + (rs $>$ 0 ? imm00 : 4)

Table 2: J-Type Instructions of MIPS-86

opcode	Mnemonic	Assembler-Syntax	Effect
Jumps			
000 010	j	j $iindex$	pc = bin_{32}(pc+4)[31:28]iindex00
000 011	jal	jal $iindex$	R31 = pc + 4 pc = bin_{32}(pc+4)[31:28]iindex00

Table 3: R-Type Instructions of MIPS-86.

opcode	fun	Mnemonic	Assembler-Syntax	Effect
Shift Operation				
000000	000 000	sll	sll $rd\ rt\ sa$	rd = sll(rt,sa)
000000	000 010	srl	srl $rd\ rt\ sa$	rd = srl(rt,sa)
000000	000 011	sra	sra $rd\ rt\ sa$	rd = sra(rt,sa)
000000	000 100	sllv	sllv $rd\ rt\ rs$	rd = sll(rt,rs)
000000	000 110	srlv	srlv $rd\ rt\ rs$	rd = srl(rt,rs)
000000	000 111	srav	srav $rd\ rt\ rs$	rd = sra(rt,rs)
Multiplication Unit Instructions				
000000	010 000	mfhi	mfhi rd	rd = HI
000000	010 001	mthi	mthi rs	HI = rs
000000	010 010	mflo	mflo rd	rd = LO
000000	010 011	mtlo	mtlo rs	LO = rs
000000	011 000	mult	mult $rs\ rt$	HI,LO = rs $\cdot$ rt
000000	011 001	multu	multu $rs\ rt$	HI,LO = rs $\cdot$ rt
Arithmetic, Logical Operation				
000000	100 000	add	add $rd\ rs\ rt$	rd = rs $+$ rt
000000	100 001	addu	addu $rd\ rs\ rt$	rd = rs $+$ rt
000000	100 010	sub	sub $rd\ rs\ rt$	rd = rs $-$ rt
000000	100 011	subu	subu $rd\ rs\ rt$	rd = rs $-$ rt
000000	100 100	and	and $rd\ rs\ rt$	rd = rs $\wedge$ rt
000000	100 101	or	or $rd\ rs\ rt$	rd = rs $\vee$ rt
000000	100 110	xor	xor $rd\ rs\ rt$	rd = rs $\oplus$ rt
000000	100 111	nor	nor $rd\ rs\ rt$	rd = rs $\overline{\vee}$ rt
Test Set Operation				
000000	101 010	slt	slt $rd\ rs\ rt$	rd = (rs $<$ rt ? 1 : 0)
000000	101 011	sltu	sltu $rd\ rs\ rt$	rd = (rs $<$ rt ? 1 : 0)
Jumps, System Call				
000000	001 000	jr	jr rs	pc = rs
000000	001 001	jalr	jalr $rd\ rs$	rd = pc + 4 pc = rs
000000	001 100	sysc	sysc	System Call
Synchronizing Memory Operations				
000000	111 111	cas	cas $rd\ rs\ rt$	rd' = m m' = (rd = m ? rt : m) flushes the SB
000000	111 110	mfence	mfence	flushes the SB
TLB Instructions				
000000	111 101	flush	flush	flushes TLB
000000	111 100	invlpg	invlpg $rd\ rs$	flushes TLB translations for addr. rd from ASID rs

Table 4: R-Type Instructions of MIPS-86, continued.

Multiplication Instructions				
opcode	fun	Mnemonic	Assembler-Syntax	Effect
011100	000 000	madd	madd $rs\ rt$	HI,LO = HI,LO + rs · rt
011100	000 001	maddu	maddu $rs\ rt$	HI,LO = HI,LO + rs · rt
011100	000 010	mul	mul $rd\ rs\ rt$	rd = (rs · rt) [31:0]
011100	000 100	msub	msub $rs\ rt$	HI,LO = HI,LO - rs · rt
011100	000 101	msubu	msubu $rs\ rt$	HI,LO = HI,LO - rs · rt

Coprocessor Instructions					
opcode	rs	fun	Mnemonic	Assembler-Syntax	Effect
010000	10000	011 000	eret	eret	Exception Return
010000	00100		movg2s	movg2s $rd\ rt$	spr[rd] := gpr[rt]
010000	00000		movs2g	movs2g $rd\ rt$	gpr[rt] := spr[rd]

Table 5: MIPS-86 Special Purpose Registers.

i	synonym	
0	sr	status register (contains masks to enable/disable maskable interrupts)
1	esr	exception sr
2	eca	exception cause register
3	epc	exception pc (address to return to after interrupt handling)
4	edata	exception data (contains effective address on pfls)
5	pto	page table origin
6	asid	address space identifier
7	mode	mode register $\in \{0^{31}1, 0^{32}\}$

Table 6: MIPS-86 Interrupt Types and Priority.

interrupt level	shorthand	internal/external	type	maskable	
0	reset	eev	abort	0	reset
1	I/O	eev	repeat	1	devices
2	ill	iev	abort	0	illegal instruction
3	mal	iev	abort	0	misaligned
4	pff	iev	repeat	0	page fault fetch
5	pfls	iev	repeat	0	page fault load/store
6	sysc	iev	continue	0	system call
7	ovf	iev	continue	1	overflow

returning control to the operating system in order to perform some task requested by a user processor), or a page fault interrupt due to a missing translation in the page tables. In contrast, there are *external interrupts* which are triggered by an external source, e.g. the reset signal or device interrupts. Interrupts of lesser importance tend to be *maskable*, i.e. there is a control register that allows the programmer to configure that certain kinds of interrupts shall be ignored by the hardware.

The possible interrupt sources and priorities of MIPS-86 are listed in table 6. Interrupts are either of type *repeat*, *abort*, or *continue*. Here *repeat* expresses that the interrupted instruction will be repeated after returning from the interrupt handler, *abort* means that the exception is usually so severe that the machine will be by default unable to return from the exception, and *continue* means that even though there is an interrupt, the execution of the interrupted execution will be completed before jumping to the interrupt-service-routine. In case of a *continue*-interrupt execution after exception return will proceed behind the interrupted execution. Note that the APIC mechanism is discussed in more detail in section 8.1.

3 Overview of the MIPS-86-Model

3.1 Configurations

We define the set K of configurations of an abstract simplified multi-core MIPS machine in a top-down way. I.e., we first give a definition of the overall concurrent machine configuration which is composed of several subcomponent configurations whose definitions follow.

Definition 1 (Configuration of MIPS-86)**.** *A configuration*

$$c = (c.p, c.\textit{running}, c.m, c.d, c.ioapic) \in K$$

of MIPS-86 is composed of the following components:

- *a mapping from processor identifier to processor configuration,* $c.p : [0 : np - 1] \to K_{\textbf{\textit{p}}}$*,*

  (np *is a parameter that describes the number of processors of the multi-core machine)*

- *a mapping from processor identifier to a flag that describes whether the processor is running,* $c.\textit{running} : [0 : np - 1] \to \mathbb{B}$*,*

  (If $c.\textit{running}(i) = 0$*, this means that the processor is currently waiting for a startup-inter-processor-interrupt (SIPI))*

- *a shared global memory component* $c.m \in K_{\textbf{\textit{m}}}$*,*

- *a mapping from device identifiers to device configurations,* $c.dev : [0 : nd-1] \to K_{\textbf{\textit{dev}}}$*, and*

  (where $K_{\textbf{\textit{dev}}} = \bigcup_{i=0}^{nd-1} K_{\textbf{\textit{dev(i)}}}$ *is the union of individual device configurations, and* nd *is a parameter that describes the number of devices considered)*

- *an I/O APIC,* $c.ioapic \in K_{\textbf{\textit{ioapic}}}$*.*

3.1.1 Processor

Definition 2 (Processor Configuration of MIPS-86)**.**

$$K_{\textbf{\textit{p}}} = K_{\textbf{\textit{core}}} \times K_{\textbf{\textit{sb}}} \times K_{\textbf{\textit{tlb}}} \times K_{\textbf{\textit{apic}}}$$

A processor $p = (p.\textit{core}, p.sb, p.tlb, p.apic) \in K_{\textbf{\textit{p}}}$ *is subdivided into the following components:*

- *a processor core* $p.core \in K_{\textbf{\textit{core}}}$*,*

  The processor core executes instructions according to the Instruction-Set-Architecture (ISA).

- *a store buffer* $p.sb \in K_{\textbf{\textit{sb}}}$*,*

  A store-buffer buffers write accesses to the memory system local to the processor. If possible, read requests by the core are served by the store-buffer. Writes leave the store-buffer in the order they were placed (first-in-first-out).

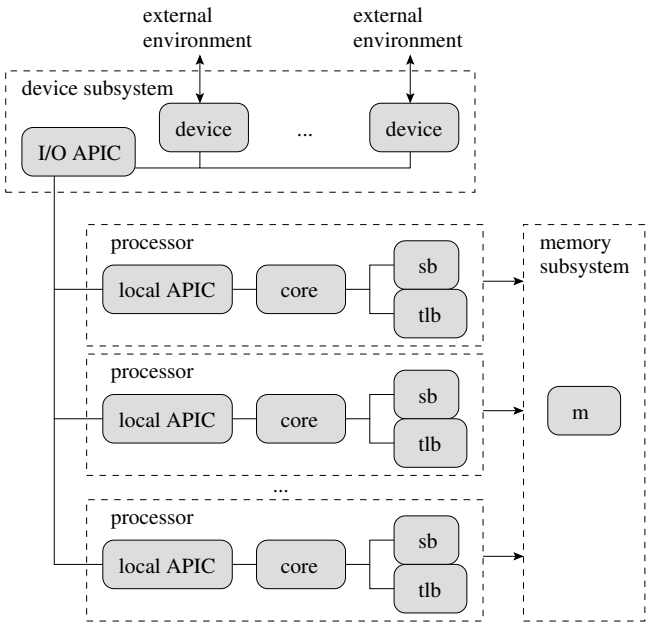

Figure 1: Overview of MIPS-86 Model Components.

- *a TLB* $p.tlb \in K_{\textbf{\textit{tlb}}}$

  A translation-lookaside buffer (TLB) performs and caches address translations to be used by the processor in order to establish a virtual memory abstraction.

- *a local APIC* $p.apic \in K_{\textbf{\textit{apic}}}$

  A local APIC receives interrupt signals from the I/O APIC and provides them to the processor. Additionally, it acts as a processor-local device that can send inter-processor-interrupts (IPI) to other processors of the system.

Definitions of $K_{\textbf{core}}, K_{\textbf{sb}}, K_{\textbf{tlb}}$, and $K_{\textbf{apic}}$ are each given in the section that defines the corresponding component in detail.

3.1.2 Memory

Definition 3 (Memory Configuration of MIPS-86)**.** *For this abstract machine, we consider a simple byte-addressable shared global memory component*

$$K_{\textbf{\textit{m}}} \equiv \mathbb{B}^{32} \to \mathbb{B}^{8}$$

which is sequentially consistent.

Definition 4 (Reading Byte-Strings from Byte-Addressable Memory)**.** *For a memory* $m \in K_{\textbf{\textit{m}}}$ *and an address* $a \in \mathbb{B}^{32}$ *and a number* $d \in \mathbb{N}$ *of Bytes, we define*

$$m_d(a) = \begin{cases} m_{d-1}(a +_{32} 1_{32}) \circ m(a) & d > 0 \\ \varepsilon & d = 0 \end{cases}$$

3.2 Transitions

We define the semantics of the concurrent MIPS machine MIPS-86 as an automaton with a partial transition function

$$\delta : K \times \Sigma \rightharpoonup K$$

and an output function

$$\lambda : K \times \Sigma \rightharpoonup \Omega$$

where Σ is the set of inputs to the automaton and Ω is the set of outputs of the automaton. In particular, in order to be able to define the semantics of our system as a deterministic automaton, these inputs do include scheduling information, i.e. they determine exactly which subcomponent makes what step. Note that, in the following sections, we will first define the semantics of all individual components before we give the definition of δ and λ in section 11.

3.2.1 Scheduling

We provide a model in which the execution order of individual components of the system is not known to the programmer. In order to prove correct execution of code, it is necessary to consider all possible execution orders given by the model. We model this non-deterministic behavior by deterministic automata which take, as part of their input, information about the execution order. This is done in such a way that, in every step, it is specified exactly which subcomponent of the overall system makes which particular step.

There are occasions where several components make a synchronous step. In such cases, our intuition is that one very specific subcomponent *actively* performs a step, while all other components make a *passive* step that merely responds to the active component in some way. The memory, in particular, is such a passive component. Also, devices can react passively to having their registers read by a processor, causing a side-effect on reading.

4 Memory

Definition 5 (Memory Transition Function)**.** *We define the memory transition function*

$$\delta_{\boldsymbol{m}} : K_{\boldsymbol{m}} \times \Sigma_{\boldsymbol{m}} \rightharpoonup K_{\boldsymbol{m}}$$

where

$$\Sigma_{\boldsymbol{m}} = \mathbb{B}^{32} \times (\mathbb{B}^8)^* \cup \mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{32}$$

Here,

- $(a, v) \in \mathbb{B}^{32} \times (\mathbb{B}^8)^*$ *– describes a write access to address a with value v, and*
- $(c, a, v) \in (\mathbb{B}^{32})^3$ *– describes a read-modify-write access to address a with compare-value c and value v to be written in case of success.*

We have

$$\delta_{\boldsymbol{m}}(m, in)(x) = \begin{cases} byte(\langle x \rangle - \langle a \rangle, v) & in = (a, v) \wedge 0 \leq \langle x \rangle - \langle a \rangle < \textbf{\textit{len}}(v)/8 \\ byte(\langle x \rangle - \langle a \rangle, v) & in = (c, a, v) \wedge m_4(a) = c \wedge 0 \leq \langle x \rangle - \langle a \rangle < 4 \\ m(x) & otherwise \end{cases}$$

Note that, in this memory model, we assume that read accesses cannot change the memory state – thus, any result of a read operation can simply be computed directly from a given memory configuration. In a more concrete model where caches are still visible, we need to consider memory reads as explicit inputs to the cache system.

5 TLB

5.1 Address Translation

Most processors, including MIPS-86 and x86-64, provide *virtual memory* which is mostly used for implementing process separation in the context of an operating system. By performing *address translation* from virtual memory addresses to physical memory addresses (i.e. regular memory addresses of the machine's memory system), the notion of a virtual memory is established – if this translation is injective, virtual memory has regular memory semantics (i.e. writing to an address affects only this single address and values being written can be read again later). Mostly, this is used to establish several virtual address spaces that are mapped to disjoint address regions in the physical memory of the machine. User processes of the operating system can then each be run by the operating system in their respective address spaces without any risk of user processes affecting each other or the operating system.

Processors tend to provide a mechanism to activate and deactivate address translation – usually by writing some special control register. In the case of MIPS-86, a special-purpose-register $mode$ is provided which decides whether the processor is running in *system mode*, i.e. without address translation, or in *user mode*, i.e. with address translation.

The translation from virtual addresses to physical addresses is usually given at the granularity of *memory pages* – in the case of MIPS-86, a *memory page* consists of 2^{12} consecutive bytes. Since MIPS-86 is a 32-bit architecture, a *page address* thus consists of 20 Bits. Defining a particular translation from virtual addresses to phyiscal addresses is done by establishing *page tables* that describe the translation. In the most simple case, a single-level translation can be given by a single page table – which is a sequence of page table entries that each describe how – and if – a given *virtual page address* is translated to a corresponding *physical page address*. The translation tends to be partial, i.e. not every virtual page address has a corresponding physical page address, which is reflected in the page table entry by the *present bit*. Trying to access a virtual address in user mode that does not have a translation set up in the page table according to the present bit then results in a *page-fault* interrupt which returns the processor to system mode – e.g. allowing the operating system to set up a translation for the faulting virtual page address.

MIPS-86, like x86-64, applies a *multi-level page table hierarchy*: Instead of translating using a single page table that describes the virtual page address to physical page address translation, there are several levels of page tables. One advantage of this is that multi-level page tables tend to require less memory space: Instead of providing a page table entry for every virtual page address, the page tables now form a graph in such a way that every level of page tables effectively describes a part of the page address translation by linking to page tables belonging to the next level of translation. Since only a part of the translation is provided, these page tables are much smaller than a single-level translation page table. MIPS-86 provides 2 levels of address translation (in comparison, x86-64 makes use of 4 levels). The first level page table – also called *page*

directory – translates the first 10 Bits of a page address by describing where the page tables for translating the remaining 10 Bits can be found, i.e. they contain addresses of second level page tables. These *terminal page tables* then provide the physical page addresses. Note that, in defining a multi-level translation via page tables, page table entries can be marked as not present in early translation levels which essentially means that no further page tables need to be provided for the given virtual page address prefix – which effectively is the reason why multi-level translations tend to require less memory space.

The actual translation is performed in hardware by introducing a circuit called *memory management unit* (MMU) which serves translation requests from the processor by accessing the page tables residing in memory. In a naive hardware implementation of address translation, the processor running in user mode could simply issue translation requests to the MMU in order as needed for instruction execution and wait for the MMU circuit to respond with a translation. Such a synchronous implementation however, would mean that the processor is constantly waiting for the MMU to perform translations, limiting the speed of execution to that of the MMU performing as many memory accesses as needed to compute the translations needed for instruction fetch and execution. Fortunately, however, it can be observed that instruction fetch in user mode to a large degree tends to require translations for virtual addresses that lie in the same page (with an appropriate programming style this is also mostly true for memory accesses performed by instructions), thus, in order not to constantly have the MMU repeat a translation for the same virtual page address, it might be helpful to keep translations available to the processor in a special processor-local cache for translations. This cache is commonly called *translation lookaside buffer* (TLB) and is updated by the MMU whenever necessary in order to serve translation requests by the processor. Note that a hardware TLB may cache partial translations for virtual page address prefixes in order to reuse them later.

Since the operating system may modify the page tables, translations in the TLB may become outdated – removing or changing translations provided by the page tables can make the TLB inconsistent with the page tables. Thus, architectures with TLB tend to provide instructions that allow the processor to control the state of the TLB to some degree. The functionality needed in order to keep the TLB consistent with the page tables is in fact quite simple: In order to ensure that all translations present in the TLB are also given by the page tables, all we need is to be able to remove particular translations (or all translations) from the TLB. Both x86-64 and MIPS-86 provide such instructions – for MIPS-86, $invlpg$ invalidates a single virtual page address, while $flush$ removes all translations from the TLB.

5.2 TLB Configuration

When the MIPS-86 processor is running in user mode, all memory accesses are subject to address translation according to page tables residing in memory. In order to perform address translation, the MMU operates on the page tables to create, extend, complete, and drop walks. A complete walk provides a translation from a virtual address to a physical address of the machine that can in turn be used by the processor core. Our TLB offers *address space identifier*s – a tag that can be used to associate translations with particular users – which reduces the need for TLB flushes when switching between users.

Definition 6 (TLB Configuration of MIPS-86)**.** *We define the set of configurations of*

a TLB as

$$K_{\textbf{\textit{tlb}}} = 2^{K_{\textbf{\textit{walk}}}}$$

where the set of walks $K_{\textbf{\textit{walk}}}$ *is given by*

$$K_{\textbf{\textit{walk}}} = \mathbb{B}^{20} \times \mathbb{B}^{6} \times \{0,1,2\} \times \mathbb{B}^{20} \times \mathbb{B}^{3} \times \mathbb{B}$$

The components of a walk $w = (w.va, w.asid, w.level, w.ba, w.r, w.fault) \in K_{\textbf{\textit{walk}}}$ *are the following:*

- $w.va \in \mathbb{B}^{20}$ *– the virtual page address to be translated,*
- $w.asid \in \mathbb{B}^{6}$ *– the* address space identifier *(ASID) the translation belongs to,*
- $w.level \in \{0,1,2\}$ *– the current level of the walk, i.e. the number of remaining walk extensions needed to complete the walk,*
- $w.ba \in \mathbb{B}^{20}$ *– the physical page address of the page table to be accessed next, or, if the walk is complete, the result of the translation,*
- $w.r \in \mathbb{B}^{3}$ *– the accumulated access rights, and*

  Here, $r[0]$ *stands for write permission,* $r[1]$ *for user mode access, and* $r[2]$ *expresses execute permission.*
- $w.fault \in \mathbb{B}$ *– a page fault indicator.*

5.3 TLB Definitions

In the following, we make definitions that describe the structure of page tables and the translation function specified by a given page table origin according to a memory configuration. Addresses are split in two page index components px_2, px_1 and a byte offset px_0 within a page:

$$a = a.px_2 \circ a.px_1 \circ a.px_0$$

Definition 7 (Page and Byte Index)**.** *Given an address* $a \in \mathbb{B}^{32}$*, we define*

- *the second-level page index* $a.px_2 = a[31:22]$*,*
- *the first-level page index* $a.px_1 = a[21:12]$*, and*
- *the byte offset* $a.px_0 = a[11:0]$

of a*.*

Definition 8 (Base Address (Page Address))**.** *The* base address *(also sometimes referred to as* page address*) of an address* $a \in \mathbb{B}^{32}$ *is then given by*

$$a.ba = a.px_2 \circ a.px_1.$$

Definition 9 (Page Table Entry)**.** *A* page table entry $pte \in \mathbb{B}^{32}$ *consists of*

- $pte.ba = pte[31:12]$ *– the base address of the next page table or, if the page table is a terminal one, the resulting physical page address for a translation,*
- $pte.p = pte[11]$ *– the present bit,*

- $pte.r = pte[10:8]$ – *the access rights for pages accessed via a translation that involves the page table entry,*
- $pte.a = pte[7]$ – *the accessed flag that denotes whether the MMU has already used the page table entry for a translation, and*

Definition 10 (Page Table Entry Address)**.** *For a base address $ba \in \mathbb{B}^{20}$ and an index $i \in \mathbb{B}^{10}$, we define the corresponding* page table entry address *as*

$$ptea(ba, i) = ba \circ 0^{12} +_{32} 0^{20} i00$$

The page table entry address needed to extend a given walk $w \in K_{\textbf{walk}}$ is then defined as

$$ptea(w) = ptea(w.ba, (w.va \circ 0^{12}).px_{w.level})$$

Definition 11 (Page Table Entry for a Walk)**.** *Given a memory $m \in K_{\textbf{mem}}$ and a walk $w \in K_{\textbf{walk}}$, we define the* page table entry needed to extend a walk *as*

$$pte(m,w) = m_4(ptea(w))$$

Definition 12 (Walk Creation)**.** *We define the function*

$$winit : \mathbb{B}^{20} \times \mathbb{B}^{20} \times \mathbb{B}^{6} \to K_{\textbf{walk}}$$

which, given a virtual base address $va \in \mathbb{B}^{20}$, the base address $pto \in \mathbb{B}^{20}$ of the page table origin and an address space identifier $asid \in \mathbb{B}^{6}$, returns the initial walk *for the translation of va.*

$$winit(va,pto,asid) = w$$

is given by

$$w.va = va$$

$$w.asid = asid$$

$$w.level = 2$$

$$w.ba = pto$$

$$w.r = 111$$

$$w.fault = 0$$

Note that in our specification of the MMU, the initial walk always has full rights ($w.r = 111$). However, in every translation step, the rights associated with the walk can be restricted as needed by the translation request made by the processor core.

Definition 13 (Sufficient Access Rights)**.** *For a pair of access rights $r, r' \in \mathbb{B}^3$, we use*

$$r \leq r' \quad \stackrel{def}{\Leftrightarrow} \quad \forall j \in [0:2] : r[j] \leq r'[j]$$

to describe that the access rights r are weaker than r', i.e. rights r' are sufficient to perform an access with rights r.

Definition 14 (Walk Extension)**.** *We define the function*

$$wext : K_{walk} \times \mathbb{B}^{32} \times \mathbb{B}^3 \to K_{walk}$$

which extends a given walk $w \in K_{walk}$ using a page table entry $pte \in \mathbb{B}^{32}$ and access rights $r \in \mathbb{B}^3$ in such a way that

$$wext(w,pte,r) = w'$$

is given by

$$w'.va = w.va$$

$$w'.asid = w.asid$$

$$w'.level = \begin{cases} w.level - 1 & pte.p \\ w.level & otherwise \end{cases}$$

$$w'.ba = \begin{cases} pte.ba & pte.p \\ w.ba & otherwise \end{cases}$$

$$w'.r = \begin{cases} pte.r & pte.p \\ w.r & otherwise \end{cases}$$

$$w'.fault = \neg pte.p \vee \neg r \leq pte.r$$

Note that, in the original x86 model, in addition to restricting the rights according to the rights set in the page table entry used to extend the walk, there was the possibility to restrict the rights of a walk even further during walk extension. This has something to do with the fact that translation requests that do not need write rights would not need to set a dirty flag in the page table entry. In this model, however, we only model the accessed bit and not the dirty bit.

Definition 15 (Complete Walk)**.** *A walk $w \in K_{walk}$ with $w.level = 0$ is called a* complete walk*:*

$$complete(w) \equiv w.level = 0$$

Definition 16 (Setting Accessed Flag of a Page Table Entry)**.** *Given a page table entry $pte \in \mathbb{B}^{32}$ and a walk $w \in K_{walk}$, we define the function*

$$\textit{set-d}(pte,w) = pte[a := 1]$$

which returns an updated page table entry in which the accessed bit is set.

Definition 17 (Translation Request)**.** *A translation request*

$$trq = (trq.asid, trq.va, trq.r) \in \mathbb{B}^6 \times \mathbb{B}^{32} \times \mathbb{B}^3$$

is a triple of

- *address space identifier $trq.asid \in \mathbb{B}^6$,*
- *virtual address $trq.va \in \mathbb{B}^{32}$, and*
- *access rights $trq.r \in \mathbb{B}^3$.*

Definition 18 (TLB Hit)**.** *When a walk w matches a translation request trq in terms of virtual address, address space identifier and access rights, we call this a* TLB hit*:*

$$hit(trq,w) \equiv w.va = trq.va[31:12] \wedge w.asid = trq.asid \wedge trq.r \leq w.r$$

Note, that a hit may be to an incomplete walk.

Definition 19 (Page-Faulting Walk Extension)**.** *A page fault for a given translation request can occur for a given walk when extending that walk would result in a fault: The page table entry needed to extend is not present or the translation would require more access rights than the page table entry provides. To denote this, we define the predicate*

$$\begin{aligned} fault(m,trq,w) \equiv \quad & /complete(w) \wedge hit(trq,w) \\ & \wedge wext(w, pte(m,w), trq.r).fault \end{aligned}$$

which, given a memory m, a translation request trq and a walk w, is fulfilled when walk extension for walk w under translation request trq in memory configuration m page-faults.

Note that a page fault may occur at any translation level. However, the TLB will only store non-faulting walks (this is an invariant of the TLB) – page faults are always triggered by considering a faulting extension of a walk in the TLB.

How page faults are triggered is defined in the top-level transition function of MIPS-86 as follows: the processor core always chooses walks from the TLB non-deterministically to either obtain a translation, or, to get a page-fault when the chosen walk has a page faulting walk extension. Note that, when a page-fault for a given pair of virtual address and address space identifier occurs, MIPS-86 flushes all corresponding walks from the TLB. Another side-effect of page-faults in the pipelined hardware implementation is that the pipeline is drained. Since the MIPS-86 model provides a model of sequential instruction execution, draining the pipeline cannot be expressed on this level, however, this behavior is needed in order to be able to prove that the pipelined implementation indeed behaves as specified by MIPS-86.

Definition 20 (Transition Function of the TLB)**.** *We define the transition function of the TLB that states the passive transitions of the TLB*

$$\delta_{\textbf{\textit{tlb}}} : K_{\textbf{\textit{tlb}}} \times \Sigma_{\textbf{\textit{tlb}}} \to K_{\textbf{\textit{tlb}}}$$

where

$$\Sigma_{\textbf{\textit{tlb}}} = \{\textbf{\textit{flush}}\} \times \mathbb{B}^6 \times \mathbb{B}^{20} \cup \{\textbf{\textit{flush-incomplete}}\} \cup \{\textbf{\textit{add-walk}}\} \times K_{\textbf{\textit{walk}}}$$

as a case distinction on the given input:

- *flushing a virtual address for a given address space identifier:*

$$\delta_{\textbf{\textit{tlb}}}(tlb, (\textbf{\textit{flush}}, asid, va)) = \{w \in tlb \mid \neg(w.asid = asid \wedge w.va = va)\}$$

- *flushing all incomplete walks from the TLB:*

$$\delta_{\textbf{\textit{tlb}}}(tlb, \textbf{\textit{flush-incomplete}}) = \{w \in tlb \mid complete(w)\}$$

- *adding a walk:*

$$\delta_{\textbf{\textit{tlb}}}(tlb, (\textbf{\textit{add-walk}}, w)) = tlb \cup \{w\}$$

6 Processor Core

Definition 21 (Processor Core Configuration of *MIPS-86*)**.** *A* MIPS-86 *processor core configuration* $c = (c.pc, c.gpr, c.spr, c.HI, c.LO) \in K_{core}$ *consists of*

- *a program counter:* $c.pc \in \mathbb{B}^{32}$*,*
- *a general purpose register file:* $c.gpr : \mathbb{B}^5 \to \mathbb{B}^{32}$*,*
- *a special purpose register file:* $c.spr : \mathbb{B}^5 \to \mathbb{B}^{32}$*, and*
- *multiplication accumulator registers:* $c.HI, c..LO \in \mathbb{B}^{32}$

Definition 22 (Processor Core Transition Function of *MIPS-86*)**.** *We define the processor core transition function*

$$\delta_{core} : K_{core} \times \Sigma_{core} \rightharpoonup K_{core}$$

which takes a processor core input from

$$\Sigma_{core} = \Sigma_{instr} \times \Sigma_{eev} \times \mathbb{B} \times \mathbb{B}$$

where

$$\Sigma_{instr} = \mathbb{B}^{32} \times (\mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\bot\})$$

is the set of inputs required for instruction execution, i.e. a pair of instruction word $I \in \mathbb{B}^{32}$ *and value* $R \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\bot\}$ *read from memory (which is only needed for* $read$ *or* cas *instructions), and*

$$\Sigma_{eev} = \mathbb{B}^{256}$$

is used to represent a vector $eev \in \mathbb{B}^{256}$ *of interrupt signals provided by the local APIC. Also, we explicitly pass the page fault fetch and page fault load/store signals pff, pfls* $\in \mathbb{B}$*.*

We define the processor core transition function

$$\delta_{core}(c, I, R, eev, pff, pfls) = \begin{cases} \delta_{jisr}(c, I, R, eev, pff, pfls) & jisr(c, I, eev, pff, pfls) \\ \delta_{rfe}(c) & eret(I) \wedge \neg jisr(c, I, eev, pff, pfls) \\ \delta_{instr}(c, I, R) & otherwise \end{cases}$$

as a case distinction on the jump-interrupt-service-routine-*signal* $jisr$ *(for definition, see 6.3) which formalizes whether an interrupt is triggered in the current step of the machine and the* return-from-exception-*signal* rfe *which is active when the next instruction to be executed is* rfe*.*

In the definition above, we use the auxiliary transition functions

$$\delta_{\textbf{instr}} : K_{\textbf{core}} \times \Sigma_{\textbf{instr}} \rightharpoonup K_{\textbf{core}}$$

which executes a non-interrupted instruction of the instruction set architecture (for definition, see section 6.2),

$$\delta_{\textbf{jisr}} : K_{\textbf{core}} \times \Sigma_{\textbf{core}} \to K_{\textbf{core}}$$

which is used to specify the state the core reaches when an interrupt is triggered (for definition, see section 6.4), and

$$\delta_{\textbf{rfe}} : K_{\textbf{core}} \to K_{\textbf{core}}$$

which specifies the return-from-exception transition (for definition, see section 6.4).

6.1 Auxiliary Definitions for Instruction Execution

In the following, we make auxiliary definitions in order to define the processor core transitions that deal with instruction execution. In order to execute an instruction, the processor core needs to read values from the memory. Of relevance to instruction execution is the instruction word $I \in \mathbb{B}^{32}$ and, if the instruction I is a $read$ or cas instruction, we need the value $R \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ read from memory.

6.1.1 Instruction Decoding

Definition 23 (Fields of the Instruction Layout)**.** *Formalizing the tables given in subsection 2.1, we define the following shorthands for the fields of the MIPS-86 instruction layout:*

- instruction opcode
$$opc(I) = I[31:26]$$
- instruction type
$$rtype(I) \equiv opc(I) = 0^6 \vee opc(I) = 010^4 \vee opc(I) = 01^30^2$$
$$jtype(I) \equiv opc(I) = 0^410 \vee opc(I) = 0^411$$
$$itype(I) \equiv \overline{rtype(I) \vee jtype(I)}$$
- register addresses
$$rs(I) = I[25:21]$$
$$rt(I) = I[20:16]$$
$$rd(I) = I[15:11]$$
- shift amount
$$sa(I) = I[10:6]$$
- function code *(used only for R-type instructions)*
$$fun(I) = I[5:0]$$
- immediate constants *(for I-type and J-type instructions, respectively)*
$$imm(I) = I[15:0]$$
$$iindex(I) = I[25:0]$$

Definition 24 (Instruction-Decode Predicates)**.** *For every MIPS-Instruction, we define a predicate on the MIPS-configuration which is true iff the corresponding instruction is to be executed next. The name of such an instruction-decode predicate is always the instruction's mnemonic (see MIPS ISA-tables at the beginning). Formally, the predicates check for the corresponding opcode and function code. E.g.*

$$lw(I) \equiv opc(I) = 100011$$
$$\ldots$$
$$add(I) \equiv rtype(I) \wedge fun(I) = 100000$$

The instruction-decode predicates are so trivial to formalize that we do not explicitly list all of them here.

Definition 25 (Illegal Opcode)**.** *Let*

$$ill(I) = \neg(lw(I) \vee \ldots \vee add(I))$$

be the predicate that formalizes that the opcode of instruction I is illegal by negating the disjunction of all instruction-decode predicates.

Note that, encountering an illegal opcode during instruction execution, an illegal instruction interrupt will be triggered.

6.1.2 Arithmetic and Logic Operations

The *arithmetic logic unit* (ALU) of *MIPS-86* behaves according to the following table:

alucon[3:0]	i	alures	ovf
0 000	*	$a +_{32} b$	0
0 001	*	$a +_{32} b$	$[a] + [b] \notin T_{32}$
0 010	*	$a -_{32} b$	0
0 011	*	$a -_{32} b$	$[a] - [b] \notin T_{32}$
0 100	*	$a \wedge_{32} b$	0
0 101	*	$a \vee_{32} b$	0
0 110	*	$a \oplus_{32} b$	0
0 111	0	$\neg_{32}(a \vee_{32} b)$	0
0 111	1	$b[15:0]0^{16}$	0
1 010	*	$0^{31}([a] < [b]?1:0)$	0
1 011	*	$0^{31}(\langle a \rangle < \langle b \rangle ?1:0)$	0

Based on inputs $a, b \in \mathbb{B}^{32}$, $alucon \in \mathbb{B}^4$ and $i \in \mathbb{B}$, this table defines $alures(a,b,alucon,i) \in \mathbb{B}^{32}$ and $ovf(a,b,alucon,i) \in \mathbb{B}$.

Definition 26 (ALU Instruction Predicates)**.** *To describe whether a given instruction $I \in \mathbb{B}^{32}$ performs an arithmetic or logic operation, we define the following predicates:*

- *I-type ALU instruction:* $compi(I) \equiv itype(I) \wedge I[31:29] = 001$
- *R-type ALU instruction:* $compr(I) \equiv rtype(I) \wedge I[5:4] = 10$
- *any ALU instruction:* $alu(I) \equiv compi(I) \vee compr(I)$

Definition 27 (ALU Operands of an Instruction)**.** *Following the instruction set architecture tables, we formalize the right and left operand of an ALU instruction $I \in \mathbb{B}^{32}$ based on a given processor core configuration $c \in K_{\textbf{core}}$ as follows:*

- *left ALU operand:* $lop(c,I) = c.gpr(rs(I))$
- *right ALU operand:* $rop(c,I) = \begin{cases} c.gpr(rt(I)) & rtype(I) \\ sxt_{32}(imm(I)) & /rtype(I) \wedge /I[28] \\ zxt_{32}(imm(I)) & otherwise \end{cases}$

Definition 28 (ALU Control Bits of an Instruction)**.** *We define the ALU control bits of an instruction $I \in \mathbb{B}^{32}$ as*

$$alucon(I)[2:0] = \begin{cases} I[2:0] & rtype(I) \\ I[28:26] & otherwise \end{cases}$$

$$alucon(I)[3] \equiv rtype(I) \wedge I[3] \vee /I[28] \wedge I[27]$$

Definition 29 (ALU Compute Result)**.** *The ALU result of an instruction I executed in processor core configuration $c \in K_{\textbf{\textit{core}}}$ is then given by*

$$compres(c,I) = alures(lop(c,I), rop(c,I), alucon(I), itype(I))$$

6.1.3 Multiplication Operations

The *multiplication unit* of *MIPS-86* takes inputs a, b, c, d, and $mulcon$ and behaves according to the following table, providing $mures(a,b,c,d,mulcon)$:

mulcon[2:0]	mures
000	$twoc_{64}([a] \cdot [b])$
001	$bin_{64}(\langle a \rangle \cdot \langle b \rangle)$
100	$cd +_{64} twoc_{64}([a] \cdot [b])$
101	$cd +_{64} bin_{64}(\langle a \rangle \cdot \langle b \rangle)$
110	$cd -_{64} twoc_{64}([a] \cdot [b])$
111	$cd -_{64} bin_{64}(\langle a \rangle \cdot \langle b \rangle)$

Definition 30 (Multiplication Control Bits)**.** *The multiplication control bits are*

$$mulcon(I) \equiv (opc(I)[2] \oplus fun(I)[1]) \circ fun[2] \circ fun[0]$$

Definition 31 (Multiplication Result)**.** *We define the* multiplication result *as*

$$mres(c,I) = mures(c.gpr(rs(I)), c.gpr(rt(I)), c.HI, c.LO, mulcon(I))$$

Definition 32 (Multiplication Accumulator Register Instructions)**.** *The predicate*

$$mulacc(I) = mult(I) \vee multu(I) \vee madd(I) \vee maddu(I) \vee msub(I) \vee msubu(I)$$

is true whenever the machine is about to execute a multiplication operation that writes a result to the HI and LO registers.

6.1.4 Jump and Branch Instructions

Jump and branch instructions affect the program counter of the machine. The difference between branch instructions and jump instructions is that branch instructions perform conditional jumps based on some condition expressed over general purpose register values. The following table defines the branch condition result $bcres(a,b,bcon) \in \mathbb{B}$, i.e. whether for the given parameters the branch will be performed or not, based on inputs $a, b \in \mathbb{B}^{32}$ and $bcon \in \mathbb{B}^4$:

bcon[3:0]	bcres(a, b, bcon)
001 0	$[a] < 0$
001 1	$[a] \geq 0$
100 *	$a = b$
101 *	$a \neq b$
110 *	$[a] \leq 0$
111 *	$[a] > 0$

Definition 33 (Branch Instruction Predicates)**.** *We define the following branch instruction predicates that denote whether a given instruction $I \in \mathbb{B}^{32}$ is a jump or successful branch instruction given configuration $c \in K_{\textbf{core}}$:*

- *branch instruction:* $b(I) \equiv opc(I)[5:3] = 0^3 \wedge itype(I)$
- *jump instruction:* $jump(I) \equiv j(I) \vee jal(I) \vee jr(I) \vee jalr(I)$
- *jump or branch taken:*

$$jbtaken(c,I) \equiv jump(I) \vee b(I) \wedge bcres(c.gpr(rs(I)), c.gpr(rt(I)), opc[2:0]rt(I)[0])$$

Definition 34 (Branch Target)**.** *We define the target address of a jump or successful branch instruction $I \in \mathbb{B}^{32}$ in a given configuration $c \in K_{\textbf{core}}$ as*

$$btarget(c,I) \equiv \begin{cases} c.pc +_{32} sxt_{30}(imm(I))00 & b(I) \\ c.gpr(rs(I)) & jr(I) \vee jalr(I) \\ (c.pc +_{32} 4_{32})[31:28]iindex(c)00 & j(I) \vee jal(I) \end{cases}$$

6.1.5 Shift Operations

Shift instructions perform shift operations on general purpose registers.

Definition 35 (Shift Results)**.** *For $a[n-1:0] \in \mathbb{B}^n$ and $i \in \{0, \ldots, n-1\}$ we define the following shift results ($\in \mathbb{B}^n$):*

- *shift left logical:* $sll(a,i) = a[n-i-1:0]0^i$
- *shift right logical:* $srl(a,i) = 0^i a[n-1:i]$
- *shift right arithmetic:* $sra(a,i) = a_{n-1}^i a[n-1:i]$

Note that, for MIPS-86, we will use the aforementioned definitions only for $n = 32$.

Definition 36 (Shift Unit Result)**.** *We define the result of a shift operation based on inputs $a \in \mathbb{B}^n, i \in \{0, \ldots, n-1\}$, and $sf \in \mathbb{B}^2$ as follows:*

$$sures(a,i,sf) = \begin{cases} sll(a,i) & sf = 00 \\ srl(a,i) & sf = 10 \\ sra(a,i) & sf = 11 \end{cases}$$

Definition 37 (Shift Instruction Predicate)**.** *We define a predicate that, given an instruction $I \in \mathbb{B}^{32}$, expresses whether the instruction is a shift instruction by a simple disjunction of shift instruction predicates:*

$$su(I) \equiv sll(I) \vee srl(I) \vee sra(I) \vee sllv(I) \vee srlv(I) \vee srav(I)$$

Definition 38 (Shift Operands)**.** *Given a shift instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\textbf{core}}$, we define the following shift operands:*

- *shift distance:* $sdist(c,I) = \begin{cases} \langle sa(I) \rangle \textbf{ mod } 32 & fun(I)[3] = 0 \\ \langle c.gpr(rs(I))[4:0] \rangle \textbf{ mod } 32 & fun(I)[3] = 1 \end{cases}$
- *shift left operand:* $slop(c,I) = c.gpr(rt(I))$

Definition 39 (Shift Function)**.** *The shift function of a shift instruction $I \in \mathbb{B}^{32}$ is given by*

$$sf(I) = I[1:0]$$

6.1.6 Memory Accesses

We define auxiliary functions that we need in order to define how values are read/written from/to the memory in the overall system's transition function.

Definition 40 (Effective Address and Access Width)**.** *Given an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\textbf{core}}$, we define the effective address and access width of a memory access:*

- *effective address:* $ea(c,I) = \begin{cases} c.gpr(rs(I)) +_{32} sxt_{32}(imm(I)) & itype(I) \\ c.gpr(rs(I)) & rtype(I) \end{cases}$
- *access width:* $d(I) = \begin{cases} 1 & lb(I) \vee lbu(I) \vee sb(I) \\ 2 & lh(I) \vee lhu(I) \vee sh(I) \\ 4 & sw(I) \vee lw(I) \vee cas(I) \end{cases}$

The effective address is the first byte address affected by the memory address and the access width is the number of bytes which are read, or, respectively, written.

Definition 41 (Misalignment Predicate)**.** *For an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{\textbf{core}}$, we define the predicate*

$$\begin{aligned} mal(c,I) \quad \equiv \quad & (lw(I) \vee sw(I) \vee cas(I)) \wedge ea(c,I)[1:0] \neq 00 \\ & \vee (lhu(I) \vee lh(I) \vee sh(I)) \wedge ea(c,I)[0] \neq 0 \end{aligned}$$

that describes whether the memory access is misaligned. To be correctly aligned, the effective address of the memory access must be divisible by the access width.

Note that misaligned memory access triggers the corresponding interrupt.

Definition 42 (Load/Store Instruction Predicates)**.** *In order to denote whether a given instruction $I \in \mathbb{B}^{32}$ is a load or store instruction, we define the following predicates:*

- *load instruction:* $load(I) \equiv lw(I) \vee lhu(I) \vee lh(I) \vee lbu(I) \vee lb(I)$
- *store instruction:* $store(I) \equiv sw(I) \vee sh(I) \vee sb(I)$

Definition 43 (Load Value)**.** *The value read from memory $R \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ is given as an input to the transition function of the processor core. In order to write this value to a general purpose register, depending on the memory instruction used, we either need to sign-extend or zero-extend this value:*

$$lv(R) = \begin{cases} zxt_{32}(R) & lbu(I) \vee lhu(I) \\ sxt_{32}(R) & \end{cases}$$

Definition 44 (Store Value)**.** *Given an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{core}$, the store value is given by the last $d(I)$ bytes taken from the general purpose register specified by $rt(I)$:*

$$sv(c,I) = c.gpr(rt(I))[8 \cdot d(I) - 1 : 0]$$

6.1.7 General Purpose Register Updates

Definition 45 (General Purpose Register Write Predicate)**.** *The predicate*

$$gprw(I) \equiv alu(I) \vee su(I) \vee lw(I) \vee cas(I) \vee jal(I) \vee jalr(I)$$
$$\vee movs2g(I) \vee mul(I) \vee mflo(I) \vee mfhi(I)$$

describes whether a given instruction $I \in \mathbb{B}^{32}$ results in a write to some general purpose register.

Definition 46 (General Purpose Register Result Destination)**.** *We define the result destination of an ALU/shift/coprocessor/memory instruction $I \in \mathbb{B}^{32}$ as the following general purpose register address:*

$$rdes(I) = \begin{cases} rd(I) & rtype(I) \wedge /movs2g(I) \vee mul(I) \\ rt(I) & otherwise \end{cases}$$

Definition 47 (Written General Purpose Register)**.** *For an instruction $I \in \mathbb{B}^{32}$, the address of the general purpose register which is actually written to is defined as*

$$cad(I) = \begin{cases} 1^5 & jal(I) \\ rdes(I) & otherwise \end{cases}$$

Definition 48 (General Purpose Register Input)**.** *We define the value written to the general purpose register specified above based on the instruction $I \in \mathbb{B}^{32}$ and a given processor core configuration $c \in K_{core}$ as*

$$gprdin(c,I,R) = \begin{cases} c.pc +_{32} 4_{32} & jal(I) \vee jalr(I) \\ lv(R) & load(I) \vee cas(I) \\ c.spr(rd(I)) & movs2g(I) \\ c.HI & mfhi(I) \\ c.LO & mflo(I) \\ alures(lop(c,I), rop(c,I), alucon(I)) & alu(I) \\ mres(c,I)[31:0] & mul(I) \\ sures(slop(c,I), sdist(c,I), sf(I)) & su(I) \end{cases}$$

6.2 Definition of Instruction Execution

Based on the auxiliary functions defined in the last subsection, we give the definition of instruction execution in closed form:

Definition 49 (Non-Interrupted Instruction Execution)**.** *We define the transition function for non-interrupted instruction execution*

$$\delta_{instr} : K_{core} \times \Sigma_{instr} \rightharpoonup K_{core}$$

where

$$\Sigma_{\textbf{\textit{instr}}} = \mathbb{B}^{32} \times (\mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\bot\})$$

as

$$\delta_{\textbf{\textit{instr}}}(c, I, R) = \begin{cases} undefined & (load(I) \vee cas(I)) \wedge R \notin \mathbb{B}^{8 \cdot d(I)} \\ c' & otherwise \end{cases}$$

where

- $c'.pc = \begin{cases} btarget(c, I) & jbtaken(c, I) \\ c.pc +_{32} 4_{32} & otherwise \end{cases}$
- $c'.gpr(x) = \begin{cases} gprdin(c, I, R) & x = cad(I) \wedge gprw(I) \\ c.gpr(x) & otherwise \end{cases}$
- $c'.spr(x) = \begin{cases} c.gpr(rt(I)) & rd(I) = x \wedge movg2s(I) \\ c.spr(x) & otherwise \end{cases}$
- $c'.HI = \begin{cases} mres(c, I)[63:32] & mulacc(I) \\ c.gpr(rs(I)) & mthi(I) \\ undefined & mul(I) \\ c.HI & otherwise \end{cases}$
- $c'.LO = \begin{cases} mres(c, I)[31:0] & mulacc(I) \\ c.gpr(rs(I)) & mtlo(I) \\ undefined & mul(I) \\ c.HI & otherwise \end{cases}$

6.3 Auxiliary Definitions for Triggering of Interrupts

MIPS-86 provides the following interrupt types which are ordered by their priority (interrupt level):

interrupt level	shorthand	internal/external	type	maskable	
0	reset	eev	abort	0	reset
1	I/O	eev	repeat	1	devices
2	ill	iev	abort	0	illegal instruction
3	mal	iev	abort	0	misaligned
4	pff	iev	repeat	0	page fault fetch
5	pfls	iev	repeat	0	page fault load/store
6	sysc	iev	continue	0	system call
7	ovf	iev	continue	1	overflow

Note that the all continue interrupts are either triggered by execution of ALU operations with overflow or execution of the sysc-Instruction.

While external event signals are provided by the local APIC as input $eev \in \mathbb{B}^{256}$ to the processor core transition function, the internal event signals $iev(c,I,\textit{pff},\textit{pfls}) \in \mathbb{B}^8$ are defined by the following table that uses the page-fault signals $\textit{pff}, \textit{pfls} \in \mathbb{B}$ which are provided by the MMU of the processor to the processor core transition function.

internal event signal	defined by
$iev(c, I, \mathit{pff}, \mathit{pfls})[2]$	$\equiv ill(I) \vee c.mode[0] = 1 \wedge (movg2s(I) \vee movs2g(I))$
$iev(c, I, \mathit{pff}, \mathit{pfls})[3]$	$\equiv mal(c, I)$
$iev(c, I, \mathit{pff}, \mathit{pfls})[4]$	$\equiv \mathit{pff}$
$iev(c, I, \mathit{pff}, \mathit{pfls})[5]$	$\equiv \mathit{pfls}$
$iev(c, I, \mathit{pff}, \mathit{pfls})[6]$	$\equiv sysc(I)$
$iev(c, I, \mathit{pff}, \mathit{pfls})[7]$	$\equiv ovf(lop(c, I), rop(c, I), alucon(I), itype(I))$

Note that even though, from the view of the processor core, the page-fault signals appear just as external as the external event vector provided by the local APIC, the difference is that the external interrupts provided by the local APIC originate from devices while the page-fault signals originate from the MMU belonging to processor itself. This justifies classifying them as internal event signals.

When an interrupt occurs, information about the type of interrupt is stored in a special purpose register to allow the programmer to discover the reason, i.e. the cause, for the interrupt.

Definition 50 (Cause and Masked Cause of an Interrupt)**.** *We define the cause $ca \in \mathbb{B}^8$ of an interrupt and masked cause $mca \in \mathbb{B}^8$ of an interrupt based on the current processor core configuration $c \in K_{\mathit{core}}$, the instruction $I \in \mathbb{B}^{32}$ to be executed, the external event vector $eev \in \mathbb{B}^{256}$ and the page-fault signals $\mathit{pff}, \mathit{pfls} \in \mathbb{B}$ as follows:*

- *cause of interrupt:*

$$ca(c,I,eev,\mathit{pff},\mathit{pfls})[j] = \begin{cases} iev(c, I, \mathit{pff}, \mathit{pfls})[j] & j \in [2:7] \\ \bigvee_{i=0}^{255} eev[i] & j = 1 \\ 0 & otherwise \end{cases}$$

- *masked cause:*

$$mca(c,I,eev,\mathit{pff},\mathit{pfls})[j] = \begin{cases} ca(c, I, eev, \mathit{pff}, \mathit{pfls})[j] & j \notin \{1, 7\} \\ ca(c, I, eev, \mathit{pff}, \mathit{pfls})[j] \wedge c.spr(sr)[j] & j \in \{1, 7\} \end{cases}$$

Only interrupt levels 1 and 7 are maskable; the corresponding mask can be found in special purpose register sr (status register) and is applied to the cause of interrupt to obtain the masked cause.

Definition 51 (Jump-to-Interrupt-Service-Routine Predicate)**.** *To denote that in a given configuration $c \in K_{\mathit{core}}$ for a given instruction $I \in \mathbb{B}^{32}$, external event signals $eev \in \mathbb{B}^{256}$, and page-fault signals $\mathit{pff}, \mathit{pfls} \in \mathbb{B}$ an interrupt is triggered, we define the predicate*

$$jisr(c,I,eev,\mathit{pff},\mathit{pfls}) \equiv \bigvee_j mca(c, I, eev, \mathit{pff}, \mathit{pfls})[j]$$

Definition 52 (Interrupt Level of the Triggered Interrupt)**.** *To determine the interrupt level of the triggered interrupt, we define the function*

$$il(c,I,eev,\mathit{pff},\mathit{pfls}) = min\{j \mid mca(c, I, eev, \mathit{pff}, \mathit{pfls})[j] = 1\}$$

Definition 53 (Continue-Type Interrupt Predicate)**.** *The predicate*

$$continue(c,I,eev,\mathit{pff},\mathit{pfls}) \equiv il(c, I, R, eev) \in \{6, 7\}$$

denotes whether the triggered interrupt is of continue type.

6.4 Definition of Handling

Definition 54 (Interrupt Execution Transition Function)**.** *We define* $\delta_{\textit{jisr}}(c, I, R, eev, \textit{pff}, \textit{pfls}) = c'$ *where* $I \in \mathbb{B}^{32}$ *is the instruction to be executed and* $eev \in \mathbb{B}^{256}$ *are the event signals received from the local APIC and* $\textit{pff}, \textit{pfls} \in \mathbb{B}$ *are the page-fault signals provided by the processor's MMU.*

Let $k = min\{j \mid eev[j] = 1\}$.

- $c'.pc = 0^{32}$
- $$c'.spr(x) = \begin{cases} 0^{32} & x = sr \\ 0^{32} & x = mode \\ c.sr & x = esr \\ zxt_{32}(mca(c, I, eev, \textit{pff}, \textit{pfls})) & x = eca \\ c.pc & x = epc \wedge /continue(c, I, eev, \textit{pff}, \textit{pfls}) \\ \delta_{\textit{instr}}(c, I, R).pc & x = epc \wedge continue(c, I, eev, \textit{pff}, \textit{pfls}) \\ ea(c, I) & x = edata \wedge il(c, I, eev, \textit{pff}, \textit{pfls}) = 5 \\ bin_{32}(k) & x = edata \wedge il(c, I, eev, \textit{pff}, \textit{pfls}) = 1 \\ c.spr(x) & otherwise \end{cases}$$
- $$c'.gpr = \begin{cases} c.gpr & /continue(c, I, eev, \textit{pff}, \textit{pfls}) \\ \delta_{\textit{instr}}(c, I, R).gpr & otherwise \end{cases}$$

Definition 55 (Return From Exception Transition Function)**.** *We define* $\delta_{\textit{rfe}}(c) = c'$.

- $c'.pc = c.spr(epc)$
- $$c'.spr(x) = \begin{cases} 0^{31}1 & x = mode \\ c.spr(esr) & x = sr \\ c.spr(x) & otherwise \end{cases}$$
- $c'.gpr = c.gpr$

7 Store Buffer

Store buffers are, in their simplest form, first-in-first-out queues for write accesses that reside between processor core and memory. In a processor model with store-buffer, servicing memory reads is done by finding the newest store-buffer entry for the given address if one is available – otherwise the read is serviced by the memory subsystem. Essentially, this means that read accesses that rely on values from preceeding write accesses can be serviced even before they reach the caches. The benefit of store-buffers implemented in hardware is that instruction execution can proceed while the memory is still busy servicing previous write accesses.

In order to allow the programmer to obtain a sequentially consistent view of memory in the presence of store-buffers, architectures whose abstract model contains store-buffers tend to provide instructions that have an explicit effect on the store-buffer, e.g. by draining the pipeline. *MIPS-86* offers a memory fence instruction $fence$ that simply drains the store buffer and a read-modify-write operation rmw that performs an atomic conditional memory update with the side-effect of draining the store-buffer.

Note that, even in a machine that has no store-buffer in hardware, pipelining of instruction execution may introduce a store-buffer to the abstract machine model. We discuss this in the next subsection before we give a definition of the store-buffer of MIPS-86.

7.1 Instruction Pipelining May Introduce a Store-Buffer

The term *pipelining* used in the context of gate-level circuit design can be used to describe splitting up a hardware construction (e.g. of a processor) that computes some result in a single hardware cycle (e.g. executes an instruction) into n smaller components which are called *pipeline stages* whose outputs are always inputs of the next one and which each computes a part of the final result in its own registers – in such a way that, initially, after n cycles, the first result is provided by the nth component and then, subsequently, every following cycle a computation finishes. The reason why this is efficient lies in the fact that, in terms of electrophysics, smaller circuits require less time for input signals to propagate and for output signals to stabilize, thus, smaller circuits can be clocked faster than larger ones. Note that the increase in delay for inserting additional registers in the subcomponents tends to be less than the delay saved by splitting the construction into pipeline stages, resulting in an overall faster computation due to the achieved parallelization.

A common feature to be found in processors is *instruction pipelining*. For a basic RISC machine (like *MIPS-86*), the common five-stage pipeline is given by the following five stages: IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, and WB = Register write back. Note that a naive hardware implementation where all that is changed from the one-cycle hardware construction is that additional registers are inserted will, in general, behave differently than the original construction: Execution of the next instruction may depend on results from the execution of previous instructions which are still in the instruction pipeline. The occurrence of such a dependency where the result of a computation in the naively pipelined machine does not match the result of the sequential machine is referred to as a *hazard*. One way to circumvent this is by software: If the programmer/compiler ensures that the machine code executed by the pipelined machine does not cause any hazard (e.g. by inserting NOOPs (no operations, i.e. instructions that do not have any effect other than incrementing the program counter) or by reordering instructions). This, however, by requiring a much more conservative style of programming, reduces the speedup gained by introducing pipelining in the first place.

In fact, instead of leaving hazard detection and handling exclusively to the programmer of the machine, modern architectures implement proper handling of most hazards in hardware. When a *data hazard* is detected (i.e. an instruction depends on some value computed by an earlier instruction that is still in the pipeline), the hardware *stalls* execution of the pipeline on its own until the required result has been computed. Additional hardware then *forwards* the result from the later pipeline stage directly to the waiting instruction that is stalled in an earlier pipeline stage. Note that even though many hazards can be detected and resolved efficiently in hardware, it is not necessarily the best thing to prevent all hazards in hardware – overall performance of a system may be better when minor parts of hazard handling are left to the programmer/compiler. In fact, for modern pipelined architectures, it is common practice to allow slight changes to the abstract hardware model at ISA level which allow for a less strict but more performant treatment of hazards.

When a memory write is forwarded to a subsequent memory read instruction to

the same address (or to the instruction fetch stage, possibly), this can be modeled by introducing a *store-buffer* between processor and memory system – even when there is no physical store-buffer present in the hardware implementation [Hot12]. For a single-core architecture, it is not overly hard to prove that a processor model with store-buffer is actually equivalent to the processor model without store-buffer. For a multi-core architecture however, it is more involved to prove that store-buffers become invisible on higher layers of abstraction: Since every processor has its own store-buffer and the values from store-buffers are not forwarded to other processors, any two processors may have different values for the same address present in their store-buffers. For an in-detail treatment of hardware construction and correctness for a pipelined simple MIPS machine, see [Pau12].

7.2 Configuration

Definition 56 (Store Buffer Configuration)**.** *The set of store buffer entries is given by*

$$K_{sbe} \equiv \{(a, v) \mid a \in \mathbb{B}^{32} \wedge v \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}\}$$

while the set of store buffer configurations is defined as follows:

$$K_{sb} \equiv K_{sbe}^*$$

We consider a store buffer modeled by a finite sequence of store buffer write accesses (a, v) where $a \in \mathbb{B}^{32}$ is an address and $v \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ is the value to be written starting at memory address a.

7.3 Transitions

A step of the store buffer produces the memory write specified by the oldest store-buffer entry – in order to be sent to the memory subsystem. When the store buffer is empty ($c.sb = \varepsilon$), it cannot make a step. We always append at the front and remove at the end of the list. Transitions of the store buffer are formalized in the overall transition relation – we do not provide an individual transition relation for the store buffer.

7.4 Auxiliary Definitions

We define some auxiliary functions for use in the definition of the system's transition function.

Definition 57 (Store Buffer Entry Hit Predicate)**.** *Given a store buffer entry $(a, w) \in K_{sbe}$ and a byte address $x \in \mathbb{B}^{32}$, we define the predicate*

$$sbehit((a,w),x) \equiv \langle x -_{32} a \rangle < |w|/8$$

which denotes that there is a store buffer hit for the given entry and address, i.e. the address is written by the write access represented by the store buffer entry.

Definition 58 (Store Buffer Hit Predicate)**.** *Given a store buffer configuration $sb \in K_{sb}$ and a byte address $x \in \mathbb{B}^{32}$, the predicate*

$$sbhit(sb,x) \equiv \exists j : sbehit(sb[j], x)$$

denotes whether there is a store buffer hit for the given address in the given store buffer.

Definition 59 (Newest Store Buffer Hit Index)**.** *Given a store buffer configuration $sb \in K_{sb}$ and a byte address $x \in \mathbb{B}^{32}$, we define the function*

$$maxsbhit(sb,x) \equiv \begin{cases} \max\{j \mid sbehit(sb[j],x)\} & sbhit(sb,x) \\ \bot & otherwise \end{cases}$$

which computes the index of the newest entry of the store buffer for which there is a hit or returns the special value $\bot$ if there is no such index.

Definition 60 (Store Buffer Value)**.** *We define the function*

$$sbv : K_{sb} \times \mathbb{B}^{32} \rightharpoonup \mathbb{B}^8$$

which, given a store buffer configuration sb and a byte-address x computes the value forwarded from the store buffer for the given address, if defined:

$$sbv(sb,x) = \begin{cases} byte(\langle x \rangle - \langle a \rangle, v) & sb[maxsbhit(sb,x)] = (a,v) \\ undefined & otherwise \end{cases}$$

Definition 61 (Store Buffer Hazard)**.** *When the processor tries to perform a read access which causes a store-buffer hit but at the same time cannot be serviced by the single newest store buffer entry (e.g. when we have a partial hit for a halfword or word access), we have a* store-buffer hazard[1]*: Given a store-buffer configuration sb, a byte-address a and an access width $d \in \mathbb{N}$, we have*

$$sbhazard(sb,a,d) \equiv$$

$$\exists i,j : 0 \leq i < j < d \wedge maxsbhit(sb, a +_{32} i_{32}) \neq maxsbhit(sb, a +_{32} j_{32})$$

8 Devices

8.1 Introduction to Devices, Interrupts and the APIC Mechanism

Interesting hardware tends to include devices of some kind, e.g. screens, input devices, timers, network adapters, storage devices, or devices that control factory robots, conveyor belts, nuclear power plants, radiation therapy units, etc. In order to trigger interrupts in a processor core, devices tend to provide *interrupt request signals*. Commonly, device interrupt request signals are distinguished in *edge-triggered* signals, i.e. the device signals an interrupt by switching the state of its interrupt request signals, and *level-triggered* signals, i.e. the interrupt request signal is raised when an interrupt is currently pending (i.e. the interrupt request signal has the digital value 1). In a simple hardware implementation of a single-core architecture, level-triggered interrupt request signals originating from devices basically just need to be connected to the processor as external interrupt signals while edge-triggered signals need to be sampled properly and then provided to the processor until it can accept them. Note that in MIPS-86, we restrict interrupt request signals to level-triggered signals.

The processor tends to communicate with devices either by means of *memory mapped input/output* (memory mapped I/O), i.e. device *ports* (which can essentially

[1] In the overall transition relation, the processor core has to wait until the write causing the store-buffer hazard leaves the store-buffer.

be considered user-visible registers of a device) are mapped into the memory space of the processor and accessed with regular memory instructions, or by using special I/O instructions that operate on a seperate port address space where device ports reside. Reading or writing device ports tends to have side-effects, e.g. such as disabling the interrupt request signal raised by the device, allowing the processor to acknowledge that it has received the interrupt, or causing the device to initiate some interaction with the external world.

For a multi-core architecture, device interrupts become more interesting: Is it desirable to interrupt all processors when a device raises an interrupt signal? In many cases, the answer is no: It is fully sufficient to let one processor handle the interrupt. The main question is mostly which device interrupt is supposed to go to which processor. In x86 architectures this is resolved in hardware by providing each processor with a *local advanced programmable interrupt controller* (local APIC) that receives *interrupt messages* from a global *input/output advanced programmable interrupt controller* (I/O APIC) which collects and distributes the interrupt request signals of devices. In order not to transmit a single interrupt to a processor more often than necessary, there is a protocol between I/O APIC and the processor (via the local APIC) in which the processor has to acknowledge the handling of the interrupt by sending an *end of interrupt* (EOI) message via the local APIC. Only after such an EOI message has been received will the I/O APIC sample the corresponding interrupt vector again. Essentially, in the abstract hardware model, both local APIC and I/O APIC can be seen as a special kind of device that does not raise interrupts on its own but can be accessed by the processor by means of memory mapped I/O just like a regular device. Since MIPS-86 implements a greatly simplified version of the x86 APIC mechanism, we will not discuss the detailed x86 APIC mechanism in the following and focus on MIPS-86 in the following.

How exactly the I/O APIC distributes device interrupt signals to the individual processor cores is specified by the *redirect table* – which can be accessed through the I/O APIC ports. This redirect table – which must be set up by the programmer of the machine – specifies the following for each device interrupt request signal: The destination processor, whether the interrupt signal is masked already at the I/O APIC, and the *interrupt vector* to be triggered. The interrupt vector of an interrupt is used to provide information about the cause of the interrupt to the processor. Device interrupt signals are sampled at the I/O APIC and subsequently sent to the destination processor's local APIC over a common bus that connects all local APICs and the I/O APIC. The local APIC associated with a processor core receives interrupt messages from the I/O APIC, collecting interrupt vectors which are then passed to the processor core by raising external interrupt signals at the processor core.

In addition to providing a means of distributing device interrupts, the APIC mechanism offers processor cores of the multi-core system the opportunity to send *interprocessor interrupts* (IPIs). This can, for example, be useful to implement communication between different processors in the context of an operating system. Sending of an inter-processor interrupt is triggered by writing a particular control register belonging to the ports of the local APIC of the processor. The content of this control register describes the destination processors, delivery status, delivery mode and interrupt vector of the inter-processor interrupt. The IPI mechanism is particularly important for booting the machine: Initially, after power on, only the *bootstrap processor* (BSP) is running while all other processors are in a halted state with their local APICs waiting for an initialization inter-processor interrupt (INIT-IPI) and a subsequent startup inter-processor interrupt (SIPI). Effectively, booting the multi-core system can be done by the bootstrap processor in a sequential fashion until it initializes and starts the other

processor cores of the system via the IPI mechanism.

8.2 Configuration

Definition 62 (Device Port Address Length)**.** *We assume a function*

$$dev_{\textbf{palen}} : [0 : nd - 1] \to \mathbb{N}$$

to be given that specifies the address length of port addresses of all $nd \in \mathbb{N}$ *devices of the system in bits.*

Definition 63 (Device Configuration)**.** *The configuration* $d \in K_{\textbf{dev(i)}}$ *of device* i *is given by*

- *I/O ports* $d.ports : \mathbb{B}^{dev_{\textbf{palen}}(i)} \to \mathbb{B}^8$*,*
- *an interrupt request signal* $d.irq \in \mathbb{B}$*, and*
- *internal state* $d.internal \in \mathcal{D}_i$

Note that the $\mathcal{D}_i$ *have to be defined by users of our model to describe the devices they want to argue about.*

8.3 Transitions

Devices react to external inputs provided to them and they have side-effects that occur when their ports are read, or, respectively, written. Note that we currently do not model read-modify-write accesses to devices and we only consider word-accesses on device ports.

Definition 64 (Device Transition Function)**.** *For every device, we assume a transition function to be given of the form*

$$\delta_{\textbf{dev(i)}} : K_{\textbf{dev(i)}} \times \Sigma_{\textbf{dev(i)}} \rightharpoonup K_{\textbf{dev(i)}}$$

with

$$\Sigma_{\textbf{dev(i)}} = \Sigma_{\textbf{ext(i)}} \cup \mathbb{B}^{dev_{\textbf{palen}}} \cup \mathbb{B}^{dev_{\textbf{palen}}} \times (\mathbb{B}^8)^* \cup \mathbb{B}^{32} \times \mathbb{B}^{dev_{\textbf{palen}}} \times \mathbb{B}^{32}$$

where the input to the transition function $in \in \Sigma_{\textbf{dev(i)}}$ *is either*

- *an external input for device* i*:* $in = ext \in \Sigma_{\textbf{ext(i)}}$*,*
- *an external input for device* i*:* $in = a \in \mathbb{B}^{dev_{\textbf{palen}}(i)}$*, or*
- *a word write-access to port address* a *with value* v*:* $in = (a, v) \in \mathbb{B}^{dev_{\textbf{palen}}(i)} \times \mathbb{B}^{32}$.

Note that all active steps of a device are modeled via external inputs, i.e. every active step of the device should be modeled by an input from $\Sigma_{\textbf{ext(i)}}$ that triggers the corresponding step. Further, $\Sigma_{\textbf{ext(i)}}$ can be used to model how the device reacts to the external world.

Depending on the device in question, reading or writing port addresses may have side-effects – for example, deactivating the interrupt request when a specific port is

read. This needs to be specified individually for the given device in its transition function. One restriction we make in this model is that even though reading ports may have side-effects, the value being read is always the one that is given in the I/O ports component of the device. This is reflected in the next section when an overall view of memory with device I/O ports mapped into the physical address space of the machine is defined.

8.4 Device Outputs

We allow devices to provide an output function

$$\lambda_{\textbf{dev(i)}} : K_{\textbf{ext(i)}} \times \Sigma_{\textbf{dev(i)}} \rightharpoonup \Omega_{\textbf{dev(i)}}$$

in order to allow interaction with some external world. This is a partial function, since a device does not need to produce an output for every given external input in a given configuration.

8.5 Device Initial State

To define a set of acceptable initial states of a device after reset, the predicate

$$initial_{\textbf{state(i)}} : K_{\textbf{dev(i)}} \to \mathbb{B}$$

shall be defined.

8.6 Specifying a Device

To specify a particular device of interest, we always need to define the following:

- $\mathcal{D}_i$ – the internal state of the device,
- $\Sigma_{\textbf{ext(i)}}$ – the external inputs the device reacts to,
- $\Omega_{\textbf{dev(i)}}$ – the possible outputs provided by the device,
- $dev_{\textbf{palen}}(i)$ – the length of port addresses of the device,
- $\delta_{\textbf{dev(i)}}$ – the transition function of the device,
- $\lambda_{\textbf{dev(i)}}$ – the output function of the device, and
- $initial_{\textbf{state(i)}}$ – the set of acceptable initial states of the device.

9 Local APIC

The local APIC receives device and inter-processor interrupts sent over the interrupt bus of the system and provides these interrupts to the processor core it is associated with. While the local APIC shares some behavior with devices (i.e. it is accessed by means of memory-mapped I/O) some of its behavior differs significantly from that of devices (i.e. communicating over the interrupt bus instead of raising an interrupt request signal, providing interrupt signals directly to the processor core).

The x86-64 model of [Deg11] provides a local APIC model that describes sending of inter-processor interrupts but ignores devices. While already simplified somewhat

compared to the informal architecture definitions, this model is still quite complex. Thus, Hristo Pentchev provides a simplified version of the x86-64 local APIC model in his upcoming dissertation in order to prove formal correctness of an inter-processor interrupt protocol implementation [ACHP10]. On the one hand, the local APIC model we present in the following is even further simplified – mostly by expressing APIC system transitions atomically instead of in terms of many intermediate steps and by reducing the possible interrupt types and target modes. On the other hand, the model provided here is more powerful in the sense that device interrupts and I/O APIC are modeled.

We have the following simplifications over x86-64:

- We only consider level-triggered interrupts.
- We reduce IPI-delivery to Fixed, INIT and Startup interrupts. The I/O APIC only delivers Fixed interrupts.
- We only model physical destination mode where IPIs are addressed to a local APIC ID (or to a shorthand). We don't consider logical destination mode.
- We do not consider the error-status-register which keeps track of errors encountered when trying to deliver interrupts.

9.1 Configuration

Definition 65 (Local APIC Configuration)**.** *The configuration of a local APIC*

$$apic = (apic.ports, apic.initrr, apic.sipirr, apic.sipivect, apic.eoipending) \in K_{\textbf{\textit{apic}}}$$

consists of

- *I/O-ports* $apic.ports : \mathbb{B}^7 \to \mathbb{B}^8$*,*
- *INIT-request register* $apic.initrr \in \mathbb{B}$*,*

  (a flag that denotes whether an INIT-request is pending to be delivered to the processor)
- *SIPI-request register* $apic.sipirr \in \mathbb{B}$*,*

  (a flag that denotes whether a SIPI-request is pending to be delivered to the processor)
- *SIPI-vector register* $apic.sipivect \in \mathbb{B}^8$*,*

  (the start address for the processor to execute code after receiving SIPI)
- *EOI-pending register* $apic.eoipending \in \mathbb{B}^{256}$

  (a register that keeps track of all interrupt vectors for which an EOI message is to be sent to the I/O APIC)

The I/O ports of the local apic can be accessed by the processor by means of memory mapped I/O. All other local APIC components cannot be accessed by other components. This is reflected in the overall transition relation of the system.

9.1.1 Local APIC ports

Let us define a few shorthands for specific regions in the local APIC ports:

- **APIC ID Register**

$$apic.\textbf{APIC\_ID} = apic.ports_4(0_7)$$

Bits	description
31-28	reserved
27-24	local APIC ID
23-0	reserved

This register contains the local APIC ID of the local APIC. This ID is used when addressing inter-processor-interrupts to a specific local APIC.

- **Interrupt Command Register (ICR)**

$$apic.\textbf{ICR} = apic.ports_8(4_7) \in \mathbb{B}^{64}$$

Bits	abbreviation	description
63-56	$dest$	destination field
55-20		reserved
19-18	dsh	destination shorthand
		00b = no shorthand, 01b = self 10b = all including self, 11b = all excluding self
17-13		reserved
12	ds	delivery status 0b = idle, 1b = send pending
11	$destmode$	destination mode 0b = physical
10-8	dm	delivery mode 000b = Fixed, 101b = INIT, 110b = Startup
7-0	$vect$	vector

This register is used to issue a request for sending an inter-processor interrupt to the local APIC.

- **End-Of-Interrupt Register**

$$apic.\textbf{EOI} = apic.ports_4(12_7) \in \mathbb{B}^{32}$$

Writing to this register is used to signal to the local APIC that the interrupt-service-routine has finished. This has the effect that the local APIC will eventually send an end-of-interrupt acknowledgement to the I/O-APIC.

- **In-Service Register**

$$apic.\textbf{ISR} = apic.ports_{32}(16_7) \in \mathbb{B}^{256}$$

This register keeps track of which interrupt vectors are currently being serviced by an interrupt-service-routine. For our simple processor, maskable interrupts (to which device interrupts belong) are by default masked in the processor core

when an interrupt is triggered. However, when the programmer explicitly unmasks device interrupts during the interrupt handler run, it can happen that a higher-priority interrupt provided by the local APIC may trigger another interrupt, resulting in several interrupt vectors being in service at the same time.

- **Interrupt Request Register**

$$apic.\mathbf{IRR} = apic.ports_{32}(48_7) \in \mathbb{B}^{256}$$

This register keeps track for which interrupt vectors there is currently a request pending. These requests are provided to the processor as external event signals. In this process, all interrupt requests of lower priority than the ones currently in service are masked by the local APIC.

Definition 66 (Processor Core External Event Signals)**.** *We define the external event vector $eev \in \mathbb{B}^{256}$ provided by the local APIC $apic \in K_{\textbf{\textit{apic}}}$ to the processor core as*

$$eev(apic)[j] = \begin{cases} 0 & \exists k \leq j : apic.\textbf{\textit{ISR}}[k] = 1 \\ apic.\textbf{\textit{IRR}}[j] & otherwise \end{cases}$$

9.2 Transitions

We simplify device accesses in such a way that we expect only aligned word-accesses to occur on device ports, i.e. halfword and byte accesses on devices are not modeled.

For all passive steps of the local APIC, we define a transition function

$$\delta_{\textbf{apic}} : K_{\textbf{apic}} \times \Sigma_{\textbf{apic}} \to K_{\textbf{apic}}$$

where

$$\Sigma_{\textbf{apic}} \equiv \mathbb{B}^7 \times \mathbb{B}^{32} \cup \{\textbf{Fixed}, \textbf{INIT}, \textbf{SIPI}\} \times \mathbb{B}^8 \cup \{\textbf{jisr}, \textbf{rfe}\}$$

A passive step of a local APIC is a write access to its ports, a receive-interrupt step, the reaction to a *jisr*-step of the processor core, or the reaction to a *rfe*-step of the processor. We define

$$\delta_{\textbf{apic}}(apic, in) = apic'$$

by a case-distinction:

- write without side-effects:

$$in = (a, v) \wedge a \neq 12_7$$

$$apic'.ports(x) = \begin{cases} byte(\langle x \rangle - \langle a \rangle, v) & in = (a, v) \wedge 0 \leq \langle x \rangle - \langle a \rangle < 4 \\ apic.ports(x) & otherwise \end{cases}$$

- write with side effects (to EOI-register):

$$in = (a, v) \wedge a = 12_7$$

$$apic'.\textbf{EOI} = v$$

$$apic'.\textbf{ISR}[j] = \begin{cases} 0 & j = min\{k \mid apic.\textbf{ISR}[k] = 1\} \\ apic.\textbf{ISR}[j] & otherwise \end{cases}$$

All other local APIC ports stay unchanged.

$$apic'.eoipending[j] = \begin{cases} 1 & j = min\{k \mid apic.\mathbf{ISR}[k] = 1\} \\ apic.eoipending[j] & otherwise \end{cases}$$

Writing to the EOI-Register puts the local APIC in a state where it will send an EOI-message over the interrupt bus in order to acknowledge handling of the highest-priority interrupt to the I/O APIC.

- receiving an interrupt:
  - $in = (\mathbf{Fixed}, vect)$

    $$apic'.\mathbf{IRR}[j] = \begin{cases} 1 & j = \langle vect \rangle \\ apic.\mathbf{IRR}[j] & otherwise \end{cases}$$

    All other ports unchanged.
  - $in = (\mathbf{SIPI}, vect)$

    Receiving a startup-interrupt is successful when there is currently no startup-interrupt pending: If $apic.sipirr \neq 0$, $apic' = apic$ (the local APIC will discard the interrupt), otherwise

    $apic'.ports = apic.ports$

    $apic'.sipirr = 1$

    $apic'.sipivect = vect$

    This records a SIPI which can in turn be used by the local APIC to set the *running* flag of the corresponding processor, effectively starting it.
  - $in = (\mathbf{INIT}, vect)$

    Receiving an INIT-interrupt is successful when there is currently no INIT-interrupt pending: If $apic.initrr \neq 0$, $apic' = apic$, otherwise

    $apic'.ports = apic.ports$

    $apic'.initrr = 1$

    When the local APIC received an INIT-IPI, it will force a reset on the corresponding processor.
- reaction to a *jisr*-step $in = \mathbf{jisr}$:

  $$apic'.\mathbf{IRR}[j] = \begin{cases} 0 & j = min\{k \mid apic.\mathbf{IRR}[k] = 1\} \\ apic.\mathbf{IRR}[j] & otherwise \end{cases}$$

  $$apic'.\mathbf{ISR}[j] = \begin{cases} 1 & j = min\{k \mid apic.\mathbf{IRR}[k] = 1\} \\ apic.\mathbf{ISR}[j] & otherwise \end{cases}$$
- reaction to a *rfe*-step $in = \mathbf{rfe}$:

  $$apic'.\mathbf{ISR}[j] = \begin{cases} 0 & j = min\{k \mid apic.\mathbf{ISR}[k] = 1\} \\ apic.\mathbf{ISR}[j] & otherwise \end{cases}$$

All components not explicitly mentioned stay unchanged between $apic$ and $apic'$.

The active steps of the local APIC (i.e. sending IPIs, sending EOI messages, applying SIPI and INIT-IPI to the processor) are treated in the overall transition relation of the system.

10 I/O APIC

The I/O APIC samples the interrupt request signals of devices and distributes the interrupts to the local APICs according to its redirect table by sending interrupt messages over the interrupt bus. Device interrupts can be masked directly at the I/O APIC.

10.1 Configuration

Definition 67 (I/O APIC Configuration)**.** *The configuration of an I/O APIC*

$$ioapic = (ioapic.ports, ioapic.redirec) \in K_{\textbf{\textit{ioapic}}}$$

is given by

- *I/O-ports* $ioapic.ports : \mathbb{B}^3 \to \mathbb{B}^8$*, and*
- *a redirect table* $ioapic.redirect : [0:23] \to \mathbb{B}^{32}$

10.1.1 I/O APIC Ports

Shorthands to the ports of the I/O APIC are

- select register $ioapic.$**IOREGSEL** $= ioapic.ports_4(0_3)$
- data register $ioapic.$**IOWIN** $= ports_4(4_3)$

Note a pecularity about the I/O APIC: instead of mapping the redirect table into the processor's memory, only a select and a data register are provided. Writing the select register has the side-effect of fetching the redirect table entry specified to the data register. Writing the data register has the side effect of also writing the redirect table entry specified by the select register. Reading from the I/O APIC ports does not have any side-effects.

10.1.2 Format of the Redirect Table

A redirect table entry $e \in \mathbb{B}^{32}$ has the following fields:

Bits	Short	Name	Description
24-31	$dest$	Destination	Local APIC ID of destination local APIC
17-24		reserved	
16	$mask$	Interrupt Mask	masked if set to 1
15		reserved	
14	$rirr$	Remote IRR	0b = EOI received 1b = interrupt was received by local APIC
13		reserved	
12	ds	Delivery Status	1b = Interrupt needs to be delivered to local APIC
11		reserved	
8-10	dm	Delivery Mode	000b = Fixed
0-7	$vect$	Interrupt Vector	

10.2 Transitions

We define a transition function for the passive steps of the I/O APIC

$$\delta_{\textbf{ioapic}} : K_{\textbf{ioapic}} \times \Sigma_{\textbf{ioapic}} \rightharpoonup K_{\textbf{ioapic}}$$

with

$$\Sigma_{\textbf{ioapic}} = \mathbb{B}^3 \times \mathbb{B}^{32} \cup \mathbb{B}^8$$

where $in \in \Sigma$ is either

- $in = (a, v) \in \mathbb{B}^3 \times \mathbb{B}^{32}$ – a write access to port address a with value v,
- $in = vect \in \mathbb{B}^8$ – receiving an EOI message for interrupt vector $vect$

We define $\delta_{\textbf{ioapic}}(ioapic, in) = ioapic'$ by case distinction on in:

- $in = (a, v) \in \mathbb{B}^3 \times \mathbb{B}^{32}$ – write access to the I/O APIC ports
  $\delta_{\textbf{ioapic}}(ioapic, in)$ is undefined iff $a \notin \{0_3, 4_3\}$. Otherwise
  - **Case** $a = 0_3$:
    $ioapic'.\textbf{IOREGSEL} = v$
    $ioapic.\textbf{IOWIN} = ioapic.redirect(\langle v \rangle)$
  - **Case** $a = 4_3$:
    $ioapic'.\textbf{IOWIN} = v$
    $$ioapic'.redirect(i) = \begin{cases} v & i = \langle ioapic.\textbf{IOREGSEL} \rangle \\ ioapic.redirect(i) & otherwise \end{cases}$$
- $in = vect \in \mathbb{B}^8$ – receiving an EOI message for interrupt vector $vect$
  $$ioapic'.redirect(i).rirr = \begin{cases} 0 & ioapic.redirect(i).vect = vect \\ ioapic.redirect(i).rirr & otherwise \end{cases}$$
  Receiving an EOI message for interrupt vector $vect$ resets the corresponding remote interrupt request signal associated with all redirect table entries associated with the interrupt vector. Note that it is adviseable to configure the system in such a way that interrupt vectors assigned to devices are unique.

All components not explicitly mentioned stay unchanged.

Active transitions of the I/O APIC can be found in the definition of the overall transition relation.

11 Multi-Core MIPS

Transitions of the abstract machine are defined as

$$\delta : K \times \Sigma \rightharpoonup K$$

Inputs of the system specify which processor component makes what kind of step and are defined below. On the level of abstraction provided, we assume that the memory subsystem does not make steps on its own, thus it may neither receive external inputs nor be scheduled to make an active step.

The transition functions of the subcomponents are given by

- memory transitions : $\delta_{\mathbf{m}} : K_{\mathbf{m}} \times \Sigma_{\mathbf{m}} \to K_{\mathbf{m}}$ (always passive, section 4),
- processor core transitions : $\delta_{\mathbf{core}} : K_{\mathbf{core}} \times \Sigma_{\mathbf{core}} \rightharpoonup K_{\mathbf{core}}$ (section 6),
- passive TLB transitions : $\delta_{\mathbf{tlb}} : K_{\mathbf{tlb}} \times \Sigma_{\mathbf{tlb}} \to K_{\mathbf{tlb}}$ (section 5, active transitions are given explicitly in the top level transition function),
- store-buffer transitions which are stated explicitly in the top level transition function,
- passive local APIC transitions : $\delta_{\mathbf{apic}} : K_{\mathbf{apic}} \times \Sigma_{\mathbf{apic}} \to K_{\mathbf{apic}}$ (see section 9),
- passive I/O APIC transitions: $\delta_{\mathbf{ioapic}} : K_{\mathbf{ioapic}} \times \Sigma_{\mathbf{ioapic}} \rightharpoonup K_{\mathbf{ioapic}}$ (section 10), and
- device transitions which are given by : $\delta_{\mathbf{dev(i)}} : K_{\mathbf{dev(i)}} \times \Sigma_{\mathbf{dev(i)}} \to K_{\mathbf{dev(i)}}$ (see section 8).

Additionally, we have an output-function

$$\lambda : K \times \Sigma \rightharpoonup \Omega$$

where

$$\Omega = \bigcup_{i=0}^{nd-1} \Omega_{\mathbf{dev(i)}}$$

that allows devices to interact in some way with the external world.

11.1 Inputs of the System

We define

$$\Sigma = \Sigma_{\mathbf{p}} \times [0 : np-1] \cup \Sigma_{\mathbf{ioapic+dev}}$$

as the union of processor inputs and I/O APIC and device inputs. In the following, we define both of them.

Definition 68 (Processor Inputs)**.** *We have*

$$\begin{aligned} \Sigma_{p} \quad = \quad & \{\textbf{\textit{core}}\} \times K_{\textbf{\textit{walk}}} \times K_{\textbf{\textit{walk}}} \cup \{\textbf{\textit{tlb-create}}\} \times \mathbb{B}^{20} \cup \{\textbf{\textit{tlb-extend}}\} \times K_{\textbf{\textit{walk}}} \times \mathbb{B}^{3} \\ & \cup \{\textbf{\textit{tlb-accessed}}\} \times K_{\textbf{\textit{walk}}} \cup \{\textbf{\textit{sb}}\} \\ & \cup \{\textbf{\textit{apic-sendIPI}}, \textbf{\textit{apic-sendEOI}}, \textbf{\textit{apic-initCore}}, \textbf{\textit{apic-startCore}}\} \end{aligned}$$

Note that the processor and the store buffer are both deterministic, i.e. they have only one active step they can perform. In contrast, the TLB and the local APIC are non-deterministic, i.e. there are several steps that can be performed, thus, the scheduling part of the system's input specifies which step is made.

Definition 69 (I/O APIC and Device Inputs)**.** *We have*

$$\begin{aligned} \Sigma_{\textbf{\textit{ioapic+dev}}} \quad = \quad & \{\textbf{\textit{ioapic-sample}}, \textbf{\textit{ioapic-deliver}}\} \times [0 : nd-1] \\ & \cup \{\textbf{\textit{device}}\} \times [0 : nd-1] \times \Sigma_{\textbf{\textit{ext}}} \end{aligned}$$

where

$$\Sigma_{\textbf{\textit{ext}}} = \bigcup_{i \in [0:nd-1]} \Sigma_{\textbf{\textit{ext(i)}}}$$

is the union of all external inputs to devices.

11.2 Auxiliary Definitions

In order to define the overall transition relation, we need a view of the memory that can serve read requests of the processor in the way we expect: depending on the address, a read request can go to a local apic, to the I/O-apic, to a device, to the store-buffer, or, if none of the aforementioned apply, to the memory. Depending on whether the machine is running in user mode or system mode, memory accesses are subject to address translation performed using the TLB component of the machine.

Definition 70 (Local APIC Base Address)**.** *The local APIC ports in this machine are mapped to address*

$$apic_{\textbf{\textit{base}}} \equiv 1^{20}0^{12}$$

Definition 71 (Local APIC Addresses)**.** *The set of byte-addresses covered by local APIC ports is given by*

$$A_{\textbf{\textit{apic}}} = \{a \in \mathbb{B}^{32} \mid 0 \leq \langle a \rangle - apic_{\textbf{\textit{base}}} < 128\}$$

Definition 72 (I/O APIC Base Address)**.** *The I/O APIC ports in this machine are always mapped to address*

$$ioapic_{\textbf{\textit{base}}} \equiv 1^{19}0^{13}$$

Definition 73 (I/O APIC Addresses)**.** *The set of byte-addresses covered by the I/O APIC ports is*

$$A_{\textbf{\textit{ioapic}}} = \{a \in \mathbb{B}^{32} \mid 0 \leq \langle a \rangle - ioapic_{\textbf{\textit{base}}} < 8\}$$

Definition 74 (Device Base Addresses)**.** *We assume a function*

$$dev_{\textbf{\textit{base}}} : [0 : nd - 1] \to \mathbb{B}^{32}$$

to be given that specifies the base address of the ports region of all devices.

Definition 75 (Device Addresses)**.** *The set of addresses covered by device i's ports is given by*

$$A_{\textbf{\textit{dev(i)}}} = \{a \in \mathbb{B}^{32} \mid 0 \leq \langle a \rangle - dev_{\textbf{\textit{base}}}(i) < 2^{dev_{\textbf{\textit{palen}}}}\}.$$

The set of all byte-addresses covered by device, local APIC and I/O APIC ports is defined as

$$A_{\textbf{\textit{dev}}} = \bigcup_{i=0}^{nd-1} A_{\textbf{\textit{dev}}}(i) \cup A_{\textbf{\textit{apic}}} \cup A_{\textbf{\textit{ioapic}}}$$

Definition 76 (Port Address)**.** *Given a memory address x, the corresponding port addresses of devices, local APIC and I/O APIC are computed as*

$$dev_{\textbf{\textit{adr}}}(i,x)(i,x) = (x -_{32} dev_{\textbf{\textit{base}}}(i))[dev_{\textbf{\textit{palen}}}(i) - 1 : 0]$$

$$apic_{\textbf{\textit{adr}}}(x)(x) = (x -_{32} apic_{\textbf{\textit{base}}}(i))[6 : 0]$$

$$ioapic_{\textbf{\textit{adr}}}(x)(x) = (x -_{32} ioapic_{\textbf{\textit{base}}}(i))[2 : 0]$$

Definition 77 (Memory System)**.** *The results of read accesses performed by the processor core are described in terms of a memory system that takes into account device ports, I/O APIC ports, local APIC ports, the store buffer and the memory. We define*

a function ms that, given these components, returns the merged memory view seen by the processor core:

$ms(dev,ioapic,apic,sb,m)(x) =$

$$\begin{cases} sbv(sb,x) & sbhit(sb,x) \\ apic.ports(apic_{\textbf{\textit{adr}}}(x)) & \neg sbhit(sb,x) \wedge x \in A_{\textbf{\textit{apic}}} \\ ioapic.ports(ioapic_{\textbf{\textit{adr}}}(x)) & \neg sbhit(sb,x) \wedge x \in A_{\textbf{\textit{ioapic}}} \\ dev(i).ports(dev_{\textbf{\textit{adr}}}(i,x)) & \neg sbhit(sb,x) \wedge x \in A_{\textbf{\textit{dev}}}(i) \\ m(x) & otherwise \end{cases}$$

Note that, in order to have a meaningful memory system, the machine must be configured in such a way that address ranges of devices, I/O APIC and local APIC are pairwise disjoint.

Definition 78 (Current Address Space Identifier)**.** *The current address space identifier is given by the last* 6 *bits of the special purpose register $asid$:*

$$asid(core) = core.spr(asid)[5:0]$$

11.3 Transitions of the Multi-Core MIPS

Let us define the transition function δ and the output function λ of MIPS-86 by a case distinction on the given input a:

$$\delta(c,a) = c'$$

Any subcomponent of configuration c' that is not listed explicitly in the following has the same value as in configuration c.

- $a = (\textbf{core}, w_I, w_R, i)$ – processor core i performs a step (using walks w_I and w_R if running in translated mode; in system mode, w_I and w_R are ignored)

  In order to formalize a processor core step of processor i, we define the following shorthands:

  - $c_i = c.p(i).core$ – the processor core configuration of processor i,
  - $ms(i) = ms(c.dev, c.ioapic, c.p(i).apic, c.p(i).sb, m)$ – the memory view of processor i,
  - $mode_i = c_i.spr(mode)[0]$ – the execution mode of processor i,
  - $trqI = (asid(c_i), c_i.pc, 011)$ – the translation request for instruction execution if processor core i is running in user mode,
  - *pff* $\equiv mode_i = 1 \wedge fault(c.m, trqI, w_I)$ – signals whether there is a page-fault-on-fetch for the given walk w_I and the translation request $trqI$, and
  - $pmaI = \begin{cases} w_I.pa \circ c_i.pc[11:0] & mode_i = 1 \\ c_i.pc & mode_i = 0 \end{cases}$

    – the physical memory address for instruction fetch of processor core i (which is only meaningful if no page-fault on instruction fetch occurs),
  - $I = ms(i)_4(pmaI)$

    – the instruction fetched from memory for processor core i (in case of a page-fault-on-fetch the value of I has no further relevance),

– $trqEA = (asid(c_i), ea(c_i, I), (store(I) \vee cas(I)) \circ 10)$ – the translation request for the effective address if processor core i is running in user mode,

– *pfls* $\equiv mode_i = 1 \wedge fault(c.m, trqEA, w_R) \wedge /$*pff* $\wedge (store(I) \vee load(I) \vee cas(I))$
– the page-fault-on-load-store signal for processor core i.

– $pmaEA = \begin{cases} w_R.pa \circ ea(c_i, I)[11:0] & mode_i = 1 \\ ea(c_i, I) & mode_i = 0 \end{cases}$
– the physical memory address for the effective address of processor core i,

– $R = \begin{cases} \perp & pff \vee pfls \\ ms(i)_{d(I)}(pmaEA) & otherwise \end{cases}$
– the value read from memory for a $read$ or cas instruction of processor core i,

$\delta(c, a)$ is defined iff all of the following hold:

– $mode_i = 1 \Rightarrow hit(w_I, trqI)$ – running in translated mode, the walk w_I must match the translation request for instruction fetch, and

– $mode_i = 1 \Rightarrow ((store(I) \vee load(I) \vee cas(I)) \wedge \neg pff \Rightarrow (hit(w_R, trqEA)))$ – running in translated mode, if there is a read or write instruction and no page-fault on fetch has occurred, the walk w_R must match the translation request for the effective address, and

– $\neg pff \Rightarrow complete(w_I)$ – if there is no page-fault on fetch, walk w_I is complete, and thus, provides a translation from virtual to physical address, and

– $\neg pfls \Rightarrow complete(w_R)$ – if there is no page-fault on load/store, walk w_R is complete, and thus, provides a translation from virtual to physical address, and

– $(cas(I) \vee mfence(I)) \Rightarrow c.sb = \varepsilon$ – a compare-and-swap or a fence instruction can only be executed when the store-buffer is empty, and

– $load(I) \Rightarrow \neg sbhazard(c.p(i).sb, pmaEA, d(I))$ – there is no store-buffer hazard for the read access the processor tries to perform

– $pmaI \notin A_{\textbf{dev}}$ – we do not fetch instructions from device ports, and

– $(cas(I) \vee d(I) \neq 4 \wedge (load(I) \vee store(I))) \Rightarrow pmaEA \notin A_{\textbf{dev}}$ – we exclude compare-and-swap accesses and byte/halfword accesses to device ports, and

– $c.running(i)$ – only processors that are not waiting for a SIPI can execute.

Then,

$$c'.p(j).core = \begin{cases} \delta_{\textbf{core}}(c_i, I, R, eev(c.p(i).apic), \textit{pff}, \textit{pfls}) & i = j \wedge (load(I) \vee cas(I)) \\ \delta_{\textbf{core}}(c_i, I, \bot, eev(c.p(i).apic), \textit{pff}, \textit{pfls}) & i = j \wedge (\neg load(I) \wedge \neg cas(I)) \\ c.p(j).core & otherwise \end{cases}$$

$$c'.p(j).sb = \begin{cases} (pmaEA, sv(c_i, I)) \circ c.p(i).sb & i = j \wedge store(I) \\ c.p(j).sb & otherwise \end{cases}$$

$$c'.p(j).apic = \begin{cases} \delta_{\textbf{apic}}(c'.p(i).apic, \textbf{jisr}) & j = i \wedge jisr(c, I, eev, \textit{pff}, \textit{pfls}) \\ \delta_{\textbf{apic}}(c'.p(i).apic, \textbf{rfe}) & j = i \wedge eret(I) \\ c.p(j).apic & otherwise \end{cases}$$

$$c'.p(j).tlb = \begin{cases} \emptyset & i = j \wedge flush(I) \\ tlb' & i = j \wedge invlpg(I) \\ \delta_{\textbf{tlb}}(c.p(i).tlb, (\textbf{flush}, asid(c_i), c_i.pc.ba)) & i = j \wedge \textit{pff} \\ \delta_{\textbf{tlb}}(c.p(i).tlb, (\textbf{flush}, asid(c_i), ea(c_i, I).ba)) & i = j \wedge /\textit{pff} \wedge \textit{pfls} \\ c.p(j).tlb & otherwise \end{cases}$$

where

$$tlb' = \delta_{\textbf{tlb}}(\delta_{\textbf{tlb}}(c.p(i).tlb, (\textbf{flush}, c_i.gpr(rs(I))[5:0], c_i.gpr(rd(I)).ba)), \textbf{flush-incomplete})$$

$$c'.m = \begin{cases} \delta_{\textbf{m}}(c.m, (c_i.gpr(rd(I)), pmaEA, sv(c_i, I))) & cas(I) \wedge pmaEA \notin A_{\textbf{dev}} \\ c.m & otherwise \end{cases}$$

$$c'.dev(j) = \begin{cases} \delta_{\textbf{dev(j)}}(c.dev(j), dev_{\textbf{adr}}(j, pmaEA)) & lw(I) \wedge pmaEA \in A_{\textbf{dev(j)}} \\ c.dev(j) & otherwise \end{cases}$$

The flag *running* cannot be modified by a processor core step; it can only be modified by the corresponding local APIC. Local APIC and I/O APIC configurations are never modified by a processor core step since neither local APICs nor I/O APIC have side-effects on reads and we do not allow compare-and-swap accesses to devices – writes to devices always go through the store buffer, thus, any side-effects on device writes are triggered when the write leaves the store buffer.

Performing a processor core step of core i, we apply the processor core transition function to the current processor core configuration, providing the instruction word I read from memory, the read value R (if needed), the external event signals $eev(c.p(i).apic)$ provided by the local APIC belonging to processor i, and the page-fault signals *pff*, and *pfls* given above. In case of a $store$-instruction, the corresponding write access enters processor i's store buffer. If there is a page-fault, the TLB reacts by flushing all translations for the page-faulting address – this is necessary in our model in order to allow the MMU to rewalk the page-tables after interrupt handling without triggering the old page-fault. Only in case of a compare-and-swap instruction, the memory component reacts directly to the compare-and-swap access (since, in all other cases, the store-buffer receives any write requests). Last, in case there is a read-access to a device, the corresponding device transition function is triggered: It specifies how the device reacts to the read-access by specifying appropriate side-effects on reading for the device.

$\lambda(c, a)$ undefined: The processor core does not interact with the external environment, this is exclusive to devices.

Note that continue interrupts can only be caused by execution of instructions that affect only the processor core – thus, continue interrupts are covered in an adequate way in the definitions given here. That is, we do not need to consider changes to other components than the processor core in the case of a continue interrupt.

- $a = (\textbf{sb}, i)$ – a memory write leaves store buffer i

  $\delta(c, a)$ is defined iff $c.p(i).sb \neq \varepsilon$ – the store buffer can only make a step when it is not empty. Then,

  $$c'.p(j).sb = \begin{cases} c.p(i).sb[|c.p(i).sb| - 1 : 1] & i = j \\ c.p(j).sb & i \neq j \end{cases}$$

  $$c'.p(j).apic = \begin{cases} \delta_{\textbf{apic}}(c.p(i).apic, (apic_{\textbf{adr}}(a), v)) & i = j \wedge c.p(i).sb[0] = (a, v) \wedge a \in A_{\textbf{apic}} \\ c.p(j).apic & otherwise \end{cases}$$

  $$c'.m = \begin{cases} \delta_{\textbf{m}}(c.m, c.p(i).sb[0]) & c.p(i).sb[0] = (a, v) \wedge a \notin A_{\textbf{dev}} \\ c.m & otherwise \end{cases}$$

  $$c'.ioapic = \begin{cases} \delta_{\textbf{ioapic}}(c.ioapic, (ioapic_{\textbf{adr}}(a), v)) & c.p(i).sb[0] = (a, v) \wedge a \in A_{\textbf{ioapic}} \\ c.ioapic & otherwise \end{cases}$$

  $$c'.dev(j) = \begin{cases} \delta_{\textbf{dev(j)}}(c.dev(j), (dev_{\textbf{adr}}(j, a), v)) & c.p(i).sb[0] = (a, v) \wedge a \in A_{\textbf{dev(j)}} \\ c.dev(j) & otherwise \end{cases}$$

  Store buffer steps never change processor core configurations, TLB configurations or the *running* flag. The oldest write in the store buffer is handled by the component the address belongs to. Note that here, we rely on the correct alignment of accesses, since otherwise, write accesses might partially cover ports and memory at the same time.

  $\lambda(c, a)$ undefined

- $a = (\textbf{tlb-create}, va, i)$ – a new walk for virtual address va is created in TLB i

  $\delta(c, a)$ is defined iff

  - $c.p(i).spr(mode)[0] = 1$ – the TLB will only create walks when the processor is running in user mode, and
  - $c.running(i)$ – the TLB will only create walks when the processor is not waiting for SIPI.
  - $c.running(i)$ – only when the processor is not waiting for a SIPI, the MMU will start translations.

  Then,

  $$c'.p(j).tlb = \begin{cases} c.p(i).tlb \cup winit(va, c_i.spr(pto).ba, asid(c.p(i)) & i = j \\ c.p(j).tlb & otherwise \end{cases}$$

  Creating a new walk in the TLB is a step that affects only the TLB.

  $\lambda(c, a)$ undefined

- $a = (\textbf{tlb-set-accessed}, w, i)$ – accessed bit of the page table entry needed to extend walk w in TLB i is set

  $\delta(c, a)$ is defined iff

  - $c.p(i).spr(mode)[0] = 1$ – page table entry flags can only be set in translated mode,
  - $w \in c.p(i).tlb \wedge \neg complete(w) \wedge w.asid = asid(c.p(i))$ – we only set the accessed bit for incomplete walks of the current address space identifier, and
  - $pte(c.m, w).p = 1$ – the MMU can only set accessed flags for page table entries which are actually present.

  Then,

  $c'.m = \delta_{\mathbf{m}}(c.m, (ptea(w), \textit{set-d}(pte(c.m, w), w)))$

  Setting the page table entry flags only affects the corresponding page table entry in memory. In this model, the MMU non-deterministically sets the accessed flag – enabling walk extension using the given page table entriy.

  $\lambda(c, a)$ undefined

- $a = (\textbf{tlb-extend}, w, i)$ – an existing walk in TLB i is extended

  $\delta(c, a)$ is defined iff

  - $w \in c.p(i).tlb$ – the walk is to be extended is contained in the TLB, and
  - $\neg complete(w)$ – the walk is not yet complete, and
  - $w.asid = asid(c.p(i))$ – the walk is for the current ASID, and
  - $pte(c.m, w).a \wedge (w.level = 1 \Rightarrow pte(c.m, w).d)$ – the accessed flag is set appropriately, and
  - $\neg wext(w, pte(c.m, w), 000).fault$ – the walk extension does not fault result in a faulty walk, and
  - $c.\textit{running}(i)$ – the TLB will only extend walks when the processor is not waiting for SIPI.

  Then,

  $$c'.p(j).tlb = \begin{cases} \delta_{\textbf{tlb}}(c.p(i).tlb, \textbf{add-walk}(wext(w, pte(c.m, w), 000))\} & i = j \\ c.p(j).tlb & otherwise \end{cases}$$

  Walk extension only affects the TLB, note however, that in order to perform walk extension, the corresponding page-table entry is read from memory.

  $\lambda(c, a)$ undefined

- $a = (\textbf{apic-sendIPI}, i)$ – local APIC i sends a pending inter-processor-interrupt to all target local APICs

  $\delta(c, a)$ is defined iff

  - $c.p(i).apic.\textbf{ICR}.ds = 1$ – there is currently an inter-processor-interrupt to be delivered, and
  - $c.p(i).apic.\textbf{ICR}.destmode \neq 0$ – the destination mode is set to something other than 0

Then,

$$c'.p(j).apic = \begin{cases} \delta_{\textbf{apic}}(apic', (type, vect)) & i = j \wedge \textit{self-target} \\ apic' & i = j \wedge \neg \textit{self-target} \\ \delta_{\textbf{apic}}(c.p(j).apic, (type, vect)) & i \neq j \wedge (t = \textbf{ID} \\ & \quad \wedge c.p(j).apic.\textbf{APIC\_ID} = c.p(i).apic.\textbf{ICR}.dest \\ & \vee t = \textbf{ALL-BUT-SELF} \vee t = \textbf{ALL}) \\ c.p(j).apic & otherwise \end{cases}$$

where

$$vect = c.p(i).apic.\textbf{ICR}.vect[7:0]$$

is the interrupt vector that is sent over the interrupt bus,

$$type = \begin{cases} \textbf{Fixed} & c.p(i).apic.\textbf{ICR}.dm = 0^3 \\ \textbf{INIT} & c.p(i).apic.\textbf{ICR}.dm = 101 \\ \textbf{SIPI} & c.p(i).apic.\textbf{ICR}.dm = 110 \end{cases}$$

is the type of interrupt as specified by the command register of the sending local APIC,

$$t = \begin{cases} \textbf{ALL} & c.p(i).apic.\textbf{ICR}.dsh = 10 \\ \textbf{ALL-BUT-SELF} & c.p(i).apic.\textbf{ICR}.dsh = 11 \\ \textbf{SELF} & c.p(i).apic.\textbf{ICR}.dsh = 01 \\ \textbf{ID} & c.p(i).apic.\textbf{ICR}.dsh = 00 \end{cases}$$

describes the target mode of the requested inter-processor interrupt,

$$\textit{self-target} \equiv (t = \textbf{SELF} \vee t = \textbf{ALL} \vee (t = \textbf{ID} \wedge c.p(i).apic.\textbf{APIC\_ID} = c.p(i).apic.\textbf{ICR}.dest))$$

expresses whether the sending local APIC is also a target of the inter-processor interrupt, and $apic'$ is identical to $c.p(i).apic$ everywhere except $apic'.\textbf{ICR}.ds = 0$.

Sending an inter-processor-interrupt only affects local APIC configurations – both of the sending local APIC and the receiving ones. All receiving local APICs perform a passive receive-interrupt transition.

$\lambda(c, a)$ undefined

- $a = (\textbf{apic-sendEOI}, i)$ – local APIC i sends an EOI message to the I/O APIC

  $\delta(c, a)$ is defined iff

  - $\bigvee_i c.p(i).apic.eoipending[i] = 1$ – there is currently an end-of-interrupt message pending

  Then,

  $$c'.p(j).apic = \begin{cases} apic' & i = j \\ c.p(j).apic & otherwise \end{cases}$$

  where $apic'$ is identical to $c.p(i).apic$ everywhere except

  $$apic'.eoipending[k] = \begin{cases} 0 & k = \langle vect \rangle \\ c.p(i).apic.eoipending[k] & otherwise \end{cases}$$

and $vect = min\{l \mid c.p(i).apic.eoipending[l] = 1\}_8$

$c'.ioapic = \delta_{\textbf{ioapic}}(c.ioapic, vect)$

Transmitting an EOI message affects only the sending local APIC and the I/O APIC. When a local APIC sends an EOI message it is always for the smallest interrupt vector for which an EOI message is pending. The I/O APIC receives the EOI message and reacts with the passive transition given by $\delta_{\textbf{ioapic}}$ that resets the remote interrupt request flag in its redirect table, re-enabling the I/O APIC to sample the corresponding device interrupt.

$\lambda(c, a)$ undefined

- $a = (\textbf{apic-initCore}, i)$ – local APIC i applies a pending INIT-IPI to processor core i

  $\delta(c, a)$ is defined iff $c.p(i).apic.initrr = 1$ – there is currently an INIT-IPI pending in the local APIC of processor i. Then,

  $$c'.p(j).core = \begin{cases} core' & i = j \\ c.p(j).core & otherwise \end{cases}$$

  where

  $core'$ is identical to $c.p(i).core$ except for $core'.pc = 0^{32}$, $core'.spr(mode) = 0^{32}$ and $core'.spr(eca) = 0^{32}$.

  $$c'.\mathit{running}(j) = \begin{cases} 0 & i = j \\ c.\mathit{running}(j) & otherwise \end{cases}$$

  $$c'.p(j).apic = \begin{cases} apic' & i = j \\ c.p(j).apic & otherwise \end{cases}$$

  where $apic'$ is identical to $c.p(i).apic$ everywhere except $apic'.initrr = 0$.

  When the local APIC applies an INIT-IPI to the corresponding processor core, the store buffer and TLB of that processor as well as the global memory and all other devices are not affected. The INIT-IPI effectively acts as a warm reset to the processor core.

  $\lambda(c, a)$ undefined

- $a = (\textbf{apic-startCore}, i)$ – local APIC i applies a pending SIPI interrupt to processor core i

  $\delta(c, a)$ is defined iff

  - $c.p(i).apic.sipirr = 1$ – there is currently a SIPI pending in the local APIC of processor i, and
  - $c.\mathit{running}(i) = 0$ – the processor is currently waiting for SIPI.

  Then,

  $$c'.p(j).core = \begin{cases} core' & i = j \\ c.p(j).core & otherwise \end{cases}$$

  where

  $core'$ is identical to $c.p(i).core$ except for $core'.pc = c.p(i).apic.sipivect \circ 0^{24}$.

$$c'.\mathit{running}(j) = \begin{cases} 1 & i = j \\ c.\mathit{running}(j) & otherwise \end{cases}$$

$$c'.p(j).apic = \begin{cases} apic' & i = j \\ c.p(j).apic & otherwise \end{cases}$$

where $apic'$ is identical to $c.p(i).apic$ everywhere except $apic'.sipirr = 0$.

Similar to the INIT-IPI, the store buffer and TLB of that processor as well as the global memory and all other devices are not affected. The interrupt vector of the SIPI is used to initialize the program counter of the processor core and the flag $c.running(i)$ is set in order to allow the processor core to perform steps.

$\lambda(c, a)$ undefined

- $a = (\textbf{ioapic-sample}, k)$ – the I/O APIC samples the raised interrupt of device k

  $\delta(c, a)$ is defined iff

  - $c.dev(k).irq = 1$ – device k does currently have an interrupt signal activated, and
  - $c.ioapic.redirect(k).mask = 0$ – the interrupt of device k is not masked, and
  - $c.ioapic.redirect(k).rirr = 0$ – handling of any previous interrupt for device k has been acknowledged by an EOI message from the corresponding local APIC.

  Then,

  $c'.ioapic$ is identical to $c.ioapic$ except for

  $$c'.ioapic.redirect(i).ds = \begin{cases} 1 & i = k \\ c.ioapic.redirect(i).ds & otherwise \end{cases}$$

  Sampling a device interrupt only affects the state of the I/O APIC. The delivery status bit of the corresponding redirect table entry is set in order to allow a subsequent **ioapic-deliver** transition.

  $\lambda(c, a)$ undefined

- $a = (\textbf{ioapic-deliver}, k)$ – the I/O APIC delivers a pending interrupt from device k to the target local APIC

  $\delta(c, a)$ is defined iff

  - $c.ioapic.redirect(k).ds = 1$ – there is currently an interrupt pending to be delivered for device k

  Then,

  $$c'.p(j).apic = \begin{cases} \delta_{\textbf{apic}}(c.p(j).apic, (\textbf{Fixed}, v)) & c.p(j).apic.\textbf{APIC\_ID} = c.ioapic.redirect(k).dest \\ c.p(j).apic & otherwise \end{cases}$$

  where $v = c.ioapic.redirect(k).vect$

  and $c'.ioapic$ is identical to $c.ioapic$ except for the following:

  - $c'.ioapic.redirect(k).ds = 0$

– $c'.ioapic.redirect(k).rirr = 1$

Delivering a pending device interrupt only affects the I/O APIC and the target local APIC specified in the redirect table of the I/O APIC. The I/O APIC sets the remote interrupt request flag in order to prevent sampling the same device interrupt several times. Only after the corresponding local APIC acknowledges handling of the interrupt vector to the I/O APIC by sending an EOI message, sampling the device interrupt is possible again.

$\lambda(c, a)$ undefined

- $a = (\textbf{device}, k, ext)$ – device k performs a step under given external input ext

  $\delta(c, a)$ is defined iff

  – $ext \in \Sigma_{\textbf{dev(k)}}$ – the external input belongs to device k.

  Then,

  $$c'.dev(j) = \begin{cases} \delta_{\textbf{dev(k)}}(c.dev(k), ext) & j = k \\ c.dev(j) & otherwise \end{cases}$$

  Only device k is affected. Execution proceeds as specified by the device transition function.

  $\lambda(c, a) = \lambda_{\textbf{dev(k)}}(c.dev(k), ext)$

  The device may provide an output to the external world as specified by its output function.

11.4 Multi-Core MIPS Computation

Given an input sequence $s : \mathbb{N} \to \Sigma$ of the Multi-Core MIPS and a sequence of Multi-Core MIPS configurations (c^i), these two define a Multi-Core MIPS computation if and only if

$$\forall i : \quad c^{i+1} = \delta(c^i, s(i))$$

12 Booting a MIPS-86 Machine

12.1 Initial Configuration after Reset

After reset, an arbitrary configuration c fulfilling the following restrictions is chosen non-deterministically:

- The program counter of all processors is initialized: $\forall i \in [0 : np - 1] : c.p(i).pc = 0^{32}$
- All processors except the boot-strap processor (BSP) are waiting for SIPI:

$$c.\textit{running}(0) = 1$$

$$\forall i \in [1 : np - 1] : \quad c.\textit{running}(i) = 0$$

- All processors are in system mode:

$$\forall i \in [0 : np - 1] : \quad c.p(i).mode = 0^{32}$$

- The store-buffers of all processors are empty:

$$\forall i \in [0 : np-1] : \quad c.p(i).sb = \varepsilon$$

- The TLBs of all processors do not contain translations:

$$\forall i \in [0 : np-1] : \quad c.p(i).tlb = \emptyset$$

- The I/O APIC does not have any interrupts to deliver and all device interrupt lines are masked:

$$\forall i : c.ioapic.redirect(i).rirr = 0$$

$$\forall i : c.ioapic.redirect(i).ds = 0$$

$$\forall i : c.ioapic.redirect(i).mask = 0$$

- The local APICs of all processors do not have pending interrupts and the APIC ID is initialized:

$$\forall i \in [0 : np-1] : \quad c.p(i).apic.\mathbf{IRR} = 0^{256}$$

$$\forall i \in [0 : np-1] : \quad c.p(i).apic.\mathbf{ISR} = 0^{256}$$

$$\forall i \in [0 : np-1] : \quad c.p(i).apic.\mathbf{ICR} = 0^{64}$$

$$\forall i \in [0 : np-1] : \quad c.p(i).apic.\mathbf{APIC\_ID} = i_8$$

- All devices are in an initial state:

$$\forall i \in [0 : nd-1] : \quad initial_{\mathbf{state(i)}}(c.dev(i))$$

12.2 Booting

Since initially, all processors except $c.p(0)$ – the boot-strap processor (BSP) – are waiting for a SIPI-interrupt, the machine executes sequentially until the BSP uses its local APIC to issue SIPIs (startup inter-processor-interrupts) to the other processors.

13 Software Conditions

There are software conditions that must be obeyed.

- Both read and write accesses must be properly aligned.
- Reads of the HI or LO special register must be separated from any subsequent instructions that write to them by two or more instructions.

References

[ACHP10] Eyad Alkassar, Ernie Cohen, Mark Hillebrand, and Hristo Pentchev. Modular Specification and Verification of Interprocess Communication. In *Formal Methods in Computer Aided Design (FMCAD) 2010*. IEEE, 2010.

[Deg11] Ulan Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, Saarland University, Saarbrücken, 2011.

[HIP05] Mark A. Hillebrand, Thomas In der Rieden, and Wolfgang Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, pages 309–316, Washington, DC, USA, 2005. IEEE Computer Society.

[Hot12] Jenny Hotzkow. Store Buffers as an Alternative to Model Self Modifying Code. Bachelor's Thesis, Saarland University, Saarbrücken, 2012.

[Pau12] Wolfgang Paul. A Pipelined Multi Core MIPS Machine - Hardware Implementation and Correctness Proof. URL: `http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur2/ws1112/layouts/multicorebook.pdf`, 2012.