On the Verification of Memory Management Mechanisms

Iakov Dalinger*, Mark Hillebrand*, and Wolfgang Paul

Saarland University, Computer Science Dept., 66123 Saarbrücken, Germany
{dalinger,mah,wjp}@wjpserver.cs.uni-sb.de

Abstract. We report on the design and formal verification of a complex processor supporting address translation by means of a memory management unit (MMU). We give a paper and pencil proof that such a processor together with an appropriate page fault handler simulates virtual machines modeling user computation. These results are crucial steps towards the seamless verification of entire computer systems.

1 Introduction

1.1 The Challenge of Verifying Entire Systems

In the spirit of the famous CLI stack [1] the research of this paper aims at the formal verification of entire computer systems consisting of hardware, compiler, operating system, communication system, and applications. Working with the Boyer-Moore theorem prover [2] the researchers of the CLI stack project succeeded as early as 1989 to prove formally the correctness of a system which provided the following components: a non pipelined processor [3], an assembler [4], a compiler for a simple imperative language [5], a rudimentary operating system kernel [6] written in machine language. This kernel provided scheduling for a fixed number of processes; each process had the right to access a fixed interval of addresses in the processor's physical memory. An attempt to access memory outside these bounds lead to an interrupt. Interprocess communication and system calls apparently were not provided.

From 1989 to 2002 to the best of our knowledge no project aiming at the formal verification of entire computer systems was started anywhere. In [7] J S. Moore, principal researcher of the CLI stack project, declares the formal verification of a system 'from transistor to software level' a grand challenge problem. A main goal of the Verisoft project [8] funded by the German Federal Government is to solve this challenge.

This paper makes two necessary steps towards the verification of entire complex systems. (i) We report about the formal verification of a processor with memory management units (MMUs). MMUs provide hardware support for address translation; address translation is needed to implement address spaces provided by modern operating

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38. Work of the second author was also partially funded by IBM Entwicklung GmbH Böblingen.

systems. (ii) We present a paper and pencil correctness proof for a virtual memory emulation based on a very simple page fault handler. As the formal treatment of I/O devices is an open problem [7] we state the correctness of a swap memory driver as an axiom.

In companion papers we address the verification of I/O devices, of a compiler for a C-like language with in-line assembler code, and of an operating system kernel [9–11].

1.2 Overview of This Paper

In Sect. 2 we briefly review the standard formal definition of the DLX instruction set architecture (ISA) for virtual machines. We emphasize interrupt handling. In Sect. 3 on physical machines we enrich the ISA by the standard mechanisms for operating system support: (i) user and system mode; (ii) address translation in user mode. In Sect. 4 we present a construction of a simple MMU and prove its correctness under nontrivial operating conditions. In pipelined processors separate MMUs are used for instruction fetch and load / store. In Sect. 5 we show how the operating conditions for both MMUs can be guaranteed by hardware *and* software implementation. Sect. 6 gives the main new arguments of the processor correctness proof under these software conventions. In Sect. 7 we present a simple page fault handler. We show that a physical machine with this handler emulates a virtual machine. In Sect. 8 we conclude and sketch further work.

1.3 Related Work

The processor verification presented here extends work on the VAMP presented in [12, 13]. The treatment of external interrupts is in the spirit of [14, 15]. Formal proofs are in PVS [16] and—except for limited use of its model checker—interactive. All formal specifications and proofs are on our website.[1] We stress that some central lemmas in [12, 14] (e.g. on Tomasulo schedulers) have similar counterparts that can be proven using the rich set of automatic methods for hardware verification. How to profit from these methods in correctness proofs of entire processors continues to be an amazingly difficult topic of research. Some recent progress is reported in [17].

As for the new results of this paper: we are not aware of previous work on the verification of MMUs. We are also not aware of previous theoretical work on the correctness of virtual machine simulations.

2 Virtual Machines

2.1 Notation

We denote the concatenation of bit strings $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ by $a \circ b$. For bits $x \in \{0,1\}$ and positive natural numbers $n \in \mathbb{N}^+$ we define inductively $x^1 = x$ and $x^n = x^{n-1} \circ x$. Thus, for instance $0^5 = 00000$ and $1^2 = 11$.

Overloading symbols like $+$, $\cdot$, and $<$ we will allow arithmetic on bit strings $a \in \{0,1\}^n$. In these cases arithmetic is binary modulo 2^n (with nonnegative representatives). We will consider $n = 32$ for addresses or registers and $n = 20$ for page indices.

[1] `http://www-wjp.cs.uni-sb.de/forschung/projekte/VAMP/`

Table 1. Special purpose registers. Indices 01100 to 01111 are not assigned.

Address	Name	Meaning
00000	SR	Status register
00001	ESR	Exception status reg.
00010	ECA	Exception cause reg.
00011	EPC	Exception PC
00100	EDPC	Exception DPC
00101	Edata	Exception data
00110	RM	Rounding mode
00111	IEEEf	IEEE flags
01000	FCC	Floating point (FP) condition code
01001	pto	Page table origin
01010	ptl	Page table length
01011	Emode	Exception mode
10000	mode	Mode

We model memories m as mappings from addresses a to byte values $m(a)$. For natural numbers d we denote by $m_d(a)$ the content of d consecutive memory cells starting at address a, so $m_d(a) = m(a+d-1) \circ \cdots \circ m(a)$. For $d = 4\mathrm{K} = 2^{12}$ and a a multiple of $4\mathrm{K}$, we call $m_d(a)$ a *page* and $4\mathrm{K}$ the *page size*. We split virtual addresses $va = va[31:0]$ into page index $va.px = va[31:12]$ and byte index $va.bx = va[11:0]$. Thus, $va = va.px \circ va.bx$. For page indices px and memories m we abbreviate $page(m, px) = m_{4\mathrm{K}}(px \circ 0^{12})$.

2.2 Specifying the Instruction Set Architecture

Virtual machines are the hardware model visible for user processes. Its parameters are:

- The number V of pages of accessible virtual memory. This defines the set of accessible virtual addresses $VA = \{a \mid 0 \leq a < V \cdot 4\mathrm{K}\}$.
- The number $e \in \mathbb{N}$ of external interrupt signals.
- The set $VSA \subseteq \{0,1\}^5$ of addresses of user visible special purpose registers. Table 1 shows the entire set of special purpose registers that will be visible for a *physical* machine. For the virtual machine only the registers RM, $IEEEf$, and FCC will be visible. Hence $VSA = \{00110, 00111, 01000\}$.
- The status register $SR \in \{0,1\}^{32}$. This is the vector of mask bits for the interrupts.

Formally, the configuration of a virtual machine is a 7-tuple $c_{\mathrm{V}} = (c_{\mathrm{V}}.PC, c_{\mathrm{V}}.DPC, c_{\mathrm{V}}.GPR, c_{\mathrm{V}}.FPR, c_{\mathrm{V}}.SPR, c_{\mathrm{V}}.vm, c_{\mathrm{V}}.p)$ with the following components:

- The normal program counter $c_{\mathrm{V}}.PC \in \{0,1\}^{32}$ and the delayed program counter $c_{\mathrm{V}}.DPC \in \{0,1\}^{32}$, used to implement the delayed branch mechanism (cf. [15]).
- The general purpose register file $c_{\mathrm{V}}.GPR : \{0,1\}^5 \to \{0,1\}^{32}$, the floating point register file $c_{\mathrm{V}}.FPR : \{0,1\}^5 \to \{0,1\}^{32}$, and the special purpose register file $c_{\mathrm{V}}.SPR : VSA \to \{0,1\}^{32}$.
- The byte addressable virtual memory $c_{\mathrm{V}}.vm : VA \to \{0,1\}^8$.
- The write protection function $c_{\mathrm{V}}.p : \{va.px \mid va \in VA\} \to \{0,1\}$. Virtual addresses in the same page have the same protection bit.

Let C_{V} be the set of virtual machine configurations. An instruction set architecture (ISA) is formally specified as a transition function $\delta_{\mathrm{V}} : C_{\mathrm{V}} \times \{0,1\}^e \to C_{\mathrm{V}}$ mapping configurations $c_{\mathrm{V}} \in C_{\mathrm{V}}$ and a vector of external event signals $eev \in \{0,1\}^e$ to the next

configuration $c'_{\mathrm{V}} = \delta_{\mathrm{V}}(c_{\mathrm{V}}, eev)$. For the DLX instruction set we outline the formal definition of this function emphasizing interrupt handling.

The instruction $I(c_{\mathrm{V}}) = c_{\mathrm{V}}.vm_4(c_{\mathrm{V}}.DPC)$ to be executed in configuration c_{V} is found in the four bytes in virtual memory starting at the address of the delayed PC. The opcode $opc(c_{\mathrm{V}}) = I(c_{\mathrm{V}})[31:26]$ consists of the leading six bits of the instruction. Many instructions can be decoded just from the opcode, e.g. a load word instruction is recognized by $lw(c_{\mathrm{V}}) = (opc(c_{\mathrm{V}}) = 100011)$. The type of an instruction determines how the bits outside the opcode are interpreted. For instance, if the opcode consists of all zeros we have an R-type instruction, $R\text{-}type(c_{\mathrm{V}}) = (opc(c_{\mathrm{V}}) = 0^6)$. Other instruction types are defined in a similar way. Depending on the instruction type the register destination address $RD(c_{\mathrm{V}})$ is found at different positions in the instruction, namely $RD(c_{\mathrm{V}}) = I(c_{\mathrm{V}})[15:11]$ if $R\text{-}type(c_{\mathrm{V}})$ and $RD(c_{\mathrm{V}}) = I(c_{\mathrm{V}})[20:16]$ otherwise. Similarly, one can define register source addresses $RS1(c_{\mathrm{V}})$ and $RS2(c_{\mathrm{V}})$, the sign extended immediate constant $simm(c_{\mathrm{V}})$, etc. The effective address of a load / store instruction is computed as the sum of the general purpose register addressed by $RS1(c_{\mathrm{V}})$ and the sign extended immediate constant, $ea(c_{\mathrm{V}}) = c_{\mathrm{V}}.GPR(RS1(c_{\mathrm{V}})) + simm(c_{\mathrm{V}})$. A load word instruction reads four bytes of virtual memory starting at address $ea(c_{\mathrm{V}})$ into the general purpose register addressed by $RD(c_{\mathrm{V}})$. This can be expressed by equations like $lw(c_{\mathrm{V}}) \implies (c'_{\mathrm{V}}.GPR(RD(c_{\mathrm{V}})) = c_{\mathrm{V}}.vm_4(ea(c_{\mathrm{V}})))$.

Components of the configuration that are not listed on the right-hand side of the implication are meant to be unchanged. This definition, however, ignores both internal and external interrupts; therefore even for virtual machines it is an oversimplification.

2.3 Interrupts

We define a predicate $JISR(c_{\mathrm{V}}, eev)$ (jump to interrupt service routine) depending on both the current configuration c_{V} and the current values $eev \in \{0,1\}^e$ of the external interrupt event signals. Only if this signal stays inactive does the above equation hold, so $(\neg JISR(c_{\mathrm{V}}, eev) \wedge lw(c_{\mathrm{V}})) \implies (c'_{\mathrm{V}}.GPR(RS1(c_{\mathrm{V}})) = c_{\mathrm{V}}.vm_4(ea(c_{\mathrm{V}})))$.

For physical machines an activation of the $JISR$ signal has a well defined effect on the program counters and the special purpose registers. The effect on virtual machine computations however is that control is handed over to the operating system kernel. This effect can only be defined in a model that includes the operating system kernel.[2]

For the definition of signal $JISR(c_{\mathrm{V}}, eev)$ for physical machines, we consider the 32 interrupts from Table 2 with indices $j \in IP = \{0, \ldots, 31\}$. For virtual machines we ignore page fault interrupts, thus we only consider $j \in IV = IP \setminus \{3, 4\}$. The activation of signal $JISR(c_{\mathrm{V}}, eev)$ can be caused by the activation of external interrupt lines $eev[j]$ or internal interrupt event signals $iev(c_{\mathrm{V}})[j]$. We define the cause vector by $ca(c_{\mathrm{V}}, eev)[j] = eev[0]$ for $j = 0$, by $ca(c_{\mathrm{V}}, eev)[j] = eev[j-12]$ for $j > 0$ external, and by $ca(c_{\mathrm{V}}, eev)[j] = iev(c_{\mathrm{V}})[j]$ otherwise.

Formally, external interrupts are input signals for the next state computation while internal interrupts are functions of the current configuration. E.g. a definition of the misalignment signal is

$$mal(c_{\mathrm{V}}) = iev(c_{\mathrm{V}})[2] = \neg(4 \mid c_{\mathrm{V}}.DPC) \vee (ls(c_{\mathrm{V}}) \wedge \neg(d(c_{\mathrm{V}}) \mid ea(c_{\mathrm{V}})))$$

[2] We do not treat this further; see the (german) lecture notes [18] or [9] for details.

Table 2. Interrupts

j	Name	Meaning	Mask.	Ext.
0	reset	Reset	No	Yes
1	ill	Illegal instruction	No	No
2	mal	Misaligned access	No	No
3	pff	Page fault on fetch	No	No
4	pfls	Page fault on load / store	No	No
5	trap	Trap	No	No
6	xovf	Fixed point overflow	Yes	No
7	fovf	FP overflow	Yes	No
8	funf	FP underflow	Yes	No
9	finx	FP inexact result	Yes	No
10	fdbz	FP division by zero	Yes	No
11	finv	FP invalid operation	Yes	No
12	ufop	Unimpl. FP operation	No	No
>12	$io[j]$	Device interrupt j–12	Yes	Yes

with $u \mid v$ indicating divisibility, $ls(c_{\mathrm{V}})$ indicating the presence of a load / store instruction, and $d(c_{\mathrm{V}}) \in \{1, 2, 4, 8\}$ indicating its memory access width in bytes.

For virtual machines, but not for physical machines, reading or writing special purpose registers other than RM, $IEEEf$, and FCC is illegal. Reading or writing these registers is achieved with commands `movi2s` or `movs2i`; the register address is given by the instruction field $SA(c_{\mathrm{V}}) = I(c_{\mathrm{V}})[10:6]$. Thus the illegal instruction signal $ill(c_{\mathrm{V}}) = iev(c_{\mathrm{V}})[1]$ has an implicant $(movi2s(c_{\mathrm{V}}) \vee movs2i(c_{\mathrm{V}})) \wedge (SA(c_{\mathrm{V}}) \notin VSA)$.

The interrupt cause for a maskable interrupt j is ignored if the associated status register bit $SR[j]$ is zero. So, we define the masked vector mca by $mca(c_{\mathrm{V}}, eev)[j] = ca(c_{\mathrm{V}}, eev) \wedge c_{\mathrm{V}}.SR[j]$ for j maskable and $mca(c_{\mathrm{V}}, eev)[j] = ca(c_{\mathrm{V}}, eev)$ otherwise. An interrupt occurs if at least one masked cause bit is on; so, $JISR(c_{\mathrm{V}}, eev) = 1$ iff there exists $j \in IV$ with $mca(c_{\mathrm{V}}, eev)[j] = 1$.

3 Physical Machines

Physical machines are the sequential programming model of the hardware as seen by the programmer of an operating system kernel. Compared with virtual machines, more details are visible in configurations $c_{\mathrm{P}} \in C_{\mathrm{P}}$ of physical machines.

- All special purpose registers are visible. Formally $c_{\mathrm{P}}.SPR : PSA \to \{0,1\}^{32}$ with $PSA \subseteq \{0,1\}^5$ consisting of the addresses in Table 1. We abbreviate $c_{\mathrm{P}}.x = c_{\mathrm{P}}.SPR(x)$ where x is the name of a special purpose register.
  The mode register $c_{\mathrm{P}}.mode$ distinguishes between system mode ($c_{\mathrm{P}}.mode = 0$) and user mode. In system mode accessing special purpose registers is legal.
- Page faults are visible; in the definition of $JISR$ the full set of indices IP is used.
- For physical machines the next state $\delta_{\mathrm{P}}(c_{\mathrm{P}}, eev)$ is defined also for an active signal $JISR(c_{\mathrm{P}}, eev)$, starting execution of the interrupt service routine (ISR) in system mode. See [15] for details. In system mode physical machines can legally execute an `rfe` (return from exception) instruction.
- Instead of a uniform virtual memory the (system) programmer now sees two memories: physical memory $c_{\mathrm{P}}.pm$ and swap memory $c_{\mathrm{P}}.sm$.
- In user mode accesses to physical memory are translated.

In the remainder of this section we specify a single-level translation mechanism and model I/O operations with the swap memory.

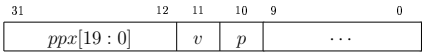

Fig. 1. Page Table Entry

3.1 Address Translation

In user mode, i.e. if $c_{\mathrm{P}}.mode = 1$, memory accesses to virtual addresses $va = c_{\mathrm{P}}.DPC$ and $va = ea(c_{\mathrm{P}})$ are subject to address translation: they either cause a page fault or are redirected to the translated physical memory address $pma(c_{\mathrm{P}}, va)$.

Let us define $pma(c_{\mathrm{P}}, va)$ first. The page table entry address for virtual address va is defined as $ptea(c_{\mathrm{P}}, va) = c_{\mathrm{P}}.pto \cdot 4\mathrm{K} + 4 \cdot va.px$ and its page table entry is defined as $pte(c_{\mathrm{P}}, va) = c_{\mathrm{P}}.pm_4(ptea(c_{\mathrm{P}}, va))$. As shown in Fig. 1, the page table entry is composed of three components, the physical page index $ppx(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[31:12]$, the valid bit $v(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[11]$, and the protection bit $p(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[10]$. We define the physical memory address by concatenating the physical page index and the va's byte index $pma(c_{\mathrm{P}}, va) = ppx(c_{\mathrm{P}}, va) \circ va.bx$.

For the definition of page faults, let the flag $w \in \{0, 1\}$ be active for write operations. The page fault flag $pf(c_{\mathrm{P}}, va, w)$ is set if (i) the virtual page index $va.px$ is greater or equal the number of accessible pages $V = c_{\mathrm{P}}.ptl + 1$, (ii) the valid bit $v(c_{\mathrm{P}}, va)$ is false, or (iii) the write flag w and the protection bit $p(c_{\mathrm{P}}, va)$ are active, indicating a write attempt to a protected page. So, overall $pf(c_{\mathrm{P}}, va, w) = (va.px \geq V) \vee \neg v(c_{\mathrm{P}}, va) \vee w \wedge p(c_{\mathrm{P}}, va)$. Thus, all entries $pte(c_{\mathrm{P}}, va)$ with $pf(c_{\mathrm{P}}, va, w) = 0$ are located in the *page table* $PT(c_{\mathrm{P}}) = c_{\mathrm{P}}.pm_{4 \cdot V}(c_{\mathrm{P}}.pto \circ 0^{12})$.

A page fault on fetch occurs if $pff(c_{\mathrm{P}}) = c_{\mathrm{P}}.mode \wedge pf(c_{\mathrm{P}}, c_{\mathrm{P}}.DPC, 0)$. In the absence of such a fault, we define the instruction word by $I(c_{\mathrm{P}}) = c_{\mathrm{P}}.pm_4(iaddr(c_{\mathrm{P}}))$ where $iaddr(c_{\mathrm{P}}) = pma(c_{\mathrm{P}}, c_{\mathrm{P}}.DPC)$ in user mode and $iaddr(c_{\mathrm{P}}) = c_{\mathrm{P}}.DPC$ otherwise. Let $ls(c_{\mathrm{P}})$ and $s(c_{\mathrm{P}})$ indicate the presence of a load / store resp. a store instruction. In the absence of a page fault on fetch, a page fault on load / store occurs if $pfls(c_{\mathrm{P}}) = c_{\mathrm{P}}.mode \wedge ls(c_{\mathrm{P}}) \wedge pf(c_{\mathrm{P}}, ea(c_{\mathrm{P}}), s(c_{\mathrm{P}}))$.

Multi-level address translation can be formally specified similarly, see e.g. [19].

3.2 Modeling an I/O Device

In order to handle page faults, one has to be able to transfer pages between the physical memory $c_{\mathrm{P}}.pm$ and the swap memory $c_{\mathrm{P}}.sm$, implemented with an I/O device. For a detailed (minimal) treatment of this process four things are necessary:

1. Define I/O ports as a portion of memory shared between the CPU and the device.
2. Specify the detailed protocol of the I/O devices.
3. Construct a driver program, say, with three parameters passed on (distinct) fixed addresses in physical memory: a physical page index $ppxp(c_{\mathrm{P}})$, a swap memory page index $spxp(c_{\mathrm{P}})$, and a physical-to-swap flag $p2s(c_{\mathrm{P}})$ indicating whether the page transfer is from physical to swap memory ($p2s(c_{\mathrm{P}}) = 1$) or vice versa.

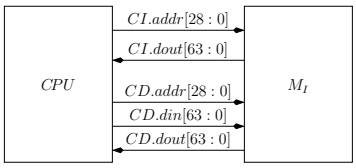

Fig. 2. Memory Interface

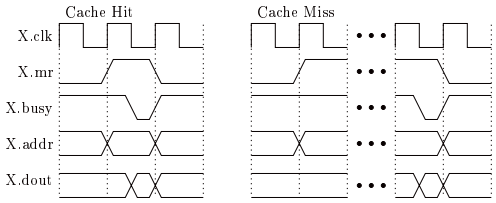

Fig. 3. Timing Diagrams for Read Accesses

4. Show: if the driver is started in configuration c_{P} and never interrupted, it eventually reaches a configuration c'_{P} with

$$page(c'_{\mathrm{P}}.sm, spxp(c_{\mathrm{P}})) = page(c_{\mathrm{P}}.pm, ppxp(c_{\mathrm{P}})) \quad \text{if} \quad p2s(c_{\mathrm{P}}) = 1\,;$$
$$page(c'_{\mathrm{P}}.pm, ppxp(c_{\mathrm{P}})) = page(c_{\mathrm{P}}.sm, spxp(c_{\mathrm{P}})) \quad \text{if} \quad p2s(c_{\mathrm{P}}) = 0\,.$$

5. Furthermore show: (i) program control returns to the location of the call of the driver, (ii) except for certain book keeping information no other parts of the configuration change, and (iii) the driver never leaves its own code region.

Here, we assume the existence of a correct driver as an axiom; in [11] we deal with this problem on a fundamental level.

4 Construction and Local Correctness of MMUs

We refer to the hardware configuration by h. Its components are registers $h.R$, often shortly written as R. For cycles t and hardware signals or registers x we denote by x^t the value of x during cycle t.

4.1 Memory Interface

We construct MMUs for processors with two first level caches, an instruction cache CI for fetches and a data cache CD for load / store instructions. Thus the CPU communicates with the memory system via two sets of busses: one connecting the CPU with the instruction cache and the other one with the data cache (data bus width is 64 bits, cf. Fig. 2). We use the same protocol for both busses. Examples of the protocol are shown in Fig. 3 for a read access with and without a cache hit. The properties of the bus protocol are:

1. Accesses last from the activation of a request signal (in the example mr) until the busy signal is turned off. Optimally, this happens in the same cycle.
2. Read and write requests may not be given simultaneously: $\neg(mr \wedge mw)$
3. During an access, CPU inputs to the memory system must be kept stable.
4. Liveness: if Conditions 2 and 3 are fulfilled, every access eventually ends.

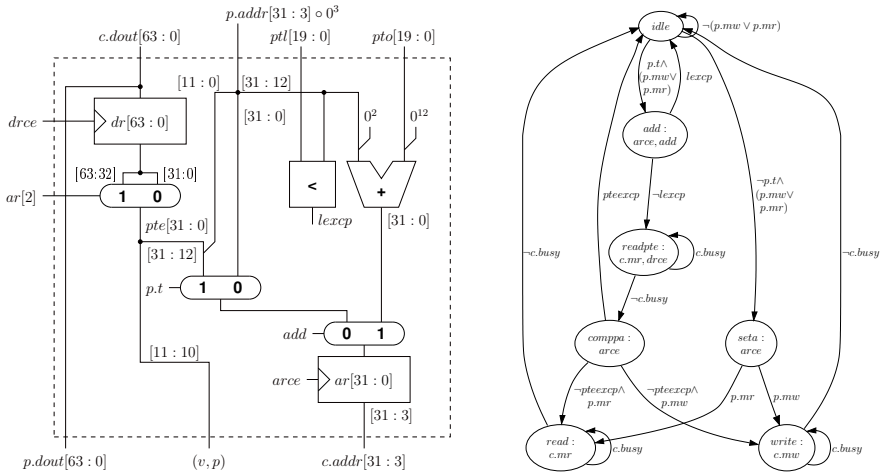

Fig. 4. MMU Datapaths and Control Automaton.

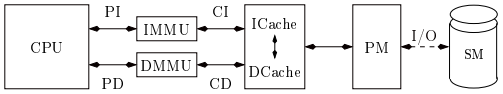

Fig. 5. Processor and MMUs

The memory system satisfies shared memory semantics: for cycles t, for $0 \leq b < 8$, and addresses a we define $last_b(a,t)$ as the last cycle t' before t, when a write access to byte b of address a ended (necessarily via the data cache). Now assume a read access to cache X with address a ends in cycle t. Then the result on bus $X.dout$ is $X.dout^t[8 \cdot b + 7 : 8 \cdot b] = CD.din^{last_b(a,t)}[8 \cdot b + 7 : 8 \cdot b]$. This definition permits to define the state of the two port memory system $m(h)$ at time t by $m(h^t)(a \cdot 8 + b) = CD.din^u$ where $u = last_b(a,t)$. For a formal and complete version of this definition (including initialization), the construction of a split cache system, and a transcript of a formal correctness proof, see [13, Pages 1–110]. Guaranteeing that the CPU keeps inputs stable (Condition 3) during *all* accesses (even when an interrupt is detected deeper down in the pipeline) requires the construction of *stabilizer circuits* for both ports of the memory system. For details see [13, Section 4.4].

4.2 MMU Construction and Operating Conditions

Figure 4 shows datapaths and control automaton of a simple non-optimized MMU implementation. Two copies of this MMU are placed between the CPU and the caches as shown in Fig. 5. In user mode this MMU will only perform address translation under non trivial operating conditions. Consider an access of the CPU to the MMU lasting from a start cycle ts to an end cycle $te \geq ts$. We have to require that no signal or register x from the groups below changes during the access, so $x^t = x^{ts}$ holds for $ts \leq t \leq te$.

G1. Inputs from the CPU to the MMU; these are $p.dout$, $p.addr$, $p.mr$, and $p.mw$.
G2. The CPU registers $h.mode$, $h.pto$, and $h.ptl$ relevant for translation.
G3. In case of a translated access the page table entry used for translation, the shared memory content $m(h)_4(ptea)$ with $ptea = h.pto \cdot 4\text{K} + 4 \cdot p.addr.px$.
G4. For reads with physical address pa, the shared memory content $m(h)_8(pa)$.

Analogous to Sect. 3.1 one can define for hardware configurations h and virtual addresses va a page table entry address $ptea(h, va)$, a page table entry $pte(h, va)$, and a physical memory address $pma(h, va)$. Note that under the operating conditions the virtual address va, the translation $pma(h, va)$, and, for reads, the data read from the memory stay the same during the whole access.

Assuming these operating conditions, the MMU's correctness proof is relatively straightforward. Guaranteeing them will be a considerably tougher issue.

4.3 Local MMU Correctness

There is an obvious case split on the kind and result of the access: (i) read / write, (ii) translated / untranslated, (iii) with / without exception. For each of the cases two lemmas about the control and the datapath of the MMU have to be proven. The proofs of these lemmas are easy and not given here. For example, the next two lemmas state the correctness for a translated read without exception. In this case, the page table entry and the memory operand are read in states $readpte$ and $read$ resp. By s^+ we denote the fact that the control stays in state s until the busy signal is taken away by the cache.

Lemma 1. *For a translated read without exception the path through the control automaton is* $idle \rightarrow add \rightarrow readpte^+ \rightarrow comppa \rightarrow read^+ \rightarrow idle$.

Lemma 2. *The result* $p.din^{te}$ *of a translated read without exception from a virtual address* $va = p.addr^{ts} \circ 0^3$ *is* $p.dout^{te} = m(h^{ts})_8(pma(h^{ts}, va))$.

5 Guaranteeing the Operating Conditions

Stable inputs from the CPU to the MMUs (Condition $G1$) can be guaranteed by using stabilizer circuits similar to those mentioned in Sect. 4.1. Condition $G4$ for loads can be guaranteed if stores are performed in-order by the memory unit. Guaranteeing the remaining operating conditions (Conditions $G2$, $G3$, and $G4$ for fetch) requires a software convention *and* a hardware construction.

5.1 Software Synchronization Convention

Consider sequential computations of the physical machine $(c_{\mathrm{P}}^0, c_{\mathrm{P}}^1, \ldots)$. Formally, for all steps i we have $c_{\mathrm{P}}^{i+1} = \delta_{\mathrm{P}}(c_{\mathrm{P}}^i, eev^i)$. Recall that for such machines the instruction address $iaddr(c_{\mathrm{P}})$ depends on $c_{\mathrm{P}}.mode$ (cf. Sect. 3.1) and the instruction $I(c_{\mathrm{P}})$ fetched in configuration c_{P} is defined as $I(c_{\mathrm{P}}) = c_{\mathrm{P}}.pm_4(iaddr(c_{\mathrm{P}}))$.

We define an instruction as *synchronizing* if the pipeline of the processor is drained before the (translation of the) fetch of the next instruction starts. The VAMP processor

already has such an instruction, namely a `movs2i` instruction with $IEEEf$ as source.[3] We now also define the `rfe` instruction as synchronizing and let the predicate $sync(c_{\mathrm{P}})$ indicate that instruction $I(c_{\mathrm{P}})$ is synchronizing.

Synchronizing instructions must be used to resolve RAW hazards for instruction fetch to prevent modification of an instruction in a pipelined machine after it has already been (pre-) fetched. Formally, let $u < w$ be two instruction indices. We require the existence of an index v with $u < v < w$ and $sync(c_{\mathrm{P}}^{v})$ under the following two conditions: 1. If $I(c_{\mathrm{P}}^{u})$ writes to $iaddr(c_{\mathrm{P}}^{w})$. 2. If $I(c_{\mathrm{P}}^{u})$ writes to the page table entry at address $ptea(c_{\mathrm{P}}^{w}.DPC)$ that is read for *user mode* instruction fetch. The first condition is already needed in pipelined machines without address translation [12, 14].

Clearly, Condition 1 addresses operating condition $G4$ in case of a fetch, whereas Condition 2 addresses $G3$. In hardware one has to address operating condition $G2$ and to implement pipeline drain once a synchronizing instruction is decoded.

5.2 Hardware Mechanisms for Synchronization

The VAMP processor has a two stage pipeline for instruction fetch and instruction decode, followed by a Tomasulo scheduler. For details see [12, 13, 20]. Thus, there are many register stages S, e.g. IF for instruction fetch and ID for instruction decode.

The clocking and stalling of individual stages is achieved by a *stall engine*. For an introduction to stall engines see [15]; for improvements see [13, 20]. Three crucial data structures / signals are associated with each stage S in the stall engine:

1. The full bit $full_S$ is on if stage S has meaningful data. Clearing it flushes the stage.
2. The local busy signal $busy_S$ is on if the circuits with inputs from register stage S do not produce meaningful data at the end of a cycle.
3. The update enable signals ue_S is like a clock enable signal. If ue_S is active in a cycle, the stage S receives new data in the next cycle.

Let $busy'_{IF}$ be the busy signal of the instruction fetch stage of the VAMP without MMUs. We define a new busy signal by $busy_{IF}(h) = busy'_{IF}(h) \vee \neg fetch(h)$ where the signal $fetch(h)$ is almost the read signal for the instruction MMU (as noted before, the read signal of the instruction MMU is stabilized to satisfy $G1$).

Signal $fetch$ is turned on if (i) no instruction changing registers pto, ptl and $mode$ is in progress and (ii) no synchronizing instruction is in decode. Instructions in progress can be in the instruction decode stage, i.e. in its instruction register I, or they are issued but not completed, thus they are in the Tomasulo scheduler and its data structures. In a Tomasulo scheduler an instruction in progress which changes a register r from a register file is easily recognized by an inactive valid bit $r.v$. Thus we define $fetch(h) = h.pto.v \wedge h.ptl.v \wedge h.mode.v \wedge fetch'(h)$ where function $fetch'(h)$ has to take care of instructions in the decode stage. Using predicates like $rfe()$ which are already defined for configurations also for the contents of the instruction register, we define

$$fetch'(h) = \neg(h.full_{ID} \wedge (sync(I) \vee movi2s(I) \vee rfe(I))) .$$

In the VAMP processor synchronizing instructions stay in the instruction decode stage until they can immediately proceed to the write-back stage.

[3] This instruction reads the floating point interrupts accumulated *so far*.

6 Processor Correctness

6.1 Correctness Criteria

We are using correctness criteria based on scheduling functions from [13–15, 20]. Register stages S of the hardware configuration h come in three flavours:

- Visible stages (with respect to the physical machine from Section 3): these stages are (i) PCs with the program counters $h.PC$, $h.DPC$, (ii) RF with the register files $h.GPR$, $h.SPR$, and $h.FPR$, (iii) stage mem' with the specified memory. This memory is not represented directly by hardware registers; instead it is simulated by the memory system with caches with the function $m(h)$ (cf. Sect. 4.1).
- Invisible stages: the registers of these stages store intermediate results used in the definition of the sequential physical machine. Stage ID with the instruction register $h.IR$ stores values $I(c_{\mathrm{P}})$, stage mem with the address input register $h.PD.addr$ for the data MMU stores $ea(c_{\mathrm{P}})$, etc.
- Stages from the data structures of the Tomasulo scheduler.

We map hardware stages S and hardware cycles t to instruction numbers i via the scheduling function sI. Assume $sI(S,t) = i$. The intention is to relate the contents of the registers in stage S in hardware configuration h^t to the physical machine configuration c^i_{P} (and its derived components). We distinguish the following cases.

For visible registers R from stages $S \neq mem'$ we require $h^t.R = c^i_{\mathrm{P}}.R$. Thus the specified value of visible hardware register R in cycle t is the same as the value of R in the specification machine before execution of the i-th instruction. Similarly, we require for the stage $S = mem'$ that $m(h^t) = c^i_{\mathrm{P}}.pm$ and for invisible registers R in stage S that $h^t.R = R(c^i_{\mathrm{P}})$. Specific correctness criteria are used for the data structures of the Tomasulo scheduler. For details see [20].

The three main definitions for scheduling functions that make this work are: (i) In-order fetch: The fetch scheduling function is incremented if the instruction decode stage receives a new instruction, $sI(fetch, t+1) = sI(fetch, t) + 1$ for $ue^t_{ID} = 1$, and stays unchanged otherwise. (ii) The scheduling of a stage S' that is not updated does not change. Hence, $ue^t_{S'} = 0$ implies $sI(S', t+1) = sI(S', t)$. (iii) If data is clocked in cycle t from stage S to S' we set $sI(S', t+1) = sI(S, t) + 1$ if S' is visible and otherwise $sI(S', t+1) = sI(S, t)$.

Thus intuitively an instruction number $i = sI(S, t)$ accompanies the data through the pipeline; upon reaching a register in a visible stage S' however, the register receives the value *after* the i-th instruction, i.e. before instruction $(i+1)$.

6.2 Correctness Proof with External Interrupt Signals

In general pipelined processors do not finish execution of one instruction per cycle. As there are more cycles t than instructions i there are necessarily more external interrupt events signals eev^t_{h} at the hardware level than event signals eev^i seen by the sequential physical machine. For the computation of the latter, given as $c^{i+1}_{\mathrm{P}} = \delta_{\mathrm{P}}(c^i_{\mathrm{P}}, eev^i)$, one has to define the interrupt signals eev^i seen by the physical machine from the signals eev^t_{h} seen by the hardware machine. This has already been observed in [14, 15].

The VAMP processor, as most processors with Tomasulo schedulers, *samples* external interrupt signals in the write-back stage. Each instruction i is in this stage only for a single cycle. Call this cycle $t = WB(i)$. The correctness proof then works with $eev^i = eev^t_{\mathrm{h}}$. It is a matter of protocol between processor and devices that no harm comes from that, i.e. no interrupts are lost [11].

6.3 Correctness Proof

We give the new part of the VAMP correctness proof for a translated instruction fetch without exceptions. The other new cases are handled similarly. Thus consider a translated read access on the instruction port of the CPU lasting from cycle ts to cycle te. Let $i = sI(fetch, ts)$ and let $t \in \{ts, \ldots, te\}$ be any cycle of the access. Let us abbreviate the address of the double word containing instruction $I(c^i_{\mathrm{P}})$ by $va := c^i_{\mathrm{P}}.DPC[31:3] \circ 0^3$. From program counter correctness we conclude that in cycle t the address bus of the instruction MMU holds the (upper 29 bits) of va, so $PI.addr(h^t) = va[31{:}3]$.

Let $i_1 = sI(RF, t) \leq i$ be the instruction in the register file stage in cycle t. By the construction of the fetch signal all instructions $x < i$ that update a special purpose register $R \in \{pto, ptl, mode\}$ have already left the pipe at cycle ts (also no instruction $x > i$ can enter the pipe while instruction $I(c^i_{\mathrm{P}})$ is being fetched). By additionally using the correctness criterion for R, we may conclude for t as above that $c^i_{\mathrm{P}}.R = c^{i_1}_{\mathrm{P}}.R = h^t.R$ and hence $pa_1 := ptea(c^i_{\mathrm{P}}, va) = ptea(h^t, va)$.

Let $i_2 = sI(mem', t)$. By Condition 2 of the software sync-convention all instructions $x < i$ that write to the address pa_1 have left the pipe already at cycle ts. Using correctness of the memory stage we get $c^i_{\mathrm{P}}.pm_4(pa_1) = c^{i_2}_{\mathrm{P}}.pm_4(pa_1) = m(h^t)_4(pa_1)$ and therefore $pa_2 := pma(c^i_{\mathrm{P}}, va) = pma(h^t, va)$. By Condition 1 of the software sync-convention all instructions that write to the physical memory address pa_2 have left the pipe at cycle ts. As above we get $c^i_{\mathrm{P}}.pm_8(pa_2) = c^{i_2}_{\mathrm{P}}.pm_8(pa_2) = m(h^t)_8(pa_2)$.

Hence the operating conditions for the MMU are fulfilled and at time te it returns the double word $PI.dout(h^{te}) = m(h^{ts})_8(pa_2) = c^i_{\mathrm{P}}.pm_8(pa_2)$. By selecting the appropriate half of this double word via bit 2 of the delayed program counter, at the end of cycle te we clock $I(c^i_{\mathrm{P}})$ into the instruction register I. Since $sI(ID, te+1) = i$, we have shown hardware correctness for the considered case:

Lemma 3. $h^{te+1}.I = I(c^i_{\mathrm{P}}) = I(c^{sI(ID,te+1)}_{\mathrm{P}})$

7 Virtual Machine Simulation

In this section we outline an *informal* proof that a physical machine with a page fault handler can simulate virtual machines (here: only a single one). Making these arguments precise is not trivial; we give some details in Sect. 8.

We extend the definitions of physical page index $ppx(c_{\mathrm{P}}, va)$ and valid bit $v(c_{\mathrm{P}}, va)$ to page indices by $ppx(c_{\mathrm{P}}, px) = ppx(c_{\mathrm{P}}, px \circ 0^{12})$ and $v(c_{\mathrm{P}}, px) = v(c_{\mathrm{P}}, px \circ 0^{12})$.

7.1 Memory Map of the Physical Machine

We partition the physical memory $c_{\mathrm{P}}.pm$ into user memory and system memory, cf. Fig. 6. Addresses below $abase \cdot 4\mathrm{K}$ are used by the page fault handler and the swap

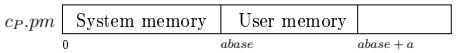

Fig. 6. Memory Map. Addresses are given as page indices.

memory driver. Starting at address $abase \cdot 4\text{K}$ we allocate $a > 1$ pages of user memory with indices $UP = \{a' \in \{0,1\}^{20} \mid abase \leq a' < abase + a\}$. Likewise, we have a swap page index $sbase$ and use $sma(va) = sbase \cdot 4\text{K} + va$ to store va on swap.

We list below the data structures used by the handler and some invariants:

- A process control block PCB to save the registers of the virtual processor.
- The page table PT as defined by the address translation mechanism (Sect. 3.1).
- The physical page index MRL of the most recently loaded page.
- A variable $b \in \{-1, \ldots, a-1\}$ and an array D of size a holding virtual page indices. User page indices $0 \leq u \leq b$ we call *full*; we require for them $v(c_\text{P}, D[u]) \wedge ppx(c_\text{P}, D[u]) = abase + u$ and $D[u] < V$ where $V = c_\text{P}.ptl + 1$ denotes the number of accessible virtual pages. Otherwise, for $b < u < a$ we require $\neg v(c_\text{P}, D[u])$. Hence, valid translations map to the user memory, which is of crucial importance.
- Parameters $ppxp$, $spxp$, and $p2s$ of the swap memory driver (cf. Sect. 3.2).

7.2 Simulation Relation

For virtual machine configurations c_V and physical machine configurations c_P we define a simulation relation $B(c_\text{V}, c_\text{P})$ stating that c_P encodes c_V. We require that the invariants of the previous subsection hold for the physical machine and that the physical machine is in user mode ($c_\text{P}.mode = 1$). Furthermore: (i) The write protection function is encoded in the protection bits of the page tables. Formally, for all virtual addresses va we require $c_\text{V}.p(va) = p(c_\text{P}, va)$. (ii) The user memory acts as a (write-back) cache for the swap memory. For virtual page indices px we require $page(c_\text{V}.vm, px) = page(c_\text{P}.pm, ppx(c_\text{P}, px))$ if $v(c_\text{P}, px)$ and $page(c_\text{V}.vm, px) = page(c_\text{P}.sm, sbase + px)$ otherwise.

Lemma 4 (Step lemma). *Let c_V and c_P be as above, assume no page fault in configuration c_P. Then, without external interrupts $B(c_\text{V}, c_\text{P}) \implies B(\delta_\text{V}(c_\text{V}, 0^e), \delta_\text{P}(c_\text{P}, 0^e))$.*

7.3 Page Fault Handler and Software Conditions

We describe a very simple handler that is never interrupted itself. Thus the handler needs only to save the general purpose registers of the physical processor into the PCB. Via the exception cause ECA we determine, if a page fault occurred. For page fault on fetch, $ECA[3:0] = 10^3$; for page fault on load / store, $ECA[4:0] = 10^4$. The virtual address xva causing the page fault is $xva = EDPC$ in the former case, $xva = EDATA$ else. It is easy to deal with page table length or protection exceptions: we stop the simulation. Thus assume a page fault occurred in a configuration c_P because the exception virtual page was invalid. Moreover assume $B(c_\text{V}, c_\text{P})$ for a virtual machine configuration c_V. From this we get $page(c_\text{P}.sm, sbase + xv) = page(c_\text{V}.vm, xv)$ where $xv = xva.px$.

If $b < a$, not all user pages are full. We increment b and let $e = abase + b$ denote the physical page index where we later swap in the exception virtual page.

Otherwise, a victim physical page index vp must be selected from the user pages. The most recently loaded page is never chosen to avoid deadlock, so $vp \in UP \backslash \{MRL\}$. Let $vp = abase + u$. Using the table D we determine the matching victim virtual page index $vv = D[u]$ of the virtual page stored at physical page vp. Because $B(c_{\mathrm{V}}, c_{\mathrm{P}})$ holds and $ppx(c_{\mathrm{P}}, vv) = abase + u = vp$ we have

$$page(c_{\mathrm{V}}.vm, vv) = page(c_{\mathrm{P}}.pm, ppx(c_{\mathrm{P}}, vv)) = page(c_{\mathrm{P}}.pm, vp)\ .$$

We copy the victim page to swap memory by running the driver with parameters $(ppxp, spxp, p2s) = (vp, sbase{+}vv, 1)$. Then we clear the valid bit of page vv, reaching a configuration c'_{P} with $v(c'_{\mathrm{P}}, vv) = 0$ and $page(c'_{\mathrm{P}}.sm, sbase + vv) = page(c_{\mathrm{P}}.pm, vp) = page(c_{\mathrm{V}}.vm, vv)$. Thus, the simulation relation $B(c_{\mathrm{V}}, c'_{\mathrm{P}})$ still holds. We set $e = vp$.

Now we swap in the exception virtual page to the physical page with index e by running the driver with parameters $(ppxp, spxp, p2s) = (e, sbase + xv, 0)$. We end up in a configuration c''_{P} with $page(c''_{\mathrm{P}}.pm, e) = page(c_{\mathrm{P}}.sm, sbase + xv) = page(c_{\mathrm{V}}.vm, xv)$. Then we update the page table entry of xv and the data structures by $v(c'''_{\mathrm{P}}, xv) = 1$, by $ppx(c'''_{\mathrm{P}}, xv) = e$, by $D[e - abase] = xv$, and by $MRL = e$ in a later configuration c'''_{P}. Thus, $B(c_{\mathrm{V}}, c'''_{\mathrm{P}})$ and the invariants hold for c'''_{P}. Finally, the handler restores the user registers from the PCB and executes an `rfe` instruction. By inspection of the handler we see that the software sync-convention holds.

7.4 Simulation Theorem

Theorem 1. *For all computations $(c^0_{\mathrm{V}}, c^1_{\mathrm{V}}, \ldots)$ of the virtual machine there is a computation $(c^0_{\mathrm{P}}, c^1_{\mathrm{P}}, \ldots)$ of the physical machine and there are step numbers $(s(0), s(1), \ldots)$ such that for all i and $S = s(i)$ we have $B(c^i_{\mathrm{V}}, c^S_{\mathrm{P}})$.*

Proof. We prove the claim by induction on i. We assume that the initialization code establishes after a certain number of steps $S = s(0)$ that $b = -1$, all virtual pages are invalid and stored in swap memory, and the simulation relation $B(c^0_{\mathrm{V}}, c^S_{\mathrm{P}})$ holds.

Concluding from i to $i + 1$, we examine the configuration after the next non-page-faulting user step. We set $s(i + 1) = \min\{s' \geq s(i) \mid c^{s'}_{\mathrm{P}}.mode \wedge \neg pfls(c^{s'}_{\mathrm{P}}) \wedge \neg pff(c^{s'}_{\mathrm{P}})\} + 1$. The minimum always exists since the victim page of a page fault is not the page swapped in for the previous page fault. Thus, there are zero to two page faults from steps $s(i)$ to $s(i+1)-1$; for $s(i+1) = s(i)+1$ one step of the virtual machine is simulated in one step of the physical machine. The theorem's claim is implied by page fault handler correctness and the step lemma (Sects. 7.2 and 7.3).

8 Summary and Further Work

We have presented two main results. First, we have reported on the formal verification of the VAMP with (simple) MMUs (Sects. 4 to 6). The correctness proof for an MMU alone is simple, but depends on nontrivial operating conditions. Guaranteeing these requires a variety of arguments, from intricate arguments about the hardware (e.g.

Sect. 5.2) to the format of page fault handlers (Sect. 7.3). Second, arguing on low level software we have shown that physical machines with suitable page fault handlers simulate virtual machines. Since operating systems support multitasking and virtual memory, these results are crucial steps towards verifying entire computer systems.

Presently we see three directions for further work. (i) The formal verification of processors with memory-mapped I/O devices, pipelined MMUs, multi level translation and translation look aside buffers. A mathematical model of a hard disk can be found in [11]. (ii) The formal proof of our virtual memory simulation theorem. This is part of an ongoing effort to verify an entire operating system kernel in the Verisoft project [8]. Mathematical proofs can be found in [18]. (iii) The verification of memory management mechanisms for shared memory multiprocessors. The thesis [19] contains such results.

References

1. Boyer, R.S., ed.: Special issue on system verification. Journal of Automated Reasoning (JAR) **5** (1989)
2. Boyer, R.S., Moore, J S.: A Computational Logic Handbook. Academic Press (1988)
3. Hunt, W.A.: Microprocessor design verification. In *JAR* [1] 429–460
4. Moore, J S.: A mechanically verified language implementation. In *JAR* [1] 461–492
5. Young, W.D.: A mechanically verified code generator. In *JAR* [1] 493–518
6. Bevier, W.R.: Kit and the short stack. In *JAR* [1] 519–530
7. Moore, J S.: A grand challenge proposal for formal methods: A verified stack. In Aichernig, B.K., Maibaum, T.S.E., eds.: 10th Colloquium of UNU/IIST '02, Springer (2003) 161–172
8. The Verisoft Consortium: The Verisoft Project. `http://www.verisoft.de/` (2003)
9. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In Hurd, J., Melham, T., eds.: TPHOLs '05. LNCS, Springer (2005)
10. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In Aichernig, B., Beckert, B., eds.: SEFM '05, IEEE Computer Society (2005)
11. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD '05, IEEE Computer Society (2005) To appear.
12. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP processor. In Geist, D., Tronci, E., eds.: CHARME '03, Springer (2003) 51–65
13. Beyer, S.: Putting It All Together: Formal Verification of the VAMP. PhD thesis, Saarland University, Saarbrücken, Germany (2005)
14. Sawada, J., Hunt, W.A.: Processor verification with precise exeptions and speculative execution. In Hu, A.J., Vardi, M.Y., eds.: CAV '98, Springer (1998) 135–146
15. Müller, S.M., Paul, W.J.: Computer Architecture: Complexity and Correctness. Springer (2000)
16. Owre, S., Shankar, N., Rushby, J.M.: PVS: A prototype verification system. In Kapur, D., ed.: CADE '92, Springer (1992) 748–752
17. Aagaard, M., Ciubotariu, V., Higgins, J., Khalvati, F.: Combining equivalence verification and completion functions. In Hu, A., Martin, A., eds.: FMCAD '04, Springer (2004) 98–112
18. Paul, W., Dimova, D., Mancino, M.: Skript zur Vorlesung Systemarchitektur. `http://www-wjp.cs.uni-sb.de/publikationen/Skript.pdf` (2004)
19. Hillebrand, M.: Address Spaces and Virtual Memory: Specification, Implementation, and Correctnesss. PhD thesis, Saarland University, Saarbrücken, Germany (2005)
20. Kröning, D.: Formal Verification of Pipelined Microprocessors. PhD thesis, Saarland University, Saarbrücken, Germany (2001)