On the verification of a ”baby” hypervisor for a RISC machine; draft 0

Eyad Alkassar and Wolfgang Paul

January 2008

Abstract

We review portions of the DLX instruction set. We specify a very rudimentary hypervisor for DLX processors, outline its implementation and state what we expect to be the main invariants of the correctness proof.

1 Overview

Hypervisors are generalized operating system kernels. Both conventional kernels and hypervisors provide i) emulation of guest processes running on separate hardware machines and ii) some form of inter process communication between these guest processes. Where the difference between both is easy to state: guest processes of kernels can use their machine only in user mode and are usually called *users*. Guest processes of hypervisors can use their machine both in system mode or in user mode and are usually called *partitions*. The intended guest of a hypervisor is a conventional kernel or operating system (OS) together with its own users, even so this is not a formal requirement.

In this article we specify an extremely simple hypervisor for a subset of the well known RISC instruction set architecture (ISA) called DLX [HP96]. Then we outline an implementation and a correctness proof. In section 2 we specify the DLX ISA with details taken from [MP00] and [BJK+03]. We state the definitions of address translation in a very slightly more general form than in previous work; this will facilitate later to talk about several translations simultaneously. In the spirit of [GHLP05] we specify in section 3 the behavior a very rudimentary[1] hypervisor and its guests by a parallel model of computation which we call ’communicating guest partitions’ (CGP). In section 4 we specify intended invariants for the simulation of CGP on a single so called *host* processor. The main work is the generalization of the B-relation from [GHLP05]. In section 5 we specify i) the simulation of step a guest partition in user mode and ii) the simulation of step a guest partition in user mode in system mode such that the invariants are - hopefully; this is paper and pencil work - maintained. Compared with conventional kernel we have to argue even in this very this very simple scenario about a new concept called *shadow page tables*. We also have to make use of write protection

[1] No inter process communication

bits even if the guest partitions does not make use of them. The induction steps for each of the occurring cases are not very complex.

It is planned to apply in a C# environment some mechanical invariant checking to the constructions of this paper. Therefore we have added some suggestions how to implement the construction and how to formalize the invariants in such an environment.

The mathematical definitions made here are meant as first attempts. They are - hopefully - not wrong, but in many places there are obvious alternative definitions which might be preferable for good reasons the authors don't see yet. Also the *names* we have used in the mathematical definitions are supposed to closely match the names used among OS experts. Suggestions from OS experts to rename mathematical concepts are explicitly solicited.

2 A subset of the DLX instruction set

Compared with [MP00] we will omit in the following specification the following instructions: i) loads and stores of half words and bytes ii) unsigned arithmetic, iii) shift instructions.

2.1 Notation

For bit strings $a = a[n-1:0] \in \{0,1\}^n$ we denote by

$$\langle a \rangle = \sum_{i=0}^{n} a_i \cdot 2^i$$

the natural number with binary representation a and we denote by

$$[a] = -2^{a_{n-1}} + \langle a[n-2:0] \rangle$$

the integer with two's complement representation a. An easy calculation shows

$$[a] = \langle a \rangle \bmod 2^n$$

For numbers $x \in \{0, \ldots, 2^n - 1\}$ the binary representation of x of length n is the bit string $bin_n(x) \in \{0,1\}^n$ satisfying:

$$\langle bin_n(x) \rangle = x$$

The n bit binary addition function $+_n : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ is defined by:

$$a +_n b = bin_n(\langle a \rangle + \langle b \rangle \bmod 2^n)$$

In what follows we will apply concepts defined for natural numbers to bits strings of some fixed length n, e.g. we will talk about the addresses $a, a+1, \ldots, a+x-1$ for some x. Such sloppy definitions can be made precise by replacing $+$ by $+_n$ and referring to the binary representations of 1, x etc of length n.

For bits x and natural numbers n we define x^n as the string obtained by concatenating x exactly n times with itself

$$x^n = x \circ \ldots \circ x$$

address	name	meaning
0	SR	status register
1	ESR	exception status register
2	ECA	exception cause register
3	EPC	returns the status of a process
4	EDPC	execpetion delayed PC
5	Edata	exception data register
6	MODE	either in system or user mode
7	PTO	page table origin
8	PTL	page table length

Table 1: Register names and meanings of the SPR

2.2 Configurations and Auxiliary Concepts

We outline how the DLX instruction set architecture (ISA) is formally specified. Processor configurations d have the following components:

1. $d.R \in \{0,1\}^{32}$ stores the current value of register R. The relevant registers are: the program counter PC, the delayed PC[2] DPC, the general purpose registers $GPR(x)$ with $x \in \{0,1\}^5$ and the special purpose register $spr(x)$ with $x \in \{0,1\}^5$. For the special purpose registers we use the synonyms from Table 1, i.e. we will write $d.x$ instead of $d.spr(x)$.
2. The byte addressable memory $d.m : A \to \{0,1\}^8$ where the set of addresses $A \subset \{0,1\}^{32}$ usually has the form $A = \{a \mid \langle a \rangle \leq d.b\}$ for some maximal available memory byte address $d.b$. The content of the memory at byte address a is given by $d.m(a)$.

The maximal available address $d.b$ does not change during an ISA computation. Therefore it is rather treated as a parameter of the model than as a component of a configuration. We will later partition memory into pages of $4K$ bytes. We assume that $d.b$ is a multiple of some page size:

$$d.b = d.ptl \cdot 4K$$

where $d.ptl$ is a mnemonic for page table length measured in word which is introduced later.[3] For addresses a, memories m, and natural numbers x we denote by $m_x(a)$ the concatenation of the memory bytes from address a to address $a+x-1$ in little endian order:

$$m_x(a) = m(a+x-1) \circ \ldots \circ m(a)$$

The instruction executed in configuration d, denoted by $I(d)$, is the memory word addressed by the delayed PC:

$$I(d) = d.m_4(d.dpc)$$

[2]The delayed PC is used to specify the delayed branch mechanism detailed in [MP00].

[3]Attention: in previous papers and theses related to the verified VAMP procesor ptl is used as page table length + 1. This is IBM notation; when counting starts at 0, then this is the index of the last entry of the page table.

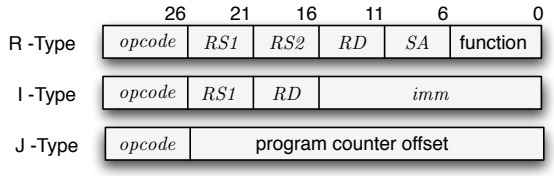

Figure 1: Instruction Types

The six high order bits of the instruction word constitute the opcode (opc):

$$opc(d) = I(d)[31:26]$$

Instruction decoding can easily be formalized by predicates on $I(d)$. In some cases it suffices to inspect the opcode only. The current instruction is for instance a 'load word' lw instruction if the opcode equals 100011:

$$lw(d) \Leftrightarrow opc(d) = 100011$$

Guided by table A one can specify the decoding of all instructions.

DLX instructions come in three instruction types as shown in Figure 1. The type of an instruction defines how the bits of the instruction outside the opcode are interpreted. The occurrence of an R-type instruction, e.g. an add or a subtract instruction, is for instance specified by:

$$rtype(d) \Leftrightarrow opc(d) = 000000$$

The remaining two types are defined through:

$$jtype(d) \Leftrightarrow opc(d) \in \{000010, 000011, 111110, 111111\}$$
$$itype(d) \Leftrightarrow opc(d) = \neg(rtype(d) \vee jtype(d))$$

Depending on the instruction type, certain fields have different positions within the instruction. For the register 'destination' operand (RD) we have

$$RD(d) = \begin{cases} I(d)[20:16] & itype(d) \\ I(d)[15:11] & \text{otherwise} \end{cases}$$

Note that this definition specifies the RD field even for J-type instructions. No harm is done, since in these situations the definition is never used. For the register 'source' operands ($RS1$ and $RS2$) we have:

$$RS1(d) = I(d)[25:21]$$
$$RS2(d) = I(d)[20:16]$$

$$SA(d) = I(d)[10:6]$$

A tricky case is the immediate constant because its length varies with the instruction type:

$$imm(d) = \begin{cases} I(d)[15:0] & itype(d) \\ I(d)[25:0] & \text{otherwise} \end{cases}$$

Fortunately the hardware, whose data paths have fixed width of 32 bits always uses the sign extended immediate constant

$$sxt(imm(d)) = imm(d)[15]^{16} \circ imm(d)$$

This turns the immediate constant into a 32-bit constant while preserving the value as a two's complement number. It is like adding leading zeros to a natural number. Hence it holds:

$$[sxt(imm(d))] = [imm(d)]$$

The effective address (ea) of load / store operations is computed as the sum of i)the content of the register addressed by the $RS1$ field $d.gpr(RS1(d))$ and ii) the immediate field $imm(d) = I(d)[15:0]$. The addition is performed modulo 2^{32} with two's complement arithmetic:

$$ea(d) = d.gpr(RS1(d)) +_{32} sxt(imm(d))$$

This definition is possible since n bit two's complement numbers and n bit binary numbers have the same value modulo 2^n. For details see e.g. Chapter 2 of [MP00].

2.3 Basic Instruction Set

With the above definitions in place we specify the next configuration d', i.e. the configuration after execution of $I(d)$. This obviously formalizes the instruction set.

Completing this definition for all instructions results in the the definition of a DLX next state function:

$$d' = \delta_D(d)$$

In the definition of d' we split cases depending on the instruction to be executed.

For all cases, except control instructions, the PC is incremented by four in 32-bit binary arithmetic and the old PC is copied into the delayed PC:

$$\begin{array}{rcl} d'.pc & = & d.pc +_{32} 4_{32} \\ d'.dpc & = & d.pc \end{array}$$

Components that are not changed have to be specified, too:

$$\begin{array}{rcll} d'.m & = & d.m & \\ d'.gpr(x) & = & d.gpr(x) & \text{for } x \neq RD(d) \\ d'.sr & = & d.sr & \end{array}$$

Writing/Reading Special Purpose Register

For our purposes we are particularly interested in the effect of the instructions movi2s resp. movs2i which move data from the general purpose register file to the special purpose register file resp. from the special purpose register file to the general purpose register file. The essential effect of $movi2s(d)$ is

$$d'.SPR(SA(d)) = d.GPR(RD(d))$$

The essential effect of $movs2i(d)$ is

$$d'.GPR(RD(d)) = d.SPR(SA(d))$$

TODO: Add both instructions to the instruction set table

Loading and storing memory locations

For each instruction case, we introduce a predicate. In case of a load word instruction we have:

$$lw(d) \Leftrightarrow opc(d) = 100011$$

Its essential effect is

$$d'.GPR(RD(d)) = d.m_4(ea(d))$$

Accordingly, for a store word instruction we get:

$$sw(d) \Leftrightarrow opc(d) = 101011$$

$$d'.m_4(x) = \begin{cases} d.m_4(ea(d^t)) & : x = ea(d) \\ d.m_4(x) & : otherwise \end{cases}$$

Arithmetic and logical operations

Arithmetic and logical operations are specified through the function $aluop$. Given a left and right operator (lop and rop) and the identifier of the operation to perform (op), it returns the computed results according to table A. The transition function computes the operands and the operation to perform and writes the result back into the destination register. The left operand is always the content of the first source Register $RS1(d)$:

$$lop = d.GPR(RS1(d))$$

Depending on the instruction type the right operand and the operation are computed differently. In case the instruction is of R-type, the right operand is taken from the second source register, whereas the right operand in an I-type instruction is the immediate constant. The following two predicates distinguish between the above mentioned two cases:

$$alucom_{imm}(d) = 1 \Leftrightarrow I(d)[30:29] = 01$$

$$alucom_{reg}(d) = 1 \Leftrightarrow rtype(d) \wedge I(d)[5:4] = 00$$

Hence, the right operator is given through:

$$rop = \begin{cases} sxt(imm(d)) & : alucom_{imm} \\ d.GPR(RS2(d)) & : otherwise \end{cases}$$

For the identifier of the ALU operation to perform, we have:

$$op = \begin{cases} I(c)[29:26] & : alucom_{imm} \\ I(c)[3:0] & : otherwise \end{cases}$$

Now we can define the transition function, in case of an ALU operation, i.e. when $alucom_{imm}(d) \vee alucom_{reg}(d)$ as:

$$d'.GPR(x) = \begin{cases} aluop(lop, rop, op) & : x = RD(d) \wedge x \neq 0^5 \\ d.GPR(x) & : otherwise \end{cases}$$

Jumping and branching

The instruction set defines five jump and two branch instructions. Jumps are either relative to the current position (j, jal), or absolute to the position specified in the source Register (jr, $jalr$). In addition the position before the jump incremented by four can be linked, i.e. saved in a register (jal, $jalr$). The definitions of the predicates $j, jal, jr, jalr$ can be easily deduced from table A.

$$jump(d) = j(d) \vee jal(d) \vee jr(d) \vee jalr(d)$$

Branch instructions are indicates through the predicate

$$branch(d) = 1 \Leftrightarrow I(d)[31:27] = 11010$$

Where the two branch instructions are indicated through

$$beqz(d) = 1 \Leftrightarrow branch(d) \wedge I(d)[26] = 0$$

$$bneqz(d) = 1 \Leftrightarrow branch(d) \wedge I(d)[26] = 1$$

The instructions $beqz, bneqz$ lead to jumps if the content of the first source register is zero, one respectively

$$btaken(d) = \begin{array}{l} (d.GPR(RS1(d))) = 0 \ \wedge \ beqz(d)) \vee \\ (d.GPR(RS1(d))) = 1 \ \wedge \ bneqz(d)) \end{array}$$

Hence, the instruction lead

$$bjtaken(d) = 1 \Leftrightarrow btaken(d) \wedge jump(d)$$

The manipulation of the PC by jump and branch instructions is now easily to define

$$d'.PC = \begin{cases} d.PC +_{32} sxt(imm(d)) & : btaken(d) \wedge j(d) \wedge jal(d) \\ d.GPR(SR(d)) & : jr(d) \wedge jalr(d) \\ d.PC +_{32} 4_{32} & : sonst \end{cases}$$

In case we have a jump and link instruction, we have to save the PC in the register $GPR(1^5)$

$$d'.GPR(x) = \begin{cases} d.PC +_{32} 4_{32} & : (jal(d) \vee jalr(d)) \wedge x = 1^5 \\ d.GPR(x) & : otherwise \end{cases}$$

Summary

Summing up, we can define the whole transition function (in system mode and except the semantics of instruction rfe, described in the interrupt section 2.4) of the ISA through the following equations.

Manipulation of the program counter

$$d'.PC = \begin{cases} d.PC +_{32} sxt(imm(d)) & : btaken(d) \wedge j(d) \wedge jal(d) \\ d.GPR(SR(d)) & : jr(d) \wedge jalr(d) \\ d.PC +_{32} 4_{32} & : sonst \end{cases}$$

Changes to memory

$$d'.m_4(x) = \begin{cases} d.m_4(ea(d^t)) & : sw(d) \wedge x = ea(d) \\ d.m_4(x) & : otherwise \end{cases}$$

Changes to the GPR

$$d'.GPR(x) = \begin{cases} 0^{32} & : x = 0^5 \\ d.SPR(SA(d)) & : movs2i(d) \wedge x = RD(d) \\ d.m_4(ea(d)) & : lw(d) \wedge x = RD(d) \wedge x \neq 0^5 \\ d.PC +_{32} 4_{32} & : (jal(d) \vee jalr(d)) \wedge x = 1^5 \\ aluop(lop, rop, op) & : x = RD(d) \wedge x \neq 0^5 \\ d.GPR(x) & : otherwise \end{cases}$$

Changes to the SPR

$$d'.SPR(SA(d)) = d.GPR(RS1(d))$$

2.4 Dealing with Interrupts

Types of interrupts

Interrupts are triggered by interrupt event signals which might be internally generated (like illegal instruction, misalignment, and overflow) or externally generated (like reset and timer interrupt). Interrupts are numbered using indices $j \in \{0, \ldots, 31\}$.

We classify the set of these indices in three ways according to table 2

1. maskable / not maskable. The set of indices of maskable interrupts is denoted by M
2. repeat / continue. This defines where a computation is resumed after the interrupt handler has completed. For a continue interrupt the computation is resumed after the interrupted instruction. For a repeat interrupt the interrupted instruction is repeated. The only repeat interrupts are page fault on fetch (pff) and page fault on load store (pfls)
3. external / internal. Dealing with external interrupts is remarkably subtle. Because they are not used here, we simply omit them.

We denote the vector of internal event signals computed in configuration d by $iev(d)$.

Index j	symbol	meaning	external	maskable	resume	set
0	reset	reset	yes	no	abort	SP
1	ill	illegal instruction	no	no	abort	SP
2	mal	misaligned address	no	no	abort	SP
3	pff	pagefault on fetch	no	no	repeat	SP
4	pfls	pagefault on load store	no	no	repeat	SP
5	trap	trap instruction	no	no	continue	Trap
6	ovf	integer overflow	no	yes	continue/abort	IS
7	xovf	IEEE overflow	no	yes	continue/abort	IS
8	unf	IEEE underflow	no	yes	continue	IS
9	inx	IEEE inexact	no	yes	continue	IS
10	dbz	IEEE division by zero	no	yes	continue	IS
11	inv	IEEE invalid operation	no	yes	continue	IS
12	unimpl	IEEE unimplemented	no	yes	continue	IS
13	timer	timer	yes	yes	continue	SP
13+i	ex_i	external devices	yes	yes	continue	IO

Table 2: Interrupts of the DLX

The masked cause vector $mca(d)$ is computed from the internal event signals $iev(d)[j]$ with the help of the interrupt mask stored in the status register: if interrupt j is maskable and $sr[j] = 0$, then j is masked out:

$$mca(d)[j] = \begin{cases} iev[j] \wedge d.sr[j] & j \in M \\ iev[j] & \text{otherwise} \end{cases}$$

If any one of the masked cause bits is on, the JISR (jump to interrupt service routine) bit is turned on

$$JISR(d) = \bigvee_j mca(d)$$

Because many interrupt lines can become active simultaneously it is important to know the smallest index of an active bit of mca. This index is called the *interrupt level* and specifies the interrupts of highest priority which is also the one which will be serviced immediately.

$$il(d) = \min\{j : mca(d)[j] = 1\}$$

This allows to define, whether the interrupt is of type repeat:

$$repeat(d) = (il(d) \in \{3, 4\})$$

Semantics of interrupts

If an interrupt occurs, many things happen:

- The PCs are forced to point to the start addresses of the interrupt service routine. We assume it starts at (binary) address 0:

$$\begin{aligned} d'.dpc &= 0_{32} \\ d'.pc &= 4_{32} \end{aligned}$$

- All maskable interrupts are masked and the masked cause register is saved into a new exception cause register.

$$\begin{array}{rcl} d'.sr & = & 0^{32} \\ d'.esr & = & d.sr \\ d'.eca & = & mca(d) \end{array}$$

- Depending on the interrupt type PCs are saved, in the exception PC register.

$$(d'.edpc, d'.epc) = \begin{cases} (d.DPC, d.PC) & : repeat(d) \\ (d.DPC, d.PC)^u & : continue(d) \end{cases}$$

  Where $(d.DPC, d.PC)^u$ simulates the computation of the next PCs as if no interrupt was raised.
- Auxiliary data for the intended interrupt handler is stored in an exception data register $edata$. We only specify the new content for the case of trap instructions. In the DLX instruction set the trap instruction has J-type format with opcode 111110 and an j. We give the trap instruction interrupt event line 5:

$$iev(d)[5] \leftrightarrow trap(d)$$

  If this event line is active and no line with higher priority is active, then a trap interrupt occurs *and* is serviced

$$traps(d) \leftrightarrow il(d) = 5$$

  In case of a trap interrupt, the sign extended (26 bit) immediate constant is saved in the exception data register

$$traps(d) \rightarrow d'.edata = sxtimm(d)$$

Return from exception

The instruction *return from exception (*rfe*)* is invoked to restore the PCs and the status register after a return. It is indicated through the predicate:

$$rfe(d) = itype(d) \wedge I[31:26] = 111111$$

In short, the effect of rfe is:

$$\begin{array}{cc} (dpc, pc) & = (edpc, epc) \\ sr & = esr \end{array}$$

2.5 Adding address translation

All definitions so far apply only if the machine works in system mode. The mode in configuration d is defined by the lowest bit of the mode register: $sysmode(d) = d.mode[0]$ and $usermode(d) = \not d.sysmode[0]$. In order to define the ISA in user mode we consider address translation with the help of page tables. Here we only describe an one-level translation, which consumes memory in a wasteful way.

We split addresses $a[31:0]$ into *page index* and *byte index* by

$$\begin{array}{rcl} a.px & = & a[31:12] \\ a.bx & = & a[11:0] \end{array}$$

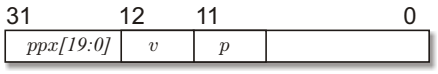

Figure 2: Page Table Entry

A *page index* is any bit string of length 22. We define a page table pt as a pair $pt = (ptl, pte)$ where $pt.ptl \in \mathbb{N}$ is the page table length and pte is a mapping from page indices to words, which are called page table entries:

$$pte : \{0,1\}^{22} \to \{0,1\}^{32}$$

For page table entries $w = pte(y)$ we define the following synonyms according to figure 2: i) the page index $w.px = w[31:12]$ ii) the valid bit $w.v = w[11]$ and iii) the write protection bit $w.p = w[10]$.

With the help of a page table $pt = (ptl, pte)$ we can translate an address a with $\langle a.px \rangle < ptl$ and get translated page index, valid bit and protection bits (see figure 3):

$$\begin{aligned} px(pt,a) &= pte(a.px).px \\ v(pt,a) &= pte(a.px).v \\ p(pt,a) &= pte(a.px).p \end{aligned}$$

In user mode we use the page table $pt(d) = (pt(d).ptl, pt(d).pte)$ with $ptl = \langle d.ptl \rangle$ and page table entries

$$pt(d).pte(y) = d.m_4(d.pto \circ 0^{12} +_{22} y)$$

The names we are using reflect the view of a hardware designer. We stress, however, that the names are apparently *not* consistent with the notations of operating system specialists, among whom any page containing a page table entry seems to be called a page table. So there is clearly a need to harmonize names and notation.

Now consider in configuration d a memory access with address a which might be writing $(w = 1)$ or reading $(w = 0)$. Such an access can be an instruction fetch; then $a = d.dpc$ and the access is reading. Otherwise the access is a reading access if $lw(d)$ or a writing access if $sw(d)$ holds.

With the help of the definitions above, we can specify when such an access will cause an illegal memory access exception. This depends on three things: whether the page index of the address lies in the range of the page table, the valid bit is true and, in case the access is writing, on the write protection bit for address a:

$$\begin{aligned} illm(d,a,w) = {} & (\langle a.px \rangle \geq pt(d).ptl \vee \\ & (\sim v(pt(d),a) \vee v(pt(d),a) \wedge w \wedge \sim p(pt(d),a))) \end{aligned}$$

In case no such exception occurs we can define the translated physical memory address for this access as

$$pma(d,a) = px(pt(d),a) \circ a.bx$$

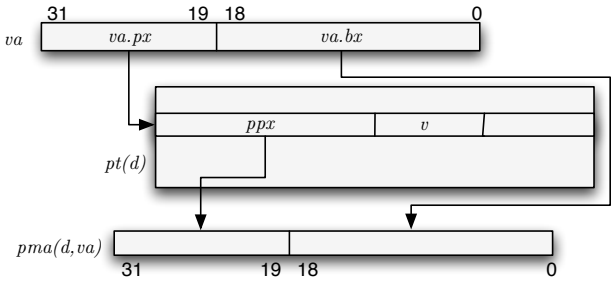

Figure 3: Address Translation

The crucial changes in the definition of the next configuration in user mode concern - in the absence of exceptions - the fetched instruction

$$I(d) = d.m_4(pma(d, d.dpc))$$

and the replacement of the effective address $ea(d)$ by $pma(d, ea(d))$ in the semantics of the load and store instruction.

3 Hypervisor semantics

Analogous to the CVM model from [GHLP05] we present a parallel model CGP for an abstract hypervisor and its guest partitions. We will make use of two transition functions: i) δ_D for DLX machines from the last section and ii) the transition function δ_C of a C machine. Thus the definition of CGM is based on *small step semantics*. In small step semantics, C configuartions are pairs $c = (c.pr, c.s)$ where $c.pr$ is the program rest and $c.s$ is the 'state'. Given some configuration c in which some expression e occurs then we denote by $va(c, e)$ the value of e in configuration c.

Parameters of this model are the number ng of guests and for each index $i \in [1 : ng]$ of a guest a fixed guest page table length $gptl(i)$. We will later put restrictions on the length of the page tables.

A configuration cgp then has the following components:

- $cgm.c$ the configuration of an abstract C machine representing an abstract hypervisor.
- $cgm.g(i)$ a DLX configuration representing guest partition i for $i \in [1 : ng]$.
- $cgm.cg \in [0 : ng - 1]$ the currently running process; $cgm.cg = 0$ means that the hypervisor is running

This C machine implements special services for the guest operating system. Hence it must be able to invoke special calls, in the following called primitives, manipulating more than its local state. In this paper we will only consider one primitive, namely *startnext* for starting a guest partition.

In order to get quickly to the point where hypervisors differ from kernels we make some severe simplifying assumptions. i) The only interrupts produced by guest partitions are traps. ii) the abstract hypervisor only consists of a trivial scheduler. A single C variable CG is maintained. It keeps track of the currently running

guest. If the hypervisor is called, it increases CG by 1 modulo ng and then - in the new configuration c starts guest $va(c, CG)$. Thus we do not have to worry about passing parameters of hypervisor calls, dispatching the right handler etc. These problems can be solved for hypervisors and kernels in a similar way. Also with a trivial abstract hypervisor no theory of linking [diss tom to appear] is needed to produce a concrete hypervisor from the abstract hypervisor. In what follows we simply describe the concrete hypervisor directly.

With these restrictions the CGP semantics, i.e. the hypervisor specification, reduces to the following rules:

- Let $i = cgm.cg > 0$ be the index of the currently running guest. If the next instruction to execute of this guest is not trap instruction in system mode ($\neg(trap(cgm.g(i)) \wedge systemmode(cgm.g(i))$), then guest i performs a local DLX step:
$$cgm'.cg(i) = \delta_D(cgm.cg(i))$$
The configurations of the hypervisor and the other guests stay unchanged: $cgm'.c = cgm.c$ and $cgm'.g(j) = cgm.g(j)$ for $j \neq i$. Even so the latter two conditions look trivial on paper, they specify far reaching guarantees of integrity.
- The hypervisor is running ($cgm.cg = 0$) and the (external) startnext function is not executed; formally the program rest $cgm.c.pr$ of the C configuration does not start with $startnext$. Then the hypervisor executes a C instruction
$$cgm'.c = \delta_C(cgm.c)$$
and the guests are not affected $cgm'.g(i) = cgm.g(i)$ for all i.
- The hypervisor is running and the program rest starts with startnext. In this case the current value $va(cgm.c, CG)$ of variable CG determines the next running guest:
$$cgm'.cg = va(cgm, CG)$$
The guests stay unchanged: $cgm'.g(i) = cgm.g(i)$ for all i. For simplicity we leave the c configuration unchanged. $cgm'.c = cgm.c$. This freezes the state $cgm.c.s$ and the program rest $cgm.c.pr$; we take care of the program rest in the next - and last - rule.
- Let $i = cgm.cg > 0$ be the index of a currently running guest whose next instruction is a trap in system mode ($trap(cgm.g(i)) \wedge systemmode(cgm.g(i))$). Then control is transferred to the hypervisor
$$cgm'.cg = 0$$
Let p be the body of the main function of the hypervisor. The hypervisor is started in the old state and with p as program rest
$$cgm'.c = (p, cgm.c)$$
The guests other than guest i stay unchanged: $cgm'.g(j) = cgm.g(j)$ for $j \neq i$. The calling guest completes its instruction
$$cgm'.g(i) = \delta_D(cgm.g(i))$$
Note that as a consequence of this last definition guest i starts with its own trap handler when it is started a second time.

4 Data Structures of the Concrete Hypervisor

The implementation model of the CGM is a DLX machine as specified in **??**wever we want to implement it in, and argue about a C program rather than about Assembler. Hence we introduce a so called concrete hypervisor, which implement the interrupt handlers, etc and with that the CGM. Linking this concrete hypervisor with the abstract one, results in the complete implementation of the hypervisor.

We denote configurations of the concrete hypervisor by cc. In addition to the variables of the abstract hypervisor (here only CG) the concrete hypervisor uses three data structures :

The process control blocks . This is an array *PCB*$[0 : ns]$ of process control blocks. Each process control block has components *PCB*$[i].R$ for each DLX register R. Obviously we store register contents of guest $i > 0$ in *PCB*$[i]$ (and register contents of the hypervisor in *PCB*[0]).

The host page table space This is a large array $HPTS[0 : L - 1]$ of page table entries together with an array $HB[1 : ng]$. This array stores the start index of the page table entries of guest i in $HPTS$. Recall that the parameter $gtpl(i)$ returns the number of pages used by guest i. We define $HB[1] = 0$ and $HB[i + 1] = HB[i] + gtpl(i)$. The size of the array is defined by $L = HB[ng] + gptl(ng)$.

Thus *HPTS*$[HB[i] : HB[i + 1] - 1]$ has as many page table entries as pages been used by guest i pages. We call this portion the host page table for process i and denote it by *HPT*(i). Exactly like in an ordinary kernel it defines how the host maps virtual addresses to physical ones of guest i. Moreover in case the guest runs in - untranslated - system mode, it is used by the hardware as the page table.

The shadow page table space . This is an array *SPTS*$[0 : L]$ 'shadowing' the host page table space. Portion $SPT(i) =$ *SPTS*$[HB[i] : HB[i+1]-1]$ is used by the hardware, when the guest runs in translated mode. In a nutshell the translation defined by the shadow page table is the composition of the translations defined by page table currently used by the guest and then of the translation defined by the host page table *HPT*(i).

5 Simulation Relation

We want to relate a CGM configuration cgm with a DLX configuration h of the so called host hardware and a configuartion cc of the concrete kernel. The simulation relation will be parameterized by the current allocation function $alloc$ of the compiler. For C variables and subvariables x and C configurations c, $alloc(c, x) \in \{0, 1\}^{32}$ indicates the base address of x in the memory of the DLX machine. Thus we want to define a relation $cgmsim(alloc)(cgm, cc, h)$.

5.1 Memory Map

First, we formulate conditions concerning the partitioning of the host memory into guest memory. Let $H \in \{0, 1\}^{32}$ be a page index, which is big enough, such that the compiled code of the concrete hypervisor and its stack and heap occupy only addresses with page indices below H. Then we require

1. no page j of a guest partition i is mapped to the memory regions occupied by the hypervisor:
$$(HPT(i)[j].px \geq H$$
for all i and j.
2. different guest partitions are mapped to disjoint memory regions
$$HPT(i)[j].px \neq HPT(i')[j'].px$$
for all $i \neq i'$ and for all j, j'.

5.2 Tracking the current guest

The current value of C variable CG of the concrete kernel should track the currently running guest partition of cgm:

$$cgm.cg > 0 \rightarrow va(cc, CG) = cgm.cg$$

5.3 Variables of the abstract kernel

The variables of the abstract kernel must be coded in the configuration cc of the concrete kernel. Here this simply reduces to

$$va(cc.CG) = va(cgm.c, CG)$$

5.4 Compiler Correctness

We need, that the current C configuration cc of the hypervisor can be abstracted from the host configuration h. This is done by the compiler simulation relation

$$consis(alloc)(cc, h)$$

Its details are explained in [SEFM05] and [MOD]. Two required and crucial conditions for $consis(alloc)(c, x)$ are:

1. e-consistency: if variable or subvariable x has an elementary type (not composite and not a pointer type) and b bytes are required to store data of this type, then
$$va(c, x) = h.m_b(alloc(x))$$
This is basically the definition of the allocation function.
2. p-consistency: if p is a reachable pointer variable pointing to variable y (formally $va(c, p) = y$), then
$$alloc(y) = h.m_4(alloc(p))$$
This defines a sub graph isomorphism between the heaps of c and h.

Using this we require for non optimizing compilers [4] which code part of the C configuration only in the memory and not in the registers of the target machine

$$consis(alloc)(cc, h)$$

[4] as used in the Verisoft project so far

If the compiler is storing parts of the C configuration in the processor registers then we would have to distinguish between running and non running hypervisor. For a non running hypervisor we have to consider the hardware configuration h' with memory from the host ($h'.m = h.m$) and registers from the process control blocks ($h'.R = va(cc.\mathit{PCB}[0].R)$. Then some extra arguments about the use of $\mathit{PCB}[0]$ by the hypervisor seem to be necessary to ensure that this definitions is not cyclic.

5.5 Register contents of guests

For guests which are not running, contents of registers R are stored in their process control block: $cgm.g(i) \neq cgm.cg \rightarrow$

$$\forall R : cgm.g(i).R = va(cc, \mathit{PCB}[i].R)$$

Now assume guest i is running $cgm.cg = i > 0$. Then registers in the set

$$E = \{pto, mode, epc, edpc, emode\}$$

get a treatment different from registers not in E. Guest registers not in E ar stored in the hardware registers of the host. Registers in E are stored in the process control block of guest u

$$cgm.g(i).R = \begin{cases} h.R & : R \notin E \\ va(cc, \mathit{PCB}[i].R) & : R \in E \end{cases}$$

5.6 Memory contents of guests

As in an ordinary kernel they are determined by the translation defined by the current value $va(cc, \mathit{HPT}(i))$ of the host page table for process i.

$$cgm.g(i).m(a) = h.m(pma(va(cc, \mathit{HPT}(i)), a)$$

5.7 Mode and page table origin of hypervisor for running guests

If a guest is running ($cgm.cp = i > 0$) then the host runs in user mode, even if the guest runs in system mode.

$$h.mode = 1$$

This requires the hypervisor to give a special treatment of privileged operations performed by the guest in system mode. Even if they do not write the page table origin or the page table length they create an illegal interrupt and must then be simulated by the hypervisor. What happens in the simulation is prescribed by the simulation relation; it has to be maintained after the simulation of each step of the CGM machine. An outline of the simulation for some simple cases is given in the next section.

However a different translation is chosen for system and user mode of the guest. If the guest is in system mode $cgm.g(i).mode = 1$, the the host page table for guest i is used for the translation. Thus the hardware page table origin

$h.pto$ is set to something like the allocated base address of subarray $\mathit{HPT}(i)$. If one has semantics with $\&$-operator we can use

$$h.pto = va(cc, \& - \mathit{HPTS}[\mathit{HB}[i]])$$

Otherwise we use the alloc function for the big array

$$h.pto = alloc(cc, \mathit{HPTS}) + 4 \cdot va(cc, HB[i])$$

If the guest runs in user mode $(cgm.g(i).mode = 1)$, then we use the shadow page tables

$$h.pto = va(cc, \mathit{SPTS}[HB[i]])$$

resp.

$$h.pto = alloc(cc, \mathit{SPTS}) + 4 \cdot va(cc, HB[i])$$

This only works if the shadow page tables have the intended content, at least for the currently running guest as indicated in the next subsection.,

5.8 Shadow page tables

For guests running in user mode $(cgm.cp = i > 0 \wedge cgm.g(i).mode = 1)$ the translation defined by the shadow page table $SPT(i)$ must be the composition of the translations defined by the guests page table $pt(cgm.g(i)$ and the host page table $\mathit{HPT}(i)$ for guest i. Formally for all page indices $b < cgm.g(i).ptl$ we *almost* require:

$$va(cc, SPT(i)(\langle b \rangle)) = va(cc, \mathit{HPT}(\langle px(cgm.g(i), b) \rangle))$$

At first sight condition (*) can be guaranteed in a simple way: whenever the guest enters user mode compute the concerned portions of the shadow page table from scratch. If the guest protects his page table against writes in user mode, then we never have to worry about the condition again while the guest is in user mode. However we cannot assume this; fortunately the write protect bits of the shadow page table can be used to solve the problem. Thus we require the above equation to hold only for the page indices and the valid bits of the shadow page tables

$$\begin{aligned} va(cc, \mathit{SPT}(i)[\langle b \rangle].px) &= va(cc, \mathit{HPT}(i)[\langle px(cgm.g(i), b) \rangle].px) \\ va(cc, \mathit{SPT}(i)[\langle b \rangle].v) &= va(cc, \mathit{HPT}(i)[\langle px(cgm.g(i), b) \rangle].v) \end{aligned}$$

However we turn on the write protect bit potentially in more situations than the user process of the guest. We define a predicate $gpti(q, i)$ to be true if page q of the guest intersects with the guest page table $cgm.g(i)$. Because page table origins are multiples of page sizes this can simply be formalized as

$$cgm.g(i).pto < q < cgm.g(i).pto + cgm.g(i).ptl$$

If $gpti(q, i)$ holds we say that q is a (guest) page table index of guest i. We now set the write protection bits in the shadow page tables as follows:

$$va(cc, SPT(i)[\langle b \rangle].p) = \begin{cases} 1 & : \\ gpti(px(cgm.g(i), b), i) & \\ va(cc, \mathit{HPT}(i)[\langle px(cgm.g(i), b) \rangle].p) & : \end{cases}$$

This creates an interrupt whenever the guest writes its own page tables in user mode and permits the hypervisor to update the shadow page tables such that the invariants are maintained.

5.9 Host page tables

We need a similar invariant to the one for shadow page tables for the host page tables, since the guest page table could also be written in system mode. We simply require that in guest system mode all host page table entries, which map to page indices of the current guest page table, are protected.

$$gpti(b, i) \wedge h.mode = 1 \wedge va(cc, cp) = i \Rightarrow$$
$$va(cc, \mathrm{HPT}(i)[\langle b \rangle].p) = 1$$

6 Implementation details

Simply spoken the concrete hypervisor implementation consists of interrupt handlers for pagefaults and illegal instructions. These interrupts are triggered by instructions executed in guest mode, and are delivered to the host. The latter either simulates them on the guest or performs some updates on its page tables.

The handlers are implemented in C0 enriched with inline assembler code. Assembler is needed to change the content of registers, and memory portions of the hardware machine not mapped to C0 variables. For simplicity we will describe the implementation only by some pseudo code, where assembler portions will be represented by their effects to some hardware machine h. For example writing $PCB[i].eca = h.eca$ would translate to the assembler program:

$$\begin{array}{lcccc} (1) & movs2i & 10 & ECA \\ (2) & sw & ba & 10 \end{array}$$

where ba denotes the allocation address of the variable $PCB[i].eca$

6.1 Auxiliary functions

First we define a set of helper functions used in the handlers. These are functions for simulating an interrupt in a guest, computing the address translation pma, saving a guest when entering the hypervisor, restoring a guest after leaving it and updating the shadow page table of a guest.

Simulating interrupts

The following function mimics the occurrence of an interrupt of level il in guest i according to the semantics given in Section 2.4. pc and dpc are set to the start address of the interrupt handler of guest i. Depending on the type of the interrupt, either the current pcs of the guest or the results of the next pc computation are stored in epc and $edpc$. The exception cause is set to il and the data special register is set to the reported one in the hardware.

```
void sim_int (i, il)
{
  PCB[i].dpc = 0;
  PCB[i].pc = 4;
  PCB[i].emode = PCB[i].mode;
  PCB[i].edpc =
  (if (repeat(il))
    PCB[i].dpc
  else
    nextDPC(PCB[i].GPR, PCB[i].PC, PCB[i].DPC));
  PCB[i].epc =
  (if (repeat(il))
    PCB[i].pc
   else
    nextPC(PCB[i].GPR, PCB[i].PC, PCB[i].DPC));
  PCB[i].eca = comp_eca(il);
  PCB[i].esr = PCB[i].sr;
```

```
  PCB[i].sr = 0;
  if (trap(il) PCB[i].edata) = h.edata
}
```

Saving a guest

Whenever the hardware changes mode from user (guest) to system (hypervisor) mode, we have to save the current state of the guest, *i.e.* the content of the GPR. Note, that the code given below is not accurate, since allocating variable k will change at least one register, which has to be stored to the PCB. Hence a correct implementation would first save the register to which k is written.

```
void save_guest (i)
{
  for (k=1;k<32;k++)
    PCB[i].GPR[k] = h.GPR[k];
}
```

Restoring a guest

The hypervisor switches back to a guest, by invoking a rfe instruction (see 2.4). But before executing rfe the hardware must be prepared. First the right pagetable is chosen, depending on the mode of the guest: in system mode it is the host page table, else the shadow page table. Then pto is set to the address of the chosen page table (we assume some address-of-operator &), ptl, $edpc$ and pc are taken from the process control block and $emode$ is set to user mode (since from hardware perspective guests always run in user mode). Finally the rfe instruction is executed, which returns control back to the guest.

```
void restore_guest (i)
{
  var pt = (if (PCB[i].mode = 0)
              HPTS[HB[i]]
            else
              SPTS[HB[i]]);

  h.pto = &(pt);
  h.ptl = PCB[i].ptl;
  for (k=1;k<32;k++) do
        h.GPR[k] = PCB[i].GPR[k];

  h.emode = 1;
  h.edpc = PCB[i].dpc;
  h.epc = PCB[i].pc;

  asm rfe;
}
```

Updating the Shadow Page Table

The following function constructs a shadow page table, as the concatenation of the guest page table (specified by pto and ptl of the guest) and the host page table. For all page table entries between a given offset off_s and length off_l the physical page index is computed using the host page table. In case the mapped page index is within the guest page table, the corresponding entriy in the shadow page table is marked as protected. This is needed to notify the hypervisor every time the guest (being in user mode) updates one of his page table entries.

```
void update_spt (i, off_s, off_l)
{
  for (k =off_s; k < off_s+off_l;k++) do
  {
    apte = pma (HPTS[HB[i]], PCB[i].pto*2^(12)+4*k);
    pte = h.mm_4 (apte);
    px = HPTS[HB[i]][pte.px]

    SPTS[i][k].px = px;
    SPTS[i][k].v = pte.v;
    if (PCB[i].pto < pte.px < PCB[i].pto+PCB[i].ptl)
      SPTS[i][k].p = 1;
    else
      SPTS[i][k].p = 0;
  }
}
```

6.2 Interrupt handlers

Reading SPR registers

The special purpose register is read by the $movs2i$ instruction (see **??**). If the guest executes $movs2i$, in system mode its effects are emulated on the process control block. In case the guest is in user mode, an illegal instruction interrupt is simulated.

```
void handle_movs2i (i)
{
    if (PCB[i].mode = 0)
    {
      instr = h.mm_4(pma(HPTS[HB[i]], PCB[i].dpc));
      sa = SA(instr);
      rd = RD(instr);
      PCB[i].GPR[rd] = PCB[i].SPR[sa];
    }
    else
      sim_int (i, ill);
    restore_guest (i)
}
```

Writing SPR registers

We again distinguish between the mode of the guest. The interesting cases when a guest writes the SPR in system mode are: changing the page table origin and changing the page table length. In both cases we have to *i)* update the shallow page table and *ii)* set the protection bits of the host page table pointing to the guest page table to one. The latter action is needed to notify the hypervisor each time the guest updates his page table in system mode. If pto is changed, we compute the shadow page table from scratch. If ptl is increased we compute the shadow page tables for the new page table entries. If ptl is decreased, we update the write protection bits for those pages for which predicate $gpti(q, i)$ does not hold any more.

TODO What happens if $ptl - pto > gptl(i)$?

```
void handle_movi2s (i)
{
  if PCB[i].mode = 0 then
  {
    instr = h.mm_4 (pma (HPTS[HB[i]], PCB[i].dpc));
    sa = SA(instr);
    rd = RD(instr);
    if (PCB[i].mode = 0)
    {
      if (sa = PTO)
      {
        new_pto = PCB[i].GPR[rd];
        old_pto = PCB[i].pto;
        PCB[i].ptl = new_pto;
        for (k=old_pto; k<PCB[i].old_pto+PCB[i].ptl;k++)
          HPTS[HB[i]][old_pto+k].p = 0;

        update_spt (i, 0, PCB[i].ptl);

        for (k=new_pto; k<new_pto+PCB[i].ptl;k++)
          HPTS[HB[i]][new_pto+k].p = 1;
      }
      else
      if (sa = PTL) {
        new_ptl = PCB[i].GPR[rd];
        old_ptl = PCB[i].ptl;
        PCB[i].ptl = new_ptl;
        if (new_ptl >= old_ptl)
        {
          update_spt (i,old_ptl, new_ptl-old_ptl));
          for (k=old_ptl; k<new_ptl;k++)
            HPTS[HB[i]][PCB[i].pto+k].p = 1;
        }
        else
          for (k=new_ptl; k<old_ptl;k++)
            HPTS[HB[i]][PCB[i].pto+k].p = 0;
      }
```

```
      else
       PCB[i].SPR[sa] = PCB[i].GPR[rd];
    }
    else
      sim_int (i, ill);
  restore_guest (i)
}
```

Trap instruction

If the guest is in user mode we just simulate the trap interrupt. However a trap instruction of the guest in system mode indicates a call to the abstract hypervisor. *E.g.* in our case this only could be the *startNext* hypercall defined in 3. Note that in case of a call to the abstract hypervisor, the guest is not restored (since a different one could be chosen). Simulate trap also if guest is in system mode?

```
void handle_trap (i)
{
  if (PCB[i].mode) = 0
    call_abstract_kernel()
  else
  {
    sim_int(i, trap)
    restore_guest(i)
  }
}
```

Rfe instruction

If the guest is in system mode, the semantics of rfe (see **??**) is mimicked on the process control block. Else an illegal instruction interrupt is simulated, since rfe is not permitted in user mode.

```
void handle_rfe (i)
{
  if (PCB[i].mode = 0)
  {
    PCB[i].dpc = PCB[i].edpc;
    PCB[i].pc = PCB[i].epc;
    PCB[i].sr = PCB[i].esr;
    PCB[i].mode = PCB[i].emode;
  }
  else
    sim_int(i, trap);
  restore_guest(i)
}
```

Handling a page fault

A page fault is either triggered by an instruction fetch, a load word or a store word instruction. First, the page index of the address causing the page fault is computed. In case this page index is not within the current page table, the interrupt is a page fault on fetch or the instruction is load word a page fault interrupt is simulated on the guest. Otherwise, *i.e.* the instruction is store word and we had a protection violation, we first compute the entry in the guest page table corresponding to the page index causing the interrupt. If this entry either sets the valid bit to false or the protected bit to true, again a page fault interrupt is delivered to the guest. Otherwise we have to update the *i)* guest page table and *ii)* either the shadow page table (guest in user mode) or the host page table (guest in system mode).

```
void handle_pf (i, pfh)
{
  pt = if (PCB[i].mode = 1)
         SPTS [HB[i]] else HPTS[HB[i]];

  instr = h.mm_4 (pma (pt, PCB[i].dpc));
  ea = if (pfh) ea(instr)
       else (PCB[i].dpc);
  px = ea.px;

  if ((px < PCB[i].ptl) || (pfh && lw(instr)))
  {
    ad_gpte = pma (pt, PCB[i].pto*2^{12}+4*px);
    gpte = h.mm_4 (ad_gpte);
    if (gpte.v && not(gpte.p) && p(pt,px))
    {
      h.mm_4(ad_gpte) = h.GPR(RS1(instr));
      update_spt (i, px, 1);
    }
    else
      sim_int(i, il(h.eca))
  } else
    sim_int(i, il(h.eca))
  restore_guest(i)
}
```

6.3 The Hypervisor dispatcher

Having all interrupt handlers described above, the hypervisor is simple to define. First we save the current state of the registers into the guest control block of the current guest cg. Then, depending on the interrupt level, the appropriate handler is chosen and executed. In case of an illegal instruction, the instruction type is determined.

```
void kernel_dispatcher ()
{
  save_guest (cg);
```

```
  il = il(h.eca);
  instr = h.mm_4 (pma (HPTS, PCB[i].dpc));

  case il
    "pff" : handle_pf (cg, false);
    "pfh" : handle_pf (cg, true);
    "ill" : if rfe(instr) handle_rfe (cg) else
            if movs2i(instr) handle_movs2i(cg) else
            if movi2s(instr) handle_movi2s(cg)
    "trap": handle_trap (cg);
}
```

A Instruction Set Architecture

R-Type Instructions

$I[31:26]$	$I[5:0]$	Mnemonic	Effect
Arithmetic / Logical Operation			
000000	100001	add	$RD = RS1 +_{32} RS2$ (no overflow)
000000	100011	sub	$RD = RS1 -_{32} RS2$ (no overflow)
000000	100100	and	$RD = RS1 \wedge RS2$
000000	100101	or	$RD = RS1 \vee RS2$
000000	100110	xor	$RD = RS1 \oplus RS2$
000000	100110	lhg	$RD = (RS2, 0^{16})$
Test Set Operation			
000000	101000	clr	$RD = 0^{32}$
000000	101001	sgr	$RD = ([RS1] > [RS2]?0^{31}1 : 0^{32})$
000000	101010	seq	$RD = ([RS1] = [RS2]?0^{31}1 : 0^{32})$
000000	101011	sge	$RD = ([RS1] \geq [RS2]?0^{31}1 : 0^{32})$
000000	101100	sls	$RD = ([RS1] < [RS2]?0^{31}1 : 0^{32})$
000000	101101	sne	$RD = ([RS1] \neq [RS2]?0^{31}1 : 0^{32})$
000000	101110	sle	$RD = ([RS1] \leq [RS2]?0^{31}1 : 0^{32})$
000000	101111	set	$RD = 0^{31}1$
Writing/ Reading SPR			
000000	010000	$moves2i$	$RD = SPR(SA)$
000000	010001	$movei2s$	$SPR(SA) = RD$

J-Type Instructions

$I[31:26]$	Mnemonic	Effect
Control Flow Operation		
000010	j	$PCP = \langle PCP \rangle +_{32} simm$
000011	jal	$R31 = \langle PCP \rangle +_{32} 4, \quad PCP = \langle PCP \rangle +_{32} simm$

I-Type Instructions

$I[31:26]$	Mnemonic	Effect
Load / Store, $mem[31:0] = m_4(ea)$		
100011	lw	$RD = mem[31:0]$
101011	sw	$mem[31:0] = RD$
Arithmetic / Logical Operation		
001001	$addi$	$\langle RD \rangle = RS1 +_{32} simm$ (no overflow)
001011	$subi$	$\langle RD \rangle = RS1 -_{32} simm$ (no overflow)
001100	$andi$	$RD = RS1 \wedge simm[31:0]$
001101	ori	$RD = RS1 \vee simm[31:0]$
001110	$xori$	$RD = RS1 \oplus simm[31:0]$
001110	$lhgi$	$RD = (simm[15:0], 0^{16})$
Test Set Operation		
011000	$clri$	$RD = 0^{32}$
011001	$sgri$	$RD = ([RS1] > [simm]?0^{31}1 : 0^{32})$
011010	$seqi$	$RD = ([RS1] = [simm]?0^{31}1 : 0^{32})$
011011	$sgei$	$RD = ([RS1] \geq [simm]?0^{31}1 : 0^{32})$
011100	$slsi$	$RD = ([RS1] < [simm]?0^{31}1 : 0^{32})$
011101	$snei$	$RD = ([RS1] \neq [simm]?0^{31}1 : 0^{32})$
011110	$slei$	$RD = ([RS1] \leq [simm]?0^{31}1 : 0^{32})$
011111	$seti$	$RD = 0^{31}1$
Control Flow Operation		
000100	$beqz$	$\langle PCP \rangle = \langle PCP \rangle +_{32} (RS = 0^{32}?[simm] : 0)$
000101	$bnez$	$\langle PCP \rangle = \langle PCP \rangle +_{32} (RS \neq 0^{32}?[simm] : 0)$
010110	jr	$PC = RS1$
010111	$jalr$	$\langle R31 \rangle = \langle PCP \rangle +_{32} 4, \quad PCP = RS1$
111110	$trap$	$causestrap$
111111	rfe	$SR = ESR, \quad PC = EPC, \quad DPC = EDPC$

References

[BJK+03] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In D. Geist and E. Tronci, editors, *Proc. of the 12th Advanced Research Working*

Conference on Correct Hardware Design and Verification Methods (CHARME), volume 2860, pages 51–65. Springer, 2003.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603, pages 1–16. Springer, 2005.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.