Putting it all together – Formal verification of the VAMP[*],[**]

Sven Beyer[1], **Christian Jacobi**[2,***], **Daniel Kröning**[3,***,†], **Dirk Leinenbach**[1,‡], **Wolfgang J. Paul**[1]

[1] Saarland University, Computer Science Department, 66123 Saarbrücken, Germany
e-mail: {sbeyer,dirkl,wjp}@cs.uni-sb.de
[2] IBM Deutschland Entwicklung GmbH, 71032 Böblingen, Germany
e-mail: cjacobi@de.ibm.com
[3] ETH Zürich, Computer Systems Institute, Zürich, Switzerland
e-mail: daniel.kroening@inf.ethz.ch

Abstract. In the VAMP (verified architecture microprocessor) project we have designed, functionally verified, and synthesized a processor with full DLX instruction set, delayed branch, Tomasulo scheduler, maskable nested precise interrupts, pipelined fully IEEE compatible dual precision floating point unit with variable latency, and separate instruction and data caches. The verification has been carried out in the theorem proving system PVS. The processor has been implemented on a Xilinx FPGA.

1 Introduction

1.1 Previous work

Work on the formal verification of processors so far has concentrated mainly on the following aspects of architectures:

i) Microprocessors with in-order scheduling, one or several pipelines including result forwarding, stalling, and interrupt mechanisms [11, 29, 49]. The verification of the very simple, non-pipelined FM9001 processor is reported in [9, 10]. Using the flushing method from [11] and uninterpreted functions for modeling functional units, superscalar processors with multicycle execution units, exceptions and branch prediction [49] have been verified by automatic BDD based methods. Also, one can transform specification machines into simple pipelines (with forwarding and stalling mechanism) by an automatic transformation, and automatically generate formal correctness proofs for this transformation [31].

ii) Tomasulo schedulers with reorder buffers for the support of precise interrupts [14, 19, 33, 43]. Exploiting symmetries, McMillan [33] has shown the correctness of a powerful Tomasulo scheduler with a remarkable degree of automation. Using theorem proving, Sawada and Hunt [20, 43, 44] show the correctness of an entire out-of-order processor, precise interrupts, and a store buffer for the memory unit. They also consider self-modifying code (by means of a *sync* instruction).

iii) Floating point units. The correctness of an important collection of floating point algorithms is shown in [40, 41] using the theorem proving system ACL2. Using a combination of theorem proving and model checking techniques, correctness proofs for the floating point units of Pentium processors are reported in [13, 38]. Based on the constructions and on the paper and pencil proofs in [37] a fully IEEE compatible floating point unit has been verified [3, 24, 25] (using mostly but not exclusively theorem proving). In [46] and [26] the verification of fused-multiply-add FPUs is reported.

iv) Caches. Multiple cache coherence protocols have been formally verified, e.g., [15, 34, 45, 47]. Paper and pencil proofs are error prone, and hence the generation of proofs for interactive theorem proving systems is slow. The method of choice is model checking. The compositional techniques employed by McMillan [34] even allow for the verification of parameterized designs, i.e., cache coherence is shown for an arbitrary number of processors.

[*] This paper has been originally published in the International Journal on Software Tools for Technology Transfer (STTT). The original publication is available at www.springerlink.com. DOI: 10.1007/s10009-006-0204-6

[**] This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors.

[***] The work reported here was done while the author was with Saarland University.

[†] Research supported by the DFG graduate program 'Effizienz und Komplexität von Algorithmen und Rechenanlagen'

[‡] Research supported by the DFG graduate program 'Leistungsgarantien für Rechnersysteme'

1.2 Simplifications, abstractions and restrictions

Except for the work on floating point units, the cache coherence protocol in [15], and the FM9001 processor [9], *none* of the papers quoted above states that the verified design actually has been implemented. *All* results cited above except [3, 9, 15, 24, 25] use several simplifications and abstractions:

i) The realized instruction set is restricted: always included are the six instructions considered in [11]: load word, store word, jump, branch equal zero, ALU register operations, ALU immediate operations. Five typical extra instructions are trap, return from exception, move to and from special registers, and sync [43]. The branch equal zero instruction is generalized in [49] by an uninterpreted test evaluation function. Most notably the verification of machines with load/store operations on half words and bytes has apparently not been reported. In [48] the authors report an attempt to handle these instructions by automatic methods which was unsuccessful due to memory overflow.
ii) Delayed branch is replaced by non-deterministic speculation (speculating branch taken/not taken).
iii) Sometimes, non-implementable constructs are used in the processors: e.g., Hosabettu et.al. [19] use tags from an infinite set. Obviously, this is not directly implementable in real hardware.
iv) The verification of Intel's and AMD's FPUs does neither cover the handling of denormal numbers nor of exception flags. The verification of a dual precision FPU has not been reported (though, obviously, Intel's and AMD's FPUs are capable of dual precision). IBM's FPU verification [26] does handle denormal numbers, exception flags, and dual precision, but does not cover the verification of the multiplier array.
v) No verification of a memory unit with caches has been reported. Eiriksson [15] only reports the verification of a bit-level implementation of a cache coherence protocol without data consistency.
vi) The verification of pipelines or Tomasulo schedulers with *instantiated* floating point units and memory units with caches and main memory bus protocol has not been reported apart from the VAMP project [6]. Indeed, in [48] the authors state: "An area of future work will be to prove that the correctness of an abstract term-level model implies the correctness of the original bit-level design."

1.3 Results and overview

In the VAMP project [8] we have designed, formally verified, and synthesized a processor with full DLX instruction set, delayed branch, Tomasulo scheduler [28], maskable nested precise interrupts [6], pipelined fully IEEE 754 [21] compatible dual precision floating point units with variable latency [2, 3, 23–25], as well as separate, coherent instruction and data caches [6]. The caches are connected to a unified main memory with arbitrary variable latency. The main memory uses a bus protocol that supports burst accesses [6]. We use only finite tags in the hardware. Thus all abstractions, restrictions, and simplifications mentioned above have been removed. Specification and verification was performed using the interactive theorem proving system PVS [39].

Our hardware is written in a small subset of the PVS language. We use recursion and module instantiation for structured design, but the complete design can be unrolled down to the level of single bits and gates. This subset of the PVS language can be easily translated into common hardware description languages. For that purpose, we have developed a translation tool `pvs2hdl` [7] that takes our PVS hardware description and translates it into gate-level Verilog HDL. The `pvs2hdl` tool is not formally verified. Using Xilinx synthesis tools, we have implemented the VAMP [32] on a Xilinx FPGA. We add a small non-verified logic to the FPGA for bridging between the external SDRAM and the VAMP main memory protocol. We have ported the `gcc` and the `glibc` to the VAMP architecture [35]. The verified VAMP is running small test-applications on the FPGA; we did not find a single bug after completing the formal verification of the VAMP in PVS.

All PVS specifications and proofs, the Verilog files, and the sources of `pvs2hdl`, `gcc`, and `glibc` are available at our web site [1].

The rest of this paper is organized as follows. In section 2 we summarize the fixed point instruction set, its floating point extension, and the interrupt support in the VAMP. We give a micro-architectural overview. Section 3 describes the correctness criterion, the main proof strategy, and the integration of the functional units into the Tomasulo core. Correctness criterion and proof strategy are based on scheduling functions [30, 37], which are similar to the stg-component of MAETTs [42]. The model of the functional unit is in a nontrivial way more general than previous models without complicating interactive proofs too much. Section 4 presents a delayed branch mechanism and summarizes the specification and verification of an interrupt mechanism for maskable nested precise interrupts and delayed PC from [37]. Section 5 describes the verification of the floating point units. We focus on the decomposition of the FPU correctness proofs into the correctness of the algorithms, the correctness of the combinational circuits that implement these algorithms, and the verification of the pipelining of these combinational circuits.

Section 6 introduces the concept of a memory interface that plays a central role in the decomposition of the correctness proofs of the VAMP with split caches. We summarize the proof ideas used in the verification of a CPU accessing such a memory interface, in particular dealing with loads and stores of variable operand width, precise interrupts, and self-modifying code. An implementation of a cache memory interface is given in section 7 together with a sketch of the proof that this implementation really fulfills the specification of a memory interface. Section 8 describes how the VAMP hardware is described in PVS, the translation tool `pvs2hdl`, and the implementation of the VAMP on a Xilinx FPGA. Section 9 gives an overview of the verification effort for various parts of the project. Finally, section 10 summarizes our work, and sketches directions of some future work.

2 Overview of the VAMP processor

2.1 Instruction set

The full DLX instruction set from [17] is realized. This includes loads and stores for double words, words, half words, and bytes, various shift operations, and two jump-and-link operations. Loads of bytes and half words can be unsigned or signed. In order to support the pipelining of instruction fetches, delayed branch with one delay slot is used. Note that delayed branch changes the sequential semantics of program execution.

The floating point extension of the DLX instruction set from [37] is supported. The floating point register file comprises 32 registers of single precision numbers as well as a single floating point condition code register FCC. Pairs of floating point registers can be accessed as registers for double precision numbers (with an even register address). Supported operations are: i) loads and stores for singles and doubles. ii) $+$, $-$, $\times$, $\div$ both for single and double precision numbers. iii) test-and-set, the result is stored in FCC. iv) conditional branches as a function of FCC. v) conversions between singles, doubles and integers. vi) moves between the general purpose register file and the floating point register file. Operations are fully IEEE compatible [21]. In particular, all four rounding modes, denormal numbers, and exponent wrapping as a function of the interrupt masks are realized. Rounding mode and interrupt masks are stored in the special purpose register file (SPR).

2.2 Interrupt support

Presently, the VAMP supports 13 internal interrupts (cf. table 1 in section 4.2). Interrupts are maskable and precise. Floating point interrupts are accumulated in 5 bits of a special purpose register *IEEEf* (IEEE flag) as required by the IEEE standard. All special purpose registers are collected into a special purpose register file (details in section 4.2). Operations supporting the interrupt mechanism are: i) moves between general purpose registers and special purpose registers. ii) trap. iii) return-from-exception.

2.3 Microarchitecture overview

Figure 1 gives a high level overview of the VAMP microarchitecture. Stages IF and ID—instruction fetch and decode—realize a pipelined implementation of delayed branch as explained in section 4. The lower three stages are “execution” EX, “completion” C, and “writeback” WB. Together they realize a Tomasulo scheduler with 5 functional units, a fair scheduling policy on the common data bus CDB, and a reorder buffer ROB for precise interrupts. The reservation stations hold up to 6 operands for any instruction as explained in section 5. The functional units can produce up to 4 results per instruction: double precision results are produced as 2 results, each 32 bit wide. This allows for a simple embedding of 64 bit wide data in a mostly 32 bit wide processor. Additionally, the functional units produce interrupt flags (called “exception cause”, ECA) and exception data (EData) that is saved in case of an interrupt. For the memory unit, e.g., the exception data consists of the effective address of the memory access. Hence, a total of up to 4 results has to be stored in the producer registers per instruction.

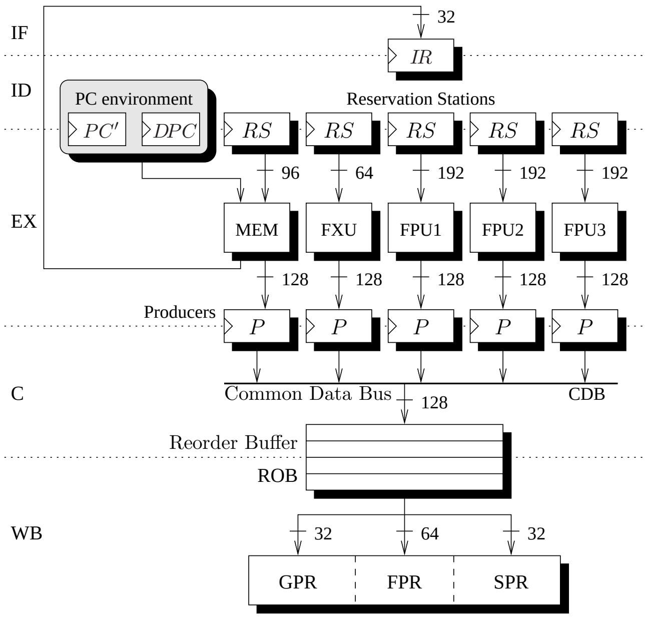


Fig. 1. Main data paths of the VAMP processor

The VAMP offers 8 reservation stations: 4 reservation stations serve the fixed point unit (FXU); the memory unit and the three FPUs are each served by one reservation station. The reorder buffer also has 8 entries. The data output of the reorder buffer is 64 bits wide. The floating point register file FPR is physically realized as 16 registers, each 64 bits wide. The general purpose registers file GPR and the special purpose register file SPR are both 32 bits wide, and have 32 and 9 entries, respectively. They are connected to the low-order bits of the ROB output.

Some instructions are not issued into any reservation station, but directly into the ROB since they do not really ‘compute’ anything and thus do not need to enter a functional unit. This special issuing occurs for return from exception, traps, branch, and jump instructions, as well as moves between SPR and GPR. We support jumps to immediate and register destinations both with or without storing the return address in register 31 of the GPR for subroutine calls.

2.3.1 Memory unit

The memory unit has two pipeline stages and supports loads and stores of bytes, halfwords, words, and doubles. The floating point memory operations support single and double precision data, the integer memory operations support bytes, halfwords, and words. For integer load operations, both signed and unsigned versions are implemented. Misaligned memory accesses as specified in section 6.2 raise an exception. The memory instructions are processed in order. Currently, there is no store buffer and no support for address translation.

The memory unit also fetches instructions from the program counter into the instruction register IR. The memory unit internally consists of a split instruction and data cache, which allows for concurrent data and instruction accesses. The two caches are kept coherent. They are connected to a unified main memory.

2.3.2 Fixed point unit

The fixed point unit supports the usual set of addition, subtraction, shifts, test, and logical operations on 32-bit integer operands. There are two versions of the addition and subtraction instructions, one that raises an arithmetical interrupt in case of an overflow and one that does not. Both arithmetical and logical shifts to the left and to the right are supported. All operations are supported in a version with two source registers as well as with one source register and a sign-extended immediate constant from the instruction word. The FXU is combinational.

2.3.3 Floating point units

The VAMP features three specialized pipelined floating point units with variable latency. FPU1 performs additions and subtractions, FPU2 multiplications and divisions, and FPU3 test-and-set as well as conversions. All units support single and double precision operations. The full set of exceptions according to [21] is supported.

3 Correctness criterion and Tomasulo algorithm

3.1 Notations

We consider a specification machine S and an implementation machine I. Configurations of these machines are tuples, whose components R_S and R_I, respectively, are registers or memories. Register contents are bit strings. Memory contents are modeled as mappings from addresses (bit strings) to bit strings. For example, PC_S denotes the program counter of the specification machine, and M_I denotes the main memory of the implementation machine. Note that M_I actually represents the abstract layer of a memory interface as introduced in section 6; thus, the actual implementation of the cache memory interface of the VAMP is hidden from the top-level proof.

The specification machine processes a sequence of instructions $I_0, I_1, \ldots$ at the rate of one instruction per step. We denote by R_S^i the content of component R *before* execution of instruction I_i. One step of the implementation machine is a hardware cycle, and we denote by R_I^t the content of component R during cycle t. The fetch of the 4 bytes of an instruction into the instruction register IR of the implementation machine during cycle t can be specified by $IR_I^{t+1} := M_I^t[PC_I^t + 3 : PC_I^t]$.

Although the instruction register is not a visible register, one can specify the desired content IR_S^i of the instruction register for the specification machine for instruction I_i as a function of the visible components by $IR_S^i = M_S^i[PC_S^i + 3 : PC_S^i]$. Defining the next configuration c_S^{i+1} of the specification machine involves many such intermediate definitions, e.g., the immediate constant imm_S^i, the effective memory access address ea_S^i, etc. Starting from the visible components R_S we extend the configuration of the specification machine in this way by numerous (redundant) secondary components.

3.2 Scheduling functions

For hardware cycles t and pipeline stages k of the implementation machine, we formally define an integer valued scheduling function $sI(k, t)$ [30], where $sI(k, t) = i$ has the intended meaning that an instruction I_i is in stage k during cycle t.

By treating instruction numbers like integer valued tags,[1] the definition of these scheduling functions for invisible registers is straightforward. We initialize $sI(k, 0) := -1$ for all stages in order to model that the register contains arbitrary data initially. We then “clock” these tags through the pipeline stages under the control of the update enable signals[2] ue_k for the output registers of stage k. If a stage is not clocked, the scheduling function is not changed, i.e., $sI(k, t + 1) := sI(k, t)$. Note that we introduce separate “stages” k for each reservation station and ROB entry.

If stage k receives data from stage k' in cycle t, we define $sI(k, t + 1) := sI(k', t)$. Note that this covers the case that a stage can receive data from two different stages, say k' and k'', since in a fixed cycle t, it receives data from only one of these stages. This occurs at the ROB, e.g., where we allow bypassing branch instructions from the instruction register directly into the ROB without going through a functional unit. Thus, the ROB can receive data from the CDB and from the instruction register. This covers all cases except for the initial pipeline stage, i.e., the decode stage.[3] Therefore, we define $sI(dec, t + 1) := sI(dec, t) + 1$ if a new instruction is fetched into the VAMP pipeline in cycle t since the con-

[1] Having integer valued tags is only a proof trick. In hardware, we only use finite tags. During the proof of correctness for the Tomasulo scheduler, we prove that these finite tags properly match to the infinite instruction number.

[2] Update enable signals are sometimes called ‘register activates’. They are used to (de-)activate updating of register contents.

[3] We introduce symbolic names for some stages k, e.g., dec and mem.

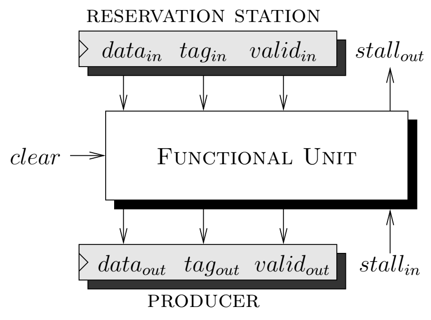


Fig. 2. Model of a functional unit

tent of the decode stage progresses by one instruction in the instruction stream $I_0, I_1, \ldots$

For visible registers on the other hand, we introduce a different kind of scheduling function. Let R be a visible register that is updated from an invisible register in some stage k, e.g., take the visible memory M that is updated from the memory stage mem in the memory unit. Initially, we define the scheduling function for the visible register by $sI(R, 0) := 0$. If ue_k^t holds, we have $sI(R, t+1) := sI(k, t) + 1$; otherwise, we simply have $sI(R, t+1) = sI(R, t)$. We apply this principle of scheduling functions for visible registers to the program counters, the register files, and the memory;[4] we introduce the shorthand notation $sI(wb, t)$ for the scheduling function of *all* register files.

In order to argue about the program counter for the correctness of instruction fetch in section 6.3, we introduce an additional scheduling function $sI(fetch, t) := sI(dec, t) + 1$ that is not used in our correctness criteria.

3.3 Correctness criterion

We are interested in the correctness of all visible registers after certain instructions I_i respectively before instruction I_{i+1}. Hence, we use the scheduling function for visible registers and claim for any $R \in \{DPC, PC', M, GPR, FPR, SPR\}$ that $R_I^t = R_S^{sI(R,t)}$ holds. For any invisible register R in some stage k, we claim $R_I^t = R_S^{sI(k,t)}$ in case of $sI(k, t) \geq 0$, where R_S^i is a redundant component of the configuration of the specification machine. We introduce the shorthand notation $corr?(t)$ for the correctness of *all* registers in the VAMP, i.e., both visible and invisible registers. In general, we prove $corr?(t)$ by induction on t. Note that we derive the initial specification configuration c_S^0 from the initial implementation configuration c_I^0 by just 'copying' all the visible registers.

The liveness criterion states that all instructions that are not interrupted reach the writeback stage. We have separate formal liveness proofs for the scheduler and the functional units, but currently no combined liveness proof for the entire machine. This is future work.

[4] Note that we introduced a 'bookkeeping' stage mem' with its corresponding scheduling function $sI(mem', t)$ for the visible memory in [8] instead of the more intuitive version of $sI(M, t)$ we present here.

Paper and pencil proofs for the correctness of Tomasulo schedulers tend to follow a canonical pattern: i) For instructions I_i and register operand R, one defines $last(i, R)$ as the index of the last instruction before I_i which wrote register R. ii) One shows by induction that the formal definitions of tags and valid bits have the intended meaning. In our setting, this means that the finite tags in hardware correspond to the integer valued tags provided by the scheduling function sI. iii) Finally, one has to show that the reservation station of instruction I_i reconstructs $R_S^{last(i,R)}$. The rest is easy.

It is important to observe that the structure of these paper and pencil proofs and their formal (theorem proving) counter parts do not depend much on the fixed or variable latency of functional units or whether these units are pipelined. The Tomasulo scheduler recognizes instructions completed by the functional units simply by examining the tags returned from the units. The situation is very different for model checking [49].

3.4 Integration of functional units

The overall correctness proof is decomposed into a scheduler proof as outlined above and several functional unit proofs by the following specifications for the functional units [23, 24]. Notations refer to figure 2.

i) $\forall t : stall_{in}^t \implies \neg valid_{out}^t$, i.e., if the scheduler asserts $stall_{in}$, the functional unit does not return a valid instruction.
ii) $\forall t \exists t' > t : \neg stall_{out}^{t'}$, i.e., the $stall_{out}$ signal is never active indefinitely.
iii) Data-liveness: instructions dispatched with $tag_{in} = tg$ at time t will eventually (at time $t' \geq t$) return a result with the same tag , i.e., $tag_{out}^{t'} = tg$. Moreover, $data_{out}^{t'} = f(data_{in}^t)$ where f is the (combinational) function the functional unit is supposed to compute.
iv) Tag-consistency: for each time t at which a result with tag tg is returned, there is an earlier time $t' \leq t$ such that an instruction with tag tg was dispatched at time t', and tag tg was not returned between t' and t. Hence, the functional units do not create spurious outputs.

These four conditions must be shown for each of the functional units provided the scheduler guarantees the following three conditions: i) No instruction is dispatched to a functional unit which sends a $stall_{out}$ signal to its reservation station. ii) The functional units are not stalled forever by the producers. iii) Tag-uniqueness: no tag which is dispatched into a functional unit is currently in use. Observe, that tags may be reused after writeback of the instruction. Other researchers [19] have verified Tomasulo-schedulers with infinite tags and without reusing tags. This makes the verification of tag-consistency and tag-uniqueness much simpler; however, infinite tag spaces are obviously not implementable in hardware.

Note that the above specification does not require that instructions leave the functional units in the same order they enter the units, i.e., functional units may reorder instructions internally. All three VAMP FPUs exploit this, as will be seen in section 5.3.

3.5 *IEEEf* implementation and correctness

The IEEE standard [21] requires that every floating-point operation computes 5 exception bits (e.g., overflow and underflow), which are accumulated in status flags. Each flag is set whenever the corresponding exception occurs, and it is reset only through explicit writes to the status flag. In the VAMP, the 5 status flags are stored in the special purpose register *IEEEf*, which is updated after every FPU instruction, and which can be written and read explicitly by means of moves between the SPR and GPR.

Since new exception bits have to be or-ed to the *IEEEf* register for every FPU instruction, formally *IEEEf* is both source and destination operand of every floating point instruction. If *IEEEf* would be handled by the Tomasulo scheduler as a regular register, at most one floating point instruction could be in all three FPUs *together* at any time. In order to significantly increase performance and keep the size of reservation stations, CDB, and ROB small, we instead update *IEEEf* during writeback in a special way: every FPU instruction computes the new exceptions and stores them in the exception cause (ECA) register that is present in the ROB anyway. When the instruction is written back from the ROB into the register file, this ECA register is logically or-ed to the *IEEEf* register. That means that the standard dependency check performed by the Tomasulo algorithm is not performed for the *IEEEf* register.

Explicit writes to *IEEEf* by means of a special move instruction are not affected by this change in the Tomasulo algorithm. However, the forwarding mechanism of the standard Tomasulo algorithm is then no longer correct for explicit reads of *IEEEf*, i.e, special moves between SPR and GPR with source register *IEEEf* require additional consideration. We therefore add a hardware synchronization for any instruction I_i explicitly reading *IEEEf*: instruction issue of I_i is stalled until the reorder buffer has run empty, i.e., until no other instruction that might possibly update the *IEEEf* register is alive in the VAMP processor.

In addition to the Tomasulo algorithm with reorder buffer, we verified the correctness of this *IEEEf* extension and its implementation in the VAMP [6, Chap. 4.4.1]. Note that this synchronization with the high penalty of letting the VAMP run empty is only necessary for explicit reads of *IEEEf* – the much more numerous standard floating point instructions are not affected. A move instruction from *IEEEf* to general purpose register 0, which is constantly 0, acts as a *sync* operation for self-modifying code as described in section 6.3.

4 Delayed branch and maskable nested precise interrupts

4.1 Delayed branch

We have separate instruction fetch and decode stages in the VAMP as depicted in figure 1. While some branch instruction I_i is in the decode stage we already fetch the next instruction I_{i+1}. In order to evaluate the branch condition prior to the next instruction fetch, one would basically have to do instruction decode and fetch in one cycle which increases cycle time considerably. Therefore, only two feasible solutions remain: i) Predict the program counter of the next instruction and perform a rollback in case of misprediction. ii) Change the semantics of the assembler instruction set such that jumps and branches take effect only with a delay of one instruction. We decided to use the delayed branch mechanism in the VAMP which delays the effect of taken branches by one instruction. In the delayed branch mechanism, taken branches yield a new PC of the form $PC + imm + 4$, and $PC + 8$ is saved to the register file during jump-and-link.

We implement the delayed branch with the equivalent delayed PC mechanism [30, 37] where *all* PC computations are delayed by one instruction. Thus, the two PCs basically form a two-stage pipeline in the *specification*. For delayed PC one uses an intermediate program counter PC' with branch targets $PC' + imm$, *all* fetches use a delayed program counter DPC, and $PC' + 4$ is saved during jump-and-link.

Figure 3 depicts a pipelined implementation of the delayed PC mechanism in the VAMP processor. Indeed, fetching instructions from the intermediate program counter PC' is—not only intuitively but formally—forwarding of DPC. The role of the multiplexers above DPC are explained in the following section about interrupts.

4.2 Interrupt mechanism and implementation

The formal specification of the interrupt mechanism for delayed PC is based on the definitions of [37, Chap. 5, 9.1]. Table 1 shows the supported interrupts.[5] The special purpose registers for the interrupt mechanism are: i) status register SR for interrupt masks, ii) two registers ECA for exception cause and $EData$ for parameters passed to the interrupt service routine, iii) *two* registers EPC and $EDPC$ for return addresses for PC' and DPC and iv) a register *IEEEf* for the accumulation of masked floating point exceptions. Return from exception is achieved by means of a simple instruction that copies the exception PCs back to the actual PCs; hence, no additional effort is needed for the verification since return from exception is already covered by the proof without interrupts.

Interrupts are detected and acted upon during instruction writeback in the VAMP. As introduced before, both exception cause and exception data are part of each instruction's results in the reorder buffer. Based on the exception cause of

[5] Page fault signals and external interrupts are presently tied to zero.

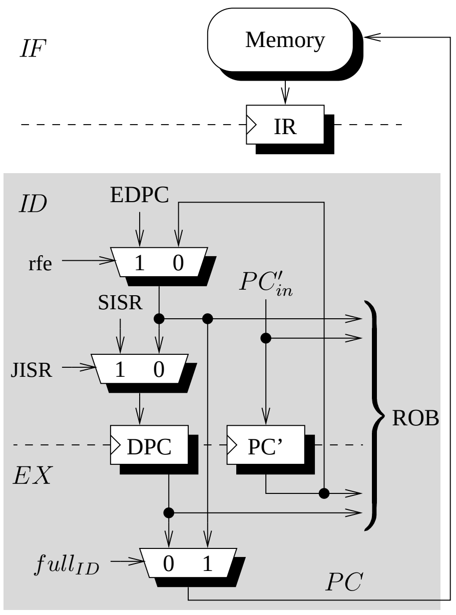


Fig. 3. VAMP PC environment

index	name	maskable	type
0	reset	no	abort
1	illegal instruction	no	repeat
2	misalignment	no	repeat
3	page fault on fetch	no	repeat
4	page fault load store	no	repeat
5	trap	no	continue
6	arithmetic overflow	yes	continue
7	FPU overflow	yes	continue
8	FPU underflow	yes	continue
9	FPU loss of accuracy	yes	continue
10	FPU division by zero	yes	continue
11	FPU invalid	yes	continue
12	FPU unimplemented	no	continue

Table 1. Implemented interrupts

the instruction that is currently written back and the value of the interrupt mask register SR in the special purpose register file, the implementation computes a signal $JISR$ that is active iff an interrupt occurs during writeback. If $JISR$ is active, the program counters are set to the start of the interrupt service routine, $SISR$, all interrupts are masked, and the interrupt cause as well as the PCs of the instruction that is supposed to be executed after the return from the interrupt service routine are saved into special purpose registers. Additionally, all instructions in the VAMP are squashed, i.e., all execution units, reservation stations, producer registers, the reorder buffer, and the instruction register are emptied and all registers in the three register files are set to valid. This results in an *initial* VAMP configuration after an interrupt, i.e., an empty VAMP where all registers are valid and the PCs point to the start of the interrupt service routine.

Depending on the type of the interrupt, either the DPC and PC' of the *interrupted* instruction or those of the next instruction have to be saved into the register $EDPC$ and EPC for return from exception. At issue time of an instruction I_i, it is unknown whether I_i will be interrupted and whether the interrupt requires to repeat the interrupted instruction or not. Therefore, we have to save *two pairs* of potential return addresses in the reorder buffer: $(PC'^{i}_{S}, DPC^{i}_{S})$ for interrupts of type 'repeat', and the results of the *uninterrupted* next PC' and next DPC computations $(PC'^{u,i+1}_{S}, DPC^{u,i+1}_{S})$ for interrupts of type 'continue'. In case of an interrupt, the correct pair of PCs in the ROB is then selected depending on the type of the interrupt. The data paths of the PC environment are shown in figure 3.

In the implementation, we have to take care that interrupts are precise with respect to all visible registers. Since the program counters are forced to the start of the interrupt service routine after an interrupt, preciseness for the PCs is trivial. For the register files, preciseness is also easy to achieve. During a normal writeback as well as on a 'continue' interrupt, the result of the instruction at the head of the reorder buffer is written back to the register file. However, on a 'repeat' interrupt, the result is not written back to the register file since the instruction is supposed to be executed again.

For the memory unit, we finally have to take care that stores are only initiated if they cannot be squashed by an interrupt any more. Therefore, a store instruction is held back in the memory unit until it is the oldest instruction in the VAMP, i.e., until its tag matches that of the instruction at the head of the ROB. Note that loads are not affected by this change.

4.3 Correctness criterion with interrupts

The correctness arguments outlined in section 3.3 can only be applied in the absence of interrupts. We proved $corr?(t)$ which guarantees correctness between specification registers and implementation registers without interrupts. For correctness with interrupts, we first note that we have to claim *less* since the VAMP with interrupts may start the execution of instructions that will later be squashed and that hence are not executed at all in the specification. These instructions may alter the internal state of the VAMP in a way that is not consistent with the specification—of course these changes have to be rolled-back when the instructions are squashed. We therefore define a scheduling function $sI(inst, t)$ that also takes interrupts into account: it is incremented when either an instruction leaves the VAMP and writes its data back into the register files or when an interrupt occurs. This definition is due to the fact that a computation step in the specification is given either by the execution of an instruction or an interrupt. The correctness criterion with interrupts is then exclusively based on this $sI(inst, t)$.

In contrast to $corr?(t)$, we introduce a new correctness with interrupts, $corr\_i?(t)$, which only covers visible registers. Additionally, we have to further restrict $corr\_i?(t)$ since the program counters may also take values never encountered in the specification and instructions may read the memory

$$\begin{tikzcd}
& & c_I^0 \arrow[rr, leftrightarrow, "corr\_i?(0)"] & & c_S^0 \\
corr? \left\{ \right. & \neg JISR \left\{ \right. & \vdots & & \vdots \\
& JISR^t & c_I^t \arrow[rr, leftrightarrow, "corr\_i?(t)"] & & c_S^{sI(inst,t)} \\
& & c_I^{t+1} \arrow[rr, leftrightarrow, "corr\_i?(t+1)"] & & c_S^{sI(inst,t+1)}
\end{tikzcd}$$

Fig. 4. Correctness after the first interrupt

which are not executed in the specification at all. Hence, our correctness criterion $corr\_i?(t)$ looks as follows: We claim full correctness for the register files, i.e., $R_I^t = R_S^{sI(inst,t)}$ for $R \in \{GPR, SPR, FPR\}$, but for the program counters and the memory, we only claim correctness in the initial cycle or in the cycle immediately after an interrupt, i.e., for $R \in \{PC', DPC, M\}$, we claim

$$(t = 0 \vee JISR_I^{t-1}) \implies R_I^t = R_S^{sI(inst,t)}.$$

Although the correctness criterion $corr\_i?$ is weaker than without interrupts, we believe it is sufficiently strong to cover an intuitive understanding of correctness. The programmer is guaranteed to observe the correct content of the register files in every cycle, and any error in the program counters or memory can easily be propagated into the register files; also, after each interrupt, the correctness criterion guarantees that the VAMP has the correct value on the complete architectural state. Note that we leave out the PCs and the memory on the intermediate cycles not because of proof complexity (we have to cover that for correctness of the register files anyway), but because of specification complexity.

4.4 Interrupt proof overview

The main idea of integrating interrupt support into the correctness proof is the following:

i) In the absence of interrupts, we trivially have $sI(wb, t) = sI(inst, t)$ since $sI(wb, t)$ counts instructions leaving the pipeline and writing back their data without interrupts. In addition, $corr\_i?(t)$ does not claim anything for the memory and the program counters in the absence of interrupts apart from the initial cycle. Therefore, $corr?(t)$ implies $corr\_i?(t)$ as long as no interrupts occur.[6]

ii) We show that correctness with interrupts holds until one cycle *after* the first interrupt. This is illustrated in figure 4. Up to the cycle of the first interrupt itself, correctness with interrupts holds by i) above. Hence, we can assume that the first interrupt occurs in cycle t and we have to show that the VAMP state after the 'interrupt' step corresponds to the specification state after the interrupt, i.e., $corr\_i?(t + 1)$ holds which means $R_I^{t+1} = R_S^{sI(inst,t+1)}$ for *all* visible registers R including PCs and memory since $JISR^t$ holds. Therefore, we have to do a brute force proof that the one hardware cycle of the interrupt corresponds to one step in the specification.

$$\begin{tikzcd}
c_I^{l+1} \arrow[rr, leftrightarrow, "corr\_i?(l+1)"] \arrow[d, equal] & & c_S^{sI(inst,l+1)} \arrow[d, equal] \\
{[c_I^0]} \arrow[rr, leftrightarrow, "{[corr\_i?(0)]}"] \arrow[d, "\left.\right\} \text{interrupt-free segment as depicted in figure 4}"'] & & {[c_S^{sI(inst,0)}]} \arrow[d] \\
{[c_I^{t-l}]} \arrow[rr, leftrightarrow, "{[corr\_i?(t-l)]}"] \arrow[d, equal] & & {[c_S^{sI(inst,t-l)}]} \arrow[d, equal] \\
c_I^{t+1} \arrow[rr, leftrightarrow, "corr\_i?(t+1)"] & & c_S^{sI(inst,t+1)}
\end{tikzcd}$$

Fig. 5. Showing overall correctness with interrupts

iii) We then use the state of the VAMP after the interrupt, c_I^{t+1}, as initial state for a new instantiation of the correctness proof without interrupts. Note that in order to do so, the correctness proof without interrupts is parameterized over an initial state instead of referring to an arbitrary, but fixed, initial state. As introduced in section 3.3, the initial state for the specification is derived from the initial VAMP state by just copying the visible registers. In this particular case with $JISR^t$ and $corr\_i?(t + 1)$ by the above item, this 'copying' of visible registers from c_I^{t+1} yields $c_S^{sI(inst,t+1)}$ as initial state of the specification for the proof without interrupts.

We show $corr\_i?(t)$ by induction; for the main case in the induction step, we know $JISR^l$ and $corr\_i?(l + 1)$ by induction hypothesis where l is the cycle of the last interrupt before t. Note that $l + 1 \leq t$ holds in particular. We use the shorthand notation $[\cdot]$ for configurations based on c_I^{l+1} as initial state, i.e., $[c_I^x] = c_I^{x+l+1}$ holds. Note that $[c_I^{t-l}]$ trivially equals c_I^{t+1}. Figure 5 illustrates our proof idea for overall correctness. By item iii) above, $[c_S^{sI(inst,0)}] = c_S^{sI(inst,l+1)}$ holds since $corr\_i?(l+1)$ and $JISR^l$ both hold. Hence, we can conclude $[c_S^{sI(inst,t-l)}] = c_S^{sI(inst,t+1)}$ and it is sufficient to show $[corr\_i?(t - l)]$. By the definition of l, the $t - l$ cycles from $l+1$ to t contain at most one interrupt, i.e., in cycle t itself. As introduced in ii), we can therefore conclude $[corr\_i?(t - l)]$ and thus finish the proof.

We now give some details on doing the 'interrupt step' proof as introduced in ii), i.e., on showing $corr\_i?(t + 1)$ given both $JISR^t$ and $corr?(t)$. In this case, $corr\_i?(t + 1)$ covers all visible registers, i.e., memory, program counters, and register files. Interrupt handling in the specification machine S depends on the components ECA and $EData$. In the implementation, the formal correctness of these components in the ROB in cycle t is asserted without additional verification effort by $corr?(t)$. Hence, the implementation raises

[6] Note that $corr?(t)$ contains a claim $M_I^t = M_S^{sI(M,t)}$ with respect to the memory's scheduling function $sI(M, t)$ and a corresponding claim for the program counters; however, this allows no immediate conclusion on the memory or the program counters with respect to $sI(inst, t)$ except when the VAMP pipeline is empty.

an interrupt in cycle t iff the specification does for the corresponding instruction $sI(wb, t)$. In case of an interrupt, the program counters are forced to the start of the interrupt service routine in both implementation and specification; thus, correctness of the program counters after an interrupt is trivial. Details on the preciseness of interrupts for the memory unit are discussed in section 6.4.

For the register files, we basically have to show the following two things: i) the result of the interrupted instruction is written back to the register file iff the interrupt is of type continue and ii) some special registers are updated in case of an interrupt according to the specification. Since exception cause and data in the implementation are correct in the cycle of an interrupt by the arguments above, the interrupt in the implementation also has the correct type, i.e., repeat or continue. By omitting writeback in case of a continue interrupt, correctness of writeback in case of an interrupt is easily achieved. Further lemmas are needed for the correctness of the PCs stored in the ROB since two of them are saved into special registers in order to allow the interrupt handler to continue the interrupted program. Details of our interrupt proof are reported in [6, Chap. 4.5].

5 Floating point unit

The Floating Point Unit actually comprises three execution units: the multiplicative unit which performs multiplication and division, the additive unit for addition and subtraction, and the "misc" unit for floating-point compares and conversions. All these units can handle both single and double precision data.

The verification of each of the FPUs is split into three steps: i) formalization of the IEEE standard [21] plus a set of definitions and theorems about rounding; ii) verification of the combinational FPUs against the definitions from i), and iii) verification of the pipelining of the FPUs. We now describe these three steps.

5.1 IEEE standard and theory of rounding

The basis of the FPU verification is a formalization of the IEEE standard 754 [21]. The formalization covers the definition of floating-point numbers as tuples of sign, exponent, and fraction. The exponent and fraction in this formalization are numbers rather than bitvectors, which facilitates the mathematical reasoning about floating-point numbers. Rounding is captured as combination of floor- and ceiling-functions similar to [36]; in analogy, exception conditions like overflow and underflow are defined in terms of numbers.

Based on the formalization of the IEEE standard, we have verified a theory of IEEE-rounding. One central definition of this theory is an equivalence relation which separates the real numbers into equivalence classes that round to the same representable floating point number. This is essentially a mathematical formulation of guard- and sticky-bits. We have verified several theorems about this equivalence relation, e.g., when numbers are multiplied by powers of 2; this captures the numerical effect of shifters in hardware.

5.2 Combinational FPUs

As a basis for the construction and verification of the FPU hardware, we have verified a generic library of arithmetic circuits like adders, multipliers, leading-zeros counters, etc. [4]. These circuits can be instantiated to arbitrary width as they are needed for the building of the actual floating-point circuits. The inputs and outputs of these circuits are tuples of bits and bitvectors, but the correctness statements for the circuits cover the mathematical intention of the circuits. For left-shifters, e.g., we have proved that the result interpreted as binary number, equals the input multiplied by the power of the shift-amount, also both interpreted as binary numbers: $\langle outp \rangle = \langle inp \rangle \times 2^{\langle sha \rangle}$, where $\langle \cdot \rangle$ converts a bitvector to its represented natural number. Here we assume that no bits are shifted out, which itself is captured as a numerical precondition, namely $\langle inp \rangle < 2^{output\_width - \langle sha \rangle}$.

Based on those generic arithmetic circuits, we can construct and verify the actual floating point circuits. The circuits are composed from the generic circuits, combined with glue-logic such as single gates or multiplexers. The mathematical function of these composed circuits can then be derived from rewriting with the mathematical specifications of the sub-circuits, plus some reasoning about the glue-logic. In that way, we describe the floating-point circuits on the gate-level, but raise the verification very quickly to the numerical level, which is easier to handle in the theorem prover.

Once the mathematical functionality of a floating-point circuit has been established, we can use the theory of rounding to derive correctness of this circuit (or a composition of multiple such circuits) according to the formalization of the IEEE standard. For example, for the verification of the floating-point rounder, we separately verify the normalization shifter, the sticky-bit computation, and the actual rounding decision with incrementer against the respective numerical definitions from the rounding theory. Using the correctness of these blocks plus the theorems from the rounding theory, we can then combine them to derive the correctness of the complete rounding unit.

Similarly, we can derive correctness of the floating-point addition circuit (including, e.g., alignment shift), and then combine it with the rounding unit to obtain correctness of the whole floating-point computation. The correctness statement of the addition circuit states that a number is computed which is equivalent to the infinitely precise addition result with respect to the equivalence relation mentioned in section 5.1. By the theorems about this equivalence relation, and the correctness of the rounding unit, it follows that the combination of addition circuit and rounding unit produces the correct overall result.

The combinational multiplicative FPU is described as a function[7] md_{comb}; similar functions are defined for the additive and misc FPUs. For later pipelining, this function is partitioned into sub-functions corresponding to the circuits of the different pipeline stages, for example, functions unp for the unpacker, $mul1$ for the first multiplier stage, etc. For multiplications on non-special operands, $md_{comb} = rd2 \circ rd1 \circ mul2 \circ mul1 \circ unp$ holds (function composition from right to left), i.e., multiplication can be performed by consecutive execution of the sub-functions. The equality is proved trivially in PVS from the construction of md_{comb} and the sub-functions. Division is implemented with Newton-Raphson iteration. This implies that a division instruction has to iterate through the multiplier (and some other hardware) several times. In the definition and verification of the combinational FPU md_{comb} the iteration is unrolled and thus this hardware is replicated multiple times. Hence, $md_{comb} = rd2 \circ rd1 \circ selfd \circ (mul2 \circ mul1)^i \circ unp$ holds, where i is the number of iterations depending on the precision. The pipelined FPU md_{pipe} described below comprises only one multiplier instance, and division instructions iterate through this multiplier multiple times. The verification of the pipelined version verifies that the manifold replication of the $mul1$ and $mul2$ functions in md_{comb} matches the iteration through the same multiplier in md_{pipe}.

The theory of rounding as well as the FPU design are based on [37]. A detailed description of the formulation and verification in PVS can be found in [3, 22, 24].

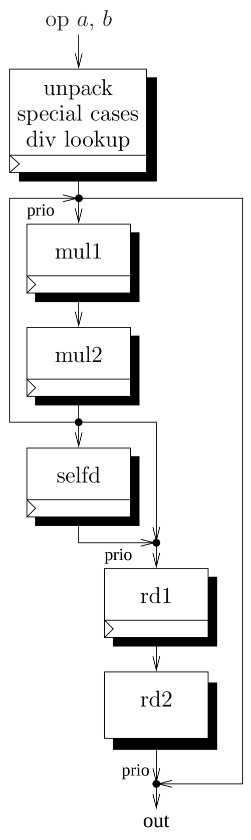


Fig. 6. Pipeline structure of the multiplicative FPU

5.3 Pipelining

After having verified the combinational FPU against its IEEE specification, the FPU is pipelined. Exemplarily, figure 6 depicts the pipeline structure of the multiplicative FPU. The first pipeline stage does operand unpacking, special case detection (NaN, ∞, etc.), and contains a lookup table for the initial approximation for divisions. If a special case is detected, the result is computed here and is bypassed to the output of the FPU. The next two pipeline stages comprise a pipelined Karatsuba-Ofman multiplier [27]. Divisions iterate through this multiplier up to 8 times, depending on the target precision. The *selfd* stage is used for divisions only, multiplications skip this stage. Finally, the results are rounded by the two-stage rounder.

Each instruction flows through the pipeline until it cannot flow further due to a structural hazard, i.e., another instruction in the pipeline requires the same stage. For example, if two divisions are iterating simultaneously through the two multiplier stages, both stages are occupied and thus a multiplication in the unpack stage has to be stalled. If multiple paths lead to the same stage, arbitration is performed by static prioritization of the longer paths; this ensures liveness.

As described in section 3.4, the execution units in the VAMP processor do not need to finish the execution of instruction in the same order as they are started. The FPU execution units exploit this in several ways: for example, instructions on special operands like ∞ are directly bypassed to the output, and hence can overtake other instructions still in the pipeline. If a single division is iterating through the multiplier, only one multiplier stage is occupied at a time; hence, a multiplication instruction can simultaneously flow through the unoccupied multiplier stages, and then finish execution while the division is still iterating. Another example are two division instructions iterating through the two multiplier stages in an intertwined way: a single precision instructions needs fewer iterations and can hence overtake a double precision even if started later.

The pipelined FPUs are composed of the combinational pipeline stages described in section 5.2, e.g., $mul1$ and $mul2$. In addition, the pipelined FPUs contain multiplexers between converging pipeline paths, registers between the stages, and the pipelining control logic; these hardware components do not exist in the combinational FPUs. The pipelined FPUs are defined as PVS functions which represent the next-state function of the pipeline (cf. section 8). For example, the pipelined multiplicative FPU is defined as a function md_{pipe} which implements the pipeline depicted in figure 6. Note that the VAMP hardware only instantiates the pipelined version of the FPUs—the combinational versions like md_{comb} are only used as intermediate definitions for verifying the numerical correctness of the FPUs.

[7] Combinational hardware blocks are modelled as functions in PVS, mapping input bits and bitvectors to output bits and bitvectors; cf. Section 8.

We have to verify the execution unit conditions from section 3.4. Data correctness of the pipelined FPUs is defined by the combinational FPU functions which have been verified against the IEEE standard. This means that the combinational FPU function, e.g. md_{comb}, acts as datapath specification function (called f in 3.4, iii) for the pipelined version md_{pipe}.

The properties i), ii), and iv) are independent of the datapath. For property iii) we have to prove that the instruction takes the correct path through the pipeline, and that hence the correct stage functions are applied to the instruction data in the correct order and number. This in particular involves verifying that iterating over the same multiplier instance i times in md_{pipe} is equivalent to the i-fold replication of the multiplier in md_{comb}. Note that this also involves verification of any resource conflicts at the multiplier, and also correct data routing through the multiplexers in the pipeline.

In a sense, we regard the instruction as a token flowing through the pipeline and collecting the function of each pipeline stage on the intermediate data. For this purpose, the actual function of each stage can be left uninterpreted, i.e., we do not care that the $rd1$ and $rd2$ stage actually comprise a floating-point rounder—we only need to check that the instruction flows through these stages.

We have tried to prove the correctness of the pipelining by theorem-proving. However, the invariants from the correctness criterion in section 3.4 are not inductive and have to be strengthened by auxiliary invariants. Due to the out-of-order behaviour, finding enough auxiliary invariants proved to be extremely hard. Instead, we came up with a method of combining both model-checking and theorem-proving for this verification task. Since we have designed the pipeline control separately from the datapath logic, we have a structural separation of the whole pipelined FPU into control and datapath for free. This enables us to use a model-checker in order to prove lemmas about the control, which then are combined with the datapath using theorem proving. Specifically, we have proved i), ii) and iv) from section 3.4 completely using the PVS-built-in model-checker.

For property iii) we have verified liveness of all individual pipeline stages, together with lemmas about the choice of the arbiters in different situations. These lemmas are then used during theorem-proving to "push" instructions through the datapath. For example, if an instruction is in the first pipeline stage, then by a model-checked lemma, the pipeline stage $mul1$ will eventually be clocked with its input multiplexer selecting the unpack stage. At this time, the instruction makes progress into the multiplier. In this way, we can push the whole instruction through the pipeline stage by stage. At every stage, the theorem prover can easily collect the pipeline stage function into the intermediate result of that instruction. At the end, the derived composition of stage functions reflects the path that the instruction has taken through the pipeline; this composition of stage functions is then compared to the datapath specification function, e.g., md_{comb}. If, for example, a division would erroneously skip the *selfd* stage, the comparison of the composed datapath functions from md_{pipe} to md_{comb} would fail.

The built-in PVS model-checker is a μ-calculus model-checker. As such, the lemmas proved by the model-checker are statements about the control of the FPU pipeline in μ-calculus. The properties i) – iv) from section 3.4, however, are not μ-calculus properties, but use universal and existential quantifiers to argue about sequences of states. For theorem-proving, this form of temporal statements seems more convenient. In order to efficiently use model-checked properties in the theorem-proving domain, we have verified theorems relating μ-calculus properties to their respective universal/existential quantifier forms. For example, we have verified for arbitrary state machines M and properties p that the μ-calculus formula of $\mathbf{AG}p$ holds if and only if for all valid state sequences S and all times t, the property $p(S(t))$ holds. These theorems have first been proved in [16] and are well known. However, they have not been verified using formal methods before, which is necessary to transform between μ-calculus and the quantifier form in a formally safe way. Using these theorems, we can use the model-checked properties in the theorem-proving domain in order to prove the necessary pipeline correctness conditions.

We remark that, due to the use of the model-checker, we have formal proofs but no complete paper and pencil proofs for the correctness and liveness of the floating point control. Details on the construction and verification of the pipelines, as well as the theorems about μ-calculus, can be found in [23, 24].

5.4 Number of operands

At first sight, floating point operations have two operands and one result. However, rounding mode (stored in a special purpose register RM) and interrupt masks (stored in SR) are two further operands of every floating point operation. Moreover, there is aliasing in connection with the addressing of the floating point registers: each single precision floating point register can be accessed by single precision operations as well as by double precision operations. The ISA does not preclude the construction of a double precision operand by two writes with single precision to the upper and lower half of a double precision register. *It can be necessary to forward these two results from separate places*. This is easily realized by treating the upper half and the lower half of double precision operands as separate operands. Thus, the reservation stations for the floating point units have a total of 6 operands.

6 Memory interface

The formal proof that the VAMP with a cache memory interface implementation simulates the specification machine is decomposed into two independent parts. First, we replace the cache memory interface by an idealized memory M_I which is basically a dual-ported memory. The VAMP with this memory M_I is verified against its specification. As a second step,

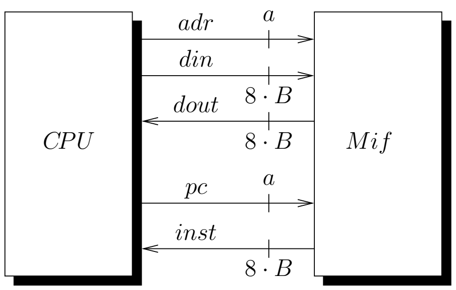


Fig. 7. A memory interface Mif

we only focus on the cache memory interface implementation; we prove that this implementation fulfills the specification given by M_I. Thus, M_I is used as a definition in the correctness proof of the whole CPU, while it serves as a specification against which we verify the cache memory interface implementation. Putting these two proofs together yields a fully formally verified CPU with a cache memory interface.

The interface of M_I is defined such that the real cache is a refinement of M_I, i.e., it behaves according to the interface signal definitions, but M_I leaves latencies and stall cycles unspecified.

6.1 *Specification*

Let $B \geq 1$ be the width of the memory interface data in bytes, and let $a \geq 1$ be the number of bits used in addressing the memory interface. Thus, our memory contains $2^a \cdot B$ bytes and B bytes can be read or written in a single memory access.

A memory interface is basically a memory with data- and instruction access ports as depicted in figure 7. In addition to the address- and data buses in the figure, there are several control signals. As inputs of the memory interface, we have $init$ as a power-up signal, mr and mw to select read- and write accesses on the data access port, respectively, mwb with B byte write signals for write accesses, and imr for read accesses on the instruction port. The control outputs are $ibusy$ and $dbusy$ indicating a pending access on the instruction- or data port, respectively. Note that the memory interface is addressed by word addresses pc and adr and always returns full data words on reads; for writes, only the bytes selected by mwb are written. In particular, this means that there is no misaligned access support in the memory interface. We call the CPU output to the memory interface valid if

i) $init$ is initially active: $init^0$
ii) the read- and write signals on the data port are never raised simultaneously: $\forall t : \neg mw^t \vee \neg mr^t$
iii) a data access is stalled by an active $dbusy$:

$$\forall t \forall b \in \{adr, din, mw, mr, mwb\} : \\ (mr^t \vee mw^t) \wedge dbusy^t \implies b^{t+1} = b^t$$

iv) an instruction access is stalled by an active $ibusy$:

$$\forall t : imr^t \wedge ibusy^t \implies \\ \{pc, imr\}^{t+1} = \{pc, imr\}^t$$

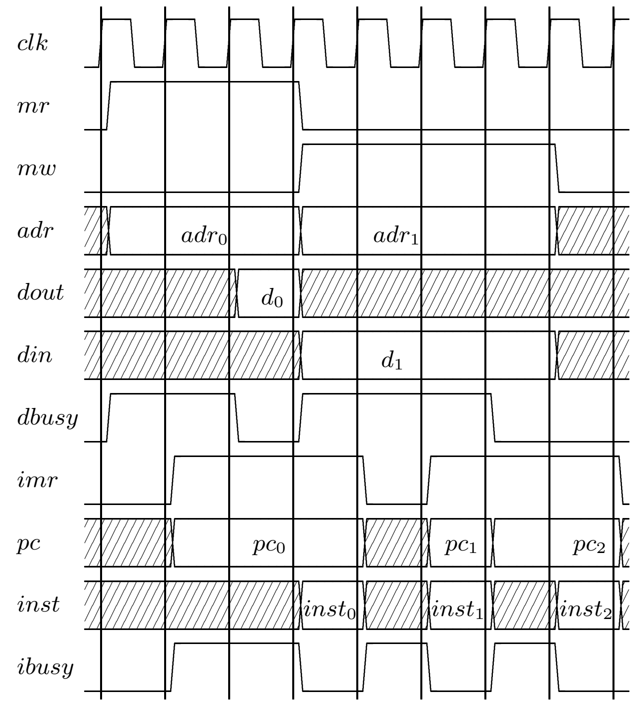


Fig. 8. Timing of the memory interface

The timing in our memory interface is simple. The CPU starts a data request in a cycle t by raising mr^t for a read or mw^t for a write. The address of the request is adr^t. In case of a write, din^t holds the data to be written and mwb^t contains the byte enables for the single bytes in din^t. All these signals keep their value until in a cycle $t' \geq t$, $dbusy^{t'}$ is lowered. In case of a read access, the data returned on $dout^{t'}$ is the requested data.

Similarly, an instruction read request is started in a cycle t by an active imr^t. The address pc^t remains stable until in a cycle $t' \geq t$, $ibusy^{t'}$ is lowered. The data output $inst^{t'}$ is the requested instruction data. Instruction and data requests may be arbitrarily interleaved. This is illustrated in figure 8.

We call a memory interface with valid input from a CPU correct if it is both live and consistent. Liveness means that any access to the memory interface eventually terminates, and consistency means that any read access yields the expected data. The following definitions formalize these concepts. Note that for any *word* address a, a byte $b < B$ in the data word addressed by a has the *byte* address $B \cdot a + b$ since we consider little endian memory organization.

Let $init\_mem$ be the initial memory content of a memory interface. We now model the visible memory content of the memory interface. The memory content is only updated at the end of a write access, and then only the bytes selected by mwb in the address given by adr. Thus, we define the memory content M_I in cycle $t \in \mathbb{N}$ recursively as follows:

$$M_I^0 := init\_mem$$
$$M_I^{t+1}[B \cdot a + b] := \begin{cases} din^t[b] & \text{if } a = adr^t \wedge mw^t \wedge \\ & \quad mwb_b^t \wedge \neg dbusy^t \\ M_I^t[B \cdot a + b] & \text{else} \end{cases}$$

We formally call a memory interface correct if the following conditions hold for any cycle t and any byte $b < B$:

i) data cache consistency:

$$mr^t \wedge \neg dbusy^t \implies dout^t[b] = M_I^t[B \cdot adr^t + b]$$

ii) instruction cache consistency:

$$imr^t \wedge \neg ibusy^t \implies inst^t[b] = M_I^t[B \cdot pc^t + b]$$

iii) data cache liveness: $\exists t' \geq t : \neg dbusy^{t'}$

iv) instruction cache liveness: $\exists t' \geq t : \neg ibusy^{t'}$

Thus, on concurrent read- and write accesses to the same address in the memory interface, there are two possible outcomes: Either the instruction read access terminates strictly after the write access and returns the memory interface content *after* the execution of the data write access, or it terminates in the same cycle or before the write and returns the *old* memory content before termination of the write access. Both scenarios are equally possible with our correctness criterion.

For the remaining section, we only consider a CPU with such an abstract memory interface M_I instantiated with parameters $a = 29$ and $B = 8$, i.e., the VAMP accesses 4 GB of memory in double words. All the arguments and lemmas only use the definition of M_I, but not the complex cache memory interface implementation. The actual implementation of the VAMP memory unit accessing an abstract memory interface Mif is depicted in figure 9. Internally, it has two pipeline stages. The first stage does address and control signal computations. The second stage performs the actual access to the data port of the memory interface via signals adr, din, and $dout$. Instructions are fetched from the instruction port of the memory interface via signals pc and $inst$. Note that we can only access the VAMP memory via the memory interface, i.e., there are *no* special instructions that directly access the physical memory.

6.2 Loads and stores with variable operand width

The formal specification of the semantics of the memory instructions is based on the definitions in [37, Chap. 3]. Memory accesses are characterized by their effective address ea and their width in bytes $d \in \{1, 2, 4, 8\}$. The access is aligned if $ea \bmod d = 0$. Effective addresses ea define a double word address $da(ea) = \lfloor ea/8 \rfloor$ and a byte address $ba(ea) = ea \bmod 8$. A simple 'alignment lemma' states that for aligned accesses, the memory operand $M[ea{+}d{-}1 : ea]$ equals bytes $[ba(ea) + d - 1 : ba(ea)]$ of the double word addressed by $da(ea)$ at the memory interface.[8] Details can be found in [37, p. 78–88]. For misaligned memory addresses, a corresponding interrupt is triggered and the VAMP does not access the memory at all.

Circuits called *shift4load* and *shift4store* are used in order to ensure that data is loaded and stored correctly. These circuits are shown in figure 9. "Shift for store" denotes shifting the data, say the halfword which is to be stored, into the correct position of a double-word before it is sent to the 64-bit wide memory interface. Similarly, "shift for load" denotes extraction of the requested portion (say halfword) of the 64-bit delivered from the memory interface. Also, sign-extension is done during "shift for load" for signed byte- and halfword-loads. Additionally, a circuit called *gen_bw* is used in order to generate the byte enable signals for a write access to the memory interface that features 64 bit wide data. Shift for store and load are implemented by means of two simplified shifters with some control logic [37]. Since $M_I^t = M_S^{sI(M,t)}$ is part of the correctness invariant, it is easy to show that an aligned memory access in the implementation yields the same result as in the specification.

Note that the remaining correctness criteria of a functional unit as outlined in section 3.4 are easy to show for the memory unit since memory instructions are executed in order in contrast to the floating point unit. Thus, the techniques introduced in section 5 did not have to be employed for the pipelined memory unit. Details of the verification of our memory unit are reported in [6, Chap. 4.4.2].

6.3 Self-modifying code

We consider self-modifying code independent of the implementation of the memory interface, i.e., the following arguments are based exclusively on the memory interface layer, and not on the cache memory interface implementation. In terms of scheduling functions, the problem of self-modifying code can be summarized as follows:

The correctness invariant guarantees $M_I^t = M_S^{sI(M,t)}$. Let PC_I be the PC actually used in the VAMP implementation to fetch an instruction according to figure 3; for the specification, $PC_S = DPC_S$ holds. From $corr?(t)$ and the implementation of the PC used in instruction fetch, we can conclude $PC_I^t = PC_S^{sI(fetch,t)}$. We then have to guarantee that in case of a fetch in cycle t, the following equation holds:

$$M_I^t[PC_I^t + 3 : PC_I^t] = M_S^{sI(fetch,t)}[PC_I^t + 3 : PC_I^t]$$

Thus, in order for the fetch to be correct, we have to guarantee that M_I was not updated on address PC_I^t by any instruction between $sI(M,t)$ and $i := sI(fetch,t)$. As an additional precondition for the correctness of the implementation, we therefore demand that in case an instruction I_i is fetched from a memory location PC_S^i, there is a special *sync*-instruction between the last write to PC_S^i and the fetch from PC_S^i.[9] From the view of the programmer, this *sync* instruction behaves just like a *nop*. For the pipeline implementation, however, the effect of the *sync* instruction is as follows:

8 Note that this specifies little endian memory organization.

9 This implies the correspondency condition from [42].

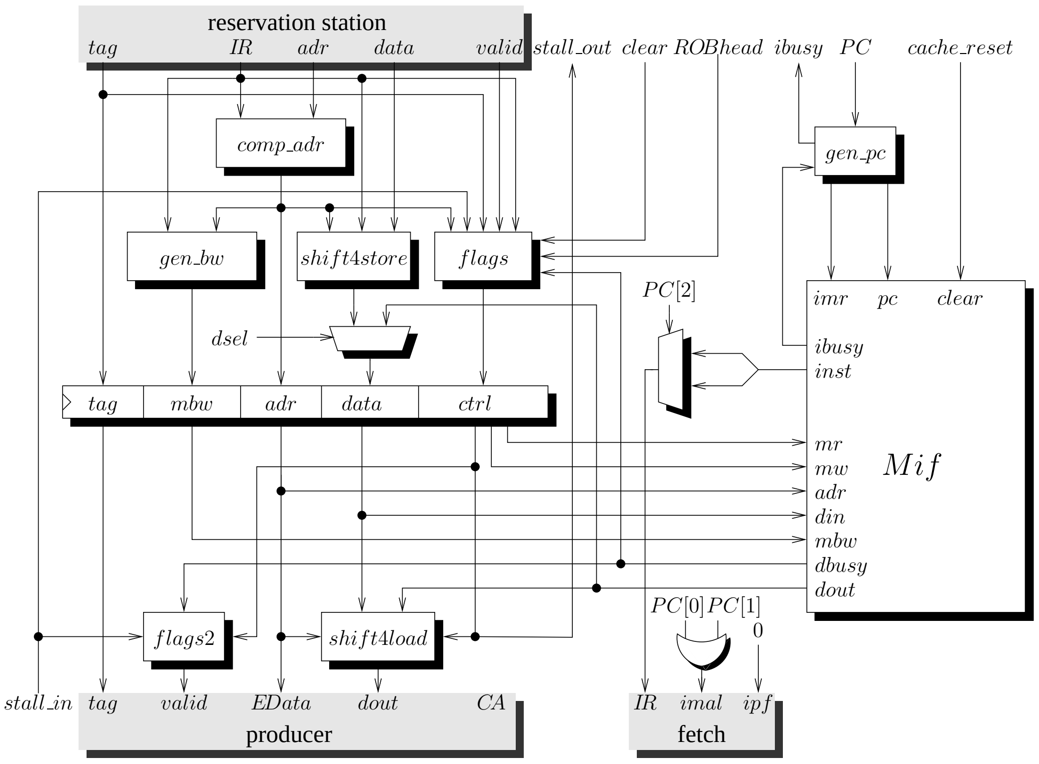


Fig. 9. Data paths of the VAMP memory unit

i) The *sync* instruction is stalled in decode until all previously issued instructions have left the pipeline. In particular, this means that all pending writes in the memory unit have completed.
ii) No instruction after the *sync* instruction is fetched while it is stalled.

Hence, the overall effect of a *sync* instruction I_i is that the first instruction after I_i is only fetched after *all* instructions before I_i have left the pipeline. In particular, all memory instructions before I_i have already been completed when the new instruction is fetched. The memory interface itself is *not* affected by the *sync* instruction; hence, caches are not flushed. However, the memory interface is still correct according to our definition; therefore, we do not have to argue about the consistency of caches since the cache memory interface implements a correct memory interface which is the only property we care about in our arguments.

In the VAMP architecture, this *sync* instruction is implemented without additional hardware by a special move from the *IEEEf* register to $R0$ as mentioned in section 3.5. We have formally verified that this use of the *sync* instruction suffices to show the correctness of the implementation in case of self-modifying code [6, Chap. 4.4.3]. Note that typical compilers trivially fulfill the $sync$ criterion by not generating self-modifying code.

6.4 *Preciseness of interrupts*

In this section, we give the arguments for preciseness of interrupts with respect to the memory. In formal terms, given $JISR^t$ and $corr?(t)$, we want to conclude the memory part of $corr\_i?(t+1)$, i.e., $M_I^{t+1} = M_S^{sI(inst,t+1)}$. Note that $corr?(t)$ guarantees $M_I^t = M_S^{sI(M,t)}$ where $sI(M,t)$ can be both smaller and greater than $sI(inst,t+1)$. Therefore, it is sufficient to guarantee two things: i) in the interrupt cycle t, no store accesses the memory, i.e., $M_I^t = M_I^{t+1}$, and ii) there is no write after an instruction that caused an interrupt accesses the memory, while all writes prior to the interrupted instruction are actually executed, i.e., $M_S^{sI(M,t)} = M_S^{sI(inst,t+1)}$.

Preciseness of interrupts for the memory unit is accomplished by stalling writes in the memory unit before their memory access until there is no older instruction left in the VAMP's pipeline. Showing that this construction is sufficient for precise interrupts with respect to the memory is not too difficult [6, Chap. 4.5.1]. Note that we do not have to argue about the complex cache memory interface implementation, but only about the memory interface layer.

7 Cache memory interface

After proving the VAMP implementation with an abstract memory interface correct against its specification in the last section, we now give an implementation of a cache memory

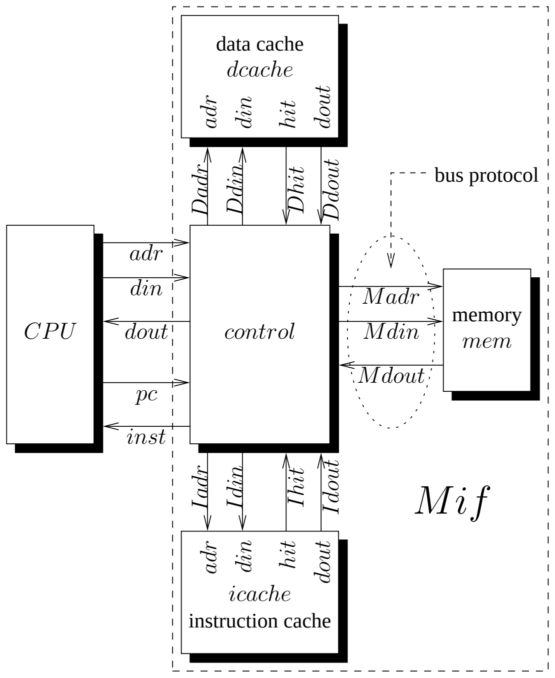


Fig. 10. Cache memory interface

interface together with a correctness proof against its specification, a memory interface.

7.1 Implementation

The memory interface is implemented with split caches connected to a single main memory as depicted in figure 10. We use a write-back policy for the data cache, i.e., on a write access of the CPU, the data cache is updated and the corresponding data is marked as dirty. Thus, a slow access to the main memory is avoided. If dirty data is to be evicted from the cache, it is written back to the main memory in order to ensure data consistency.

The protocol used to keep the caches coherent works as follows: If a cache signals a hit on a CPU access, the data is read directly from the cache or written to it, depending on the type of the CPU access. This allows for memory accesses that take only one cycle to complete. If, on the other hand, the cache signals a miss, the corresponding data has to be loaded into the cache. The control first examines the other cache in order to find out if it holds the required data. In this case, the data in the other cache is invalidated. If the data to be invalidated is dirty, this requires an additional write back to the main memory.

This consistency protocol guarantees *exclusiveness*, i.e., for any address, at most one of the two caches signals a hit. In this way, we ensure that on a hit of the instruction cache, the data cache does not contain newer data.

The instruction and data caches are implemented as k-way sectored set-associative caches using an LRU replacement policy. Cache sectors consist of 4 double words since the bus protocol supports bursts of length 4. However, in the overall correctness proof of the cache memory interface, we do not want to argue about unnecessary details like the associativity or replacement policy of the underlying caches. Therefore, the proof that the cache memory interface implements a memory interface is decomposed into two cleanly separated parts: First, we introduce the abstract notion of *consistent caches* and show the cache memory interface with such black-boxed consistent caches to be correct. As a second step, we only focus on the cache level and show different implementations, among them k-way set-associative and fully associative caches, to implement these consistent caches [6, Chap. 2]. On the one hand, this decomposition facilitates arguments considerably, while on the other hand, it actually makes it easy to instantiate and verify a cache memory interface with different associativity for instruction and data cache as we did for the VAMP.

7.2 Typical lemmas

The inductive invariant used in order to show consistency of split caches as described above consists of *three* parts. Two of these parts are obvious: if the data or instruction cache, respectively, signals a *hit*, then its output data equals the specified memory content. However, an invariant consisting only of these two claims is *not* inductive since caches are reloaded from the main memory. Therefore, we need a third part of our invariant stating the consistency of data in the main memory. Thus, we also claim that on a clean hit or a miss in cycle t on address $Dadr^t$ in the data cache, the main memory mem on this address $Dadr^t$ contains the specified memory content M_I. Note that on a clean hit in the data cache, we thus claim data consistency in both the data cache and the main memory. Formally, we have the following claim:

$$\begin{aligned}
&Ihit^t \implies Idout^t[b] = M_I^t[8 \cdot Iadr^t + b] \wedge \\
&Dhit^t \implies Ddout^t[b] = M_I^t[8 \cdot Dadr^t + b] \wedge \\
&\neg(Dhit^t \wedge dirty^t) \implies \\
&\quad mem^t[8 \cdot Dadr^t + b] = M_I^t[8 \cdot Dadr^t + b].
\end{aligned}$$

This invariant is strong enough to show that the cache memory interface implements a memory interface according to section 6.1 since the data word returned to the CPU on a read access is just the cache output in case of a hit, or the data written to the cache during cache load in case of a miss. Note that the invariant relies on the exclusiveness property of the protocol, which has to be verified as part of the proof of the invariant. Details on the verification are reported in [6, Chap. 3].

7.3 Bus protocol

The main memory is accessed via a bus protocol featuring bursts. The bus protocol signals ready data by raising $brdy$ one cycle in advance. A sample timing of a 4-burst write is depicted in figure 11. Note that the data input din one cycle *after* $brdy$ is written to the main memory and that the end of the access is signaled by $\neg reqp \wedge brdy$.

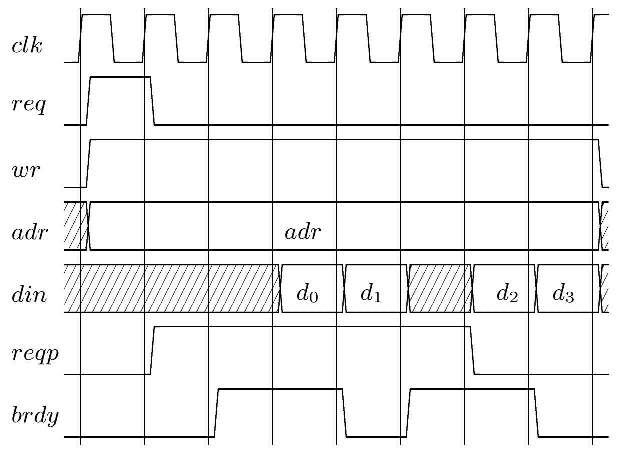


Fig. 11. 4-burst write timing diagram

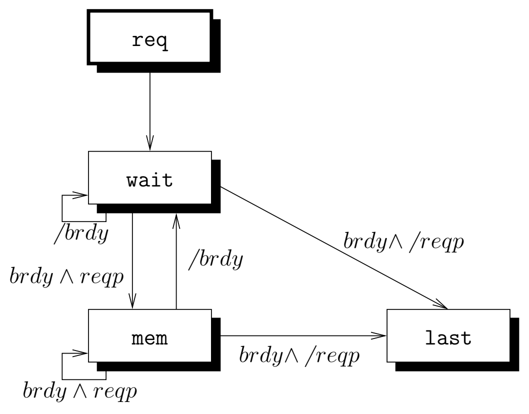


Fig. 12. Burst control FSD

As part of our correctness proof for the memory interface, we have formalized this bus protocol and proved that an automaton[10] according to figure 12 implements this protocol correctly by means of theorem proving [6, Chap. 3.1]. The main invariant for this proof is the following: in the cycle of the i-th memory access of the burst, i.e., after the i-th $brdy$, the automaton is in state `mem` for the i-th time. In the cycle of the last memory access, the automaton is in state `last`.

8 Modeling hardware, translation, and synthesis

In this section we describe how the PVS language is used to model the VAMP hardware. We also describe how the PVS hardware specifications are translated into Verilog HDL, and provide some results of the implementation of the VAMP on a Xilinx FPGA.

[10] Note that this bus control FSD is only a part of the FSD for the cache memory interface.

8.1 Modeling hardware

The basic data types used to model hardware are the PVS data types *bit* and *bvec[n]* for bits and bitvectors of length n, respectively [12]. In addition to the standard boolean operators on bits and bitvectors, we use if statements for modeling multiplexers, λ-expressions to compose bitvectors of single bits, and let-expressions to eliminate common sub-terms. Combinational hardware modules are modeled as PVS functions mapping the input bits and bitvectors to the output bits and bitvectors of the module. Inputs and outputs can both be nested records of bits and bitvectors. Function calls model the instantiation of hardware sub-modules.

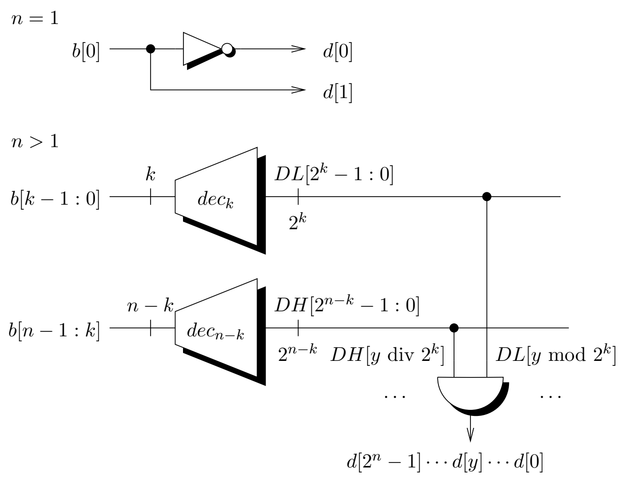
Recursive functions are used in order to model hardware with recursive structure. This is an important feature since it allows for inductive reasoning on recursively defined circuits. In order to define parameterized designs (e.g., adders of arbitrary width), we use integer parameters in function definitions. Some simple expressions on those integers are supported—however, those expressions are only valid if they can be evaluated to constants during the translation process, and hence need not be implemented in hardware.

Figure 13 exemplarily depicts the standard construction of a combinational binary decoder circuit [2] in PVS. The decoder computes for n-bit inputs an output of 2^n bits; if the input interpreted as a binary number has value i then exactly bit i of the output will be one. This construction also shows the restricted use of integer parameters in the hardware designs: the integers are only used to define the recursion. If the decoder is used with a fixed n, then by unrolling the recursion all integer expressions can be evaluated to constants, which leaves an unrolled gate-level description of the decoder circuit without any integer expressions. Figure 13 also shows the correctness statement for the decoder, which is proved by induction on n.

Clocked circuits are modeled as standard transition systems (S, δ) with state set S, and combinational next-state function $\delta : I \times S \rightarrow O \times S$ mapping inputs and current state to outputs and next state. The state set S is a nested record of bits and bitvectors, and δ is defined as a combinational function as described above. Composition of multiple such clocked circuits follows the standard composition of transition systems from automaton theory: take the Cartesian product of the state sets and combine the transition functions accordingly. Essentially, this means that all registers of a complex clocked circuit are lifted to the top-level in the description of that circuit—admittedly, this is not very convenient for designing hardware, but for us it eases the verification of large composed systems.

Based on the definition (S, δ) of a clocked circuit, it is straightforward to recursively define the sequence of states that the circuit traverses under a given sequence of inputs. The signals $stall_{in}^t$, $valid_{out}^t, \ldots$ as they are used in section 3.4, for example, are formally defined as such sequences in PVS.

The described approach of modelling clocked circuits assumes a fully synchronous design, i.e., all registers share the



```
decoder(n: posnat, b: bvec[n]): Recursive bvec[2^n] =
  If n = 1 Then
    fill[1](b(0)) o fill[1](NOT b(0)) % gates for n=1
  Else
    Let
      k = ceiling(n/2),
      dec_lo = decoder(k, b^(k-1, 0)),  % sub-circuit
      dec_hi = decoder(n-k, b^(n-1, k)) % sub-circuit
    In
      Lambda (y: below(2^n)):
        Let
          i = mod(y, 2^k), j = div(y, 2^k)
        In
          dec_lo(i) AND dec_hi(j)       % AND gates
  Endif
Measure n;

decoder_correct: Lemma
  Forall(n: posnat, b: bvec[n], y: below(2^n)):
    (decoder(n,b))(y) IFF bv2nat(b) = y;
```

In the definition of the decoder `fill[1](b)` converts a single bit b to a bitvector of length one, `o` denotes vector concatenation, `ceiling(a)` means $\lceil a \rceil$, `b(i)` returns bit i of bitvector b, `b^(i,j)` extracts the bits i to j from bitvector b, and `bv2nat(b)` returns the value of b interpreted as a binary number.

Fig. 13. Construction and correctness statement of a decoder

same clock and RAM blocks for register files or caches are also updated synchronously to this clock; thus, concerning timing, they can be treated like registers. In such a fully synchronous design, valid data is needed only at the clock edges with certain setup- and hold-times. In particular, we fully ignore any *glitches*, i.e., instabilities in signals during a clock period, since these glitches do not influence the fully synchronous designs. This approach therefore cannot cope with designs that need stable signals during the whole clock cycle, i.e., where glitches must not occur. For example, this is the case for asynchronous EDO-RAM chips that need stable addresses for a fixed amount of time. Since we use synchronous RAM chips, our proofs guarantee the correctness of the design regardless of any occurring glitches.

8.2 Translation

We have written a translation tool called `pvs2hdl` [7] that translates PVS hardware descriptions into a standard hardware description language. We have chosen Verilog HDL, but any other language like VHDL would have worked as well. The translation of PVS hardware descriptions to Verilog works as follows: starting from a user defined top level function f, a Verilog module is generated for each PVS function in the sub-function call tree of f. If the PVS function has integer parameters to represent parameterized circuits, a Verilog module for each different occurring parameterization is generated. If the PVS function is recursive, the recursion is unrolled.

The translation of standard boolean operators to Verilog is trivial; λ-expressions are translated separately for each bit, which are then concatenated to yield the desired bitvector. A $let\ x = expr$ construct is translated by first translating $expr$, then assigning a wire-name to the translated $expr$, and using this wire-name in Verilog where in PVS the alias x is referenced. Thereby, the hardware for expression $expr$ is only generated once, although the alias x may be used several times. PVS function calls are translated to module instantiations in Verilog. All expressions in the domain of integers must be completely evaluated to constants by `pvs2hdl`; if the tool fails to evaluate an expression, it aborts the translation with an error message. Nested records of bitvectors are flattened into individual bitvectors by `pvs2hdl` since records are not supported by Verilog.

In order to translate a clocked circuit (S, δ) to Verilog, the tool generates an additional top level module M. This module contains a register of type S (flattened for records, again), and an instantiation of the translated δ-module, which is connected to the register in the obvious way. Additional inputs and outputs of δ are connected to respective primary inputs and outputs of M.

While we support the translation of a certain subset of the PVS language to Verilog, there are numerous PVS language elements not supported by `pvs2hdl`. For example, expressions in PVS may contain quantifiers ($\forall, \exists$) or higher-order logic statements. As these constructs have no gate-level equivalent, they are not supported by `pvs2hdl`, and are consequently not used in our VAMP hardware design. However, those language elements may yet be used in the PVS correctness specifications and proofs of the hardware designs.

8.3 Synthesis and results

We have successfully translated the PVS hardware description of the VAMP processor to Verilog HDL. The Verilog representation of the processor (including caches and floating point unit) has been synthesized, implemented, and tested on a Xilinx FPGA hosted on a PCI board. Some additional unverified hardware for controlling the VAMP processor and for accessing its memory from the host PC has also been synthesized on this FPGA [32]. The VAMP processor occupies

Part	Effort [years]	Lemmas	Proof steps
Tomasulo & ALU	2	521	14367
FPU	3	1046	25936
Memory Interface	2	566	24432
Putting together	1	445	23299
Total	8	2548	88034

Table 2. Verification effort

about 18000 slices of a Xilinx Virtex FPGA, which accounts for a gate count of 1.5 million gates as reported by the Xilinx tools. The design contains 9100 bits of registers (not counting memory and caches) and runs at 10 MHz.

We have ported the `gcc` and the GNU C library for the VAMP in order to execute test programs on the VAMP [35]. As it was to be expected from our verified design, we found no errors in the VAMP processor.

9 Verification effort

The formal verification of the VAMP microprocessor took about eight person-years; for the translation tool and synthesis on the FPGA, an additional person-year was required. Table 2 summarizes the verification effort for the different parts of the VAMP including the number of lemmas and interactive proof-steps in PVS. We are confident that the number of lemmas and proof steps could be significantly reduced by employing PVS strategies or integrating automated methods since we made only limited use of PVS strategies. Hence, the numbers are just intended to give an idea on the general complexity of the proofs and the high degree of interactivity in our proofs.

Note especially that in table 2, "Putting it all together" took a whole person-year for several reasons. First of all, the proof of the Tomasulo core from [28] was only generic and had to be applied to the VAMP architecture, especially the VAMP instruction set. Unfortunately, in spite of thorough planning on our part, the interfaces between the different parts did *not* match exactly. Thus, a considerable effort went into patching these interfaces. Additionally, self-modifying code and the *IEEEf*-extension to the Tomasulo algorithm had to be considered. Also, interrupt support and a memory unit still had to be added to the formally verified Tomasulo core. Last but not least, PVS does not scale too well for projects this large; typechecking of the VAMP alone takes already more than two hours on our fastest machine.

10 Summary and future work

To the best of our knowledge, we have reported for the first time the formal verification of i) a processor with the full DLX instruction set including load and store instructions for bytes, half words, words, and double words, ii) a processor with delayed branch, iii) a processor with maskable nested interrupts, iv) a processor with integrated floating point unit, v) a memory system with separate instruction and data cache. More importantly, all the above mentioned constructions and proofs are integrated into a single design and a single correctness proof. Thus, we can be sure that no oversimplifications have been made in any part of the design. PVS ensures that there are no proof gaps left.

The VAMP design is synthesized[11] and implemented on an FPGA. The complexity of the design is comparable to industrial controllers with FPUs. To the best of our knowledge, VAMP is by far the most complex processor formally verified so far.

We see several directions for further work in the near future. i) Adding a store buffer to the memory unit. ii) The treatment of a memory management unit with separate address translation units for data and instructions. iii) Proving formally that a machine with memory management unit and appropriate page fault handlers as part of the operating system gives a single user program the view of a uniform virtual memory. This requires to argue about hardware and software simultaneously; a paper and pencil proof is given in [18]. iv) Completing the liveness proof of the VAMP with memory management unit. v) Redoing as much as possible of the present correctness proof with automatic methods. For such methods any subset of our lemmas lends itself as a benchmark suite with a very nice property: we know that it can be completed to the correctness proof of a full bit-level design.

As a part of the Verisoft[12] project funded by the German federal government, we are currently carrying the hierarchical proof approach several steps further like the CLI stack [5] project of Moore and Hunt. We add a formally verified compiler of a subset of the C language, a simple operating system, and page-fault handlers which are at least partially written in assembler; at the top, we run an email client, some signature software, and TCP/IP-protocol support as applications. The ambitious goal is to seamlessly integrate all these different layers into one single correctness statement that signing and sending or receiving and checking of the signature, respectively, are correct *on the VAMP architecture*. Several additional layers with appropriate black boxes between instruction set architecture level and application software have to be added. We intend to achieve these goals by mid 2007.

References

1. The VAMP project. Website. http://www-wjp.cs.uni-sb.de/projects/verification.
2. C. Berg. Formal verification of an IEEE floating point adder. Master's thesis, Saarland University, Germany, May 2001.
3. C. Berg and C. Jacobi. Formal verification of the VAMP floating point unit. In *Proc. 11th CHARME*, volume 2144 of *LNCS*, pages 325–339. Springer, 2001.
4. C. Berg, C. Jacobi, and D. Kröning. Formal verification of a basic circuits library. In *IASTED International Conference on Applied Informatics*. ACTA Press, 2001.

[11] The trivial proof of synthesizability.

[12] http://www.verisoft.de

5. W. R. Bevier, W. A. Hunt, J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5:411–428, Dec. 1989.
6. S. Beyer. *Putting it all together – Formal Verification of the VAMP (under appraisal, draft available at www-wjp.cs.uni-sb.de/publikationen/Be04.pdf)*. PhD thesis, Saarland University, Germany, 2004.
7. S. Beyer, C. Jacobi, D. Kröning, and D. Leinenbach. Correct hardware by synthesis from PVS. Internal Report, available at www-wjp.cs.uni-sb.de/publikationen/BJKL02.pdf, 2002.
8. S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. Paul. Instantiating uninterpreted functional units and memory system: functional verification of the VAMP. In D. Geist and E. Tronci, editors, *CHARME 2003*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.
9. B. Brock, W. A. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. Technical Report Technical Report 86, Computational Logic Inc., 1994.
10. B. C. Brock and W. A. Hunt. The DUAL-EVAL hardware description language and its use in the formal specification and verification of the FM9001 microprocessor. *Formal Methods in System Design*, 11:71–107, July 1997.
11. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In *CAV 94*, volume 818, pages 68–80, Standford, California, USA, 1994. Springer-Verlag.
12. R. W. Butler, P. S. Miner, M. K. Srivas, D. A. Greve, and S. P. Miller. A bitvectors library for PVS. Technical Report 110274, NASA Langley Research Center, Aug 1996.
13. Y.-A. Chen, E. M. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. W. O'Leary, and X. Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In *FMCAD*, volume 1166 of *LNCS*, pages 19–33. Springer, 1996.
14. W. Damm and A. Pnueli. Verifying out-of-order executions. In *Charme IFIP WG10.5*, pages 23–47, Montreal, Canada, 1997. Chapman & Hall.
15. A. P. Eiriksson. The formal design of 1M-gate ASICs. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98*, volume 1522 of *LNCS*, pages 49–63. Springer, 1998.
16. E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, volume 85 of *LNCS*. Springer, 1980.
17. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.
18. M. Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctnesss (under appraisal, draft available at www-wjp.cs.uni-sb.de/publikationen/Hil05.pdf)*. PhD thesis, Saarland University, Germany, 2005.
19. R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In *Computer-Aided Verification, CAV '99*, volume 1633, pages 47–59, Trento, Italy, 1999. Springer-Verlag.
20. W. A. Hunt and J. Sawada. Verifying the FM9801 microarchitecture. *IEEE Micro*, pages 47–55, May-June 1999.
21. Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.
22. C. Jacobi. A formally verified theory of IEEE rounding. Unpublished, available at www-wjp.cs.uni-sb.de/~cj/ieee-lib.ps, 2001.
23. C. Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In *CAV*, volume 2404 of *LNCS*. Springer, 2002.
24. C. Jacobi. *Formal Verificaton of a fully IEEE compliant floating point unit*. PhD thesis, Saarland University, Germany, 2002.
25. C. Jacobi and C. Berg. Formal verification of the VAMP floating point unit. In *Formal Methods in System Design*. Kluwer Academic Publishers, 2005. To appear.
26. C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner. Automatic formal verification of fused-multiply-add FPUs, (to appear). In *DATE*, 2005.
27. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7, 1963.
28. D. Kröning. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Germany, 2001.
29. D. Kröning, S. Müller, and W. Paul. Proving the correctness of pipelined micro-architectures. In *3ITG-/GI/GMM-Workshop Methoden und Beschreibungsprachen zur Modellierung und Verifikation von Schaltungen und System*, pages 89–98. VDE Verlag, 2000.
30. D. Kröning, S. Müller, and W. Paul. Proving the correctness of processors with delayed branch using delayed PCs. *Numbers, Information and Complexity*, pages 579–588, 2000.
31. D. Kröning and W. Paul. Automated pipeline design. In *Proc. of the 38th Design Automation Conference*, pages 810–815. ACM Press, 2001.
32. D. Leinenbach. Implementierung eines maschinell verifizierten Prozessors. Master's thesis, Saarland University, Germany, 2002.
33. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV 98*, volume 1427. Springer, June 1998.
34. K. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME 2001*, volume 2144 of *LNCS*. Springer, 2001.
35. C. Meyer. Entwicklung einer Laufzeitumgebung für den VAMP-Prozessor. Master's thesis, Saarland University, Germany, 2002.
36. P. S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Report TM-110167, NASA Langley Research Center, 1995.
37. S. M. Müller and W. J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.
38. J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, Q1, 1999.
39. S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *CADE 11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.
40. D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.
41. D. M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. In *FMCAD-00*, volume 1954 of *LNCS*. Springer, 2000.
42. J. Sawada and W. A. Hunt. Trace table based approach for pipelined microprocessor verification. In *CAV 97*, volume 1254 of *LNCS*. Springer, 1997.
43. J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In *CAV 98*, volume 1427 of *LNCS*. Springer, 1998.

44. J. Sawada and W. A. Hunt. Verification of the FM9801 microprocessor: An out-of-order microprocessor model with speculative execution, exceptions, and self-modifying code. *Formal Methods in Systems Design*, 20(2):187–222, March 2002.
45. X. Shen, Arvind, and L. Rudolph. CACHET: an adaptive cache coherence protocol for distributed shared-memory systems. In *International Conference on Supercomputing*, 1999.
46. A. Slobodova and K. Nagalla. Formal verification of floating point multiply add on Itanium processors. In *Workshop on Designing Correct Circuits*, Mar. 2004.
47. J. Stoy, X. Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *FME*, volume 2021 of *LNCS*. Springer, 2001.
48. M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In *CHARME*, volume 1703 of *LNCS*. Springer, 1999.
49. M. N. Velev and R. E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *DAC*. ACM, 2000.