Instantiating Uninterpreted Functional Units and Memory System: Functional Verification of the VAMP

Sven Beyer[1], Chris Jacobi[2]*, Daniel Kröning[3]**, Dirk Leinenbach[1]***, and Wolfgang J. Paul[1]

[1] Saarland University, Computer Science Department, 66123 Saarbrücken, Germany
{sbeyer,dirkl,wjp}@cs.uni-sb.de

[2] IBM Deutschland Entwicklung GmbH, Processor Dev. II, 71032 Böblingen, Germany
cjacobi@de.ibm.com

[3] Carnegie Mellon University, Computer Science, Pittburgh, PA
kroening@cs.cmu.edu

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

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

i) Processors with in-order scheduling, one or several pipelines including forwarding, stalling and interrupt mechanisms [3,13,28]. The verification of the very simple, non-pipelined FM9001 processor has been reported in [2]. Using the flushing method from [3] and uninterpreted functions for modeling execution units, superscalar processors with multicycle execution units, exceptions and branch prediction [28] 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 [15].

ii) Tomasulo schedulers with reorder buffers for the support of precise interrupts [5,8, 16,24]. Exploiting symmetries, McMillan [16] has shown the correctness of a powerful Tomasulo scheduler with a remarkable degree of automation. Using theorem proving, Sawada and Hunt [24] 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).

* 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’

D. Geist and E. Tronci (Eds.): CHARME 2003, LNCS 2860, pp. 51–65, 2003.

iii) Floating point units(FPU). The correctness of an important collection of floating point algorithms is shown in [21,22] using the theorem prover ACL2. Correctness proofs using a combination of theorem proving and model checking techniques for the FPUs of Pentium processors are claimed in [4,19]. As the verified unit is part of an industrial product not all details have been published. Based on the constructions and on the paper and pencil proofs in [18] a fully IEEE compatible FPU has been verified [1,11] (using mostly but not exclusively theorem proving).

iv) Caches. Multiple cache coherence protocols have been formally verified, e.g., [6,17, 25,26]. Paper and pencil proofs are extremely 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 [17] even allow for the verification of parameterized designs, i.e., cache coherence is shown for an arbitrary number of processors.

Simplifications, Abstractions, and Restrictions. Except for the work on floating point units, the cache coherence protocol in [6], and the FM9001 processor [2], *none* of the papers quoted above states that the verified design actually has been implemented. *All* results cited above except [1,2,6,11] use several simplifications and abstractions:

i) The realized instruction set is restricted: always included are the six instructions considered in [3]: load word, store word, jump, branch equal zero, three register ALU operations, ALU immediate operations. Five typical extra instructions are trap, return from exception, move to and from special registers, and sync [24]. The branch equal zero instruction is generalized in [28] 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 [27] 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 verification of the processors: e.g., Hosabettu et.al. [8] use tags from an infinite set. Obviously, this is not directly implementable in real hardware.

iv) The verification of the 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).

v) No verification of a memory unit with caches has been reported. Eiriksson [6] 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. Indeed, in [27] 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."

Results and Overview. 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 754 [9] compatible dual precision floating point unit with variable latency, as well as separate, coherent instruction and data caches. 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 [20]. All formal specifications and proofs are on our web site.[1] The hardware description was automatically extracted from PVS and translated into Verilog HDL by a tool sketched in section 7. Hardware with non verified rudimentary software is up and running on a Xilinx FPGA. The Verilog design can also be downloaded from our web site.

In section 2, we summarize the fixed point instruction set, its floating point extension, and the interrupt support realized. We give a micro-architectural overview with a focus on the memory system. Section 3 describes the correctness criterion, the main proof strategy, and the integration of the execution units into the Tomasulo core. Correctness criterion and proof strategy are based on scheduling functions [14,18] (similar to the stg-component of MAETTs [23]). The model of the execution unit is in a nontrivial way more general than previous models without complicating interactive proofs too much.

Section 4 presents a delayed branch mechanism, which is automatically constructed and proven correct by the methods for automatic pipeline construction from [15] and summarizes the specification of an interrupt mechanism for maskable nested precise interrupts and delayed PC from [18]. Section 5 deals with the integration of the floating point unit from [11] into our Tomasulo scheduler. Section 6 deals with loads and stores of double words, words, half words, and bytes at a 64 bit cache/memory interface. We also sketch correctness proofs of the implementation of a simple coherence protocol between data cache and instruction cache, as well as the implementation of a main memory bus protocol. Section 7 describes the implementation of the VAMP on a Xilinx FPGA. Section 8 gives an overview of the verification effort for various parts of the project, summarizes our work, and sketches directions of some future work.

2 Overview of the VAMP Processor

Instruction Set. The full DLX instruction set from [7] 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 [18] is supported. The user sees a floating point register file with 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.

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

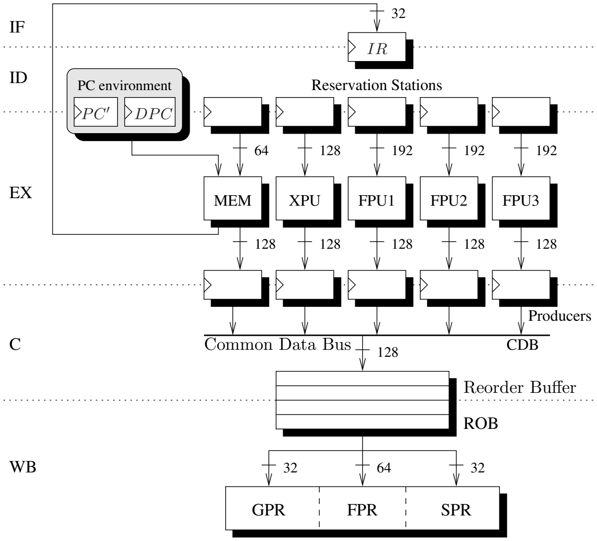

Fig. 1. Main data paths of the VAMP processor

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 [9]. In particular, all four rounding modes, denormal numbers, and exponent wrapping as a function of the interrupt masks are realized.

Interrupt Support. Presently, the interrupts from table 1 in section 4 are supported. Interrupts are maskable and precise. Floating point interrupts are accumulated in 5 bits of a special purpose register *IEEEf* (IEEE flag) as prescribed by the IEEE standard. All special purpose registers (details in section 4) are collected into a special purpose register file. Operations supporting the interrupt mechanism are: i) moves between general purpose registers and special purpose registers. ii) trap. iii) return-from-exception.

Microarchitecture Overview. Figure 1 gives a high level overview of the VAMP microarchitecture. Stages IF and ID are a pipelined implementation of delayed branch as explained in section 4. Stages EX, C and WB realize a Tomasulo scheduler with 5 execution units, a fair scheduling policy on the common data bus CDB, and a reorder buffer ROB (for precise interrupts). The execution units are i) MEM: memory unit with variable latency and internal pipelining. There is presently no store buffer. ii) XPU: the fixed point unit. iii) FPU1 to FPU3: specialized pipelined floating point units with variable latency. FPU1 performs additions and subtractions. FPU2 performs multiplications and divisions. FPU3 performs test-and-set as well as conversions. 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.

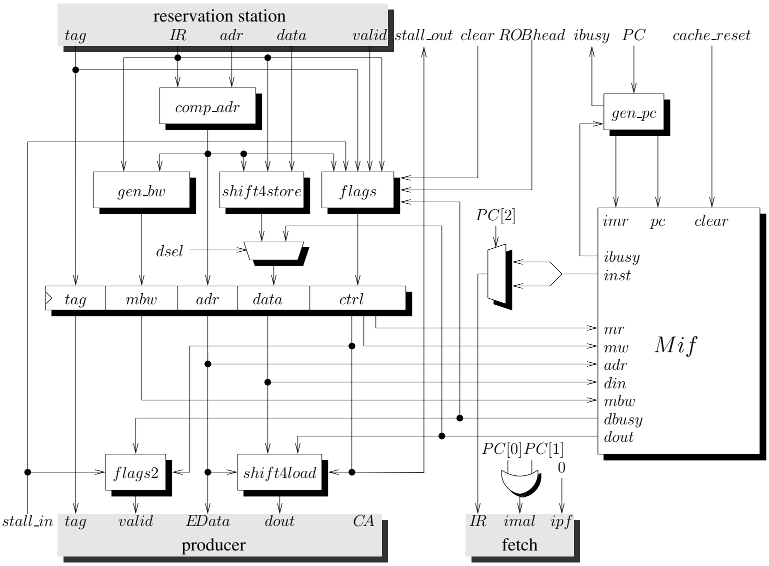

Fig. 2. Data paths of the VAMP memory unit

Figure 2 depicts a simplified view of the memory unit. Internally, it has two pipeline stages. The first stage does address and control signal computations. The second stage performs the actual data cache access via signals adr, din, and $dout$. Instructions are fetched from the instruction cache via signals pc and $inst$. The memory interface *Mif* internally consists of a data cache, an instruction cache, and a main memory. The caches are kept coherent (this does not suffice to guarantee correct execution of self-modifying code). Details are explained in section 6.

3 Correctness Criterion and Tomasulo Algorithm

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 mem_I denotes the main memory of the implementation machine.

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 an instruction into the instruction register *IR* of the implementation machine during cycle T can be specified by $IR_I^{T+1} := mem_I^T[PC_I^T + 3 : PC_I^T]$.

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 $IR_S^i = mem_S^i[PC_S^i + 3 : PC_S^i]$. 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 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.

Scheduling Functions. For hardware cycles T and pipeline stages k of the implementation machine, we formally define an integer valued scheduling function $sI(k,T)$ [14], where $sI(k,T) = i$ has the intended meaning that an instruction I_i is during cycle T in stage k.

By treating instruction numbers like integer valued tags,[2] the definition of these functions is straightforward. We initialize $sI(k,0) := 0$ for all stages. We then "clock" these tags through the pipeline stages under the control of the update enable signals[3] ue_k for the output registers of stage k. If a stage is not clocked, the scheduling function is not changed, i.e., $sI(k,T) := sI(k,T-1) \quad \text{if} \quad /ue_k^{T-1}$. Note that we introduce separate "stages" k for each reservation station and ROB entry.

For the fetch stage[4], e.g., we define $sI(fetch,T) := sI(fetch,T-1)+1$ if ue_{fetch}^{T-1}, meaning that the content of the fetch stage progresses by one instruction in the instruction stream $I_0, I_1, \ldots$ If stage k receives data from stage k' in cycle T, we define $sI(k,T) := sI(k',T-1)$. Note that this covers the case that a stage can receive data from two different stages 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 an execution unit. Thus, the ROB can receive data from the CDB and from the instruction register.

As a form of bookkeeping for the memory unit, we introduce an additional "stage" mem'. The corresponding scheduling function $sI(mem',T)$ equals $sI(mem,T)$ if the memory unit is empty or the instruction in the unit has not accessed the main memory yet. Otherwise, we set $sI(mem',T) := sI(mem,T)+1$. We need this bookkeeping function in order to model whether the memory is already updated by a store instruction.

Correctness Criterion. We are interested in the content of the main memory mem and the register files $RF \in \{GPR, FPR, SPR\}$ after certain instructions I_i respectively before instruction I_{i+1}. The main memory is an output "register" of stage mem and the register files are output "registers" of stage wb. The functional correctness criterion requires an instruction I_i in stage mem' of the implementation machine I to see the same memory content as the corresponding instruction of the specification machine S; formally $mem_I^T = mem_S^{sI(mem',T)}$. The corresponding condition for register files RF is $RF_I^T = RF_S^{sI(wb,T)}$. In general, we prove by induction on T for all stages k and all output registers R of stage k that $R_I^T = R_S^{sI(k,T)}$, where R_S^i can be a visible or

[2] 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.

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

[4] We introduce symbolic names for some stages k, e.g., $fetch$ and mem.

redundant component of the configuration of the specification machine. Note that for technical reasons, we claim for the instruction register that $IR_I^T = IR_S^{sI(fetch,T)-1}$.

The liveness criterion states that all instructions that are not interrupted reach the writeback stage. At the time of submission of this paper, we have separate formal liveness proofs for the scheduler and the execution units; we are currently working on combining them into a single formal liveness proof for the entire machine.

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.

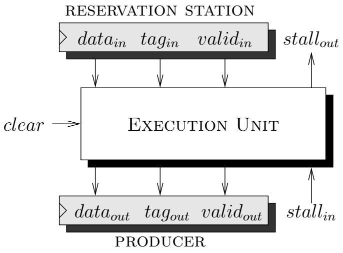
RESERVATION STATION
$data_{in}$ tag_{in} $valid_{in}$
$stall_{out}$
$clear$
Execution Unit
$data_{out}$ tag_{out} $valid_{out}$
$stall_{in}$
PRODUCER

Fig. 3. Model of an execution unit

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 execution units or whether these units are pipelined. The scheduler recognizes instructions completed by the execution units simply by examining the tags returned from the units. The situation is very different for model checking [28].

Integration of Execution Units. The proofs for the scheduler and the proofs for the execution units are separated by the following specifications for the execution units [11, 10]. Notations refer to figure 3.

i) $stall_{in}^T \implies /valid_{out}^T$, i.e., if the scheduler asserts $stall_{in}$, the execution unit does not return a valid instruction.
ii) $\forall T \exists T' > T : /stall_{out}^{T'}$, i.e., the $stall_{out}$ signal is never active indefinitely.
iii) 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 (combinatorial) function the execution unit is supposed to compute.
iv) 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 execution units do not create spurious outputs.

Note that the instructions do not need to leave the execution units in the order they enter the units; all FPUs, e.g., exploit this by allowing instructions on some special operands to overtake other instructions. Moreover, multiplications may overtake divisions (cf. [10] for details).

The four conditions above must be shown for each of the execution units provided the scheduler guarantees the following three conditions: i) No instruction is dispatched to an execution unit which sends a $stall_{out}$ signal to its reservation station. ii) The execution units are not stalled forever by the producers. iii) Tag-uniqueness: no tag which is dispatched into an execution unit is already in use.

4 Delayed Branch and Maskable Nested Precise Interrupts

In the delayed branch mechanism, taken branches yield a new PC of the form $PC + imm+4$, taken branches are delayed, and $PC+8$ is saved to the register file during jump-and-link. In the equivalent delayed PC mechanism [14,18], 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 4 depicts a pipelined implementation of the delayed PC mechanism in the VAMP processor. This construction and its formal correctness proof are automatically obtained by the method for automatic pipeline construction from [15]. Indeed, fetching instructions from the intermediate program counter PC' is—not only intuitively but formally—forwarding of DPC. The role of the multiplexers above PC' and DPC are explained in the following paragraphs about interrupts.

The formal specification of the interrupt mechanism for delayed PC is based on the definitions of [18, 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.

At issue time of an instruction I_i, it is unknow 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". The data paths of the PC environment are shown in figure 4.

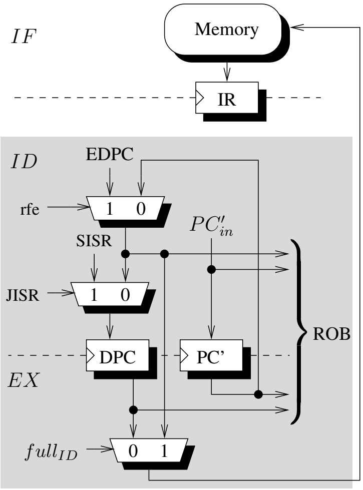

Fig. 4. VAMP PC Environment

Interrupt handling in the specification machine S depends on the components ECA and $EData$. In the implementation, these two registers are treated as additional results of the execution units; thus, execution units have up to *four* 32-bit results. This affects the width of the ROB. The formal correctness of these components in the ROB at writeback time is asserted without additional verification effort by the consistency of the Tomasulo scheduler. Further lemmas are needed for the correctness of the PCs stored in the ROB. The return-from-exception instruction is treated like any other instruction; no special effort is needed here.

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

Table 1. Implemented interrupts

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

Since the main memory is updated before writeback of an instruction, one has to guarantee that in case of an interrupt, all stores *prior* to the interrupted instruction are executed, but none of the instructions *after* it. Especially, one has to show that a store that has reached the writeback stage *also* has accessed the main memory, i.e., it did not enter the wrong execution unit.

5 Floating Point Unit

Execution Units. The FPUs and their verification are described in [11]. The construction and verification of the combinatorial circuits is based on the paper and pencil proofs from [18]. The internal control of the iterative unit for multiplication and division is complex: during cycles, when the division unit performs a subtraction step, the multiplier can be used by multiplication operations or by multiplication steps of other division operations. Moreover, operations with special operands are processed in a single cycle. Thus in general, the units do not process instructions in order, but that is not required by the specifications from section 4. We remark that we have formal proofs but no paper and pencil proofs for the correctness and liveness of the floating point control. The control was constructed and verified with the help of a model checker[10].

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 whether the double precision operand is read.* This is easily realized by treating the upper half and the lower half of double precision operands as separate operands. Thus, reservation stations for dual precision floating point units have 6 operands.

IEEE Flags and Synchronization. The exception flags for interrupts 6 to 12 are part of the result of every floating point operation I_i. They are accumulated in special purpose register *IEEEf* during writeback of I_i. We have already seen in section 4 that this affects the width of the reorder buffer. A move operation I_j which reads from register *IEEEf* is issued only after the entire reorder buffer is empty. This simple modification of the issue logic makes it very easy to prove that the flags of all floating point operations preceding I_j are accumulated when *IEEEf* is read by I_j. A move instruction from *IEEEf* to general purpose register 0, which is constantly 0, acts as a *sync* operation for self-modifying code as explained at the end of the following section.

6 Memory Interface

Loads and Stores with Variable Operand Width. The formal specification of the semantics of the memory instructions is based on the definitions in [18, Chap. 3]. 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 $mem[ea + d - 1 : ea]$ equals bytes $[ba(ea) + d - 1 : ba(ea)]$ of the double word addressed by $da(ea)$ at the memory interface.[6] Details can be found in [18].

Circuits called *shift4load* and *shift4store* are used in order to ensure that data is loaded and stored correctly. These circuits are shown in figure 2. "Shift for store" denotes shifting the data, say the halfword which is to be stored, into the correct position of a doubleword 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. Shift for store and load are implemented by means of two simplified shifters with some control logic [18].

The proof of correctness of the VAMP memory interface is structured hierarchically. First, we verify the VAMP with an idealized memory interface m_spec, a dual-ported memory without caches. Second, we show that a cache memory interface with split caches backed up by a unified main memory m_impl behaves exactly like the dual-ported memory m_spec. Thus, m_spec serves as the specification for the cache memory interface. By putting these two independent proofs together, we obtain the correctness of the VAMP with split caches with respect to the memory mem_S of the specification machine.

Cache Specification and Implementation. The memory m_spec is defined recursively, i.e., it is updated on the double word address a iff a write access to address a terminates. Separate byte-enables mwb_b allow for updating only some of the 8 bytes stored on address a. Formally, we have for any byte $b < 8$ and any double word address a:

$$m\_spec[8 \cdot a + b]^{T+1} := \begin{cases} din[b]^T & a = adr^T \wedge mw^T \wedge mwb_b^T \wedge \ /dbusy^T \\ m\_spec[8 \cdot a + b]^T & \text{else} \end{cases}$$

The memory interface is implemented with split caches connected to a single main memory as depicted in figure 5. We use a write-back policy for the data cache, i.e., on a

[6] Note that this specifies little endian memory organization.

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.

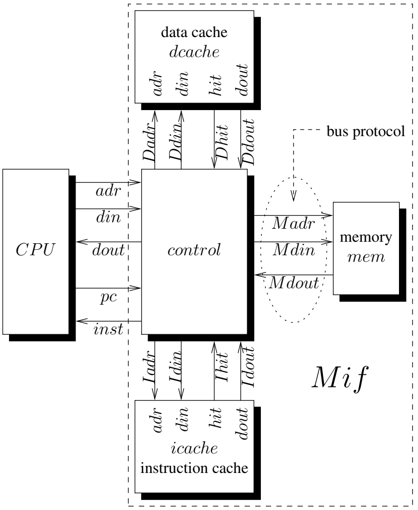

Fig. 5. Cache memory interface

The instruction and data caches are implemented as k-way sectored set-associative caches using a LRU replacement policy. Cache sectors consist of 4 double words since the bus protocol supports bursts of length 4.

Typical Lemmas. The inductive invariant used 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 m_impl on this address $Dadr^T$ contains the specified memory content. 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:

$$\begin{array}{rcll} Ihit^T & \Longrightarrow & Idout[b]^T & = m\_spec[8 \cdot Iadr^T + b]^T \wedge \\ Dhit^T & \Longrightarrow & Ddout[b]^T & = m\_spec[8 \cdot Dadr^T + b]^T \wedge \\ /(Dhit^T \wedge dirty^T) & \Longrightarrow & m\_impl[8 \cdot Dadr^T + b]^T & = m\_spec[8 \cdot Dadr^T + b]^T . \end{array}$$

This invariant is strong enough to show transparency of the whole memory interface 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 reload 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.

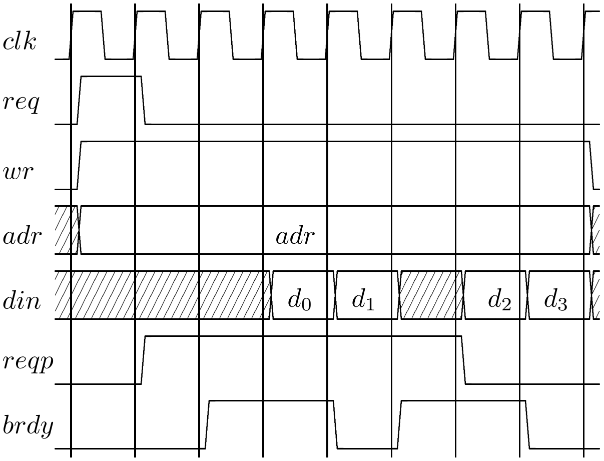

Fig. 6. 4-burst write timing diagram

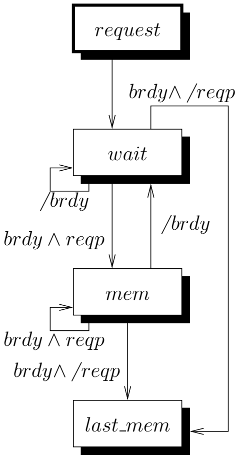

Fig. 7. Burst control FSD

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 6. 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 $/reqp \wedge brdy$.

As part of our correctness proof for the memory interface, we have formalized this bus protocol and proved that an automaton[7] according to figure 7 implements this protocol correctly by means of theorem proving. 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 $last\_mem$.

Self-Modifying Code. We consider self-modifying code independent of the implementation of the memory interface. As an additional precondition for the correctness of code, we demand that in case an instruction is fetched from a memory location adr, there is a special *sync*-instruction between the last write to adr and the fetch of adr.[8] 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 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.

7 Synthesis

We have translated the PVS hardware description of the VAMP processor to Verilog HDL using an automated tool called `pvs2hdl`. The tool unrolls recursive definitions and then performs fairly straightforward translation. The Verilog representation of the

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

[8] This implies the correspondency condition from [23].

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 is also present on this FPGA. The VAMP processor occupies about 18000 slices of a Xilinx Virtex FPGA. This 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.

Note that we assume a fully synchronous design, i.e., all registers share the same clock and RAM blocks for register files or caches are also updated synchronous to this clock; thus, concerning timing, they can be treated like registers. In a fully synchronous design, valid data is needed only at the rising edge of the clock with certain setup- and hold-times. The synthesis software analyzes all paths between inputs and registers, registers and registers, and registers and outputs; thus, it can guarantee that our logical design can be implemented with a certain maximum clock speed preserving all our proved properties. In particular, we fully ignore any *glitches*, i.e., instabilities in signals during a clock period that are resolved until the next rising edge of the clock since these glitches do not influence fully synchronous designs. Thus, our approach does not cover designs where certain signals must be kept stable for several cycles, i.e., where glitches must not occur. 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.

We have ported the `gcc` and the GNU C library for the VAMP in order to execute test programs on the VAMP. As it was to be expected from our verified design, we found no errors in the VAMP processor. When testing some cases of denormal results of floating point operations, however, we found differences between the VAMP FPU and Intel's Pentium II FPU. This is due to some discrepancies of Intel's FPU to the IEEE standard. See [11] for further details.

8 Conclusion

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. Note especially that "Putting it all together" took a whole person-year for several reasons. First of all, the proof of the Tomasulo core from [12] 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 lot of effort went into patching the interfaces. Additionally, self-modifying code and the special implementation of the *IEEEf*-register 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 really scale too well for projects this large; typechecking of the VAMP alone takes already more than two hours on our fastest machine.

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

Table 2. Verification effort

Part	Effort in years	Lemmas	Proof steps
Tomasulo core & ALU	2	521	14367
FPU	3	1046	25936
Cache Memory Interface	2	566	24432
Putting it all together	1	415	23887
Total	8	2548	88622

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, 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 design is synthesized[9] 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 translation look aside buffers 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. iv) 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.

References

1. 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.
2. B. Brock, W. A. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. Technical Report Technical Report 86, Computational Logic Inc., 1994.
3. 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.
4. 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.
5. W. Damm and A. Pnueli. Verifying out-of-order executions. In *Charme IFIP WG10.5*, pages 23–47, Montreal, Canada, 1997. Chapman & Hall.
6. 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.
7. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[9] The trivial proof of synthesizability.

8. 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.
9. Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.
10. 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.
11. C. Jacobi. *Formal Verificaton of a fully IEEE compliant floating point unit*. PhD thesis, Saarland University, Germany, 2002.
12. D. Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.
13. 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.
14. D. Kröning, S. Müller, and W. Paul. Proving the correctness of processors with delayed branch using delayed PCs. pages 579–588, 2000.
15. D. Kröning and W. Paul. Automated pipeline design. In *Proc. of the 38th Design Automation Conference*, pages 810–815. ACM Press, 2001.
16. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV 98*, volume 1427. Springer, June 1998.
17. K. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME 2001*, volume 2144 of *LNCS*. Springer, 2001.
18. S. M. Mueller and W. J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.
19. 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.
20. 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.
21. 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.
22. 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.
23. J. Sawada and W. A. Hunt. Trace table based approach for pipelined microprocessor verification. In *CAV 97*, volume 1254 of *LNCS*. Springer, 1997.
24. J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In *CAV 98*, volume 1427 of *LNCS*. Springer, 1998.
25. X. Shen, Arvind, and L. Rudolph. CACHET: an adaptive cache coherence protocol for distributed shared-memory systems. In *International Conference on Supercomputing*, 1999.
26. J. Stoy, X. Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *FME*, volume 2021 of *LNCS*. Springer, 2001.
27. 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.
28. M. N. Velev and R. E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *DAC*. ACM, 2000.