Correct Hardware by Synthesis from PVS

Sven Beyer, Christian Jacobi, Daniel Kroening[⋆,⋆⋆], and Dirk Leinenbach
{sbeyer,cj,kroening,dirkl}@wjpserver.cs.uni-sb.de

Saarland University
Computer Science Department
D-66123 Saarbrücken, Germany
Tel/Fax: +49/681/302-{4129,4290}

April 9, 2003

Abstract. We present a methodology for obtaining provably correct hardware. We model and prove complex hardware in the theorem proving system PVS. The hardware is synthesized by translating the PVS designs into synthesizeable Verilog HDL by means of an automated tool. The paper describes how we model both combinatorial and clocked circuits in the theorem prover. We describe how the tool translates the hardware to Verilog. Finally, we give experimental results, which include the translation of a complete and provably correct IEEE floating point unit. The FPU has been implemented and tested on a Xilinx FPGA.

1 Introduction

Nowadays, microprocessors and other custom made chips are embedded in nearly all life-critical devices. Therefore, the verification of microprocessors becomes increasingly important. To avoid any gaps in the verification process due to human error, the usage of formal methods is required. However, obtaining provably correct hardware of significant complexity is usually a very challenging and time-consuming task; therefore, it is often considered to be too expensive or even infeasible.

The traditional way for verification is to take a given design in a hardware description language or in form of a net list and to argue about this design in a formal way, e.g., using a theorem prover or model-checker. Both theorem proving and model-checking get less feasible with increasing design size. The reason for this is that both methods have no or only limited capabilities to exploit the hierarchical structure of the design. For example, it is not possible to define parameterized, recursive modules such as adders or decoders, e.g., in Verilog. In order to verify a design, one would have to develop and verify each occurrence of such a module separately, and one could not easily verify the design in a hierarchical way.

[⋆] Currently at Carnegie Mellon University.

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

We therefore propose a completely different approach to the problem; we suggest designing hardware in a higher level language, such as used by theorem proving systems. We are using PVS [29] for this task. However, the design is still using gate level primitives such as AND, OR gates and inverters. Using these primitives, we construct complex designs in a hierarchical way. As the hierarchy of the design is available in detail, the amount of manual work required to verify the designs is significantly reduced. This allows for verifying even complex designs like microprocessors.

The designs developed that way are not directly synthesizeable. In particular, we employ recursive constructions and parameterized modules. In order to get actual hardware, we have developed a tool called `pvs2hdl` that converts the hardware design into synthesizeable Verilog HDL. Using this tool, we have translated a formally verified IEEE compliant floating point unit (FPU) to Verilog and implemented it on a Xilinx FPGA board. Since neither the Xilinx synthesis tools nor `pvs2hdl` are formally verified, we tested the implemented design of our formally verified FPU with several hundred thousand test vectors without discovering a bug. In a few weeks, we expect a complete microprocessor called VAMP [15] to be implemented on the FPGA board in the same way.

The actual verification of our hardware in PVS is not discussed in this paper. We refer the reader to [2, 3, 16, 20, 21] for details on this topic.

Project status. The work presented here is part of our institute's VAMP project which aims at the formal verification and implementation of a complete microprocessor. The VAMP is a variant of the DLX [13, 25] microprocessor, a 32 bit RISC processor based on the MIPS instruction set. The VAMP processor features a Tomasulo scheduler, delayed branch, a cache memory interface, precise interrupts, and an IEEE compliant floating point unit.

The verification of an in-order CPU core is complete, the verification of the Tomasulo out-of-order core will be completed in a few weeks [20, 21]. The verification of the cache memory interface is almost finished. The verification of the combinatorial floating point circuits and the FPU pipeline control is complete [2].

The development of the `pvs2hdl` tool is complete. All PVS specifications and proofs, the Verilog files, and `pvs2hdl` are available at our web site.[1]

Paper outline. Related work is discussed in section 2. In section 3 we describe how we model combinatorial circuits in PVS and how `pvs2hdl` translates these to Verilog. In section 4 we extend to clocked circuits and RAM and ROM. We give experimental results in section 5 including the translation and implementation of a floating point unit. Finally, in section 6 we discuss the results, benefits, and drawbacks of our approach to creating provably correct hardware.

[1] http://www-wjp.cs.uni-sb.de/projects/verification/

2 Related Work

Our approach to creating provably correct hardware is the translation of designs from the theorem-proving system PVS to Verilog. Of course, one could take the reverse approach: design the hardware in some conventional HDL and translate it to the desired formal methods tool. For example, Brayton et. al. [6] translate a subset of Verilog to their VIS system in order to model-check the designs. McMillan's SMV model-checker [24] has built-in Verilog support. While model checking is a good way to verify control dominated designs, the verification of combinatorially complex designs such as floating point units is currently beyond the scope of model-checkers.

Formal verification of hardware correctness is not a new issue. Schneider and Kropf [17], for instance, combine theorem-proving and model-checking in order to efficiently verify hardware correctness. However, they only report the verification of circuits consisting of about one hundred gates and about 20 flip-flops. This sharply contrasts to the verification of a complete FPU with many thousand gates and several hundred flip-flops.

Translation of hardware descriptions in higher-order-logics into hardware is also not new, e.g., it is done by the industrial Lambda Toolkit [33]. However, examples of translation and verification as large as a whole floating point unit are not reported. A similar approach is done by Hanna et al. [12]. The verification of designs of the complexity of a FPU is not reported in their work.

Russinoff translates AMD's register transfer language into input for the theorem prover ACL2 [32]. As an example, he verifies a simple floating point multiplier. However, the complexity of his circuits is far less then the complexity of the circuits we have verified in PVS. Borrione's PREVAIL system [5, 4] is capable of translating VHDL designs to the Nqthm theorem prover. However, the verification of complex designs is not reported in their work.

In order to verify very complex circuits, one has to exploit the structure of the designs. This is not so important for the verification of simple designs or for model-checking since model-checkers exploit the design structure only in a limited way. In order to achieve as much structure as possible in the hardware description, we design the hardware directly in PVS. This allows, e.g., for the definition of recursive designs, and the development of a library of general purpose circuits such as adders and decoders which may be used for arbitrary bit widths [3]. This considerably simplifies the verification and would not be possible in the Verilog-to-PVS approach.

Brock and Hunt have verified the FM9001 processor on a netlist level [7]. The netlist was defined in Nqthm, and then automatically translated to LSI Logic's Netlist Description Language (NDL). The NDL file served as basis for an ASIC implementation of the FM9001. However, the complexity of the FM9001 is not comparable to our VAMP processor; e.g., the FM9001 has no floating point unit.

Schlör [34] develops a prover for VHDL-based hardware design in the FORMAT project exploiting both theorem proving and model checking. He partly integrates design and verification by the usage of a graphical specification language. This considerably facilitates the usage of formal methods for the hardware designer, but the verification of very complex circuits is again not reported.

The PROSPER toolkit [11] also offers easy integration of formal methods into a design environment. Arbitrary verification tools can be incorporated into the PROSPER system which allows for a maximum flexibility. One main objective is a high automation of the verification, i.e., the proof support shall become invisible to the designer. However, only small parts of the design are verified in order to help eliminating bugs. Complete formal verification against a full specification is not an issue in this work; it is considered to be largely of academic interest [11, section 1]. This claim is clearly refuted by the design and verification of our FPU.

We only consider synchronous circuits and we do not investigate delay of combinatorial circuits. Russinoff, on the other hand, develops a formalization of a subset of VHDL in Nqthm[31] and he extensively deals with gate delay in this paper. He proves the correctness of small circuits achieving asynchronous communication, but he does not deal with designs of a complexity even remotely comparable to ours.

Mycroft and Sharp [26, 27] model hardware in a functional programming language and translate it to Verilog. Their goal is the ability to design hardware on a very high level instead of gate-level. However, formal verification is not an issue in their work.

3 Modeling, Verifying and Translating Combinatorial Circuits

Modeling. In this section we describe the subset of the PVS language which we use to model combinatorial hardware and which is supported by our `pvs2hdl` translation tool.

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 [8]. Nested records of bits and bitvectors are supported. The basic operators are *and, or, not, xor, if* and *cond*. These operators are defined on bits and bitvectors of equal length.

Let bv_1, bv_2 be bitvectors, and b be a bit. Then $bv_1(i)$ denotes the ith bit of bv_1 (indexed $n-1, \ldots, 0$); $fill[n](b)$ denotes the bitvector of length n consisting only of b's; $bv_1 \circ bv_2$ denotes the concatenation of bv_1 and bv_2; $bv_1\hat{}(i, j)$ denotes the sub-bitvector $bv_1(i) \ldots bv_1(j)$; $nat2bv[n](i)$ and $int2bv[n](i)$ denote the bitvectors of length n with binary and 2's-complement value i, respectively.

Bitvectors may be composed of single bits by λ-expressions. For example, $\lambda(k : \{0, 1, ..., n\}) : bv_1(n-1-k)$ flips bv_1, i.e., bit k of the resulting bitvector equals bit $n-1-k$ of bv_1.

The *let* $x = expr$ construct assigns the alias x to the expression $expr$. In this way, the common sub-term $expr$ may be eliminated by using x instead of $expr$.

The definition of functions is supported which corresponds to the definition of hardware modules. Function calls correspond to the usage of hardware modules.

Functions have bits and bitvectors (or records thereof) as inputs and outputs. In order to allow parameterized designs (e.g., a carry-chain adder of arbitrary size), additional integer parameters may be used in function definitions. In order to define hardware with recursive structure, the definition of recursive functions is possible. This is an important feature since it allows for inductive reasoning on recursively defined circuits (see below).

In order to allow convenient description of hardware, the use of expressions involving $+, -, \times, \div, \lfloor\cdot\rfloor, \lceil\cdot\rceil, mod, div$, exponentiation and comparisons are supported, e.g., for the selection of sub-bitvectors, in expressions used in λ-terms, and as integer parameters in function calls.

PVS features not supported by pvs2hdl. While we support the translation of a certain subset of the PVS language to Verilog, there are numerous PVS language elements not supported, most of which are not useful in designing hardware, anyway. For example, expressions in PVS may contain quantifiers $(\forall, \exists)$. Having no gate-level-equivalent, these quantifiers are not supported by `pvs2hdl`. PVS also is capable of handling higher-order logic statements (e.g., the choice-function on non-empty sets), which are of course not translatable to Verilog.

If a function is to be translated by `pvs2hdl`, its parameters and return value may only be nested records of bitvectors, and naturals for parameterized designs. Any other types supported by PVS (e.g., reals, lists, function types...) are not supported by the translation tool. In order to decrease translation effort, overloading of functions, operators, or data types is not supported.

The language elements not supported by the translator may yet be used in the PVS specifications and proofs of the hardware designs.

Example. Figure 1 shows the schematic construction of a full-adder and an n-bit carry-chain adder. Figure 2 shows the corresponding PVS code; `fulladder` is a function mapping three input bits a, b, c to a bitvector of length 2. The connectives between the input bits in the PVS definition resemble the gates in the schematic construction.

The carry-chain adder is modeled by the function `carry_chain`, which has two input bitvectors a and b, and a carry-in cin. The additional parameter $n \in \mathbb{N}^+$ specifies the width of the inputs and output. If the length of the inputs is only 1 bit, the carry-chain adder is just a full-adder; otherwise, the lower-order bits $n-2, \ldots, 0$ of the inputs are added using an $(n-1)$-bit adder; the most significant bit of this adder and of the operands are fed into a full-adder. The results of the full-adder and the $(n-1)$-bit adder are then concatenated. This matches the schematic construction of the carry-chain adder in figure 1.The *MEASURE* statement in the function `carry_chain` is needed by PVS for recursive functions, but it is ignored by our translation tool.

Verification. The verification of the carry-chain adder in the above example is very easy. One first verifies the correctness of the full-adder by automatically checking all 8 possible inputs, and then uses induction on the recursive structure of the adder to show that it correctly sums up the inputs. The proof does not

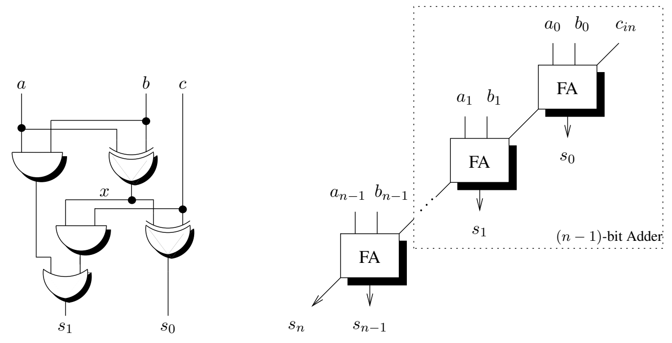

Fig. 1. Schematic construction of full-adder and n-bit carry-chain adder

```
fulladder(a, b, c: bit): bvec[2] =
    LET x = a XOR b IN
     ((a AND b) OR (c AND x)) o (x XOR c);

carry_chain(n: posnat, a, b: bvec[n], cin: bit):
        RECURSIVE bvec[n+1] =
    IF n = 1 THEN
      fulladder(a(0), b(0), cin)
    ELSE
      LET chain = carry_chain(n-1, a^(n-2, 0), b^(n-2, 0), cin) IN
        fulladder(a(n-1), b(n-1), chain(n-1)) o chain^(n-2,0)
    ENDIF
    MEASURE n;
```

Fig. 2. Full-adder and carry-chain adder

depend on a concrete bit-width n, but is for arbitrary n. Details on the verification of a carry-chain adder in PVS can be found in [22], the verification of more complex general purpose circuits is reported in [3], e.g., encoders, decoders, parallel prefix circuits, and Wallace tree multipliers.

Translation. The translation of PVS hardware descriptions to Verilog works as follows: for each PVS function, a Verilog module is generated. 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, and a Verilog module is generated for all integer parameters occurring in the recursion. This is necessary since Verilog does not support recursion.

The translation of *and, or, not, xor, if, cond, fill*, bitvector concatenation and extraction, and constant generation with *nat*2*bv* and *int*2*bv* to Verilog is straightforward, since these constructs have their literal counterparts in Verilog. Verilog does not support records, thus records are flattened into individual bitvectors during the translation to Verilog.

```
module fulladder(a0,a1,a2,out3);
    input a0;           //  a
    input a1;           //  b
    input a2;           //  c
    output [1:0] out3;  //  out

    wire wire0;         //  x

    assign wire0 = (a0 ^ a1); // ^ is XOR in Verilog
    assign out3 = {((a0 & a1) | (a2 & wire0)),
                   (wire0 ^ a2)};
endmodule

module carry_chain_1(a0,a1,a2,out3);
    ...
    fulladder m0 (a0,a1,a2,out3);
endmodule

...

module carry_chain_4(a0,a1,a2,out3);
    input [3:0] a0;     //  a
    input [3:0] a1;     //  b
    input a2;           //  cin
    output [4:0] out3;  //  out

    wire [3:0] wire0;   // chain
    wire [1:0] wire1;   // fulladder

    assign out3 = {wire1,wire0[2:0]};

    carry_chain_3 m0 (a0[2:0],a1[2:0],a2,wire0);
    fulladder m1 (a0[3],a1[3],wire0[3],wire1);
endmodule
```

Fig. 3. A part of a translated 4-bit carry-chain adder in Verilog

The λ-expressions are translated separately for each bit, and the bits are then concatenated to yield the desired bitvector. A *let* $x = \mathit{expr}$ 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. While there may be multiple instantiations of the same module—resulting in multiple

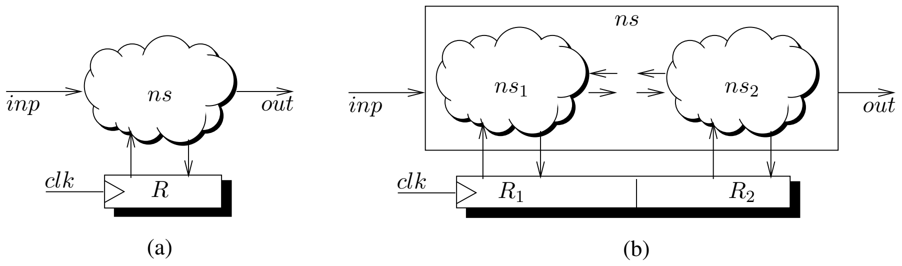

Fig. 4. Model of clocked circuits

occurrences of the module in hardware—the module itself is translated to Verilog only once.

Integer expressions are evaluated by the translation tool. For example, the expressions $n-1$ and $n-2$ in the above carry-chain example are evaluated given the concrete instantiation of n.

Example (cont.). Figure 3 shows (sanitized) parts of the result of the translation process of a 4-bit carry-chain adder. The translation tool generates unique names for the wires in a Verilog module. The PVS names for function parameters and *LET* aliases are added as comments to the Verilog code in order to increase readability. The module `fulladder` has a wire *wire*0 corresponding the the *LET* $x = \ldots$ in the PVS code. The output of the full-adder is generated by interconnecting the input bits and x as in the PVS definition. The module `carry_chain_1` represents the PVS function `carry_chain` where the parameter n is 1. The inputs of this module are simply passed to a sub-module `fulladder`. This reflects the recursion start. Module `carry_chain_4` shows the implementation of the PVS function `carry_chain` for $n = 4$. It instantiates a 3-bit adder module `carry_chain_3` and a full-adder module, as suggested by the recursive PVS definition.

4 Modeling Clocked Circuits

The subset of PVS we use to model combinatorial hardware is similar to a functional programming language, thus offering no direct support for global, state-holding variables as opposed to imperative programming or hardware description languages. Therefore, the concept of registers needs some extra consideration in PVS. We only consider single clock-domains.

We may regard a clocked circuit as a circuit with only one state-holding register R (which may consist of many bits) and a *combinatorial* circuit ns (cf. Fig. 4a). The circuit ns takes as inputs the current state of register R and some external inputs, and computes some outputs and the next state of register R. The combinatorial circuit ns can be represented as a PVS function as described in section 3:

$$\begin{aligned} ns : State \times Input &\rightarrow State \times Output \\ (current\_state, inp) &\mapsto (next\_state, out) \end{aligned}$$

The *State*, *Input*, and *Output* types may be arbitrarily nested records of bitvectors.

Multiple clocked circuits can be combined to one larger clocked circuit by interconnecting inputs and outputs, and using the Cartesian product of the two state types as new state type (cf. Fig. 4b). In this way, e.g., we embed the FPU into the processor; the result is one single combinatorial next-state function operating on the state of the processor and the state of the FPU.

In order to translate the clocked circuit to Verilog, the name of the next-state function (say *ns*) and the state type (say *State*) are passed to the `pvs2hdl` tool. The tool then first translates the function *ns* to a combinatorial Verilog module `ns` as described in section 3. The translator then creates a Verilog module `ns_clk` without either *current_state* as input nor *next_state* as output, but taking an additional clock input *clk*. A local register variable R of type *State* is declared in the module (in case *State* is a record type, there may be multiple registers due to flattening). The clocked module `ns_clk` has a single sub-module `ns`. The inputs and outputs of `ns_clk` are connected to the corresponding inputs of `ns`, and the register R is connected to the *current_state* and *next_state* input/output of the `ns` module, as depicted in figure 4a.

Modeling RAM and ROM. Apart from registers, modern microprocessors also contain RAM, e.g., for register files or caches, since RAM is considerably cheaper than an equivalent amount of registers. In PVS, RAM is modeled as a special clocked circuit, which may be embedded in other clocked circuits as described above.

In order to model RAM, we define a special parameterized type *ram_t[k,N]* representing a $2^k \times N$ SRAM. A special function *ram_rw(RAM,adr,din,we)=(RAM',dout)* handles accesses to the RAM. The parameter *RAM* reflects the old state of the RAM; *adr*, *din*, and *we* represent input address, data input and write-enable, respectively. The return of the function is the new RAM state *RAM'* and the data output. Similar RAM access functions are defined to model multi-port RAMs. The RAM access functions are not synthesizeable by means described above. Instead, `pvs2hdl` replaces it by a Verilog module containing the RAM definition in the translation output.

ROM is modeled as a case-statement in PVS mapping addresses to data; `pvs2hdl` translates the case-statement to a file which then is used as ROM-description during Verilog synthesis.

5 Experimental Results

In this section we present experimental results. We have translated various combinatorial and clocked circuits from PVS to Verilog, and implemented them for usage on Xilinx Virtex-E FPGAs [35]. We compare the cost and delay of the generated circuits to implementations in Verilog using special Xilinx macros. The cost and delay of the circuits are determined using the Xilinx Foundation software. The unit of cost is a *Virtex-E slice*, which reflects to FPGA-area. The unit of delay is a nanosecond (ns).

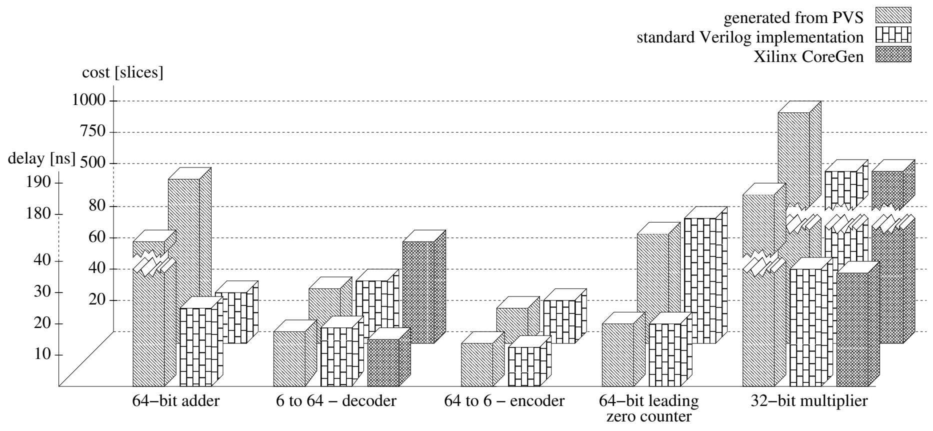

Fig. 5. Comparison of the cost of translated designs and optimized macros

Adder. We have translated a 64-bit carry chain adder as defined in figure 2. The implementation resulted in a cost of 106 slices and a delay of 79 ns, while the use of a standard Xilinx adder macro resulted in a cost of 33 slices and a delay of 23,4 ns. This large gap is due to special hardware resources called *fast carry logic* used by the optimized adder macro. Since the `pvs2hdl` translator does not trigger the usage of these architecture-dependent resources, our implementation falls very short of the standard macro. This phenomenon shows up only for adders, incrementers, and multipliers.

We therefore have added the support for pre-defined Verilog modules to the `pvs2hdl` tool, which allows for the replacement of basic modules (such as adders) defined in PVS by cheaper and faster Xilinx macros. Of course, the correctness of the complete circuit then depends on the correctness of the Xilinx macros.

Decoder. We have implemented a decoder recursively in PVS. The implementation of the 6-to-64–bit decoder has a cost of 36 slices and a delay of 18.2 ns. Using a standard Verilog implementation as in [9, pg. 328] yields 40 slices and 19.5 ns. A decoder generated by the Xilinx CoreGen software yields 64 slices and 16.3 ns. So our implementation is the cheapest, and is nearly as fast as the Xilinx CoreGen variant, although the latter has nearly twice the cost. This is a remarkable achievement since designs generated by synthesis from a common HDL mostly result in significantly slower and larger circuits in FPGAs[10]. This is due to the fact that the generic HDL descriptions do not exploit the special FPGA resources.

Figure 5 shows comparisons of various circuits implemented in PVS with implementations in Verilog or with special Xilinx macros.

Floating Point Unit. We use an IEEE compliant floating point unit (FPU) as a larger example. The FPU supports multiplication and division, addition and subtraction, single and double precision floating point numbers, normal and denormal numbers, conversion, and IEEE exceptions. Details on the design and verification of the FPU can be found in [2].

The main components of the FPU are a 58-bit multiplier, three 64-bit shifters, two 64-bit leading zero counters, a 58-bit half-decoder, and various adders and incrementers. The 58-bit multiplier is built from two 29-bit and one 30-bit multiplier using the Karatsuba-Ofman scheme [18]. The smaller multipliers and all adders and incrementers in the FPU are replaced using faster and cheaper Verilog modules.

The translation of the multiplication & division unit from PVS to Verilog yields a design using 4692 slices. The Xilinx software reports a gate count of 88.000 for the implementation. The maximum clock frequency is 18.8 MHz. The add & subtract unit and the misc unit which for instance converts between fixed- and floating-point numbers take each about 2000 slices and they can be clocked at 18 MHz. So the complete FPU uses about 9000 slices and runs at 18 MHz. We achieved this implementation without any floorplannig by straight synthesis from VERILOG. We expect a considerable speedup by extensive flooplanning, but we have not yet investigated this possibility.

We ran several hundred thousands of test vectors on the implemented FPU on the FPGA without discovering a bug in our FPU. We have compared the results delivered by our FPU with those of Intel's Pentium II processor, and have discovered discrepancies of Intel's FPU to the IEEE standard in some cases of denormal results. The same discrepancies apply for AMD processors. For details, we refer the reader to [14].

To the best of our knowledge, our FPU is the first publicly available and completely formally verified FPU.

6 Discussion and Conclusion

We have described a methodology to design hardware in the verification system PVS, and to automatically obtain synthesizeable Verilog code from the PVS hardware description. There are several benefits to this approach:

1. The use of high-level constructs such as recursion and λ-expressions allows for the concise description of structured hardware.
2. The description of hardware in PVS enables the formal verification of the hardware descriptions against some formal specification.
3. The PVS system offers support for both theorem proving as well as model checking. Thus, we can exploit both techniques in our proofs without a tedious and error-prone translation between two different verification systems.

4. The verification can exploit the structured and modular description of the hardware; one can verify general purpose circuits for arbitrary bit widths, and use the correctness results in the verification of larger and larger circuits. In this way, it is possible to design, verify and implement hardware of almost arbitrary complexity.

The latter points are particularly important, as the design of complex hardware systems is very error prone, and verification is therefore an increasingly important part of the development cycle. We have specified and verified as complex circuits as complete floating point units, which proves the feasibility of the approach.

The verification heavily exploits the structure of the hardware. We have verified a library of parameterized general purpose circuits [3], upon which hierarchically more and more complex circuits are built and verified. The verification of each circuit uses the correctness statements of its sub-circuits, so that the hierarchy is maintained during verification. This considerably eases the verification task. If we had designed the hardware in Verilog and then translated it to PVS, we would have had less structure, and so the verification would have been harder.

The hardware is specified and verified in PVS on the gate level. In order to obtain real hardware, we have developed the `pvs2hdl` tool to automatically translate the PVS hardware descriptions to Verilog. Several other tools (synthesizer, place & route tools, etc.) then transform Verilog to real hardware. Each of the steps involved is not formally verified and could introduce new errors into the design. In fact, even the PVS proof checker could have bugs which hide errors in the "verified" PVS hardware description.

However, there is a great benefit in having verified the PVS gate-level description of the hardware: the design is free of *logical* errors (if we have not been trapped by bugs in PVS). Nowhere an *and*-gate is used where an *or*-gate would have been correct, no adder is too small in size, ... Although each of the tools mentioned above could introduce new errors, the confidence in the logical correctness of the gate-level greatly improves the confidence in the correctness of the ultimate hardware. The famous Pentium bug, for example, was a logical bug [30] which would have been discovered in our verification. There are approaches to verified synthesis tools [1, 23]. However, the formal verification of real-size synthesis tools is far beyond the capabilities of current software verification techniques.

The FPU mentioned above has been implemented on a Xilinx FPGA. The FPU worked on the first try. No debugging of the FPU circuits was necessary after having verified the gate-level in PVS. We have run hundred-thousands of test-vectors without finding a bug.

There are drawbacks in the use of PVS as hardware development & verification system, which we don't want to be left unmentioned:

1. Designing combinatorial circuits in a functional programming language and our notion of clocked circuits is not common practice for hardware designers.

2. The support for fast simulation and visualization is common in modern development systems, but not available in PVS. In the design phase, many obvious errors can be found by simulation. The harder errors could then be found during formal verification. The theorem prover ACL2 [19], on the other hand, offers efficient lisp-based support for simulation. However, ACL2 cannot handle higher-order logic in contrast to PVS which would considerably complicate our proofs.
3. We support only a single clock domain. Thus, we cannot directly model a common SDRAM interface of a CPU in PVS where the SDRAM is clocked independently of the CPU. An extension of our PVS hardware model to cover multiple clock domains is possible, but we have not yet investigated this possibility.
4. Our PVS hardware model supports only a small subset of the Verilog hardware description language which is sufficient to design any combinatorial circuit or clocked circuit with one single clock domain. However, by designing hardware in PVS, we disallow any "dirty" design tricks employed in common HDLs in order to achieve a maximum optimization of the design. Therefore, it may not be possible to design hardware as thoroughly optimized for speed as the new Pentium, for instance. However, it is not our project goal to compete with modern microprocessors in performance, but to offer formally verified correctness guarantees for microprocessors in safety-critical devices. Many of these safety-critical devices do not need a clock frequency of more than 100 MHz which could be achieved by our approach. We see a considerable market for formally verified microprocessor of comparably modest performance, e.g., in medical devices, nuclear reactors, and military applications.
5. A considerable part of the verification effort is needed for very low-level circuits for which appropriate automatic methods are available, e.g., equivalence-checking. One could save a great deal of time by automatically verifying small sub-circuits, and restrict interactive proof development to the composition of such sub-circuits to larger circuits which are too large for automatic verification. However, these automatic methods are not available in PVS.

There are publicly available tools supporting some of these features, but non integrates all features needed for an integrated development & verification system. There are such tools in industry, e.g. Intel's Forte system [28], but these tools are not publicly available, they are not even sold. In order to develop and formally verify large hardware systems against a high-level specification, we believe our approach is currently the only feasible.

Future work. In order to further improve confidence in the correctness of the design, one could verify the PVS hardware specification against the netlist generated by the Verilog synthesizer. This would close the verification gap involving our tool and Verilog tools. The verification of the netlist could be performed using equivalence-checkers; however, these tools probably do not scale to the required circuit size. We have not investigated the verification of netlists yet.

Part of our group at Saarland University is currently starting the verification of large software systems. It is likely that they will use a variant of our `pvs2hdl` tool to specify software in PVS and then translate it to a conventional language, e.g., C++.

Acknowledgments. The authors would like to thank Christoph Berg, Wolfgang Paul, and Jochen Preiß for valuable discussions.

References

1. M. Aagaard and M. Leeser. Verifying a logic-synthesis algorithm and implementation: A case study in software verification. *IEEE Trans. on Software Engeneering*, 21(10), Oct 1995.
2. C. Berg and C. Jacobi. Formal verification of the VAMP floating point unit. In *Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*. Springer, 2001.
3. C. Berg, C. Jacobi, and D. Kroening. Formal verification of a basic circuits library. In *Proc. 19th IASTED International Conference on Applied Informatics, Innsbruck (AI'2001)*, pages 252–255. ACTA Press, 2001.
4. D. Borrione, H. Bouamama, D. Deharbe, C. le Faou, and A. Wahba. HDL-based integration of formal methods and CAD tools in the PREVAIL environment. In *FMCAD'96*, volume 1166 of *LNCS*. Springer, 1996.
5. D. Borrione, L. Pierre, and A. Salem. Formal verification of VHDL descriptions in the PREVAIL environment. *IEEE Design&Test Magazine*, 9(2), June 1992.
6. R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, et al. VIS: a system for verification and synthesis. In *CAV 96*, volume 1102 of *LNCS*. Springer, 1996.
7. B. C. Brock and W. A. Hunt, Jr. 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(1):71–104, July 1997.
8. 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.
9. M. D. Ciletti. *Modeling, Synthesis, and Rapid Prototyping with the VERILOG HDL*. Prentice Hall, 1999.
10. K. Claessen, M. Sheeran, and S. Singh. The design and verification of a sorter core. In *Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*. Springer, 2001.
11. L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, G. Robinson, M. J. C. Gordon, and T. F. Melham. The PROSPER toolkit. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 78–92, 2000.
12. F. Hanna and N. Daeche. Specification and verification using higherorder logic. In *Proceedings of the 7th International Conference on Computer Hardware Design Languages*, pages 418–433, 1985.
13. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.
14. C. Jacobi. *Formal Verification of a fully IEEE compliant Floating Point Unit (Draft)*. PhD thesis, University of Saarland, Computer Science Department, 2001.

15. C. Jacobi and D. Kroening. Proving the correctness of a complete microprocessor. In *Proc. of the 30. Jahrestagung der Gesellschaft für Informatik*. Springer-Verlag, 2000.
16. C. Jacobi and D. Kroening. Proving the correctness of a complete microprocessor. In *GI Jahrestagung 2000*. Springer, 2000.
17. K. Schneider and T. Kropf. Verifying Hardware Correctness by Combining Theorem Proving and Model Checking. Technical Report SFB358-C2-5/95, Institut für Rechnerentwurf und Fehlertoleranz, 1995.
18. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7, 1963.
19. M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.
20. D. Kröning and W. Paul. Automated pipeline design. In *Proc. of 38th Design Automation Conference (DAC)*, 2001.
21. D. Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.
22. M. Srivas, H. Rueß, and D. Cyrluk. Hardware verification using PVS. In *Formal Hardware Verification – Methods and Systems in Comparison*, volume 1287 of *LNCS*. Springer, 1997.
23. S. McKeever and W. Luk. Towards provably-correct hardware compilation tools based on pass separation techniques. In *Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*. Springer, 2001.
24. K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
25. S. M. Müller and W. J. Paul. *Computer Architecture: Complexity and Correctness.* Springer-Verlag, 2000.
26. A. Mycroft and R. Sharp. Hardware synthesis using safl and application to processor design. In *Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*. Springer, 2001.
27. A. Mycroft and R. Sharp. Hardware synthesis using SAFL and application to processor design. In *Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*. Springer, 2001.
28. J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. IA-64 floating point operations and the IEEE standard for binary floating-point arithmetic. *Intel Technology Journal*, Q4, 1999.
29. 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.
30. V. R. Pratt. Anatomy of the pentium bug. In *TAPSOFT'95*, volume 915 of *LNCS*, pages 97–107, Aarhus, Denmark, 1995. Springer-Verlag.
31. D. Russinoff. Specification and verification of gate-level VHDL models of synchronous and asynchronous circuits. Technical report, Computational Logic, Inc., 1994.
32. D. Russinoff. Mechanical verification of register-transfer logic: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer, 2000.
33. S. Finn, M.P. Fourman, M.D. Francis, and B. Harris. Formal system design - interactive synthesis based on computer assisted formal reasoning. In *Intern. Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.
34. R. Schlör. A prover for vhdl-based hardware design. In *International Conference on Computer Hardware Description Languages and Their Applications*, 1995.

35. Xilinx, Inc. *Virtex-E Data Sheet*, 2002. available at `http://www.xilinx.com/partinfo/ds022.htm`.