Towards Verifying VHDL Descriptions of Processors

Laurent Arditi Hélène Collavizza

Université de Nice – Sophia Antipolis, I3S, CNRS-URA 1376
arditi@unice.fr, helen@essi.fr

Abstract

We present a system for the formal verification of processors which combines a computer algebra simplification tool with an object-oriented approach. It has been successfully used for verifying the DP32 processor described in the VHDL Cookbook. A general VHDL description style for proving processors is derived from this case study.

1 Introduction

1.1 Motivations

In order to produce correct circuits, attention has been paid these last ten years on formal verification methods [1]. Formally verifying a circuit consists in proving that its specification is logically equivalent to its implementation. When dealing with complex circuits such as processors, the *specification* of the circuit is a description of an abstract level (such as the assembly language level), while its *implementation* is a description at a more concrete level (for instance, the microinstruction level).

The formal verification then consists to prove the behavioural equivalence between two abstraction levels. More precisely, let $level_i$ be an abstract specification level, and $level_{i+1}$ a more concrete one. An action at $level_i$ is realized at $level_{i+1}$ by composing a set of actions at $level_{i+1}$. So to verify the correctness between two adjacent levels one must show that the following diagram commutes:

$$\begin{tikzcd}
\text{state}_i \arrow[r, "instr_i"] & \text{state}_{i,after} \\
\text{state}_{i+1} \arrow[u, "abstraction"] \arrow[r, "instr^1_{i+1} \circ \ldots \circ instr^n_{i+1}"] & \text{state}_{i+1,after} \arrow[u, "abstraction"]
\end{tikzcd}$$

1.2 Aims of the paper and related works

The first significant work on processor verification was initiated by Gordon[2] and followed by two main case studies: the proofs of Viper with HOL[3] and FM8501 with Nqthm[4].

These examples raised the need for a general methodology of processor verification. A general functional model has first been proposed in [5] for describing processors at abstract levels. More recently, Windley has proposed a methodology based on generic interpreters [6]. An interpreter describes the processor state and behaviour at any abstraction level. The verification consists to prove the equivalence between two interpreters. This approach is generic and is a first step for designing a CAD framework dedicated to processor verification. But in our opinion, these previous works display two kinds of shortcomings:

- they are based on general logical proof tools which are too complex to support efficiently particular problems and require a logical background,
- they do not provide any user-friendly interface to describe processors and verifications to execute.

So we have proposed a new framework for processor verification. The key idea is to combine a "computer algebra system" and an "object-oriented implementation of generic interpreters" in a single framework[7]. The computer algebra system is used to efficiently simplify expressions. The object-oriented approach is well-suited to implement the generic interpreters.

In this paper, we get over another step for integrating formal verification methods into CAD frameworks taking VHDL as a specification interface. The common approach to formally verify VHDL descriptions is to identify a provable VHDL subset and to define a semantics of this subset [8, 9]. Furthermore, some systems provide translators from VHDL to an intermediate form as input to proof tools [10, 11].

We emphasize here that our framework is well-suited to verify some kind of VHDL descriptions of processors, in a very efficient way and with a small specification effort. We exhibit the specification and verification of the DP32 processor [12]. Its verification has derived a general VHDL style to describe processors, that our system is able to prove.

Layout of the paper: in section 2, we introduce our verification method on a simple example. In section 3, we give our general specification and verification methodology. Section 4 presents our implementation prototype. In section 5 we detail the proof of the DP32 processor. We last conclude in section 6.

2 Overview of our method

Before discussing the technical details of our method, let us introduce it on a simple example.

Suppose we are given a simple processor and we wish to prove the "addition" instruction in direct memory addressing mode. We assume that this processor is described in VHDL at the assembly language level (level$_1$) and at the microinstruction level (level$_2$).

At the assembly language level, the state of the processor consists of the memory `mem`, the accumulator `acc` and the program counter `pc`. The whole VHDL description consists of a process with a state variable declarative part, and a body part that describes the processor cycle: fetch an instruction and execute a set of assignments. The addition instruction in direct memory addressing mode involves the two assignments T_1 and T_2:

```
acc := add(acc,                                    -- T1
          mem(concat('0000',mem(pc)(7 downto 4))));
pc := add(pc, '00000001');                         -- T2
```

In our object-oriented framework, this is described as an object, "proc-level$_1$", with the following attributes (see part 3.1):

- `state` contains the tuple <`mem`,`pc`,`acc`> where each component is built up from an object-oriented library: `mem` is an instance of the `Memory` class while `pc` and `acc` are instances of the `Register` class.
- `transitions` describes the instruction semantics,
- `select` is a function that selects the instruction in memory at address `pc`.

At the microinstruction level, a more concrete state is considered. It includes the assembly language level state plus the instruction register `ir`, the operand register `rop` and the current microinstruction pointer `mpc`. The addition instruction is executed by a sequence of microinstructions M_0, M_1 and M_2:

```
case mpc is
 when 0 => ir := mem(pc);                          --M0
          mpc := 1;
 when 1 => rop := concat('0000', ir(7 downto 4)); --M1
          pc  := add(pc, '00000001');
          mpc := ...;
 when 2 => acc := add(acc, mem(rop));              --M2
          mpc := 0; ...
```

In our framework, the processor at the microinstruction level is also described as an object "proc-level$_2$":

- `state` is <`mem`,`pc acc`,`ir`,`rop`,`mpc`>,
- `transitions` defines semantics of microinstructions,
- `select` selects the microinstruction that is in the microprogram memory at address `mpc`.

In order to perform the proof, between $level_1$ and $level_2$, we must show that the sequence M_0, M_1, M_2 correctly realizes the transitions T_1 and T_2. The proof is specified by an object that has four attributes (see part 3.1):

- `specification` points the object "proc-level$_1$",
- `implementation` points the object "proc-level$_2$",
- `abstraction` defines the state abstraction from <`mem`,`pc`,`acc`,`ir`,`rop`,`mpc`> to <`mem`,`pc`,`acc`>.
- `sync` defines the temporal abstraction. Since a microinstruction sequence begins at the address 0, the predicate `sync` is here `mpc = 0`.

The proof is performed as explained in 3.2. In this simple case, it does not require any simplification and the state values obtained at the two levels are exactly the same. However, for more complex processors this can involve reducing expressions on bit vectors.

The above specification and proof process will be the same for any processor at any abstraction level.

3 Specification and proof methodology

In this section, we give our general specification and proof methodology, that follows the approach of generic interpreters of [6].

3.1 Specification process

One important point in processor verification is that the specification process must be reusable for all processors of a same category. In fact, we need *horizontal* genericity (the model should be reusable to specify different processors) and *vertical* genericity (at each abstraction level, the same model and proof method should be reused).

Processor specification. Any processor at any level is defined as an `Interpreter` that has the three attributes below:

- `state`: the set of all processor components visible at the current level.
- `transitions`: a set of state transitions, indexed by keys, that defines the behaviour of the processor, and
- `select`: a function from `state` that returns the key of the current transition selected by the control part.

This ensures the genericity of the description.

Proof specification. In order to ensure the genericity of the proof, a verification between two specification levels is a `Proof`, that links two interpreters. A `Proof` has two attributes pointing the two interpreters and two others describing the abstraction functions:

- `specification`: specification interpreter,
- `implementation`: implementation interpreter.

- `abstraction`: it defines the state abstraction from the implementation to the specification state.
- `sync` defines the temporal abstraction. Its value is *true* only when the two levels are synchronized. This predicate is usually defined as a test on the implementation program counter.

3.2 Proof process

Assume we have defined two `Interpreters` that specify a processor at $level_i$ and $level_j$, and one `Proof`, that links these two interpreters. To perform the proof we have to execute a transition at $level_i$ and a set of transitions at $level_j$. The specification is correct if $level_i$ final state is an abstraction of $level_j$ final state when `sync` is true. Then, the correctness proof is performed according to the following steps:

```
1:  $S_j$ := init($state_j$)
2:  $S_i$ := abstraction($S_j$)
3:  $trans$ := transitions(select($S_i$))
    $S_i$ := exec-trans($trans$, $S_i$)
4:  repeat
        $trans$ := transitions(select($S_j$))
        $S_j$ := exec-trans($trans$, $S_j$)
    until sync($S_j$)=true
5:  verify that   $S_i \equiv$ abstraction($S_j$)
```

Broadly speaking, points 1 and 2 compute initial states using symbolic values. Points 3 and 4 perform the state transitions, "`exec-trans`" carries out the transitions while simplifying expressions (see part 4.2). Point 5 consists in a syntactic checking of the equality of the two expressions that have first been simplified by our computer algebra system (see part 4.3). Syntactic checking is generally sufficient for instructions that do not involve loops.

4 Framework implementation

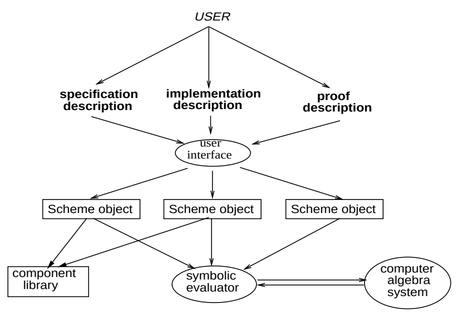

Figure 1: **Overview of the framework**

Our object-oriented framework (see figure 1) is realized using the functional and object-oriented language STk [13]. Processors are described through a textual user-friendly interface and a parser automatically generates the Scheme functions.

4.1 An object-oriented description of interpreters

A straightforward way to implement interpreters and proofs is to follow an object-oriented approach. This answers the need for genericity and reusability. So we have defined a class hierarchy for `Interpreters` and `Proofs` and a component library[7].

State components are built by inheriting from one structural class (a data type) and one behavioural class ("stored" or "instantaneous" kind [14]). Figure 2 shows a simplified class hierarchy.

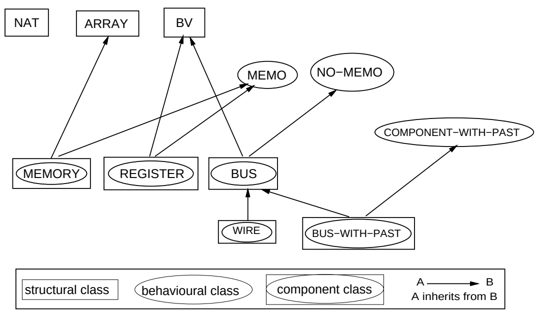

Figure 2: **Hierarchy of component classes**

4.2 A symbolic evaluator of transitions

The key to the proof algorithm described in part 3.2 is the symbolic execution of transitions. Assume we are given a transition $dest \leftarrow source$, it is executed as follows:

- call on `read` method to get the value of *source*,
- call on the simplification system in order to simplify the resulting value if possible,
- call on `write` method to modify the value of *dest*.

The symbolic evaluator takes convenience of the object-oriented approach: this execution process is generic since `read` and `write` are methods associated to all component classes.

4.3 A computer algebra system

In order to prove the equality of the expressions involved in state transitions, we use a simplification system based on a computer algebra approach. Some other simplification systems use rewriting techniques[15, 16]. We choose the computer algebra point of view which is much more efficient since the simplification of an expression is driven by its head operator. This system is less powerful than general provers as HOL or Nqthm, but is particularly well adapted for this specific problem where expressions to be proved are simple but numerous.

Having shown the structure of our system, we are now ready to exhibit how it can be used to verify processors described in VHDL.

5 Case study of the DP32 processor

In this section we consider the case study of the DP32 processor described in [12] and derive general remarks on the VHDL style we are able to prove.

The DP32 is specified in [12] at two levels: the "behavioural description" level (i.e assembly language level), and the "register transfer architecture" level. We follow the specification plan of the VHDL description except for the time modeling (see 5.2).

5.1 Behavioural description

The VHDL behavioural description is a single process. Its declarative part defines the set of visible components, and its body defines the fetch sequence and the instruction semantics. This is translated into an interpreter without any restrictive modification. The declarative part forms the `state` attribute of the interpreter, while the process body is decomposed into the `transitions` attribute and the `select` attribute.

Declarative part. In VHDL, processor components are variables of types arrays of bit-vectors, bit-vectors, bits and natural numbers (these components are stored variables). They are represented by instances of our component classes.

Here is an extract of the VHDL declarative part[1]:

```
    variable reg: array (natural) of bit_32;
    variable PC: bit_32;
```
and its translation in Scheme:
```
 ( (make ARRAY :size 32 :name 'reg)
   (make REGISTER :size 32 :name 'PC) )
```

The `state` attribute of the interpreter is the list of all these instances.

The process body. In VHDL, the behavioural description is a single process that is decomposed into four main sequential steps: (1) the word at address `PC` is fetched in the instruction register `current_instr`, (2) `PC` is incremented, (3) fields of `current_instr` are assigned to variables, and (4) a case statement over `op` selects the instruction and executes its behaviour. Here is an extract of the VHDL process body:

```
memory_read(PC,true,current_instr);
add(PC, bits_to_int(PC), 1);
op := current_instr(31 downto 24);
r3 := bits_to_natural(current_instr(23 downto 16));...
case op is
   when op_add =>  -- behaviour of the ADD instruction
   when op_sub =>  -- behaviour of the SUB instruction
```

Each instruction behaviour just consists of assignments to components. For example, the behaviour of "add" instruction is:

```
add(reg(r3),bits_to_int(reg(r1)),bits_to_int(reg(r2)));
```

[1]We applied some minor modifications in the original VHDL source and in our Scheme code in order to shorten the extracts.

In our model, this process body is mapped into the attributes `select` and `transitions` of the interpreter: `select` reflects the structure of the VHDL `case` while `transitions` reassemble the instruction behaviours defined in the case statements. The `select` attribute of the interpreter is defined as follows:

```
(if reset
    reset_pseudo_instruction      ; reset sequence
    (case (field (memory_read PC) 24 31)
       (op_add add_instr)
       (op_sub sub_instr) ...))
```

Unlike in VHDL, we do not consider intermediate steps in our model. As a consequence, we do not model temporary variables (`current_instr`, `r3`...) but use the "let" constructor instead. This is more close to the intuitive view of a processor because, at the assembly language level, the instruction register is not usable. Here is an extract of translation for the addition instruction:

```
(let ((current_instr (memory_read PC)))
  (let ((r3 (field current_instr 16 23))
        (r2 (field current_instr 0 7))
        (r1 (field current_instr 8 15)))
    ( PC := (add PC (word32 1)) )
    ( (reg ($ r3)) := (add (reg ($ r1)) (reg ($ r2))))
```

Each instruction is defined by a set of state transitions in the same way as described for "add". The set of all instruction behaviours constitutes the `transitions` attribute of the interpreter. So, mapping the VHDL source into an interpreter is simple for such a single process.

5.2 Time modeling

Before presenting the "register transfer architecture" let us discuss now on time modeling principles, that would be important at this more concrete abstraction level.

Time granularity. In the VHDL description of concrete levels, assignments are always synchronized with the clock and scheduled with a delay that corresponds to the gate propagation delay.

In our interpreter model, which is specialized to functional aspects of processor verification, we do not consider a fine time granularity. At each level, the clock period is implicitly the duration of a state transition: at the assembly language level it corresponds to the duration of an instruction (so we do not consider intermediate steps) and at the architecture level, it corresponds to the real clock of the processor.

Clock phases. In the VHDL description, signal connections are always performed on the first clock phase ϕ_1, while assignments to registers are performed on the second phase: ϕ_2. Roughly speaking, the VHDL description is as follows:

```
wait until $\phi_1$='1';    signal connections;
wait until $\phi_2$='1';    assignments to registers;
```

In our object-oriented model, the distinction between the two clock phases is naturally induced by the class of the component which is assigned: "instantaneous variables" are immediately assigned (i.e. during ϕ_1 in VHDL) while "stored variables" are assigned only at the end of the current cycle and so their results are only visible at the next cycle (i.e. during ϕ_2 in VHDL).

Temporal behaviour of variables. In VHDL, signals keep their values until they change. So in DP32 description from [12], signals that keep the same value as in the previous cycle are not reassigned.

In our processor modeling, registers and memories keep their values until they change but signals, as buses, keep their values only during one clock period. Therefore, we need to reassign them on each cycle. Our approach is closer to the real structure of processors because it better reflects the microcode execution. In fact, on each cycle a microcode word is fetched from the memory by the controller. Bits of this word are connected to signals of the operative part. Therefore they change on each cycle.

Having detailed the time modeling, let us now give the translation of the register transfer architecture.

5.3 Register transfer architecture

The register transfer architecture is a structural description of the processor that follows four steps. It describes behaviour of sub-modules and declares signals and buses. Then it instantiates sub-modules and connects them by these signals. And last, it describes the controller as a single process.

Sub-module and signal descriptions. A "sub-module" is a behavioural description of an entity. It has a name, a port list and a process which describes its behaviour. It represents a component of the architecture.

For example, the Scheme description of the DP32 register file follows:

```
(reg_file_32_rrw                              ; module name
 (a1 q1 en1 a2 q2 en2 a3 d3 en3)              ; ports
 ((reg memory 32))                            ; local variables
 ((if en3 ((reg ($ r3)) := d3)                ; behaviour
  (if en1 ((q1 .= (reg ($ r1))) null)
  (if en2 ((q2 .= (reg ($ r2))) null) ))
```

The different sub-modules are connected by buses inside the operative part, and by signals between the operative part and the controller. These buses and signals are declared at the beginning of the VHDL description. The translation of this declarative part is, as for the previous level, a set of instantiations of component classes (mainly `BUS` and `SIGNAL`).

Structural description. The structure of the processor then consists of instantiations of sub-modules using predeclared buses and signals to connect their ports. For example, the register file is instantiated in VHDL as follows:

```
reg_file: reg_file_32_rrw
  port map(a1 => instr_a1, q1 => op1_bus,...);
```

Our Scheme specification is very similar and the translation in Scheme is automatic.

The controller. Each sub-module of the operative part has ports connected to the controller. At each cycle, the controller provides specific values to these ports, depending on the current microinstruction name (`micro-pc`), and determines the next microinstruction to execute. Here is an extract of the Scheme specification of the controller that reflects the VHDL code.

```
(case micro-pc (fetch_0 ( (reg_port1_en .= "0")
                          (PC_out_en .= "1")
                          (micro-pc := fetch_1) ))
               (fetch_1 ( (read .= "1")
                          (PC_out_en .= "1")
                          (micro-pc := fetch_2) )) ...
```

Having translated sub-modules and signal declarations, the structural description, and the controller, a parser puts this in a single interpreter. This is entirely automatic and consists in flattening the description and renaming ports and internal variables. So we have shown that an interpreter may be generated even from a structural description.

5.4 Proof specification and run

We are now able to prove that the register transfer architecture correctly implements the behavioural description. The proof is described by an object of the `Proof` class:

- its `specification` and `implementation` attributes point the two interpreters.
- the `abstraction` is just a state injection that keeps `pc`, `reg`, `cc`, `mem`, and `reset`.
- The `sync` attribute describes the condition required to synchronize the interpreters. Since the result of an operation is written in the register file during the next instruction fetch, the interpreters are synchronized when `micro-pc = fetch_1` if a write back is required, else when `micro-pc = fetch_0`.

The Scheme code of the proof specification is:

```
(make <Proof>
  :specification  DP32's behavioural description
  :implementation  DP32's RT architecture
  :abstraction '(pc reg cc mem reset)
  :sync '(or (and (= micro_pc fetch_1)
                  (= write_back_pending "1"))
             (and (= micro_pc fetch_0)
                  (= write_back_pending "0"))))
```

Thanks to the genericity of our system for interpreters and proofs, running the proof is entirely automatic. The DP32 proof raised a minor error: the controller does not always wait for the `ready` signal from the memory. This produces an error when a memory operation is longer than the clock period.

6 Conclusion

6.1 VHDL description style we can prove

From the significative case study of DP32 processor, we can now derive the form of a VHDL description of processors that we are able to prove.

At the assembly language level, the processor behaviour is a single process defined by a case on the operation code. Each part of the case defines the state transitions involved by a particular instruction.

At the microinstruction or register transfer architecture level, the description is structural. The behaviour of the processor is described by a state machine, that represents the processor controller. This may be a process or a sub-module defined by a case on the microinstruction name. Each part of the case expresses the transitions that occur on the two clock phases.

As we explained in part 5.2, the main restriction we make is on time model. We only consider systems which are synchronized by a global clock and, we ignore time delays which are smaller than the global clock period. We also make an important restriction on sequentiality which is only implicitely determined by the clock cycle.

6.2 Other results and future works

We have presented an object-oriented framework for processor specification and verification. It has been used to verify several microprocessors without any change to the class hierarchy (see Table 1). The specifications are concise and systematic, and Scheme expressions are built from a textual interface. The verification process is generic, automatic and proof times are very satisfactory compared with other methods.

We have shown in this paper that this system may be used to verify some kind of VHDL descriptions. We have now to precisely identify a VHDL subset that is sufficient to model processors in a style that can be proved by our system, and to implement a translator from this subset to our textual interface.

Microprocessor	SS (pages)	PT (seconds)
AVM-1	22 (100 in [6])	1775 (58 hours in [6])
DP32	19	470
Proc. in [14].	4	183
Tamarack-3	3 (17 in [15])	197 (10 days in [15])

Table 1: **Microprocessors we specified and proved.** SS: specification code size, PT: proof time on a SUN IPC workstation.

References

[1] A. Gupta, "Formal hardware verification methods: a survey," *Formal Methods in System Design*, vol. 1, pp. 151–238, Oct. 1992.

[2] M. Gordon, "HOL, a machine oriented formulation of higher order logic," Tech. Rep. 68, University of Cambridge, Computer Laboratory, 1985.

[3] A. Cohn, "A proof of correctness of the Viper microprocessor: the first level," in *VLSI Specification, Verification and Synthesis*, (Calgary), Jan. 1987.

[4] W. A. Hunt Jr., "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, Dec. 1989.

[5] D. Borrione, P. Camurati, J. Paillet, and P. Prinetto, "A functional approach to formal hardware verification: The MTI experience," in *ICCD'88*, (Port Chester, New-York), Oct. 1988.

[6] P. J. Windley, *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Division of Computer Science, 1990.

[7] L. Arditi and H. Collavizza, "An object-oriented framework for the formal verification of processors," in *ECOOP'95*, (Aarhus, Denmark), Aug. 1995.

[8] J. P. van Tassel, *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant*. PhD thesis, Univ. of Cambridge, July 1993.

[9] P. T. Breuer, L. S. Fernandez, and C. Delgado Kloos, "Clean formal semantics for VHDL," in *European Design Automation Conference*, (Paris, France), 1994.

[10] D. Borrione, L. Pierre, and A. Salem, "Formal verification of VHDL descriptions in the PREVAIL environment," *IEEE Design and Test of Computers*, vol. 9, pp. 42–56, June 1992.

[11] F. Nicoli and L. Pierre, "From VHDL to formal verification," in *EURO-VHDL Conference*, (Grenoble, France), Sept. 1994.

[12] P. J. Ashenden, *The VHDL Cookbook*. Dept. Computer Science, Univ. of Adelaide, July 1990.

[13] E. Gallesio, "STklos: a Scheme object oriented system dealing with the TK toolkit," in *Xhibition 94*, (San Jose), ICS, Jul. 1994.

[14] F. Anceau, *The Architecture of Microprocessors*. Addison-Wesley Publishing Company, 1986.

[15] V. Stavridou, *Formal Methods in Circuit Design*. No. 37 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1993.

[16] M. Allemand, "A rewriting based method for the formal verification of microprocessors," in *CHDL'93*, (Ottawa), Apr. 1993.