N94-18363

Verification of VLSI Designs

P. J. Windley
Department of Computer Science
University of Idaho
Moscow, ID 83843
m 208.885.6501

Abstract: **In this paper we explore the specification and verification of VLSI designs. The paper focuses on abstract specification and verification of functionality using mathematical logic as opposed to low–level boolean equivalence verification such as that done using BDDs and Model Checking. Specification and verification, sometimes called *formal methods*, is one tool for increasing computer dependability in the face of an exponentially increasing testing effort.**

1 Introduction

Reliable computer systems are becoming increasingly difficult to engineer. The successes of IC fabrication technology have put VLSI engineers in the position of building dependable computers that are orders of magnitude more complex than the largest computers of even a decade ago. With even larger numbers of transistors promised in the near term, research is being done to make the reliable engineering of complex VLSI designs practical.

There are two complimentary approaches to computer reliability: fault tolerance and fault exclusion. The former is most useful in handling dynamic faults occurring during system operation due to component failure or other unexpected events. The latter is a static process intended to remove errors in design and implementation before the computer system is in service.

Testing and simulation are well–known fault exclusion techniques. Testing and simulation are used extensively in the design, implementation, and manufacturing of computer systems. The problem is that testing and simulation can never exhaustively cover every possible situation that the circuit might encounter. Pygott [13] states

> "A comparatively simple 8–bit microprocessor such as the Z80 has 208 internal memory elements and 13 input signals, meaning that the circuit is capable of 2^{221} different state transitions. Even if a transition could be simulated every microsecond, it would take 10^{53} years to examine all the possible changes (this is far larger than the age of the universe)."

Clearly, only a tiny fraction of the possible state transitions can be tested. This situation has led to VLSI devices going to market with design faults which were not caught in testing.

One possible answer to the inadequacies of testing and simulation is hardware synthesis from high–level circuit descriptions written in an appropriate hardware description language (HDL) such as VHDL [7]. Synthesis from an HDL description certainly has much promise. Textual descriptions are easy to store, manipulate, and process. Also, synthesis tools are likely to be reliable since the social process of hundreds of users using a synthesis program tends to exorcise any latent bugs.

Unfortunately, synthesis of VHDL circuit descriptions is not sufficient for dependable computing. As a case in point, consider that high–level programming languages have been in use for 20 years and programs still contain numerous errors. There are a number of reasons why this is so:

1. HDLs are generally verbose making them hard to read.

2. HDL constructs are not usually amenable to formal analysis. Thus it is nearly impossible to show that a particular description has desired properties.

3. Constructs that can be synthesized are frequently not abstract enough to be of use as system specifications.

4. Contrary to what the marketers of synthesis systems would have one believe, circuit descriptions outside of a small subset of a HDL cannot be synthesized. This is particularly true of abstract descriptions. One need not search further than a multiplier to find an example of this.

Because of these limitations in testing, simulation, and synthesis, much effort is being expended in the formal specification and verification of hardware. Formal methods offer hope of overcoming some of these shortcomings because they are based on logic and can thus take advantage of the decades of mathematical research on using logical analysis.

1. Logical circuit descriptions are often more concise than conventional HDL descriptions.

2. Numerous formalisms can be embedded in logic. This allows the circuit specifier to use the most appropriate formalism for the job [10].

3. One can prove properties about logic descriptions directly using a proof system such as predicate calculus. This can be very effective for establishing that a specification meets its requirements [17].

4. Analysis can be applied to the specification and less–abstract structural circuit descriptions to show functional correctness [2,9,16].

5. Logic provides behavioral, structural, data, and temporal abstraction mechanisms for reducing the complexity of the description [12].

For these and other reasons, we believe that formal methods can play an important part in increasing the reliability of computer systems.

Note that we are not suggesting that formal methods *replace* testing, simulation, and synthesis, but rather that they *complement* these techniques. Figure 1 shows an idealized ASIC design process (adapted from [11]). The *RTL circuit description* is written in an

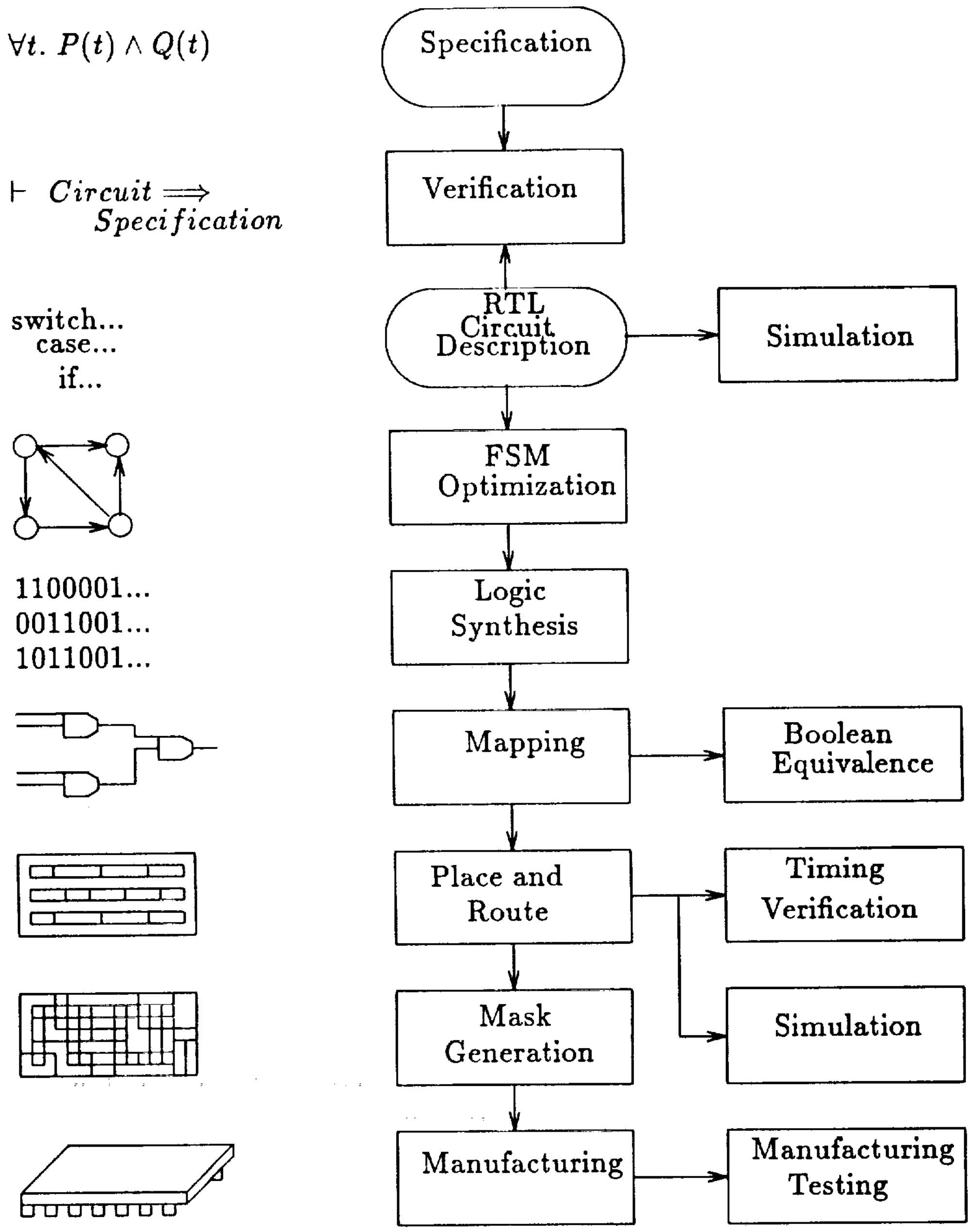

Figure 1: The ASIC Design Process.

appropriate HDL and subjected to synthesis and simulation. The *specification* is a more abstract declarative description which is subject to formal analysis.

Having described the benefits of formal methods, there are a number of directions that we could take. We could, for example, focus on how formal methods interacts with conventional CAD tools or discuss the pros and cons of the design process in Figure 1. Instead, we will show how logic can be used to specify circuits and how proof functions as a mathematical analysis tool for reasoning about those specifications. We do not attempt to give a complete survey of the field, but rather focus on a demonstration of techniques.

2 Using Logic to Specify Hardware.

A circuit is a collection of devices composed by interconnection. Each of these devices has ports which are used for input, output, or both. The behavior of a device can be expressed in terms of its ports. Each of the devices in a circuit can, in turn, be viewed as a composition of still other devices. This hierarchy of devices eventually leads to the devices that the designer considers primitive. The smallest devices we will deal with in this paper are logic gates and indeed, in many cases, we will stop much higher than even gates.

Clocksin describes several ways to specify circuit structure [3]:

- We can use imperative declarations of the circuit structure (this is referred to as the extensional method).
- We can use functions to describe the output in terms of the input.
- We can use predicates in a quantified logic to relate the ports of a device using behavioral or structural constraints.

Each of these methods has advantages and disadvantages. The extensional method has the advantage of being familiar to designers since it resembles imperative languages such as Pascal that most designers have used. Most modern hardware descriptions languages (e.g. VHDL) use the extensional method. The largest disadvantage of the extensional method is that it is difficult to treat formally, just as imperative programming languages are hard to treat formally.

The functional model is widely used; Hunt's specification of the FM8501 microprocessor, for example, is functional [6]. To specify the behavior of sequential circuits functionally, the specification language must support recursion. Hunt uses recursion to describe the sequential operation of his CPU.

In the functional model, circuit interconnection is given by the syntactic structure of function application. This can cause several problems:

- Describing circuits with bi-directional ports is difficult since functional specifications differentiate between input and output syntactically.
- The purpose of a structural specification is to show how components are connected together. Since the only means of expressing connection is function application, even returning a tuple is insufficient for describing circuits with more than one output.

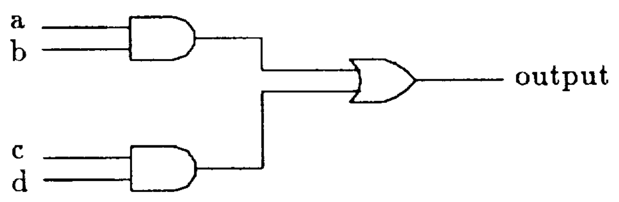

Figure 2: Implementation of a simple circuit, D

- Sequential circuits feedback on themselves. Recursion is the best alternative; but that can be inadequate for circuits with multiple feedback paths.

The predicate method is a widely used specification technique [5] and is the one we will demonstrate in this paper. A disadvantage of the predicate method is that designers are likely to find it the most unfamiliar of the three and thus difficult to use. In addition, to use the predicate method, the logic must support existential quantification, either explicitly or implicitly. (Prolog is an example of a language with implicit existential quantification.) The predicate method does, however lend itself to a wide variety of circuit types, including those with multiple outputs and bi-directional ports.

2.1 Specifying Circuits with Predicates.

As an example of the predicate model, we will specify the behavior and structure of a very simple circuit we call D. The predicate that specifies the behavior of the circuit can be given by the following logic definition:

$\vdash_{def}$ `D(a,b,c,d,out) = out = (a` $\wedge$ `b)` $\vee$ `(c` $\wedge$ `d)`

Notice that the inputs and outputs are all included in the arguments and the behavior is expressed as a constraint among the outputs and the inputs.

One possible implementation for D is shown in Figure 2. As was mentioned earlier, each device can be thought of as representing a constraint on its inputs and outputs. For example, the top *And* gate constrains a, b, and p in a manner consistent with the behavior of the device.

$\vdash_{def}$ `And(a, b, p) = (p = a` $\wedge$ `b)`

To get the constraint represented by the entire device, we can compose the individual constraints using conjunction.

`And(a, b, p)` $\wedge$ `And(c, d, q)` $\wedge$ `Or(p, q, out)`

This expression constrains the values not only on the ports of the device, a, b, c, d, and out, but also on the internal lines p and q. We normally wish to regard such a device as a "blackbox" and consequently are only interested in the values of the external lines. We can hide the internal lines using existentially quantified variables and define a predicate D_imp that represents the structure of the circuit.

```
⊢def D_imp(a, b, c, d, out) =
     ∃ p q.  And(a, b, p) ∧ And(c, d, q) ∧ Or(p, q, out)
```

While this formula looks confusing at first, we should note that this level of specification can be produced automatically from netlists or traditional HDL models.

For comparison, the following specification describes the same circuit using functions:

```
⊢def D(a,b,c,d) =  Or(And(a,b),And(c,d))
```

The outputs are not mentioned explicitly; the result of the function is taken to be the output of the circuit.

Similarly, we can write a extensional specification of the circuit in a hardware description language such as VHDL [1]:

```
Entity D_imp is
   port(a, b, c, d :in Bit; outp :out Bit);
end D_imp;

architecture Structure of D_imp is
   component ANDGate port(i1,i2:in Bit; outp :out Bit);
   component ORGate port(i1,i2:in Bit; outp :out Bit);
   signal p, q: Bit

   G1: ANDGate port map (a, b, p);
   G2: ANDGate port map (c, d, q);
   G3: ORGate port map (p, q, outp);

end Structure;
```

The difference between this specification and the predicate model of the circuit structure is largely superficial. The primary difference is the abundance of keywords in the extensional specification. The biggest impediment to using specification languages such as VHDL is that they sometimes lack a clear semantics. This problem can be overcome by defining a semantics of the specification language in the object language of a verification tool such as HOL. Van Tassel has done just that using VHDL and HOL in [14,15].

2.2 Specifying Sequential Behavior.

The last section specified a simple combinatorial circuit. We specify the behavior of sequential circuits in higher-order logic using an explicit representation of time.

For example, we can specify the behavior of a simple latch as follows:

```
⊢def latch in out set = ∀ t. out (t+1) = set t → in t | out t
```

In the specification, in, out, and set are functions of time. The value of a signal at time t is returned when the function representing the signal is applied to t. The specification says that the value of out at time $t+1$ gets the value of the input port, in, at time t if the **set** line is high and remains unchanged otherwise. Universal quantification over time is used in defining the predicate.

We can also use existential quantification to describe temporal operators. For example, suppose that we wish to define a predicate that says that a signal will *eventually* go high. The following is a definition of an EVENTUALLY operator:

```
⊢def EVENTUALLY d t1 = ∃ t2.  t2 > t1 ∧ d t2
```

When applied to the signal d, and the current time, t1, the predicate states that there exists a time, t2, in the future when the signal d will be true. The use of existential quantification over time is also used to specify the behavior of asynchronous interconnections between devices. Joyce [9] has shown how temporal logic can be embedded in higher-order logic.

2.3 Behavioral Abstraction and Specification.

There are many ways of specifying the same circuit. For example, in specifying a two input binary decoder, one might write:

```
⊢def decoder_spec s0 s1 o0 o1 o2 o3 =
        (o0 = (s1 → (s0 → F | F) | (s0 → F | T))) ∧
        (o1 = (s1 → (s0 → F | F) | (s0 → T | F))) ∧
        (o2 = (s1 → (s0 → F | T) | (s0 → F | F))) ∧
        (o3 = (s1 → (s0 → T | F) | (s0 → F | F)))
```

While this specification is correct, its meaning is not very clear.

Here is another specification for the same behavior:

```
⊢def decoder_spec s0 s1 o0 o1 o2 o3 =
        (o0 = ¬s1 ∧ ¬s0) ∧
        (o1 = ¬s1 ∧  s0) ∧
        (o2 =  s1 ∧ ¬s0) ∧
        (o3 =  s1 ∧  s0)
```

This specification closely models one possible implementation for the circuit; consequently, using it as the behavioral specification would make the verification easier, but would not tell us much about the abstract behavior of the decoder.

The next specification is more abstract and says more about the behavior of the decoder:

```
⊢def decoder_spec s0 s1 o0 o1 o2 o3 =
        (o0 ↔ ((s1,s0) = (F,F))) ∧
        (o1 ↔ ((s1,s0) = (F,T))) ∧
        (o2 ↔ ((s1,s0) = (T,F))) ∧
        (o3 ↔ ((s1,s0) = (T,T)))
```

This specification clearly shows the binary numbers being represented by the inputs. Moreover, the specification does not suggest any particular implementation. In general, the more abstract a specification, the easier it is to understand, but more difficult it is to verify.

We can make the above specification even more abstract by defining a function, **pairval**, that converts boolean pairs into numbers and then writing the specification as follows.

```
decoder_spec s0 s1 o0 o1 o2 o3 =
        let n = pairval(s1,s0) in
        (o0 ↔ (n = 0)) ∧
        (o1 ↔ (n = 1)) ∧
        (o2 ↔ (n = 2)) ∧
        (o3 ↔ (n = 3))
```

This specification can be readily generalized to have n inputs and 2^n outputs.

2.4 Specifying a Microprocessor

So far, the circuits we have described have been simple, for expository purposes. One should not assume that all specifications must be of small devices. Indeed, logic is most useful when used on large, abstract specifications. To demonstrate the use of formal specification on a larger example, we will present the specification of a small microprocessor called Tamarack.

There have been numerous efforts to verify microprocessors [4,8,6]. Most of these have used the same implicit behavioral model. In general, the model uses a state transition system to describe the microprocessor. A microprocessor specification has four important parts:

1. A representation of the state, **S**. This representation varies depending on the verification system being used.

2. A set cf state transition functions, **J**, denoting the behavior of the individual instructions of the microprocessor. Each of these functions takes the state defined in step (1) as an argument and returns the state updated in some meaningful way.

3. A selection function, **N**, that selects a function from the set **J** according to the current state.

4. A predicate, **I**, relating the state at time $t+1$ to the state at time t by means of **J** and **N**.

In some cases, the individual state transition functions, **J**, and the selection function, **N**, are combined to form one large state transition function.

To make all of this mode concrete consider the top-level specification of Tamarack presented by Joyce in [9].

```
⊢def TamarackBeh (ireq, mem, pc, acc, rtn, iack) =
      ∀t:time.
          (mem (t+1),pc (t+1),acc (t+1),rtn (t+1),iack (t+1)) =
          NextState (ireq t,mem t,pc t,acc t,rtn t,iack t)
```

The top-level specification relates the state of the assembly language level registers at time $t+1$ to their state at time t using the function `NextState`. The level of abstraction in the top-level specification is roughly that found in an assembly language reference manual. The difference is that the formal specification is less ambiguous and more complete.

The next state function chooses among the many individual instructions according to a selection criteria which describes, in an abstract way, instruction decoding:

```
⊢def NextState (ireq, mem, pc, acc, rtn, iack) =
       let opcval = OpcVal (mem,pc) in
       ((ireq ∧ ¬iack)      →  IRQ_SEM (mem,pc,acc,rtn,iack) |
        (opcval = JZR_OPC) →  JZR_SEM (mem,pc,acc,rtn,iack) |
        (opcval = JMP_OPC) →  JMP_SEM (mem,pc,acc,rtn,iack) |
        (opcval = ADD_OPC) →  ADD_SEM (mem,pc,acc,rtn,iack) |
        (opcval = SUB_OPC) →  SUB_SEM (mem,pc,acc,rtn,iack) |
        (opcval = LDA_OPC) →  LDA_SEM (mem,pc,acc,rtn,iack) |
        (opcval = STA_OPC) →  STA_SEM (mem,pc,acc,rtn,iack) |
        (opcval = RFI_OPC) →  RFI_SEM (mem,pc,acc,rtn,iack) |
                             NOP_SEM (mem,pc,acc,rtn,iack))
```

Each of the instructions available to the programmer as well as actions that take place on instruction boundaries such as interrupts are defined using a function on the state and environment variables that returns a new state updated as appropriate for the instruction being specified. We use the ADD instruction as an example:

```
⊢def ADD_SEM (mem:*memory,pc:*wordn,acc:*wordn,rtn:*wordn,iack:bool) =
       let inst = fetch (mem,(address pc)) in
       let operand = fetch (mem,(address inst)) in
       (mem, inc pc, add(acc,operand), rtn, iack)
```

This instruction increments the program counter and stores the result of adding the accumulator to the contents of memory pointed to by the current instruction in the accumulator. No other state changes occur.

There are at least three kinds of abstraction taking place between the register transfer level (RTL) description of Tamarack and the top-level specification given above.

1. **Behavioral Abstraction** — The RTL description of Tamarack is a structural model that says how the major blocks are connected. The top-level specification says nothing about the structure of the microprocessors, but rather describes the required behavior.

2. **Data Abstraction** — The RTL description contains registers that are not of interest in the top-level specification. A good example of these types of registers is the instruction register which is vital to the correct functioning of the microprocessor, but is not considered in the top-level specification.

3. **Temporal Abstraction** — Events at the RTL level happen at a much finer time granularity than events at the top-level. Events at the top-level are measured on a time-scale that coincides with the execution of macro-level instructions. Events at the RTL level are measured by the sub-cycle clock. Many RTL level events must take place to cause one top-level event to happen.

3 Using Proof to Analyze Specifications

Proof can be used in at least two ways to analyze specifications. The first methods asks the question *Is my specification correct?* The second methods asks the question *Does my implementation meets the specification?*

3.1 Design Verification

Determining whether or not a specification is correct is not a question that can be subjected to exhaustive mathematical analysis since the design is an intellectual artifact, not a mathematical one. We can, however, determine whether a specification meets its requirements to the extent that those requirements can be formulated in logic.

An example of this is the verification of two important properties of the supervisory mode of a microprocessor called *AVM-1* [17]. *AVM-1* has a supervisory mode that is controlled by the supervisory mode bit in a register called the program status word (PSW). When the processor is in supervisory mode, certain registers in the register file (which does *not* include the PSW) become writable. Otherwise they can only be read.

One of the design requirements can be stated informally as follows:

Property 1 (Integrity of Privileged Registers) *If the CPU is not in supervisory mode and the next instruction is not an external or user-generated interrupt, then every privileged register remains unchanged.*

The integrity of the privileged registers is only important at the assembly language programmer's level of the CPU. We do not care if the registers change on a finer time scale so long as they remain the same when viewed by the outside world.

The formalization of this requirement is not difficult. The following expression captures the essence of the problem:

```
∀ n . (IS_SUP_REG n) ⇒
      (EL n (macro_reg (t+1)) =
               (EL n (macro_reg t)))))
```

The expression states that the register file (represented by a list) is the same for every supervisory mode register at time $t + 1$ as it was at time t. [1]

The basic requirement, stated above, must follow from the definition of the top-level of *AVM-1* (`AVM_Beh`) and is subject to the following conditions:

1. The CPU is not currently in supervisory mode (expressed as `¬get_sm (psw t)`).
2. The next instruction is not an internal or external interrupt (expressed in the specification as `¬(Opcode ...= INT_OPCODE)` and `¬(Opcode ...= EINT_OPCODE)`.

[1] EL selects the n^{th} member of a list.

```
⊢ AVM_Beh
    (λ t. (reg_list t,psw t,pc t,mem t,ivec t))
    (λ t. (ireq_e t)) ⇒
  (∀ t.
    ¬get_sm (psw t) ∧
    ¬(Opcode (reg_list t,psw t,pc t,mem t,ivec t)
            (ireq_e t) = INT_OPCODE) ∧
    ¬(Opcode (reg_list t,psw t,pc t,mem t,ivec t)
            (ireq_e t) = EINT_OPCODE ) ⇒
    (∀n. IS_SUP_REG n ⇒
         (EL n(reg_list(t + 1)) = EL n(reg_list t))))
```

This theorem is not difficult to establish and, when combined with a correctness proof (see Section 3.2), gives confidence that the supervisory mode works as it should.

3.2 Functional Verification

A second, and complimentary, use of proof is in showing that our specification is correctly implemented by the structure that we have chosen for the RTL model.

A simple example is given by the circuit D specified in Section 2.1. To show that the implementation (represented by D_imp) meets its specification (represented by D), we prove the following theorem:

```
⊢ ∀ a b c d out . D_imp(a,b,c,d,out) ⇒  D(a,b,c,d,out)
```

This theorem could be proven using any number of techniques. Indeed, while it is a simple example, it has little to do with the kinds of proofs of correctness that occur most frequently or that are the most interesting.

A more interesting example is given in the proof of correctness of Tamarack [9] since the proof involves behavioral, data, and temporal abstraction. We have already seen the specification of the top-level of Tamarack (see Section 2.4). The RTL model is a fairly large, but conventional description of the large grain structure of the microprocessor.

In order to understand the correctness theorem, we must describe the temporal abstraction that takes place between the RTL model and the top-level behavioral description. As we have already mentioned, different levels in the specification have different views of time. We use temporal abstraction to produce a function that maps time at one level to time at another. Figure 3 shows a temporal abstraction function $\mathcal{F}$. The circles represent clock ticks. Note that the number of clock ticks required at the bottom-level to produce one clock tick at the top-level is irregular.

The predicate, $\mathcal{G}$, is true whenever there is a valid abstraction from the lower level to the upper level. We can define a generic temporal abstraction function in terms of $\mathcal{G}$. In a microprocessor specification, $\mathcal{G}$ is usually a predicate indicating when the lower level machine is at the beginning of its cycle—a condition that is easy to test.

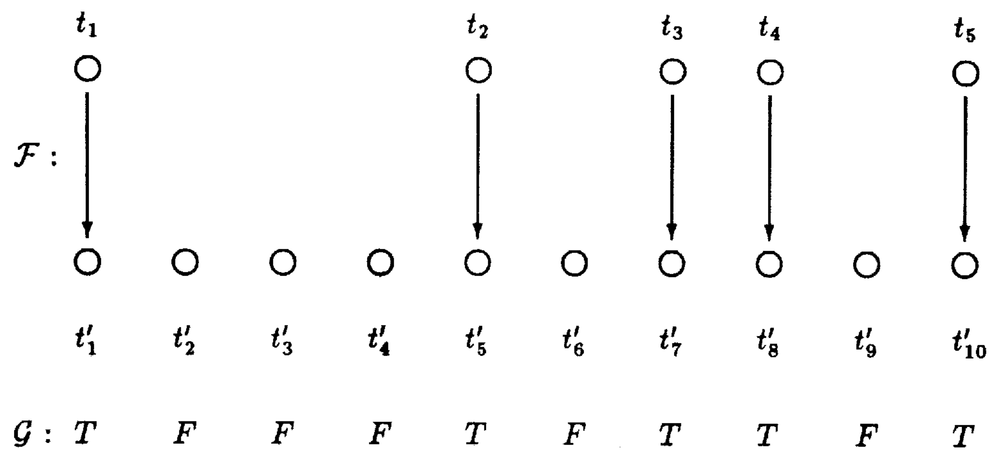

Figure 3: The function $\mathcal{F}$, which maps time at one level to another, can be defined in terms of a predicate, $\mathcal{G}$, which is true only when the mapping occurs.

We will use a function `TimeOf` as our temporal abstraction function. The function is defined recursively so that `(TimeOf g 0)` is the first time that the predicate g is true and `(TimeOf g (n+1))` is the next time after time n when g is true. We will not develop the details of the temporal abstraction function here, but refer the interested reader to [9].

The final correctness theorem for Tamarack states that the behavioral model (defined by `TamarackBeh`) follows from a system (`AsynSystem`) composed of the RTL model and an asynchronous memory subsystem.

```
⊢ AsynSystem (idreq,mpc,mar,pc,acc,ir,rtn,arg,buf,idack,dack,mem) ∧
   ((val4 o mpc) 0 = 0)
   ⇒
   let f = TimeOf (((val4 rep) o mpc) Eq 0) and (not dack)) in
   TamarackBeh (idreq o f,mem o f,pc o f,acc o f,rtn o f,idack o f)
```

The function `f` is the function $\mathcal{F}$ of Figure 3. We also have a reset condition that requires that the value of the microprogram counter, `mpc`, be 0 at time 0.

Presenting the proof of the correctness theorem for Tamarack is beyond the scope of this paper. The proof is actually quite straightforward in most cases, involving standard proof techniques such as substitution, case analysis, and induction. Indeed, much of the difficulty is caused by the size of the proof effort rather than the puzzling nature of the theorems. Tamarack is, of course, far from being the largest device with a verified correctness. Recent research has developed techniques for managing much of the complexity of proofs of this sort [16]. The techniques are demonstrated in the proof of correctness of *AVM-1* .

One should not, of course, accept that the microprocessor is correct simply because there is a theorem. The idea is that proof constitutes engineering analysis and like an engineering analysis, must be documented and subject to review. What we have presented here is not, of course, an engineering analysis.

4 Conclusions

This paper has shown how logic can be used to specify and analyze hardware designs. The use of formal methods has a number of advantages.

- Specifications give a clear and precise statement of the intended behavior of a design.
- Specifications can be analyzed to determine whether or not they meet the requirements of the design.
- Functional correctness can be demonstrated through analysis rather than testing.
- Assumptions are made explicit.

We do not suggest that formal methods replace conventional engineering practices, but augment them. Work is continuing to bring tools based on formal methods into the designers toolbox:

- We are developing new high–level models of common hardware devices which guide the specification and verification of those devices.
- We are writing translators between hardware description languages used by conventional CAD tools and verification tools.
- We are doing case studies to serve as examples of specification and verification.

These efforts, and similar efforts at other institutions promise to make formal methods tractable for large–scale use in VLSI design.

References

[1] J. R. Armstrong. *Chip–Level Modeling with VHDL.* Prentice Hall, 1989.

[2] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs.* Elsevier Scientific Publishers, 1987.

[3] W. F. Clocksin. Logic programming and digital circuit analysis. *The Journal of Logic Programming*, 4:59–82, 1987.

[4] A. Cohn. Correctness properties of the VIPER block model: The second level. Technical Report 134, University of Cambridge Computer Laboratory, May 1988.

[5] M. J. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. Elsevier Scientific Publishers, 1986.

[6] W. A. Hunt. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. Elsevier Scientific Publishers, 1987.

[7] IEEE Std 1076-1987. *IEEE Standard VHDL Language Reference Manual*, 1987.

[8] J. J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Press, 1988.

[9] J. J. Joyce. *Multi-Level Verification of Microprocessor-Based Systems*. PhD thesis, Cambridge University, December 1989.

[10] J. J. Joyce. More reasons why higher-order logic is a good formalism for specifying and verifying hardware. In *Proceedings of the ACM/SIGDA International Workshop in Formal Methods in VLSI Design*, January 1991.

[11] K. Keutzer. Panel discussion: Model checking, theorem proving, and CAD. In *ACM/SIGDA International Workshop in Formal Methods in VLSI Design*, January 1991.

[12] T. Melham. Abstraction mechanisms for hardware verification. In G. Birtwhistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[13] C. Pygott. Noden_HDL: an engineering approach to hardware verification. In G. Milne, editor, *The fusion of Hardware Design and Verification*. Elsevier Science Publ. B.V.IFIP, 1988.

[14] J. P. V. Tassel. The semantics of VHDL with VAL and HOL: Towards practical verification tools. Master's thesis, Department of Computer Science and Engineering, Wright State University, 1989.

[15] J. P. V. Tassel and D. Hemmendinger. Toward formal verification of VHDL specifications. In L. Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, November 1989. Elsevier Science Publishers.

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

[17] P. J. Windley. Using correctness results to verify behavioral properties of microprocessors. In *Proceedings of the IEEE Computer Assurance Conference*, June 1991.