Technical Report

UCAM-CL-TR-196
ISSN 1476-2986

Number 196

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

The semantics of VHDL with Val and Hol: towards practical verification tools

John Peter Van Tassell

June 1990

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

ABSTRACT

Van Tassel, John Peter. M.S., Department of Computer Science and Engineering, Wright State University, 1989. The Semantics of VHDL with VAL and HOL: Towards Practical Verification Tools.

The VHSIC Hardware Description Language (VHDL) is an emerging standard for the design of Application Specific Integrated Circuits. We examine the semantics of the language in the context of the VHDL Annotation Language (VAL) and the Higher Order Logic (HOL) System with the purpose of proposing methods by which VHDL designs may be converted into these two forms for further validation and verification. A translation program that utilizes these methods is described, and several comprehensive VHDL design examples are shown.

Contents

List of Figures

Chapter 1

Introduction

1.1 Rationale

With the growing complexity of digital systems, it is becoming readily apparent that in the near future the validation of their designs is not going to be feasible using current tools. These methods have been based upon the simulation approach where it is even now impossible to simulate a design over all inputs, and still maintain a rapid development cycle. Rather, a "critical" subset of these inputs is fed into the simulation, but the derivation of that subset can sometimes be problematic.

The advent of new CAD systems has made the job easier for the engineer by helping him to establish a sound basis for the structural underpinnings of a given design. Examples of these systems include standard cell libraries and artificial intelligence packages [25] that point the designer in the right direction regarding the physical configuration of the device. In doing so, however, companies have necessarily tended to develop proprietary products. These then present a problem when designs for the same circuit from different vendors are being compared by a central agency. Further, the validation of designs developed using these tools is still encumbered by the problems of simulation.

To overcome the testability issues involved in current design validation practices, a different approach has emerged. It is one that seeks to reason about the structure of hardware from a proof theoretic point of view. Ideally, the engineer would specify a design in some formal language, and then apply the principles of mathematical proof to it to determine its validity with respect to the original specification. A potential drawback to such a scheme is that the designer needs to be familiar with the principles of formal proofs. Much of the process of formal verification is involved with the rewriting of terms into a standard form, which can be accomplished in a very mechanical and automatic fashion. The designer would then be required to interface with the proof system only at points where specific properties of his design are being examined. We believe that the tools for such verifications already exist, and can be coupled with a newly developed hardware description language to provide a prototypical environment for the development of future hardware systems.

Before any work on actual verification of designs can begin, it is essential to understand the semantics of the source language. The following chapters present methods to translate from this language into more formal representations. Our goal, therefore, is to understand and formally specify the operational semantics of an important subset of a current hardware description language. We now introduce the languages that will be used in the upcoming discussions.

1.2 Languages

1.2.1 VHDL

The VHSIC Hardware Description Language is an emerging standard in the design of Application Specific Integrated Circuits (ASIC's). The language originated with the U.S. Air Force as a way to provide a standard simulation environment for designs that it received from its various contracting agencies. The original VHDL that appeared for this purpose was Version 7.2 from Intermetrics. VHDL has since gone through the standardization process of the IEEE, to emerge in its current form as IEEE Std–1076–1987. The version of VHDL that is used in the discussions that follow is Intermetrics' implementation of the 1076 standard.

VHDL is meant to encompass a wide range of designs; from the highest level of abstraction down to the most basic components of an integrated circuit. These in turn may be implemented in different technologies (CMOS, NMOS, TTL, etc). Currently the only way that a given architecture may be validated is through exhaustive testing, which because of the complex nature of most commercial circuits, may not catch all bugs in the system. We seek to not only solidify the foundations of the design process, but also to move toward the formal verification of VHDL designs.

VHDL has a built in construct that allows the designer to insert logical assertions about the state of the simulation into his design. With these `assert` statements, we have a rudimentary system for specifying and checking on the operation of a particular component, but certain basic operations are not explicitly supported within them. These will be demonstrated and examined in the context of a translation program in Chapter 2.

1.2.2 VAL

The VHDL Annotation Language (VAL) was developed at Stanford to address the limitations of raw VHDL assertions, and does so without straying too far from their syntax. By using the translation methods of Chapter 2, it is possible to output VAL statements rather than pure VHDL. In doing so, we provide a more robust method of specifying the operation of a given design which can then be used within the context of traditional validation methods.

VAL presents a first step in the process of moving from simulation to verification. It allows for a more precise specification of the internal relationships of various components while remaining within the simulation environment of VHDL. These specifications are certainly not the formal statements acceptable to a theorem prover. They are encumbered by VHDL–specific constructs which relate to the underlying simulation environment, and are generally concerned with the timing model of the simulator. So, before a final transformation into an appropriate formalism can be made, we need to specify what that model is and how it operates.

1.2.3 HOL

The Higher Order Logic (HOL) system developed by Dr. Michael Gordon at Cambridge incorporates the functional language ML and a suite of predefined rules and tactics that are used in the derivation of formal proofs. It is a general–purpose theorem–proving environment allowing both forward and backward proofs, but has been used to verify devices from multipliers to simple microprocessors. HOL also incorporates an extensive sub–goal system to help break the proof down into more manageable pieces.

By refining the translation methods used to go from VHDL to VAL, we can provide a way of translating much of VHDL into appropriate HOLstructures. We can then make use of the sub–goal package previously mentioned, as well as the powerful rewriting tools provided by the HOLsystem to automatically perform many of the early steps involved in the verification of a given design (removal of universally and existentially quantified variables, normalization, etc.). HOL does, however, require a significant amount of user interaction after these basic steps are completed, and while this limits HOL's appeal, recent research at Cambridge suggests further automation is possible; leading the way towards formal verification of VHDL designs.

Chapter 2

VHDL

2.1 Background

The translator that is to be described was created by the author as a part of the 1988 U.S. Air Force Summer Graduate Student Research Program administered by Universal Energy Systems. During the course of the research, it was determined that VHDL's `assert` statement would provide a good basis for the investigation of the semantics of the language. The methodologies developed could then be used as a starting point for conversions into more formal languages.

The program consists of a translation grammar implemented in Prolog, and is based on the BNF description of the language [23]. While the whole of VHDL is parsed, emphasis is placed upon the semantics of certain key statements. The main assumption in the development of the program was that the VHDL source had already been analyzed as correct. The reason was that we did not want to implement a working compiler. Rather, we wished to develop a prototype for an automated translation program of certain features, and investigate its behavior. In that respect Prolog has proved to be an excellent tool for the construction of the program.

The output of the translator is an assertion that is to be reinserted into the original VHDL architecture. In general, the output tends to be difficult to read for two reasons. The first is that many conditional operations are expressed in terms of raw Boolean logic instead of using the more understandable, but still Boolean *if–then* notation. The second problem is that each expression is parenthesized to maintain the structure of the original. The obvious result is a lisp–like output stream that is not very legible.

The generated assertions could be used in the context of traditional circuit validation in the following manner. If a given design is known to be correct, an assertion can be generated to express its behavior. Then, when we design a new circuit that is supposed to exhibit the same behavior, we can insert the assertion that describes the correct design into the entity declaration for the new one. When the new design is simulated, any assertion violations that occur will point out areas where the new design has strayed from the old, correct one. Further use of these assertions can be made by understanding and specifying the underlying timing and simulation models used by VHDL. They also provide a more concise specification for the designer to work with when designing and debugging a given design.

```
C: process (a,b)
   begin
     :
   end process C;
```

Figure 2.1: An example of the process statement

2.2 Simulation Cycle and Timing Model

In order to understand the way in which the assertions operate, it is essential to understand the VHDL simulation cycle and timing model. The first concerns the sequence of actions that result during the elaboration (execution) of a signal assignment. The second deals with the timing relationships that VHDL implements to allow for the proper synchronization of those assignments.

VHDL is a discrete–event simulation language where concurrent statements are synchronized by being either explicitly or implicitly surrounded by a process statement. The statement can be thought of as a non–terminating subprogram that sleeps until certain conditions are met, executes once, and returns to the inactive state. An example of the skeleton of the construct is provided in Figure 2.1. The execution of process C is governed by the signals a and b which appear in its *sensitivity list.* An event on either one causes the process to become active [23].

Each signal assignment is an event, and is posted to an event queue for evaluation. All signals that are on the queue and that are supposed to be executed at the current time, make up an instance of the *simulation cycle* [23]. So, if we had events A, B and C that were scheduled for elaboration at the current time, they would comprise a complete simulation cycle. Any events that might result from their execution might also take part in future cycles. Figure 2.2 demonstrates three signal assignments that take place in the same cycle, and the simulation report that they generate (----- indicates a constant value). It must be noted that events do not occur in zero time. Even if there is no explicitly specified propagation delay time associated with the event, there is an implicit *delta* time in which it occurs. The *+n* in the figure indicate these deltas.

It is best to think of the simulation cycle being a complete sequence of steps executed by a finite state machine. The machine is governed by the advance of simulation time. When we advance one time unit into the future, we have experienced one "tick" of the overall simulation clock. The event queue is subdivided into blocks of events that are scheduled to be elaborated at some specific moment during the total simulation. When each tick of the simulation clock occurs, the block of the queue associated with that time is dequeued and elaborated. Unless the circuit has reached a state of quiescence, each block will generate more events. Each of those events is placed into the event queue inside a block associated with the appropriate point in simulation time. A side effect of the scheme is that new events overwrite any that might have been posted to a given block during some previous tick of the simulation clock if that posting was for a later simulation

```
a <= transport TRUE after 1ns;
b <= transport a;
c <= transport b;

TIME  |----------------SIGNAL NAMES-----------------|
      |
(NS)  |    A              B                C
      |
  1   |  TRUE           -----            -----
 +1   |  -----          TRUE             -----
 +2   |  -----          -----            TRUE
```

Figure 2.2: A simulation cycle

time. Such a scheme can prove confusing when first encountered, and points out one of the over–riding issues in the design of VHDL specifications; namely, that one must always keep in mind the underlying simulation cycle.

In the description of the generated assertions that follows, we will make use of both of these models. We want to sample the output of a waveform at the moment that it receives a new value, namely when an event occurs on it. Further, when that instant of time arrives, we want to be able to sample the values that the inputs had when the assignment began, and be able to make some generalizations about the states of the signals over the interval of time associated with the assignment.

2.3 Statements

The emphasis was on investigating the specification of VHDL in terms of its `assert` statements, with the later design of carrying the techniques used towards a translation into a theorem–proving environment. The particular features of VHDL that were investigated in detail are enumerated in the sub–sections that follow.

No attempt was made to distinguish between the concurrent and sequential aspects of VHDL. Assertions are themselves structures that can be placed in an environment of either parallel or sequential statements. Some of the more elaborate statements in the language such as generics, and bus resolution functions were not covered when writing the translator. These are important constructs, and translation schemes for some of them will be proposed in subsequent chapters.

2.3.1 Architectures

The VHDL architecture is the basic building block for the translation, as all statements that we were interested in occurred within them. In the early stages of development, they did nothing more than provide a VHDL comment about which block the upcoming

```
                    architecture A of B is
                         ⋮
                    end A;
-------------------------------------------------------------
-- For architecture B(A), the following assertion can be made:
-------------------------------------------------------------
```

Figure 2.3: VHDL architecture skeleton and generated comment

assertion pertained to. As an example, a simple architecture skeleton and its generated comment are provided in Figure 2.3.

In the course of later development, it became necessary to simulate a rudimentary VHDL design–library system, and the architecture name then also became the name of an output file into which the assertion would be placed. The reason for doing so was to facilitate the generation of assertions about architectures that are made up of many individual components. So, if architecture A was composed of components B, C, and D, we would make a composite assertion for the architecture by conjoining the generated assertions for B, C, and D, and perform the appropriate variable substitutions. The new assertion could then be reinserted into A. In so doing, we have preserved the hierarchical nature of the original VHDL, while at the same time condensing the assertions of each individual component into a single `assert` statement. This was an experiment in rewriting, and is not necessary if the assertions for each of the low–level components are in place in their corresponding architectures. The ability to consolidate expressions in this fashion will be more relevant when trying to formally verify designs.

There is a property of the port declarations used in the specification of entities which will have an impact on much of the discussions that follow. It is that ports declared to be of mode `out` cannot be read by statements within the corresponding architecture [23]. In order to get around the problem, we have made the generalization that all `out` ports are in fact `inout` ones. While it might appear on the surface that no real change has been made, each `out` port now represents a bus; thereby allowing for inappropriate actions on those ports to be accidentally made. Further, if we have many levels of entities with such redeclarations within our design hierarchy, unforseen glitches might result from compounding the problem.

2.3.2 Signal Assignments

There are two classes of signal assignments. One is associated with transport delay, the other with inertial delay of assignments. In inertial delay, there is a constraint placed on the inputs to the signal that they must be stable for a certain length of time. In assignments such as the one in Figure 2.4 where there is implicit inertial propagation delay time, `INPUT` must be stable for at least one delta time prior to the assignment to `OUTPUT`. In general, however, statements characterized by Figure 2.5 require that `INPUT`

```
OUTPUT <= INPUT;
```

Figure 2.4: Inertial delay signal assignment

```
OUTPUT <= INPUT after X;
```

Figure 2.5: Inertial delay with propagation time

be stable for X time before the signal assignment takes place. The whole purpose of the scheme is to prevent spurious spikes in the inputs to be carried over into the output.

To model signal assignments with inertial delay, the translation program makes use of both the 'Stable and the 'Delayed attributes of VHDL. 'Stable returns a Boolean value based on the stability of the given signal. So, if we wanted to know if signal INPUT has had an event in the last 4 nanoseconds, we test the value returned by INPUT'Stable(4ns). S'Delayed(X) will return the value that S had X time ago. If no argument is given, then the attribute returns the value the signal had one delta time ago. The translator, therefore, understands that it needs to make the following sort of statement about inertial delay: "if each one of the input signals has been stable X amount of time, then the output signal's value should be the same as the value of the input X time ago". Using the example of inertial delay in Figure 2.5, the translator would output the assertion in Figure 2.6 (edited for clarity). The NOT x or y scheme is equivalent to an *if* x *then* y construction.

We can simplify our assertion by changing over to transport delay. Here we have a type of signal assignment where no stability constraints need be modeled, but we still require the 'Delayed attribute. Transport delay is denoted by including the VHDL reserved word transport as a part of the input waveform. We can modify the previous example accordingly, and arrive at the signal assignment statement of Figure 2.7. The semantics of the statement are simply that "the output signal should always be equal to the value that the input had at the time denoted by the propagation delay". So, our assertion now becomes that of Figure 2.8.

With both of these forms of signal assignment specified, we are prepared to explore the translation of other structures that make use of them. In later examples, we will assume that both types of signal assignment will be translated into Boolean expressions such as these.

```
ASSERT (NOT INPUT'Stable(X) or
          (OUTPUT = INPUT'Delayed(X)));
```

Figure 2.6: Generated assertion for inertial delay

```
OUTPUT <= transport INPUT after X;
```

Figure 2.7: Signal assignment with transport delay

```
ASSERT (OUTPUT = INPUT'Delayed(X));
```

Figure 2.8: Generated assertion for transport delay

2.3.3 Conditionals

There are a number of ways to express conditional operations in VHDL, not all of which deal with "traditional" programming constructs. There is, of course, an *if–then–else* statement, and the *case* construct. In addition to these, we have conditional and guarded signal assignments as well as looping structures at our disposal.

The *if–then–else* statement in VHDL is the basis for most later translations of conditionals that will be performed. The pattern is not quite the same as that used in signal assignments. We still use primitive Boolean operators, but in order to mimic the implicit structure of the conditional, we must provide a method that takes the general structure of the *if* statement in Figure 2.9, and comes up with the VHDL translation shown in Figure 2.10. Note that each of the $statements_n$ is a conjunction of other sequential statements.

The *case* statement is nothing more than a shorthand for the *if* statement. In VHDL, it takes the form shown if Figure 2.11. The `others` clause is optional, but is provided to demonstrate the full construct. The translation into an assertion is the same as for the preceding *if* where `others` functions as an `else`, and `X` is equated with each of the $conditional_n$.

The conditional signal assignment statement is one with the basic structure of a *case* statement. We do, however, have to deal with the extra problems associated with signal

```
if condition1 then
   statements1;
elsif condition2 then
   statements2;
else
   statements3;
end if;
```

Figure 2.9: General form of the `if` statement

```
ASSERT ((condition1 and statements1) OR
        (not condition1 AND
             (condition2 and statements2) OR
             (not condition2 AND
                  statements3)));
```

Figure 2.10: Generated assertion for the `if` statement

```
case X is
   when conditional1 => statements1;
   when conditional2 => statements2;
   when others => statement3;
end case;
```

Figure 2.11: General form of the `case` statement

activity. Specifically, if we have a conditional signal assignment such as the one in Figure 2.12, we are not only dealing with the implicit inertial delay of the assignment, but at least 2 different delay times and conditions. With the simulation model of VHDL, the assertion that we create should not necessarily be true at all deltas within the simulation, but only after there has been an event on `Output`. The translator takes all but the final constraint into account, and generates the assertion of Figure 2.13.

The assertion would be evaluated at each delta if it is reinserted into the original design in its current form, and so must have some mechanism imposed upon it to cause it to be evaluated only when something happens to `Output`. The solution is to enclose it within a process that is sensitive to the signal in question. The assertion is then evaluated at the instant `Output` takes on a new value. The inability to put relevant guards directly into the assertion is the first example of a severe limitation of the pure `assert` statement.

Guarded signal assignments are those that appear inside block statements, and make use of an implicit Boolean signal in VHDL called `GUARD`. Such a statement might look like the one in Figure 2.14. The semantics of these statements are that if the implicit signal `GUARD` has just become true, or if it currently is true and one of the inputs that it is made

```
Output <= Input1 after X when A and B else
          Input2 after Y when C or D else
          Input3;
```

Figure 2.12: Sample conditional signal assignment statement

```
ASSERT (
  ((A and B and
      (not Input'Stable(X) or Output = Input'Delayed(X))) OR
   (not (A and B) and
      (((C or D) and
          (not Input2'Stable(Y) or Output = Input2'Delayed(Y))) OR
          (not (C or D) and
             (not Input3'Stable or Output = Input3'Delayed)))))
    );
```

Figure 2.13: Generated assertion for a conditional signal assignment

```
Output = guarded transport Input1 after X;
```

Figure 2.14: Sample guarded signal assignment

up of changes, then the signal assignment is allowed to occur [23]. In terms of an assertion, it becomes a matter of wrapping another condition around the assertion generated by the waveform alone. If the guarding statement was `A and B`, then the translator would give the assertion found in Figure 2.15. Here, we have gone back to the *not–or* pattern to represent *if–then* notions.

The only types of loops that have been addressed in the translation program are those with explicit iteration schemes. So, a looping statement similar to that in Figure 2.16 would become the assertion of Figure 2.17. Note that no attempt is made to derive loop invariants. Currently, the iteration schemes are restricted to those that make use of simple Boolean tests rather than those that enumerate a discrete range. An extension to encompass these should be made, and is quite simple. Had *condition* been the statement `I in J'High to J'Low`, then it would have become `(I >= J'High) and (I =< J'Low)`.

```
ASSERT (
   (NOT
     (GUARD AND
       (NOT GUARD'Stable or
            (A'Event or B'Event))))
   OR
   (Output = Input'Delayed(X)));
```

Figure 2.15: Sample guarded signal assignment

```
while condition loop
    statement_1;
    statement_2;
        ⋮
    statement_n;
end loop;
```

Figure 2.16: Sample loop statement

```
ASSERT (condition AND
          (statement_1 AND
           statement_2 AND ... AND
           statement_n));
```

Figure 2.17: Generated assertion for sample loop statement

2.3.4 Processes and Blocks

The `process` and `block` statements of VHDL provide a comment in much the same fashion as the `architecture` statement. We use the comment as a guide to where the assertion should be placed when we are done with the translation because an assertion about a process or block has no meaning outside that construct. The only process statements that have been addresses in the translation program are those that have sensitivity lists. A sample process and its generated comment are given as Figure 2.18. *<stmnts>* denotes the actual statements of the process. The same treatment is given to a sample block in Figure 2.19. We hold onto the guard `C` and `D` for use in future guarded signal assignment statements.

We now have the basic tools and methods for translating VHDL descriptions into logical

```
P1: process (A,B)
    begin
       <stmnts>
    end process P1;

-------------------------------------------------------
-- For process P1 the following can be asserted:
-------------------------------------------------------
```

Figure 2.18: `process` statement and generated comment

```
B1: block (C and D)
    begin
       <stmnts>
    end block B1;

------------------------------------------------------------
-- For block B1 the following can be asserted:
------------------------------------------------------------
```

Figure 2.19: `block` statement and generated comment

statements about their operations. It will become evident in Section 2.4 for reasons other than lack of legibility and awkward construction that these are not adequate to specify assertions that accurately reflect the hardware for purposes of simulation. They would be more useful in a translation to a more formal language that does not have constraints imposed upon it by the VHDL simulation cycle. In order to make relevant assertions about the operation of a given design, a more powerful language is required. Therfore, discussions of translation schemes for `assert` statements will exploit those already demonstrated, but will be enhanced later through the use of VAL constructs.

2.4 Example

To fully demonstrate the content, form, and use of the derived assertions, the following extended example is presented. It is based on a parity checker specified in [12] whose formal definition is included as Appendix B. A simple diagram of the circuit is shown in Figure 2.20. It is taken from [12]. A description of its components follows.

2.4.1 ONE

`One` is an entity whose sole purpose is to generate a constant high value. Note that it could be omitted by declaring a signal at the outermost level initialized to a high value. The VHDL description of the component is found in Figure 2.21. The assertion generated by the translator is very simple (generated comments have been omitted in this and subsequent assertions for brevity), and is shown in Figure 2.22.

2.4.2 Inverter

The component at the next level of complexity is `Inverter`. In VHDL it can be described as shown in Figure 2.23. The corresponding assertion generated is found in Figure 2.24.

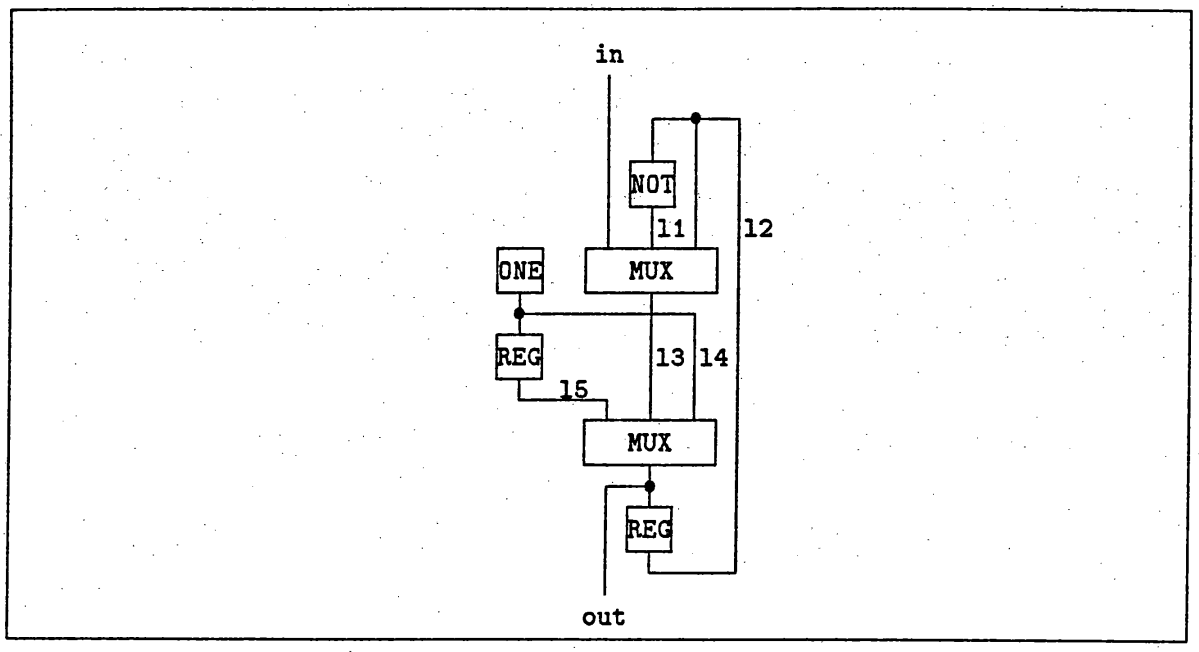

Figure 2.20: Diagram of the parity checker

```
entity One is
    port (Out_1: inout bit);
end One;

architecture Data_Flow of One is
begin
    Out_1 <= transport '1'; end Data_Flow;
```

Figure 2.21: Definition of One

```
ASSERT (Out_1 = '1');
```

Figure 2.22: Generated assertion for One

```
entity Inverter is
    port (IN_1: in bit; OUT_1: inout bit);
end Inverter;

architecture Data_Flow of Inverter is
begin
    OUT_1 <= transport not IN_1;
end Data_Flow;
```

Figure 2.23: Definition of Inverter

```
ASSERT (OUT_1 = not IN_1'Delayed(Ofs));
```

Figure 2.24: Generated assertion for Inverter

2.4.3 Register

The register uses unit-delay, and is unclocked. Its VHDL description is given in Figure 2.25. The assertion that it translates into is demonstrated in Figure 2.26.

2.4.4 Multiplexer

The most complex primitive unit in the design is a multiplexer. It is stated in Figure 2.27. The assertion then becomes the one in Figure 2.28.

```
entity Register is
    port (In_1: in bit; Out_1: inout bit);
end Register;

architecture Data_Flow of Register is
begin
    Out_1 <= transport In_1 after 1ns;
end Data_Flow;
```

Figure 2.25: Definition of Register

```
ASSERT (Out_1 = In_1'Delayed(1ns));
```

Figure 2.26: Generated assertion for Register

```
entity Multiplexer is
    port (A_in, B_in, Selector: in bit;
          Selected: inout bit);
end Multiplexer;

architecture Data_Flow of Multiplexer is
begin
   Selected <= transport (A_in and Selector) or
                         (B_in and not Selector);
end Data_Flow;
```

Figure 2.27: Definition of Multiplexer

```
ASSERT (Selected =
         ((A_in'Delayed(0fs) and Selector'Delayed(0fs)) or
          (B_in'Delayed(0fs) and not Selector'Delayed(0fs))));
```

Figure 2.28: Generated assertion for Multiplexer

```
entity Parity_Check is
    port (In_1: in bit: Out_2: inout bit);
end Parity_Check;

architecture Behavior of Multiplexer is
begin
   Out_2 <= transport not Out_2 after 1ns when In_1 = '1' else
                      Out_2 after 1ns;
end Behavior;
```

Figure 2.29: Definition of high–level version of Parity_Check

```
ASSERT ((((In_1 = '1') and (Out_2 = not Out_2'Delayed(1ns))) OR
        ((not (In_1 = '1') and (Out_2 = Out_2'Delayed(1ns))))));
```

Figure 2.30: Generated assertion for high–level version of Parity_Check

2.4.5 Parity Checker

The high–level specification of the behavior of the parity–checker is set forth in the component in Figure 2.29. The high–level assertion then becomes that of Figure 2.30.

We now have a simple specification of the parity checker. If it is correct, then it can serve as a basis for later designs of the same circuit. We could actually create a standard interface for such designs by modifying the original entity declaration for the component, and inserting the assert into it as in Figure 2.31.

If we now build the parity checker from the low–level components, we arrive at the description in Figure 2.32. We shall assume that the entity declaration is the "standard" one just created. By using out primitive library system to find the assertions for the various components, and by performing a sequence of variable substitutions, we derive the assertion about the low–level specification given in Figure 2.33.

```
entity Parity_Check is
    port (In_1: in bit: Out_2: inout bit);
begin
ASSERT ((((In_1 = '1') and (Out_2 = not Out_2'Delayed(1ns))) OR
        ((not (In_1 = '1') and (Out_2 = Out_2'Delayed(1ns))))));
end Parity_Check;
```

Figure 2.31: Prototypical entity declaration for Parity_Check

```
use work.One; use work.Inverter;
use work.Reg; use work.Multiplexer;

architecture Gate_Level of Parity_Check is

   component ones
      port (Out_1: out bit);
   end component;

   for all: ones use entity One(Data_Flow);

   component invert
      port (In_1: in bit; Out_1: out bit);
   end component;

   for all: invert use entity Inverter(Data_Flow);

   component mux
      port (A_in, B_in, Selector: in bit;
            Selected: out bit);
   end component;

   for all: mux use entity Multiplexer(Data_Flow);

   component regstr
      port (In_1: in bit; Out_1: out bit);
   end component;

   for all: regstr use entity Reg(Data_Flow);

   signal L1, L2, L3, L4, L5: bit := '0';

begin
   on1: ones port map (L4);
   inv: invert port map (L2,L1);
   mu1: mux port map (L1,L2,In_1,L3);
   re1: regstr port map (Out_2,L2);
   re2: regstr port map (L4,L5);
   mu2: mux port map (L3,L4,L5,out_2);
end Gate_Level;
```

Figure 2.32: Definition of low-level version of `Parity_Check`

```
ASSERT
    ((L4 = '1') AND
     (L1 = not L2'Delayed(0fs)) AND
     (L3 = (L1'Delayed(0fs) and In_1'Delayed(0fs)) or
           (L2'Delayed(0fs) and not In_1'Delayed(0fs))) AND
     (L2 = Out_2'Delayed(1ns)) AND
     (L5 = L4'Delayed(1ns)) AND
     (Out_2 = (L3'Delayed(0fs) and L5'Delayed(0fs)) or
              (L4'Delayed(0fs) and not L5'Delayed(0fs))));
```

Figure 2.33: Generated assertion for low-level `Parity_Check`

When the low-level architecture is simulated with the high-level assertion embedded in it, we find that assertion violations are generated from it (Figure 2.34). Is such behavior caused by bugs in the low-level description, or is it based on out not fully taking into account the way in which the simulator works? The answer stems from the latter. The assertion that is included as a part of the entity declaration is required to be true at all deltas in the simulation. It is not so during initialization or "power-up" of the design, nor is it so while an event on the inputs is being propagated through the system. In order to represent the proper behavior, we would be forced to have the assertion inside a `process` statement that was sensitive to changes in `Out_2` and `In_1`. What we would really like to say in the assertion is "when there is a change in the value of `In_1` from low to high, then `Out_2` will have the inverse of its current value when all signals settle down. Otherwise, it remains the same. Further, there should be no inadvertent glitches during the course of that propagation". Such a statement is not feasible in the pure form of the `assert` statement, but can be expressed in VAL.

```
      TIME  |--------------------SIGNAL NAMES--------------------|
      (FS)  |            IN_1                    OUT_1
            |
         0  |            '0'                     '0'
        +1  |                                    '0'
        +2  | %VHDSIM-W-ASSERTV Assertion Violation after 0 fs
            | Assertion violation
            |                                    '1'
   1000000  | %VHDSIM-W-ASSERTV Assertion Violation after 1 ns
            | Assertion violation
            |            '1'
        +1  |                                    '0'
        +2  | %VHDSIM-W-ASSERTV Assertion Violation after 1 ns
            | Assertion violation
            |                                    '1'
        +3  |                                    '0'
   2000000  |            '0'
   3000000  | %VHDSIM-W-ASSERTV Assertion Violation after 3 ns
            | Assertion violation
            |            '1'
        +2  |                                    '1'
   4000000  | %VHDSIM-W-ASSERTV Assertion Violation after 4 ns
            | Assertion violation
            |            '1'
        +3  |                                    '0'
   5000000  | %VHDSIM-W-ASSERTV Assertion Violation after 5 ns
            | Assertion violation
```

Figure 2.34: Simulation results for Parity_Check

Chapter 3

VAL

3.1 Overview

The VHDL Annotation Language is an effort to loosen the restrictions imposed by the strong typing of VHDL, and to provide for the specification of more complex timing relationships. It is a pre–processor for VHDL, where provisions are made for the description of the history of actions that were exhibited by the model. Further, the model of time may be specified using different modalities. Because a signal may propagate over the course of many deltas (Figure 2.2), a mechanism is also provided to make sure that assertions are not checked until everything has settled down at the end of the simulation cycle.

To accurately describe the way in which components are actually simulated, VAL implements a system that can be used to keep track of an entity's past behavior. This *entity state* [1] is helpful in the discussion of the structure of a given component. It will not be relevant in the translation schemes below, as we are trying to create assertions about the actions associated with the temporal properties of signal assignments, rather than attempting to specify the state transitions of all the statements involved in a given model.

When dealing with assertions about the timing constraints imposed by signal assignments, VAL utilizes three different modes to describe the model of time. The most general is *sometime* which says that a given assertion about a signal must be true somewhere in the current simulation cycle. The second level is implemented by *finally*. Here we specify that an assertion must be true at the end of the interval. Finally, the most restrictive condition is reflected in the *eventually* constraint. It is stronger than *finally* in that once an assertion becomes true, it cannot revert to a false value; and thus forbids oscillations in the waveform.

3.2 Translation Methods

Using VAL, we are able, to make the generated assertions not only more readable, but also more precise. We shall examine the different types of assertions generated in the previous chapter, and show their translation into VAL constructs.

3.2.1 Architectures

As in the previous chapter, these simply provide a comment about the block currently being transformed. There is no need to attempt to consolidate the assertions about all

```
--| when INPUT'Stable(X) then
--|    assert eventually OUTPUT = INPUT;
--| end when;
```

Figure 3.1: Inertial delay

the individual components as was done in pure VHDL. Since VAL is meant to enhance the testability of a given design in a simulation environment, such a scheme would not improve our ability to validate a design. We would only be duplicating work.

VAL's superiority over raw VHDL assertions is demonstrated already at this outer level. We previously had problems in dealing with out parameters (Section 2.3.1) in that they were not able to be read within the current architecture. In VAL, a set of dummy signals that get their values from the actual signals are used to alleviate the problem. When the user writes an annotation about signal X of mode out, VAL replaces references to X by those to the corresponding dummy. Note that these substitutions only take place in the context of the annotation and do not affect the description of the entity or its elaboration during simulation. VAL merely wraps a process around the annotation, and then plugs the process into the simulation in much the same way that one attaches an oscilloscope to a circuit to test it.

3.2.2 Signal Assignments

We have already derived the various statements necessary for the proper description of signal assignments. In the case of inertial delay, we found that stability constraints needed to be expressed for the inputs of the waveform in question. In order to accurately portray the relationship, it was necessary to resort to an awkward Boolean construction. We can use the when statement provided by VAL instead, and couple it to one of the special timing modalities mentioned above. So, the inertial delay signal assignment expressed earlier (Figure 2.5) could be transformed into the VAL statement in Figure 3.1. The *eventually* form of the assertion is used because it provides the most robust fashion of describing input–output relationships. The --| notation is the way in which VAL annotations are set off in the VHDL model. The -- part denotes the rest of the current line as a comment, and the | part tells us that it is a special, or formal [1] one. The VAL system, therefore, does not have to be a part of a user's VHDL environment for the model to be simulated.

In a like manner, transport delay may be expressed. It is not, however, necessary to resort to the when statement as no stability constraints are required. We can simply make a VAL assertion about the waveform as demonstrated in Figure 3.2.

Conditional signal assignments present an even more straightforward mapping into VAL. If we have the waveform in Figure 3.3 the translated annotation would become that of Figure 3.4. Had the assignment been one involving inertial delay instead, we could have simply nested the when statements of inertial signal assignment within the arms of the conditional signal assignment and arrived at the VAL statement in Figure 3.5.

```
--| assert eventually OUTPUT = INPUT'Delayed(X)
```

Figure 3.2: Transport delay

```
Output <= transport In_1 when Condition1 else
                    In_2 when Condition2 else
                    Output;
```

Figure 3.3: Transport delay conditional signal assignment (VHDL)

```
--| when Condition1 then
--|    assert eventually Output = In_1;
--| elsewhen Condition2 then
--|    assert eventually Output = In_2;
--| else
--|    assert eventually Output = Output;
--| end when;
```

Figure 3.4: Transport delay conditional signal assignment (VAL)

```
--| when Condition1 then
--|    when In_1'Stable then
--|       assert eventually Output = In_1;
--|    end when;
--| elsewhen Condition2 then
--|    when In_2'Stable then
--|       assert eventually Output = In_2;
--|    end when;
--| else
--|    when Output'Stable then
--|       assert eventually Output = Output;
--|    end when;
--| end when;
```

Figure 3.5: Inertial delay conditional signal assignment (VAL)

```
--| when Condition1 then
--|     Statements1
--| elsewhen Condition2 then
--|     Statements2
                ⋮
--| else Statementsn
--| end when;
```

Figure 3.6: Translation of `if` statement

```
while condition loop
   statements;
end loop
more statements;
```

Figure 3.7: Sample `loop` statement

3.2.3 Conditionals

The other major conditional constructs other than conditional signal assignments are the `if`, `case`, and `loop` structures of VHDL. We have shown transformations of these into assertions in the previous chapter. Now we shall show their conversion into VAL annotations. For the most part, the task will be simpler.

As mentioned previously, the `if` statement of VHDL has the form found in Figure 2.9. We can write the corresponding VAL by simply replacing `if` by `when` to derive the VAL statement in Figure 3.6. The `case` statement is addressed in the same fashion with the proviso that the individual conditions are equated with the test expression.

The translation of `loop` statements presents us with again a simpler task than the transformation into a plain VHDL assertion. once more, no attempt is made to derive loop invariants. So, if a loop containing a particular iteration scheme was expressed as in Figure 3.7, we could express it in terms of VAL using the `when` construction found in Figure 3.8. *condition* is the transformation for iteration schemes discussed earlier.

VAL makes the translation of guarded signal assignments both easier to perform and to understand. If we use the assignment from the previous chapter (Figure 2.14), we can obtain the VAL `when` statements of Figure 3.9.

3.3 Example

The example of the parity checker is now presented in a form where the derived assertions are expressed in VAL. Throughout the course of the exposition, we will not re–state the

```
--| when condition then
--|   statements
--| else
--|   more statements
--| end when;
```

Figure 3.8: VAL translation of `loop` statement

```
--| when GUARD and (NOT GUARD'Stable or (A'Event or B'Event)) then
--|    assert eventually Output = Input'Delayed(X);
--| end when;
```

Figure 3.9: VAL translation of a guarded signal assignment

original VHDL, but merely give the VAL assertions that are generated from them. The reader is encouraged to refer to the originals.

3.3.1 One

There are no conditions associated with the description of `One`. We therefore express the relationship in terms of a simple VAL assertion (Figure 3.10).

3.3.2 Inverter

Figure 3.11 shows the VAL expression describing the action of `Inverter`. Like `One`, there are no explicit conditions associated with its operation. So, we can again express the relationship in terms of a simple VAL assertion.

3.3.3 Register

`Register` is also a simple component with no conditions on the input–output relationship other than the propagation time. The VAL assertion that describes the entity is given as Figure 3.12.

```
--| assert eventually out_1 = '1';
```

Figure 3.10: VAL version of `One`

```
--| assert eventually out_1 = not in_1'Delayed(0fs));
```

Figure 3.11: VAL version of `Inverter`

```
--| assert eventually out_1 = in_1'Delayed(1ns));
```

Figure 3.12: VAL version of `Register`

3.3.4 Multiplexer

With the specification of `Multiplexer`, we see the use of the VAL `when` statement to state the conditions under which a given assertion should be true. The actual VAL is shown in Figure 3.13.

3.3.5 Parity Checker

As noted previously, we shall not consolidate the assertions that specify the behavior of the individual parts of a component. Therefore, we only give the VAL assertion for the high–level version of `Parity_Check`. As can be seen in Figure 3.14, the VAL version is not only more readable than its pure VHDL predecessor, but also states the timing relationships more clearly. The assertion in its current form can be inserted into the entity declaration for `Parity_Check` to create a much more precise version of the "standard" interface shown in Section 2.4.

3.3.6 Simulation Results

If we simulate the parity checker, and include the high–level VAL assertion about it in the entity declaration for the low–level specification, we can demonstrate the differences between the three VAL timing notations. We begin with the losest restriction possible by using the *sometime* construct. The simulation results are shown in Figure 3.15. We note

```
--| when selector'Delayed(0fs) then
--|    assert eventually selected = a_in'Delayed(0fs);
--| elsewhen not selector'Delayed(0fs) then
--|    assert eventually selected = b_in'Delayed(0fs);
--| end when;
```

Figure 3.13: VAL version of `Multiplexer`

```
--| when In_1 = '1' then
--|   assert eventually Out_2 = not Out_2'Delayed(1ns);
--| elsewhen not In_1 = '1' then
--|   assert eventually Out_2 = Out_2'Delayed(1ns);
--| end when;
```

Figure 3.14: VAL version of high–level `Parity_Check`

TIME	SIGNAL NAMES	
(FS)	IN_1	OUT_2
0	'0'	
+1		'0'
+2		'1'
1000000	'1'	
+1		'0'
+2		'1'
+3		'0'
2000000	'0'	
3000000	'1'	
+2		'1'
4000000	'1'	
+3		'0'

Figure 3.15: Simulation results for *sometime* version

that no assertion violations are triggered, as it is the case that the conditions specified by our high–level are true at some delta within each simulation cycle.

We now include the restriction that the conditions specified in the assertion must be met at the end of each simulation cycle; namely, when `Out_2` has attained its final value and settled down. The *finally* timing constraint is used to accomplish the task, and picks up the indeterminate behavior of the circuit during "power–up" (Figure 3.16). The problem is caused by the `'Delayed(1ns)` attribute having defaulted to a low ('0') value before such time as it had any meaning.

We now place the most restrictive condition on the assertion by using the *eventually* constraint. The result is the detection of the "glitch" in `Out_2` during the period 1 nanosecond after the start of the simulation (Figure 3.17). The bug is caused by a race condition present in the system when the first high value is input into it.

```
          TIME  |------------------SIGNAL NAMES----------------|
          (FS)  |            IN_1                    OUT_2
                |
             0  |            '0'                     '0'
            +1  |                                    '0'
            +2  |                                    '1'
             1  | %VHDSIM-W-ASSERTV Assertion Violation after 1 fs
                | VAL violation (finally)
       1000000  |            '1'
            +1  |                                    '0'
            +2  |                                    '1'
            +3  |                                    '0'
       2000000  |            '0'
       3000000  |            '1'
            +2  |                                    '1'
       4000000  |            '1'
            +3  |                                    '0'
```

Figure 3.16: Simulation results for *finally* version

```
          TIME  |------------------SIGNAL NAMES------------------|
          (FS)  |            IN_1                    OUT_2
                |
             0  |            '0'                     '0'
            +1  |                                    '0'
            +2  |                                    '1'
             1  | %VHDSIM-W-ASSERTV Assertion Violation after 1 fs
                | VAL violation (eventually)
       1000000  |            '1'
            +1  |                                    '0'
            +2  |                                    '1'
            +3  |                                    '0'
       1000001  | %VHDSIM-W-ASSERTV Assertion Violation after 1000001 fs
                | VAL violation (eventually)
       2000000  |           '0'
       3000000  |           '1'
            +2  |                                    '1'
       4000000  |           '1'
            +3  |                                    '0'
```

Figure 3.17: Simulation results for *eventually* version

Chapter 4

HOL

4.1 Overview

The Higher–Order Logic system is a theorem–proving environment that is based on the natural–deduction approach. The source language for the system is ML, and much of the system is implemented in terms of it. HOL is still evolving, but the basic core has been in use for a number of years.

HOL incorporates a wealth of rules and tactics to aid in the proof process. The rules are used for forward proof, and the tactics are used to go in the reverse direction. They may be intermixed in any given proof. Tactics may be either user or system–defined, and provide the most suitable tools for the verification of hardware, although forward methods have been used.

In a translation from VHDL into HOL we will be using certain specific HOL constructs. It is going to be possible to model more than just architectures, as HOL does not separate the design into architectural components and functional components in the manner of VHDL. Rather, it allows for both kinds of structures through the use of the built–in function `new_definition`.

The translation methods from VHDL into HOL will now be outlined. Their exposition will, in general, follow the same pattern as that for pure VHDL assertions and VAL annotations. It will, however, be necessary to give a thorough explanation of the typing issues involved in these translations. Also, we will be making use of four HOL logical constructs. Universal quantification ($\forall$) will be represented by `!`, and existential quantification ($\exists$) by `?`. Logical conjunction becomes `/\`, and disjunction is `\/`. HOL strings are enclosed within a pair of backquotes (` `), whereas terms are offset by double quotes (`""`) [13].

4.2 Types

The strong typing provided by both VHDL and HOL makes the process of transforming the former into the latter easier, and allows for some generalizations to be made. The most obvious are for Booleans, strings, and natural numbers. For Boolean operations, HOL provides the type `:bool`, which is analogous to `BOOLEAN` in VHDL. In a like manner, HOL's `:tok` is VHDL's `STRING`, and `:num` represents `NATURAL`. Each of these types may be used in the translation of VHDL constants.

The above are all static typing issues, and are not particularly useful in describing dynamic systems such as those found in hardware. In order to make any headway in deciphering the meanings of digital components, it is necessary to have a way of representing time.

```
new_type_abbrev('BIT', ":bool");;
new_type_abbrev('BIT_SIG', ":time->BIT");;
new_type_abbrev('BIT_VECTOR', ":num->BIT");;
new_type_abbrev('BIT_VECTOR_SIG', ":time->BIT_VECTOR");;
```

Figure 4.1: HOL declarations for BIT and BIT_VECTOR

The differences between HOL and VHDL are readily apparent on this issue. VHDL, being a simulation language, has a notion of timing relationships built into the run-time system that the user must be constantly aware of (Section 2.2). Conversely, HOL is a general theorem proving environment, and therefore requires the user to supply a timing model. In order to accomplish this, we make use of a non-decreasing counter as a part of all types used to describe components of the hardware. We are in effect, mimicking the VHDL notion of time. It should, however, be noted the HOL does not have a true capability for representation of real numbers. So, we can run into problems when using the different intervals available in VHDL (nanoseconds, femtoseconds, etc.). The difficulty can be overcome by re–scaling all times to the smallest one used in the model in question. In general, time is specified by the HOL definition `new_type_abbrev ('time', ":num")`, which in turn could be used to model a Boolean signal as `new_type_abbrev ('wire',":time->bool")`. Since during the course of translation there will generally be a need to have both static and dynamic structures, two translations of declared VHDL types will be made. The first is the static translation (as in those for STRING and BOOLEAN just mentioned), the other will take those just–abbreviated types, and define a signal abbreviation by mapping `:time` onto them (Figure 4.1).

There is a further limitation of the HOL system with regard to the translation of VHDL descriptions, albeit a minor one. It is simply that there is no built in theory about individual bits. We can reconcile VHDL to HOL by making all references to objects of type BIT into those for the HOL type `:bool`, but an abbreviation for BIT can be defined to allow for a transparent translation. In doing so, each bit must be converted into its Boolean equivalent when encountered during the course of a translation. The predefined type BIT_VECTOR can be represented in a similar fashion by the declaration of an array of the newly specified type `:BIT`. These declarations are given in Figure 4.1.

The HOL declaration for BIT_VECTOR is a sample of the kind of declaration that is required for all array types in VHDL. Specifically, a type was declared to map the natural numbers to individual bits. When a array of signals is desired, `:time` is mapped onto the array type itself. A general VHDL array declaration and its corresponding HOL translation would then take the form of those in Figure 4.2 Note that the VHDL has an explicit range constraint on the length of the array, while the HOL translation does not. Such a situation can be problematic unless one includes specific range checking constraints in all operations on variables of the type `:ARRAY_TYPE` and `:ARRAY_TYPE_SIG`.

Simple VHDL record types can be represented as n–tuples. A type definition for them is made up of the types for each element used in the record declaration. Figure 4.3

```
type ARRAY_TYPE is array (0 to 31) of ELEMENT_TYPE;

new_type_abbrev('ARRAY_TYPE', ":num->ELEMENT_TYPE");;
new_type_abbrev('ARRAY_TYPE_SIG', ":time->ARRAY_TYPE");;
```

Figure 4.2: Translation of a VHDL array type definition

```
type VHDL_record is record
                     A: BOOLEAN;
                     B: BIT_VECTOR;
                    end record;
signal Q: VHDL_record;
signal C: BIT_VECTOR;

Q.A <= transport TRUE;
Q.B <= transport C after 1ns;

new_type_abbrev ('VHDL_record', ":BOOLEAN_SIG # BIT_VECTOR_SIG");;
new_type_abbrev ('VHDL_record_SIG', ":time->VHDL_record");;

!t. ? Q_A:BOOLEAN_SIG Q_B:BIT_VECTOR_SIG.
  Q = (Q_A,Q_B) /\
  Q_A t = T /\
  Q_B t = C (t-1) /\
  Q t = (Q_A t,Q_B t)
```

Figure 4.3: HOL translation of a VHDL record

demonstrates an assignment of values to a signal that was defined as a VHDL record. The symbol # is used to represent a Cartesian product. The semantics of the assignment statement will be discussed in Section 4.3.2.

4.3 Statements

4.3.1 Architectures

To represent the structure of hardware, HOL provides one mechanism for describing all possibilities, while VHDL has three main ones. In VHDL, these classes are the procedure, function, and architectural entity. In HOL, each of them is simply an instance of `new_definition`. In a one-way translation from VHDL to HOL, therefore, we are not left with much of a decision as to how to best represent the outermost VHDL construct in

```
entity A is
   generic (G1, G2: NATURAL; G3: BIT);
   port(P1,P2: BIT; P3: out NATURAL);
end A;

let A_DEF = new_definition
    (`A`,
     "A (G1:NATURAL,G2:NATURAL,G3:BIT)
        (P1:BIT_SIG,P2:BIT_SIG,P3:NATURAL_SIG) = ...");;
```

Figure 4.4: Translation of entity declaration

```
new_definition (
     `Stable`,
     "Stable (interval,t) sig =
         !delta. (((t-interval) =< delta) /\ (delta < t)) ==>
             (((sig delta):*) = ((sig (t-interval)):*))");;
```

Figure 4.5: HOL Specification of `'Stable`

question.

The derivation of the arguments for the HOL definition of a VHDL architecture comes from the entity declaration. HOL, unlike VHDL does not make use of "direction" modes for its parameters. So, our translation does nothing more than pull off the name of the entity and its various ports to make the translated HOL specification (Figure 4.4). Note also that generics are treated in the same fashion, but non–signal declarations are used. Generics are translated in this manner because they denote static objects, and therefore do not have a time parameter associated with them. The HOL type `NATURAL` results from `new_type_abbrev(`NATURAL`,":time->num")`, and represents a signal over the natural numbers.

4.3.2 Signal Assignment

The driving force behind a VHDL simulation is the signal assignment statement. As seen in previous chapters, they can take several forms, and have different delays associated with them. It will be of paramount importance to maintain the propagation scheme imposed by the VHDL simulator in the translation process. In order to do so, we provide a HOL definition of `'Stable` (Figure 4.5) adapted from [16]. `'Delayed` is to be modeled by explicitly stating the propagation time with each signal.

Armed with a definition of signal stability, we are now prepared to embark on the task of specifying the various forms of signal assignment present in VHDL as HOL definitions.

```
!t. OUTPUT t = INPUT (t-X)
```

Figure 4.6: HOL version of transport delay

```
!t. OUTPUT t = (Stable (X t) INPUT => INPUT (t-X) |
                                      OUTPUT t)
```

Figure 4.7: HOL version of inertial delay

The first of these is the simplest flavor available in VHDL; namely, those assignments incorporating transport delay. The statement shown in Figure 2.7 is translated into the HOL term of Figure 4.6. Note that the HOL corresponds quite closely with the raw VHDL assertion of Figure 2.8.

The next step in the specification of signal assignments is to look at those that make use of inertial delay. If we take the assignment of Figure 2.5 as a general example of the statement, we must again make the same kind of statement about signal stability that we did when constructing the VHDL assertion. In order to do so, the HOL definition of `Stable` must be used. The result is shown in Figure 4.7. Roughly translated into English, the statement reads "at all times, if the signal `INPUT` has been stable since X, then the signal `OUTPUT` reflects the value that `INPUT` had X time ago, otherwise it remains the same". We have just introduced the basic form of conditional expression used in HOL. The symbol `=>` is used to represent the *if–then* part of the operation, and `|` denotes an alternative choice. We shall abstract away from the constraints imposed by the VHDL simulator, and assume that all signals are stable at the current time `t` because no model of delta time exists.

We have also previously examined the conditional signal assignment. We can generalize the HOL conditional just used to encompass this particular VHDL structure. We shall use the statement of Figure 2.12 as the basis for an example translation, and can easily derive the HOL statement of Figure 4.8. If the assignment had been one associated with transport delay, all that would have to be changed is the removal of the stability constraints from each branch of the conditional.

The final type of signal assignment statement that we have examined has been the guarded signal assignment statement. It uses the implicit VHDL signal `GUARD` to govern the way that it is executed. Since `GUARD` is nothing more than another condition to be evaluated before the signal assignment takes place, we can re–write the operation in much the same fashion as in Figure 2.15. The only extra change that will have to be made is the expansion of `GUARD` into the actual condition that it represents. Also, since we have declared that all signals are stable at the current time, we have no need for the calls to `'Event`. So, the statement of Figure 2.14 would be translated into the HOL condition of Figure 4.9.

```
!t. Output t = ((A /\ B) /\ Stable (X t) Input1 =>
                     Input1 (t-X)   |
                (C \/ D) /\ Stable (Y t) Input2 =>
                     Input2 (t-Y)   |
                Input3 t)
```

Figure 4.8: HOL version of conditional signal assignment

```
!t. Output t = ((A /\ B) => Input (t-X) |
                            Output t
```

Figure 4.9: HOL version of a guarded signal assignment

A previously untranslated structure within the context of signal assignments has been the bus resolution function. Its purpose is to guarantee that an output signal with multiple inputs functions properly. These functions can be used to implement operations such as *wired–and*'s or those that guarantee that only one signal at a time will be trying to drive a bus [23]. HOL, because it is a language not given to simulation, does not have a corresponding structure. If there is a signal that has more than one driver (such as a bus), the user is required to specify a predicate whose output guarantees the property desired. In a translation environment, we would recognize those signals that were dependent upon a resolution signal, and make all assignments to them as if there was no function involved. A note would be generated to the user that a resolved signal was being used, and that the way in which it is resolved may or may not have to be further specified. It may appear as if the issue of translation of the actual function is being avoided, but it is one rooted in the way each language is used. In VHDL, the simulator must be told how to deal with every situation, while in HOL, a resolution function is an entity outside the scope of specification of the physical hardware.

In the case of the signal assignments for records shown in Figure 4.3, it is necessary to specify each part of the record structure through existential quantification. It is then possible to operate on the individual parts of the record, and then re–assemble them when all operations are completed. The solution is an awkward one, but it provides the most generality in accessing the various components of the structure.

4.3.3 Conditionals

With a construct that represents conditional operations, the task of translating the conditional structures of VHDL becomes much the same as it was for VAL. We simply map the structure of the VHDL version onto that of the HOL one. So, in the case of the `if` statement of Figure 2.9, the resulting HOL would be expressed by the conditional shown

```
condition1 => statements1 |
condition2 => statements2 |
              statements3
```

Figure 4.10: HOL translation of `if` statement

```
(X = conditional1) => statements1 |
(X = conditional2) => statements2 |
                      statements3
```

Figure 4.11: HOL version of a `case` statement

in Figure 4.10. The HOL conditional looks more like the VHDL `case` statement than the `if`. The relationship can be readily exploited, and if we use the statement of Figure 2.11 as an example, the result is the HOL conditional of Figure 4.11.

4.3.4 Processes and Blocks

The model of the VHDL process and block that we shall use is that of Section 2.3.4. From the point of view of the VHDL translation, these provided a comment about where the generated assertion was to be placed. In HOL, they take on a meaning similar to that of the entity declaration of Section 4.3.1. We separate them from the current architecture, and make a new definition from them. We then insert a call to that definition into the parent, and conjoin it with the rest of the statements. In the case of blocks, we hold onto the guard (if present) for later insertion into any translations for guarded signal assignments (Figure 4.9). So, if we were currently involved in translating the architecture `Dummy`, and encountered the process of Figure 2.18 and the block of Figure 2.19, we would arrive at the HOL translation seen in Figure 4.12. *<other_vars>* simply is where other variables present in the definition of `Dummy` would be placed.

4.4 Summary

We have certainly not specified the translation of the whole of VHDL. Emphasis has been placed on the statements originally encountered by the program described in Chapter 2. In the process of describing the HOL versions these constructs, we have approximated the formal specification of the operational semantics for many common VHDL statements. With the development of a better model of VHDL's timing model, these will become fully specified. To illustrate the potential of the translation methods just proposed, we will present in the next chapter a large–scale example of a circuit previously verified in HOL.

```
PROC_P1_DEF = new_definition (
  'PROC_P1',
  "PROC_P1(A,B,<other_vars>) = <stmnts>");;

BLK_B1_DEF = new_definition (
  'BLK_B1',
  "BLK_B1(C,D,<other_vars>) = <stmnts>");;

Dummy_DEF = new_definition (
  'Dummy',
  "Dummy(A,B,C,D,<other_vars>) =
       ... /\
       PROC_P1(A,B,C,D,<other_vars>) /\
       BLK_B1(C,D,A,B,<other_vars>) /\
       ... ");;
```

Figure 4.12: Extracted process and block definitions

Chapter 5

Tamarack

5.1 Overview

To demonstrate the viability of the translation of VHDL designs into HOL specifications, the following extended example is presented. The design is for the Tamarack microprocessor, and is based on the descriptions of it that appear in [10], [17], and [21]. We will give the VHDL description of each unit that makes up the chip, and then transform those descriptions into HOL via the translation methods of Chapter 4. The results of each translation are then compared to the most recent HOL implementation found in [18].

The computer is described as an n–bit system in HOL, but was fabricated as an 8–bit implementation. Memory was implemented as a 32–word × 8–bit RAM. The VHDL version that is presented will follow the more abstract original representation through the use of generics. We can then remain flexible in the implementation that is chosen.

The computer consists of two basic units. The first is a control unit which includes a microcoded ROM, a decoder for it, and next address logic. The second is the data path, and is made up of a memory address register, main memory, program counter, accumulator, instruction register, and an ALU with an associated buffer and register. The register–transfer level system diagram is provided as Figure 5.1, and is taken from [21].

5.2 Basic Definitions

We begin with the underlying types and functions used in the representation of Tamarack. These are described in VHDL through the use of a package and three architectures. The VHDL source for each of them will examined in the sub–sections that follow.

5.2.1 Package `Tamarack`

The package declaration of Figure 5.2 defines all the types, functions, and procedures that will be used in the VHDL model, except for those that are specific to the specification of the microcode. The subtype `TSI` is used to give a tristate meaning to the natural numbers. A value of -1 on the part of any object declared to be of type `TSI` means that a high–impedance condition is being represented. We want to be able to have a system bus in the Tamarack that is of type `TSI` as well. In order to do so, it will be necessary to provide a resolution function for that bus. The subtype `BUSTYPE` is used for the purpose, and has been tied to the resolution function `RESOLVED`. The type `MEMTYPE`

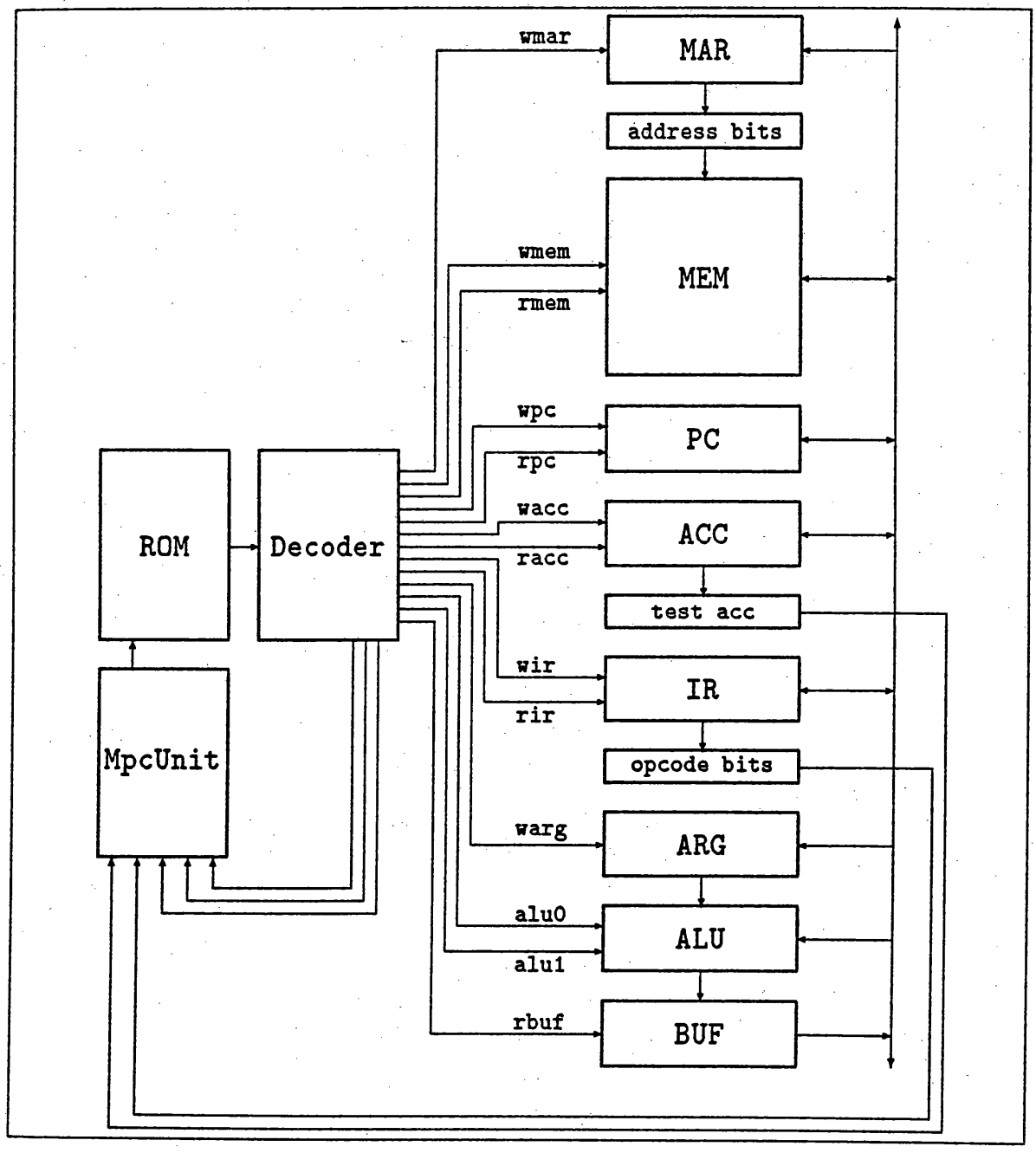

Figure 5.1: Register–Transfer Level Architecture

```
package Tamarack is

    subtype TSI is INTEGER range -1 to INTEGER'high;
    type INP_VECT is array (INTEGER range <>) of TSI;

    function RESOLVED (INPT: INP_VECT) return TSI;
    subtype BUSTYPE is RESOLVED TSI;

    type MEMTYPE is array (INTEGER ranger <>) of TSI;

    function INCn (N, A: in TSI) return TSI;
    function ADDn (N: NATURAL; A, B: in TSI) return TSI;
    function SUBn (N: NATURAL; A, B: in TSI) return TSI;
    function Bits (N, M: NATURAL; W: in TSI) return TSI;
    procedure TNZ (signal INPT: in TSI;
                   signal FLAG: out BOOLEAN);
    procedure HWC (constant C: in Natural;
                   signal OUTP: out TSI);

end Tamarack;
```

Figure 5.2: Package declaration for Tamarack

describes the configuration of main memory. The function and procedures listed in the package declaration are defined in the body of the package (Figure 5.3).

The HOL type abbreviations for the three primary types are given in Figure 5.4. Here two definitions of TSI are shown. The first is for static objects that have no impact on the timing relationships within the system. The second is for objects that are signals. TSI is expressed in terms of :num rather than regular integers. HOL's underlying language, ML, does allow operations on negative numbers, but it is not possible to define an abbreviation for items of type :int. In any event, the negative case will be needed in only a few instances, and can be either ignored or expressed in a different manner. Since BUSTYPE is associated exclusively with signals, it is mapped directly into TSI_SIG. When the translation is made, note is taken that all signals of type BUSTYPE are resolved. Entities that make use of these signals will have to be treated specially when they are converted to HOL by generating a note to the user. MEMTYPE and MEMTYPE_SIG are derived in the same fashion as TSI and TSI_SIG. It is unknown at the current stage of translation whether both will be needed, so the translations are made. Only MEMTYPE_SIG will eventually be used.

Using the function and procedure definitions in the body of the package, we can make the translations into HOL found in Figure 5.5. RESOLVED is the resolution function for signals of type BUSTYPE. It simply goes through each of the signals that are trying to drive the bus, and assigns the last one that was not in a high-impedance state to the variable

```
package body Tamarack is
    function RESOLVED (INPT: INP_VECT) return TSI is
       variable RESOLVED_VALUE: TSI;
    begin
       for I in INPT'range loop
          if INPT(I) /= -1 then
             RESOLVED_VALUE :=INPT(I);
          end if;
       end loop;
       return RESOLVED_VALUE;
    end RESOLVED;

    function INCn (N, A: in TSI) return TSI is
    begin return ((A + 1) mod (2 ** N)); end INCn;

    function ADDn (N: NATURAL; A, B: in TSI) return TSI is
    begin return ((A + B) mod (2 ** N)); end ADDn;

    function SUBn (N: NATURAL; A, B: in TSI) return TSI is
    begin return ((A - B) mod (2 ** N)); end SUBn;

    function Bits (N, M: NATURAL; W: in TSI) return TSI is
    begin return ((W / (2 ** N)) mod (2 ** M)); end Bits;

    procedure TNZ (signal INPT: in TSI;
                   signal FLAG: out BOOLEAN) is
    begin FLAG <= not (INPT = 0); end TNZ;

    procedure HWC (constant C: in Natural;
                   signal OUTP: out TSI) is
    begin OUTP <= C; end HWC;
end Tamarack;
```

Figure 5.3: Package body for Tamarack

```
new_type_abbrev('TSI', ":num");;
new_type_abbrev('TSI_SIG', ":time->TSI");;
new_type_abbrev('BUSTYPE', ":TSI_SIG");;
new_type_abbrev('MEMTYPE', ":num->TSI");;
new_type_abbrev('MEMTYPE_SIG', ":time->MEMTYPE");;
```

Figure 5.4: HOL translations of the basic types

```
let INCn = new_definition (
   'INCn',
   "INCn(N:TSI,A:TSI) = ((A + 1) MOD (2 EXP N))");;

let ADDn = new_definition (
   'ADDn',
   "ADDn(N:num,A:TSI,B:TSI) = ((A + B) MOD (2 EXP N))");;

let SUBn = new_definition (
   'SUBn',
   "SUBn(N:num,A:TSI,B:TSI) = ((A - B) MOD (2 EXP N))");;

let Bits = new_definition (
   'Bits',
   "Bits(N:num,M:num,W:TSI) =
         ((W Div (2 EXP N)) MOD (2 EXP M))");;

let TNZ = new_definition (
   'TNZ',
   "TNZ(INPT:TSI_SIG,FLAG:wire) =
         !t. FLAG t = ~(INPT t = 0)");;

let HWC = new_definition (
   'HWC',
   "HWC(C:num,OUTP:TSI_SIG) = !t. OUTP t = C");;
```

Figure 5.5: Translation of basic procedures and functions

RESOLVED_VALUE. There should only be one of these, and we will define an architecture to ensure that the bus exhibits this property. Since RESOLVED is a resolution function, note will be made of what type it is tied to, but it will not be converted. The other functions make use of modulus arithmetic to operate on an arbitrary "word" of length n that is passed into them. INCn is used to increment the given word by one, ADDn adds two such words together, and SUBn subtracts them. Bits is used to extract a sub-field of bits from a word. TNZ tests a given signal against zero, and HWC changes an arbitrary constant into a signal.

The original HOL specifications for the primitives are shown in Figure 5.6, and are taken from [18]. They do not make use of many of the typing abbreviations made for the VHDL version. The derived HOL is not inconsistent with the original because the HOL typechecker maps variables to types based on the operations performed on them [13]. Other differences result from the discrepancies between the names used for type abbreviations.

```
new_type_abbrev (‘time‘, ":num");;
new_type_abbrev (‘wire‘, ":time->bool");;
new_type_abbrev (‘bus‘, ":time->num");;
new_type_abbrev (‘mem‘, ":time->num->num");;

let INCn = new_definition (
   ‘INCn‘,
   "INCn n a = (a + 1) MOD (2 EXP n)");;

let SUBn = new_definition (
   ‘SUBn‘,
   "SUBn n (a,b) = (a - b) MOD (2 EXP n)");;

let ADDn = new_definition (
   ‘ADDn‘,
   "ADDn n (a,b) = (a + b) MOD (2 EXP n)");;

let Bits = new_definition (
   ‘Bits‘,
   "Bits (n,m) w = ((w Div (2 EXP n)) MOD (2 EXP m))");;

let TNZ = new_definition (
   ‘TNZ‘,
   "TNZ (in:bus,flag:wire) = !t. flag t = ~(in t = 0)");;

let HWC = new_definition
   (‘HWC‘, "HWC c (b:bus) = !t. b t = c");;
```

Figure 5.6: Original HOL specifications

```
entity PWR is
   port (OUTPUT: out BOOLEAN);
end PWR;

architecture BEHAVIOR of PWR is
begin
   OUTPUT <= transport TRUE;
end BEHAVIOR;

entity GND is
   port (OUTPUT: out BOOLEAN);
end GND;

architecture BEHAVIOR of GND is
begin
   OUTPUT <= transport FALSE;
end BEHAVIOR;
```

Figure 5.7: VHDL behaviors for power and ground

```
let PWR = new_definition (
   'PWR',
   "PWR(OUTPUT:BOOLEAN_SIG) = !t. OUTPUT t = T");;

let GND = new_definition (
   'GND',
   "GND(OUTPUT:BOOLEAN_SIG) = !t. OUTPUT t = F");;
```

Figure 5.8: HOL translations of PWR and GND

5.2.2 Power and Ground

To model power and ground for the system, two architectures will be provided. For the former, TRUE is mapped to the output at all times, while the latter puts out FALSE. These can be described by the VHDL architectures of Figure 5.7.

The HOL translations of these are very straightforward. The entity declarations provide the parameters for each instance of new_definition, and the architectures are used to derive the waveform representation. The results are shown in Figure 5.8, and differ from those originally given by Joyce (Figure 5.9) [18] in a superficial way.

```
let PWR = new_definition (
   'PWR',
   "PWR(w:wire) = !t. w t = T");;

let GND = new_definition (
   'GND',
   "GND(w:wire) = !t. w t = F");;
```

Figure 5.9: Original HOL versions of PWR and GND

```
entity BITS is
   generic (N, M: NATURAL);
   port (INP: in TSI; OUTP: out TSI);
end BITS;

architecture BEHAVIOR of BITS is
begin
   OUTP <= transport work.Tamarack.Bits (N, M, INP);
end BEHAVIOR;


let BITS = new_definition (
   'BITS',
   "BITS(N:NATURAL,M:NATURAL,
             INP:TSI_SIG,OUTP:TSI_SIG) =
      !t. OUTP t = Bits (N,M,INP t)");;
```

Figure 5.10: VHDL architecture and derived HOL for BITS

5.2.3 Words

It will be necessary to examine subsections of some of the "words" used in the Tamarack. A function that does the extraction was defined earlier. It will now be incorporated into an architecture for use in the actual implementation. The original VHDL and the derived HOL are included in Figure 5.10. Joyce's HOL is given in Figure 5.11. The differences in grouping of the parameters are stylistic in nature.

```
let BITS = new_definition
        ('BITS',
         "BITS (n,m) (in:bus,out:bus) =
            !t. out t = Bits (n,m) (in t)");;
```

Figure 5.11: HOL original for BITS

```
use work.Tamarack.all;

package MicroCode is

   type Cntl_type is array (0 to 12) of BOOLEAN;
   type Tst_type is array (0 to 1) of BOOLEAN;

   type Field_type is record
                        Tst_field: Tst_type;
                        Adr_field: TSI;
                      end record;

   type McodeType is record
                       Cntl_word: Cntl_type;
                       Fields: Field_type;
                     end record;

   procedure NextMpc (TOK: in STRING; ADDR: in TSI;
                      RESULT: out Field_type);
   procedure Cntls (Tok1, Tok2: in STRING;
                    RESULT: out Cntl_type);

end MicroCode;
```

Figure 5.12: Package declaration for the microcode

5.3 Microcode and ROM

5.3.1 Package MicroCode

The package declaration that holds the definitions of the microcode types is found in Figure 5.12, and the procedures used to implement it are are given in Figure 5.14 and Figure 5.15. The microcode is modeled by a record of two types. `Cntl_word` represents the 13–bit control word that is the output of the decoder. `Fields` is employed to hold both the test bits and next address information that are also output by the decoder. Procedures

```
new_type_abbrev (‘Cntl_type‘, ":num->bool");;
new_type_abbrev (‘Tst_type‘,":num->bool");;
new_type_abbrev (‘Field_type, ":Tst_type # TSI");;
new_type_abbrev (‘McodeType‘, ":Cntl_type # Field_type");;
new_type_abbrev (‘McodeType_SIG‘, ":time -> McodeType");;
```

Figure 5.13: HOL translation of types

```
procedure Cntls (Tok1, Tok2: in STRING;
                 RESULT: out Cntl_type) is
begin
   RESULT(0)  := (Tok2 = "wmem");
   RESULT(1)  := (Tok1 = "rmem");
   RESULT(2)  := (Tok2 = "wmar");
   RESULT(3)  := (Tok2 = "wpc");
   RESULT(4)  := (Tok1 = "rpc");
   RESULT(5)  := (Tok2 = "wacc");
   RESULT(6)  := (Tok1 = "racc");
   RESULT(7)  := (Tok2 = "wir");
   RESULT(8)  := (Tok1 = "rir");
   RESULT(9)  := (Tok2 = "warg");
   RESULT(10) := (Tok2 = "sub");
   RESULT(11) := (Tok2 = "inc");
   RESULT(12) := (Tok1 = "rbuf");
end Cntls;
```

Figure 5.14: Procedure `Cntls` from package `MicroCode`

```
procedure NextMpc (TOK: in STRING; ADDR: in TSI;
                   RESULT: Field_type) is
begin
   if TOK = "jop" then
      RESULT.Tst_field(0) := TRUE;
      RESULT.Tst_field(1) := FALSE;
      RESULT.Adr_field := ADDR;
   elsif TOK = "jnz" then
      RESULT.Tst_field(0) := FALSE;
      RESULT.Tst_field(1) := TRUE;
      RESULT.Adr_field := ADDR;
   elsif TOK = "jmp" then
      RESULT.Tst_field(0) := TRUE;
      RESULT.Tst_field(1) := TRUE;
      RESULT.Adr_field := ADDR;
   else
      RESULT.Tst_field(0) := FALSE;
      RESULT.Tst_field(1) := FALSE;
      RESULT.Adr_field := ADDR;
   end if;
end NextMpc;
```

Figure 5.15: Function NextMpc from package MicroCode

are used to implement the operations that assign values to these fields. The only difference between the original (Figure 5.16) and derived (Figure 5.17) HOL specifications is in the explicit enumeration of the array slots in each word, as well as the necessity of using extra existential variables to denote record sub–fields.

5.3.2 Architectures ROM and Decode

ROM (Figure 5.18) is used to describe the behavior of the device. A physical representation could be expressed in terms of a two dimensional array, but is unnecessary at the current level of abstraction. The differences in the derived (Figure 5.19) and original (Figure 5.20) HOL are minor and center around the use of VHDL procedures to model the operations Cntls and NextMpc as well as the mapping of processes to separate definitions.

The decoder (Figure 5.21) is the interface between ROM and the data path of the system. It simply breaks MIW into its components and assigns them to the appropriate output signals. The original HOL description (Figure 5.22) did it in one step, but in the derived version (Figure 5.23), the various parts of the record must be itemized before assignment takes place.

```
let Cntls = new_definition (
   'Cntls',
   "Cntls(Tok1:STRING,Tok2:STRING,RESULT:Cntl_type) =
       (RESULT  0 = (Tok2 = 'wmem')) /\
       (RESULT  1 = (Tok1 = 'rmem')) /\
       (RESULT  2 = (Tok2 = 'wmar')) /\
       (RESULT  3 = (Tok2 = 'wpc'))  /\
       (RESULT  4 = (Tok1 = 'rpc'))  /\
       (RESULT  5 = (Tok2 = 'wacc')) /\
       (RESULT  6 = (Tok1 = 'racc')) /\
       (RESULT  7 = (Tok2 = 'wir'))  /\
       (RESULT  8 = (Tok1 = 'rir'))  /\
       (RESULT  9 = (Tok2 = 'warg')) /\
       (RESULT 10 = (Tok2 = 'sub'))  /\
       (RESULT 11 = (Tok2 = 'inc'))  /\
       (RESULT 12 = (Tok1 = 'rbuf'))");;

let NextMpc = new_definition (
   'NextMpc',
   "NextMpc(TOK:STRING,ADDR:TSI,RESULT:Field_type) =
      ? RESULT_Tst_field:Tst_type RESULT_Adr_field:TSI.
        ((TOK = 'jop') => ((RESULT_Tst_field 0 = T) /\
                           (RESULT_Tst_field 1 = F) /\
                           (RESULT_Adr_field = ADDR) /\
                           (RESULT = (RESULT_Tst_field,
                                      RESULT_Adr_field))) |
                .
                .
                .
                          ((RESULT_Tst_field 0 = F) /\
                           (RESULT_Tst_field 1 = F) /\
                           (RESULT_Adr_field = ADDR) /\
                           (RESULT = (RESULT_Tst_field,
                                      RESULT_Adr_field))))");;
```

Figure 5.16: Translation of NextMpc and Cntls

```
let Cntls = new_definition (
        'Cntls',
        "Cntls (tok1,tok2) =
          ((tok2 = 'wmem'),
           (tok1 = 'rmem'),
           (tok2 = 'wmar'),
           (tok2 = 'wpc'),
           (tok1 = 'rpc'),
           (tok2 = 'wacc'),
           (tok1 = 'racc'),
           (tok2 = 'wir'),
           (tok1 = 'rir'),
           (tok2 = 'warg'),
           (tok2 = 'sub'),
           (tok2 = 'inc'),
           (tok1 = 'rbuf'))");;

let NextMpc = new_definition (
        'NextMpc',
        "NextMpc (tok,addr:num) =
          (tok = 'jop') => ((T,F),addr) |
          (tok = 'jnz') => ((F,T),addr) |
          (tok = 'jmp') => ((T,T),addr) |
                           ((F,F),addr)");;
```

Figure 5.17: Original HOL for microcode operations

```
use work.Tamarack.all;
use work.MicroCode.all;

entity ROM is
   port (ADDR: in TSI; DATA: out McodeType);
end ROM;

architecture BEHAVIOR of ROM is
   variable Actual_ROM: McodeType;
begin
   process (ADDR)
   begin
      if ADDR = 0 then
         Cntls("rpc","wmar",Actual_ROM.Cntl_word);
  NextMpc("inc",0,Actual_ROM.Fields);
      elsif ADDR = 1 then
  Cntls("rmem","wir",Actual_ROM.Cntl_word);
  NextMpc("inc",0,Actual_ROM.Fields);
      elsif ADDR = 2 then
  Cntls("rir","wmar",Actual_ROM.Cntl_word);
  NextMpc("jop",0,Actual_ROM.Fields);
      .
      .
      .
      elsif ADDR = 14 then
  Cntls("rbuf","wacc",Actual_ROM.Cntl_word);
  NextMpc("jmp",10,Actual_ROM.Fields);
      else
  Cntls("none","none",Actual_ROM.Cntl_word);
  NextMpc("jmp",10,Actual_ROM.Fields);
      end if;
         DATA <= transport Actual_ROM;
   end process;
end BEHAVIOR;
```

Figure 5.18: VHDL architecture for the ROM

```
let ROM = new_definition (
   'ROM',
   "ROM(ADDR:TSI;DATA:McodeType) =
      ? Actual_ROM_Cntl_word:Cntl_type
        Actual_ROM_Fields:Field_type.
         PROC_ROM (ADDR,DATA,ACTUAL_ROM_Cntl_word,
                   Actual_ROM_Fields)");;

let PROC_ROM = new_definition (
   'PROC_ROM',
   "PROC_ROM (ADDR:TSI,...,Acual_ROM_Fields:Field_type) =
        ((ADDR = 0) =>
             (Cntls('rpc','wmar',Actual_ROM_Cntl_word)  /\
              NextMpc('inc',0,Actual_ROM_Fields))         |
         (ADDR = 1) =>
             (Cntls('rmem','wir',Actual_ROM_Cntl_word)  /\
              NextMpc('inc',0,Actual_ROM_Fields))         |
         (ADDR = 2) =>
             (Cntls('rir','wmar',Actual_ROM_Cntl_word)  /\
              NextMpc('jop',0,Actual_ROM_Fields))         |
         .
         .
         .
         (ADDR = 14) =>
             (Cntls('rbuf','wacc',Actual_ROM_Cntl_word) /\
              NextMpc('jmp',10,Actual_ROM_Fields))        |
             (Cntls('none','none',Actual_ROM_Cntl_word) /\
              NextMpc('jmp',0,Actual_ROM_Fields))) /\
        (DATA = (Actual_ROM_Cntl_word,Actual_ROM_Fields))");;
```

Figure 5.19: HOL translation of the ROM

```
let Microcode = new_definition (
 'Microcode',
 "Microcode n =
   ((n = 0)  => (Cntls ('rpc','wmar'),  NextMpc ('inc',0))  |
    (n = 1)  => (Cntls ('rmem','wir'),  NextMpc ('inc',0))  |
    (n = 2)  => (Cntls ('rir','wmar'),  NextMpc ('jop',0))  |
    %                    JZR                                %
    (n = 3)  => (Cntls ('none','none'), NextMpc ('jnz',10)) |
    %                    JMP                                %
    (n = 4)  => (Cntls ('rir','wpc'),   NextMpc ('jmp',0))  |
    %                    ADD                                %
    (n = 5)  => (Cntls ('racc','warg'), NextMpc ('jmp',12)) |
    %                    SUB                                %
    (n = 6)  => (Cntls ('racc','warg'), NextMpc ('jmp',13)) |
    %                    LD                                 %
    (n = 7)  => (Cntls ('rmem','wacc'), NextMpc ('jmp',10)) |
    %                    ST                                 %
    (n = 8)  => (Cntls ('racc','wmem'), NextMpc ('jmp',10)) |
    %                    NOP                                %
    (n = 9)  => (Cntls ('none','none'), NextMpc ('inc',0))  |
    %                    NOP                                %
    (n = 10) => (Cntls ('rpc','inc'),   NextMpc ('inc',0))  |
    (n = 11) => (Cntls ('rbuf','wpc'),  NextMpc ('jmp',0))  |
    (n = 12) => (Cntls ('rmem','add'),  NextMpc ('jmp',14)) |
    (n = 13) => (Cntls ('rmem','sub'),  NextMpc ('inc',0))  |
    (n = 14) => (Cntls ('rbuf','wacc'), NextMpc ('jmp',10)) |
                (Cntls ('none','none'), NextMpc ('jmp',0)))");;
```

Figure 5.20: Original HOL specification of the ROM

```
use work.Tamarack.all;
use work.MicroCode.all;

entity Decoder is
  port (MIW: in McodeType;
        WMEM, RMEM, WMAR, WPC, RPC, WACC, RACC,
        WIR, RIR, WARG, ALU0, ALU1, RBUF, TEST0,
        TEST1: out BOOLEAN; ADDR: out TSI);
end Decoder;

architecture BEHAVIOR of Decoder is
begin
   WMEM <= transport MIW.Cntl_word(0);
   RMEM <= transport MIW.Cntl_word(1);
   WMAR <= transport MIW.Cntl_word(2);
   WPC  <= transport MIW.Cntl_word(3);
   RPC  <= transport MIW.Cntl_word(4);
   WACC <= transport MIW.Cntl_word(5);
   RACC <= transport MIW.Cntl_word(6);
   WIR  <= transport MIW.Cntl_word(7);
   RIR  <= transport MIW.Cntl_word(8);
   WARG <= transport MIW.Cntl_word(9);
   ALU0 <= transport MIW.Cntl_word(10);
   ALU1 <= transport MIW.Cntl_word(11);
   RBUF <= transport MIW.Cntl_word(12);

   TEST0 <= transport MIW.Fields.Tst_field(0);
   TEST1 <= transport MIW.Fields.Tst_field(1);

   ADDR <= transport MIW.Fields.Adr_field;
end BEHAVIOR;
```

Figure 5.21: VHDL source for the decoder

```
let Decoder = new_definition (
   'Decoder',
   "Decoder(MIW:McodeType,...,ADDR:TSI_SIG) =
     !t. ? Cntl_word Fields.
           MIW = (Cntl_word,Fields) /\
         ? Tst_field Adr_field.
           Fields = (Tst_field,Adr_field) /\
        WMEM t = Cntl_word 0 /\
        .
        .
        .
        ADDR t = Adr_field");;
```

Figure 5.22: Derived HOL for the decoder

```
let Decoder = new_definition (
        'Decoder',
        "Decoder (
          miw:time->^miw_ty,test0,test1,addr,
          wmem,rmem,wmar,wpc,rpc,wacc,racc,wir,rir,
          warg,alu0,alu1,rbuf) =
          !t.
            ((wmem t,rmem t,wmar t,wpc t,rpc t,wacc t,
              racc t,wir t,rir t,warg t,alu0 t,alu1 t,
              rbuf t),
             ((test0 t,test1 t),addr t)) =
            miw t");;
```

Figure 5.23: Original HOL for the decoder

5.4 Primitive System Components

The gates used in the system are trivial translations of the VHDL originals of Figure 5.24. The operations and and or get mapped to /\ and \/ respectively to provide the operations required for ANDGATE and ORGATE. MUX is a simple example of conditional signal assignment. The derived versions in Figure 5.25 are identical to the original HOL specifications of Figure 5.26. The translations of ADDER (Figures 5.27, 5.28, and 5.29) and CheckCntls, which defines the mutual exclusion function for the bus, (Figures 5.30, 5.31, and 5.32) are performed in a similar fashion.

DEL is a simple unit–delay register. Its translation is conducted in the same manner as the basic gates, and is identical to the original HOL specification. REG is more complex, and makes use of the guarded signal assignment to express the same relationship as Joyce's description. The use of the signal kind bus will be discussed in Section 5.6.

5.5 ALU

The ALU (Figure 5.39) makes use of the functions Incn, Addn, and Subn defined in the package Tamarack. Through use of the conditional signal assignment, the input conditions can be tested, and generate a high–impedance (−1) value when invalid control signals are received. Because of the use of this condition, we are forced to describe the operation in a special manner. An existentially quantified variable, high_impedance, is used to represent the value. All that has been done is to remove the explicit reference to −1, and replace it with an abstract one. While the arms of the conditional statement appear different in the derived (Figure 5.40) and original (Figure 5.41) HOL, they express the same relationship.

5.6 Memory

With main memory (Figure 5.36), a heretofore untranslated structure is encountered. It is the declaration of a port of type bus. The declaration is significant from the standpoint of simulation in that it is used to delimit a signal that is disconnected from its driver when the implicit VHDL signal GUARD becomes false [23]. Because memory is expressed as a two–dimensional array, the fetch and store operations are specified in the signal assignment statement itself. In terms of HOL, the bus declaration is used to convert the guarded signal assignment statement into a conditional one if it is not already in that form, and to assign the output to itself as the "else" branch. When translated (Figure 5.37), the relationships are more simply stated than in the original (Figure 5.38), where the external function Update is used for accessing memory.

```
use work.Tamarack.all;

entity ANDGATE is
   port (A, B: in BOOLEAN; OUTP: out BOOLEAN);
end ANDGATE;

architecture BEHAVIOR of ANDGATE is
begin
   OUTP <= transport A and B;
end BEHAVIOR;

use work.Tamarack.all;

entity ORGATE is
   port (A, B: in BOOLEAN; OUTP: out BOOLEAN);
end ORGATE;

architecture BEHAVIOR of ORGATE is
begin
   OUTP <= transport A or B;
end BEHAVIOR;

use work.Tamarack.all;

entity MUX is
   port (CNTL: in BOOLEAN; A, B: in TSI;
         OUTP: out TSI);
end MUX;

architecture BEHAVIOR of MUX is
begin
   OUTP <= transport B when CNTL else A;
end BEHAVIOR;
```

Figure 5.24: VHDL description of the basic gates

```
let ANDGATE = new_definition (
   'ANDGATE',
   "ANDGATE(A:BOOLEAN_SIG,B:BOOLEAN_SIG,OUTP:BOOLEAN_SIG) =
         !t. OUTP t = A t \/ B t");;

let ORGATE = new_definition (
   'ORGATE',
   "ORGATE(A:BOOLEAN_SIG,B:BOOLEAN_SIG,OUTP:BOOLEAN_SIG) =
        !t. OUTP t = A t \/ B t");;

let MUX = new_definition (
   'MUX',
   "MUX(CNTL:BOOLEAN_SIG,A:TSI_SIG,B:TSI_SIG,OUTP:TSI_SIG) =
      !t. OUTP t = (CNTL t => B t | A t)");;
```

Figure 5.25: Translated HOL versions of the basic gates

```
let AND = new_definition
   ('AND',
    "AND (a:wire,b:wire,out:wire) = !t. out t = a t /\ b t");;

let OR = new_definition
   ('OR',
    "OR (a:wire,b:wire,out:wire) = !t out t = a t \/ b t");;

let MUX = new_definition
   ('MUX',
    "MUX (cntl:wire,a:bus,b:bus,out:bus) =
      !t. out t = (cntl t => b t | a t)");;
```

Figure 5.26: Original HOL versions of the basic gates

```
use work.Tamarack.all;

entity ADDER is
   generic (N: NATURAL);
   port (A, B: in TSI; OUTP: out TSI);
end ADDER;

architecture BEHAVIOR of ADDER is
begin
   OUTP <= transport Addn (N, A, B);
end BEHAVIOR;
```

Figure 5.27: VHDL description of ADDER

```
let ADDER = new_definition (
   'ADDER',
   "ADDER (N:NATURAL,...,OUTP:TSI_SIG) =
      OUTP t = Addn (N,A t,B t)");;
```

Figure 5.28: Derived HOL description of ADDER

```
let ADDER = new_definition
     ('ADDER',
      "ADDER n (a:bus,b:bus,out:bus) =
         !t. out 1 = ADDn n (a t,b t)");;
```

Figure 5.29: Original HOL description of ADDER

```
use work.Tamarack.all;

entity CheckCntls is
   port (RMEM, RPC, RACC, RIR, RBUF: in BOOLEAN;
         P: out BOOLEAN);
end CheckCntls;

architecture BEHAVIOR of CheckCntls is
begin
  P <= transport not(RPC or RACC or RIR or RBUF) when RMEM else
                 not(RACC or RIR or RBUF) when RPC else
                 not(RIR or RBUF) when RACC else
                 not RBUF when RIR else
                 TRUE;
end BEHAVIOR;
```

Figure 5.30: VHDL description of the control checker

```
let CheckCntls = new_definition (
   `CheckCntls`,
   "CheckCntls (RMEM:BOOLEAN_SIG,...,P:BOOLEAN_SIG) =
     !t.
       P t = (RMEM t => ~(RPC t \/ RACC t \/ RIR t \/ RBUF t) |
              RPC t => ~(RACC t \/ RIR t \/ RBUF t) |
              RACC t => ~(RIR t \/ RBUF t) |
              RIR t => ~RBUF t | T");;
```

Figure 5.31: Derived HOL description of the control checker

```
let CheckCntls = new_definition
  (`CheckCntls`,
   "CheckCntls (rmem,rpc,racc,rir,rbuf,P) =
     !t.
       P t =
         ((rmem t) => (~(rpc t \/ racc t \/ rir t \/ rbuf t)) |
         ((rpc t)  => (~(racc t \/ rir t \/ rbuf t)) |
         ((racc t) => (~(rir t \/ rbuf t)) |
         ((rir t)  => (~(rbuf t)) | T))))");;
```

Figure 5.32: Original HOL description of the control checker

```
use work.Tamarack.all;

entity DEL is
   port (INP: in TSI; OUTP: out TSI);
end DEL;

architecture BEHAVIOR of DEL is
begin
   OUTP <= transport INP after 1ns;
end BEHAVIOR;

use work.Tamarack.all;

entity REG is
   port (W, R: in BOOLEAN; INPT: in TSI;
  BUSS: out BUSTYPE bus; OUTP: inout TSI; P: in BOOLEAN);
end REG;

architecture BEHAVIOR of REG is
   signal GUARD: BOOLEAN := FALSE;
begin
   B: block (P and R)
      begin
         OUTP <= transport INPT after 1 ns when W else
                           OUTP;
         BUSS <= guarded transport OUTP;
      end block B;
end BEHAVIOR;
```

Figure 5.33: VHDL specifications of the registers

```
let DEL = new_definition (
   'DEL'
   "DEL (INP:TSI_SIG,OUTP:TSI_SIG) =
        !t. OUTP t = INP (t - 1)");;

let REG = new_definition (
   'REG'
   "REG (W:BOOLEAN_SIG,...,P:BOOLEAN_SIG) =
      !t. OUTP t = (W t => INPT (t - 1) | OUTP t) /\
          BUSS t = (P t /\ R t => OUTP t | BUSS t)");;
```

Figure 5.34: Derived HOL specifications of the registers

```
let DEL = new_definition
     ('DEL',
      "DEL (in:bus,out:bus) = !t. out (t+1) = in t");;

let REG = new_definition
     ('REG',
      "REG ((w:wire,r:wire,in:bus,bus:bus,out:bus),P) =
       !t.
          ((out (t+1) = (w t => in t | out t)) /\
           (P t ==> r t ==> (bus t = out t)))");;
```

Figure 5.35: Original HOL specifications for the registers

```
use work.Tamarack.all;

entity MEM is
   port (W,R: in BOOLEAN;
         ADDR: in TSI;
         BUSS: inout BUSTYPE bus;
         P: in BOOLEAN;
         MEMORY: inout MEMTYPE);
end MEM;

architecture BEHAVIOR of MEM is
begin
   B: block (P and R)
      begin
         MEMORY(ADDR) <= transport BUSS after 1ns when W else
                                   MEMORY(ADDR);
         BUSS <= guarded transport MEMORY(ADDR);
      end block B;
end BEHAVIOR;
```

Figure 5.36: VHDL description of main memory

5.7 Major Subsystems

With the specification of the logic for the microcode program counter (MPC), the first use of external architectural components is encountered. The declarations for them are shown in Figure 5.42, but will be omitted from later architectures. The instantiations of HWC are concurrent procedure calls, and do not need to be labeled. In HOL, the components are instantiated in the derived version exactly as those in Joyce's original. Internally declared signals are represented as existentially quantified variables. The control unit, data path, and implementation are treated in a like manner, and are unchanged in translated form from the original specifications.

```
let MEM = new_definition (
   'MEM',
   "MEM(W:BOOEAN_SIG,R:BOOLEAN_SIG,ADDR:TSI_SIG,
        BUSS:BUSTYPE_SIG,P:BOOLEAN_SIG,
        MEMORY:MEMTYPE_SIG) =
      !t. (MEMORY t (ADDR) = (W t => BUSS (t-1) |
                                     MEMORY t (ADDR)) /\
           BUSS t = ((P t /\ R t) => MEMORY t (ADDR) |
                                     BUSS t))");;
```

Figure 5.37: Derived HOL description of main memory

```
let Update = new_definition
     ('Update',
      "Update (s:*->**,x,y) = \x. (x = x') =¿ y — (s x')");;'

let MEM = new_definition
     ('MEM',
      "MEM n ((w:wire,r:wire,addr:bus,bus:bus),(P,mem:mem)) =
        !t.
          (mem (t+1) = (w t => Update (mem t,addr t,bus t) |
                               mem t)) /\
          (P t ==> r t ==> (bus t = mem t (addr t)))");;
```

Figure 5.38: Original HOL description of main memory

```
use work.Tamarack.all;

entity ALU is
  generic (N: NATURAL);
  port (F0, F1: in BOOLEAN;
        A: in TSI;
        B: in BUSTYPE;
        OUTP: out TSI);
end ALU;

architecture BEHAVIOR of ALU is
begin
  OUTP <= transport
              Incn (N, B) when ((not F0) and F1) else
              Addn (N, A, B) when (not F0 and not F1) else
              Subn (N, A, B) when (F0 and (not F1)) else
              -1;
end BEHAVIOR;
```

Figure 5.39: VHDL description of the ALU

```
let ALU = new_definition (
   'ALU',
   "ALU(N:NATURAL,F0:BOOLEAN_SIG,F1:BOOLEAN_SIG,
        A:TSI_SIG,B:BUSTYPE_SIG,OUTP:TSI_SIG) =
      !t. ? high_impedance.
        OUTP t =
          ((~F0 t /\ F1 t)  => Incn (N,B t)      |
           (~F0 t /\ ~F1 t) => Addn (N,A t,B t) |
           (F0 t /\ ~F1 t)  => Subn (N,A t,B t) |
           high_impedance)");;
```

Figure 5.40: Derived HOL description of the ALU

```
let ALU = new_definition
    ('ALU',
     "ALU n (f0:wire,f1:wire,a:bus,b:bus,out:bus) =
       !t.
        ?w.
         out t =
            (((f0 t,f1 t) = (T,T)) => w |
             ((f0 t,f1 t) = (F,T)) => INCn n (b t) |
             ((f0 t,f1 t) = (F,F)) => ADDn n (a t,b t) |
                                      SUBn n (a t,b t))");;
```

Figure 5.41: Original HOL description of the ALU

```
entity MpcUnit is
   port (TEST0, TEST1, ZEROFLAG: in BOOLEAN;
         OPCODE,ADDR: in TSI; MPC: inout TSI);
end MpcUnit;

architecture BEHAVIOR of MpcUnit is
   component MUX
      port (CNTL: in BOOLEAN; A, B: in TSI; OUTP: out TSI);
   end component;
   for all: MUX use entity work.MUX(BEHAVIOR);

   component ANDGATE
      port (A, B: in BOOLEAN; OUTP: out BOOLEAN);
   end component;
   for all: ANDGATE use entity work.ANDGATE(BEHAVIOR);

   component ORGATE
      port (A, B: in BOOLEAN; OUTP: out BOOLEAN);
   end component;
   for all: ORGATE use entity work.ORGATE(BEHAVIOR);

   component DEL
      port (INP: in TSI; OUTP: out BUSTYPE);
   end component;
   for all: DEL use entity work.DEL(BEHAVIOR);

   component ADDER
      generic (N: NATURAL);
      port (A, B: in TSI; OUTP: out TSI);
   end component;
   for all: ADDER use entity work.ADDER(BEHAVIOR);

   signal ZERO, ONE, THREE: TSI;
   signal W1, W2: BOOLEAN;
   signal B1, B2, B3, B4, B5: TSI;
```

Figure 5.42: Declarations in VHDL description of the MPC unit

```
begin
   and1: ANDGATE port map (TEST1, ZEROFLAG, W1);
   or1:  ORGATE port map (TEST0, W1, W2);
   Mux1: MUX port map (TEST1, OPCODE, ADDR, B1);
   Mux2: MUX port map (W2, MPC, B1, B2);
         HWC(0,ZERO);
         HWC(3,THREE);
   Mux3: MUX port map (TEST1, THREE, ZERO, B3);
         HWC(1,ONE);
   Mux4: MUX port map (W2, ONE, B3, B4);
   add1: ADDER generic map (4)
               port map (B2, B4, B5);
   del1: DEL port map (B5, MPC);
end BEHAVIOR;
```

Figure 5.43: Main body of the VHDL description of the MPC unit

```
let MpcUnit = new_definition (
   'MpcUnit',
   "MpcUnit(TEST0:BOOLEAN_SIG,TEST1:BOOLEAN_SIG,
            ZEROFLAG:BOOLEAN_SIG,OPCODE:TSI_SIG,
            ADDR:TSI_SIG,MPC:TSI_SIG) =
     ? ZERO ONE THREE W1 W2 B1 B2 B3 B4 B5.
         ANDGATE(TEST1,ZEROFLAG,W1) /\
         ORGATE(TEST0,W1,W2)        /\
         MUX(TEST1,OPCODE,ADDR,B1)  /\
         MUX(W2,MPC,B1,B2)          /\
         HWC(0,ZERO)                /\
         HWC(3,THREE)               /\
         MUX(TEST1,THREE,ZERO,B3)   /\
         HWC(1,ONE)                 /\
         MUX(W2,ONE,B3,B4)          /\
         ADDER(4,B2,B4,B5)          /\
         DEL(B5,MPC)");;
```

Figure 5.44: Derived HOL description of the MPC unit

```
let MpcUnit = new_definition (
        'MpcUnit',
        "MpcUnit (test0,test1,zeroflag,opcode,addr,mpc) =
          ?w1 w2 zero one three b1 b2 b3 b4 b5.
            AND (test1,zeroflag,w1) /\
            OR (test0,w1,w2) /\
            MUX (test1,opcode,addr,b1) /\
            MUX (w2,mpc,b1,b2) /\
            HWC 0 zero /\
            HWC 3 three /\
            MUX (test1,three,zero,b3) /\
            HWC 1 one /\
            MUX (w2,one,b3,b4) /\
            ADDER 4 (b2,b4,b5) /\
            DEL (b5,mpc)");;
```

Figure 5.45: Original HOL description of the MPC unit

```
use work.Tamarack.all;
use work.Microcode.all;

entity CntlUnit is
   port (ZEROFLAG: in BOOLEAN; OPCODE: in TSI;
         WMEM, RMEM, WMAR, WPC, RPC, WACC, RACC,
         WIR, RIR, WARG, ALUO, ALU1, RBUF: out BOOLEAN;
         MPC: inout TSI);
end CntlUnit;

architecture BEHAVIOR of CntlUnit is
   <component declarations>

   signal MIW: McodeType;
   signal TESTO, TEST1: BOOLEAN;
   signal ADDR: TSI;

begin
   romP: ROM port map (MPC, MIW);
   decP: Decoder port map (MIW, WMEM, RMEM, WMAR, WPC,
                           RPC, WACC, RACC, WIR, RIR, WARG,
                           ALUO, ALU1, RBUF, TESTO, TEST1,
                           ADDR);
   mpcP: MpcUnit port map (TESTO, TEST1, ZEROFLAG,
                           OPCODE, ADDR, MPC);
end BEHAVIOR;
```

Figure 5.46: VHDL description of the control unit

```
let CntlUnit = new_definition (
   'CntlUnit',
   "CntlUnit(ZEROFLAG:BOOLEAN_SIG,OPCODE:TSI_SIG,
             WMEM:BOOLEAN_SIG,...,MPC:TSI_SIG) =
     ? MIW TESTO TEST1 ADDR.
        ROM (MPC,MIW) /\
        Decoder (MIW,WMEM,RMEM,WMAR,WPC,RPC,WACC,RACC,
                 WIR,RIR,WARG,ALUO,ALU1,RPC,
                 TESTO,TEST1,ADDR) /\
        MpcUnit (TESTO,TEST1,ZEROFLAG,OPCODE,ADDR,MPC)");;
```

Figure 5.47: Derived HOL description of the control unit

```
let CntlUnit = new_definition (
      'CntlUnit',
      "CntlUnit (
        (zeroflag,opcode,
         wmem,rmem,wmar,wpc,rpc,wacc,racc,
         wir,rir,warg,alu0,alu1,rbuf),
        mpc) =
        ?miw test0 test1 addr.
          ROM Microcode (mpc,miw) /\
          Decoder (
            miw,test0,test1,addr,
            wmem,rmem,wmar,wpc,rpc,wacc,racc,
            wir,rir,warg,alu0,alu1,rbuf) /\
          MpcUnit (test0,test1,zeroflag,opcode,addr,mpc)");;
```

Figure 5.48: Original HOL description of the control unit

```
use work.Tamarack.all;
entity DataPath is
   generic (N: NATURAL);
   port (WMEM, RMEM, WMAR, WPC, RPC, WACC, RACC,
         WIR, RIR, WARG, ALUO, ALU1, RBUF: in BOOLEAN;
         ZEROFLAG: out BOOLEAN; OPCODE: out TSI;
         MEMORY: inout MEMTYPE;
         MAR, PC, ACC, IR, ARG, BUF: inout TSI);
end DataPath;

architecture BEHAVIOR of DataPath is
   <component declarations>

   signal P, POWR, GRND: BOOLEAN; signal BUSS: BUSTYPE bus;
   signal ADDR: NATURAL; signal ALU: TSI;
begin
   chkP: CheckCntls port map (RMEM,RPC,RACC,RIR,RBUF,P);
   memP: MEM port map (WMEM,RMEM,ADDR,BUSS,P,MEMORY);
   marP: REG port map (WMAR,GRND,BUSS,BUSS,MAR,P);
   bit1: BITS generic map (0,N)
              port map (MAR,ADDR);
   pc_P: REG port map (WPC,RPC,BUSS,BUSS,PC,P);
   accP: REG port map (WACC,RACC,BUSS,BUSS,ACC,P);
         TNZ(ACC,ZEROFLAG);
   ir_P: REG port map (WIR,RIR,BUSS,BUSS,IR,P);
   bit2: BITS generic map (N,3)
              port map (IR,OPCODE);
   argP: REG port map (WARG,GRND,BUSS,BUSS,ARG,P);
   aluP: A_L_U generic map (N+3)
               port map (ALUO,ALU1,ARG,BUSS,ALU);
   bufP: REG port map (POWR,RBUF,ALU,BUSS,BUF,P);
   pwr1: PWR port map (POWR);
   gnd1: GND port map (GRND);
end BEHAVIOR;
```

Figure 5.49: VHDL description of the data path

```
let DataPath = new_definition (
   'DataPath',
   "DataPath(WMEM:BOOLEAN_SIG,...,BUF:TSI_SIG) =
      ? P POWR GRND BUSS ADDR ALU.
         CheckCntls(RMEM,RPC,RACC,RIR,RBUF,P) /\
         .
         .
         .
         GND(GRND)");;
```

Figure 5.50: Derived HOL description of the data path

Chapter 6

Conclusions and Further Research

In the course of the description of three translation schemes, the operational semantics of a subset of VHDL have been specified. The significance of the effort lies in the fact that no such description has previously been made available. Certainly, the information in [23] is helpful in the understanding of the language and has been used to implement it, but in order to do any meaningful work on the verification of VHDL designs, a more formal representation is needed.

The translation methods are also relevant outside the context of the formal specification of a hardware description language. It is wasteful of both time and energy to attempt to formally verify a first design for a system. Some simulation should be done to determine if it appears to exhibit the desired behavior. After the basic functionality of the circuit has been determined with some certainty, formal verification is attempted. By using both simulation and verification, the best of both worlds can then be obtained. Simulation can be used to give basic results early in the design cycle, and formal verification can be done on a mature design to ensure correctness. VHDL coupled with VAL accomplishes the first objective, and HOL the second.

To accomplish the specification of the entire language, a formal definition of the VHDL simulation environment should be made. HOL presents a suitable basis, and the theory that results could be used in conjunction with another theory encompassing VHDL types to aid in the development of better methodologies for the transformation of VHDL descriptions into HOL specifications. Further, the work would be relevant to the designers of VHDL systems, in that they would then have "the" definition of what a simulator should do. Because of the commercial availability of only one complete simulator at the present time, the problem of diverging language implementations has yet to be encountered, but will surface in the near future as more vendors begin to market VHDL tool sets.

The translated VAL versions of VHDL specifications were constructed in a mechanical fashion, and allowed for the examination through simulation of the underlying timing model used by the language. The information is useful in the overall understanding of VHDL, and can be incorporated into a more concrete definition of the simulation environment. The goal would be to have VAL timing annotations embedded into the various components that make up any future VHDL test suite. New simulators could then be evaluated more quickly and thoroughly.

In the translation of VHDL descriptions into HOL specifications, several problems were encountered. Some were trivial, others were of greater significance; and most of them dealt with the representation of complex types. The multiple uses of `new_type_abbrev` did not actually create any new types, but simply established a shorthand notation to describe the signals and variables used in a given system. Records presented a unique

problem in that the whole structure of the type had to be re–iterated during the course of a definition. Also, the representation of any numerical value other than those associated with the natural numbers proved to be difficult in some cases or impossible in others (i.e. reals). These issues would have to be addressed in the definitional theories just mentioned through the specification of new types.

The Tamarack example demonstrated that a non–trivial VHDL design could be converted into HOL. The methods used were for the most part mechanical, but in many cases the simplicity of the original HOL specification was lost. The problems resulted from the use of complex structures such as arrays and records as well as unrepresentable values (-1). The last of these introduced non–mechanical methods into the translation scheme by forcing the use of existentially quantified variables to specify them. The refinement of the way in which the conversions are effected is therefore essential to the future of the verification of VHDL designs. To accomplish the task, more examples of real systems need to be examined in the same fashion as the Tamarack, and the lessons learned from them included in the development of subsequent algorithms.

More can also be learned by further refinement of the Tamarack into a true reflection of the actual chip. Transport delay, while simple to use, may not express more complex relationships where inertial delay may be more appropriate. Further, the timing specifications currently center around one nanosecond–delay registers, and a more precise specification of the timing behavior of the various components is needed. The result could again be converted into HOL, and proved to be a special case of the more general design.

References

[1] Larry M. Augustin, Benoit A. Gennart, Youm Huh, David C. Luckham, and Alec C. Stanculescu. An Overview of VAL. Technical Report CSL-TR-88-367, Stanford University, Stanford, California, October 1988.

[2] Harry G. Barrow. VERIFY: A Program for Proving Correctness of Digital Hardware Designs. *Artificial Intelligence*, 24:437–491, 1984.

[3] G. Birtwistle and P. A. Subrahmanyam. *VLSI Specification, Verification, and Synthesis.* Academic Press, New York, 1987.

[4] Graham Birtwistle, Jeff Joyce, Breen Liblong, Tom Melham, and Rick Schediwy. Specification and VLSI Design. In *Formal Aspects of VLSI Design*, pages 83–97. Elsevier Publishers (North-Holland), 1986.

[5] D. Borrione. *From HDL Descriptions to Guaranteed Correct Circuit Design.* North-Holland, Amsterdam, 1987.

[6] Albert Camilleri. Executing Behavioral Definitions in Higher–Order Logic. Technical Report No. 140, University of Cambridge Computer Laboratory, 1988.

[7] Albert Camilleri, Mike Gordon, and Tom Melham. Hardware Verification using Higher–Order Logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. Elsevier Science Publishers B. V. (North-Holland), 1987.

[8] Avra Cohn. A Proof of Correctness of the VIPER Microprocessor: The First Level. In *VLSI Specification, Verification, and Synthesis.* Kluwer Academic Publishers, Boston, 1988.

[9] W. J. Cullyer. VIPER - Correspondence Between the Specification and the "Major State Machine". Technical Report 86004, Royal Signals and Radar Establishment, Malvern, Worchestershire, England, January, 1986.

[10] Mike Gordon. Proving a Computer Correct. Technical Report 42, Cambridge University Computer Laboratory, 1983.

[11] Mike Gordon. Why Higher Order Logic is a Good Formalism for Specifying and Verifying Hardware. In *Formal Aspects of VLSI Design.* Elsevier Scientific Publishers, 1986.

[12] Mike Gordon. A Proof Generating System for Higher–Order Logic. Technical Report 103, University of Cambridge Computer Laboratory, 1987.

[13] Mike Gordon. *The HOL Manual.* Computer Laboratory, Cambridge University, 1987.

[14] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.

[15] Jeffery Joyce. Using Higher–Order Logic to Specify Computer Hardware and Architecture. In *Proceedings of the IFIP TC10 Working Conference on Design Methodology in VLSI and Computer Architecture*, Amsterdam, 1988. North-Holland.

[16] Jeffery J. Joyce. Tamarck Microprocessor Proof. Example provided with the HOL system, 1985.

[17] Jeffery J. Joyce. Formal Verification and Implementation of a Microprocessor. In *VLSI Specification, Verification, and Synthesis.* Kluwer Academic Publishers, Boston, 1988.

[18] Jeffery J. Joyce. Tamarck Microprocessor Proof. Example provided with the HOL system, 1988.

[19] Jeffery J. Joyce. Formal Specification and Verification of Synthesized MOS Structures. To appear in the proceedings of VLSI–89 (Munich), July 1989.

[20] G. Milne and P. A. Subrahmanyam. *Formal Aspects of VLSI Design.* Academic Press, New York, 1987.

[21] George Milne. Hardware Verification in Higher–Order Logic. Unpublished notes, 1989.

[22] Paul Naish and Peter Bishop. *Designing ASICS.* Ellis Horwood, Ltd., Chichester, 1988.

[23] Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual.* IEEE Press, New York, 1988.

[24] Larry Paulson. A Higher Order Implementation of Rewriting. In *Science of Computer Programming*, volume 3, pages 119–149. North-Holland, 1983.

[25] Robin L. Steele. An Expert System Application in Semicustom VLSI Design. *NCR Journal*, 1(1), 1987.