HDL to Verification Logic Translator

J. W. Gambles and P. J. Windley
NASA Space Engineering Research Center for VLSI Systems Design
University of Idaho, Moscow, Idaho 83843
jgambles@groucho.mrc.uidaho.edu, 208-885-9041
windley@panther.cs.uidaho.edu, 208-885-6501

Abstract -**The increasingly higher number of transistors possible in VLSI circuits compounds the difficulty in insuring correct designs. As the number of possible test cases required to exhaustively simulate a circuit design explodes, a better method is required to confirm the absence of design faults. Formal verification methods provide a way to prove, using logic, that a circuit structure correctly implements its specification. Before verification is accepted by VLSI design engineers, the stand alone verification tools that are in use in the research community must be integrated with the CAD tools used by the designers.**

One problem facing the acceptance of formal verification into circuit design methodology is that the structural circuit descriptions used by the designers are not appropriate for verification work and those required for verification lack some of the features needed for design. We offer a solution to this dilemma: an automatic translation from the designers' HDL models into definitions for the higher-ordered logic (HOL) verification system. The translated definitions become the low level basis of circuit verification which in turn increases designers confidence in the correctness of higher level behavioral models.

1 Introduction

As higher transistor counts increase the complexity of VLSI circuits and the number of potential test cases explode, traditional simulation methods can expose only a fraction of design faults – not guarantee their absence. Formal verification methods, which *prove* circuit correctness, will play an important role in design fault exclusion. It is common in modern design methodologies to utilize abstract circuit models in a hierarchical design:

- An architectural model (i.e. highly abstract) can be used to simulate an entire system, at an early date, to help confirm that the system specification truly meets the customers needs.
- In a top-down design, a model of the system's architecture is refined to a less abstract model, and this decomposition process proceeds iteratively from algorithmic description, to large functional blocks, to detailed logic, and right down to the circuit level.
- After the circuit structure is modeled and designed, the logic simulation of complex systems can become very slow. Simulations run faster using behavioral models.

A problem with these design approaches is that there is no *formal* way to relate a circuit's structural model to its abstract behavioral model. Formal verification allows these models

to be related through mathematical analysis so that designers can enjoy increased confidence that behavioral models are *correct* abstractions of their structure. Before formal verification is accepted by design engineers, stand alone verification tools that are used in academic research must be integrated with the CAD tools used by VLSI designers.

The hardware description languages (HDL) used by VLSI CAD tools can provide the link between these tools and the verification environment. Engineers can design using their HDL and the models can be automatically translated for use in the verification tool. The translation process consists of two steps.

- Recognizing the syntax of the HDL.
- Constructing the translation from the syntax to the HDL's semantic domain.

The parser, for recognizing the syntax, and the translation semantic construction functions can be built directly into the verification system.

The NOVA simulation engine, one of the CAD tools being developed and used at the NASA Space Engineering Research Center (SERC) for VLSI Systems Design, located on the University of Idaho campus, uses the BOLT (Block Oriented Logic Translator) HDL. BOLT was chosen for this research because it provides ready access to many real-world VLSI designs at the SERC. This paper presents a translator from BOLT to the HOL theorem proving system.

Much has been published about theories for modeling MOS circuits in a verification environment [4, 5, 7, 11, 15]. Our work linking verification with VLSI design tools is related, but has a different motivation. While we are concerned that the model accurately reflect the true behavior of the devices being specified, we must also be concerned that the HOL circuit primitive definitions are consistent with the BOLT primitives used in NOVA. Correct modeling of MOS circuits requires a complex multi-valued, multi-strength data type for signal values[3]. Reasoning about such a signal value system can be done in HOL, where the signal value type (`STATE`) definition and a collection of theorems about manipulating `STATE` values is known collectively as the `STATE` theory[6]. Other work has been published dealing with translating HDLs to verification logics [2, 14]. Our interest in tying HOL to the NOVA simulator has motived our work to include lower level structures (i.e. multi-strength signal values and their resolution functions) where these problems have been largely ignored by others.

2 HOL

HOL, an acronym for higher-order logic, is a general theorem proving system developed at the University of Cambridge [4, 8] based on Church's theory of simple types. Higher-order logic is suitable for specifying all aspects of hardware, including both structure and behavior [8, 10]. In using higher-order logic, predicates are defined to represent both circuit primitives and behavioral definitions [4]. First-order logic is well suited to represent simple combinational circuits, but not sequential circuits. In higher-order logic, variables are allowed to range over functions and predicates, which makes it possible to represent sequential circuit behavior

[10]. HOL is not an automated theorem prover but is more than simply a proof checker. It could, more appropriately, be called a proof assistant.

The HOL system is implemented on top of Cambridge LCF, which is a direct descendant of the work of Robin Milner [8]. Milner originally developed an approach to mechanizing logic for a system called Logic of Computable Functions (LCF) designed for reasoning about higher-order recursively defined functions. The LCF meta-language is called ML, a functional programming language.

3 The BOLT to HOL Translator

The BOLT to HOL translator is comprised of a syntax parser, built using a parser-generator tool included with version 2.0 of the HOL system[13], and a set of ML functions that construct the semantics of the parsed BOLT syntax into HOL definition terms. The semantic construction functions were written *ad hoc*.

3.1 The Syntax Parser

The parser-generator takes as input a grammar representing the formal syntax of BOLT given in a modified Backus Naur Form (BNF) notation similar to Prolog's definite clause grammar (DCG) [13]. The output of the generator is a ML program that recognizes the HDL syntax and makes appropriate calls to the ML semantic construction functions, whose names are included as action symbols in the input grammar. The BOLT syntax is defined in [1]. The HOL parser-generator library developed at the University of Cambridge was found to be very useful in building the syntax recognizer portion of the translator. For example the syntax of a statement-body, as given in the BOLT manual, is:

```
BEGIN

[ { Module-Invocation | Case-Statement } ... ]

END ;
```

Where upper-case words are keywords, the expression [ ] contains an optional item that may be omitted, { } indicates choose one of the enclosed items, and ... indicates that an item may be repeated any number of times is entered into the parser-generator input grammar as the following recursive production:

```
statement_body  --> [BEGIN] invocation_list [END] [;]

invocation_list --> mod_invocation invocation_list |
                    case_statement invocation_list |
                    □.
```

3.2 The Semantic Construction Functions

Because no formal semantic definitions for BOLT exist, the semantic construction functions have been initially written in an *ad hoc* fashion. In order to construct a HOL definitional translation the following data from the BOLT module is required:

1. The module name from the BOLT module declaration. The same name is used for the HOL definition.
2. The set of ports declared to be external to the top module in the BOLT declaration. The signals on these ports must be universally quantified in the HOL definition.
3. The names of the component modules that are invoked inside the BOLT module. Each of these module invocations will cause an identically named HOL predicate to be conjoined in the HOL definition predicate.
4. The set of ports declared to be an output of any invoked module. This set will be used to determine the signals that are driven by more than one device. A `JOIN` predicate must be added to the HOL definition to resolve all interconnected outputs. This set will also be used in the identification of the internal ports that are to be hidden. In BOLT, the ports are not explicitly declared to be module input or outputs. By *convention* the outputs are listed before the module name and the inputs are listed following the module name. **Our translator relies on conformance to this convention.**
5. The set of ports declared to be an input to any invoked module. The union of this set and the set of invoked module output ports defines the set of all ports in the module. The set difference of the set of all ports minus the set of externally declared ports is used to define the set of internal signals that must be existentially quantified in the HOL definition.
6. The only module parameter with any meaning to our translation is the `STR` parameter, which in BOLT is used to define the output strength of an invoked module. If no `STR` parameter is included in the module invocation then the default strength is active.

As the BOLT syntax is parsed, the ML functions construct a data structure from the parsed tokens. It is this structure that is used for the creation of the HOL definition terms once the BOLT module `END;` statement is found. As the BOLT syntax is parsed, the ML functions construct a data structure from the parsed tokens. It is this structure that is used for the creation of the HOL definition terms once the BOLT module `END;` statement is parsed. The data structure is implemented as a list of lists of lists of strings. The form of the structure is:

$$
\begin{aligned}
Structure &\triangleq head : Header; body : Body \\
Header &\triangleq name : Identifier; ext\_out, ext\_in, int\_out, int\_in : Nodes \\
Identifier &\triangleq id : String \\
Nodes &\triangleq Identifiers^{*} \\
Body &\triangleq Invocation^{+} \\
Invocation &\triangleq name : Identifier; out, in : Nodes; param : STR \\
STR &\triangleq Identifier^{*}
\end{aligned}
$$

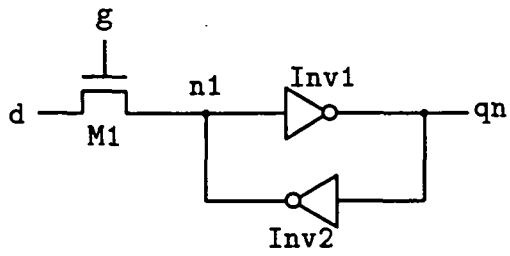


Figure 1: Latch Schematic

In this grammar, the asterisk ("Kleene star") has the standard language theory meaning — a list with zero, one, or more elements [12]. The plus means a list with one or more elements. The first sublist is the header part. It contains the module name and node lists corresponding to the external outputs, external inputs, internal outputs, and internal inputs. The body part is a list of component module invocations. Each invocation contains a module name, a list of output ports, a list of input ports, and a possibly empty paramenter list containing device output strength information. An example structure is shown in Section 4.5.

As each new module invocation is parsed, the module name, output node names, input node names, and optional output strength parameters are added to the data structure. Additionally, a check is made to see if the invoked module output node(s) are already a member of the set of internal output nodes for the current module. If it is, then two outputs are connected to drive the signal value on that node and a `join` resolution function from the `STATE` theory is required[6]. The `join` function is added by renaming the first instance of that output node name to the decorated (primed) variation of the name and the current invocation output is given the double-decorated node name variation. An invocation of `JOIN` is then added to the end of the data structure where the output of the `JOIN` is the original node name and the inputs are the new decorated and double-decorated nodes. If either the decorated or double-decorated names are already used then the first two unused decoration variations are added. A new blank sub-list is also appended to the end of the structure in anticipation of the next module invocation.

When the `END;` statement is encountered, the set of external output nodes unioned with the external input nodes are universally quantified in the resulting HOL definition. The set of internal nodes subtract the external nodes are existentially quantified (hidden). HOL terms are then generated by matching each invoked module with a previously defined HOL constant whose name and type both match the structure built by the parser and construction functions. The terms from all of the invoked modules are conjoined to complete the HOL definition for the current module. A stack is maintained for current module data structures so that embedded BOLT modules can be properly defined and translated.

4 Translator Demonstration

A data latch, implemented with gate level and pass transistor primitives, is used to demonstrate the translator (Figure 1). This circuit is interesting because without a signal value

representation and resolution function that realizes output dominance this circuit cannot be correctly modeled. Fundamental to the operation of this circuit is that the output strength of pass-transistor M1 dominates the output of inverter Inv2 to force node n1 to the state of the input d while the gate g is 1 (high voltage). The feedback inverter Inv2 acts to store the state, by dominating the pass-transistor after the gate goes to 0, turning the transistor off.

4.1 The BOLT Structural Description

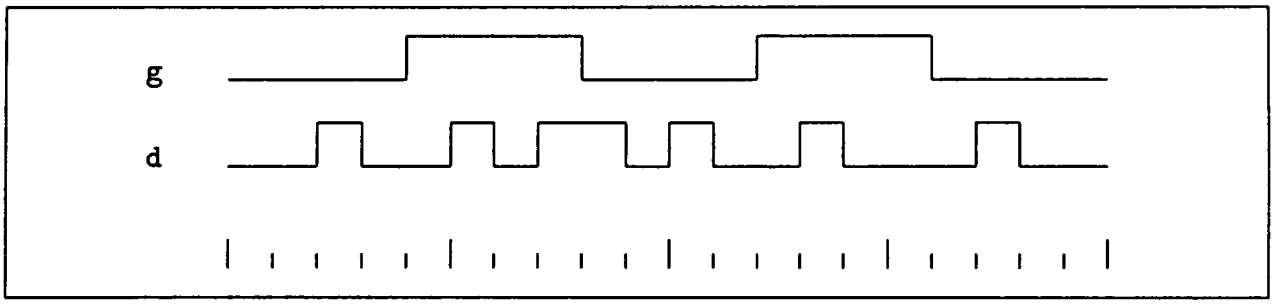
A BOLT description of the latch is:

```
MODULE qn .LATCH g d;
  BEGIN
    n1  .NTRAN g d;
    qn  .INVR  n1;
    n1  .INVR  qn  (STR='RR');
  END;
```

The STR='RR' parameter in the second .INVR invocation defines the output strength of that inverter as resistive. The default value used for the first invocation is active.

4.2 Simulating the Latch

The operation of the latch can be tested by exercising it with the NOVA simulator. The LATCH module was run in NOVA with the following waveforms on the g and d inputs:



The resulting simulator output is shown in Table 1, where the symbol 1 represents 1aa, 0 represents 0aa, and X represents Xaa.

4.3 The HOL Circuit Primitives

The latch structure includes three predicate definitions; a NMOS-transistor element, inverter element, and the JOIN operation. These primitive element definitions must be made in HOL before they can be used in a translation from BOLT. In HOL, time is represented as a stream of natural numbers (num), the signal values are defined to be of type STATE, and circuit signals are defined to be functions of type (num → STATE).

A simplified transistor model is used defining that the signal at the source is equal to the signal at the drain if the gate is a one, else it is Nil.

	q			q
	g d n			g d n
	* * *			* * *
00001>0	0 X		00011>0	1 0
00002>0	0 X		00012>0	0 0
00003>0	1 X		00013>1	0 1
00004>0	0 X		00014>1	1 0
00005>1	0 1		00015>1	0 1
00006>1	1 0		00016>1	0 1
00007>1	0 1		00017>0	0 1
00008>1	1 0		00018>0	1 1
00009>0	1 0		00019>0	0 1
00010>0	0 0		00020>0	0 1

Table 1: Latch Simulation Data

```
⊢def NTRAN (s,g,d) =
    (∀ t.
      s t = (((g t =laa)∨(g t =lar)∨
              (g t =lrr)∨(g t =laf)∨
              (g t =lrf)∨(g t =lff)) → d t |
                                       Nil))
```

The inverter predicate definition has five arguments. The first three arguments are of type STATE and define the possible inverter output values (i.e. the ouput strength). The first is the output STATE for a true state, the second for a false output, and the third the unknown state. The unknown output value is derived from the strongest 1 and 0 strengths. The fourth and fifth arguments are signal functions of type (num → STATE). The fourth is the inverter output and the fifth is the input.

```
⊢def INVR ls Os Xs (out,in) =
    (∀ t.
      out t = (((in t =laa)∨(in t =lar)∨
                (in t =lrr)∨(in t =laf)∨
                (in t =lrf)∨(in t =lff)) → Os |
               (((in t =Oaa)∨(in t =Oar)∨
                (in t =Orr)∨(in t =Oaf)∨
                (in t =Orf)∨(in t =Off)) → ls |
                                           Xs)))
```

4.4 JOIN

The JOIN predicate performs two tasks. It determines the resulting signal value of resolving the combination of circuit outputs by applying the join function from the STATE theory. The second task is related to the sequential behavior of a charge storage node. The capacitance of a node may result in a time delay when the node is driven to a new signal level. The

delay increases as the capacitance increases or as the strength of the driving signal decreases. This sequential behavior is modeled as having a variable delay, whose length is based on the strength of the `join` function result. [5, 9].

The `JOIN` used in the latch is modeled as having two possible delays. When the pass-transistor is turned on, the storage node at the join is driven by an active strength and the delay is defined to be zero. When the pass-transistor is turned off, the storage node is driven by the resistive strength of the feed-back inverter and the delay is defined to be one.

```
⊢def JOIN (s,s',s'') =
    (∀ t. let sig = join (s' t) (s'' t) in
          (((sig = 0aa) ∨
            (sig = 1aa) ∨
            (sig = Xaa) ∨
            (sig = Xar) ∨
            (sig = Xra)) → (s t = sig)     |
                           (s (t+1) = sig)))
```

4.5 The Translation of the Structural Specification

The HOL structural specification is obtained by translating the BOLT description. The translator may be invoked to operate on a file containing the BOLT description or on BOLT text included between the keywords `BEGIN_BOLT` and `END_BOLT` within the HOL operating environment. The result of translating the cell description is:

```
BEGIN_BOLT
  MODULE qn .LATCH g d;
    BEGIN
      n1  .NTRAN g d;
      qn  .INVR  n1;
      n1  .INVR  qn  (STR='RR');
    END;
END_BOLT
```

```
⊢def LATCH (qn,g,d) =
    (∃ n1 n1' n1''.
       NTRAN (n1',g,d) ∧
       INVR 1aa 0aa Xaa (qn,n1) ∧
       INVR 1rr 0rr Xrr (n1'',qn) ∧
       JOIN (n1,n1',n1''))
```

The data structure built by the parser and construction functions is:

```
[[['LATCH'];
  ['qn'];
  ['g'; 'd'];
  ['n1'; 'qn'; 'n1''; 'n1'''];
  ['g'; 'd'; 'n1'; 'qn']];
 [['NTRAN']; ['n1'']; ['g'; 'd']; []];
 [['INVR']; ['qn']; ['n1']; []];
 [['INVR']; ['n1''']; ['qn']; ['R'; 'R']];
 [['JOIN']; ['n1']; ['n1''; 'n1''']; []];
 [[]; []; []; []]]
: string list list list
```

4.6 The Behavioral Description

When the gate of the pass-transistor is true the latch is enabled and the output, qn, follows as the inverse of d. When the gate is false the latch stores the previous data. It is desirable to simplify the description as much as possible at each level. At the behavioral level the operation no longer depends on a device's output charge sourcing ability so this specification is written in terms of boolean signal values, not the more complex STATE data type. The HOL behavioral description is:

```
⊢def LATCH_SPEC (qn,g,d) =
    (∀ t.
      (g t → (qn t = ¬d t)    |
             (qn (t+1) = qn t)))
```

4.7 The Latch Verification

The proper operation of the latch requires that the output of the pass-transistor dominate the resistive strength output of INV2. The pass-transistor is not an amplifier so there is a validity condition that the signal applied to input d must be stronger than resistive.

```
⊢def Is_bool_active (d) =
    (∀ t. (d t = 1aa ) ∨ (d t = 0aa))
```

Because the behavior of the latch is defined only for boolean value signals at the gate, there is a validity condition for the gate that it be either a 1 or 0 state. This condition yields a 12 way case analysis in the proof that is easily reduced to considering only the two cases of enabled and latching.

```
⊢def Is_bool (g) =
    (∀ t.
      (g t = 1aa) ∨ (g t = 1ar) ∨
      (g t = 1rr) ∨ (g t = 1af) ∨
      (g t = 1rf) ∨ (g t = 1ff) ∨
      (g t = 0aa) ∨ (g t = 0ar) ∨
      (g t = 0rr) ∨ (g t = 0af) ∨
      (g t = 0rf) ∨ (g t = 0ff))
```

The verification of the latch entails proving that the latch structural description and validity conditions logically imply the behavioral specification. Because the latch behavorial specification is defined in terms of boolean values the signal functions must be composed with a STATE abstraction function from the STATE theory[6]. The theorem proven is:

```
⊢ ((Is_bool_active (d) ∧
    Is_bool (g) ∧
     LATCH (qn,g,d))   ⇒
     LATCH_SPEC (STATES_ABS o qn,
                 STATES_ABS o g,
                 STATES_ABS o d))
```

5 Future Work

The BOLT to HOL translator presented in this paper represent an important step in integrating formal verification with CAD tool environments. Future steps include:

1. Expanding and validating the library of HOL definitions corresponding to the primitive components in the NOVA library.

2. Developing an abstract syntax and denotational semantics for the circuit structure level of HDLs.

3. Using the abstract syntax and denotational semantics, developing a translator generator that will, given a grammar representing the concrete syntax of a HDL, automatically create a translation program for that HDL.

4. Integrating the circuit structure level translation work with the results of other ongoing research aimed at HDL behavioral model levels to create a complete link between HDL's and verification logics.

6 Conclusion

The goal of our work is to improve CAD functional fault exclusion techniques for VLSI design by making the use of formal circuit verification at the transistor and gate level tractable. In this paper we have described and demonstrated a translator for moving circuit structure

descriptions from the realm of the CAD design tool to formal verification. This is an important step facilitating the development of correct designs as VLSI circuits become increasingly complex.

7 Acknowledgements

This research was supported in part by NASA under Space Engineering Research Grant NAGW-1406 and by the NSF under Research Initiation Grant MIP-9109618.

References

[1] AMI: A Subsidiary of Gould Inc. *BOLT Users Manual.*

[2] R. Boulton, M. Gordon, J. Herbert, and J. van Tassel. "The HOL Verification of ELLA Designs". In *1991 International Workshop on Formal Verification in VLSI Design*, Miami, January 1991.

[3] K. B. Cameron and J. C. Shovic. "Calculating Minimum Logic State Requirements for Multi-Strength Multi-Value MOS Logic Simulators". In *1987 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 672–675, Rye Brook, New York, October 1987. IEEE Computer Society Press.

[4] A. Camilleri, M. Gordon, and T. Melham. "Hardware Verification Using Higher Order Logic". In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. Elsevier Scientific Publishers (North-Holland), 1987. Also Technical Report No. 91, University of Cambridge Computer Laboratory, September, 1986.

[5] I. S. Dhingra. "Formal Validation of An Integrated Circuit Design Style". In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 293–321. Kluwer Academic Publishers, Boston, 1988. Also Technical Report No. 115, University of Cambridge Computer Laboratory, August, 1987.

[6] J. W. Gambles and P. J. Windley. "A Verification Logic Representation of Indeterministic Signal States". In *Third NASA Symposium on VLSI Design*, pages 10.2.1–10.2.12, Moscow, Idaho, October 1991. NASA Space Engineering Research Center, University of Idaho.

[7] M. J. C. Gordon. "Why Higher Order Logic is a Good Formalism for Specifying and Verifying Hardware". In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. Elsevier Scientific Publishers (North-Holland), 1986. Also Technical Report No. 77, University of Cambridge Computer Laboratory, 1985.

[8] M. J. C. Gordon. "HOL: A Proof Generating System for Higher-Order Logic". In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and*

Synthesis, pages 73–128. Kluwer Academic Publishers, Boston, 1988. Also Technical Report No. 103, University of Cambridge Computer Laboratory, August, 1987.

[9] J. P. Hayes. "A Unified Switching Theory with Applications to VLSI Design". *Proceedings of the IEEE*, Vol. 70(No. 10):1140–1151, October 1982.

[10] T. F. Melham. "Abstraction Mechanisms for Hardware Verification". In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291. Kluwer Academic Publishers, Boston, 1988. Also Technical Report No. 106, University of Cambridge Computer Laboratory, May, 1987.

[11] T. F. Melham. "Using Recursive Types to Reason About Hardware Verification". In G. Milne, editor, *Design for Behavioural Verification*, Glosgow, July 1988. IFIP WG 10.2. Also Technical Report No. 135, University of Cambridge Computer Laboratory, May, 1988.

[12] B. Meyer. *Introduction To The Theory Of Programming Languages.* Prentice Hall International, 1990.

[13] J. P. van Tassel. *The HOL Parser Library.* University of Cambridge Computer Laboratory, July 1991.

[14] J. P. van Tassel and D. Hemmendinger. "Toward Formal Verification of VHDL Specifications". In L. Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 261–270, Houthalen, Belgium, November 1989. IMEC-IFIP WG 10.2/WG 10.5, Elsevier Scientific Publishers (North-Holland).

[15] G. Winskel. "A Compositional Model of MOS Circuits". In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 323–347. Kluwer Academic Publishers, Boston, 1987. Also Technical Report No. 105, University of Cambridge Computer Laboratory, 1987.