A Production Based System for Formal Verification of Digital Signal Processing Architectures

Khaled M. Elleithy & Mostafa A. Aref

King Fahd University of Petroleum and Minerals
Dhahran 31262, Saudi Arabia

Abstract

In this paper a new formal hardware verification approach for Digital Signal Processing Architectures based on a production system environment is introduced. The PROPER system (PROduction system for hardware VERification) is implemented using CLIPS (C Language Integrated Production System). A cell library of different hardware components has been implemented. Components in the cell library are described at the transistor level, circuit level, gate level, logical level, and functional level. An example of Carry Select Adder using PROVER is given in the paper.

1: Introduction

The design process is a transformation between different specifications (Figure 1).

![Diagram](image)

Figure 1. A Hierarchical Representation of Design and Verification.

An input algorithm may be specified using a specific algorithmic specification language. An architecture may be specified using a realization specification language. The role of Design can be viewed as a transformation process between the algorithm specification language and the realization specification language.

The objective of any design procedure is to produce an architecture that correctly implements the required behavior subject to a given set of constraints on area and timing. It is very expensive to fabricate a design before verifying the functional correctness of the design. There are two approaches for verification; simulation and formal verification. Simulation is efficient with small size Architectures where it is possible to exhaustively run the simulator. Formal verification is suitable for large size architectures.

A verification methodology is formal if it satisfies the following characteristics[1]:

- There is a formal framework to describe the architecture
- There is a formal technique to prove the equivalency of the implementation and the specification without physically construct or simulate the design.
- It is possible to manipulate and study the design's performance without the physical implementation.

The heart of any formal verification methodology, then, is the availability of a formal specification language where formal proofs can be driven. Logic is one of the widely used specification languages for verification. First order logic has been used in a number of systems[2-4]. Higher order logic has been used in a number of applications[5-8]. Joyce[7-8] used the HOL system to verify a microprocessor. Temporal logic is an appropriate approach for specifying timing characteristics of a design. Temporal logic has been successfully used for verification in[9-10].

Automated Theorem Provers are efficient in proving the correctness in large scale architectures where the proof of correctness is done automatically. In this paper we are
introducing a novel approach for formal verification based on a production system. The PROVER (PROduction system for hardware VERification) is implemented using the CLIPS[11] (C Language Integrated Production System). The paper is organized as follows: in section two an introduction to the CLIPS environment is given, in section 3 the PROVER system is introduced, a DSP example using the PROVER for verification is given in section 4. Finally, section 5 offers conclusions and future extensions.

2: CLIPS Environment

CLIPS was developed in 1985 by NASA at the Lyndon B. Johnson Space Center [11]. It is designed to overcome a number of difficulties in using lisp based tools. CLIPS is written in C to support the goal of high portability, extendibility, capabilities, low-cost and ease of integration with external systems [12]. CLIPS version 5.1, been used in this work, does provide object oriented features, including defining classes, multiple inheritance and message handlers.

CLIPS uses rules as its primary knowledge representation approach with a lisp-like syntax. CLIPS supports a rich pattern-matching language for specifying rule conditions. CLIPS, as a production system [13], provides pattern-directed control of a problem-solving process. CLIPS consists of knowledge base, fact list, agenda, and cycle of execution. Knowledge base contains a set of production rules. Fact list (working memory) contains a description of the current state of the problem.

Cycle of execution is the control structure of CLIPS. Once a knowledge base is built and the fact list is prepared, CLIPS is ready to execute rules. The basic cycle of execution of CLIPS is as follows:
1. The knowledge base is examined to see if the conditions of any rule have been met.
2. All rules whose conditions currently are met are activated and placed on the agenda.
3. The top rule on the agenda is selected, and its actions are executed.

As a result of actions execution, new rules can be activated or deactivated. This cycle is repeated until all rules that can be fired have done so or the rule limit is reached. The number of rule firings allowed in a cycle may be set by the programmer.

3: PROVER System

PROVER is a production system for formal hardware verification. PROVER gets its inputs as a circuit description and a behavioral description of that circuit to be verified. The circuit description would be one or a combination of different hardware descriptions. These descriptions include transistors, gates, logical, functional, and module descriptions. PROVER has a knowledge base consists of a Cell Library and Rules. Cell library contains a predefined set of hardware components. It consists of five sub libraries represent the five level of hardware descriptions. These sub libraries are Transistor-level Library (TL), Gate-level Library (GL), Logic-level Library (LL), Function-level Library (FL), Module-level Library (ML). The block diagram of PROVER is shown in Figure 2. The rules define possible transformation from one level to another. Also, they reflect the semantic of each level description.

![Figure 2. An overview of the PROVER Environment](image)

The incremental approach is used in developing PROVER. In this approach, a subset of the domain is considered first and a prototype is built. Then this prototype is expanded to the other subsets of the domain. Currently, the knowledge base of PROVER has transistor, gate, logical, and functional level sub libraries and rules handling these levels.

4: Case Study

The performance of DSP processors depends heavily on the speed of adders used in the core. Carry select adder is a fast asynchronous adder based on a carry acceleration approach. In this section a two stage carry select adder shown in Figure 3 is verified using PROVER. The methodology can be applied to verify an n-bit adder.

4.1: Input

The PROVER's input consists of the modules functional description and interconnections of modules represented using CLIPS. This input is as follows:
Figure 3. An 8-bit Carry Select Adder.

*Inputs*(A0-7, B0-7, C-1)  
*Outputs*(Sum0-7, Cout)

**Section Adder Rule**

If there is a section adder of size \( n \)  
then delete the section adder description and express the \( n \) outputs in terms of the inputs.

This rule is fired 4 times generating 16 logical expressions which includes S0-7 assuming the input carry is 0 and S0-7-1 assuming the input carry is 1. For example,

- \( S0-0 = A0 + B0 + 0 \), and \( S0-1 = A0 + B0 + 1 \).

**Multiplexer Rule**

If a 2-input multiplexer is used  
then express the output as two implications.

This rule is fired 8 times converting the 16 logical expression into 16 implications. For example,

- \( \neg C-1 \Rightarrow A0 + B0 + 0 \), and \( C-1 \Rightarrow A0 + B0 + 1 \).

**Reduction Rules**

If two implications are in the form:  
E => y = f(x1,x2,...,1,) and \( \neg E \Rightarrow y = f(x1,x2,...,0,) \)
then remove these two implications and add the fact $y = f(x_1, x_2, \ldots, E_i)$.

This rule is fired 8 times which reduces the 16 implications into eight equations for the eight outputs of the adder.

If there is And or Or gate then express the output of terms of the inputs.

This rule is fired several times and two expression for $C_3$ and Cout are formed.

If an output expression contains internal terms and the internal terms are expressed to the input terms then replace these internal terms by the expression of the input terms.

This rule is fired several times to include the expression of $C_3$ in the Sum expressions as follows.

- $S_0 = \text{sum}(A_0, B_0, C_{-1})$
- $S_1 = \text{sum}(A_1, B_1, \text{carry}(A_0, B_0, C_{-1}))$
- $S_2 = \text{sum}(A_2, B_2, \text{carry}(A_1, B_1, \text{carry}(A_0, B_0, C_{-1})))$
- $S_3 = \text{sum}(A_3, B_3, \text{carry}(A_2, B_2, \text{carry}(A_1, B_1, \text{carry}(A_0, B_0, C_{-1}))))$
- $S_4 = \text{sum}(A_4, B_4, \text{carry}(A_3, B_3, \text{carry}(A_2, B_2, \text{carry}(A_1, B_1, \text{carry}(A_0, B_0, C_{-1}))))$
- $S_5 = \text{sum}(A_5, B_5, \text{carry}(A_4, B_4, \text{carry}(A_3, B_3, \text{carry}(A_2, B_2, \text{carry}(A_1, B_1, \text{carry}(A_0, B_0, C_{-1}))))$
- $S_6 = \text{sum}(A_6, B_6, \text{carry}(A_5, B_5, \text{carry}(A_4, B_4, \text{carry}(A_3, B_3, \text{carry}(A_2, B_2, \text{carry}(A_1, B_1, \text{carry}(A_0, B_0, C_{-1}))))))$
- $S_7 = \text{sum}(A_7, B_7, \text{carry}(A_6, B_6, \text{carry}(A_5, B_5, \text{carry}(A_4, B_4, \text{carry}(A_3, B_3, \text{carry}(A_2, B_2, \text{carry}(A_1, B_1, \text{carry}(A_0, B_0, C_{-1}))))))$

To summarize the proof strategy, the interconnections between modules are checked for correctness as well as the modules. Section Adders (SA) are assumed to be functionally correct. Therefore, sum and carry functions are used directly for substitution. A proof for the multiplexers functionality is carried out. Carry selector blocks are functionally proved from the logical level. The final output is to verify that the circuit is an implementation of the function value.

4.4: Analysis

The previous proof can be easily extended to n-bit carry select adder through the general definitions of the used modules. Table 1 shows PROVER results for verifying 8, 16, and 32 bits carry select adders.

<table>
<thead>
<tr>
<th>N</th>
<th>Execution Time (sec)</th>
<th>Rules Fired</th>
<th>Mean # of Facts</th>
<th>Mean # of Activation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.76923</td>
<td>40</td>
<td>33</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>1.75841</td>
<td>76</td>
<td>45</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>4.560439</td>
<td>148</td>
<td>68</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 1. Results for 8, 16, and 32 bit Carry select adders

5: Conclusions

Verifying large scale systems is no more a straightforward process that can be completely achieved using traditional approaches of simulation. In this paper a formal verification approach based on the CLIPS production system as a host environment has been introduced. An example of Carry select adder has been PROVER for 8, 16 and 32 bits. Results show the a 32 bit Carry Select Adder can be verified functionally in less than five seconds. Currently we are investigating the applicability of PROVER to large size problems.

Acknowledgment

The authors wish to acknowledge King Fahd University of Petroleum and Minerals for utilizing the various facilities in preparation and presentation of this paper.

References


