Formal Verification of a Basic Circuits Library

Christoph Berg, Christian Jacobi, Daniel Kroening*
Saarland University, Computer Science Department
66123 Saarbrücken, Germany
{cb,cj,kroening}@wjpserver.cs.uni-sb.de
Tel +49-681-302-4490, Fax -4290

Keywords: Architectures, Basic Circuits, Formal Verification, Theorem Proving

December 20, 2000

Abstract

We describe the results and status of a project aiming to provide a provably correct library of basic circuits. We use the theorem proving system PVS in order to prove circuits such as incrementers, adders, arithmetic units, multipliers, leading zero counters, shifters, and decoders. All specifications and proofs are available on the web.

1 Introduction

In large hardware design projects, standard hardware components like adders and shifters are frequently used. The correctness of the whole design depends on the correctness of these standard components, and the correctness of the glue between them.

At the University of Saarbrücken, we are currently working on the verification of a pipelined, out-of-order RISC processor based on the DLX architecture [1], featuring a dual precision floating point unit (FPU) [2]. In this paper, we describe the formal verification of standard hardware components that are used in this project. The verified components are adders, arithmetic units, multipliers, shifters, decoders, half decoders, leading zero counters, and parallel prefix computations. The designs and the proof ideas are taken from [3]. The designs are of arbitrary bit-width, i.e., the user of the library can instantiate the circuits to any number of bits. The verification is done on gate level.

We use the theorem proving system PVS [4, 5] as verification tool. The use of theorem proving is motivated as follows: in order to prove the correctness of a system as large as a complete microprocessor, we need a formal, mathematical specification of functional components. From these, one successively derives larger circuits that are correct with respect to a formal specification. In the end, the correctness of the complete microprocessor is achieved.

It is impractical to use equivalence checking, since this yields no mathematical specification of the component (which is needed when using the module in a larger design) – unless the reference design had such a formally verified mathematical specification. Unfortunately, we were unable to locate such specifications in published literature. In this sense, we offer formally verified reference designs for the most commonly used hardware components, which can serve to other researchers in the field of hardware verification.

Another drawback of equivalence checking in comparison with theorem proving is that one cannot verify components with variable size. However, in contrast to equivalence checking, theorem proving usually requires a considerable amount of user interaction. Of course, this was also the case in our work.

Project status The verification of the most common hardware components as listed above is completed, new components are added as required by the DLX project. The verification of the out-of-order integer core of the processor employing a Tomasulo scheduler [6], as well as the verification of the FPU is nearly finished. In order to obtain synthesizable hardware, we aim towards a tool that will translate our hardware specifications from the PVS language into Verilog HDL.

Related work There is a vast amount of literature on the formal verification of hardware. The verification of a simple adder and an ALU using PVS is reported in [7]. In our verification project, we use the lemmas in the PVS bitvector library [8], which includes a carry chain adder. The

*supported by the DFG graduate program 'Leistungsgarantien für Rechnersysteme'

component	input width	cost	delay
multiplexer	n	n	1
or tree	n	n	$\log n$
zero tester	n	n	$\log n$
equality tester	n	n	$\log n$
halfadder	1	1	1
fulladder	1	1	1
carry chain incrementer	n	n	n
incrementer (generic)	n	n	n
absolute value	n	n	n
carry chain adder	n	n	n
carry save adder	n	n	1
compound adder	n	n	$\log n$
generic parallel prefix	$n = 2^m$	n	$\log n$
carry lookahead adder	$n = 2^m$	n	$\log n$
adder (generic)	n	n	n
arithmetic unit	n	n	n
wallace tree	$n = 2^m$	n^2	$\log n$
WT multiplier	$n = 2^m$	n^2	$\log n$
decoder	n	2^n	$\log n$
half decoder	n	2^n	n
leading zero counter	$n = 2^m$	n	$\log n$
barrel shifter	$n = 2^m$	$n \log n$	$\log n$
logic left shifter	n	$n \log n$	$\log n$
logic right shifter	n	$n \log n$	$\log n$

Table 1: The components contained in the library in increasing complexity

verification of an adder using various verification systems is described in [9]. In [10], Bryant verifies fixed size arithmetic circuits against a mathematical specification.

Given a reference design and assuming its correctness, it is state-of-the-art to automatically verify equivalence with a new design. There are several approaches to this, e.g., boolean equivalence checkers using BDDs or variations [11–13]. In [14], Clarke et.al. use function abstraction and BDDs for equivalence checking. In [15], Stanion proves the equivalence of two fixed bit width multipliers.

2 The Library

Our circuit library consists of various components as listed in table 1. Some circuits are limited to sizes that are powers of two. Cost and delay are given as asymptotic measures, i.e. n means $O(n)$.

Trivial constructions are multiplexer, or tree, zero tester and equality tester. Incrementers and various adders are built from simple half and fulladders. The circuits marked "(generic)" above are wrappers to hide the actual adder/incrementer implementation. By changing the wrapper's implementation the user can choose among the different adder implementations. The generic parallel prefix can be instantiated by any associative function; we use it for the carry lookahead adder.

The 'wallace tree' is a wallace tree multiplier without the final adder stage; making it available separately allows for inserting a pipeline stage in a processor design. Decoder and half decoder convert binary numbers to unary notation. A barrel shifter shifts its input cyclically to the left; the logical shifter variants pad with zeros.

All circuits come with a correctness statement and the corresponding PVS proof. Circuit specifications, implementations and proofs are available at our web site[1] and may be used by other researchers in their projects.

As we aim to build hardware, all specifications use a 'synthesizable' subset of the PVS language. The circuit library described in this paper provides the basic building blocks for our DLX processor. We currently develop a translation tool that will convert our PVS sources to Verilog HDL. When this tool is finished, we will have a running processor that is fully verified on the gate level. The Verilog files will also be available on the web.

3 Circuit Verification

Exemplarily, we demonstrate the correctness proof for the leading zero counter in this section. A leading zero counter is a circuit that outputs the number of successive zero bits at the left (most significant) side of its input; it is used by floating point rounding units for instance. The formalization of other circuits and their correctness criteria are stated in a similar way, the methods used in the leading zero counter example apply for the other circuits as well. Except for one example, the definitions and theorems are not given in the PVS language in this paper but in the mathematical notation for readability. These notations can be easily translated to PVS (or other theorem provers, as we believe).

3.1 Formal Specification of Functionality

Notations used A bitvector b of length n is indexed by $b[i]$ with i ranging from $n-1$ to 0. For $i \geq j$, $b[i,j]$ denotes the bit vector consisting of bits i to j of b; $\circ$ is the concatenation operator. $\mathbf{x}^i$ with $\mathbf{x} \in \{\mathbf{0}, \mathbf{1}\}$ denotes the bit $\mathbf{x}$ repeated i times. The natural number represented by b is denoted by $\langle b \rangle := \sum_{i=0}^{n} 2^i \cdot b[i]$.

[1] http://www-wjp.cs.uni-sb.de/projects/verification/

Formal definition In order to prove a leading zero counter implementation correct, we need a formal notion of 'leading zeros'. We define a function $lzero$ on bitvectors of length n:

$$lzero(b) = \max\{i \in \mathbb{N} \mid i = 0 \vee (i \leq n \wedge b[n-1, n-i] = \mathbf{0}^i)\}$$

Since PVS provides a finite sets library featuring the maximum function, the $lzero$ function is easily defined in the PVS language:

```
lzero(b):nat = max({ i:nat | i=0 OR
(i<=n AND b^(n-1, n-i) = fill[i](FALSE)) })
```

We start with some lemmas on the $lzero$ function. All these lemmas are fairly obvious, but their proofs are technically complicated in PVS. We omit the proofs due to lack of space.

Lemma 1 *$lzero$ essentially depends on the position of the first $\mathbf{1}$-bit:*[2]

1. $lzero(b) = 0 \Leftrightarrow b = \mathbf{1} \circ b[n-2, 0]$
2. *For all* $1 \leq i \leq n-2$:
$$lzero(b) = i \Leftrightarrow b = \mathbf{0}^i \circ \mathbf{1} \circ b[n-i-2, 0]$$
3. $lzero(b) = n-1 \Leftrightarrow b = \mathbf{0}^{n-1} \circ \mathbf{1}$
4. $lzero(b) = n \Leftrightarrow b = \mathbf{0}^n$

Lemma 2 *$lzero$ is bounded by n:*

$$lzero(b) \leq n$$

Lemma 3 *Leading zero concatenation: For all $l \in \mathbb{N}, l \geq 1$, it holds*

$$lzero(\mathbf{0}^l \circ b) = l + lzero(b)$$

Lemma 4 *An inverter can be used to increment a bitvector's value:*

$$\langle b[n-1] \circ \overline{b[n-1]} \circ b[n-2, 0] \rangle = 2^{n-1} + \langle b \rangle$$

3.2 Circuit Implementation

Our circuit is defined recursively on bitvectors of length $n = 2^m$ as depicted in figure 1. Of course, one cannot

[2]The case split is necessary to avoid bitvectors of zero length.

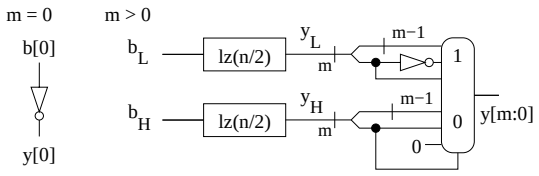

Figure 1: leading zero counter

prove the correctness of a picture. We therefore have to formalize the implementation by means of a function lz_impl. For $m = 0$, we simply set

$$lz\_impl(b) := \overline{b[0]}$$

For $m > 0$, let $y_H = lz\_impl(b_H)$ and $y_L = lz\_impl(b_L)$, where $b_H = b[2^m - 1, 2^{m-1}]$ and $b_L = b[2^{m-1} - 1, 0]$. We set

$$lz\_impl(b) := \begin{cases} y_L[m-1] \circ \overline{y_L[m-1]} \circ y_L[m-2, 0] \\ \qquad\qquad \text{if } y_H[m-1] \\ \mathbf{0} \circ y_H \qquad \text{otherwise} \end{cases}$$

This definition is easily translated to a hardware specification language such as Verilog HDL.

3.3 Proof

The implementation is correct if our implementation counts the number of leading zeros for all inputs:

Theorem 1 *For all $m \in \mathbb{N}, b \in \{\mathbf{0}, \mathbf{1}\}^{2^m}$:*

$$\langle lz\_impl(b) \rangle = lzero(b)$$

Proof As the recursive construction suggests, the proof is by induction on m. The induction base $m = 0, n = 2^0$ is easily proven using lemmas 1.4 and 2. The induction step first does a case split on bit $y_H[m-1]$. When set, it follows that

$$2^{m-1} \leq \langle lz\_impl(b_H) \rangle.$$

The induction hypothesis and lemma 2 yield

$$2^{m-1} \leq lzero(b_H) \leq 2^{m-1},$$

which implies equality. Lemma 1.4 leads to

$$b_H = \mathbf{0}^{2^{m-1}}.$$

By lemma 3, and the induction hypothesis we have

$$\begin{aligned} lzero(b) &= lzero(\mathbf{0}^{2^{m-1}} \circ b_L) \\ &= 2^{m-1} + lzero(b_L) \\ &= 2^{m-1} + \langle lz\_impl(b_L) \rangle. \end{aligned}$$

With lemma 4, the output y of the multiplexer satisfies

$$lzero(b) = 2^{m-1} + \langle lz\_impl(b_L)\rangle = \langle y\rangle.$$

The other case is handled analogous.

4 Conclusion

We implemented a library of basic circuits required for microprocessor design and verified the gate level correctness with respect to a mathematical specification using a theorem proving system. The circuits are designed to be reusable and are available at our web site. We therefore encourage designers to incorporate our library in larger projects, as we do in the DLX verification project.

The verification of such circuits using theorem proving systems involved more manual work than required by using equivalence checking instead. However, we think that this extra effort pays off since we can provide generic circuits and proceed using a mathematical specification. Thus, we reach a high level of abstraction hiding the gate level. This abstraction level enables us to prove the correctness of successively larger circuits, up to complete floating point units and processors.

References

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[2] Christian Jacobi and Daniel Kroening. Proving the correctness of a complete microprocessor. In *GI Jahrestagung 2000*. Springer, 2000.

[3] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.

[4] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

[5] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, Baco Raton, Florida, 1995.

[6] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11 (1), pages 25–33. IBM, 1967.

[7] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In *2nd International Conference on Theorem Provers in Circuit Design*, volume 901 of *LNCS*, pages 203–222. Springer, 1994.

[8] Ricky Butler, Paul Miner, Mandayam Srivas, and Dave Greve. A bitvectors library for PVS. Technical Report TM-110274, NASA Langley Research Center, 1996.

[9] V. Stavridou, H. Barringer, and D.A. Edwards. Formal specification and verification of hardware: A comparative case study. In *Proceedings of the 25th ACM/IEEE conference on Design Automation*, pages 197–204, 1988.

[10] Y. Chen and R. Bryant. ACV: An arithmetic circuit verifier. In *In Proc. of IEEE ICCD '96*, pages 361–365. IEEE, 1996.

[11] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[12] R. E. Bryant and Y.-A. Chen. Verification or Arithmetic Circuits with Binary Moment Diagrams. In *32nd ACM/IEEE Design Automation Conference*, Pittsburgh, June 1995. Carnegie Mellon University.

[13] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In *ICCAD*, pages 159–163, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[14] Somesh Jha, Yuan Lu, Marius Minea, and Edmund M. Clarke. Equivalence checking using abstract BDDs. In *Proc. of IEEE ICCD '98*, pages 332–337. IEEE, 1997.

[15] Ted Stanion. Implicit verification of structurally dissimilar arithmetic circuits. In *Proc. of IEEE ICCD '99*, pages 46–50. IEEE, 1999.