Modelling Generic Hardware Structures by Abstract Datatypes

Klaus Schneider, Ramayya Kumar and Thomas Kropf

Institute of Computer Design and Fault Tolerance, University of Karlsruhe
(Prof. Dr. D. Schmid) P.O. Box 6980, W-7500 Karlsruhe, Germany,
e-mail:{schneider, kumar, kropf}@ira.uka.de

Abstract
Although formal hardware specifications using natural numbers result in clear and succinct descriptions, the proofs using them are not easy to automate. In this paper, an alternative means of specifying simple cascadable hardware using predefined operators is presented. These operators give the user a feeling that natural numbers are being used for specification and at the same time lend themselves to easy automation. Such operators work on $n$-bit bitvectors and their validation theorems, i.e. their correspondence to operations on natural numbers within the $n$-bit range, are also given. An example circuit is used to illustrate the use of these operators for specification and verification.

Keywords: Hardware Verification; Bit-vector theory; Arithmetic and Logic Circuits

1 Introduction
In digital circuit implementations natural numbers are always represented by bitvectors. If one wants to prove the correctness of such a circuit relative to a specification which uses the peano-axiomatized natural numbers, the disparity in the representations makes the automation of such proofs difficult. In this paper we present an abstract datatype for natural numbers, represented as bitvectors, which lends itself to easy automation. The advantage of this form of representation is that it is decidable and can therefore be automated. Furthermore, the semantical clarity of the operators are also retained.

In our approach we use bitvectors with a variable length $n$ as opposed to the ‘wordn’ [HOL91] library implementations, which use an arbitrary but instantiated length of the bitvector. This allows us to define single $n$-bit operators for each function instead of defining operators for each specific bitvector. The ‘wordn’ library is more suited for interactive proofs where a set of possibly automatically derived theorems are used. On the other hand using the abstract data types as defined in this paper, most of the process can
be automated from within MEPHISTO, a specialized hardware verification environment
embedded in HOL [ScKK91b], [KuKS91b]. MEPHISTO provides a set of tactics for semi-
automatically breaking up the proof goals into smaller subgoals and is augmented with
a first-order prover called FAUST [ScKK91a], [KuKS91a], [ScKK92a], [ScKK92b], which
is invoked to automatically prove the subgoals. In the context of specifications using
the abstract data types, the proofs are performed using induction over \( n \) and the subgoals are
automatically solved within MEPHISTO. When the recursive implementation has been
proven correct with respect to a specification, the circuit implementation with a specific
bitwidth is also generated automatically.

This paper is organized as follows: the next section deals with the implementation and
validation of the abstract data types for natural numbers. Section 3 illustrates the use of
the pre-defined operators and the tactics within MEPHISTO for specification, verification
and generation of specific implementations. The paper is concluded with a summary and
comments about our future work.

## 2 The Abstract Datatype

### 2.1 Representing Natural Numbers using Bitvectors

Natural numbers are represented by bitvectors which are in turn represented by pairs
\((n, f)\), where \( n \in \mathbb{N} \) and \( f : \mathbb{N} \to \text{bool} \). \( n \) denotes the length of the bitvector and the
function \( f \) \(^1\) determines the values of the bits, e.g. the bitvector \((b_n, \ldots, b_0)\) is represented
by a pair \((n, f)\) such that \( f(i) = b_i \) for \( i \in \{0, \ldots, n\} \). Of course there are infinitely many
pairs for each bitvector, so we cannot define any isomorphism between them. However,
the following equivalence relation can be defined:

\[
(n, f) \approx (m, g) \iff [m = n] \land \forall i. i \in \{0, \ldots, m\} \left[ f(i) = g(i) \right]
\]

In other words, two pairs \((n, g)\) and \((n, f)\) are identical if and only if the functions \( f \)
and \( g \) are equal for all numbers less than or equal to \( n \). Let \( \mathcal{BV}_n \) denote the set of all pairs
\((n, f)\), and \( \mathcal{BV}_n^\approx \) denote the set of all equivalence classes of \( \approx \). The notion of equivalence
classes can then be used in validating the operators which are defined over such bitvectors.

### 2.2 Validation of the Operators on Bitvectors

We define the following function for conversion between bitvectors and \( \mathbb{N} \)

\[
\text{VAL} \ n \ f := \sum_{i=0}^{n-1} \Omega(f(i)) \times 2^i
\]

where \( \Omega : \text{bool} \to \mathbb{N} := \begin{cases} 1 & \text{if } f(i) = T \\ 0 & \text{if } f(i) = F \end{cases} \)

\(^1\)when sequential specifications are to be described then \( f : \mathbb{N} \to \mathbb{N} \to \text{bool} \),
i.e. \( f \ t : \mathbb{N} \to \text{bool} \), where \( t \) represents time
VAL assigns each element of $BV_n^\infty$ to one element of $Z_n := \{0, \ldots, 2^n-1\}$. Suppose, we have an operator $\pi : BV_n^\infty \to BV_n^\infty$, which should represent a function $\pi_V : N \to N$, then we validate the operator by proving the following theorem:

$$\vdash \forall n. \forall f. \pi_V(VALE n \ f) = VAL \ \pi(n, f)$$

Figure 1 illustrates the homomorphism used for the validation of the unary operator $\pi$.

![Validation of ADT-Operators](image)

Figure 1: Validation of ADT-Operators

The validation theorem for a binary operator $\pi' : BV_n^\infty \times BV_n^\infty \to BV_n^\infty$ would have the following form, (given that, $\pi'_V : N \times N \to N$) —

$$\vdash \forall n. \forall f. \forall g. \pi'_V((VAL n \ f), (VAL n \ g)) = VAL \ \pi'((n, f), (n, g))$$

### 2.3 Predefined Operators

We now give a short description of the operators that we have defined within MEPHISTO and state the corresponding validation theorems. These validation theorems have been manually proved, once and for all, in HOL.

**ALLZERO** ($n$, ALLZERO) represents the zero-valued $n$-bit vector.

Validation Theorem: $VAL n \ ALLZERO = 0$

**SCARRY** $n \ f$ is true when a carry bit is generated by incrementing the pair $(n, f)$.

Validation Theorem: $SCARRY n f = ((VAL n f) + 1 = 2^{n+1})$

**SUCCE** $n f g$ is true iff $(n, g)$ is the successor of $(n, f)$ modulo $2^{n+1}$.

Validation Theorem: $SUCCE n f g = VAL n g = ((VAL n f) + 1) \ MOD 2^{n+1}$

**PCARRY** $n f g c$ is true when a carry bit is generated while adding $(n, f)$ and $(n, g)$ with the input carry bit $c$.

Validation Theorem:

$PCARRY n f g c = [(VAL n f + VAL n g + BV^2 c) \ DIV 2^{n+1} = 1]$

**PLUS** $n f g h c$ is true iff $(n, h)$ is the sum of $(n, f)$, $(n, g)$ and the carry bit $c$ modulo $2^{n+1}$.

Validation Theorem:

$a$ function from bool $\to$ N for converting a bit to 1 or 0.
\[ \text{PLUS} \, n \, f \, g \, h \, c \, = \, VAL \, n \, h \, = \, (VAL \, n \, f + VAL \, n \, g + BV \, c) \mod 2^{n+1} \]

\textbf{ADDER} \, n \, f \, g \, h \, c \text{ is true iff } (n + 1, h) \text{ is the sum of } (n, f) \text{ and } (n, g) \text{ the carry bit } c. \\
Validation \text{ Theorem:} \\
\[ \text{ADDER} \, n \, f \, g \, h \, c \, = \, (VAL(SUC \, n)h = VAL \, n \, f + VAL \, n \, g + BV \, c) \]

\textbf{LS} \, n \, f \, g \text{ is true iff } (n, f) \text{ is less than } (n, g). \\
Validation \text{ Theorem:} \, \text{LS} \, n \, f \, g \, = \, (VAL \, n \, f) \lt (VAL \, n \, g) \]

\textbf{EQQ} \, n \, f \, g \text{ is true iff } (n, f) \text{ and } (n, g) \text{ belong to the same equivalence class.} \\
Validation \text{ Theorem:} \, \text{EQQ} \, n \, f \, g \, = \, (VAL \, n \, f) = (VAL \, n \, g) \]

\textbf{COMPL} \, n \, f \, g \text{ is true iff } (n, g) \text{ is the one’s complement of } (n, f). \\
Validation \text{ Theorem:} \, \text{COMPL} \, n \, f \, g \, = \, (VAL \, n \, f + VAL \, n \, g + 1) = 2^{n+1} \]

\textbf{L\_SHIFT} \, n \, f \, g \text{ is true iff } (n, g) \text{ is the left shifted value of } (n, f). \\
Validation \text{ Theorem:} \, \text{L\_SHIFT} (SUC \, n) \, f \, g \, = \, (VAL(SUC \, n)g) = 2 \times (VAL \, n \, f) \]

\textbf{R\_SHIFT\_HELP} \, n \, f \, g \text{ is true iff } \forall i.0 \leq i < n \rightarrow g(i) = f(i + 1) \\
Validation \text{ Theorem:} \\
\[ \text{R\_SHIFT\_HELP} (SUC \, n) \, f \, g \, = \, (VAL \, n \, g) = ((VAL(SUC \, n)f) \div 2) \]

\textbf{R\_SHIFT} \, n \, f \, g \text{ is true iff } [\forall i.0 \leq i < n \rightarrow g(i) = f(i + 1)] \land g(n) = F \\
Validation \text{ Theorem:} \, \text{R\_SHIFT} \, n \, f \, g \, = \, (VAL \, n \, g) = ((VAL \, n \, f) \div 2) \]

In the next section, we shall illustrate the use of these operators via an example.

3 Using the ADT

The definition of the pre-defined operators could also be interpreted as a hardware implementation of it, because they involve functions on bitvectors. Verifying another circuit for the same function now becomes much easier, since both the specification and the implementation are formulated at the same level of abstraction. The proof itself waters down to a mere comparison of the two ‘circuits’. Specifications which are given in ‘natural’ form can be easily translated in a formula using the ADT.

For example consider the following specification for a comparator having two inputs \( I_1 \) and \( I_2 \) and two outputs \( \text{grtout} \) and \( \text{lessout} \):

\[ (\text{grtout} :\leftrightarrow I_1 > I_2) \land (\text{lessout} :\leftrightarrow I_1 < I_2) \]

Using the operators defined above and compiler construction techniques, the specification can be automatically translated into:
\[ GrtLss\_N\_SPEC\ n\ in1\ in2\ grtout\ lessout\ := \]
\[ (grtout \leftrightarrow LS\ n\ in2\ in1) \]
\[ (lessout \leftrightarrow LS\ n\ in1\ in2) \]

Starting from this specification we illustrate the interactions with MEPHISTO using HOL-sessions given in numbered boxes. The specification in HOL syntax is given below:

```
new_definition
  ('GrtLss_N_SPEC',
   "GrtLss_N_SPEC n in1 in2 grtout lessout =
    (grtout = LS n in2 in1) \ (lessout = LS n in1 in2)"
   );
```

The circuit implementing this function corresponds to a cascade of the 1-bit comparators which may correspond to the following basic circuit \((GrtLss\_1\_IMP)\):

![Diagram of 1-bit comparator circuit](image)

Figure 2: Implementation of 1-bit comparator \((GrtLss\_1\_IMP)\)

The corresponding formal description of \(GrtLss\_1\_IMP\) is automatically derived by MEPHISTO and is given as follows:

```
new_definition
  ('GrtLss_1_IMP',
   "GrtLss_1_IMP grtin lessin in1 in2 grtout lessout =
    ? l1.
    XNOR(in1, in2, l1) \ XNOR(l1, grtin, in1, grtout) \ MUX(l1, lessin, in2, lessout)"
   );
```

The 1-bit module is now used to recursively define an \(n + 1\)-bit implementation \((GrtLss\_N\_IMP)\) as shown in figure 3. This implementation, uses an \(n\)-bit blackbox and is interconnected to the 1-bit module.

Using the \texttt{prim_rec_definition} this structure can be formally coded as:
new_prim_rec_definition
('GrtLss_N_IMP',
"(GrtLss_N_IMP 0 in1 in2 grtout lessout =
    GrtLss_1_IMP F F (in1 0) (in2 0) grtout lessout) /
    (GrtLss_N_IMP (SUC n) in1 in2 grtout lessout =
     ?11 12.
     GrtLss_N_IMP n in1 in2 11 12 /
     GrtLss_1_IMP 11 12 (in1 (SUC n)) (in2 (SUC n)) grtout lessout)"");

The goal to be proven, i.e. the equivalence between the recursive structure and the specification, can now be set:

#set_goal
([],
"!in in1 in2 grtout lessout.
    GrtLss_N_IMP n in1 in2 grtout lessout =
    GrtLss_NSPEC n in1 in2 grtout lessout");;

The first step in proving this goal is to apply induction and to rewrite with the definitions of the n-bit specification and implementation and the 1-bit implementation. This yields two subgoals as shown below:
Applying the tactic SIMPLIFY_TAC within MEPHISTO for removing the internal line 11, yields a simplified goal which can then be automatically solved by the first-order, automatic prover, FAUST\(^3\).

\(^3\)The output churned out of HOL will be abbreviated to cut down the size of the paper
Having solved the base case of the induction, the step case is also solved using the same principle. An application of SIMPLIFY_TAC, removes the internal lines and the simplified goal can then be solved using PROP_FAUST_TAC.

```plaintext
#e(SIMPLIFY_TAC);;
OK.
"(grotout = ((in1(SUC n) = in2(SUC n)) => LS n in2 in1) /
  (~(in1(SUC n) = in2(SUC n)) => in1(SUC n))) /
(lessout = ((in1(SUC n) = in2(SUC n)) => LS n in1 in2) /
  (~(in1(SUC n) = in2(SUC n)) => in2(SUC n))) =
(grotout = ((in2(SUC n) = in1(SUC n)) => LS n in2 in1) /
  (~(in2(SUC n) = in1(SUC n)) => ~(in2(SUC n) \ in1(SUC n))) /
(lessout = ((in1(SUC n) = in2(SUC n)) => LS n in1 in2) /
  (~(in1(SUC n) = in2(SUC n)) => ~(in1(SUC n) \ in2(SUC n)))"
  [ "!in1 in2 grotout lessout.
    GrtLss_N_IMP n in1 in2 grotout lessout =
    GrtLss_N_SPEC n in1 in2 grotout lessout" ]

() : void
Run time: 1.5s
Intermediate theorems generated: 571
```

```plaintext
#e(POP_ASSUM (\x. PROP_FAUST_TAC));;
OK..
goal proved

..............................
|- !n in1 in2 grotout lessout.
  GrtLss_N_IMP n in1 in2 grotout lessout =
  GrtLss_N_SPEC n in1 in2 grotout lessout

Previous subproof:
goal proved
()
Run time: 3.3s
Intermediate theorems generated: 26
```
This correctness theorem is then stored for future use.

```
#save_top_thm 'GrtLss_N_CORRECT';
|- !n in1 in2 grtout lessout.
   GrtLss_N_IMP n in1 in2 grtout lessout =
   GrtLss_N_SPEC n in1 in2 grtout lessout
Run time: 0.1s
```

Having proven an \( n \)-bit comparator, we can now generate an implementation corresponding to a specific bitwidth, using the function called \texttt{WRAP\_OUT}. This function takes in the name of the generalized \( n \)-bit component and the number of the bits, and creates a new definitional constant for the specific bitwidth and also returns an appropriate correctness theorem. For example, instantiating the \( n \)-bit comparator to a 4-bit comparator, yields a circuit as shown in figure 4. The HOL session contains the theorem that has been proved using the new constant \texttt{GrtLss\_4\_IMP}, which has been introduced. It is to be noted that, the \( n \)-bit correctness theorem, the specification and the implementation are accessed by the function \texttt{WRAP\_OUT}, in order to generate the specialized implementation and the correctness theorem. We are building an interface between \textsc{mephisto} and a commercial circuit design framework \textsc{cadence}, which will then generate a netlist corresponding to the specialized implementation and introduce the module within the user’s design library.

```
#WRAP\_OUT 'GrtLss_N' 4;;
|- GrtLss\_4\_IMP in1 in2 grtout lessout =
   GrtLss\_N\_SPEC 3 in1 in2 grtout lessout
Run time: 1.8s
Intermediate theorems generated: 359
```

The use of the generalized recursive implementation followed by a specialization process, makes it possible to generate large formally verified circuits within acceptable times as shown below:

```
#WRAP\_OUT 'GrtLss_N' 64;;
|- GrtLss\_64\_IMP in1 in2 grtout lessout =
   GrtLss\_N\_SPEC 63 in1 in2 grtout lessout
Run time: 70.8s
Intermediate theorems generated: 19199

#WRAP\_OUT 'GrtLss_N' 256;;
;;;; GC
;;;; GC
;;;; GC
;;;; GC
|- GrtLss\_256\_IMP in1 in2 grtout lessout =
   GrtLss\_N\_SPEC 255 in1 in2 grtout lessout
Run time: 1471.4s
Garbage collection time: 20.4s
Intermediate theorems generated: 273023
```
4 Summary and future work

We have defined and validated several arithmetical operations on bitvectors represented by elements of $\mathcal{B} \mathcal{Y}_n^z$ such as addition, complementation, etc. The length of the bitvectors is an extra parameter for the operation, so a single operation can deal with arbitrary lengths. Such representations are cut out for use within MEPHISTO. The disadvantage of the representation is that a single bit of a vector cannot be changed, since this changes the whole function which is the second component of the representation pair. Thus, changing single bits means changing the whole pair.

Up to now we have only dealt with one dimensional digital circuits, but the concepts stated above also carry over to multi-dimensional circuits. In our future work we shall look at circuits which are realized using more complex interconnections, such as trees, arrays, etc.
References

[HOL91] Public Domain Distribution of HOL, *wordn Library in eval*


