# Parametric Circuit Representation Using Inductive Boolean Functions*

**Aarti Gupta and Allan L. Fisher**

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891.

**Abstract.** We have developed a methodology based on symbolic manipulation of inductive Boolean functions (IBFs) for formal verification of inductively-defined hardware. This methodology combines the techniques of reasoning by induction and symbolic tautology-checking in an automated and potentially efficient way. In this paper, we describe a component of this methodology that regards various mechanisms used to represent inductively-defined circuits in the form of IBFs. The focus is on general parameterization issues, such as multiple parameter functions, multiple output functions, interaction of different parameters for supporting compositions etc. These mechanisms, which may be useful in other applications involving parametric circuit descriptions, are illustrated through practical circuit examples along with preliminary results. We also describe an application of our formal verification methodology, where a proof by induction is performed by automatic symbolic manipulation of parametric circuit representations.

## 1 Introduction

The high level of complexity of current hardware systems has led to an interest in formal methods for proving their correctness. We have developed a methodology based on symbolic manipulation of inductive Boolean functions (IBFs) for formal verification of inductively-defined hardware, i.e. circuits where the structure can be described inductively (iteratively, or recursively) in terms of size parameters. Since reuse of existing designs has become an important issue in practice, use of parametric designs in the form of standard libraries has been emphasized as an emerging trend [12]. By directly addressing the issue of parameterization in our approach, as described in this paper, we provide the necessary framework for representation and verification of such designs. To provide a perspective for the research described in this paper, we start with a brief background description of our verification methodology.

### 1.1 Motivation

Previous verification work with parametric descriptions of circuits includes reasoning by induction both in theorem-proving systems [7, 13, 16, 20], and within model checking/language-containment paradigms [9, 17, 21] (an extended bibliography can be found in a recent survey [14]). The main advantage with these approaches is that a single proof serves to establish the functional or behavioral correctness of an entire family of circuits. However, most available approaches are semi-automated, typically requiring user guidance and heuristic search for a proof.

* This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

On the other hand, verification of non-parametric circuits has been successfully performed by various techniques based on symbolic manipulation of Boolean functions [1, 4, 5, 6, 8, 10, 11, 18]. The success of these techniques is largely due to the availability of automatic symbolic Boolean manipulation algorithms (using canonical Boolean function representations) that are efficient in practice [3, 4]. There have been some recent efforts in applying these techniques for iterative systems [19], but unlike our approach, they use automata-based methods.
The primary motivation for our verification methodology was to combine reasoning by induction and symbolic tautology-checking in a way that would incorporate the advantages of both. Our approach is based on automatic symbolic manipulation of classes of *inductive* Boolean functions, i.e. Boolean functions that are defined parametrically in terms of induction parameters (formal definitions are given later). Just as symbolic Boolean function manipulation is facilitated by using a canonical representation for Boolean functions — such as Bryant's Binary Decision Diagram (BDD) [4] — our manipulations use canonical representations for inductive Boolean functions. These representations, for different classes of IBFs, can be viewed as inductive extensions of BDDs, where canonicity is ensured by equality checking with a built-in principle of induction. The main idea is that by building in this principle into the IBF representations, a proof by induction is performed automatically through symbolic IBF manipulation. For verification applications, this allows us both to automate an induction proof, and to eliminate the heuristic search usually encountered in proof-theoretic frameworks. Note that we do not claim to address the general problem of induction in a first-order (or higher-order) logic setting, as is done in theorem-proving systems with formal logics [2, 7].

### 1.2 Overview

Our verification methodology based on symbolic IBF manipulation can be naturally described in terms of the following components:

- characterization of useful classes of IBFs, and a representation schema for each (consisting of a canonical representation and symbolic manipulation algorithms for functions in that class),
- representation of inductively-defined circuits and specifications in the form of IBFs, and
- translation of induction proofs into appropriate symbolic manipulations on the IBF representations.

The schemata details for the two classes of IBFs we have identified so far — linearly inductive functions (LIFs), and exponentially inductive functions (EIFs) — can be found elsewhere [15], and are summarized for convenience in Section 2. The main focus of this paper is on representation of parametric circuits in the form of LIFs and EIFs. The emphasis is on general mechanisms addressing important parameterization issues such as multiple parameters (described in Section 3), as well as multiple outputs, composition of descriptions with different parameters etc. (described in Section 4). These mechanisms are illustrated with the help of practical circuit examples (including a serial adder, parallel parity circuit, tree carry lookahead adder, decoder, register file), along with preliminary results for LIF circuit representation. The use of these mechanisms is not limited to our verification methodology, but can potentially extend to other applications involving parametric hardware descriptions. In Section 5, we describe a simple verification task where a proof by induction for a specification property, expressed as a Boolean combination of LIFs, translates to our automated method for tautology-checking of the corresponding specification formula. Finally, our conclusions and directions for future work are presented in Section 6.

## 2 Circuit Representation with IBF Schemata

### 2.1 Linearly Inductive Functions (LIFs)

Informally, an inductive Boolean function is defined in terms of an induction parameter, with the arguments of the function also parameterized accordingly. Formally,

**Definition 1**: A Linearly Inductive (Boolean) function (LIF) $f$ satisfies the following three conditions:

- Condition 1: For $i = 1$, the 1-instance of $f$ (denoted $f^1$) is a Boolean combination (denoted $\mathcal{B}^1$) of the 1-instance inputs (denoted $X^1$),
  i.e. $f^1 = \mathcal{B}^1(X^1)$.
- Condition 2: For all $i > 1$, the $i$-instance of $f$ (denoted $f^i$) is a Boolean combination (denoted $\mathcal{B}^i$) of only the $i$-instance inputs (denoted $X^i$) and some $(i-1)$-instance functions (denoted $G^{i-1}$), each of which is also an LIF,
  i.e. $f^i = \mathcal{B}^i\ (X^i, G^{i-1})$.
- Condition 3: For all $i, j > 1, \mathcal{B}^i \equiv \mathcal{B}^j$,
  i.e. $f^i$ is related to $X^i$ and $G^{i-1}$ in the same manner as $f^j$ is to $X^j$ and $G^{j-1}$, respectively.

**Example 1**: The MSB of a serial adder with data inputs $a, b$ and a carry input $c\_in$, can be represented parametrically as an LIF $sum$ as follows:
for $i = 1, sum^1 = a^1 \oplus b^1 \oplus c\_in$,
for $i > 1, sum^i = a^i \oplus b^i \oplus carry^{i-1}$, where the LIF $carry$ is defined as:
for $i = 1, carry^1 = (a^1 \wedge b^1) \vee ((a^1 \vee b^1) \wedge c\_in)$,
for $i > 1, carry^i = (a^i \wedge b^i) \vee ((a^i \vee b^i) \wedge carry^{i-1})$.

In the LIF schema, each LIF $f$ is identified with a *function descriptor*, consisting of the following two structures :

1. a standard BDD representing the 1-instance of $f$, called the *Basis BDD* for $f$.
2. a parametric $i$-level BDD (with internal nodes consisting of $i$-level variables, and with leaf nodes that may contain pointers to other function descriptors), representing all $i$-instances of $f$ for $i > 1$, called the *Linearly Inductive BDD (LIBDD)* for $f$.

The function descriptors for LIFs representing the serial adder of Example 1 are shown in Figure 1. Note the parameter substitution on pointers from leaf nodes of LIBDDs to function descriptors, indicated as '$(i-1)/i$' in the figure. The important features of our LIF representations are:

- They are canonical in terms of the parametric inputs (with respect to a given variable ordering).
- Their size complexity, and complexity of symbolic manipulation algorithms, is independent of the parameter of inductive description. Thus, we have a fixed-sized representation for all size instances of a parametric circuit.

Our method for enforcing canonicity of function descriptors involves checking equality of pairs of LIF function descriptors using a built-in principle of induction [15]. This automatic built-in induction is at the heart of our verification methodology, as it allows automatic incorporation of all useful basis cases and hypotheses. This not only results in automation of the induction proof, but also eliminates the associated heuristic search.

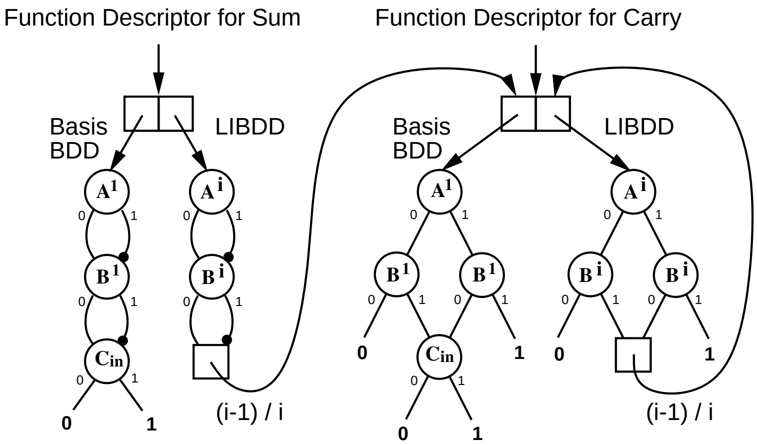


**Fig. 1.** LIF Representation for a Serial Adder

### 2.2 Exponentially Inductive Functions (EIFs)

Inductive instances of functions in this class are formed from Boolean combinations of lower-instance functions applied to the left and right halves of a parametric vector of inputs. If we take an $i$-instance parameterized input to be a vector of $2^i$ scalar inputs, we can again define these functions parametrically. Formally,

**Definition 2**: An exponentially inductive function (EIF) $f$ satisfies the following conditions:

- Condition 1: For $i = 0$, the 0-instance of $f$ ($f^0$) is a Boolean combination ($\mathcal{B}^0$) of 0-instance inputs ($V^0$), each 0-instance input consisting of a scalar input,
  i.e. $f^0 = \mathcal{B}^0(V^0)$.
- Condition 2: For all $i > 0$, the $i$-instance of $f$ ($f^i$) is a Boolean combination ($\mathcal{B}^i$) of $(i-1)$-instances of three functions $e, g$ and $h$, each of which is an EIF,
  $\mathcal{B}^i$ is restricted such that $e^{i-1}$ is applied to the left half of the $i$-instance input (denoted $V^i_L$)), $g^{i-1}$ and $h^{i-1}$ are applied to the right half (denoted $V^i_R$),
  i.e. $f^i = \mathcal{B}^i(e^{i-1}(V^i_L), g^{i-1}(V^i_R), h^{i-1}(V^i_R))$.
- Condition 3: For all $i, j > 0$, $B^i \equiv B^j$.

Space restrictions do not permit us to describe the details of EIF representation and symbolic manipulation, which are similar to those for LIFs. Instead, we give the following examples for EIF representation of practical circuits.

**Example 2**: Consider an $i$-instance tree parity circuit with parameterized input $a$, represented as a $2^i$-bit vector $\tilde{a}[2^i]$. We can express the parity output function $p^i$ as:

$$\text{for } i = 0, p^0(a[1]) = a[1]; \text{ for } i > 0, p^i(\tilde{a}[2^i]) = p^{i-1}(\tilde{a}[2^i]_L) \oplus p^{i-1}(\tilde{a}[2^i]_R).$$

Its EIF representation is shown in Figure 2 (where double circles denote super-nodes that are allowed to point to function descriptors, and "L"/"R" denote substitutions on pointers from super-nodes indicating the left/right half of inputs respectively).

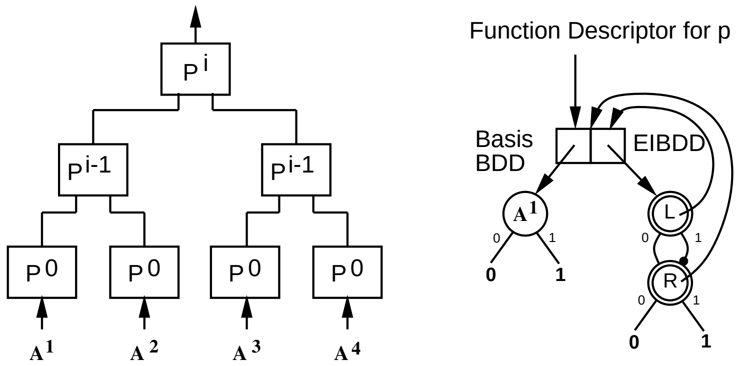


**Fig. 2.** EIF Representation for a Tree Parity Circuit

**Example 3**: The following EIFs $p$ and $g$, represent the carry-propagate and carry-generate functions of a typical log-tree carry lookahead adder with data inputs $x$ and $y$.

for $i = 0, p^0(x[1], y[1]) = x[1] \vee y[1]$,
for $i > 0, p^i(\tilde{x}[2^i], \tilde{y}[2^i]) = p^{i-1}(\tilde{x}[2^i]_L, \tilde{y}[2^i]_L) \wedge p^{i-1}(\tilde{x}[2^i]_R, \tilde{y}[2^i]_R)$.
for $i = 0, g^0(x[1], y[1]) = x[1] \wedge y[1]$,
for $i > 0, g^i(\tilde{x}[2^i], \tilde{y}[2^i]) = (g^{i-1}(\tilde{x}[2^i]_L, \tilde{y}[2^i]_L) \wedge p^{i-1}(\tilde{x}[2^i]_R, \tilde{y}[2^i]_R)) \vee g^{i-1}(\tilde{x}[2^i]_R, \tilde{y}[2^i]_R)$.

## 3 Multiple Parameter Representation Framework

In this section, we describe a canonical representation for a multiple parameter function. This representation can work in conjunction with any schema for canonical representation of a single parameter function, and is not restricted to the single parameter IBF schemata described above. However, we can currently handle descriptions involving *independent* parameters only, i.e. where the function corresponding to any point in the parameter hyperspace can be obtained by inductively applying the parametric definitions, *one parameter at a time*, to the basis case.

**Example 4** : Consider the following inductive descriptions for a function $f$, defined in terms of two induction parameters $i$ and $j$ (denoted $f^{(i,j)}$):

- **Description 1**: basis case for $i = 1, j = 1$ : $f^{(1,1)} = \mathcal{B}_1(X^{(1,1)})$
  inductive case for $i = 1, j > 1$ : $f^{(1,j)} = \mathcal{B}_2(X^{(1,j)}, G^{(1,j-1)})$
  inductive case for $i > 1, j > 1$ : $f^{(i,j)} = \mathcal{B}_3(X^{(i,j)}, G^{(i-1,j)})$
  With this description, the representation of $f^{(m,n)}$ can be obtained by following the solid-line trajectory in the parameter hyperspace as shown in Figure 3. Note that it corresponds to first applying induction along $j$, starting from the basis case $f^{(1,1)}$, and resulting in $f^{(1,n)}$. This latter function then forms the basis for an induction along $i$, resulting finally in $f^{(m,n)}$.
- **Description 2**: basis case for $i = 1, j = 1$ : $f^{(1,1)} = \mathcal{B}_1(X^{(1,1)})$
  inductive case for $i > 1, j > 1$ : $f^{(i,j)} = \mathcal{B}_4(X^{(i,j)}, G^{(i,j-1)})$
  inductive case for $i > 1, j > 1$ : $f^{(i,j)} = \mathcal{B}_3(X^{(i,j)}, G^{(i-1,j)})$
  With this description $f^{(m,n)}$ can be obtained by following an arbitrary orthogonal trajectory, as shown by a dashed line in Figure 3.

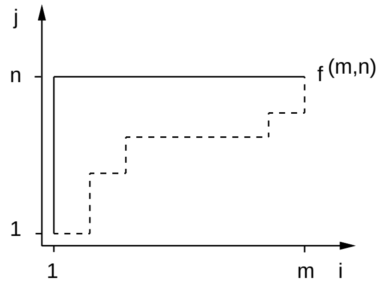


**Fig. 3.** Induction Trajectories in Multiple Parameter Hyperspace

In order to handle multiple case descriptions, we generalize the function descriptor notation described in the previous section. Instead of using two fixed slots for the basis and inductive case descriptions, we use a decision tree representation as shown in Figure 4 for Description 1. The leaf nodes (indicated as boxes) contain pointers to canonical representations of the basis case and inductive case descriptions (in our schema, the Basis BDDs and LIBDDs/EIBDDs respectively). The non-leaf nodes (indicated as circles), called comparison nodes, involve comparison of induction parameters against their basis values (1/0 for LIFs/EIFs respectively). The relative ordering of comparison nodes in a decision tree captures the required induction trajectory as follows. Consider the comparison node for $j$. Its 0-branch is a leaf node with a pointer to the Basis BDD (with $i = 1, j = 1$), and its 1-branch is a leaf node with a pointer to the LIBDD for $j$ (with $i = 1, j > 1$). Thus, the subtree rooted at the comparison node for $j$ effectively captures the inductive representation for $f^{(1,j)}$. This subtree, in turn, forms the 0-branch of the comparison node for $i$, thereby providing a basis for induction along $i$. In this way, the structure of the decision tree provides the appropriate basis functions for different induction steps along the trajectory. This property is crucial in generalizing this representation for capturing arbitrary ordering of the inductive cases.

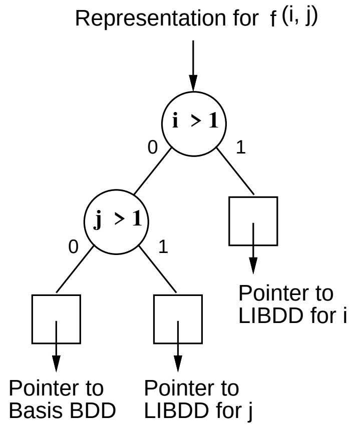


**Fig. 4.** Representation for Multiple Parameter Functions

As for canonicity, note that the leaf nodes of the decision tree can be made canonical using IBF schemata (or any other schema for canonical representation of a single parameter function). For rest of the decision tree, we impose a total ordering on the comparison nodes, specified as a global user-given ordering on the induction parameters. For example, the decision tree of Figure 4 is consistent with a user-given ordering $i < j$, but is in conflict with $j < i$ (and would therefore be unacceptable for this ordering). An ordered decision tree can be reduced by elimination of redundant nodes and isomorphic subgraphs, to obtain a canonical dag (in a manner similar to BDDs). Note that since an induction trajectory implicitly specifies an ordering on the underlying parametric variables ($X^{(i,j)}$), the resulting representations are canonical only with respect to a given ordering of the comparison nodes (as with standard BDDs).

## 4 Parameterization Issues in Circuit Representation

### 4.1 Parametric Circuit Inputs

All parametric circuit inputs in our system are implicitly regarded as infinite vectors of variables (with initial subscript 1). Parameterization is denoted by specifying the length and offset of relevant subvectors in terms of induction parameters. This allows expression of both

- LIF inputs — where an $i$-instance function explicitly depends upon an $i$-instance input and the remaining inputs are implicit arguments of the $(i-1)$-instance functions, and
- EIF inputs — where the left and right halves of an $i$-instance input are implicit arguments of the $(i-1)$-instance functions.

Also, as a matter of convention we place non-parametric control variables before parametric control/data variables in the variable ordering. This allows us to use non-parametric variables for steering the choice of an appropriate inductive function.

### 4.2 Multiple Circuit Outputs

With the representation machinery developed so far, we can represent a single $i$-instance output of an inductively-defined circuit as an $i$-instance IBF. Therefore, a circuit with a fixed number of outputs for each $i$-instance description can be represented easily as a set of IBFs, one corresponding to each output. However, in many cases, the number of outputs of an $i$-instance circuit is also a function of the induction parameter $i$. In such cases, it is not possible to explicitly represent each output as an IBF. Fortunately, there is usually some kind of inductive structure among related multiple outputs, which can be exploited in representing them parametrically using an *output index*. In other words, we use a single IBF to parametrically represent a multiple output function, and provide access to a particular output of interest through explicit use of an output index. We have explored two different alternatives for representation of an output index, as described in the remainder of this section. Note again, that both these alternatives can be used in conjunction with any general schema for canonical representation of a single output function.

**Output Index as a Parameter:** We use the multiple parameter framework described in Section 3, such that each circuit output is regarded as an appropriate projection of an IBF defined over a multi-dimensional parameter hyperspace. However, we now extend it such that comparison nodes are allowed to involve comparison of output index parameters against partitions in their

range. Each path through comparison nodes in a decision tree identifies a particular region of the parameter hyperspace, and the leaf node points to the function description corresponding to that region. Since we use IBF-style function descriptions, we obtain canonical representations of the leaf nodes as before. The remaining task is to represent the decision tree with comparison nodes in a canonical form, *and* to represent only non-null regions in the parameter hyperspace.
The essential idea is to restrict ourselves to a set of totally ordered partitions for each output index parameter. Comparison nodes for different parameters can then be ordered by additionally utilizing the user-given ordering on parameters. Again, the resulting ordered decision tree can be reduced by eliminating isomorphic subgraphs and redundant nodes, to obtain reduced ordered dags that are canonical. Typically, both the range and partitions for output index parameters are specified as functions of size parameters of the circuit (and numerical constants). For an $n$-dimensional parameter hyperspace, we consider partitions formed by hyperplanes only, i.e. each partition can be represented as a linear equation (with real coefficients) in $n$ parameters. We have explored the following two alternatives for ordering the partitions, based on whether or not intersection of hyperplanes is allowed:

– Non-intersecting hyperplanes — The partitions can be totally ordered according to the range of integral values taken by the parameter on each side of the hyperplane.
– Intersecting hyperplanes — Any ordering based strictly on the syntactic representation of the hyperplanes can be used. However, in order to detect non-null regions, a linear programming problem corresponding to each path in the decision tree may need to be solved.

All the circuit examples we have encountered so far, can be handled by non-intersecting hyperplane partitions (e.g. Example 6 in Section 4.5).

**Output Index as a Binary-encoded Argument:** Another technique for representation of an output index is to encode it as a parameterized Boolean-valued argument of a function. In this respect, it is treated identically as a parameterized Boolean-valued circuit input, along with the same syntactic restrictions in the case of IBF representations. A typical application of this method is for an $i$-instance circuit with $2^i$ outputs, such that an $i$-bit argument encodes the output index for an $i$-instance IBF.

**Example 5** : An $i$-instance decoder with $2^i$ outputs can be described in terms of an $i$-bit circuit input argument $a$ (denoted $\tilde{a}[i]$), and an $i$-bit output index argument $j$ (denoted $\tilde{j}[i]$) as follows:

for $i = 1 : dec^1(a^1, j^1) \quad = (\neg j^1 \wedge \neg a^1) \vee (j^1 \wedge a^1)$ ;
for $i > 1 : dec^i(\tilde{a}[i], \tilde{j}[i]) = (\neg j^i \wedge \neg a^i \wedge dec^{i-1}(\tilde{a}[i-1], \tilde{j}[i-1])) \vee$
$\qquad (j^i \wedge a^i \wedge dec^{i-1}(\tilde{a}[i-1], \tilde{j}[i-1]))$.

Note that the $i^{th}$ bit of $j$ (denoted $j^i$) is used for choosing the appropriate half in the range of $j$, and the remaining $(i-1)$ bits are passed as an implicit argument to the $(i-1)$-instance function $dec^{i-1}$. This exactly matches an LIF description, and its LIF representation is shown in Figure 5. We have found this mechanism particularly useful for circuits that are organized inductively in the form of a tree, e.g. prefix parity circuits, carry outputs of a carry lookahead adder etc. Essentially, the vector of outputs at depth $i$ can be divided into halves, such that the $j^{th}$ output within each half depends upon the $j^{th}$ output at depth $(i-1)$. It is the binary-encoding of the output index that allows the exponential nature of a tree circuit to be captured in a linear form.

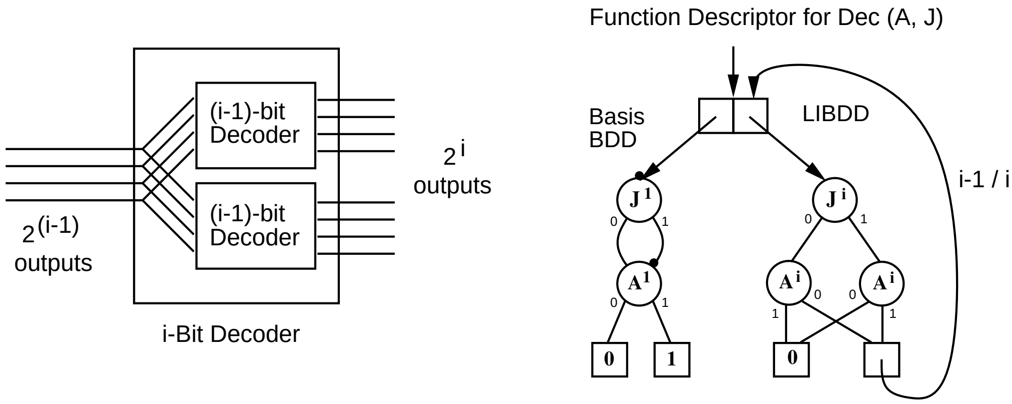


**Fig. 5.** LIF Representation for Decoder Outputs

### 4.3 Vectors of Outputs

The focus of the previous section was on representation of an individual output of a multiple output circuit, where the range of the output index implicitly defines a vector of outputs. However, in some cases we may be interested in representing explicit subvectors of this vector of outputs. This is particularly useful in handling compositions of inductively-defined hardware units, where an output vector from one unit is used as the input vector for another unit. Our current approach is the same as we use for denoting subvectors of inputs, i.e. we identify a subvector of outputs by specifying its length and offset of its first element (with respect to the underlying output vector). We represent both of these components as separate parameters of our inductive description. Similar to the output index, an explicit subvector index is used to represent each output within the range of the subvector, as described in the previous section.

### 4.4 Composition of Inductively-defined Circuits

With most verification approaches, while correctness of *individual* hardware units can be proved by induction, no mechanisms are provided to reason about a *composition* of inductively-defined units. In our methodology, we provide support for handling compositions by automatically obtaining representation of a circuit from representations of its components. Since the component descriptions are themselves parametric, this typically involves resolution of various kinds of parameter interactions as follows:

- Single-instance composition — for a given $i$, $f^i$ is substituted as an $i$-instance input $x^i$ in $g^i$ (denoted $g^i|_{x^i=f^i}$). This can be represented by Boolean operations and restriction [4] as:

  $$g^i|_{x^i=f^i} = f^i \wedge g^i|_{x^i=1} \vee \neg f^i \wedge g^i|_{x^i=0}.$$

  These operations are easily handled by symbolic LIF manipulation algorithms. Note that the substitution for $x^i$ does not affect any $(i-1)$-instance functions appearing in the definition of $g^i$.
- All-instance composition with same parameters — for all $i \geq 1$, $f^i$ is substituted as an $i$-instance input $x^i$ in $g^i$, e.g. composition of an $i$-bit serial parity circuit inputs with outputs

from an $i$-bit serial adder.

This case is an extension of the previous one, where substitution for all $<$-$i$-instances is implicitly represented by applying this operation on all $(i-1)$-instance functions also.

- All-instance composition with different parameters — for all $i \geq 1, j \geq 1, f^{(i,j)}$ is substituted as an $i$-instance input $x^i$ in $g^i$, e.g. the read/write ports of a $(2^i \times j)$-bit register file (Example 6) can be composed with the inputs/outputs, respectively of an $i$-bit serial adder (Example 1).
  This case is more interesting as it requires introduction of an extra parameter in the composed description for $g$. It is handled by representing explicit vectors of outputs (described in Section 4.3) to capture dependence of the $x$ inputs on the extra parameter.
- Basis instance composition — for $i = 1$, $f(y)$ is substituted as the basis instance input $x^1$ in $g^1$, where $y$ represents a set of inputs completely disjoint from the inputs of $g$, e.g. in a CLA, carry-in for the right half of a carry generate-propagate unit (Example 3) is provided by a function on left half of the inputs.
  Currently, we simply substitute in the canonical representation for $f$ in the Basis BDD for $g$.

### 4.5 Results for Practical Circuit Examples

Consider the following circuit example which illustrates an interesting combination of the various representation mechanisms described so far.

**Example 6**: A schematic for a 3-port register file, with $i$-bit address lines $(a, b, c)$, $j$-bit data lines $(in, OutA, OutB)$, and clock/write control signals $(p1, p2, w)$, is shown in Figure 6, Part(a). The new register file contents (after a clock cycle) can be represented using a multiple output LIF $Reg$, where the $2^i \times j$ outputs are indexed by using a binary-encoded $i$-bit argument $k$ and the parameter $j$ itself (with comparison against the basis partition). The LIF representation of $Reg$ is shown in Figure 6, Part(b) (where $r$ denotes the parametric inputs representing the old register cell contents).

In Table 1 we report the memory usage and the CPU time required to obtain canonical LIF representations for the circuits described in this paper. (All measurements were made on a Sparc Station 1+, with 40 Mb of main memory.) In the same table, we have also shown the memory requirements for a BDD representation of the corresponding 4-bit circuit slice. Note that our requirements are quite modest in comparison, and are especially effective in capturing *all* outputs of multiple output parametric circuits.

| Circuit | LIF Representation | | BDD (4-bit Slice) Memory (Kb) | |
|---|---|---|---|---|
| | CPU Time (sec.) | Memory (Kb) | MSB | All Outputs |
| $i$-bit Serial Adder | 0.5 | 1.1 | 0.7 | 1.1 |
| $i$-bit Decoder | 0.5 | 0.6 | 0.2 | 0.8 |
| Register File with $i$-bit Address, $j$-bit Data | | | | |
| Register Cell Outputs | 0.6 | 2.5 | 0.7 | 44.3 |
| Read-port Outputs | 0.6 | 3.5 | 1.9 | 7.7 |

**Table 1.** Results for LIF Representation of Parametric Circuits

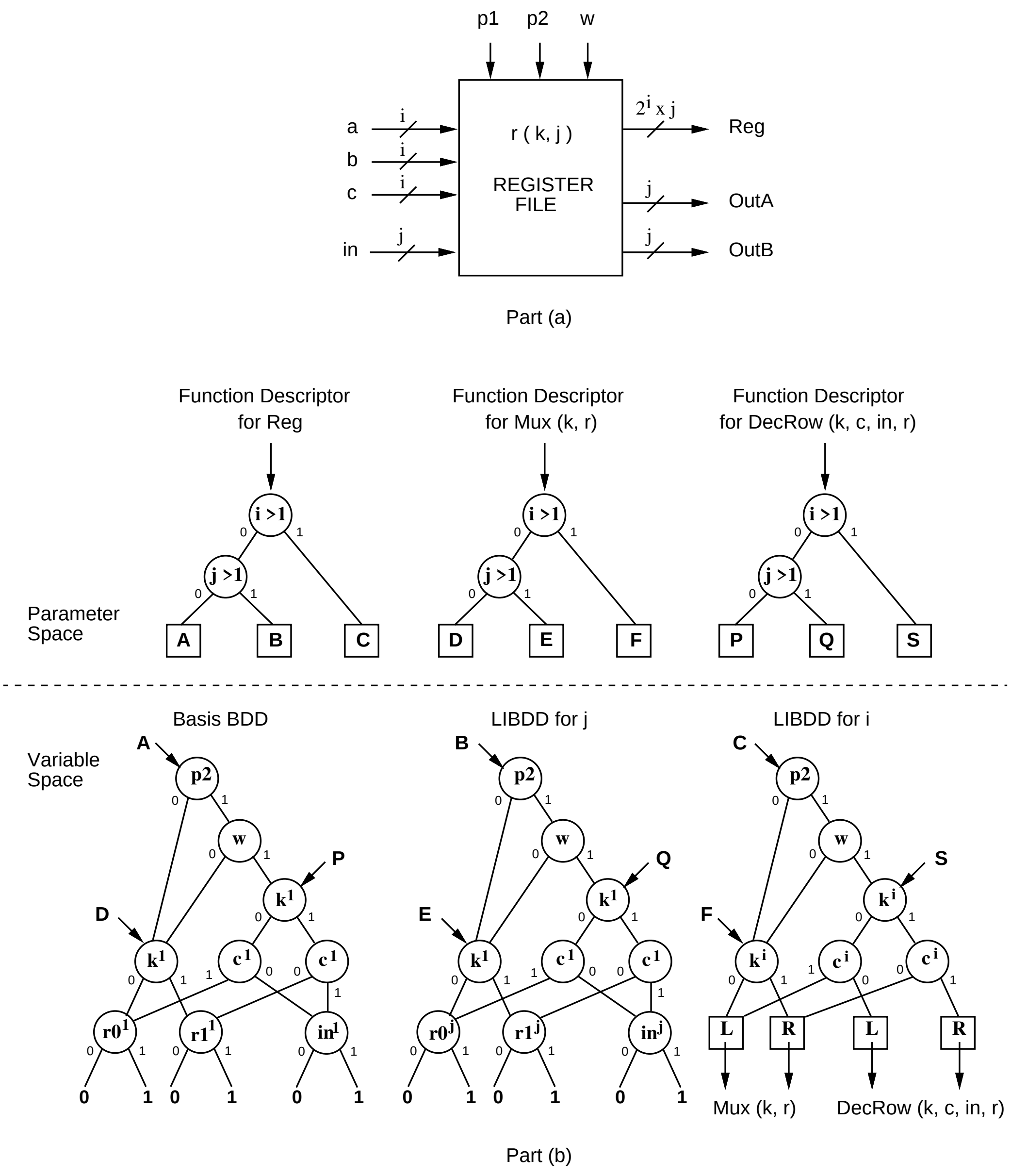


**Fig. 6.** LIF Representation for a Register File

## 5 A Formal Verification Example

Consider the task of verifying the functional correctness of a decoder as described in Example 5 (Section 4.2.2). The property we wish to verify is mutual exclusion of its outputs, i.e for every $i$-instance decoder, only one of its $2^i$ outputs is true, rest are all false. In conventional (non-LIF) notation, this specification property can be stated as follows:

$$\forall i.\exists n.(dec^i[n] \wedge (\forall m \neq n.\neg dec^i[m])),$$

where $dec^i[n]$ represents the $n^{th}$ output of an $i$-instance decoder ($1 \leq n \leq 2^i$).
Recall from our LIF representation of the decoder outputs (shown in Figure 5, Section 4.2.2), that we use a binary-encoded argument $j$ to represent the $i$-bit output index. This trick allows us to replace quantification over integers by quantification over Boolean-valued variables. Thus, the above specification property can be rewritten in our LIF terminology as follows:

$$\forall i.\forall \tilde{a}[i].\exists \tilde{j}[i].(dec^i(\tilde{a}[i], \tilde{j}[i]) \wedge (\forall \tilde{k}[i].(\tilde{k}[i] \oplus \tilde{j}[i]) \Rightarrow \neg dec^i(\tilde{a}[i], \tilde{k}[i]))),$$

where $k$ is another $i$-bit binary-encoded argument. This, in turn, can be translated to the following statement (by using the definitions of $\exists$ and $\forall$ in a Boolean context):

$$\begin{aligned}&\forall i.((newdec^i(\tilde{a}[i], \tilde{j}[i])|_{j^i=0} \oplus newdec^i(\tilde{a}[i], \tilde{j}[i])|_{j^i=1}))_{a^i=0} \wedge \\ &\qquad ((newdec^i(\tilde{a}[i], \tilde{j}[i])|_{j^i=0} \oplus newdec^i(\tilde{a}[i], \tilde{j}[i])|_{j^i=1}))_{a^i=1}.\end{aligned}$$

where $f|_{x=0/1}$ denotes the restriction of a function $f$ for $x = 0/1$, and $newdec$ is a new function derived from $dec$ by pushing in the quantifications for $\tilde{j}[i-1]$ and $\tilde{a}[i-1]$. This statement, denoted $spec^i$, can be viewed as an LIF formula, formed from Boolean combinations of LIFs. Its LIF representation can be automatically derived by simple manipulations on the LIF representation of $dec$, and is shown in Figure 7. Furthermore, a proof of the original specification property by

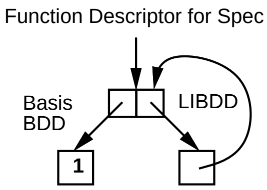


**Fig. 7.** LIF Representation for the LIF Specification

induction on $i$, translates to tautology-checking of $spec^i$ (i.e. checking that $spec^i$ is true for all $i \geq 1$) [15]. Again, this is performed automatically by our method for maintaining canonical LIF representations. For $spec^i$, the tautology-check is very simple as follows:

- The Basis BDD is equal to '1' (truth), signifying that $spec^1$ is true.
- The LIBDD is also equal to '1' (truth), since its leaf node immediately provides the induction hypothesis. In other words, the LIBDD structure implies that $spec^i$ is true if $spec^{i-1}$ is true. Since $spec^{i-1}$ is true due to the induction hypothesis, $spec^i$ is true by induction.

Thus the positive tautology-check establishes the correctness of the decoder circuit for all sizes $i \geq 1$.

## 6 Conclusions and Future Directions

We have presented several general mechanisms for canonical representation of parametric circuits. Of particular interest is the use of a geometric hyperspace framework to capture both induction trajectories and hyperspace regions for representation of multiple parameter functions. BDD-style node ordering and reduction principles are used to argue about canonicity of the decision tree (reduced to dag) representations. In fact, the ordering requirement is an important factor affecting the potential efficiency and expressive power of this framework, e.g. comparison against parameter basis values for simple induction trajectories, and use of non-intersecting hyperplanes for partitioning into regions. Another interesting technique is the use of binary-encoding for representation of an output index. This turned out to be useful both for representation of an exponential size tree circuit in a linear form, and for changing the domain of quantification from integers to Boolean-valued variables in a specification property.

Used in conjunction with IBF representation schemata, we have given several examples and preliminary results for representation of practical circuits in the form of IBFs. These representations form an essential component of our verification methodology based on automatic symbolic manipulation of IBFs. Though a complete description of the verification process is beyond the scope of this paper, our simple decoder example demonstrates the essential ideas of automatic incorporation of the basis and hypotheses in an induction proof.

In terms of future directions, development of a formal logic for specification using IBFs is potentially useful. We also plan to examine other interesting inductive circuits, e.g. multipliers. This may require characterization of additional classes of IBFs (other than LIFs and EIFs), though our main ideas and algorithms can probably be generalized. Another interesting direction is the use of our representations to reason about functional equivalence of different inductive descriptions, e.g. equivalence of a serial parity circuit and a parallel tree parity circuit. Preliminary work indicates that by deliberate manipulation of the two representations, it may be possible to automate a proof of their equivalence. Such proofs are of practical importance in verification of correctness preserving transformations, an essential component of most automated tools for hardware design.

## References

1. S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic simulation and temporal logic. In L. M. J. Claesen, editor, *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Belgium, 1989*, volume II, pages 151–158. North-Holland, Amsterdam, 1990.
2. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.
3. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45. IEEE Computer Society Press, Los Alamitos, CA, June 1990.
4. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. R. E. Bryant. A methodology for hardware verification based on logic simulation. Technical Report CMU-CS-87-128, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, June 1987.

6. J. Burch, E. M. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society Press, Washington, D.C., June 1990.
7. A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. North-Holland, Amsterdam, 1987.
8. E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of Futurebus+ cache coherence protocol. Technical Report CMU-CS-92-206, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1992.
9. E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 294–303. ACM, New York, August 1987.
10. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 1989*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, New York, 1990.
11. O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Springer-Verlag, New York, 1991.
12. A. J. de Geus. High level design: A design vision for the 90's. In *Proceedings of the IEEE International Conference on Computer Design*, page 8. IEEE Computer Society Press, Silver Spring, MD, 1992.
13. S. M. German and Y. Wang. Formal verification of parameterized hardware designs. In *Proceedings of the IEEE International Conference on Computer Design*, pages 549–552. IEEE Computer Society Press, Silver Spring, MD, 1985.
14. A. Gupta. Formal hardware verification methods : A survey. In R. K. Brayton, E. M. Clarke, and P. A. Subrahmanyam, editors, *Formal Methods in System Design*, volume 1 (Nos. 2/3). Kluwer Academic Publishers, Boston, October 1992.
15. A. Gupta and A. L. Fisher. Representation and manipulation of inductive Boolean functions. Technical Report CMU-CS-92-129, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, April 1992.
16. W. A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.
17. R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 239–247. ACM, New York, 1989.
18. K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, 1991.
19. J.-K. Rho and F. Somenzi. Inductive verification of iterative systems. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 628–633. IEEE Computer Society Press, Los Alamitos, CA, June 1992.
20. D. Verkest, P. Johannes, L. Claesen, and H. De Man. Correctness proofs of parameterized hardware modules in the Cathedral-II synthesis environment. In *Proceedings of the European Design Automation Conference Glasgow*. IEEE Computer Society Press, Washington, D.C., 1990.
21. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 1989*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, New York, 1990.