Revisiting BLIF-MV, An Intermediate Format For Verification and Synthesis of Hierarchical Networks of FSMs

Abstract

This document serves to define and extend BLIF-MV, adding some new constructs and making some syntactical changes. BLIF-MV is an intermediate format used as input to synthesis and verification systems. Synthesis subsets of high-level languages, such as Verilog, VHDL, and Esterel, can be mapped to BLIF-MV. Our contributions are as follows:

1. Defining a precise semantics for BLIF-MV, and allowing synchronous and asynchronous components to co-exist.

2. Introducing constructs for dealing with integer variables.

3. Defining a set-theoretic language and other constructs for more compactly expressing relations.

4. Introducing constructs for controlling the collapsing extra relations introduced by compilation from high-level languages.

5. Precise definition of hierarchical development and restrictions imposed by BLIF-MV.

6. Introducing application-specific attributes, which can be used, for example, for source-level debugging and marking of temporary variables created during translation from high-level languages.

7. Allowing multiple output relations, and system-defined relations.

This document describes the full syntax of BLIF-MV, including the new changes. The changes make BLIF-MV a more compact and expressive intermediate format for formal verification, logic synthesis, and possibly hardware/software co-design. At UC Berkeley, it is used as input to HSIS, a hierarchical synthesis and verification system.

1 Introduction

Formal verification is the process of verifying designs by proving properties about them. To make formal verification possible in real designs, several problems must be solved. We consider several integration problems, and discuss techniques to solve them.

The most important issue in integration is making verification compatible with logic synthesis and simulation. We have addressed this problem by making our synthesis and verification tool, HSIS, use the same input language and internal data structures. Our input language is an intermediate format, BLIF-MV (HSIS uses BLIF-MV as an output language also). One can map the synthesis subsets of all high-level languages into BLIF-MV. Note that the synthesis subset of high-level languages describes networks of FSMs. Currently, we support a synthesis subset of Verilog, a simulation language, plus some constructs added for dealing with non-determinism. Using this paradigm, verification, simulation, and synthesis, all use the same description.

Our design methodology for synthesis and verification is as follows. The design is first written in abstract terms, using non-determinism, in the user's favorite high-level language. The design is then translated into BLIF-MV, and verification is performed on it. Simulation may be used at this point to find easy bugs (HSIS provides a state-based simulator), whereas formal verification can be used to catch the harder ones.

The user has two options after verifying all or part of the design. If the user is satisfied with the verified parts, they can be synthesized. For synthesis, we use our sequential synthesis tool SIS ([Sen92]), which shares its data structures with verification. SIS can only handle one FSM at a time. However, research is on-going to augment the current techniques by methods to handle the synthesis of hierarchical networks of FSMs. One can imagine that some parts of the design may be synthesized into software. In this case, BLIF-MV is used for hardware/software co-design.

If the user is not satisfied with verification, the user can proceed by adding more detail to the design. As long as new behavior is not added, certain types of properties proved at higher levels (basically properties expressible by an ω-automaton ([Hoj93])) of abstraction will continue to hold at lower ones.

However to make verification possible in practice, verification tools must be more user-friendly. So the following issues should be resolved.

1. The semantics of the intermediate format should be well-defined, so that any user's specification is translated un-ambiguously. We define these semantics. Moreover, we allow synchronous and asynchronous design to co-exist.

2. The interaction with the user must feed back variable names defined by the user. Many high-level languages allow for symbolic or multi-valued variables. BLIF-MV accommodates this by allowing multi-valued (MV) variables.

3. The generated code for the intermediate format should remain relatively small, so that parsing time is reasonable. BLIF-MV uses relations to describe (possibly non-deterministic) behavior. We introduce a set-theoretic language and other constructs, which makes the specification of these relations more compact. We also introduce facilities for easily dealing with integers, since functions defined on integers, such as addition, cannot be described compactly using multi-valued relations.

4. It is useful to display error traces, returned by the verifier, in terms of the user's original source code. For example, we plan to use the application-specific attributes, mechanism of BLIF-MV described below, to pass line number information to the verification system. The verification system can then determine where the program counters are when a global state is encountered in a debug trace (a program counter is placed wherever an assignment to some state or output variable is made). This information can then be reported back in a user-friendly manner.

5. It is important that the original structure of a user's specification is preserved as much as possible for two reasons. First, interfacing with the user becomes easier. Second, some algorithms may be able to take advantage of the initial structure. BLIF-MV preserves two important structural aspects: hierarchical specification, and concurrency model. By introducing high-level system-defined relations (described below), more of the structure of the original structure is preserved.

In addition to the changes described above, we have made small changes to the language, such as allowing multiple output relations which make compilations into BLIF-MV easier and more compact.

As examples of use of BLIF-MV's use, we show how to translate models specified in various widely-used concurrency models into BLIF-MV. We consider the following concurrency models: selection/resolution model ([Agg83]) used in COSPAN ([Har90]), shared-variable used in Murphi([Dil92]), and CSP-like message-passing used in many systems and languages.

This paper summarizes our experiences in the past three years with building an intermediate format for verification and synthesis. The first, [Bra91], was implemented in HSIS. This document is the result of efforts to overcome certain bottlenecks and deficiencies. We believe that BLIF-MV can be easily used as an intermediate format for verification and synthesis systems, possibly for hardware/software co-synthesis as well, spanning an even larger spectrum in system design.

Currently, Verilog is the main language compiled into BLIF-MV. We expect in the near future Esterel and VHDL compilers will become available. This will facilitate the exchange of examples, and comparisons of various tools and algorithms, and will ease the use of synthesis and verification tools in practice.

The document is organized as follows. In section 2, we define the concurrency model/semantics of BLIF-MV, giving a high-level description of the language. In section 3, we describe the basic syntax of BLIF-MV, and in section 4, we describe some advanced features; specifically, integer handling, handling of “temporary” relations introduced by compilation, and use of system-defined relations. Section 5 includes some examples. Conclusions are given in section 6.

2 Combinational/Sequential Concurrency Model[1]

In this section, we describe the concurrency model of BLIF-MV, which we call the combinational/sequential (CS) concurrency model. There are four primitives: variables, tables (intuitively non-deterministic gates), wires, and latches.

Definitions A ***variable*** takes values from some finite domain. A ***table*** is a relation defined over a set of variables. Hence, the domain of a table is the cartesian product of the domains of its variables. The variables of a table are divided into two sets: inputs and outputs. Tables are connected by ***wires***, which connect an output variable of a table to an input variable to a table or latch (where both variables are the same). We assume a variable is the output of at most one table. The possible values of the wires are the set of elements in the domain of its variable. Special elements called ***latches*** may be placed on wires. When a latch is placed on a wire, the wire is broken into two pieces, one called the input to the latch, and the other the output of the latch. With every latch we associate a set of initial values, which are a subset of the set of values of its wire. [2]

Definitions A ***state*** is an assignment of values to the latches of a model, where a value assigned to a latch must be in its domain. An ***initial state*** is a state where every latch takes a value from its set of initial values.

If the tables are deterministic, then we can think of them as gates. In this case, the above model is syntactically the same as synchronous hardware. The concurrency model of ***synchronous CS*** is exactly the same as synchronous hardware also, modulo the fact that tables may be non-deterministic. At every point, the system is in some state, with the ouputs of the tables having some stable (consistent) set of values. At every step, all latches update their values, i.e. the outputs of the latches take the values of their inputs. These values then propagate through the tables, until all lines have a consistent set of values. If during propagation a latch is encountered, i.e. an output of a table is an input to a latch, we stop the propagation of values. Note that because of non-determinism, given a state, there may be several consistent set of values. This process continues indefinitely in time. We assume the first state is an initial state of the model.

In ***general CS***, at every step, only a subset of latches update their values. We are given a set of rules, described below, that determine which set of latches should update their values. These rules may be non-deterministic, i.e. at every step we may have a finite set of subsets of latches, from which one subset is chosen non-deterministically. The chosen latches update their values, and these are propagated through the tables. In ***asynchronous CS***, which is at the opposite end of spectrum compared to synchronous CS, only one latch may update its value at each step. The latch to be updated is chosen non-deterministically. Usually, when we have asynchronous communication, there are some parts of the model which communicate synchronously. Hence, asynchronous CS does not occur frequently in practice.

We use the notion of a concurrency hierarchy or tree of a model to determine which subsets of latches could update their values.

---

1. The definition of combinational/sequential concurrency model is due to Ramin Hojati.
2. The name combinational/sequential comes from the fact that a wire with a latch placed on it is called a sequential wire, whereas a wire without a latch is called a combinational wire.

Definition A ***concurrency hierarchy*** of a model is a tree whose leaves are exactly the latches of the model. Each tree node is labeled by concurrency attributes with value A (for asynchronous) or S (for synchronous).

To choose what latches to update, we use the following recursive algorithm. The algorithm, given a node in a concurrency tree of a model, determines what latches under the node's subtree should be updated.

0 If the node is a leaf, return the latch.
1 If the node is labeled by A, return the set returned by calling the algorithm on one of the node's children (chosen non-deterministically).
2 If the node is labeled by S, return the union of the sets returned by calling the algorithms recursively on all of the node's children.

Synchronous CS is a powerful concurrency model, able to compactly express all useful concurrency models known to the authors except for partial order models. Specifically, synchronous CS is able to compactly express general CS. However, the use of interleaving semantics (exemplified by an A node) has two advantages. First, there are (BDD-based) algorithms which can take advantage of the interleaving semantics, and hence may be more efficient. Second, some synthesis algorithms, for example hardware/software co-design algorithms, may take advantage of the additional information provided by general CS. In summary, although synchronous CS is sufficient to express most practical systems, the additional structure preserved by general CS may be useful to some algorithms. In the next two sections, we will describe BLIF-MV, which implements this concurrency model.

3 Basic Syntax

In this section we describe the basic syntax of BLIF-MV. The primitives of the language are: multi-valued variable declarations, tables for describing relations, multi-valued latches, models for describing a process, and sub-circuits for instantiations of models.

3.1 Variables

A multi-valued variable is declared as follows.

.mv <variable-list> <range> [<value-list>];

where <> implies a necessary argument, and [] an optional one. A variable-list is a set of comma-separated variables. A value-list is a set of space-separated names. The default on value list is 0,..., range-1. Example are,

```
.mv var1, var2 3 apple orange banana;
.mv four-bit-integer 16;
```

3.2 Tables

Tables are used in BLIF-MV for describing relations. In this section, we first give the basic method of defining a table, including a set-theoretic language for defining table entries. More advanced features are introduced next. Symbolic entries are used to give equality relations between two variables. Default values are used whenever an "else" in a high-level language would be used.

3.2.1 Table Definition

A table describes a relation between a set of variables. A table has a set of inputs and a set of outputs. A table is defined as follows.

.names <in-1> ... <in-n> => <out-1> .... <out-m>;
<entry-1>;
.
<entry-p>;

Inputs and outputs are sets of variables. If there is only one output, then the symbol "=>" can be omitted. A table may have no inputs, but at least one output. The entries of the table describe a relation between the variables. Each entry has $m + n$ components. Each component of an entry describes a set of values, which is a subset of the values of the corresponding variable. Each entry denotes the (cartesian) product of these components. To describe a component, we use a set theoretic language, where "-" denotes the universal set, i.e. the set of all values for a variable,

"," denotes union, and "!" denotes complementation. The language is fully parenthesized. As an example, assume variables *x* and *y* can each take 100 values, from 0-99. Consider the following table.

```
.names x y;
(10,21) !(3);
5 -;
(3-8) (5-15);
```

This first entry describes a relation between 10 and 21 and any number between 0 and 99 except for 3. The second entry says that 5 is in relation with any number. The third entry says that numbers between 3 and 8 are in relation to numbers between 5 to 15. In this case, *x* is the input and *y* is the output. We restrict parenthesis to only one level. So, (5-7,!(5-12)) is not allowed.

3.2.2 Symbolic Entries

One can also use ***symbolic entries*** in table specifications. Assume under a column corresponding to variable x, we have the variable y (the syntax is '=*y*). Then, the relation corresponding to that entry is multiplied by the relation $y = x$. For example, consider a multi-valued multiplexer, which has the following semantics: if $x = 0$, then $z = y$, otherwise $z = w$, where x, is a binary variable controlling the multiplexer, and y, z, and w are multi-valued variables of the same type. The table for this multiplexer is describe as follows.

```
.names x y w z;
0 - - '=y;
1 - - '=w;
```

The only restriction with symbolic entries is that the two variables which are made equal must have the same domain, i.e. be of the same type.

3.2.3 Default Values

To make specifications of functions where only a subset of the relations needs to be given, the ".def" (for defaults) construct is introduced. With every table, a default value for the outputs can be given, which is assigned to all the minterms in the input domain which were not assigned in the table. An example is as follows.

```
.names x1 x2 => y2 y2;
.def 0 0;
1 - 1 1;
```

Now, the two unspecified minterms, i.e 00 and 01, are assigned to 00.

In BLIF ([BlifRef]), an intermediate format for describing hardware, we automatically use the default notion for binary variables. For example, the BLIF description

```
.names a b c;
.def 0;
0 - 1;
1 1 1;
```

is written in BLIF-MV as

```
.names a b c;
0 - 1;
1 1 1;
```

3.3 Latches

3.3.1 Latch Definition

A latch is declared as follows.

.latch <latch-input> <latch-output>;

where latch-input and latch-output are variables of the same type. Latch-input must be the output of some table, or an input to the model (we describe models shortly). Latch-output can be fed into another table, or can be the output of the model. An example is,

```
.latch ns ps;
```

where "ns" and "ps" are variables, both of the same type, say (apple, orange, banana).

3.3.2 Latch Initialization

A latch is initialized (in general with a set of values) as follows.

```
.r <in-1> ... <in-n> <latch-output>;
<entry-1>;
.
<entry-m>;
```

where latch-output is the output variable of the latch we want to initialize, and in-1,...,in-n are any set of variables. The syntax for describing the entries is the same as the syntax for describing tables. An example is,

```
.r initial-state-input state;
thinking thinking;
reading reading;
```

This example says that if the variable initial-state-input is thinking (reading), then the state is initialized to thinking (reading). Hence, it is possible to write models where the initial state of the latches are parametrized. A latch may have more than one initial state. Below, we show three ways to specify that the latch "ps" has two initial states, 0 and 1.

.r ps; 0; 1;	or	.r ps; (0,1);	or	.r ps; -;

Note that since ps is the only variable in the table, it is the output. Equivalently, ".r => ps;" could have been used.

3.4 Hierarchical Development

We describe how hierarchies are defined and referenced. Some restrictions placed by BLIF-MV on hierarchies are described next. Since verification usually deals with hierarchies of depth one, we describe how hierarchies are flattened for verification.

3.4.1 Module Definition

In BLIF-MV, hierarchy is defined as a syntactic notion. This allows one to take a portion of a model and put a box around it. This portion can then be re-used. The unit of hierarchical development is called a model or module, and is declared as follows.

```
.model <model-name>;
.inputs <variable-1> ... <variable-n>;
.outputs <variable-1> ... <variable-m>;
<command-1>;
.
<command-p>;
.end;
```

A command is any one of the following: variable declaration, table definition, model instantiation (or reference), latch definition, or initial state declarations. Given a model's definition (in isolation), we place the following restrictions.

1. In BLIF-MV, we distinguish between four sets of variables: inputs (in), outputs (out), present state (ps), and next state (ns). The following table shows if possible overlaps between these sets of variables can exist.

	in	out	ps	ns
in		no	no	yes
out	no		yes	yes
ps	no	yes		no
ns	yes	yes	no	

Every entry in this table specifies whether the corresponding sets of variables can have the same actual variable. For example, an in variable can be a ns variable, whereas an in variable cannot be a ps variable.

The justification for these rules are

1a. in != out, which means an input and an output variable cannot be the same. In a model definition, BLIF-MV requires inputs and outputs to be distinct. However, as we will see, one can connect the inputs and outputs of a model instantiation.

1b. in != ps. This rule requires the output of a latch defined in a model not to be an input to the model.

1c. in = ns. This is allowed because we may want to input the next state of a neighboring model.

1d. out = ps. We may want to directly communicate the present state value of a latch defined in one of the sub-circuits with another one, without having to introduce another variable "out" with the relation out = ps[1].

1e. out = ns. Once the next state value is calculated, we may want to communicate that value with another block.

1f. ps != ns. In order to build a transition relation for the FSM corresponding to a model, the present and next state variables must be distinct.

3.4.2 Module References

A model is instantiated as follows.

.subckt <model-name> <instance-name> <formal-actual-list>;

The formal-actual-list defines the correspondence between formal and actual variables. This list is a sequence of assignments, each separated by a space. The declaration is as follows.

formal-1=actual-1 ... formal-n=actual-n

The actual variables are ones declared in the current model, whereas the formal variables are declared in the referenced model.

A hierarchy in BLIF-MV is always a tree, which implies that a sub-circuit cannot be referenced from any other sub-circuits except for its parent. The first model encountered in a BLIF_MV file is the root of the hierarchy.

3.4.3 Restrictions on Hierarchies

We need a few more definitions to explain the rest of the rules.

Definitions Let a ***collapsed hierarchy*** be a hierarchy where there are no module references, i.e. there is no hierarchy. One can turn a hierarchy into a collapsed one by recursively substituting the definition of a module, whereever there is a reference to one. An ***actual variable*** of a variable x is the variable associated with x in the collapsed hierarchy. To find the actual variable the hierarchy is traversed upward until a definition for the variable is found.

The rules for use of hierarchy in for actual variables are summarized as follows.

---

1. This is simply an efficiency issue, since computational efficiency directly relates to the number of variables in the system.

	in	out	ps	ns
in		yes	yes	yes
out	yes		yes	yes
ps	no	yes		no
ns	yes	yes	no	

The justification for the rules which are different than those for model definitions is given below.

1. in = out, which means an input and an output variable can correspond to the same actual variable. An example is where some module definition is given, and the user would like to connect an output to an input.

2. in = ps. The only situation where this is allowed is when the ps variable is an output variable which is in turn an input variable. If not, the ps actual variable, call it x is driven by more than one source when we collapse the hierarchy; the two drivers are the latch and the table or latch which produces the input variable.

3. ps != ns. In order to build a transition relation for the FSM corresponding to a model, the present and next state variables have to be distinct. Since, ps and ns variables are distinct in a model definition, the only way ps and ns variables can have the same actual variable is if the output of a latch is directly fed into the latch. This describes a one-state FSM, which always holds its initial value. To describe this behavior, a buffer must be inserted between the output and the input of the latch.

3.4.4 Verification and Hierarchies

Verification is usually done on a network of FSMs. To obtain this, the hierarchy is reduced (but not demolished) by the procedure below.

Definition Let the ***flattened view of a hierarchy*** be a set of interacting collapsed hierarchies, obtained as follows.

1. For each sub-circuit, we add all the variables used as input (output) variables to module references as inputs (outputs).

2. Replace all the formal inputs/outputs by the actual ones.

3. Remove all the module references.

Intuitively, this is a hierarchy of only one level, where all sub-circuits of the original hierarchy are promoted to the first level.

A question that comes up whether we allow several sub-circuits in a flattened view to correspond to the same actual variable. We first need a definition.

Definition There are two sets of output variable in a model.

1. ***Driven output variables***, which are the output of some latch or table.

2. ***Through output variables***, which are the output of some child. This comes up in a hierarchical specification when an output o of a child is the output of the parent also, which makes o the outputs of both the child and the parent in the flattened view.

In BLIF-MV, we allow several outputs to correspond to the same actual variable, so long that in the collapsed hierarchy all variables are driven by only one source. This implies that all instances of an output variable are through variables except for exactly one. From the BLIF-MV description, one can easily find the through output variables, which are those output variables of a block which are not set in the module but are the output of a sub-circuit call (any other scenario violates the requirement that each variable in the collapsed hierarchy has only one source).

3.5 Concurrency Specification

One can also specify the concurrency model associated with a given BLIF-MV model using ".conc" construct. The concurrency model is specified using an arithmetic expression with the identifiers being latches and sub-circuits included in the model, and the operators being "*" (for synchronous composition) and "+" (for asynchronous composition). If a latch or sub-circuit is not explicitly specified, then it is assumed to be in synchronous composition with the root model of the concurrency specification. As an example, assume a model has two sub-circuits sub1 and sub2, and three latches, lch1, lch2, and lch3. Consider the following specification.

```
.conc (sub1 * lch1) + sub2 +  lch2;
```

This specification says that sub1 and lch1 are in synchronous composition, and their result is in asynchronous composition with sub2 and lch2. Since lch3 is un-specified, it is assumed that it is in synchronous composition with the root of the concurrency model. The effect is as if the following concurrency model was give.

```
.conc ((sub1 * lch1) + sub2 +  lch2) * lch3;
```

Note that combinational logic is always assumed to be in synchronous composition with all other modules.

3.6 Others

To specify comments the /* */ construct is used.

4 Additional Features

In this section, additional features of BLIF-MV are described.

4.1 Integers

There are some arithmetic operations which are very expensive to describe in terms of symbolic variables. Assume we want to specify the function $c = a + b$, where a and b are 16-bit integers. The symbolic table will be of size $2^{16}2^{16} = 2^{32}$! However, one can easily describe a 16-bit adder using the binary-valued components of the symbolic variables.

To handle such situations, the "bundle" construct is used. A bundle is a set of binary variables associated with an integer variable. A bundle is declared as follows.

.bundle variable-name bit-n ... bit-1;

where bit-n is the most significant bit. As an example, to describe the relation $c = a + b$, we use the following reference.

```
.subckt adder16 myadder16 x=a y=b z=c;
```

At some later point, we define adder16 as follows.

```
model adder16 x y z;
.inputs x y;
.outputs z;
.bundle x x15 x14 ... x0;
.bundle y y15 y14 ... y0;
.bundle z z16 z15 ... z0;
/* a serial bit description of 16-bit adder */
.
.end;
```

4.2 General Attributes

There is some information which is specific to certain tools. To allow for such information, called attributes, BLIF-MV uses the following general-purpose attribute construct.

%attribute-name[information]%

One can associate an arbitrary number of attributes with BLIF-MV's basic objects. Specifically, attributes can be associated with tables, variables, latches, sub-circuits, and models. The attributes associated with an object must be specified in pre-defined positions. For tables, attributes must be given right after ".names" keyword; for variables, right after ".mv" keyword; for latches, right after ".latch" keyword; for initialization of latches, right after ".r" keyword; for sub-circuits right after ".subckt" keyword; for models, right after ".model" keyword. Examples of attributes are temporary variables recognized by some tools, and source-code debugging information. For instance, variables "temp1" and "temp2" are defined to be temporary as follows.

```
.mv %tempvar:% temp1, temp2 5;
```

Other examples are keywords to identify operations to be performed, and "don't touch" indicator to not synthesize a model.

4.3 Temporary Sub-circuits

Our synthesis tools make the assumption that every declared sub-circuit (using ".subckt" command) is part of the initial hierarchy. However, sometimes during translation from high-level languages, temporary sub-circuits are created. For example, to translate an if-statement, a multiplexer may be created. Hence, these extra sub-circuits are just aides in describing the calling process. To signify that such constructs will not be in the initial hierarchy, and should be collapsed into models which use them, the ".macro" command is used. Note that this construct is different than macro constructs used in programming languages to denote in-line expansion of code. A temporary subckt is declared exactly the same way as a ".subckt", except the word ".subckt" is replaced by ".macro".[1] An example is as follows.

```
.macro mymux mux2 control=a in1=b in2=c out=z;
```

4.4 System-Defined Functions

4.4.1 Definition

To improve the efficiency of parsing, some tools may define a set of pre-defined models. A system-defined model is used like any other model. A system-defined model can be referenced using both ".subckts" and ".macro" constructs. The following is an example.

```
.subckt HSIS_ADDER16 myadder16 x=a y=b z=c;
```

which references a 16-bit adder defined by HSIS. If the example is to be distributed to tools which do not define these system-defined models, a definition of the model in BLIF-MV must be included, using the bundle construct and a bit description of the adder. Hence, there is a BLIF-MV model associated with every system-defined relations.

4.4.2 Relations Defined by HSIS

In HSIS, we have defined the following set of functions.

1. Integer arithmetic.

1a. Addition. HSIS_adder_bitnum (x, y, z), where x, y, z have bitnum bits, and the relation expressed is $z = x + y$, x and y are inputs and z is the output.

1b. Subtraction. HSIS_subtract_bitnum (x, y, z), where x, y, z have bitnum bits, and the relation expressed is $z = x - y$, x and y are inputs and z is the output. If $x > y$, then $z = 0$.

1c. Multiplication. HSIS_subtract_bitnum (x, y, z), where x, y have bitnum bits and z has (2 bitnum) -1 bits, and the relation expressed is $z = x \times y$, x and y are inputs and z is the output.

1d. Division (integer). HSIS_subtract_bitnum (x, y, z), where z, y have bitnum bits and x has (2 bitnum) -1 bits, and the relation expressed is $z = x / y$, x and y are inputs and z is the output.

2. Comparison.

---

1. The difference between .macro and in-line expansion is that for .macro, we have the choice of building the transition relation once and then substituted into each instance of the macro, while for in-line expansion, a new transition relation has to be built for each instance.

2a. Equality. HSIS_eq_valnum (x, y, b), where x, y have valnum values and b is a bit specifying the result of the comparison.

2b. Non-equality. HSIS_neq_valnum (x, y, b).

3. Integer Comparison.

3a. Less than or equal. HSIS_leq_bitnum (x, y, b).

3b. Less than. HSIS_lt_bitnum (x, y, b).

3c. Greater than or equal. HSIS_geq_bitnum (x, y, b).

3d. Greater than. HSIS_gt_bitnum (x, y, b).

4. Logical operations on binary variables. The following are provided: HSIS_and, HSIS_or, HSIS_xor, HSIS_xnor, HSIS_not.

5. General multiplexor. The relation HSIS_mux_bitnum (ctl, x, y, z), where x, y and z have bitnum bits and ctl is a bit controlling the multiplexor. If $ctl = 0$, then $z = x$, otherwise $z = y$.

4.5 Miscellaneous

The ".include" command can be used to include files. An example is as follows.

```
.include utilities.mv;
```

5 Conclusion

In this document, we have solved several problems of BLIF-MV by adding constructs or making small syntactical changes. Specifically, we have defined a precise semantics for BLIF-MV, defined constructs for dealing with integers, introduced a set-theoretic language for defining relations that makes specification of relations more compact, and introduced syntax for specifying multiple-output relations and arbitrary attributes.

Contributions

This document was prepared by Ramin Hojati, Thomas R. Shiple and Robert K. Brayton. Many of our decisions about BLIF-MV's syntax were made in group meetings. Gerard Berry, Szu-Tsung Cheng, and Macs Chiodo for made several useful suggestions. Contributions of Adnan Aziz, Wendell Baker, Rejeev Ranjan, Herve Touati, Gary York, and Huey Wang are respectfully acknowledged.

6 References

[Agg83] S. Aggarwal, R. P. Kurshan, K. Sabnani, "*A Calculus for Protocol Specification and Validation*", in Protocol Specification, Testing and Verification III (193) North-Holland, pp. 19-34.

[Bra91] Robert K. Brayton, Massimiliano Chiodo, Ramin Hojati, Timothy Kam, Kolar Kodandapani, Robert P. Kurshan, Sharad Malik, Alberto L. Sangiovanni-Vincentelli, Ellen M. Sentovich, Thomas R. Shiple, Kanwar J. Singh, Huey-Yih Wang, "BLIF-MV: *An Interchange Format for Design Verification and Synthesis"*, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, UCB/ERL M91/97, Nov 1991.

[Dil92] D. Dill, A. J. Drexler, A. J. Hu, C. H. Yang, "*Protocol Verification as a Hardware Design Aid"*, ICCD 92.

[Har90] Z. Har'El and R. P. Kurshan, "Software *for Analytical Development of Communication Protocols"*, AT&T Journal, Jan 1990, 45-59.

[BlifREf]