BLIF-MV

Yuji Kukimoto

The VIS Group
University of California, Berkeley
vis@ic.eecs.berkeley.edu

May 31, 1996

BLIF-MV is a language designed for describing hierarchical sequential systems with non-determinism. A system can be composed of interacting sequential systems, each of which can be again described as a collection of communicating sequential systems. This makes it possible to describe systems in a hierarchical fashion. Although BLIF, the input language for the logic optimization system SIS, also has constructs for describing hierarchies, they are automatically flattened into a single-level circuit once they are read in because the internal data structure of SIS does not support hierarchical representations. In VIS, however, the original hierarchy is preserved in internal data structures so that true hierarchical synthesis/verification is possible. Another important extension to BLIF is that BLIF-MV can describe non-deterministic behaviors. This is done by allowing non-deterministic gates in descriptions. Non-deterministic gates generate an output arbitrarily from the set of pre-specified outputs. These allow us to model non-deterministic systems in the VIS environment, which is crucial in formal verification since designs in early stages are likely to contain non-determinism. Lastly BLIF-MV supports multi-valued variables, which can be used to simplify system descriptions.

1 Syntax

1.1 Models

A model is a system that can be used in defining a hierarchical system. Any BLIF-MV file contains one or more model definitions. If there is more than one model definition, one model is specified as the root model by putting the `.root` construct in the next line of the `.model` declaration. An entire hierarchy is created from this model recursively. If no model is declared as the root model, the first model serves as the root mode. The `.root` construct can optionally have an argument, which specifies the instance name of the root node. If no argument is used, the instance name is the same as the model name. A model is declared as follows:

```
.model <model-name>
.inputs <input-list>
.outputs <output-list>
<command>
...
<command>
.end
```

- *model-name* is a string giving the name of the model. A string should be composed of lower-case/upper-case alphabetic characters, numbers, or symbols \$, $<$, $>$, _, ?, |, +, *, @. Any string used in BLIF-MV has to satisfy this constraint.
- *input-list* is a white-space separated list of strings (terminated by the end of the line) giving the formal input terminals for the model being declared. If this is the root model, then signals can be identified as the primary inputs of this system. *input-list* can be null, in which case the `.inputs` line may be removed altogether. Multiple `.inputs` lines are allowed, and the lists of inputs are concatenated.
- *output-list* is a white-space separated list of strings (terminated by the end of the line) giving the formal output terminals for the model being declared. If this is the root model, then signals can be identified as the primary output of this system. *output-list* can be null, in which case the `.outputs` line may be removed altogether. Multiple `.outputs` lines are allowed, and the lists of outputs are concatenated.
- Input variables and output variables have to be disjoint, i.e. a variable cannot be both an input and an output in a model.
- *command* is one of `.mv`, `.table`, `.latch`, `.reset` and `.subckt`, which defines the detailed functionality of the model. All the `.mv` declarations must precede the others in a model declaration.

1.2 Multi-valued Variables

A multi-valued variable is a variable that can take a finite number of values. There are two classes of multi-valued variables. The class of *enumerative variables* are variables whose domain is the n integers $\{0, \ldots, n-1\}$. Note that Boolean variables are enumerative variables where $n = 2$. Enumerative variables are declared as follows.

```
.mv <variable-name-list> <number-of-values>
```

- *variable-name-list* is a comma separated list of strings (terminated by the end of the line) giving the names of variables being declared.

- *number-of-values* is a natural number, which specifies the number of values n.
- Example: `.mv x,y 3`.

The second class, *symbolic variables*, is more general than the first one. A symbolic variable can take a set of arbitrary values. For example, a variable that takes three values red, green, and blue is a symbolic variable. Symbolic variables are declared as follows.

```
.mv <variable-name-list> <number-of-values> <value-list>
```

- *variable-name-list* and *number-of-values* are the same as for the declaration of enumerative variables.
- *value-list* is a white-space separated list of strings (terminated by the end of the line) giving the list of values the variable can take. The number of values declared and the range size should match.
- Example: `.mv x,y 3 red green blue`.

If a variable is not defined using `.mv` in a model, then the variable is assumed to be a Boolean variable.

Two variables are said to have the same type if

1. the variables are enumerative variables with the same domain size, or
2. the variables are symbolic variables with the same domain size and the same symbolic values defined in the same order in the `.mv` construct.

Consider the following example.

```
.mv x 2
.mv y 2 red blue
```

x and y are not of the same type because x is an enumerative variable and y is a symbolic variable although both are two-valued variables.

```
.mv x 2 red blue
.mv y 2 blue red
```

x and y in the above example are not of the same type because symbolic values are defined in different orders.

1.3 Tables

A table is an abstract representation of a physical gate. A table is driven by inputs and generates outputs following its functionality. Although a real gate generates an output deterministically depending on what inputs are supplied, tables in BLIF-MV can represent non-deterministic behaviors as well. The functionality of the table is described as a symbolic relation, i.e. the table enumerates symbolically all the valid combination of values among the inputs and the outputs. Note that BLIF-MV can handle multi-output tables, unlike BLIF, where every table is single-output. A table without input represents a constant generator. If the table allows more than one value for its output, then the table is a *nondeterministic* constant generator, which we call *pseudo input*. Tables are declared in the following way.

```
.table <in-1> <in-2> ... <in-n> -> <out-1> <out-2>... <out-m>
<relation>
...
<relation>
```

- *in-1*,...,*in-n* are strings giving the names of the inputs to the table being defined. The variables have to be defined using the `.mv` construct before the table. Otherwise, they are assumed to be Boolean variables.
- *out-1*,...,*out-m* are strings giving the names of the outputs to the table being defined. The variables have to be defined using the `.mv` construct before the table. Otherwise, they are assumed to be Boolean variables. Any table must have at least one output.
- If a table has a single output, -> is optional.

A *relation* is a white-space separated non-null list of $n + m$ strings, giving a valid combination of values among inputs and outputs. The i-th string in a relation specifies a set of values for the i-th variable in the input/output declaration of `.table`. Each relation denotes the Cartesian product of all the sets of values. The input-output relation of a table is defined as the union of all the relations. A set of values can be declared recursively in the following form.

1. a value v, or
2. $-$, which is the universe, or
3. a range $\{v_1 - v_2\}$, or
4. a list $(S_1, S_2, \ldots, S_l)$, where S_i $(i = 1, \ldots, l)$ is a set of values, or
5. $!S$, which is a complement of a set of values S.

Let x be an enumerative variable which takes 4 values. The following are examples of a set of values for x.

- 1
- $-$
- $\{2 - 3\}$

- $(0, \{2-3\})$
- $!\{2-3\}$

If a variable is a symbolic variable, the range construct in the above cannot be used since {red-green}, for example, does not make sense.

Let us consider the following example.

```
.mv x,y 4
.table x -> y
!2 {1-3}
- 0
2 (0,3)
```

The relation specified in this table is: $[(0,1,3) \times (1,2,3)] \cup [(0,1,2,3) \times (0)] \cup [(2) \times (0,3)]$.

1.3.1 = Construct

One can also use the = *construct* in table specifications. Assume that in the column corresponding to variable y, we have $= x$ as in the following example.

```
.table x -> y
- =x
```

The interpretation of this construct is that the value of y should be equal to x. This enables us to describe a multi-valued multiplexor compactly (see below).

```
.mv select 2
.mv data0,data1,output 256
.table select data0 data1 -> output
0 - - =data0
1 - - =data1
```

Note that two variables related with the = construct should be of the same type. Any variable referred to with the = construct must be an input of the table.

1.3.2 Default Output

It is sometimes convenient to define a default output for the input patterns not specified in a given relation. The `.default` construct is used for this purpose. In the following example, no relation is specified for the case where either $x1$ or $x2$ is 0. Since we have a default statement in the table, output 00 is related for those unspecified input patterns. Therefore, the relation of this table is: $[(1) \times (1) \times (1) \times (1)] \cup [(0) \times (0) \times (0) \times (0)] \cup [(0) \times (1) \times (0) \times (0)] \cup [(1) \times (0) \times (0) \times (0)]$. Each table can have at most one `.default` declaration.

```
.mv x1,x2,y1,y2 2
.table x1 x2 -> y1 y2
.default 0 0
1 1 1 1
```

The `.default` construct can be used even for tables without inputs. However, one has to be careful about the semantics. There are two possible cases. One case is that a table has a default declaration, but has no relation specified, where the interpretation of the table is that it always takes the default. The other case is that a table has both a default declaration and a non-null relation specified, then the default can be simply ignored.

1.4 Latches and Reset Tables

A latch is a storage element which updates its stored value at every clock tick. A latch has an input and an output. At each clock tick the latch output is set to the latch input value before the tick, and keep the value till the next clock tick. Every latch has to be initialized although the latch is allowed to have more than one initial value, in which case the latch takes an initial value nondeterministically from the specified values. A latch can be seen as a multi-valued flip-flop with possibly multiple initial states. In BLIF-MV, there is an implicit assumption that the whole system is clocked by a single global clock although the clock is never declared in BLIF-MV declarations.

A latch is declared as follows.

```
.latch <latch-input> <latch-output>
```

latch input and *latch output* are strings, giving the name of the latch input and the latch output. The two variables should be of the same type. A latch must have one reset table, which is used to initialize the latch output at the beginning. A reset table is a single output table whose only output is the output of a latch. Notice that we use `.reset` instead of `.table` for reset tables. If a latch is reset to a constant value, then the latch table has no input. The following example is for the latch *latch_output* whose reset state is 0.

```
.reset latch_output
0
```

One can specify multiple initial states by specifying more than one value in the latch output. Adding one more line to the above example, the latch has now two initial states.

```
.reset latch_output
0
1
```

This is one way to introduce non-determinism in system descriptions. Also, one could create complex reset circuitry sensitive to other variables by introducing inputs to the latch table. The following `.reset` statement initializes the latch to 1 if x is 0 and to 0 if x is 1.

```
.reset x latch_output
0 1
1 0
```

1.5 Subcircuits

In a model, another model can be instantiated as a subcircuit using the `.subckt` construct.

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

This construct instantiates a reference model *model-name* as an instance *instance-name* in the current model. *formal-actual-list* specifies the association between each formal variable in *model-name* and its corresponding actual variable in the current model. Formal variables are declared in the reference model, while actual variables are variables declared in the current model. *formal-actual-list* is a list of assignments separated by a white space. The declaration of *formal-actual-list* is of form:

```
formal-1 = actual-1 formal-2 = actual-2 ... formal-n = actual-n
```

The order of formal variables is unimportant.

1.6 Miscellaneous Features

1.6.1 Comments

Any line starting from # is a comment. It is ignored by the parser.

1.6.2 Including Files

The `.include` construct can be used to include another file from a file being read. The syntax is `.include fileName`.

1.7 Old Syntax

Previously, the three constructs `.table`, `.default`, `.reset` were called `.names`, `.def`, `.r` respectively. The `read_blif_mv` command in VIS supports these old constructs as well.

2 Semantics

In this section we describe the semantics of BLIF-MV. The semantics is defined over flattened networks where all the `.subckt` constructs are substituted recursively until leaf models. Leaf models are models without any `.subckt` declarations. In the following, a flattened network is called a *system*.

At every time point, the system is in some state, where each latch has a value. An initial state of the system is a state where every latch is set to an initial state declared using the `.reset` constructs. Notice that the system can have more than one initial state in general. At every clock tick, all the latches update their values. These values then propagate through tables until all the wires have a consistent set of values. If a latch is encountered during the propagation, i.e. an output of a table is an input of an latch, the propagation process is stopped. Note that because of nondeterminism, given a single state, there may be several consistent sets of values.

The semantics can be seen as a simple extension of the standard semantics of synchronous single-clocked digital circuits. In fact, if every table is deterministic and every latch has a single initial state, the two semantics are exactly equal. The only differences are in the interpretation of nondeterministic tables and latches with multiple initial states as described in the above.

3 The VIS-v Subset of BLIF-MV

VIS-v can only work on a strict subset of BLIF-MV although any synthesis-related commands like `read_blif` and `write_blif`, are applicable to the full-set of BLIF-MV. If the user generates BLIF-MV files using VL2MV following a certain restriction (See the VIS users' manual for details), the files are guaranteed to be in the subset. However, if BLIF-MV files are generated manually, the user must make sure that the files are in the VIS-v subset. Otherwise, `init_verify` simply fails, thereby making it impossible to perform the verification.

The restriction we pose is as follows.

- The only allowable nondeterministic tables are *pseudo-input tables*, which are no-input, single-output tables which generate more than one output nondeterministically.

Note that one can always transform any BLIF-MV file to its equivalent BLIF-MV file in the VIS-v subset by determinizing all intermediate nondeterministic tables by adding pseudo-inputs.