Berkeley Logic Interchange Format (BLIF)

University of California
Berkeley

September 6, 1996

The goal of BLIF is to describe a logic-level hierarchical circuit in textual form. A circuit is an arbitrary combinational or sequential network of logic functions. A circuit can be viewed as a directed graph of combinational logic nodes and sequential logic elements. Each node has a two-level, single-output logic function associated with it. Each feedback loop must contain at least one latch. Each net (or signal) has only a single driver, and either the signal or the gate which drives the signal can be named without ambiguity.

In the following, angle-brackets surround nonterminals, and square-brackets surround optional constructs.

1 Models

A model is a flattened hierarchical circuit. A BLIF file can contain many models and references to models described in other BLIF files. A model is declared as follows:

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

decl-model-name is a string giving the name of the model.

decl-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 first or only model, then these signals can be identified as the primary inputs of the circuit. Multiple *.inputs* lines are allowed, and the lists of inputs are concatenated.

decl-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 first or only model, then these signals can be identified as the primary outputs of the circuit. Multiple *.outputs* lines are allowed, and the lists of outputs are concatenated.

decl-clock-list is a white-space-separated list of strings (terminated by the end of the line) giving the clocks for the model being declared. Multiple *.clock* lines are allowed, and the lists of clocks are concatenated.

	`<logic-gate>`	`<generic-latch>`	`<library-gate>`
command is one of:	`<model-reference>`	`<subfile-reference>`	`<fsm-description>`
	`<clock-constraint>`	`<delay-constraint>`	

Each *command* is described in the following sections.

The BLIF parser allows the *.model, .inputs, .outputs, .clock* and *.end* statements to be optional. If *.model* is not specified, the *decl-model-name* is assigned the name of the BLIF file being read. It is an error to use the same string for *decl-model-name* in more than one model. If *.inputs* is not specified, it can be inferred from the signals which are not the outputs of any other logic block. Similarly, *.outputs* can be inferred from the signals which are not the inputs to any other blocks. If any *.inputs* or *.outputs* are given, no inference is made; a node that is not an output and does not fanout produces a warning message.

If *.clock* is not specified (e.g., for purely combinational circuits) there are no clocks. *.end* is implied at end of file or upon encountering another *.model*.

Important: the first model encountered in the main BLIF file is the one returned to the user. The only *.clock, clock-constraint,* and *timing-constraint* constructs retained are the ones in the first model. All subsequent models can be incorporated into the first model using the *model-reference* construct.

Anywhere in the file a '`#`' (hash) begins a comment that extends to the end of the current line. Note that the character '`#`' cannot be used in any signal names. A '\' (backslash) as the last character of a non-comment line indicates concatenation of the subsequent line to the current line. No whitespace should follow the '\'.

Example:

```
.model simple
.inputs a b
.outputs c
.names a b c        #    .names described later
11 1
.end


#   unnamed model
.names a b  \
c                   #    `\´ here only to demonstrate its use
11 1
```

Both models "simple" and the unnamed model describe the same circuit.

2 Logic Gates

A *logic-gate* associates a logic function with a signal in the model, which can be used as an input to other logic functions. A *logic-gate* is declared as follows:

```
.names <in-1> <in-2> ... <in-n> <output>
<single-output-cover>
```

output is a string giving the name of the gate being defined.

in-1, in-2, ... in-n are strings giving the names of the inputs to the logic gate being defined.

single-output-cover is, formally, an n-input, 1-output PLA description of the logic function corresponding to the logic gate. {0, 1, –} is used in the n-bit wide "input plane" and {0, 1} is used in the 1-bit wide "output plane". The ON-set is specified with 1's in the "output plane," and the OFF-set is specified with 0's in the "output plane." The DC-set is specified for primary output nodes only, by using the construct *.exdc*.

A sample *logic-gate* with its *single-output-cover*:

```
.names v3 v6 j u78 v13.15
1--0 1
-1-1 1
0-11 1
```

In a given row of the *single-output-cover*, "1" means the input is used in uncomplemented form, "0" means the input is complemented, and "–" means not used. Elements of a row are ANDed together, and then all rows are ORed.

As a result, if the last column (the "output plane") of the *single-output-cover* is all 1's, the first n columns (the "input plane") of the *single-output-cover* can be viewed as the truth table for the logic gate named by the string *output*. The order of the inputs in the *single-output-cover* is the same as the order of the strings *in-1, in-2, ..., in-n* in the *.names* line. A space between the columns of the "input plane" and the "output plane" is required.

The translation of the above sample *logic-gate* into a sum-of-products notation would be as follows:

```
v13.15 = (v3 u78´) + (v6 u78) + (v3´ j u78)
```

To assign the constant "0" to some logic gate `j`, use the following construct:

```
.names j
```

To assign the constant "1", use the following:

```
.names j
1
```

The string *output* can be used as the input to another *logic-gate* before the *logic-gate* for *output* is itself defined.

For a more complete description of the PLA input format, see *espresso(5)*.

3 External Don't Cares

External don't cares are specified as a separate network within a model, and are specified at the end of the model specification. Each external don't care function, which is specified by a *.names* construct, must be associated with a primary output of the main model and specified as a function of the primary inputs of the main model (hierarchical specification of external don't cares is currently not supported).

The external don't cares are specified as follows:

```
.exdc
.names <in-1> <in-2> ... <in-n> <output>
<single-output-cover>
```

ėxdc indicates that the following *.names* constructs apply to the external don't care network.

output is a string giving the name of the primary output for which the conditions are don't cares.

in-1, in-2, ... in-n are strings giving the names of the primary inputs which the don't care conditions are expressed in terms of.

single-output-cover is an n-input, 1-output PLA description of the logic function corresponding to the don't care conditions for the output.

The following is an example circuit with external don't cares:

```
.model a
.inputs x y
.outputs j
.subckt b x=x y=y j=j
.exdc
.names x j
1 1
.end

.model b
.inputs x y
.outputs j
.names x y j
11 1
.end
```

The translation of the above example into a sum-of-products notation would be as follows:

```
j = x * y;
external d.c. for j = x;
```

4 Flip flops and latches

A *generic-latch* is used to create a delay element in a model. It represents one bit of memory or state information. The *generic-latch* construct can be used to create any type of latch or flip-flop (see also the *library-gate* section). A *generic-latch* is declared as follows:

```
.latch <input> <output> [<type> <control>] [<init-val>]
```

input is the data input to the latch.

output is the output of the latch.

type is one of {fe, re, ah, al, as}, which correspond to "falling edge," "rising edge," "active high," "active low," or "asynchronous."

control is the clocking signal for the latch. It can be a *.clock* of the model, the output of any function in the model, or the word "NIL" for no clock.

init-val is the initial state of the latch, which can be one of {0, 1, 2, 3}. "2" stands for "don't care" and "3" is "unknown." Unspecified, it is assumed "3."

If a latch does not have a controlling clock specified, it is assumed that it is actually controlled by a single global clock. The behavior of this global clock may be interpreted differently by the various algorithms that may manipulate the model after the model has been read in. Therefore, the user should be aware of these varying interpretations if latches are specified with no controlling clocks.

Important: All feedback loops in a model must go through a *generic-latch*. Purely combinational-logic cycles are not allowed.

Examples:

```
.inputs d              # a clocked flip-flop
.output q
.clock c
.latch d q re c 0
.end

.inputs in             # a very simple sequential circuit
.outputs out
```

```
.latch out in 0
.names in out
0 1
.end
```

5 Library Gates

A *library-gate* creates an instance of a technology-dependent logic gate and associates it with a node that represents the output of the logic gate. The logic function of the gate and its known technology dependent delays, drives, etc. are stored with the *library-gate*. A *library-gate* is one of the following:

```
.gate <name> <formal-actual-list>
.mlatch <name> <formal-actual-list> <control> [<init-val>]
```

name is the name of the *.gate* or *.mlatch* to instantiate. A gate or latch with this name must be present in the current working library.

formal-actual-list is a mapping between the formal parameters of *name* (the terminals of the *library-gate*) and the actual parameters of the current model (any signals in this model). The format for a *formal-actual-list* is a white-space-separated sequence of assignment statements of the form:

```
formal1=actual1 formal2=actual2 ...
```

All of the formal parameters of *name* must be specified in the *formal-actual-list* and the single output of *name* must be the last one in the list.

control is the clocking signal for the mlatch, which can be either a *.clock* of the model, the output of any function in the model, or the word “NIL” for no clock.

init-val is the initial state of the mlatch, which can be one of {0, 1, 2, 3}. “2” stands for “don't care” and “3” is “unknown.” Unspecified, it is assumed “3.”

A *.gate* refers to a two-level representation of an arbitrary input, single output gate in a library. A *.gate* appears under a technology-independent interpretation as if it were a single *logic-gate*.

A *.mlatch* refers to a latch (not necessarily a D flip flop) in a library. A *.mlatch* appears under a technology-independent interpretation as if it were a single *generic-latch* and possibly a single *logic-gate* feeding the data input of that *generic-latch*.

*.gate*s and *.mlatch*es are used to describe circuits that have been implemented using a specific library of standard logic functions and their technology-dependent properties. The library of *library-gate*s must be read in before a BLIF file containing *.gate* or *.mlatch* constructs is read in.

The string *name* refers to a particular gate or latch in the library. The names "nand2," "inv," and "jk_rising_edge" in the following examples are descriptive names for gates in the library. The following BLIF description:

```
.inputs v1 v2
.outputs j
.gate nand2 A=v1 B=v2 O=x  # given: formals of this gate are A, B, O
.gate inv A=x O=j          # given: formals of this gate are A & O
.end
```

could also be specified in a technology-independent way (assuming "nand2" is a 2-input NAND gate and "inv" is an INVERTER) as follows:

```
.inputs v1 v2
.outputs j
.names v1 v2 x
0- 1
-0 1
.names x j
0 1
.end
```

Similarly:

```
.inputs j kbar
.outputs out
.clock clk
.mlatch jk_rising_edge J=j K=k Q=q clk 1   # given: formals are J, K, Q
.names q out
0 1
.names kbar k
0 1
.end
```

could have been specified in a technology-independent way (assuming “jk_rising_edge” is a JK rising-edge-triggered flip flop) as follows:

```
.inputs j kbar
.outputs out
.clock clk
.latch temp q re clk 1      # the .latch
.names j k q temp           # the .names feeding the D input of the .latch
-01 1
1-0 1
.names q out
0 1
.names kbar k
0 1
.end
```

6 Model (subcircuit) references

A *model-reference* is used to insert the logic functions of one model into the body of another. It is defined as follows:

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

model-name is a string giving the name of the model being inserted. It need not be previously defined in this file, but should be defined somewhere in either this file, a *.search* file, or a master file that is *.search*ing this file. (see *.search* below)

formal-actual-list is a mapping between the formal terminals (the *decl-input-list, decl-output-list,* and *decl-clock-list*) of the called model *model-name* and the actual parameters of the current model. The actual parameters may be any signals in the current model. The format for a *formal-actual-list* is the same as its format in a *library-gate.*

A *.subckt* construct can be viewed as creating a copy of the logic functions of the called model *model-name*, including all of *model-name*'s *generic-latch*es, in the calling model. The hierarchical nature of the BLIF description of the model does not have to be preserved. Subcircuits can be nested, but cannot be self-referential or create a cyclic dependency.

Unlike a *library-gate*, a *model-reference* is not limited to one output.

The formals need not be specified in the same order as they are defined in the *decl-input-list, decl-output-list,* or *decl-clock-list*; elements of the lists can be intermingled in any order, provided the names are given correctly. Warning messages are printed if elements of the *decl-input-list* or *decl-clock-list* are not driven by an actual parameter or if elements of the *decl-output-list* do not fan out to an actual parameter. Elements of the *decl-clock-list* and *decl-input-list* may be driven by any logic function of the calling model.

Example: rather than rewriting the entire BLIF description for a commonly used subcircuit several times, the subcircuit can be described once and called as many times as necessary:

```
.model 4bitadder
.inputs A3 A2 A1 A0 B3 B2 B1 B0 CIN
.outputs COUT S3 S2 S1 S0
.subckt fulladder a=A0 b=B0 cin=CIN     s=S0 cout=CARRY1
.subckt fulladder a=A3 b=B3 cin=CARRY3  s=S3 cout=COUT
.subckt fulladder b=B1 a=A1 cin=CARRY1  s=XX cout=CARRY2
.subckt fulladder a=JJ b=B2 cin=CARRY2  s=S2 cout=CARRY3
                  # for the sake of example,
.names XX S1      # formal output `s´ does not fanout to a primary output
1 1
.names A2 JJ      # formal input `a´ does not fanin from a primary input
1 1
.end

.model fulladder
.inputs a b cin
.outputs s cout
.names a b k
10 1
01 1
.names k cin s
10 1
01 1
.names a b cin cout
11- 1
1-1 1
-11 1
.end
```

7 Subfile References

A *subfile-reference* is:

```
.search <file-name>
```

file-name gives the name of the file to search.

A *subfile-reference* directs the BLIF reader to read in and define all the models in file *file-name*. A *subfile-reference* does not have to be inside of a *.model*. *subfile-reference*s can be nested.

Search files would usually be used to hold all the subcircuits referred to in *model-reference*s, while the master file merely searches all the subfiles and instantiates all the subcircuits it needs.

A *subfile-reference* is not equivalent to including the body of subfile *file-name* in the current file. It does not patch fragments of BLIF into the current file; it pauses reading the current file, reads *file-name* as an independent, self-contained file, then returns to reading the current file.

The first *.model* in the master file is always the one returned to the user, regardless of any *subfile-reference*s than may precede it.

8 Finite State Machine Descriptions

A sequential circuit can be specified in BLIF logic form, as a finite state machine, or both. An *fsm-description* is used to insert a finite state machine description of the current model. It is intended to represent the same sequential circuit as the current model (which contains logic), but in FSM form. The format of an *fsm-description* is:

```
.start_kiss
.i <num-inputs>
.o <num-outputs>
[.p <num-terms>]
[.s <num-states>]
[.r <reset-state>]
<input> <current-state> <next-state> <output>
          .
          .
          .
<input> <current-state> <next-state> <output>
```

```
.end_kiss
[.latch_order <latch-order-list>]
[<code-mapping>]
```

num-inputs is the number of inputs to the FSM, which should agree with the number of inputs in the *.inputs* construct for the current model.

num-outputs is the number of outputs of the FSM, which should agree with the number of outputs in the *.outputs* construct for the current model.

num-terms is the number of "<input> <current-state> <next-state> <output>" 4-tuples that follow in the FSM description.

num-states is the number of distinct states that appear in "<current-state>" and "<next-state>" columns.

reset-state is the symbolic name for the reset state for the FSM; it should appear somewhere in the "<current-state>" column.

input is a sequence of *num-inputs* members of $\{0, 1, -\}$.

output is a sequence of *num-outputs* members of $\{0, 1, -\}$.

current-state and *next-state* are symbolic names for the current state and next state transitions of the FSM.

latch-order-list is a white-space-separated sequence of latch outputs.

code-mapping is newline separated sequence of:

```
.code <symbolic-name> <encoded-name>
```

num-terms and *num-states* do not have to be specified. If the *reset-state* is not given, it is assigned to be the first state encountered in the "<current-state>" column.

The ordering of the bits in the *input* and *output* fields will be the same as the ordering of the variables in the *.inputs* and *.outputs* constructs if both an *fsm-description* and logic functions are given.

latch-order-list and *code-mapping* are meant to be used when both an *fsm-description* and a logical description of the model are given. The two constructs together provide a correspondence between the latches in the logical description and the state variables in the *fsm-description*. In a *code-mapping*, *symbolic-name* consists of a symbolic name from the "<current-state>" or "<next-state>" columns, and *encoded-name* is the pattern of bits ($\{0, 1\}$) that represent the state encoding for *symbolic-name*. The *code-mapping* should only be given if both an *fsm-description* and logic functions are given. *.latch-order* establishes a mapping between the bits of the *encoded-name*s of the *code-mapping* construct and the latches of the network. The order of the bits in the encoded names will be the same as the order of the latch outputs in the *latch-order-list*. There should be the same number of bits in the *encoded-name* as there are latches if both an *fsm-description* and a logical description are specified.

If both *logic-gate*s and an *fsm-description* of the model are given, the *logic-gate* description of the model should be consistent with the *fsm-description*, that is, they should describe the same circuit. If they are not consistent there will be no sensible way to interpret the model, which should then cause an error to be returned.

If only the *fsm-description* of the network is given, it may be run through a state assignment routine and given a logic implementation. A sole *fsm-description*, having no logic implementation, cannot be inserted into another model by a *model-reference*; the state assigned network, or a network containing both *logic-gate*s and an *fsm-description* can.

Example of an *fsm-description*:

```
.model 101          # outputs 1 whenever last 3 inputs were 1, 0, 1
.start_kiss
.i 1
.o 1
0 st0 st0 0
1 st0 st1 0
0 st1 st2 0
1 st1 st1 0
0 st2 st0 0
1 st2 st3 1
0 st3 st2 0
1 st3 st1 0
.end_kiss
.end
```

Above example with a consistent *fsm-description* and logical description:

```
.model
```

```
.inputs v0
.outputs v3.2
.latch    [6] v1   0
.latch    [7] v2   0
.start_kiss
.i 1
.o 1
.p 8
.s 4
.r st0
0 st0 st0 0
1 st0 st1 0
0 st1 st2 0
1 st1 st1 0
0 st2 st0 0
1 st2 st3 1
0 st3 st2 0
1 st3 st1 0
.end_kiss
.latch_order v1 v2
.code st0 00
.code st1 11
.code st2 01
.code st3 10
.names v0 [6]
1 1
.names v0 v1 v2 [7]
-1- 1
1-0 1
.names v0 v1 v2 v3.2
101 1
.end
```

9 Clock Constraints

A *clock-constraint* is used to set up the behavior of the simulated clocks, and to specify how clock events (rising or falling edges) occur relative to one another. A *clock-constraint* is one or more of the following:

```
.cycle <cycle-time>
.clock_event <event-percent> <event-1> [<event-2> ... <event-n>]
```

cycle-time is a floating point number giving the clock cycle time for the model. It is a unitless number that is to be interpreted by the user.

event-percent is a floating point number representing a percentage of the clock cycle time at which a specific *.clock_event* occurs. Fifty percent is written as "50.0."

event-1 through *event-n* are one of the following:

```
<rise-fall>´<clock-name>
(<rise-fall>´<clock-name> <before> <after>)
```

where *rise-fall* is either "r" or "f" and stands for the rising or falling edge of the clock and *clock-name* is a clock from the *.clock* construct. The apostrophe between *rise-fall* and *clock-name* is a seperator, and serves no purpose in and of itself.

before and *after* are floating point numbers in the same "units" as the *cycle-time* and are used to define the "skew" in the clock edges. *before* represents maximum amount of time before the nominal time that the edge can arrive; *after* represents the maximum amount of time after the nominal time that the edge can arrive. The nominal time is *event-percent*% of the *cycle-time*. In the unparenthesized form for the *clock-event*, *before* and *after* are assumed "0.0."

All events, *event-1 ... event-n*, specified in a single *.clock_event* are to be linked together. A routine changing any one edge should also modify the occurrence time of all the related clock edges.

Example 1:

```
.clock clock1 clock2
.clock_event 50.0 r´clock1 (f´clock2 2.0 5.0)
```

Example 2:

```
.clock clock1 clock2
.clock_event 50.0 r´clock1
.clock_event 50.0 (f´clock2 2.0 5.0)
```

Both examples specify a nominal time of 50% of the cycle time, that the rising edge of clock1 must occur at exactly the nominal time, and that the falling edge of clock2 may occur from 2.0 units before to 5.0 units after the nominal time.

In Example 1, if r'clock1 is later moved to a different nominal time by some routine then f'clock2 should also be changed. However, in Example 2 changing r'clock1 would not affect f'clock2 even though they originally have the same value of *event-percent*.

10 Delay Constraints

A *delay-constraint* is used to specify parameters to more accurately compute the amount of time signals take to propagate from one point to another in a model. A *delay-constraint* is one or more of :

```
.area <area>
.delay <in-name> <phase> <load> <max-load> <brise> <drise> <bfall> <dfall>
.wire_load_slope <load>
.wire <wire-load-list>
.input_arrival <in-name> <rise> <fall> [<before-after> <event>]
.default_input_arrival <rise> <fall>
.output_required <out-name> <rise> <fall> [<before-after> <event>]
.default_output_required <rise> <fall>
.input_drive <in-name> <rise> <fall>
.default_input_drive <rise> <fall>
.max_input_load <load>
.default_max_input_load <load>
.output_load <out-name> <load>
.default_output_load <load>
```

rise, *fall*, *drive*, and *load* are all floating point numbers giving the rise time, fall time, input drive, and output load.

in-name is a primary input and *out-name* is a primary output.

before-after can be one of {b, a}, corresponding to "before" or "after," and *event* has the same format as the unparenthesized form of *event-1* in a *clock-constraint*.

.area sets the area of the model to be *area*.

.delay sets the delay for input *in-name*. *phase* is one of "INV," "NONINV," or "UNKNOWN" for inverting, non-inverting, or neither. *max-load* is a floating point number for the maximum load. *brise, drise, bfall,* and *dfall* are floating point numbers giving the block rise, drive rise, block fall, and drive fall for *in-name*.

.wire_load_slope sets the wire load slope for the model.

.wire sets the wire loads for the model from the list of floating point numbers in the *wire-load-list*.

.input_arrival sets the input arrival time for the input *in-name*. If the optional arguments are specified, then the input arrival time is relative to the *event*.

.output_required sets the output required time for the output *out-name*. If the optional arguments are specified, then the output required time is relative to the *event*.

.input_drive sets the input drive for the input *in-name*.

.max_input_load sets the maximum load that the input *in-name* can handle.

.output_load sets the output load for the output *out-name*.

.default_input_arrival, .default_output_required, .default_input_drive, .default_output_load set the corresponding default values for all the inputs/outputs whose values are not specifically set.

There is no actual unit for all the timing and load numbers. Special attention should be given when specifying and interpreting the values. The timing numbers are assumed to be in the same "unit" as the *cycle-time* in the *.cycle* construct.