Logic Synthesis and Optimization Benchmarks User Guide
Version 3.0

Saeyang Yang[1]

This report is issued to provide documentation for the benchmark examples used in conjunction with the 1991 MCNC International Workshop on Logic Synthesis and the extention of the 1989 Logic Synthesis and Optimization Benchmarks User Guide. Its distribution is limited to peer communication and to participants of the workshop.

For information about the ideas expressed herein, contact the author(s) directly. For information about the MCNC Technical Report Series, or Industrial Affiliates Program, contact Corporate Communications, MCNC, P.O. Box 12889, Research Triangle Park, NC 27709; (919) 248-1842.

January 15, 1991

---

[1]Microelectronics Center of North Carolina, P. O. Box 12889, Research Triangle Park, NC 27709

Contents

1 Acknowledgements

The logic synthesis and optimization benchmarks distributed by the Microelectronics Center of North Carolina include the ISCAS'85 and ISCAS'89 set. Additional benchmarks have been provided by the following individuals and organizations:

Nico Benschop
Phillips Research Laboratories
Eindhoven, The Netherlands

Giovanni De Micheli
Stanford University
Department of Electrical Engineering
Palo Alto, CA

Petra Michel
Siemens AG
Dept. ZFE F2 DES1
Munchen, Germany

OCT Tools Distribution
Electronics Research Laboratory
University of California, Berkeley
Berkeley, CA

Ellen Sentovich
Electronics Research Laboratory
University of California, Berkeley
Berkeley, CA

Fabio Somenzi
University of Colorado
Department of Electrical & Computer Engineering
Boulder, CO

Louise Trevillyan
IBM Corporation
0B4/4A17, Route 100
Somers, NY

This user guide extends the version 2.0 prepared by Robert Lisanke in December 1988. ACM/SIGDA funded Doug Maltais, a graduate student at NCSU, to assist with preparing data for version 3.0. Funding for distribution of the benchmarks for the 1991 MCNC International Workshop on Logic Synthesis has been provided by ACM/SIGDA.

2 Benchmark Distribution

The benchmark examples for the 1991 International Workshop on Logic Synthesis can be obtained from MCNC in two ways.

1. If you have access to an ARPA Net connection you may establish an FTP connection to host "mcnc.mcnc.org" and copy the files using the "get" command. The login name is "anonymous" and the password is any string (we ask that you use your name or home login id). The files are in the "/pub/benchmark/LGSynth91" directory.

   Check the FTP man page for details. The following command sequence will obtain all the benchmark data contained in subdirectories fsmexamples (FSMs in KISS2 format), smlexamples (sequential multi-level "extended blif" examples), cmlexamples (combinational multi-level "blif" examples), twolexamples (two-level examples in ESPRESSO format), wkslibrary (the workshop libraries to be used for synthesis), and wksdoc (this document in LaTeX form).

   The command sequence to establish a connection to the host machine is:

   ```
   ftp mcnc.mcnc.org
   anonymous  (in response to the "Name" prompt)
   (your id)  (in response to the "password" prompt)
   cd pub/benchmark/LGSynth91
   ```

   If you have a UNIX(tm) machine on the Arpanet, check to see if you have the "uncompress" command available. Shipping the benchmarks over in compressed form is much faster than in plaintext. If you have uncompress, then do

   ```
   binary
   get LGSynth91.tar.Z
   bye
   ```

   on your local machine, uncompress the file with the command

   ```
   uncompress LGSynth91.tar.Z
   ```

   un-tar the file with the command

   ```
   tar -xvf LGSynth91.tar
   ```

   If you do not have the uncompress utility, but do have the "tar" facility then take the uncompressed version, as in

   ```
   binary
   get LGSynth91.tar
   bye
   ```

   and then on your local machine

   ```
   tar -xvf LGSynth91.tar
   ```

If you do not have the "tar" utility available, you may have to copy the files explicitly.

```
binary
cd fsmexamples
mget *
cd ../smlexamples
mget *
cd ../cmlexamples
mget *
cd ../twolexamples
mget *
cd ../wkslibrary
mget *
cd ../wksdoc
mget *
bye
```

2. We are prepared to send you a 9-track tape in TAR format. Contact Jeri Williams by phone (919 248-1938) or email (benchmarks@mcnc.org) and tell her your requirements. The cost of materials and handling are being covered by a grant from the Association of Computing Machinery, Special Interest Group on Design Automation (ACM/SIGDA). You are free to copy and distribute material further to your colleagues as long as you don't charge for these services.

Benchmarks are updated/corrected periodically and you may ftp to "mcnc.mcnc.org" and cd to /pub/benchmark/LGSynth91/bench.update for an update. If you have technical questions about the benchmarks, try to contact the benchmark originators whenever possible. If you want to share your results and comments, send e-mail to benchmarks@mcnc.org and we will try to post them. For more detailed information about these benchmarks, you may contact Franc Brglez, Workshop Chair, (919) 248-1925; email brglez@mcnc.org or Saeyang Yang at (919) 248-1886; e-mail syang@mcnc.org.

3 Benchmark Examples Included in v3.0 Release

The logic synthesis and optimization benchmark set consists of examples from four broad categories.

1. Finite-state tables in KISS2 format.
2. Sequential Multi-level logic in extended BLIF(Berkeley Logic Interchange Format) or SLIF(Structure Logic Interchange Format).
3. Combinational Multi-level logic in BLIF or SLIF.
4. Two-level logic in Berkeley PLA (ESPRESSO) format or SLIF.

The following sections present tables for each benchmark category. The tables list the names of examples included in the v3.0 benchmark set along with some characteristics for each benchmark.

3.1 FSM Examples

FSM Name	Inputs	Outputs	Products	States
bbara	4	2	60	10
bbsse	7	7	56	16
bbtas	2	2	24	6
beecount	3	4	28	7
cse	7	7	91	16
dk14	3	5	56	7
dk15	3	5	32	4
dk16	2	3	108	27
dk17	2	3	32	8
dk27	1	2	14	7
dk512	1	3	30	15
donfile	2	1	96	24
ex1	9	19	138	20
ex2	2	2	72	19
ex3	2	2	36	10
ex4	6	9	21	14
ex5	2	2	32	9
ex6	5	8	34	8
ex7	2	2	36	10
keyb	7	2	170	19
kirkman	12	6	370	16
lion	2	1	11	4
lion9	2	1	25	9
mark1	5	16	22	15
mc	3	5	10	4
modulo12	1	1	24	12
opus	5	6	22	10
planet	7	19	115	48
planet1	7	19	115	48
pma	8	8	73	24
s1	8	6	107	20
s1a	8	6	107	20
s8	4	1	20	5
s820	18	19	232	25
s1494	8	19	250	48
s208	11	2	153	18
s27	4	1	34	6
s420	19	2	137	18
s832	18	19	245	25
s1488	8	19	251	48
s510	19	7	77	47
s386	7	7	64	13
s298	3	6	1096	218
sand	11	9	184	32

FSM Name	Inputs	Outputs	Products	States
scf	27	56	166	121
shiftreg	1	1	16	8
sse	7	7	56	16
styr	9	10	166	30
tav	4	4	49	4
tbk	6	3	1569	32
tma	7	6	44	20
train11	2	1	25	11
train4	2	1	14	4

3.2 Sequential Multi-Level Examples

Circuit Name	Circuit Function	Inputs	Outputs	Latches	Approx. Gates
bigkey	Key Encription	262	197	221	4765
clma	Bus Interface	382	82	33	~35000
clmb	Bus Interface	382	0	33	~35000
dsip	Encription Circuit	228	197	224	2097
mm30a	Minmax Circuit	33	30	90	1549
mm4a	Minmax Circuit	7	4	12	153
mm9a	Minmax Circuit	12	9	27	492
mm9b	Minmax Circuit	12	9	26	538
mult16a	Multiplier	17	1	16	208
mult16b	Multiplier	17	1	30	212
mult32a	Multiplier	33	1	32	416
mult32b	Multiplier	32	1	62	436
sbc	Snooping Bus Controller	40	56	28	645
s27	Logic	4	1	3	10
s208.1 *	Digital Fractional Multiplier	10	1	8	104
s298	PLD	3	6	14	119
s344	4-bit Multiplier	9	11	15	160
s349	4-bit Multiplier	9	11	15	161
s382	Traffic Light Controller	3	6	21	158
s386	Controller	7	7	6	159
s400	Traffic Light Controller	3	6	21	162
s420.1 *	Digital Fractional Multiplier	18	1	16	218
s444	Traffic Light Controller	3	6	21	181
s510	Controller	19	7	6	211
s526n	Traffic Light Controller	3	6	21	194
s526	Traffic Light Controller	3	6	21	193
s641	PLD	35	24	19	379
s713	PLD	35	23	19	393
s820	PLD	18	19	5	289
s832	PLD	18	19	5	446
s838.1 *	Digital Fractional Multiplier	34	1	32	288
s1196	Logic	14	14	18	529
s1423	Logic	17	5	74	657
s1488	Controller	8	19	6	653
s1494	Controller	8	19	6	647
s5378	Logic	35	49	179	2779
s38417	Logic	28	106	1636	22179
s9234.1 *	Logic	36	39	211	5597
s13207.1 *	Logic	62	152	638	7951
s15850.1 *	Logic	77	150	534	9772
s38584.1 *	Logic	38	304	1426	19253

* These examples corrected with respected to original versions. See /bench.update for details.

3.3 Combinational Multi-Level Examples

Circuit Name	Circuit Function	Inputs	Outputs	Approx. Gates
9symml	Count Ones	9	1	43
C1355	Error Correcting	41	32	546
C17	Logic	5	2	6
C1908	Error Correcting	33	25	880
C2670	ALU and Control	233	140	1193
C3540	ALU and Control	50	22	1669
C432	Priority Decoder	36	7	160
C499	Error Correcting	41	32	202
C5315	ALU and Selector	178	123	2307
C6288	16-bit Multiplier	32	32	2406
C7552	ALU and Control	207	108	3512
C880	ALU and Control	60	26	383
alu2	ALU	10	6	335
alu4	ALU	14	8	681
apex6	Logic	135	99	452
apex7	Logic	49	37	176
b1	Logic	3	4	13
b9	Logic	41	21	125
c8	Logic	28	18	164
cc	Logic	21	20	47
cht	Logic	47	36	229
cm138a	Logic	6	8	17
cm150a	Logic	21	1	69
cm151a	Logic	12	2	33
cm162a	Logic	14	5	43
cm163a	Logic	16	5	42
cm42a	Logic	4	10	17
cm82a	Logic	5	3	27
cm85a	Logic	11	3	38
cmb	Logic	16	4	41
comp	Logic	32	3	151
cordic	Logic	23	2	102
count	Counter	35	16	143
cu	Logic	14	11	48
dalu	Dedicated ALU	75	16	1697
decod	Decoder	5	16	22

Circuit Name	Circuit Function	Inputs	Outputs	Approx. Gates
des	Data Encription	256	245	~4000
example2	Logic	85	66	277
f51ml	Arithmetic	8	8	43
frg1	Logic	28	3	105
frg2	Logic	143	139	1004
i1	Logic	25	16	46
i10	Logic	257	224	2260
i2	Logic	201	1	109
i3	Logic	132	6	90
i4	Logic	192	6	120
i5	Logic	133	66	285
i6	Logic	138	67	340
i7	Logic	199	67	471
i8	Logic	133	81	1831
i9	Logic	88	63	522
k2	Logic	45	45	1201
lal	Logic	26	19	114
majority	Voter	5	1	9
mux	Mux	21	1	91
my_adder	Adder	33	17	223
pair	Logic	173	137	1434
parity	Parity	16	1	68
pcle	Logic	19	9	68
pcler8	Logic	27	17	84
pm1	Logic	16	13	39
rot	Logic	135	107	691
sct	Logic	19	15	91
t481	Logic	16	1	2072
tcon	Logic	17	16	41
term1	Logic	34	10	358
too_large	Logic	38	3	578
ttt2	Logic	24	21	200
unreg	Logic	36	16	97
vda	Logic	17	39	585
x1	Logic	51	35	285
x2	Logic	10	7	42
x3	Logic	135	99	715
x4	Logic	94	71	369
z4ml	2-bit Add	7	4	20

3.4 Two-Level Examples

Circuit Name	Inputs	Outputs	Product Terms
5xp1	7	10	75
9sym	9	1	87
apex1	45	45	206
apex2	39	3	1035
apex3	54	50	280
apex4	9	19	438
apex5	117	88	1227
b12	15	9	431
bw	5	28	87
clip	9	5	167
con1	7	2	9
cordic	23	2	1206
cps	24	109	654
duke2	22	29	87
e64	65	65	65
ex4	128	28	620
ex5	8	63	256
ex1010	10	10	810
inc	7	9	34
misex1	8	7	32
misex2	25	18	29
misex3	14	14	1848
misex3c	14	14	305
o64	130	1	65
pdc	16	40	2406
rd53	5	3	32
rd73	7	3	141
rd84	8	4	256
sao2	10	4	58
seq	41	35	1459
spla	16	46	2296
t481	16	1	481
vg2	25	8	110
xor5	5	1	16
Z5xp1	7	10	128
Z9sym	9	1	420

4 Data Formats for Benchmark Examples

Each benchmark category has its own data representation or format. FSM examples use KISS2 format. The sequential multi-level examples use an extended BLIF or a SLIF, and combinational multi-level examples use a BLIF or SLIF. All two-level examples use the ESPRESSO format or SLIF.

4.1 FSM Format

FSM benchmarks are distributed in KISS2 format. To improve uniformity, FSM benchmarks received have been modified to adhere to the following conventions:

1. KISS2 headers include the following information:

```
.i  n    # number of inputs
.o  m    # number of outputs
.p  p    # number of products
.s  s    # number of states used
.r  r    # reset state
```

2. KISS2 format don't care states are '*' . A current-state don't-care condition indicates that no matter what state you are in, a specified input produces a transition to a given next state and output condition. For example,

```
  .     .    .      .
  .     .    .      .
 1---   *  RESET  0000 # go to state RESET if bit 0 is true,
                       # regardless of the current state
```

   Don't-care next states are usually indicated if an input and current-state condition can not occur. In this case, outputs would also be don't care (indicated by - for each output).

3. Unused states are assumed by default in KISS2 to have don't-care next states and outputs. If only k out of 2**n states are used, then the (2**n - k) unused states have the following interpretation:

```
----  unused_1        *   ----   # don't care about next or out
----  unused_2        *   ----   # will never occur
  .      .            .    .
  .      .            .    .
  .      .            .    .
----  unused_2**n-k   *   ----
```

4.1.1 Examples: dk27.kiss2

dk27.kiss2

```
.i 1
.o 2
.p 14
.s 7
.r START
0 START state6 00
0 state2 state5 00
0 state3 state5 00
0 state4 state6 00
0 state5 START 10
0 state6 START 01
0 state7 state5 00
1 state6 state2 01
1 state5 state2 10
1 state4 state6 10
1 state7 state6 10
1 START state4 00
1 state2 state3 00
1 state3 state7 00
```

4.2 Multi-Level Formats

4.2.1 Introduction

The multi-level sequential or combinational logic is represented in two different formats, (extended) BLIF or SLIF. The BLIF and SLIF may be read by the Berkeley MISII/SIS program. Also, SLIF can be read by the Stanford synthesis tool, OLYMPUS.

4.2.2 BLIF Multi-Level Format Description

Berkeley Logic Interchange Format (BLIF)
University of California
Berkeley

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.

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.

command is one of:

`<logic-gate>`	`<generic-latch>`	`<library-gate>`
`<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. 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.

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," the OFF-set is specified with 0's in the "output plane," and the DC-set is specified with –'s in the "output plane." The logic gate can have either its ON-set and DC-set specified, or its OFF-set and DC-set specified.

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)*.

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;
```

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
```

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
```

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
```

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.

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
```

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*.

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>
.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*.

.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.

4.2.3 SLIF Description

Structure Logic Interchange Format (SLIF)
Stanford University

Description: SLIF is a concise format used to describe a structural view of logic circuits and their interconnections. It is an hierarchical, non-procedural notation that is described in ASCII files.

Syntax: SLIF is a free-format notation; i.e., statements may begin at any point on a line, and whitespace may be used freely. Each statement must be terminated by a semicolon. Statements may appear in any order within the description of a model, with the restriction that inputs, outputs, inouts and types must be declared before they are used and that the last statement in the model description must be the .endmodel statement (see the Commands section below for more details).

Identifiers are character strings restricted to alphanumeric characters and the following symbols:

: ^ % [ ] _ . / -

Variables, model names and instance names are all identifiers. There are two special variables, "1" and "0", which represent the logic values TRUE and FALSE, respectively.

Commands in SLIF are command words preceded by a period (e.g., .library). and are summarized in the next section. Any declaration that does not begin with a command is a logic statement and has the form

$$var = expression;$$

where var is an identifier and expression is an expression in Boolean form, consisting of variables and operators. The operators +, * and ′ represent Boolean addition, multiplication and inversion (i.e., AND, OR and NOT), respectively; the '*' operator is optional and may be omitted. e.g.,

$$out = reset' + clock * (in0' + (in1 * in2));$$

is equivalent to

$$out = reset' + clock(in0' + in1in2);$$

An expression, like a literal, may be complemented using the prime (i.e., apostrophe) symbol; e.g.,

$$x = (a(b + c)' + d)';$$

By default, the expression represents the ON SET of the variable var. Two symbols, $'$ and ~ are appended to var to indicate the expression is its OFF SET or DON'T CARE SET respectively. The ~ can also be used in the expression to indicate the DON'T CARE SET of a variable. Used alone, ~ means the global DON'T CARE SET of the surrounding model.

There are two built-in functions. The arguments of these functions must be variables (not expressions). The built-in functions are:

D(a,c) A flow-through generic D-type latch, which has input a and is clocked by c.

T(a,b) A three-state gate whose output is a when b is true, or high-impedance otherwise.

The use of a built-in function is indicated by the '@' symbol; e.g.,

$$out1 = @D(sig1, clock');$$

In addition to built-in functions, library functions may be called; these are defined as a separate model (see .library below).

Comments are identified by the symbol '#'. This symbol indicates that the remainder of the line is to be ignored by any program reading the SLIF description.

Commands:

.attribute type_name variable_name parameters ; Specifies parameters for one variable (or one instance), named variable_name. The parameters consist of a sequence of strings, integers and floats, defined in the type type_name. If the type used allows for a variable number of parameters, the corresponding list has to be enclosed in parentheses "(" and ")".

.call instance_name model_name (inputs ; inouts ; outputs) ; Creates an instance instance_name of the SLIF model model_name, which may be described in the same file or in a file specified by a .search statement. The called model may be a library element. Variables are linked according to the parameter listing; inputs, inouts and outputs are lists of variables separated by commas, which must agree in number and order with those in the called model.

.date time_stamp ; Specify the time of the last modification (optional). The time_stamp format is YYMMDDHHmmSS where YY is the year, MM the month, DD the day, HH the hour, mm the minutes, and SS the seconds. Each element of the time_stamp is a two-digit number.

.endmodel name ; Terminates the model. Each model has to be terminated by this declaration. There may be more than one model within the same file.

.global_attribute type_name parameters ; Specifies parameters valid for an entire model.

.include file_name ; Indicates that the information in file_name will be read as if it was part of the current file.

.inouts var1 var2 ... varn ; Declares variables var1 ... varn as primary bidirectional "inouts."

.inputs var1 var2 ... varn ; Declares variables var1 ... varn as primary inputs.

.library ; Identifies the model as a library element.

.model name ; Indicates the beginning of a new model and assigns it name name. Each model has to be declared using this declaration. Multiple models may be described in a single file.

.net var1 var2 ... varn ; Lists variables that are connected together. The net will be named after one of the variables. If there are primary inputs, outputs or inouts then the net will be inherit one of their names; otherwise it will be named after var1.

.outputs var1 var2 ... varn ; Declares variables var1 ... varn as primary outputs.

.search file_name ; Indicates that models included in file_name may be used, if they are needed. Users are encouraged to use the absolute path to the file.

.type type_name spec1 spec2 ... specn ; Declares a type type_name as a sequence of specifications spec1 spec2 ... specn where spec is any of %d %f %s (integer, float or string). A number may be used in front of a spec, to tell how many specs are to be used. A spec or set of specs can also be included inside parentheses, to indicate a variable number of that spec (or set of specs). A type is used whenever a .attribute or .global_attribute command is used. The type defines all the information that follows the type name. For .attribute, a string HAS to be inserted between the type name and the typed information. This string indicates the variable (or instance) to which the attribute will be attached.

Example:

```
fileA:
.model main ;                          # definition of model "main"
.inputs a b c d ;                      # inputs list
.outputs w x y z ;                     # outputs list
.inouts t ;                            # inouts list
.include fileC ;                       # fileC will be inserted here
.search fileB ;                        # fileB may contain needed
                                       # models
.type FORMAT1 2 %s %d ;                # type definition
.type FORMAT2 (%s (%f));
.attribute FORMAT2 r (a (2.0)) ;       # annotation of signal "r"
.attribute FORMAT1 inst0 z b 5 ;       # annotation of instance "inst0"
q = a b ; r = a b' + a' b ;            # logic equations
x = r' ; s = a' b' c' ;
y = d + d' (s + c q) ;
w = @ T (y, enable) ;                  # tristate element
.net enable clock r ;                  # all 3 signals are the same net
.call inst0 OR2 (b, c; ; z) ;          # OR-gate described externally
.call inst1 d_latch (c, clock; ;w);    # D-latch described externally
.endmodel main ;                       # end definition (model "main")

fileB:

# Externally-called models. Calling model must have
# argument lists of correct size and in correct order.
.model d_latch ;
.inputs a b ;
.outputs t ;
t = @ D(a,b) ;                         # built-in function
.endmodel d_latch ;

.model OR2;
.inputs x y;
.outputs z;
.library;                              # identifies as a library element
.endmodel OR2;

fileC:
```

```
# Information that will be inserted in model main

.type FORMAT1 2 %s %d ;              # types may be redefined if all
.type FORMAT2 (%s (%f)) ;            # definitions are consistent

.global_attribute FORMAT1 cap low 5;
.global_attribute FORMAT2 (min_res (3.0) typ_res (5.0 0.2));
.global_attribute FORMAT2 (delay (0.1 0.3 1.0 2.1));
```

Comments: Problems, comments and suggestions should be addressed to mailhot@Pegasus.Stanford.EDU.

Authors: Giovanni DeMicheli
Philip Johnson
David Ku
Frederic Mailhot

4.3 Two-Level Format

The two-level benchmarks are represented in either the ESPRESSO-MV format or the SLIF. The ESPRESSO-MV format is described by man(5) page in the ESPRESSO distribution from the University of California at Berkeley. The ESPRESSO-MV format description is repeated here.

4.3.1 ESPRESSO INPUT FILE DESCRIPTION

ESPRESSO accepts as input a two-level description of a Boolean switching function. This is described as a character matrix with keywords embedded in the input to specify the size of the matrix and the logical format of the input function. Comments are allowed within the input by placing a pound sign (#) as the first character on a line. Comments and unrecognized keywords are passed directly from the input file to standard output. Any white-space (blanks, tabs, etc.), except when used as a delimiter in an embedded command, is ignored. It is generally assumed that the PLA is specified such that each row of the PLA fits on a single line in the input file.

4.3.2 KEYWORDS

The following keywords are recognized by ESPRESSO. The list shows the probable order of the keywords in a PLA description. [d] denotes a decimal number and [s] denotes a text string.

.i [d] Specifies the number of input variables.

.o [d] Specifies the number of output functions.

.type [s] Sets the logical interpretation of the character matrix as described below under "Logical Description of a PLA". This keyword must come before any product terms. [s] is one of f, r, fd, fr, dr, or fdr.

.phase [s] [s] is a string of as many 0's or 1's as there are output functions. It specifies which polarity of each output function should be used for the minimization (a 1 specifies that the ON-set of the corresponding output function should be used, and a 0 specifies that the OFF-set of the corresponding output function should be minimized).

.pair [d] Specifies the number of pairs of variables which will be paired together using two-bit decoders. The rest of the line contains pairs of numbers which specify the binary variables of the PLA which will be paired together. The binary variables are numbered starting with 1. The PLA will be reshaped so that any unpaired binary variables occupy the leftmost part of the array, then the paired multiple-valued columns, and finally any multiple-valued variables.

.kiss Sets up for a KISS-style minimization.

.p [d] Specifies the number of product terms. The product terms (one per line) follow immediately after this keyword. Actually, this line is ignored, and the ".e", ".end", or the end of the file indicate the end of the input description.

.e (.end) Marks the end of the PLA description.

4.3.3 LOGICAL DESCRIPTION OF A PLA

When we speak of the ON-set of a Boolean function, we mean those minterms which imply the function value is a 1. Likewise, the OFF-set are those terms which imply the function is a 0, and the DC-set (don't care set) are those terms for which the function is unspecified. A function is completely described by providing its ON-set, OFF-set and DC-set. Note that all minterms lie in the union of the ON-set, OFF-set and DC-set, and that the ON-set, OFF-set and DC-set share no minterms.

The purpose of the ESPRESSO minimization program is to find a logically equivalent set of product-terms to represent the ON-set and optionally minterms which lie in the DC-set, without containing any minterms of the OFF-set.

A Boolean function can be described in one of the following ways:

1) By providing the ON-set. In this case, ESPRESSO computes the OFF-set as the complement of the ON-set and the DC-set is empty. This is indicated with the keyword .type f in the input file, or -f on the command line.

2) By providing the ON-set and DC-set. In this case, ESPRESSO computes the OFF-set as the complement of the union of the ON-set and the DC-set. If any minterm belongs to both the ON-set and DC-set, then it is considered a don't care and may be removed from the ON-set during the minimization process. This is indicated with the keyword .type fd in the input file, or -fd on the command line.

3) By providing the ON-set and OFF-set. In this case, ESPRESSO computes the DC-set as the complement of the union of the ON-set and the OFF-set. It is an error for any minterm to belong to both the ON-set and OFF-set. This error may not be detected during the minimization, but it can be checked with the subprogram "-do check" which will check the consistency of a function. This is indicated with the keyword on the command line.

4) By providing the ON-set, OFF-set and DC-set. This is indicated with the keyword .type fdr in the input file, or -fdr on the command line.

If at all possible, ESPRESSO should be given the DC-set (either implicitly or explicitly) in order to improve the results of the minimization.

A term is represented by a "cube" which can be considered either a compact representation of an algebraic product term which implies the function value is a 1, or as a representation of a row in a PLA which implements the term. A cube has an input part which corresponds to the input plane of a PLA, and an output part which corresponds to the output plane of a PLA (for the multiple-valued case, see below).

4.3.4 SYMBOLS IN THE PLA MATRIX AND THEIR INTERPRETATION

Each position in the input plane corresponds to an input variable where a **0** implies the corresponding input literal appears complemented in the product term, a **1** implies the input literal appears uncomplemented in the product term, and **-** implies the input literal does not appear in the product term.

With logical type **f**, for each output, a **1** means this product term belongs to the ON-set, and a **0** or **-** means this product term has no meaning for the value of this function. This logical type corresponds to an actual PLA where only the ON-set is actually implemented.

With logical type **fd** (the default), for each output, a **1** means this product term belongs to the ON-set, a **0** means this product term has no meaning for the value of this function, and a **-** implies this product term belongs to the DC-set.

With logical type **fr**, for each output, a **1** means this product term belongs to the ON-set, a **0** means this product term belongs to the OFF-set, and a **-** means this product term has no meaning for the value of this function.

With logical type **fdr**, for each output, a **1** means this product term belongs to the ON-set, a **0** means this product term belongs to the OFF-set, a **-** means this product term belongs to the DC-set, and a ~ implies this product term has no meaning for the value of this function.

Note that regardless of the logical type of PLA, a ~ implies the product term has no meaning for the value of this function. **2** is allowed as a synonym for **-**, **4** is allowed for **1**, and **3** is allowed for ~. Also, the logical PLA type can also be specified on the command line.

4.3.5 Example con1.pla

The following two-level logic description is an example of the ESPRESSO format.

```
.i 7
.o 2
.p 9
-1--1-- 10
1-11--- 10
-001--- 10
01---1- 10
-0--0-- 01
1---0-- 01
0-----0 01
01--1-- 01
10-0--- 01
.e
```

5 Cell Library Descriptions

When comparing results of synthesis programs, the target library must be standardized. For maximum flexibility, and to encourage more participation, we have selected four libraries. The benchmarks may be realized using anyone of these libraries. When reporting results, it is important to specify which library was used for implementing a logic example.

5.1 Lib1: The Small Unit-Delay Model

5.1.1 Lib1.1: Combinational

This library consists of a subset of gates that are commonly used in ASIC libraries. The gates have a simple timing model. The delay through the logic is one unit per cell plus 0.2 units for each fanout of a cell. The complete library is available in electronic form (MISII and textual) with the benchmark distribution. A hard copy of the library can be created by the user and appended to this document.

5.1.2 Lib1.2: Sequential

This library consists of a subset of gates and Flipflops that are commonly used in ASIC libraries. The gates and flipflops have a simple timing model. The delay through the logic is one unit per cell plus 0.2 units for each fanout of a cell. The complete library is available in electronic form (SIS) with the benchmark distribution. A hard copy of the library can be created by the user and appended to this document.

5.2 Lib2: The MOSIS 2u standard cell library

This is a library of combinational cells that is somewhat more realistic. It contains 29 gates including four different size inverters for buffering purposes. The timing model is separate rise/fall with a separate delay equation and capacitive load associated with each input pin. The complete library is available in electronic form (MISII and textual) with the benchmark distribution. A hard copy of the library can be created by the user and appended to this document.

5.3 Lib3: The ADVANCELL_D library

This is a library obtained from industry. It contains 66 gates including six different size inverters and four non-inverting buffers for buffering purposes. The timing model is separate rise/fall with a separate delay equation and capacitive load associated with each input pin. The complete library is available in electronic form (MISII) with the benchmark distribution. A hard copy of the library can be created by the user and appended to this document.

6 Cell Library Formats

The cell libraries are made available in both the MISII/SIS and MCNC formats. The MISII/SIS format can be read by the MISII and SIS program. The MCNC format is made available as an easily human-readable format, and an optional parser/reader is available for this format upon request. Both formats contain the same information.

6.1 MISII/SIS Library Format

A cell is specified in the following format:

```
GATE <cell-name> <cell-area> <cell-logic-function>
<pin-info>
    .
    .
<pin-info>
```

<**cell-name**> is the name of the cell in the cell library. The resulting net-list will be in terms of these names.

<**cell-area**> defines the relative area cost of the cell. It is a floating point number, and may be in any unit system convenient for the user.

<**cell-logic-function**> is an equation written in conventional algebraic notation using the operators '+' for OR, '*' for AND, '!' for NOT, and parentheses for grouping. The names of the literals in the equation define the input pin names for the cell; the name on the left hand side of the equation defines the output of the cell. The equation terminates with a semicolon.

Only single-output cells may be specified. The '!' operator may only be used on the input literals, or on the final output; it is not allowed internal to an expression. (This constraint may disappear in the future).

Also, the actual factored form is significant when a logic function has multiple factored forms. In principle, all factored forms could be derived for a given logic function automatically; this is not yet implemented, so each must be specified separately. Note that factored forms which differ by a permutation of the input variables (or by De Morgan's law) are not considered unique.

Each <**pin-info**> has the format:

```
PIN <pin-name> <phase> <input-load> <max-load>
    <rise-block-delay> <rise-fanout-delay>
    <fall-block-delay> <fall-fanout-delay>
```

<**pin-name**> must be the name of a pin in the <cell-logic-function>, or it can be * to specify identical timing information for all pins.

<**phase**> is INV, NONINV, or UNKNOWN corresponding to whether the logic function is negative-unate, positive-unate, or binate in this input variable respectively. This is required for the separate rise-fall delay model. (In principle, this information is easily derived from the logic function; this field may disappear in the future).

<**input-load**> gives the input load of this pin. It is a floating point value, in arbitrary units convenient for the user.

<**max-load**> specifies a loading constraint for the cell. It is a floating point value specifying the maximum load allowed on the output.

<**rise-block-delay**> and <rise-fanout-delay> are the rise-time parameters for the timing model. They are floating point values, typically in the units nanoseconds, and nanoseconds/unit-load respectively.

<**fall-block-delay**> and <**fall-fanout-delay**> are the fall-time parameters for the timing model. They are floating point values, typically in the units nanoseconds, and nanoseconds/unit-load respectively.

All of the delay information is specified on a pin-by-pin basis. The meaning is the delay information for the most critical pin is used to determine the delay for the gate.

6.1.1 MISII/SIS Library Format Example

```
GATE xor 5.5 O=a*!b+!a*b; PIN * UNKNOWN 2 999 1.9 0.5 1.9 0.5
GATE xor 5.5 O=!(a*b+!a*!b); PIN * UNKNOWN 2 999 1.9 0.5 1.9 0.5

GATE xnor 5.5 O=a*b+!a*!b; PIN * UNKNOWN 2 999 2.1 0.5 2.1 0.5
GATE xnor 5.5 O=!(!a*b+a*!b); PIN * UNKNOWN 2 999 2.1 0.5 2.1 0.5

GATE mux21 4.5 O=a*s+b*!s;
    PIN a NONINV 1 999 1.6 0.4 1.6 0.4
    PIN b NONINV 1 999 1.6 0.4 1.6 0.4
    PIN s UNKNOWN 2 999 2.0 0.4 1.6 0.4
GATE mux21 4.5 O=!(!a*s+!b*!s);
    PIN a NONINV 1 999 1.6 0.4 1.6 0.4
    PIN b NONINV 1 999 1.6 0.4 1.6 0.4
    PIN s UNKNOWN 2 999 2.0 0.4 1.6 0.4
```

6.2 MCNC Library Format

The MCNC library file has the form

```
--------------------------
<cell_description>
<cell_description>
<cell_description>
  "     "
  "     "
  "     "
--------------------------


Where <cell_description> is

cell begin <name>
  area=<float>
  equation="<logic equation>"
  max_loads=<float>
  primitive=<string>

  termlist
    <terminal_name>
    unateness=<string: INV, NONINV, UNKNOWN>
    loads=<float>
 /* requires at least one set of best, worst or nominal */
    nominal_rise_delay=<float>
    nominal_rise_fan=<float>
    nominal_fall_delay=<float>
    nominal_fall_fan=<float>

    worst_rise_delay=<float>
    worst_rise_fan=<float>
    worst_fall_delay=<float>
    worst_fall_fan=<float>

    best_rise_delay=<float>
    best_rise_fan=<float>
    best_fall_delay=<float>
    best_fall_fan=<float> ;
cell end <name>



1.  All above text lines are optional except "cell begin" and "cell end."
```

6.2.1 MCNC Library Format Example

Example:

```
cell begin xnor
  area=5.0
  equation="y = ! ( (a + b) * (!a + !b) )"
  max_loads=10
  primitive=XNOR

  termlist
    a
    unateness=UNKNOWN
    loads=2.0
    nominal_rise_delay=1.4
    nominal_rise_fan=2.9
    nominal_fall_delay=2.3
    nominal_fall_fan=3.6 ; /* terminal "b" with attributes */
    b
    unateness=UNKNOWN
    loads=2.0
    nominal_rise_delay=1.4
    nominal_rise_fan=2.9
    nominal_fall_delay=2.3
    nominal_fall_fan=3.6 ; /* terminal "b" with attributes */
    y ;  /* output terminal, no attributes */

cell end xnor
```

7 Guidelines for Reporting Results

A survey of researchers indicated that the following procedures and quality criteria would be useful for a meaningful comparison of various synthesis and optimization methods.

7.1 Suggested Benchmark Subsets

Because of the large number of examples in each benchmark set, it will be difficult to run and report on all examples. To insure some overlap of examples between researchers reporting results, please attempt to include the following listed examples from each category in your subset of examples. If you can report on all examples, so much the better.

Suggested for the FSM set:

1. cse
2. donfile
3. dk16
4. dk512
5. ex1
6. keyb
7. styr
8. s1
9. s1a
10. s298
11. tbk
12. tma

Suggested for the sequential multi-level set:

1. Optimization
   (a) s298
   (b) s400
   (c) s444
   (d) s510
   (e) s526
   (f) s713
   (g) s820
   (h) s1488

   (i) s5378
   (j) s9234.1
2. Verification
   (a) clma
   (b) clmb
   (c) mm30a
   (d) bigkey
   (e) mult32b
   (f) sbc
   (g) s38584.1
   (h) s15850.1
   (i) s13207.1

Suggested for the combinational multi-level set:

1. C432
2. C1355
3. C1908
4. C2670
5. C3570
6. C6288
7. C7552
8. t481
9. rot
10. b9
11. dalu
12. des
13. k2

Suggested for the two-level set:

1. duke2
2. rd84
3. misex2
4. misex3c
5. b12
6. cordic
7. cps
8. ex4
9. ex1010
10. pdc
11. spla

Index		Factored Literals Before Mapping		After Mapping
		Init. Ckt	Optimized	Tr-pr
1	b9	236	152	186
2	ttt2	341	211	268
3	apex7	290	292	367
4	example2	366	377	488
5	C1908	1497	535	672
6	C1355	1064	558	638
7	C2670	2075	936	1119
8	s1488	1387	717	913

Table 1: Counting literals does not translate into transistor pairs

7.2 Quality Criteria

Researchers may report their results in terms of multiple objectives such as transistor pair counts, propagation delay, and CPU time. It is also important to address such issues as follows:

- Was your circuit synthesized by totally automatic means ?
- How many trials or different scripts (multiple runs) were used ?
- What were the best results ever seen for an example, regardless of whether the method was automatic or if multiple trials were used ?

7.2.1 Area Results

One of the problems in multi-level area minimization is how to choose a cost function that will abstractly and accurately represent the final objective cost, say transistor pairs after the technology mapping or the layout area after the placement and routing. Often, results of optimization are reported in terms of the total number of literal counts. This problem arises with the two phases of optimization: the technology independent phase and the technology dependent phase. The literal count, typically used in technology independent phase, is too optimistic to predict the transistor pairs after the technology mapping. In Table 1, we examined 8 circuits with the initial literal count ranging from 54 to 2075 (Init. Ckt). Using algebraic factorization in misII (with script.algebraic) the literal count appears reduced in most instances and ranges from 152 to 936 (optimized). After completing the technology mapping optimization phase (the library used is mcnc.genlib), we report the number of literals as transistor pairs, ranging from 186 to 1119 (Tr-pr). We calculated the percentages in gain/loss in literal count before the technology mapping with the corresponding percentages in gain/loss in transistor count after the technology mapping. The results are shown in Table 2. Note that for circuits 1, 5, 7, and 8 the difference in the two counts are substantial. Even worse, for circuit 3 and 4 the reduction in literals does not represent the upper bound on the reduction in transistor pairs.

The results shown here give a clear indication that in order to measure effectiveness of a given multi-level optimization strategy, one must unambiguously state the results of technology mapping *before and after* the technology independent phase of optimization.

As it has been observed by this experiment that the literal counts before the technology mapping gives poor and inaccurate area estimator measures, it is strongly recommended to report the area estimator after a technology mapping. The results may be reported with the following information:

Index		Before Mapping			After Mapping		
		Init	Opt	%	Init	Opt	%
1	b9	236	151	35.6	202	186	7.9
2	ttt2	341	211	38.1	429	268	37.5
3	apex7	290	292	-0.7	403	367	8.9
4	example2	366	377	-3.0	494	488	1.2
5	C1908	1497	535	64.3	1962	672	30.1
6	C1355	1064	558	47.6	1057	638	39.6
7	C2670	2075	936	54.9	1409	1119	20.6
8	s1488	1387	717	48.3	1125	913	18.8

Table 2: Minimizing literals need not minimize transistor pairs

- Total Area - The sum of the cell areas as defined for each cell in the library (specify lib1, lib2 or lib3).
- Transistor Pairs - Optionally, report the number of transistor pairs in a multi-level implementation using the libraries (specify lib1, lib2 or lib3).
- Gates - The total number of gates or cells used (again, specify library used).
- Grids - The sum of gates and transistor pairs.

7.2.2 Delay Results

When using library 1 (LIB1) gate delay computation reduces to a simple strategy. Compute each gate delay as 1.0 plus 0.2 times the number of fanouts at the gate's output. Then compute the longest delays to all points from inputs to outputs.

For library 2 or library 3 results, the following delay results would be more useful:

- Critical path delay time through the netlist using the intrinsic-plus-fanout delay model given in the libraries (Lib2 and Lib3).
- The preferred critical-path-delay analysis method is to use LATCH-TO-LATCH delays for computing the longest path through the logic. This assumes that all primary inputs and outputs are latched. To accomplish this assume standard load on all outputs (DFF latch) and assume the standard drive on all inputs (DFF latch). The load and drive of the standard latch is the same as for the inv2x inverter.

  Note that latch-to-latch delays must be used to account for the difference in loads that appear on the inputs of different circuit realizations. Otherwise, circuits with arbitrarily large input loads could be realized without adding delay.

7.2.3 Preserving I/O Behavior in Sequential Optimization

When the sequential resynthesis and retiming technique is applied to a sequential circuit, the I/O behavior should be preserved. Especially, the output value of resynthesized and(or) retimed circuit should be same as that of original circuit after the initializing sequence is applied. Therefore, when the result of sequential resynthesis and retiming technique is reported, describe the method that solves this initialization problem.

7.2.4 CPU Results

Indicate total CPU requirements for your procedure (Time and machine). Indicate how many trials were needed to obtain your presented results.

8 Readings and References

The following references cover background, algorithms, and further insights on multi-level sequential logic synthesis[**?**, **?**], multi-level combinational logic synthesis[**?**, **?**], decomposition and factoring[**?**, **?**], technology mapping[**?**, **?**, **?**], two-level logic minimization [**?**, **?**], and previous logic synthesis benchmarking[**?**, **?**].

References

[1] S. Malik, E.M. Sentovich, R.K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques , In *IEEE Transactions on CAD* pages 74 – 84, Jan. 1991.

[2] G. De Micheli. Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization , In *IEEE Transactions on CAD* pages 63 – 73, Jan. 1991.

[3] D. Bostick, G. Hachtel, R. Jacoby, P. Moceyunas, C. Morrison, and D. Ravenscroft. The Boulder Optimal Logic Design System. In *IEEE International Conference on Computer-Aided Design*, pages 62 – 65, November 1987.

[4] R. Lisanke, G. Kedem, and F. Brglez. *DECAF: Decomposition and Factoring for Multi-Level Logic Synthesis.* Technical Report, Microelectronics Center of North Carolina, Research Triangle Park, NC, August 1987.

[5] R. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In *IEEE International Symposium on Circuits And Systems*, pages 49 – 54, May 1982.

[6] R. Brayton and C. McMullen. Synthesis and Optimization of Multi-Level Logic. In *IEEE International Conference on Computer Design*, pages 23 – 28, October 1984.

[7] D. Gregory, K. Bartlett, A. deGeus, and G. Hachtel. SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic. In *IEEE 23rd Design Automation Conference*, pages 79 – 85, June 1986.

[8] K. Keutzer. DAGON: Technology Binding and Local Optimization by DAG Matching. In *IEEE 24th Design Automation Conference*, pages 341 – 347, June 1987.

[9] R. Lisanke, F. Brglez, and G. Kedem. McMAP: A Fast Technology Mapping Procedure for Multi-Level Logic Synthesis. In *IEEE International Conference on Computer Design*, October 1988.

[10] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Press, Boston, 1984.

[11] R. Rudell. *Multiple-Valued Logic Minimization for PLA Synthesis.* Technical Report, University of California, Electronics Research Laboratory, Berkeley, CA, June 1986.

[12] A. deGeus. Logic Synthesis and Optimization Benchmarks. In *IEEE 23rd Design Automation Conference*, page 78, June 1986.

[13] R. Lisanke. Logic Synthesis and Optimization Benchmarks User Guide: Version 2.0 Technical Report, Microelectronics Center of North Carolina, Dec. 1988.