

RevLib – File Format Documentation

Version 2.0.1 – May 22, 2011
<http://www.revlib.org>
revlib@informatik.uni-bremen.de

Contents

1	Introduction	1
2	HDL Specifications	1
2.1	Module and Signal Declarations	3
2.2	Statements	4
2.3	Examples	5
3	Boolean Functions	7
3.1	PLA Format (Sum of Products)	7
3.2	SPEC Format (Truth Table)	9
4	Circuits	10
4.1	Simple Combinatorial Circuits	11
4.2	Sub-circuits	12
4.3	Buses	13
4.4	Sequential Circuits	13
4.5	Annotations	14
5	Simulation Patterns	15
6	Submit Files to RevLib	16
7	Version History	16
8	Acknowledgments	16

1 Introduction

RevLib is an online resource for reversible specifications and reversible circuits aiming to make recent results in the domain of reversible logic accessible to other researchers. RevLib provides a benchmark suite of reversible functions, embeddings of irreversible functions (containing constant inputs and garbage outputs), as well as high level specifications based on the reversible hardware description language (HDL) SyReC [?]. Furthermore, for each function or specification at least one reversible circuit realization is given. These circuits are obtained from exact synthesis approaches (e.g. [?, ?]), from heuristic synthesis approaches (e.g. [?, ?]), or from HDL synthesizers (e.g. [?]).

Using the benchmarks provided at RevLib, the results of (new) synthesis or optimization approaches can be compared to existing realizations. Furthermore, the provided circuits can serve as benchmarks e.g. for the empirical evaluation of verification approaches, testing methods, or similar purposes. Besides that, RevLib is the main file format for RevKit, a public domain toolkit for reversible circuit design [?]. In order to enlarge the database any researcher is invited to submit specifications, functions, and/or circuit realizations to RevLib.

In this document, the file format as used in RevLib is described. We distinguish thereby between the formats for

- specifications given in the HDL SyReC,
- reversible functions or embeddings of irreversible functions, as well as
- reversible circuits, as well as
- simulation patterns.

In the following section, the respective file formats are introduced. The main aspects are thereby kept brief, but are illustrated by means of examples. To check the correct formatting of own files, the above mentioned RevKit tool (available at www.revkit.org) can be utilized, which further provides parsing routines for each file format described in this document.

2 File Format for HDL Specifications Using SyReC

SyReC is a hardware description language for reversible circuits proposed in [?] and is based on the reversible software language Janus [?]. It provides fundamental constructs to define control and data operations, while still preserving reversibility. The syntax of a SyReC specification is outlined in Figure 1 and briefly described in the following¹.

¹Note that the current version of SyReC includes some extensions which are not described in [?].

Program and Modules

```

<program> ::= <module> {<module>}
<module> ::= 'module' <identifier> '(' [<parameter-list>] ')' {<signal-list>} <statement-list>
<parameter-list> ::= <parameter> {',' <parameter>}
<parameter> ::= ('in' | 'out' | 'inout') <signal-declaration>
<signal-list> ::= ('wire' | 'state') <signal-declaration> {',' <signal-declaration>}
<signal-declaration> ::= <identifier> {'[' <int> ']' | ['(' <int> ')']}

```

Statements

```

<statement-list> ::= <statement> {';' <statement>}
<statement> ::= <call-statement> | <for-statement> | <if-statement> | <unary-statement> |
<assign-statement> | <swap-statement> | <skip-statement>
<call-statement> ::= ('call' | 'uncall') <identifier> '(' (<identifier> {',' <identifier>}) ')'
<for-statement> ::= 'for' [['$' <identifier> '='] <number> 'to'] <number> ['step' ['-']
<number>] <statement-list> 'rof'
<if-statement> ::= 'if' <expression> 'then' <statement-list> 'else' <statement-list> 'fi'
<expression>
<unary-statement> ::= ('~' | '++' | '--') '=' <signal>
<assign-statement> ::= <signal> ('~' | '+' | '-') '=' <expression>
<swap-statement> ::= <signal> '<=>' <signal>
<skip-statement> ::= 'skip'
<signal> ::= <identifier> {'[' <expression> ']' | ['. ' <number> [': ' <number>]]}

```

Expressions

```

<expression> ::= <number> | <signal> | <binary-expression> | <unary-expression> |
<shift-expression>
<binary-expression> ::= '(' <expression> ('+' | '-' | '~' | '*' | '/' | '%' | '*>' | '&&' | '||' | '&' |
'|' | '<' | '>' | '=' | '!=' | '<=' | '>=') <expression> ')'
<unary-expression> ::= ('!' | '~') <expression>
<shift-expression> ::= '(' <expression> ('<<' | '>>') <number> ')'

```

Data-types

```

<letter> ::= ('A' | ... | 'Z' | 'a' | ... | 'z')
<digit> ::= ('0' | ... | '9')
<identifier> ::= ('_' | <letter>) {'_' | <letter> | <digit>}
<int> ::= <digit> {<digit>}
<number> ::= <int> | '#' <identifier> | '$' <identifier>
| '(' <number> ('+' | '-' | '*' | '/') <number> ')'

```

Figure 1: Syntax of the hardware language SyReC

2.1 Module and Signal Declarations

Each SyReC program (denoted by $\langle program \rangle$) consists of one or more modules (denoted by $\langle module \rangle$). A module is introduced with the keyword **module** and includes an identifier (represented by a string), a list of parameters representing global signals (denoted by $\langle parameter-list \rangle$), local signal declarations (denoted by $\langle signal-list \rangle$), and a sequence of statements (denoted by $\langle statement-list \rangle$). The top-module of a program is defined by the special identifier **main**. If no module with this name exists, the last module declared is used instead by convention.

SyReC uses a *signal* representing a non-negative integer as its sole data type. The bit-width of signals can optionally be defined by round brackets after the signal name. If no bit-width is specified, a default value is assumed. For each signal, an *access modifier* has to be given. For a parameter signal (used in a module declaration) this can be **in**, **out**, and **inout**. Local signals can either work as internal signals (denoted by **wire**) or in case of sequential circuits as state signals² (denoted by **state**). The access modifier influences properties in the synthesized circuits as given in Table 1.

Table 1: Signal access modifiers and implied circuit properties

Modifier	Constant Value	Garbage	State	Initial Value
in	–	yes	no	given by primary input
out	0	no	no	0
inout	–	no	no	given by primary input
wire	0	yes	no	0
state	–	no	yes	given by pseudo-primary input

Signals can be grouped to multi-dimensional arrays of constant length using square brackets after the signal name and before the optional bit-width declaration.

Examples:

```
// module declaration with two inputs and one output:
module adder( in a, in b, out c )

// the same declaration with 16-bit signals :
module adder( in a(16), in b(16), out c(16) )

// an adder summing up 4 inputs given as an array
module adder( in inputs[4](16), out c(16) )

// module with local variables and state variables
module myCircuit( in input1, in input2, out output )
  wire auxSignal(16)
  state stateSignal
```

²On sequential reversible circuits, see also Section 4.4.

2.2 Statements

Statements include call and uncall of other modules, loops, conditional statements, and various data operations (i.e. unary operations, reversible assignment operations, swap statements). The empty statement can explicitly be modelled using the **skip** keyword. Statements are separated by semicolons. Signals within statements are denoted by *<signal>* allowing access to the whole signal (e.g. *x*), a certain bit (e.g. *x.4*), or a range of bits (e.g. *x.2:4*). The bit-width of a signal can also be accessed (e.g. *#x*).

To call another module, simply the keyword **call** (**uncall**) together with the identifier of the module to be called along with the parameters have to be applied.

Example:

```
// calling a module identified by adder
wire a, b, c
call adder(a, b, c )
```

In loops, the number of iterations (and therewith the number of duplications of the respective code block) needs to be defined. This number has to be available prior to the compilation, i.e., dynamic loops are not allowed. Therefore, e.g. fix integer values, the bit-width of a signal, or internal *\$*-variables can be applied. Furthermore, the current value of internal counter variables can be accessed during the iterations. Using the optional keyword **step**, also the iteration itself can be modified. A loop is terminated by **rof**.

Examples:

```
for 1 to 10 do
  // statements
rof

// iterating over the bit-width of a variable
wire x
for $i = 0 to #x do
  /* statements (possibly using $i)
   the i-th bit of x can be accessed by x.$i
   a range of bits can be accessed e.g. by x.0:$i */
rof

for $counter = 1 to 10 step 2 do
  // statements
  // the loops iterates 5 times (i.e., $counter is set to 1, 3, 5, 7, and 9 only)
rof
```

Conditional statements need an expression to be evaluated followed by the respective then-block and else-block. Each of these blocks is a sequence of statements. In order to ensure reversibility, a conditional statement is terminated by **fi** and an adjusted expression.

Example:

```

if ( b = 5 ) then
  x += y // statements that are executed if b = 5
else
  x -= y // statements that are executed if b != 5
fi ( b = 5 )

```

Finally, statements can express various data operations. Operations are thereby distinguished between reversible assignment operations (denoted by *assign-statement*), unary operations (denoted by *unary-statement*), the swap operation (denoted by *swap-statement*), and the not necessarily reversible *binary operations* (denoted by *binary-expression*) as well as the *shift operations* (denoted by *shift-expression*). Table 2 and 3 lists all statements and expressions, respectively, along with their semantics. Variable accesses are referred to as *x* and *y*, expressions as *e* and *f*, and natural numbers as *n*.

Reversible assignment operations assign values to a signal on the left-hand side. Therefore, the respective signal must not appear in the expression on the right-hand side. Furthermore, only a restricted set of assignment operations exists, namely increase (+=), decrease (-=), bit-wise XOR (^=). These operations preserve the reversibility (i.e., it is possible to compute these operations in both directions). The same holds for the unary operations, namely bit-wise negation (~), increase by one (++), and decrease by one (--), as well as for the swap operation (<=>).

In contrast, binary operations, i.e., arithmetic (+, *, /, %, *>), bit-wise (&, |, ^), logical (&&, ||), relational (<, >, =, !=, <=, >=), and shifting (<<, >>) operations, may not be reversible. Thus, they can only be used in right-hand expressions (denoted by *expression*) which preserve, i.e., do not modify, the values of the respective inputs. In doing so, all computations remain reversible since the input values can be applied to reverse any operation. For example, to specify the multiplication $a*b$ in SyReC, a new free signal *c* must be introduced which is used to store the product. That results in the expression $c^{\wedge}=a*b$. In comparison to common (irreversible) programming languages, statements such as $a=b + (5 * a)$ are not allowed.

2.3 Examples

Using SyReC, complex reversible circuits can be specified. Some examples are provided in the following.

Arithmetic Logic Unit:

```

module alu( in op(2), in a, in b, out c )
  if ( op = 0 ) then
    c ^= ( a + b )
  else
    if ( op = 1 ) then
      c ^= ( a - b )
    else

```

Table 2: Statements in SyReC

Operation	Semantic
$\sim = x$	Bit-wise negation of x
$++ = x$	Increment of x
$-- = x$	Decrement of x
$x \hat{=} e$	Bit-wise XOR assignment of e to x , i.e. $x \leftarrow x \oplus e$
$x += e$	Increase by value of e to x , i.e. $x \leftarrow x + e$
$x -= e$	Decrease by value of e to x , i.e. $x \leftarrow x - e$
$x \langle = \rangle y$	Swapping value of x with value of y

Table 3: Expressions in SyReC

Operation	Semantic
$e + f$	Addition of e and f
$e - f$	Subtraction of e and f
$e * f$	Lower bits of multiplication of e and f
$e * > f$	Upper bits of multiplication of e and f
e / f	Division of e and f
$e \% f$	Remainder of division of e and f
$e \hat{ } f$	Bit-wise XOR of e and f
$e \& f$	Bit-wise AND of e and f
$e f$	Bit-wise OR of e and f
$\sim e$	Bit-wise negation of e
$e \&\& f$	Logical AND of e and f
$e f$	Logical OR of e and f
$!e$	Logical NOT of e
$e < f$	True, if and only if e is less than f
$e > f$	True, if and only if e is greater than f
$e = f$	True, if and only if e equals f
$e != f$	True, if and only if e not equals f
$e \leq f$	True, if and only if e is less or equal to f
$e \geq f$	True, if and only if e is greater or equal to f
$e \ll n$	Logical left shift of e by n
$e \gg n$	Logical right shift of e by n

```

    if ( op = 2 ) then
      c ^= ( a * b )
    else
      c ^= ( a / b )
    fi ( op = 2 )
  fi ( op = 1 )
fi ( op = 0 )

```

Sequential counter:

```

module counter( in reset(1),
                in enable(1)
                in jump,
                inout count_val )

  wire zero

  if ( reset ) then
    count_val <=> zero
  else
    if ( enable ) then
      count_val <=> jump
    else
      ++=count_val
    fi ( enable )
  fi ( reset )

module main( in reset( 1 ), in enable( 1 ), in jump )
  state count_val

  call counter( reset, enable, jump, count_val )

```

Further examples are available at RevLib.

3 File Format for Boolean Functions

In RevLib, Boolean functions are either defined by means of two-level representations based on the PLA format (as also used by Espresso [?]) or by a truth-table like representation based on a so called SPEC format. While the PLA format enables to specify larger functions, the SPEC format can explicitly be used to specify embeddings of irreversible functions.

3.1 PLA Format (Sum of Products)

Boolean functions can be represented by two-level descriptions, such as *Sum of Products* (SoPs). Here, the function is defined by a disjunction of conjunctions, which allows

x_1	x_2	x_3	x_4	x_5	f_1	f_2	f_3
1	-	-	0	-	1	0	0
0	0	-	-	-	0	1	0
1	1	-	-	1	0	0	1
-	1	0	-	-	0	0	1
1	0	-	1	-	1	0	1
1	1	-	1	0	1	0	1

(a) Sum of Products

```
. i 5
. o 3
. ilb x1 x2 x3 x4 x5
. ob f1 f2 f3
1--0-- 100
00---- 010
11--1 001
-10-- 001
10-1- 101
11-10 101
```

(b) File format

Figure 2: Sum of Products and its representation in the PLA format

a compact specification in many cases. As an example, consider function given in Figure 2(a). The column on the left-hand side gives the respective conjunctions of the primary inputs, where a 1 on the i^{th} position denotes a positive literal (i.e., x_i) and a 0 denotes a negative literal (i.e., \bar{x}_i), respectively. A '-' denotes that the respective variable is not included in the product. The right-hand side gives the respective primary output patterns. The disjunction of all rows leads to the overall function. This kind of description is frequently used in logic synthesis. Espresso [?] is a prominent example relying on that.

Any function given as SoP can be defined in the PLA format. Therefore, an ASCII file containing the following information needs to be created:

- *Comments:* Comments provide human-readable information about the respective instance, e.g. a short description, authors, or similar. A comment line starts with a # character. All comment lines will be ignored by programs.
- *Number of Inputs:* The line starting with .i signifies the number of primary inputs. The number has to be a positive integer.
- *Number of Outputs:* The line starting with .o signifies the number of primary outputs. The number has to be a positive integer.
- *Input/Output Labels:* The line starting with .ilb (.ob) signifies the ordered list of names of the respective inputs (outputs). A name has to be a sequence of letters, digits and underscores. The number of names have to correspond to the number of inputs (outputs) previously defined in the line starting with .i (.o). Different names are separated by spaces. Defining the input/output labels is optional.

Followed by this, the respective products of the SoP description are listed. For each product, a new line is added. The first characters denote thereby the phases of the respective literals within this product (again, a 1 denotes a positive literal, a 0 denotes a negative literal, and a '-' denotes that the respective variable is not included in the

product). Separated by a single space, the respective primary output patterns are listed. After all products are listed, the end of file is denoted by `.e`.

As an example, Figure 2(b) shows the file format for the SoP provided in Figure 2(a).

3.2 SPEC Format (Truth Table)

Alternatively, functions can be specified in the SPEC format. Here, the respective truth table is defined. In particular for concrete embeddings of irreversible functions, this file format is important as it allows an explicit definition of the embedding including e.g. the concrete values of constant inputs or the the concrete position of garbage outputs. This might be crucial, since many synthesis approaches generate different circuits depending on the concrete embedding.

Using the SPEC format, a function is defined in an ASCII file consisting of two parts: the header and the actual function description. The header contains information about the characteristics of the instance, i.e.:

- *Version*: In order to support further enhancements, any changes will be associated with a version number. The version of a respective file is given by the line starting with `.version`. The current version of this revision is `2.0`.
- *Comments*: Comments provide human-readable information about the respective instance, e.g. a short description, authors, or similar. A comment line starts with a `#` character. All comment lines will be ignored by programs.
- *Number of Variables*: The line starting with `.numvars` signifies the number `n` of variables (lines) of the respective function (circuit). The number has to be a positive integer.
- *Variables*: The line starting with `.variables` signifies the ordered list of identifiers for the respective variables (lines). An identifier has to be a sequence of letters, digits and underscores. In total, `n` different identifiers have to be defined in a `.variables` line (separated by spaces).
- *Inputs/Outputs*: The line starting with `.inputs (.outputs)` signifies the ordered list of names of the respective inputs (outputs). A name has to be a sequence of letters, digits, and underscores. Alternatively, names can be embedded into quotation marks (e.g. `"Select Input"`). In total, `n` different names have to be defined in a `.inputs (outputs)` line (separated by spaces). Defining the inputs (outputs) is optional.
- *Constant Inputs*: Constant inputs can be defined by a line starting with `.constants` followed by a string containing the values for the input constants (`'1'` for constant one, `'0'` for constant zero and `'-'` if the respective input line is not constant) in the order they appear without any spaces. Defining constant inputs is optional. By default all inputs are not constants (i.e., for each input the value is `'-'`).

1	a	b	f	g ₁	g ₂
0	0	0	-	-	-
0	0	1	-	-	-
0	1	0	-	-	-
0	1	1	-	-	-
1	0	0	0	-	-
1	0	1	0	-	-
1	1	0	0	-	-
1	1	1	1	-	-

(a) Boolean function

```
# Embedded AND function
.version 2.0
.numvars 3
.variables x y z
.inputs 1 a b
.outputs f g1 g2
.constants 1---
.garbage -11
.begin
----
----
----
----
0--
0--
0--
1--
.end
```

(b) File format

Figure 3: Embedded Boolean function and its representation in the SPEC format

- *Garbage Outputs:* Garbage outputs can be defined by a line starting with `.garbage` followed by a string indicating whether an output is garbage or not ('1' if the output is garbage and '-' if the output is not garbage) in the order they appear without any spaces. Defining garbage outputs is optional. By default all outputs are not garbage (i.e., for each output the value is '-').

Then, the actual function is specified. This starts with a line containing the string `.begin` and ends with a line containing the string `.end`. A function is specified by all its truth table entries. The i th line ($0 \leq i < 2^n$) of the specification gives the respective output values (1 for one, 0 for zero, and - for don't care) of the i^{th} line of the truth table by a string in the order they appear without any spaces.

As an example, Figure 3(a) shows an embedded irreversible function including one constant input as well as garbage outputs and don't care outputs given as truth table. The respective SPEC file for this function is shown in Figure 3(b).

4 File Format for Circuits

In RevLib, reversible circuits are specified in the so called REAL format. Depending on the application and/or the circuit itself, different levels of details are possible. First, the file format of a simple combinatorial reversible circuit is introduced. Afterwards, the respective extensions supporting the integration of sub-circuits/modules, sequential reversible circuits, grouping to buses, and additional annotations are described.

4.1 Simple Combinatorial Circuits

Using the REAL format, a circuit is defined in an ASCII file consisting of two parts: the header and the actual circuit description. The header contains information about the characteristics of the instance and is equal to the respective header for SPEC-files already introduced in Page 9 (whereby in context of the circuit description, each variable corresponds to a circuit signal).

Then, the actual circuit is defined. This starts with a line containing the string `.begin` and ends with a line containing the string `.end`. Between these keywords, all gates of the circuit are listed in the order they appear in the realization. It is thereby distinguished between the following gate types:

- *Multiple Control Toffoli gates (MCT)*

A (multiple control) Toffoli gate is signified by the character `t` and an integer indicating the size of the gate followed by a list of signals for the respective lines such that the target line is at the end of the list. Note that Toffoli gates include (controlled) NOT gates as well.

Example: `t3 a b c`

- *Multiple Control Fredkin gate (MCF)*

A (multiple control) Fredkin gate is signified by the character `f` and an integer indicating the size of the gate followed by a list of signals for the respective lines such that the the targets of the gates are at the end of the list. Note that Fredkin gates include SWAP gates as well.

Example: `f3 a b c`

- *Peres gate (P)*

A Peres gate is signified by the character `p` and an integer indicating the size of the gate followed by a list of signals for the respective lines such that the the targets of the gates are at the end of the list.

Example: `p3 a b c`

- *V gate*

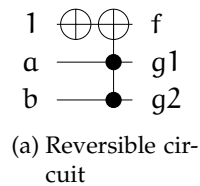
A V gate – one of the elementary quantum gates (EQ) – is signified by the character `v` and an integer indicating the size of the gate (i.e., 2) followed by a list of signals for the respective lines such that the the target line is at the end of the list.

Example: `v2 a b`

- *V+ gate*

A V+ gate – one of the elementary quantum gates (EQ) – is signified by the characters `v+` and an integer indicating the size of the gate (i.e., 2) followed by a list of signals for the respective lines such that the the target line is at the end of the list.

Example: `v+2 a b`



```
.version 2.0
.numvars 3
.variables x0 x1 x2
.inputs 1 a b
.outputs f g1 g2
.constants 1---
.garbage -11
.begin
t1 x0
t3 x1 x2 x0
.end
```

(b) File format

Figure 4: Reversible circuit and its representation in the REAL format

As an example, Figure 4(b) shows the file format for the reversible circuit provided in Figure 4(a).

4.2 Sub-circuits

Existing (sub-)circuits, specified in other REAL files and denoted by modules in the following, can be included to a circuit description. Therefore, the respective module needs to be declared first. This is done in the header with the keyword `module` followed by an identifier and the name of the file to be included. The identifier has to be a sequence of letters, digits, and underscores. Already used names (e.g. “t” for the Toffoli gate) are forbidden. The name of the file needs to be provided in quotation marks.

After the declaration, the module can be applied like gates in the actual circuit description. The list of signals followed by the identifier must be consistent with the signals in the module. Declarations of constant inputs or garbage outputs in the module are thereby overwritten. That is, if e.g. a module works correctly with constant inputs only, the respective constant input needs to be provided at the respective top-level.

Control Lines are specified in the same manner as with other elementary gates. The last n gates are ordered *target gates*, where n is the number of lines of the sub-circuit. All lines specified before are used as *control gates*.

Example:

```
# 2-bit adder
.version 2.0
.numvars 7
.variables x1 x2 x3 x4 x5 x6 x7
.inputs a0 a1 b0 b1 cin 0 0
.outputs g g g g cout sum0 sum1
#constant input declaration consistent to module
.constants -----00
#garbage output declaration consistent to module
.garbage 1111---
```

```
.module adder "1bitadder.real"
.begin
adder x1 x3 x5 x6
adder x2 x4 x5 x7
.end
```

```
# 1-bit adder (1bitadder . real )
.version 2.0
.numvars 4
.variables x1 x2 x3 x4
.inputs a b cin 0
.outputs g g cout sum
.constants ---0
.garbage 11--
.begin
t3 x2 x1 x4
t2 x1 x2
t3 x3 x2 x4
t2 x2 x3
.end
```

4.3 Buses

Often it may be useful to group several input/output signals. Therefore, buses can be defined. An inputbus (outputbus) is specified in the header using the keyword `.inputbus` (`.outputbus`) followed by an identifier and a list of signals to be grouped.

Example:

```
.version 2.0
.numvars 6
.variables x0 x1 x2 x3 x4 x5
.inputs a_0 a_1 a_2 b_0 b_1 b_2
.outputs -- c_0 c_1 c_2 -
.constants -----
.garbage 11---1
.inputbus a x0 x1 x2
.inputbus b x3 x4 x5
.outputbus c x2 x3 x4
.begin
# Gates ...
.end
```

4.4 Sequential Circuits

According to the basic definitions, feedback is not directly allowed in reversible circuits. Nevertheless, first steps towards the realization of sequential reversible circuits have

been performed. Different paradigms are thereby applied (see e.g. [?, ?]). Consequently, the REAL format also supports sequential circuit representations.

State signals are defined like buses, i.e. they start with `.state` followed by an identifier for the bus name and then a list of signals who are grouped in this state signal. Of course, this can only be one signal. In addition to the input and output buses, an initial value can be defined for the state as natural number after the last signal name.

Example:

```
# 1-bit adder (1 bitadder . real )
.version 2.0
.numvars 4
.variables x1 x2 x3 x4
.inputs a b cin 0
.outputs g g cout sum
.constants ---0
.garbage 11--
.state ff1 x1
.state ff2 x2 x3 2 # init . value
.begin
t3 x2 x1 x4
t2 x1 x2
t3 x3 x2 x4
t2 x2 x3
.end
```

4.5 Annotations

Finally, additional remarks can be annotated to a circuit defined in a REAL file. In contrast to comments, annotations are supposed to have a special purpose for programs. Such annotations can be used e.g. to link gates of the synthesized circuit to the corresponding code line of the HDL description from which the circuit has been derived.

Annotations can be added to the circuit using `#@ [id]=[value]`, whereby `[id]` defines an identifier and `[value]` the actual value³. Annotations are always placed after a gate and correspond to that single gate only. Multiple key/value pairs are separated by spaces.

Example:

```
.version 2.0
.numvars 3
.variables a b c
.begin
t3 a b c #@ color=1
t1 c      #@ color=2
t2 a b   #@ color=1 name="Test"
```

³Since `#` denotes a usual comment, annotations are ignored by programs who do not support an explicit handling of them.

```
t2 b c
t2 b c  #@ name="Test"
.end
```

5 Simulation Patterns

In order to pre-define simulation patterns to be applied to a circuit, the SIM format can be applied. Using this format, patterns are specified in an ASCII file including

- the `version`-tag as already applied in the SPEC- and REAL-files,
- a line starting with `.inputs` signifying the names of the respective primary inputs to which a simulation pattern should be applied, as well as
- the input patterns to be simulated (enclosed between a `.begin`- and `.end`-tag).

Natural numbers representing the binary expansion can thereby be applied. For example, the file

```
.version 2.0
.inputs a b
.begin
4 6
.end
```

represents an assignment 100 and 110 to a circuit in case of 3-bit inputs denoted by a and b. If the inputs were of size 4, the assignments would be interpreted as 0100 and 0110. However, if the inputs were of size 2, the assignments would be interpreted as 00 and 10, thus, the bit-vector is truncated.

Accordingly, state signals and sequences of input patterns can be specified. This is necessary, if sequential circuits should be simulated (see Section 4.4). Therefore, an optional line starting with `.init` followed by the name of a state signal as well as the desired initial value can be added (by default all state variables are assumed to be initialized to zero). The sequence of input patterns is specified between the `.begin`- and `.end`-tag. For example, the file

```
.version 2.0
.init c 0
.inputs reset
.begin
0
0
0
1
1
0
0
```

1
.end

initially sets the state variable *c* to zero. Afterwards, eight clock cycles are simulated whereby the primary input *reset* is set to 0, 0, 0, 1, 1, 0, 0, and 1, respectively.

6 Submit Files to RevLib

Researchers are welcome to submit own specifications, functions, or circuits to RevLib. If you wish to publish new benchmarks or synthesis results, please attach the respective file(s) in the proposed file format at an e-mail to

revlib@informatik.uni-bremen.de.

We would appreciate, if you add some few words about your submission. In case of specifications and functions, we are particularly interested in what kind of function is provided. If you submit circuit realizations, we would like to know how you did obtain the result (e.g. which synthesis approach has been applied for that matter).

7 Version History

- Version 2.0 (published December 2010):
 - Support of SyReC descriptions
 - Support of PLA descriptions
 - Extension of the REAL format:
 - * Support of sub-circuits
 - * Support of signal buses
 - * Support of sequential circuits
 - * Support of additional annotations
- Version 1.0 (published May 2008): Initial Version

8 Acknowledgments

We are indebted to the following users for their help in building RevLib, implementing tools and/or submitting new function specifications or realizations:

- Eleonora Schönborn and Bastian Blachetta: For their help in defining the simulation file format.
- Mathias Soeken, Stefan Frehse, and Sebastian Offerman: For the fruitful joint work and their help in writing the documentation.

- The students project “QBit” from the University of Bremen: For the fruitful discussions.
- Jacqueline E. Rice: For her circuit contributions.
- Zach Hamza: For his help in updating RevLib and his circuit contributions.
- D. Michael Miller: For his circuit contributions and his very helpful suggestions to improve the page.
- Mehdi Saeedi: For his circuit contributions.
- Nathan Scott: For implementing the very helpful Quiver.
- Uwe Forgber: For the technical support behind RevLib.
- Dmitri Maslov: For for his suggestions and the permission to take some functions from his private page.

Beyond that, some Boolean functions provided in RevLib have been taken from the LGSynth benchmark library.