EXLIF: Extended Logic Interchange Format Syntax

Describing Circuits for Forte

Table of Contents

1 Introduction

The exlif format is meant for describing circuits to be processed by the Forte/FL system. For all the relevant details on how to compile circuits in exlif for uploading them into Forte please study chapter 9 of the Forte/FL user guide. In this document we will concentrate on the exlif syntax, which is an extension of the Berkeley SIS system's LIF format. LIF is a format that describes combinational circuits as truth tables. For each internal driven signal the LIF holds a truth table, its fan in and its resolution function. The EXLIF holds, in addition, constructs to model sequential elements, like transparent latches and master slave flip-flops. EXLIF format also has constructs for describing structural hierarchy, tri-state drivers, various kinds of assertions etc.

There are some compelling characteristics that makes EXLIF useful:

- *Simplicity:* EXLIF is (intended to be!) unambiguous, simple to read and understand and easy to manipulate. EXLIF uses a minimal number of constructs to specify the digital systems at structural level. EXLIF's simplicity allows for an easy translation from various HDLs.
- *Generality:* EXLIF is capable of describing a large range of synchronous digital systems. It can be used to specify both the control and datapath logic.
- *CAD tool independency:* EXLIF serves as a standard interchange format for formal verification, simulation, synthesis and emulation tools.
- *HDL independency:* EXLIF is hardware design language independent. EXLIF provides a suitable interface to various hardware description languages as VHDL, and Verilog.
- *Abstraction level independency:* EXLIF is suitable to represent the digital systems at both RTL and gate level.

Fundamentally, an exlif file consists of a structural hierarchy, where the hierarchy consists of "models" and each model contains some instantiations of other models, combinational logic, sequential latches/flip-flops and various assertions. Exlif is a command oriented syntax where most commands consist of a line starting with a period (.) followed by a keyword followed by a series of arguments and/or new lines with further commands.

2 Basic syntactic objects

2.1 White space

Space, tab, and new-line characters are all considered white space. It is almost always legal to increase the number of white spaces. The only exception is in the definition of a truth table (more about that later) in which no extra white spaces can be inserted.

2.2 Comments

A line starting with # (hash symbol) is completely ignored. In addition, a white space before a # symbol signals the beginning of a comment and the rest of the line until the line break is ignored.

2.3 Line continuation

A line terminated by two backslashes (\\) is viewed as continuing with the next line (basically we get one single long line).

2.4 Node names

A node name consists of either of a base name of a quoted name. A base name consists of any non white space character except "=", followed by zero or more non white space characters except "=" and "\\". More precisely, the regular expression for a base name is given by:

[^\=\n\ \t][^\\\n\ \t\=]*

A quoted name is simply any sequence of characters enclosed in quotes ("..."). Note that vector names are also possible. A vector name is a base name followed by a square bracket surrounded integer range. For example, here is a collection of valid node (and vector) names:

ab
ab[2]
abc[8:2]
"a strange node name"[3:1]
cdf[2:9]
Note that exlif is case sensitive and thus a node "a" is different from a node "A". (When converting a exlif file into an exe file to be used by Forte, the case sensitivity can be suppressed by giving nexlif2exe the –lower flag.)

2.5 Model names

Model names follow basically the same convention as node names with the exception that vector notation is not supported (since it does not make much sense).

3 Specifying a FUB(Functional Unit Block) in EXLIF

A FUB corresponds to a VERILOG *Module* or a VHDL *Entity*. A FUB declaration has the following structure in EXLIF

```
.model FUB-name
<FUB interface template>
<declaration of signals>
<FUB body>
.end
```

Note that you can have more than one FUB declared inside an exlif file. You can also nest .model declarations. If there is more than one top-level .model, the first one will be viewed as the main model and only models instantiated inside this top-level model will be included in the final exe file. In exlif there is no requirement that a model is declared before being used.

3.1 Declaring a FUB in EXLIF

3.1.1 .model *FUB-name*

A functional block(FUB) is represented by the construct **.model**, when *FUB-name* represents the name of the FUB.

3.1.2 .end

The EXLIF construct **.end** specifies the end of a FUB specification.

3.2 Specifying a FUB interface in EXLIF

3.2.1 .inputs [*inputs*]

The construct **.inputs** specifies the *inputs*, a list of input signals of the model (FUB).For example, the EXLIF statements,

```
.model sample
.inputs in1 in2
```

specify the inputs *in1* and *in2* of FUB *sample*. Signals can either be scalar or vectors. A vector is defined by using square bracket notation, e.g., abc[12:10] denotes a vector of size 3 (with elements abc[12], abc[11] and abc[10]).

3.2.2 .outputs [*outputs*]

The construct **.outputs** specifies the *outputs*, a list of output signals of the model(FUB). For example,

```
.model sample
.outputs out1 out2
```

specify the outputs *out1* and *out2* of FUB *sample*.

NOTE: it is legal to use the same name for an input and an output. This is used to model pass-through wires.

3.2.3 .vector *signal-name upper-bound lower-bound*

The construct **.vector** specifies the signal *signal-name* to be a multi-bit signal. The most and least significant bits of the signal are specified by *upper-bound* and *lower-bound*, respectively. Legal values for *upper-bound* and *lower-bound* are integers. This is followed optionally by a list of sub-fields of the vector that are its atoms.

NOTE: Although .vector is supported, it is strongly suggested to avoid using abbreviated names and instead use explicit wire names with explicit ranges. When doing so, there is really no need to ever use the .vector declaration.

3.3 Specifying attributes in EXLIF

3.3.1 .attr *name [attr-list]*

The construct **.attr** is used to specify the attributes of FUBS, subFUBS, and signals.

?? **name** - the name of the signal, model, or model instance for which an attribute is to be specified.

?? **attr-list** - a list of attributes where each attribute is defined as **attr-name=attr-value** pair.

attr-name specifies the name of the attribute and **attr-value** specifies the value of the attribute. The value of an attribute may be a string or a number. If an attribute name, **attr-name**, is a member of the attribute list and it does not have a value, then the attribute name has to be double quoted. The list has to contain at least a double quoted attribute name or an attribute name and an attribute value pair.

For instance, both the examples below have the same meaning:

```
1. .attr x file="foo.hdl"
.attr x line="15"
2. .attr x file="foo.hdl" line="15"
```

where x is a signal.

Note that there can be more than one line with .attr for the same signal, model or model instance.

3.4 Specifying FUB body in EXLIF

3.4.1 Combinational signals

There are three ways of defining the next state function for a combinational node: .names, .expr, and .bexpr statements. Although all three are available, the .names construct is the oldest and the most heavily used in practice.

3.4.1.1 .names [inputs] output

Each look-up table is specified by the **.names** construct as in BLIF, by taking binary inputs and producing a binary output. The truth table either indicates the on-set or the off-set, depending on whether the value for the output is 1 or 0. If some inputs are vectors, then the output must also be a vector of the same size and the equation is interpreted as a bitwise extension of the truth table.

In the following example, in1, in2, ... are names of input signals to the look-up table (not necessarily primary inputs), and output is the name of the binary-valued output signal produced by this particular look-up table. The subsequent lines define the on or off-sets:: **in1_value** is one of 0, 1, or – (don't care), and **out_value** is one 0 or 1, and so on.

```
.names in1 in2 ... out
in1_value_1 in2_value_1 ... out_value_1
in1_value_2 in2_value_2 ... out_value_2
```

For example, a two-input AND gate with inputs a b and output foo could be defined as:

.names a b foo
11 1
a two input OR gate could either be defined as
.names a b foo
00 0
or as
.name a b foo
1- 1

-1 1

Similarly, a set of 3 2-input NAND gates may be defined as:

```
.names a[2:0] b[4:2] res[12:10]
11 0
or as
.names a[2] b[4] res[12]
11 0
.names a[1] b[3] res[11]
11 0
.names a[0] b[2] res[10]
11 0
```

Finally, if the output is a vector and (possibly) some of the inputs are vectors, the remaining nodes will be duplicated. For example, a two-input MUX can be defined as:

```
.names sel1 v1[3:0] v2[3:0] res[3:0]
11- 1
0-1 1
```

3.4.1.2 .expr output = <expr>

For humans, the .expr syntax is easier to write and read. The operators and the precedence among the operators are as follows:

<expr> :: <expr> & <expr>	AND
|| <expr> + <expr>	OR
|| <expr> ^ <expr>	XOR
|| <expr> ‘	NOT
|| T	true
|| F	false
|| “vector_name[range1:range2]”	vector
|| scalar_name	node name
|| (<expr>)	

Precedence: ‘ > & > ^ > +.

3.4.2 Defining sequential elements

3.4.2.1 .latch latch-input latch-output type control-signal [latch-control-list]

The construct **.latch** specifies a storage element having an input and an output. The value of latch-output is the value latch-input had in the previous state. Latches and flip-flop do not have any initial value.

3.4.2.2 Specifying Latch Types

The constructs **re**, **fe**, **ah**, and **al** specify whether the latch is edge-sensitive or level-sensitive. If a latch is driven only at rising or falling edge of a signal then it is edge-sensitive. If the latch is driven when a signal is active high or active low then it is level-sensitive.

?? **re** control-signal

The construct **re** specifies that the latch is driven at the rising edge of the *control-signal*.

?? **fe** control-signal

The construct **fe** specifies that the latch is driven at the falling edge of the *control-signal*.

?? **ah** control-signal

The construct **ah** specifies that the latch is driven when the *control-signal* is active high.

?? **al** control-signal

The construct **al** specifies that the latch is driven when the *control-signal* is active low.

```
.latch in1 out1 re clk
```

The signal **in1** drives the signal **out1** on the rising edge of the signal **clk**.

```
.latch in1 out1 fe clk
```

The signal **in1** drives the signal **out1** on the falling edge of the signal **clk**.

```
.latch in1 out1 ah clk
```

The signal **in1** drives the signal **out1** if the signal **clk** is active high.

```
.latch in1 out1 al clk
```

The signal **in1** drives the signal **out1** if the signal **clk** is active low.

3.4.2.3 Latch control constructs

?? **as**=signal-name

The construct **as** specifies the asynchronous set signal of the latch.

?? **ar**=signal-name

The construct **ar** specifies the asynchronous reset signal of the latch.

?? **en**=signal-name

The construct **en** specifies the enable signal of the latch.

For example, the EXLIF statement below,

```
.latch in1 out1 re clk as=set ar=reset en=en1
```

specifies a latch which is driven at the rising edge of signal **clk** with an input signal **in1**, an output signal **out1**, an asynchronous **reset** signal reset and a asynchronous set signal **set**.

3.4.3 .subckt *template-name instance-name [interface-list]*

The **.subckt** statement in EXLIF represents a model instantiation statement where: **template-name** is the template name of the instance,

instance-name is the instance name of the instance, and

`interface-list` is the list of actual signal and interface signal pairs when each pair is defined as **`formal-name=actual-name`**.

3.4.4 .nondet *signal-name*

The construct .nondet specifies a non-deterministic signal in the model, i.e. the signal has random value.

3.4.5 Specifying Transition relations in EXLIF

In order to represent transition relations in the exlif a **.trans** construct is used like this:

.trans ***<transition-name> <signal-name> <transition-type>***

Where <transition-name> is the name assigned to the transition. <signal-name> is the name of the signal whose transition is described, and <transition-type> is the type of the transition being described. Currently <transition-type> can be one of the following:

“high-before”, “high-after”, “low-before”, “low-after”, “rise”, “fall” , “change”, “stable”.

4 Simple example of exlif file

```
.model top
.inputs a b v1[2:0] v3[2:0] clk
.outputs out1 out2 vres[2:0]
.expr out1 = a & b
.expr “vres[2:0]” = “v1[2:0]” & a + “v2[2:0]”’
.names a b tmp
00 0
.subckt bar bar1 clk=clk in[1:0]=v1[2:1] out[1:0]=tmp2[1:0]
.subckt bar bar2 clk=clk in[1:0]=v3[1:0] out[1:0]=tmp3[1:0]
.names tmp2[1] tmp2[0] tmp3[1] tmp3[0] out2
11-- 1
--11 1
.end

.model bar
.inputs clk in[1:0]
.outputs out[1:0]
.latch in[1:0] out[1:0] re clk
.end
```