An Implementation of Lola-2
or
Translating from Lola to Verilog

N.Wirth, 30.11.2014

1. Introduction

The hardware description language Lola (Logic Language) was designed in 1990 as an effort to present a simple and effective textual description of digital circuits. At that time, the conventional style was still graphical (circuit charts), and it was not evident that textual descriptions would replace them entirely within 20 years. Also, there were no means available to automatically transfer them into physical circuits of electronic components.

However, field-programmable gate arrays (FPGA) appeared, and although they were far too restrictive (small) for most practical purposes, they seemed to be a promising gateway towards introducing textual specifications with the hope of future automatic generation of real circuits. That this hope was well-founded is now evident.

The difficult part of implementation in 1990 was not the compilation of the textual descriptions into net lists of gates and wires. It was rather the placement of components and routing of wires. And this still remains so. But even if this task is achieved, the compiled output is to be down-loaded into the FPGA. For this purpose, the format of the data, the bit-stream format, must be known. Whereas at the time we obtained this information from two FPGA manufacturers, it is now strictly proprietary in the case of the dominating manufacturers, a severe case of interface secrecy.

In the course of reviving activities of 25 years ago around Oberon, also the hardware description language (HDL) Lola reappeared. Now textual descriptions of hardware are common place, the preferred languages being Verilog and VHDL. With the view of a teacher, I have the same grave objections against them as for the wide-spread programming languages: they are by far too complex. Thereby they make it difficult and pitfall-prone for beginners, apart from causing headaches for teachers. I felt that Lola could play the same role for HDLs as Oberon for PLs, easing learning for beginners and relieving teachers from headaches.

The solution to make Lola also practically usable now, was not to strive for a straight translation into FPGA configurations, but to translate Lola into Verilog (or VHDL), in the same way as many PLs are translated not into binary code, but into the language C. This way, unfortunately, does not let us avoid the usage of Verilog (or VHDL) compilers with all their idiosyncracies, but it lets us rely on far stricter (type) checking and on restriction to "safe" constructs. But the main advantage offered is the better structured language with fewer and simpler design rules.

Another goal was to exhibit the similarities of HDLs and PLs. Common are the concepts of declarations and assignments, of variables, expressions and assignments. Module types take over the role of procedure declarations, and instantiations that of procedure calls. Recursion is evidently impossible. There are also genuine, inherent differences. Whereas the principal idea of programs (software) was sequential execution, i.e. reuse of (hardware) facilities for every consecutive instruction, the intrinsic property of hardware is parallelism, the concurrent activity of all circuit elements.

As HDLs express static designs, every variable can be attributed (assigned) a value exactly once. This makes HDLs amenable to the functional style. Restricting our view to *synchronous* circuits, the concept of time is eliminated like in PLs (in the first place). Time reappears in the difference between variables and registers. The latter represent the input value assigned in the *previous* clock cycle, whereas variables appear only in the role of names for expressions.

1. The language Lola-2

This HDL is structured in the style of Oberon, a descendant of Algol and Pascal. It features declarations of constants, types, variables and registers. Expressions contain logical and arithmetic operators and relations. Expressions are assigned to variables and registers by statements. But unlike in Oberon, all assignments are static, and their order is irrelevant. The language is defined in https://www.inf.ethz.ch/personal/wirth/Lola/Lola2.pdf

There are two constructs in Lola which have no counterparts in Oberon; They are the *constructor* and the *range*, and they are inspired by Verilog. Both form sequences of bits (bitstrings), and they breach the otherwise strict typing rules. The range denotes a subsection of a bitstring variable. If *x* is a bitstring (of type *[N] BIT, x[n:m]* denotes the subrange of bits *x[n] ... x[m]* with a length of *n-m+1*. (n >= m). A constructor consists of a sequence of elements (fields), each denoting a bit sequence. An example, where *x* and *y* are variables (of 8 bits each), is

```
{x, y[3:0], 10'4, x!2}
```

It has a length of 8 + 4 + 4 + 2*8 = 32. 10'4 denotes the integer 10 represented by 4 bits (4'b1010 in Verilog), and !2 indicates that x is to occur twice. The high-order fields are listed first. Integers within constructors must always be followed by a length indication.

Two brief examples show the style of Lola: The first is a 4-bit binary counter expressed solely by logical expressions. The second is also a 4-bit counter, but with reset and enable signals and expressed using the addition operator. The input *clk* is by convention used to drive the registers.

```
MODULE Counter0 (IN clk: BIT; OUT d: [4] BIT);
  REG R: [4] BIT;
BEGIN
  R := {R.3 ^ R.3 & R.2 & R.1 & R.0,      (*R.3*)
        R.2 ^ R.2 & R.1 & R.0,            (*R.2*)
        R.1 ^ R.1 & R.0,                  (*R.1*)
        ~R.0};                            (*R.0*)
  d := R
END Counter0.

MODULE Counter1 (IN clk, rst, enb: BIT; OUT d: [4] BIT);
  REG R: [4] BIT;
BEGIN
  R := rst -> 0 : enb -> R + 1 : R;
  d := R
END Counter1.
```

Lola allows to specify modules in the form of types. They can be instantiated as variables. If, for example, a type is declared as

```
TYPE Counter  := MODULE (IN clk, rst, enb: BIT; OUT d: [4] BIT);
  REG R: [4] BIT;
BEGIN
  R := rst -> 0 : enb -> R + 1 : R;
  d := R
END Counter
```

the variable declarations

```
C0, C1, C2: Counter
```

instantiate three such counters. They are assigned "values" by assignments such as

```
C0(clk, rst, enb0); C1(clk, rst, enb1); C2(clk, rst, enb2)
```

where *clk, rst, enb0, enb1, enb2* are declared variables.

As an aside, the type declaration is in analogy to the procedure (type) declaration in programming languages, and the instantiations to procedure calls. This facility allows the construction of arrays

and even matrices of modules, implying the replication of circuits upon compilation. A top-level module appears as a module type declaration merged with a single instance.

The generator facility of Verilog has been omitted from Lola-2 (although it was present in Lola). Whether this is an asset or a hindrance is unclear. In any case, it was no impediment to expressing a complete processor and its environment .

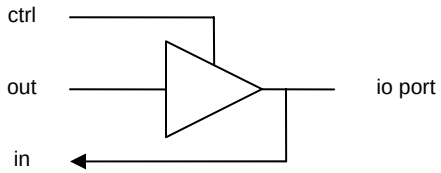
https://www.inf.ethz.ch/personal/wirth/ProjectOberon/PO.Computer.pdf

There is a single generic statement in Lola. It expresses a tri-state port. Early designs of FPGAs contained tri-state gates as circuit elements. In more recent desihns, they have been elininated (except for ports), because erroneous programming could lead to physical destruction of the entire FPGA due to short circuits. A tri-state port *io* is specified by the symbol INOUT in the module's parameter list. Ihe connections are specified by the statement

TS(io, in, out, ctrl)

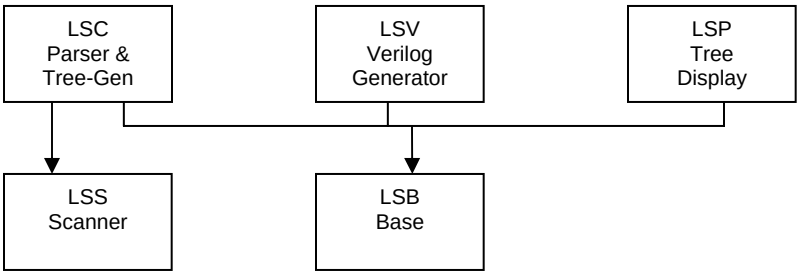
where

io	tri-state parameter,
in	input to the circuit from the port
out	output from the circuit to the port
ctrl	control: 0 for input, 1 for output



3. The Lola-2 compiler

The design of the Lola translator follows quite strictly the principles described in *Compiler construction* (http://www.inf.ethz.ch/personal/wirth/CompilerConstruction/index.html). The main module (LSP) is the parser. It relies on the scanner (LSS) translating input character sequences into language symbols. The parser, a top-down, recursive-descent algorithm, requests symbols from the scanner by calling procedure *Get*. The output, generated while reading source text, is not another text, but a binary tree. A generator module (LSV), described later, subsequently traverses this tree.



The relevant commands are *LSC.Compile @* and *LSV.List filename*.

The structure of the elements of the tree is defined in a base module (LSB). This is in order to let modules generating translations access this tree without having to refer to the parser. The tree's elements (nodes) are called *Items*. Elements that carry a name are defined as *extensions of Items*, and they are called *Objects*. Every item has a type defined by *Type* and its extensions *ArrayType* and *UnitType*. There is the basic type BIT, from which arrays and arrays of arrays can be constructed. The list of all declared objects is anchored in the global variable *root*.

```
TYPE Item = POINTER TO ItemDesc;
     Object = POINTER TO ObjDesc;
     Type = POINTER TO TypeDesc;
     ArrayType = POINTER TO ArrayTypeDesc;
     UnitType = POINTER TO UnitTypeDesc;

     ItemDesc = RECORD
        tag: INTEGER;
        type: Type;
        val, size: INTEGER;
        a, b: Item
     END ;

     ObjDesc = RECORD (ItemDesc)
        next: Object;
        name: ARRAY 32 OF CHAR;
        marked: BOOLEAN
     END

     TypeDesc = RECORD len, size:  INTEGER; typobj: Object END ;
        ArrayTypeDesc = RECORD (TypeDesc) eltyp: Type END ;
        UnitTypeDesc = RECORD (TypeDesc) firstobj: Object END ;

     VAR root: Object;
```

New items and objects are generated by the function procedures *New(tag, a, b)* and *NewObject(class)*. The main attributes of an item are a tag, typically representing an operator, and a data type. The item's branches are *a* and *b*. The additional attributes of objects are their *name* and a link *next* which is used to form lists of variables and registers.

An item is generated for each occurrence of an operator. For example, the parser routine *SimpleExpression,* (here simplified) analyzes two terms *x* and *y*, and calls *New*:

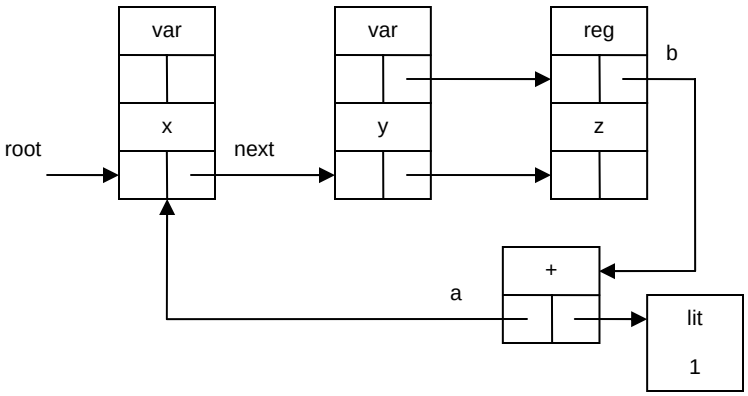
```
PROCEDURE SimpleExpression(VAR x: LSB.Item);
   VAR y, z: LSB.Item;
BEGIN ... term(x);
   WHILE sym = LSS.plus DO
      LSS.Get(sym); term(y); z := New(add, x, y); CheckTypes(x, y, z); x := z
   END
END SimpleExpression;
```

Tag values are

```
const = 1;  typ = 2; var = 3; lit = 4; sel = 7; range = 8; cons = 9;
repl = 10; not = 11; and = 12; mul = 13; div = 14; or = 15; xor = 16; add = 17; sub = 18;
eql = 20; neq = 21; lss = 22; geq = 23; leq = 24; gtr = 25;
then = 30; else = 31; next = 32;
```

As an example, the following simple piece of text is translated into the tree shown below.

```
MODULE M (IN x: BIT; OUT y: BIT);
   REG z: BIT;
BEGIN z := x + 1; y := z
END M.
```



Apart from parsing the source text and generating the tree, a primary task of module LSP is type checking. Every item and object has a data type, and for each operation the compatibility of the operand types must be verified and the result type determined. For relations the result type is always BIT. Checking for compatibility is done by *CheckTypes(x, y, z)* in expressions, and by *CheckAssign(x, y)* for assignments.

These two routines are relatively complex, althouth the rules are quite simple. A complication arises from the circumstance that the types of a varibale and the size (no. of elements) are stored in the associated type descriptor. However, every integer (literal), every constructor, and every range intrinsically defines its own - not explicitly declared - type. As we do not wish to allocate a specific type descriptor for each such occurence, the type information (size) is stored in the item representing the variable itself. This makes it necessary to access the size in a different - more direct - way in the case of integers, constructors, and ranges, requiring case discriminations in the checking routines.

The module type construct is not completely implemented by this translator, but rather tailored to the target language Verilog, which does not feature local modules, but instead imports sparately declared modules. Here we simply replace a module's body by an arrow sign (^). For example:

```
MODULE Counter0 (IN clk: BIT; OUT d: [4] BIT) ^ ;
```

This heading suffices to generate instantiations with proper type checking.

As a consequence of this compiler structure, the generation of target code is entirely separated from the parser. Actually, we first designed a module (LSP) visualizing the tree, and only later a module (LSV) producing Verilog text. This method eases the construction of generators for other target languages. Here are shown the results of translating the two sample modules shown at the beginning of chapter 1.

```
`timescale 1ns / 1 ps
module Counter0(   // translated from Lola
input clk,
output [3:0] d);
reg [3:0] R;
assign d = R;
always @ (posedge clk) begin
R <= {(R[3] ^ (((R[3] & R[2]) & R[1]) & R[0])), (R[2] ^ ((R[2] & R[1]) & R[0])), (R[1] ^ (R[1] & R[0])), ~R[0]};
end

endmodule

`timescale 1ns / 1 ps
module Counter1(   // translated from Lola
input clk, rst, enb,
output [3:0] d);
```

```
reg [3:0] R;
assign d = R;
always @ (posedge clk) begin
R <= rst ? 0 : enb ? (R + 1) : R;
end
endmodule
```

The generator procedure *LSV.List* traverses the list of declared objects, starting from the global variable *root*, three times. In the first pass procedure *ObjList0* visits variables (including parameters) and outputs their declarations in the syntax of Verilog. For this purpose, type information is accessed.

In the second pass, assignments to variables are processed. For simple variables and arrays, the form *v := x* is converted into Verilog's *assign v = x;* Also, module instantiations are processed. For example, given variable declarations

```
clock, reset, enable: BIT; data [4] BIT;
```

the Lola instantiation

```
C0 (clock, reset, enable, data)
```

is transposed into the Verilog statement

```
Counter C0 (.clk(clock), .rst(reset), .enb(enable), .d(data))
```

Thereby type compatibility between formal and actual parameters is checked and enforced in the same way as for assignments.

In the third pass, assignments to registers are handled. If no clock is explicitly specified, a default variable or parameter *clk* is assumed. For exmple, the assignments

```
R0 := x; R1 := y
```

are converted into

```
always @ (posedge clk) begin
  R0 <= x; R1 <= y;
end
```

If a clock is specified, it appears in the always clause.. For example, given the declaration

```
REG (clk50) R
```

the assignments are converted into

```
always @ (posedge clk50) begin
  R0 <= x; R1 <= y;
end
```

The structure of module LSV is less regular than one might wish. This is mostly due to some peculiarities of the syntax of Verilog. For example, the declaration of a matrix A, in Lola simply specified as *A: [m] [n] BIT*; is declared in Verilog as [n-1:0] A [m-1:0], requiring the last dimension to be treated differently from the others. Another example is that parameter lists use commas to separate elements, whereas in variable declaration semicolons are used.

To show the viability of Lola, the entire RISC processor, including its environment and device interfaces, has successfully been expressed in Lola:

(www.inf.ethz.ch/personal/wirth/Lola/index.html)