blooRTLs Reference Manual

Peter Burrows, Zhibo (Andy) Wan, Apurv Gaurav, Pinhong He
phb2114, zw2327, ag3596, ph2482
EE/CE Department, Columbia University

June 15, 2015

1 Introduction

The Behavioral Language for Object-Oriented Register Transfer Level Specs (blooRTLs) is a high-level RTL desription language. The language is packaged with an interpreter that translates blooRTLs source code to synthesizable sequential VHDL for RTL designs. The blooRTLs interpreter also builds a test bench to simulate the VHDL code.

While blooRTLs does not embody the structure of an established "object-oriented" language, it does use the notion of objects (as described in Section 5), and in a general sense of the "object-oriented" style, blooRTLs does contribute a modular approach to the RTL design domain.

In regards to the syntax of blooRTLs, the interpreter assumes that a reasonable clock frequency is to be used for the desired RTL spec. As a result, the complexity and types of arithmetic operations that can be performed within a given cycle has been limited by the blooRTLs syntax. Just as an RTL implementation that uses a reasonable clock frequency (i.e. MHz range) cannot load a register with data that has propagated through a substantial number of gates within the same clock cyle, blooRTLs does not permit multiple arithmetic operations to be performed within the same assignment expression. The legal expressions are discussed further in Section 7.

Overall, the modular and syntactical styles of blooRTLs offer an efficient and effective means of developing syntehsizable bit manipulation methods and RTL algorithms. The following reference manual borrows the framework of Dennis Ritchie's *C Reference Manual* to afford readers a better understanding of blooRTLs [1].

2 Lexical Conventions

The lexical conventions implemented in blooRTLs consist of comments, identifiers, keywords, and constants.

2.1 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits used to represent a register in the RTL design; the first character must be alphabetic. The underscore “_” counts as alphabetic. Upper and lower case letters are considered different.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

`NOT`	`XNOR`	`binmap`
`AND`	`CLOCK`	`repeat`
`NAND`	`INPUTS`	`until`
`OR`	`OUTPUTS`	`if`
`NOR`	`DEFINE`	`else`
`XOR`	`binary`	`function`

2.4 Constants

There are two types of constants, as follows:

- **Binary Constants**: A binary constant is a series of ones (1) and zeros (0).
- **Integer Constants**: An integer constant is a series of numerical digits (0-9) without a preceding sign character.

3 Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in the `typewriter` style.

4 What's in a Name?

blooRTLs suports one fundamental identifier type, which takes the form of the binary constant; hence, declaring indentifiers of the fundamental type is prohibited by blooRTLs due to redundancy. The remaining types are described in this section.

- `DEFINE`: the `DEFINE` type transposes the value of a particular fundamental type to the form of a name.
- `function`: Identifiers can be defined by this derived type which can be constructed using any or all of the blooRTLs types
- `binmap`: the `binmap` type creates objects, as defined in Section 5, from identifiers

The following types are used to build the VHDL test bench:

- `CLOCK`: the `CLOCK` type is used to set the clock frequency
- `INPUTS`: the `INPUTS` type is used to set the inputs
- `OUTPUTS`: the `OUTPUTS` type is used to set the outputs

5 Identifiers and Objects

Identifiers and Objects are related and defined, as follows:

- *identifier*: An identifier is interpreted as a binary constant that is assumed to be loaded into a region of storage within the RTL design (i.e. a register)
- *identifier.object*: An object is an indexed component of the Identifier.

Since identifiers are used to represent storage components within the data path of the RTL design, all identifiers are technically objects. However, identifiers and objects are not fully equivalent structures. When identifiers are segmented into smaller regions of storage, these resulting components are referred to as the objects. The means of partitioning identifiers into objects is discussed in Section 8.

6 Conversions

blooRTLs has one conversion operator:

- `binary`: the binary operator converts an integer constant value to a binary constant value. A trivial example is provided, as follows:

```
/* Load the integer value of 9 into data */
data := binary(9);
/* data now holds the integer to binary converted value of 1001 */
```

This single `binary` conversion operator is sufficient for creating a multitude of RTL designs using blooRTLs. In fact, a variety of conversion operators were intentially omitted from the language, as it was the intent of blooRTLs to avoid incurring any limitations on the scope of RTL algorithms and applications that could be implemented.

A floating point converter, for example, was omitted because blooRTLs prefers to let its users decide on the formatting and arithmetic methodologies of these complex data types, which is possible through the `binmap` declaration specifier, as described in Section 8.

7 Expressions

7.1 Primary Expression

primary-expression :
  identifier
  constant
  (expression)

- An *identifier* is a primary expression, provided it has been suitably declared as discussed in Section 8.
- A *constant* is either a binary or an integer constant, as specified in Section 2.4.
- An *(expression)* is an expression that takes the highest precedence.

7.2 Unary Expression

Unary expressions group left to right and are listed as follows

expression `+` *expression* : binary constant add
expression `-` *expression* : binary constant add
expression `>>` *expression* : binary constant shift right
expression `<<` *expression* : binary constant shift left
expression `NOT` *expression* : binary constant bitwise `NOT`
expression `AND` *expression* : binary constant bitwise `AND`
expression `NAND` *expression* : binary constant bitwise `NAND`
expression `OR` *expression* : binary constant bitwise `OR`
expression `NOR` *expression* : binary constant bitwise `NOR`
expression `XOR` *expression* : binary constant bitwise `XOR`
expression `XNOR` *expression* : binary constant bitwise `XNOR`
expression `::` *expression* : binary constant bitwise concatanation
expression `:` *expression* : binary constant bitwise concatanation increment operator

7.3 Assignment

identifier `:=` *expression*

7.4 Relational

expression `=` *expression* : returns boolean `true` if equal
expression `!=` *expression* : returns boolean `true` if not equal
expression `>` *expression* : returns boolean `true` if greater than
expression `<` *expression* : returns boolean `true` if less than
expression `>=` *expression* : returns boolean `true` if greater than or equal to
expression `<=` *expression* : returns boolean `true` if less than or equal to
expression `&` *expression* : returns boolean `true` if both *expressions* are `true`
expression `||` *expression* : returns boolean `true` if the leftmost *expression* or the rightmost *expression* is `true`

8 Declarations

Declarations take the general form indicated, as follows:

> *declaration:*
>   *decl-specifiers declarator-list*

The fundamental type, the binary constant, is prohibited from declaration due to its redundancy (blooRTLs infers binary constants). The remaining declaration specifiers are listed, as follows:

decl-specifiers:

```
DEFINE
CLOCK
INPUTS
OUTPUTS
function
binmap
```

declarator-list:
  declarator
  declarator, declarator-list

declarator:

```
identifier
identifier.object
declarator( )
(declarator)
```

8.1 Identifier

An *identifier* can only assume the form of a binary constant. Hence, integer constants must first be converted into a binary constant, via the `binary` conversion call, before being declared as an identifier. Likewise, `function` calls must return a binary constant if being used by an*identifier*. For example, one who wishes to identify the integer, 42, with a variable, called `var` could write the following code:

```
var42 := binary(42); /* var42 holds the integer ‘‘42’’ as a binary
    constant */
```

8.2 DEFINE

The `DEFINE` *decl-specifier* transposes words with a binary constant. An example is provided, as follows:

```
DEFINE nine 1001;
var := nine;
/* var holds the binary constant value of 1001 */
```

Another useful usage of the `DEFINE` *decl-specifier* can be observed, as follows:

```
DEFINE A binary(65);
char_var := A;
/* char_var holds the binary constant value of 1000001, which is
   the ASCII integer representation of 'A' */
```

8.3 Test Bench Declaration Specifiers

The test bench *decl-specifiers* must be included in every blooRTLs program; otherwise, the blooRTLs translator will fail to build the desired VHDL files. In addition to providing waveform test vectors, the bench *decl-specifiers* help the translator map the input and output ports for the VHDL output files. A trivial “increment by 1” example helps exhibit how these test bench parameters can be implemented in blooRTLs:

```
/* Label the clock as ``clk'' and set its clock cycle period to 8
   us */
CLOCK clk binary(8);

/* Declare the name of the input variable as ``input_var'' and set
    its test vector to 000 */
INPUTS input_var 000;
/* Declare the name of the output variable as ``output_var'' */
OUTPUTS output_var;

output_var := input_var+1;
/* data will load the binary constant value of 001 on the first
   falling edge of the clock (or after 1 period [= 8us] have
   elapsed) */
```

8.4 function

The `function` *decl-specifier* takes public arguments and returns a value of type binary constant or null. Overall, the purpose of the `function` call is to scale and modularize complex algorithms. A simple example, which multiplies an input by 2, adds 1 to that product, and stores the incremented product into the output, is seen, as follows:

```
in_var := 01;
out_var := multiplyby2_and_add1(in_var);
/* out_var now holds the value 11 */

function out = multiplyby2_and_add1(in) {
        temp := IN;
        temp := temp<<1;
        out := temp+1;
}
```

Alternately, this `function` call can be modified to optimize the RTL design. For example, the variable "temp" can be preallocated outside of the function and on the same clock cycle that "in_var" is loaded. This can be seen in the modified example, as follows:

```
in_var := 01;
temp := 01;
out_var := multiplyby2_and_add1(in_var);
/* out_var now holds the value 11 */

function out = multiplyby2_and_add1(in,temp) {
        temp := temp<<1;
        out := temp+1;
}
```

The aforemented `function` call examples illustrate how blooRTLs programmers must be strategic with their variable declaration. At the RTL level, the slightest modifications in blooRTLs can save tremendous amounts of dynamic power in the long run.

`function` *decl-specifiers* may also return null values since the unspecified identifiers within each *function* are inherently public. Recreating the same `multiply_and_add1()` function with a null return, `[]`, is demonstrated as follows:

```
multiplyby2_and_add1();
/* out_var now holds the value 11 */

function [] = multiplyby2_and_add1() {
        in_var := 01;
        in_var := in_var<<1;
        out_var := in_var+1;
}
```

8.5 binmap

The `binmap` *decl-specifier* creates an object within an identifier, which enables usage of the `identifier.object` *declarator*. This idea is illustrated in the following example:

```
/* Within the identifier ``var'', binmap is used to declare an
   object called ``LSB'' that indexes bit location 0, and ``MSB,''
    which indexes bit location 4 */
binmap var{
        LSB := 0;
        MSB := binary(4);
}

var := 1000;
var.LSB := 1;
var.MSB := 0;
/* var now holds the binary value 0001 */
```

9 Statements

Statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the following form:

> *expression;*

Usually expression statements are assignments or function calls.

9.2 Compound statement

The use of several statements, where one is expected, is noted as follows:

> *compound-statement:*
>   *statement*
>
> *statement-list:*
>   *statement*
>   *statement statement-list*

9.3 Conditional statement

The two forms of the conditional statement are listed as follows:

`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*

9.4 repeat until statement

The `repeat until` takes the following form:

`repeat` (*expression*) `until` (*expression*)

10 Scope

blooRTLs source text must be kept in a single file.

References

[1] Ritchie, Dennis M. *C Reference Manual*, Bell Telephone Laboratories, Murray Hill, New Jersey 07974 1973.