Functional Bus Description Language

Revision 0.0

4 January 2023

Abstract

keywords: bus interface, code maintenance, computer languages, control interface, design automation, design verification, documentation generation, electronic design automation, EDA, electronic systems, Functional Bus Description Language, FBDL, hardware design, hardware description language, HDL, hierarchical register description, memory, programming, register addressing, register synthesis, software generation, system management

Table of Contents

Participants

MichaÅ Kruszewski, *Chair, Technical Editor, mkru@protonmail.com*

1. Overview

1.1. Scope

This document specifies the syntax and semantics of the Functional Bus Description Language (FBDL).

1.2. Purpose

This document is intended for the implementers of tools supporting the language and for users of the language. The focus is on defining the valid language constructs, their meanings and implications for the hardware and software that is specified or configured, how compliant tools are required to behave, and how to use the language.

1.3. Motivation

Describing and managing registers can be a tedious and error-prone task. The information about registers is utilized by software, hardware and verification engineers. Typically a specification of the registers is designed by the hardware designer or system architect. During the design and implementation phases it changes multiple times due to different reasons such as bugs, requirement changes, technical limitations, etc. A simple change in a single register may imply adjustments in both hardware and software. These adjustments cost money and time.

Several formal and informal tools exist to address issues related with registers management. However, they all share the same concept of describing registers at very low level. That is, the user has to implicitly define the registers layout. For example, in case of register containing multiple statuses, its user responsibility to specify the bit position for every status.

The FBDL is different in this terms. The user specifies the functionality of the elements that must be placed in the registers. The register layout is generated based on the functional requirements of the elements. Such an approach allows to generate much more hardware description and software code than classical approach. Not only the register masks, addresses, and single read, write functions can be generated, but complete custom functions with optimized access methods. This in turn leads to shorter design iterations and fewer bugs.

1.4. Word usage

The terms "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may", and "optional" in this document are to be interpreted as described in the IETF Best Practices Document 14, RFC 2119.1.

1.5. Syntactic description

The formal syntax of the FBDL is described by means of context-free syntax using a simple variant of the Backus-Naur Form (BNF). In particular:

a) Lowercase words in `constant-width` font, some containing embedded underscores, are used to denote syntactic categories, for example:

```
single_import_statement
```

Whenever the name of a syntactic category is used, apart from the syntax rules themselves, underscores are replaced with spaces thus, "single import statement" would appear in the narrative description when referring to the syntactic category.

b) Boldface words are used to denote keywords, for example:

mask

Keywords shall be used only in those places indicated by the syntax.

c) A production consists of a left-hand side, the symbol "::=" (which is read as can be replaced by), and a right-hand side. The left-hand side of a production is always a syntactic category, the right-hand side is a replacement rule. The meaning of a production is a textual-replacement rule. Any occurrence of the left-hand side may be replaced by an instance of the right-hand side.

d) A vertical bar (|) separates alternative items on the right-hand side of a production unless it occurs immediately after an opening brace, in which case it stands for itself, for example:

```
decimal_digit ::= zero_digit | non_zero_decimal_digit
choices ::= choice { | choice }
```

In the first instance, an occurrence of decimal digit can be replaced by either zero digit or non zero decimal digit. In the second case, "choices" can be replaced by a list of "choice", separated by vertical bars, see item f) for the meaning of braces.

e) Square brackets [ ] enclose optional items on the right-hand side of a production. For example, the following two productions are equivalent:

```
parameters_list ::= ([parameters])
parameters_list ::= () | (parameters)
```

Note, however, sometimes square brackets in the right-hand side of the production are part of the syntax. In such cases bold font is used.

f) Braces { } enclose a repeated item or items on the right-hand side of a production. The items may appear zero or more times.

g) The term *declared identifier* is used for any occurrence of an identifier that already denotes some declared item.

2. References

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

- IETF Best Practices Document 14, RFC 2119,
- IEEE Std 754™-2019, IEEE Standard for Floating-Point Arithmetic.

3. Lexical elements

FBDL has following types of lexical tokens:

- comment,
- identifier,
- indent,
- keyword,
- literal,
- newline.

3.1. Comments

There is only a single type of comment, a *single-line comment*. A single line comment starts with '#' character and extends up to the end of the line. A single-line comment can appear on any line of a FBDL file and may contain any character. The presence or absence of comments has no influence on whether a description is legal or illegal. Their sole purpose is to enlighten the human reader.

3.2. Identifiers

3.3. Keywords

Keywords cannot be used as identifiers.

FBDL has following keywords: **atomic, bus, const, default, doc, false, func, import, mask, memory, package, param, range, return, static, stream, true**.

3.4. Literals

3.4.1. Number literals

```
underscore ::= _
zero_digit ::= 0
non_zero_decimal_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
decimal_digit ::= zero_digit | non_zero_decimal_digit
binary_base ::= 0B | 0b
binary_digit ::= 0 | 1
octal_base ::= 0O | 0o
octal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex_base ::= 0X | 0x
hex_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | a | B | b | C | c | D | d
| E | e | F | f
```

3.4.2. Integer literals

```
decimal_literal ::= non_zero_decimal_digit {[underscore] decimal_digit}
binary_literal ::= binary_base binary_digit {[underscore] binary_digit}
octal_literal ::= octal_base octal_digit {[underscore] octal_digit}
hex_literal ::= hex_base hex_digit {[underscore] hex_digit}
```

3.4.3. Real literals

The real literals shall be represented as described by IEEE Std 754, an IEEE standard for double-precision floating-point numbers.

Real numbers can be specified in either decimal notation (for example, 17.83) or in scientific notation (for example, 13e8, which indicates 13 multiplied by 10 to the eighth power). Real numbers expressed with a decimal point shall have at least one digit on each side of the decimal point.

3.4.4. String literals

3.4.5. Bit string literals

We will see if there is a need for for bit string literal.

4. Data types

There are 4 data types in FBDL:

- bool,
- integer,
- real,
- string.

Types are implicit and are not declared. The type of the value evaluated from an expression must be checked before any assignment or comparison. If there is a type mismatch that can be resolved with implicit rules, then it should be resolved. In case of type mismatch that cannot be resolved, an error must be reported.

Conversion from integer to real in expressions is implicit. Conversion from real to integer must be explicit, and must be done by calling any function returning integer type, for example `ceil(), floor()`. Conversion from integer to bool in expressions is implicit. Bool cannot be converted to any type.

4.1. Bool

The value of bool type can be created explicitly using `true` or `false` literals. The value of bool type shall be implicitly converted to the value of integer type in places where the value of integer type is required. The boolean `false` value shall be converted to integer value 0. The boolean `true` value shall be converted to integer value 1. In the following example, the value of `I1` evaluates to 1, and the value of I2 evaluates to 2.

```
const B0 = false
const B1 = true
const I1 = B0 + B1
const I2 = B1 + B1
```

The bool - integer conversion is asymmetric. Implicit conversion of a value of the integer type to a value of the bool type is forbidden. This is becuase values of the bool type are often used to count the number of elements or to arbitrarily enable/disable an element generation. However, a value of the integer type appearing in a place where a value of the bool type is required is usually a sign of a mistake. To convert a value of the integer type to a value of the bool type the built-in **`bool`**`()` function must be called.

5. Expressions

An expression is a formula that defines the computation of a value.

5.1. Operators

FBDL operators

Operator token	Name	Name
+	foo	bar
*	foo	bar

5.2. Functions

The FBDL does not allow defining custom functions for value computations. However, FBDL has following built-in functions:

`bool``(x integer) bool`
: Bool returns a value of the bool type converted from a value x of the integer type. If x equals 0, then the `false` is returned. In all other cases the `true` is returned.

`ceil``(x float) integer`
: Ceil returns the least integer value greater than or equal to x.

`floor``(x float) integer`
: Floor returns the greatest integer value less than or equal to x.

`log2``(x float) integer|float`
: Log2 returns the binary logarithm of x.

`log10``(x float) integer|float`
: Log10 returns the decimal logarithm of x.

6. Functionalities

Each element property must have concrete type.

6.1. Block

6.2. Config

6.3. Mask

6.4. Memory

6.5. Param

6.6. Proc

6.7. Return

6.8. Static

6.9. Status

6.10. Stream

7. Scope and visibility

7.1. Import and package system

7.2. Scope rules

The following elements define a new scope in the FBDL:

- package,
- functionality definition,
- functionality anonymous instantiation.