Functional Bus Description Language

Revision 0.0

11 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

Michal 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 functionalities that must be provided by the registers. The register layout is generated based on the functional requirements. 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. Concept

The concept behind the FBDL is based on the fact, that if there is a part with the registers that can be accessed, then there is at least one more part accessing these registers. The part acessing the registers is called the *requester*. The part containing the registers is called the *provider* as it provides functions via particular functionalities.

The code generated from the FBDL description can be conceptually divided into two parts, the requester part and the provider part. The requester code usually refers to the generated software or firmware implemented in typical programming languages such as: Ada, C, C++, Go, Java, Python, Rust etc. The provider code usually refers to the generated gateware or hardware implemented in hardware description languages or frameworks such as: VHDL, SystemVerilog, SystemC, Bluespec, PipelineC, MyHDL, Chisel etc. However, implementing the provider for example as a firmware, using the C language and a microcontroller, is practically doable and valid.

1.5. 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.

1.6. 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, including glyphs and special characters. 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.

3.1.1. Documentation comments

Documentation comments are comments that appear immediately before package scope constant definitions and before functionality instantiations with no intervening newlines. Following code shows examples of documentation comments:

```
# Number of receivers
const RECEIVERS_COUNT = 7
Main bus
    # Data receivers
    Receivers [RECEIVERS_COUNT]block
        # 0 disable receiver, 1 enable receiver.
        Enable config; width = 1
        # Number of frames in the buffer
        Frame_Count status
        # Read_Frame reads single data frame
        Read_Frame proc
            data [4]return; width = 8
```

3.2. Identifiers

Identifiers are used as names. An identifier shall start with a letter.

```
uppercase_letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | R | S | T | U | V | W | X | Y | Z

lowercase_letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | r | s | t | u | v | w | x | y | z

letter ::= uppercase_letter | lowercase_letter

letter_or_digit ::= letter | decimal_digit

identifier ::= letter { underscore | letter_or_digit }
```

Following code contains some valid and invalid identifiers.

```
const C_20 = 20 # Valid
const _C20 = 20 # Invalid
Main bus
    cfg1 config # Valid
    1cfg config # Invalid
```

3.2.1. Declared identifier

Declared identifier is used for any occurrence of an identifier that already denotes some declared item.

```
declared_identifier ::= letter { underscore | letter_or_digit }
```

3.2.2. Qualified identifier

The qualified identifier is used to reference a symbol from foreign .

```
qualified_identifier ::= declared_identifier.declared_identifier
```

The first declared identifier denotes the package, and the second one denotes the symbol from this package.

3.3. Indent

The indentation has semantics meaning in the FBDL. There is only single indent character, the horizontal tab (U+0009). It is hard to express the indent and dedent using BNF. Ident is the increase of the indentation level and dedent is the decrease of the indentation level. In the following code the indent happens in the lines number 2, 5 and 7 and the dedent happens in the line number 4. What is more 2 dedents happens at the EOF. The number of indents always equals the number of dedents in the syntactically and semantically correct file.

```
1: type cfg_t config
2:     atomic = false
3:     width = 64
4: Main bus
5:     C cfg_t
6:     Blk block
7:         C cfg_t
8:         S status
```

Not only the indent alignment is important, but also its level. In the following code the first type definition is correct, as the indent level for the definition body is increased by one. The second type definition is incorrect, even though the indent within the definition body is aligned, as the indent level is increased by two.

```
type cfg1_t config
    atomic = false
    width = 8
type cfg2_t config
        atomic = false
        width = 8
```

3.4. Keywords

Keywords cannot be used as identifiers.

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

3.5. Literals

3.5.1. 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**

3.5.2. 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}

3.5.3. 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.

3.5.4. String literals

3.5.5. Bit string literals

4. Data types

There are 6 data types in FBDL:

- bit string,
- bool,
- integer,
- real,
- string,
- time.

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. Bit string

4.2. 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.

4.3. Integer

4.4. Real

4.5. String

4.6. Time

The value of time type can be created explicitly using the time literal.

```
time_unit ::= ns | us | ms | s
time_literal ::= integer_literal time_unit
```

The time type is only used for assigning value to the properties expressed in time. Values of time type can be added regardless of their time units. Values of the time type can also be multiplied by values of the integer type. All of below property assignments are valid.

```
sleep = 1 s + 1 ms + 1 us + 1 ns
sleep = 5 * 60 s # Sleep for 5 minutes.
sleep = 10 ms * 4 + 7 * 8 us
```

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:

`abs``(x integer|real) integer|real`
Abs returns the absolute value of x.

`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

Functionalities are the core part of the FBDL. They define the capabilities of the provider. Each functionality is distinct and in an unambiguous way defines the provider behavior and the interface that must be generated for the requester. There are 11 functionalities:

1) block,
2) bus,
3) config,
4) mask,
5) memory,
6) param,
7) proc,
8) return,
9) static,
10) status,
11) stream.

6.1. Properties

Each functionality has associated properties. Properties allow configuration of functionalities. Each property must have concrete type. The default value of each property is specified in the round brackets () in the functionality subsections. If the default value is `bus width`, then it means that the default value equals the actual value of the bus `width` property. If the default value is `undefined`, then it shall be represented as the undefined meta value at the provider side. If the target language for the provider code does not have a concept of undefined value, then values such as 0, `Null`, `None`, `nil` etc. shall be used.

```
property_assignment ::= declared_identifier = expression
```

```
property_assignments ::=
    property_assignment
    { ; property_assignment }
    newline
```

```
semicolon_and_property_assignments ::= ; property_assignments
```

6.2. Instantiation

A functionality can be instantiated in a single line or in multiple lines.

```
instantiation ::= single_line_instantiation | multi_line_instantiation
```

```
single_line_instantiation ::=
    identifier
    [ array_marker ]
    declared_identifier | qualified_identifier
    [ argument_list ]
    newline | semicolon_and_property_assignments
```

Following code shows examples of single line instantiations:

```
C config
C config; width = 8
```

```
M [8]mask; atomic = false; width = 128; default = 0
err error_t(48); atomic = false
```

6.3. Block

The block functionality is used to logically group or to encapsulate functionalities. The block is usually used to separate functionalities related with particular peripherals such as UART, I2C transceivers, timers, ADCs, DACs etc. The block might also be used to limit the access for particular provider to only a subset of functionalities.

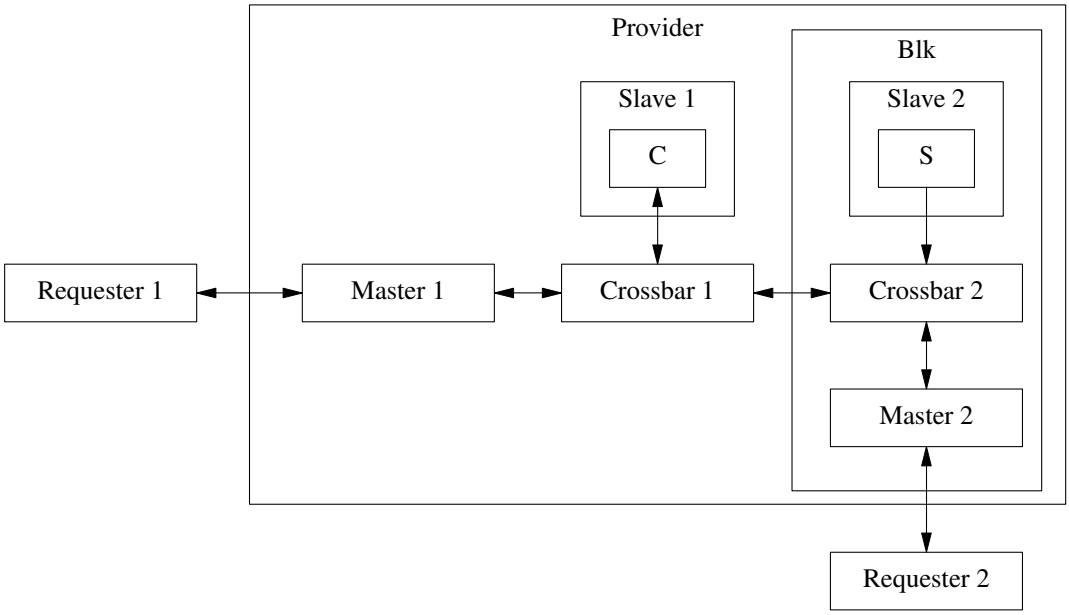
The block functionality has following properties:

masters integer (1)
The masters property specifies the number of block masters.

The following example presents how to limit the scope of access for particular requester.

```
Main bus
    C config
    Blk block
        masters = 2
        S status
```

The logical connection of the system components may look as follows:



The requester number 1 can acces both config `C` and status `S`. However, the requester number 2 can access only the status `S`.

6.4. Bus

The bus functionality represents the bus structure. Every valid description must have at least one bus instantiated, as the the bus named `Main` is the entry point for the description used for the code generation.

The bus functionality has following properties:

masters integer (1)
The masters property specifies the number of bus masters.

width integer (32)
The width property defines the bus data width.

The bus address width is not explicitly set, as it implies from the address space size needed to pack all functionalities included in the `Main` bus description.

6.5. Config

The config functionality represents a configuration data. A configuration data is a data that is automatically read by the provider from its registers. As the config is automatically read by the provider, there is no need for additional signal, associated with the config, indicating config write by the requester. By default, a config can be written and read by the requester.

The config functionality has following properties:

atomic bool (**true**)
The atomic property defines whether an access to the config must be atomic. If atomic is true, then the provider must guarantee that any change of the config value, triggered by the requester write, is seen as an atomic change by the other modules of the provider. This is especially important when the config spans more than single register, as in case of single register write the change is always atomic.

default bit string | integer (undefined)
The default property defines the initial value of the config.

once bool (**false**)
The once property determines whether the config can be written only once.

width integer (bus width)
The width property defines the bit width of the config.

The code generated for the requester must provide methods for writing and reading the config.

6.6. Mask

The mask functionality represents a bit mask. The mask is a data that is automatically read by the provider from its registers. By default, a mask can be written and read by the requester. The mask is very similar to the config. The difference is that the config is value oriented, whereas the mask is bit oriented. From the provider's perspective the mask and the config are the same. From the requester's perspective the code generated for interacting the mask and the config is different.

The mask functionality has following properties:

atomic bool (**true**)
The atomic property defines whether an access to the mask must be atomic. If atomic is true, then the provider must guarantee that any change of the mask value, triggered by the requester write, is seen as an atomic change by the other modules of the provider. This is especially important when the mask spans more than single register, as in case of single register write the change is always atomic.

default bit string | integer (undefined)
The default property defines the initial value of the mask.

width integer (bus width)
The width property defines the bit width of the mask.

The code generated for the requester must provide methods for setting and clearing particular bits of the mask.

6.7. Memory

6.8. Param

The param functionality is an inner functionality of the proc and stream functionalities. It represents a data fed to a proc or streamed by a downstream.

The param functionality has following properties:

width integer (bus width)
The width property defines the bit width of the param.

Following example presents the definition of a downstream with three parameters.

```
Sum_Reduce stream
    type param_t param; width = 16
    a param_t
    b param_t
    c param_t
```

6.9. Proc

The proc functionality represents a procedure called by the requester and carried out at the provider. The proc functionality might contain param and return functionalities. Params are procedure parameters and returns represent data returned from the procedure.

The proc has associated signals at the provider side, the `call` signal and the `exit` signal. The call signal must be driven active for one clock cycle after all registers storing the parameters have been written. The exit signal must be driven active for one clock cycle after all registers storing the returns have been read. An empty proc (proc without params and returns) has only the call signal. A proc having only parameters has only the call signal. **TODO: With proc having only returns is not yet clear what is the best approach.**

The proc functionality has following properties:

The code generated for the requester must provide method for calling the procedure.

6.10. Return

The return functionality is an inner functionality of the proc and stream functionalities. It represents a data returned by a proc or streamed by an upstream.

The return functionality has following properties:

width integer (bus width)
The width property defines the bit width of the return.

Following example presents the definition of a procedure returning 4 element byte array, and a single bit flag indicating whether the data is valid.

```
Read_Data proc
    data [4]return; width = 8
    valid return; width = 1
```

6.11. Static

The static functionality represents an information, placed at the provider, that shall never change.

The static functionality has following properties:

default bit string | integer (obligatory)
The default property defines the value of the static.

once bool (**false**)
The once property determines whether the static can be read only once. If once is true, then after the first read the provider must return the value of ... property on every static read.

width integer (bus width)
The width property defines the bit width of the static.

The static functionality may be used for example for versioning, bus id, bus generation timestamp or for storing secrets, that shall be read only once. Example:

```
Secret static
    width = 8
    default = 113
    once = true
    ... = 0xFF
```

6.12. Status

The status represents an information that is produced by the provider and is only read by the requester.

The status functionality has following properties:

atomic bool (**true**)
The atomic property defines whether an access to the status must be atomic. If atomic is true, then the provider must guarantee that any change of the status value is seen as an atomic change by the requester. This is especially important when the status spans more than single register, as in case of single register read the change is always atomic.

once bool (**false**)
The once property determines whether the status can be read only once.

width integer (bus width)
The width property defines the bit width of the status.

The code generated for the requester must provide method for reading the status.

6.13. Stream

The stream functionality represents a stream of data to a provider (downstream), or a stream of data from a provider (upstream). An empty stream (stream without any params or returns) is always a downstream. It is useful for triggering cyclic action with constant time interval. A downstream must not have any returns. An upstream shall not have any params, and must have at least one return.

The stream property is very similar to the proc property, but they are not the same. There are two main differences. The first one is that the stream must not containt both params and returns. The second one is that the code for the stream, generated for the requester, shall take into account the fact that an access to the stream is multiple and an access to the proc is single. For example lets consider the following bus description:

```
Main bus
    P proc
        p param
    S stream
        p param
```

The code generated for the requester, implemented in the C language, might include following function prototypes:

```
int Main_P(const uint32_t p);
int Main_S(const uint32_t * p, size_t count);
```

The stream functionality has following properties.

7. Parametrization

The FBDL provides following three ways for description parametrization:

- constants,
- type definitions,
- types extending.

7.1. Constant

The constant represents a constant value. The value might be used in expression evaluations. The following code presents bus with three functionalities, all having the same array dimensions and width.

```
Main width
    const ELEMENT_COUNT = 4
    const WIDTH = 8
    C [ELEMENT_COUNT]config; width = WIDTH
    M [ELEMENT_COUNT]mask; width = WIDTH
    S [ELEMENT_COUNT]status; width = WIDTH
```

Constants must be included in the generated code, both for the provider and for the requester. This allows for having a single source of the constant value.

A constant can be defined in a single line in the single line constant definition or as a part of the multi constant definition.

```
single_constant_definition ::= const identifier = expression newline
```

Examples of single constant definition:

```
    const WIDTH = 16
    const FOO = 8 * BAR
    const LIST = [1, 2, 3, 4, 5]
```

```
multi_constant_definition ::=
    const newline
    indent
    identifier = expression newline
    { identifier = expression newline }
    dedent
```

Examples of multi constant definition:

```
    const
        WIDTH = 16
        FOO = 8 * BAR
        LIST = [1, 2, 3, 4, 5]
    const
        ONE = 1
        TWO = ONE + 1
        THREE = TWO + 1
```

7.2. Type definition

7.3. Type extending

The the extending allows extending any custom defined type, either by instantiation or by defining new type. This is especially, but not only, useful when there are similar blocks with only slightly different set of

functionalities.

Example:

```
type blk_common_t block
    C1 config
    M1 mask
    S1 status
Main bus
    Blk_C blk_common_t
        C2 config
    Blk_M blk_common_t
        M2 mask
    Blk_S blk_common_t
        S2 status
```

This description is equivalent to the following description:

```
type blk_common_t block
    C1 config
    M1 mask
    S1 status
type blk_C_t blk_common_t
    C2 config
type blk_M_t blk_common_t
    M2 mask
type blk_S_t blk_common_t
    S2 status
Main bus
    Blk_C blk_C_t
    Blk_M blk_M_t
    Blk_S blk_S_t
```

The type nesting has no depth limit. However, no property already set in one of the ancestor types can be overwritten. Also no symbol identifier defined in one of the ancestor types can be redefined.

8. Scope and visibility

8.1. Import and package system

The FBDL has a concept of packages and allows importing packages into the file scope using the import statements. A package consists of files with `.fbd` extension placed in the same directory. A package must have at least one file and shall not be placed in more than single directory. A package is uniquely identified by its path. The name of a package is equivalent to the last part of its path. That is, it is the same as the name of the directory containing package files. However, if the package name starts with the "fbd-" prefix, then the prefix is not included in the package name. For example, two packages with following paths `foo/bar/uart` and `baz/zaz/fbd-uart` have exactly the same name `uart`.

A package can be imported in a single line using the single line import statement or as a part of the multi import statement.

```
single_import_statement ::= import [ identifier ] string_literal
```

Examples of single import statement:

```
    import "uart"
    import spi "custom_spi"
```

```
multi_import_statement ::=
    import newline
    indent
    [ identifier ] string_literal
    { [ identifier ] string_literal }
    dedent
```

Example of multi import statement:

```
    import
        "uart"
        spi "custom_spi"
```

The string literal is the path of the package. The path might not but be complete, but shall be unambiguous. For example, if two paths are visible by the import statement (*"foo/bar/uart"* and *"baz/zaz/uart"*), and both ends with *"uart*", then *"uart"* path is ambiguous, but *"bar/uart"* and *"zaz/uart"* are not.

The optional identifier is an identifier that shall denote the imported packaged within the importing file. If the identifier is omitted, then the implicit identifier for the package is the last part of its path.

8.2. Scope rules

The following elements define a new scope in the FBDL:

- package,
- type definition,
- functionality instantiation.

Following example presents all scopes.

```
    const WIDTH = 16
    const WIDTHx2 = WIDTH * 2
    Main bus
        width = WIDTH
        const C20 = 20
        Blk block
```

```
const C30 = 30
type cfg_t(WIDTH = WIDTH) config
    atomic = false
    width = WIDTH
Cfg16 cfg_t
Cfg20 cfg_t(C20)
Cfg30 cfg_t(C30)
```

The `WIDTH` constant has package scope, and it is visible at the package level, in the `Main` bus instantiaiton and in the `Blk` block instantiaiton. It would also be visible in the `cfg_t` type definition. However, the `cfg_t` type has the parameter with the same name `WIDTH`. As a result, only the `WIDTH` parameter is visible within the type definition. The `WIDTH` parameter has default value that equals 16. This is becuase at this point the name `WIDTH` denotes the package level `WIDTH` constant. Type parameters are visible inside the type definition, but not in the type parameter list. The `Cfg16` is thus a non-atomic config of width 16, the `Cfg20` is a non-atomic config of width 20 and the `Cfg30` is a non-atomic config of width 30.