Functional Bus Description Language

Revision 0.0

2024-05-29

Abstract

This document is the official specification of the Functional Bus Description Language. Its primary purpose is to define the syntax and semantics of the language. Functional Bus Description Language is a domain-specific language for bus and register management. Its main characteristic is the paradigm shift from the register-centric approach to the functionality-centric approach. In the register-centric approach, the user defines registers and then manually lays out the data into the registers. In the functionality-centric approach, the user defines the functionality of the data, and the registers, module hierarchy, and access codes are later automatically inferred. By defining the functionality of the data placed in the registers, it is possible to generate more code, increase code robustness, improve system design readability, and shorten the implementation process.

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

Contents

Participants

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

Glossary

Not all terms defined in the glossary list are used in the specification. Some of them are formally defined because they are helpful when discussing, for example, compiler implementation.

call register

The call register term is used to refer to the `proc` register with the associated call pulse signal. When the call register is written, the call pulse is generated.

data

The data term is used to refer to the content of the registers. Unless it is used in the context of internal data types of the language.

downstream

The downstream is a stream from the requester to the provider.

exit register

The exit register term is used to refer to the `proc` register with the associated exit pulse signal. When the exit register is read, the exit pulse is generated.

functionality

The functionality is the functionality of given data. It can be seen as a type of the data. In case of functionalities encapsulating other functionalities, such as `bus`, `block`, `proc` or `downstream`, the functionality is used to denote a broader context of encapsulated data.

gap

The gap term is used to refer to unused bits within a register.

gateware

The gateware term is used to refer to the overall configuration of the logic placed in the FPGA to make it behave according to the desired description. The term is not formally defined anywhere, however it is used to unburden the firmware term. IEEE Std 610.12-1990 also mentions that the firmware term is too overloaded and confusing.

generator

The generator term is used to refer to the part of a compiler directly responsible for the target code generation based on registerification results.

information

The information term is used to refer to the metadata on the functionality data. The metadata describes where the data is located, for example bit masks and register addresses, and how to access the data.

means

The means term is used to refer to the automatically generated method or data that shall be used by the requester to request particular functionality. A means in particular programming language is usually a function, method or procedure that shall be called or class, dictionary, map or structure containing information on how to access particular functionality.

provider

The provider is the system component containing the generated registers and providing described functionalities.

pure call register

The term pure call register is used to refer to the call register containing no `proc` returns.

pure exit register

The term pure exit register is used to refer to the exit register containing no `proc` params.

registerification

The registerification is the process of placing data of functionalities into the registers. The process includes assigning data bit masks, register addresses as well as block addresses and masks. The term is new in the field and is coined in the specification.

requester

The requester is the system component accessing the generated registers and requesting described functionalities.

strobe register

The strobe register term is used to refer to the `streamr`egister with the associated strobe pulse signal. When the strobe register is written (downstream), or read (upstream) the strobe pulse is generated.

target

The target term is used to refer to the transpilation target. For example, a target can be a requester Python code allowing to access functionalities of the provider in an asynchronous fashion. A VHDL code providing description of the functionality registers and exposing AXI compliant interface is a valid provider target. A JSON file describing registerification results is for example a valid documentation target. The target depends on several factors, but the most important ones are programming/description language, synchronous or asynchronous access interface, bus type, dynamic or static address map reloading. Each target has its recipient. It is either provider, requester or documentation.

upstream

The upstream is a stream from the provider to the requester.

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, or user feedback. 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 to register management. However, they all share the same concept of describing registers at a very low level. That is, the user has to implicitly define the layout of the registers. For example, in the case of a register containing multiple statuses, its the user responsibility to specify the bit position for every status.

The FBDL is different in this term. The user specifies the functionalities that must be provided by the data stored in the registers. The register layout is automatically generated based on the functional requirements. Such an approach increases the amount of automatically generated hardware description and software code and decreases the amount of code requiring manual implementation compared to the register-centric approach. Not only the register masks, addresses, and single read and 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. 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 (declared by a user or by specification, for example, built-in function name).

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 the 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,
- IETF UTF-8, a transformation format of ISO 10646, RFC 3629,
- IEEE Std 754™–2019, IEEE Standard for Floating-Point Arithmetic.

3 Concepts

The core concept behind the FBDL is based on the fact that if there is a system part with the registers that can be accessed, then there is at least one more system part accessing these registers. The part accessing 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.

The description of functionalities shall be placed in files with the `.fbd` extension. By default, the bus named `Main` is the entry point for the description used for the code generation. A compiler is free to support a parameter for changing the name of the main bus.

```
description ::=
    import_statement |
    constant_definition |
    type_definition |
    instantiation
```

3.1 Properties

Each data in the FBDL description has associated functionality and each functionality has associated properties. Properties allow the configuration of functionalities. Each property must have a 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 the default value equals the actual value of the bus `width` property. If the default value is `uninitialized`, then it shall be represented as the uninitialized meta value at the provider side. If the target language for the provider code does not have a concept of uninitialized value, then values such as `0`, `Null`, `None`, `nil` etc. shall be used.

Each property either defines or declares functionality feature or behavior. Definitive properties specify the desired behavior of the automatically generated code. They specify elements directly managed by the FBDL. Examples of definitive properties include `atomic` or `width` properties. Declarative properties describe the behavior of external elements that automatically generated code only interacts with. Declarative properties are required to generate valid logic, and it is the user's responsibility to make sure their values match the behavior of external components. Examples of declarative properties include `access` or `in-trigger` properties.

```
property_assignment ::= property_identifier = expression

property_assignments ::=
    property_assignments
    { ; property_assignment }
    newline

semicolon_and_property_assignments ::= ; property_assignments

property_identifier ::=
    access | add-enable | atomic | byte-write-enable | clear | delay |
    enable-init-value | enable-reset-value | groups | init-value |
```

```
    in-trigger | masters | out-trigger | range | read-latency |
    read-value | reset | reset-value | size | width
```

3.2 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

multi_line_instantiation ::=
    identifier
    [ array_marker ]
    declared_identifier | qualified_identifier
    [ argument_list ]
    functionality_body

array_marker ::= [ expression ]

functionality_body ::=
    newline
    indent
    {
        constant_definition |
        type_definition |
        property_assignments |
        instantiation
    }
    dedent
```

The following code shows examples of single line instantiations:

```
C config
C config; width = 8
M [8]mask; atomic = false; width = 128; init-value = 0
err error_t(48); atomic = false
```

The following code shows examples of multi line instantiations:

```
My_Config config
  width = 96
  atomic = false
My_Irq irq
  add-enable = true
  in-trigger = "Edge"
```

3.3 Addressing

The FBDL specification does not impose byte or word addressing. There is also no property allowing to switch between these two addressing modes. The addressing mode handling is completely left to the particular compiler implementation. If the compiler has a monolithic structure (no distinction be-

tween the compiler frontend and backend), then it is probably the best decision to use the addressing mode used by the target bus (for example, byte addressing for AXI or word addressing for Wishbone). Another option is providing a compiler flag or parameter to specify the addressing mode during the compiler call. However, in the case of a compiler frontend implementation, it is recommended to use word addressing with a word width equal to the bus width. As it is not known whether the compiler backend will use the word or byte addressing, using the word addressing in the compiler frontend is usually a more straightforward approach, as the byte addresses are word addresses multiplied by the number of bytes in the single word.

3.4 Positive logic

The FBDL uses only positive logic. An active level in positive logic is a high level (binary 1), and an active edge is a rising edge (transition from the low level to the high level, from binary 0 to binary 1). It does not mean that FBDL cannot be used with external components using negative logic. To connect external negative logic components to the generated FBDL positive logic components, one shall negate the signals at the interface connection level. Supporting both positive and negative logic would unnecessarily complex the language and would create a second way for solving the same problem making the set of possible solutions non-orthogonal.

3.5 Domain-specific language

The FBDL is a domain-specific language with its own syntax. Some of the register-centric tools are built on top of standard file formats or markup languages such as JSON, TOML, XML or YAML. Such an approach allows for fast prototyping and has a lower entry threshold. However, it becomes a burden when more conceptually advanced features, for example parametrization, have to be supported. The description quickly begins to gain in volume, and the overall feeling is it is needlessly verbose. What is more, having its own adjusted language syntax allows for more informative compiler error messages.

4 Lexical elements

FBDL has following types of lexical tokens:

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

4.1 Comments

There is only a single type of comment, a *single-line comment*. A single-line comment starts with the '#' character and extends up to the end of the line. A single-line comment can appear on any line of an 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.

4.1.1 Documentation comments

Documentation comments are comments that appear immediately before constant definitions, type definitions, and functionality instantiations with no intervening newlines. The 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
    # Documentation comments can consist of
    # multiple single-line comments.
    Read_Frame proc
      data [4]return; width = 8
```

4.2 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 }
```

The 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
```

4.2.1 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 }
```

4.2.2 Qualified identifier

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

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

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

4.3 Indent

The indentation has semantics meaning in the FBDL. The indent sequence consists of two space characters (U+0020). 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, double dedent 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.

```
# Valid indent
type cfg1_t config
  atomic = false
  width = 8
# Invalid indent, indent increased by two
type cfg2_t config
    atomic = false
    width = 8
```

4.4 Keywords

FBDL has following keywords: **`atomic`**, **`block`**, **`bus`**, **`clear`**, **`config`**, **`const`**, **`false`**, **`import`**, **`init-value`**, **`irq`**, **`mask`**, **`memory`**, **`param`**, **`proc`**, **`range`**, **`reset`**, **`read-value`**, **`reset-value`**, **`return`**, **`static`**, **`stream`**, **`true`**, **`type`**, **`in-trigger`**, **`out-trigger`**.

Keywords can be used as identifiers with one exception. Keywords denoting built-in types (functionalities) cannot be used as identifiers for custom types.

4.5 Literals

4.5.1 Bool literals

```
bool_literal ::= false | true
```

4.5.2 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
```

4.5.3 Integer literals

```
integer_literal
    binary_literal |
    octal_literal |
    decimal_literal |
    hex_literal

binary_literal ::= binary_base binary_digit { [ underscore ] binary_digit }

octal_literal ::= octal_base octal_digit { [ underscore ] octal_digit }

decimal_literal ::= non_zero_decimal_digit { [ underscore ] decimal_digit }

hex_literal ::= hex_base hex_digit { [ underscore ] hex_digit }
```

4.5.4 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.

4.5.5 String literals

A string literal is a sequence of zero or more UTF-8 characters enclosed by double quotes (“”).

```
string_literal ::= "{UTF-8 character}"
```

4.5.6 Bit string literals

A bit string literal is a sequence of zero or more digit or meta value characters enclosed by double quotes (“”) and preceded by a base specifier. The meta value characters are supported because of hardware description languages, that also have a concept of metalogical values.

```
meta_character ::= - | U | W | X | Z
```

The meta characters have following meaning:
- ‘-’ - don’t care,
- ‘U’ - uninitialized,
- ‘W’ - weak unknown,
- ‘X’ - unknown,
- ‘Z’ - high-impedance state.

```
binary_or_meta ::= binary_digit | meta_character
```

```
octal_or_meta ::= octal_digit | meta_character
```

```
hex_or_meta ::= hex_digit | meta_character
```

There are three types of bit string literals: binary bit string literal, octal bit string literal and hex bit stirng literal.

```
bit_string_literal ::=
    binary_bit_string_literal |
    octal_bit_string_literal |
    hex_bit_string_literal
```

```
binary_bit_string_base = B | b
```

```
binary_bit_string_literal = binary_bit_string_base "{binary_or_meta}"
```

```
octal_bit_string_base = O | o
```

```
octal_bit_string_literal = octal_bit_string_base "{octal_or_meta}"
```

```
hex_bit_string_base = X | x
```

```
hex_bit_string_literal = hex_bit_string_base "{hex_or_meta}"
```

If meta value is present in a bit string literal, then it is expanded to the proper width depending on the bit string base. For example, following equations are true:

```
o"XW" = b"XXXWWW"
x"U-" = b"UUUU----"
```

4.5.7 Time literals

A time literal is a sequence of integer literal and a time unit.

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

```
time_literal ::= integer_literal time_unit
```

Time literals are used to create values of time data type, required for example by the `delay` property.

5 Data types

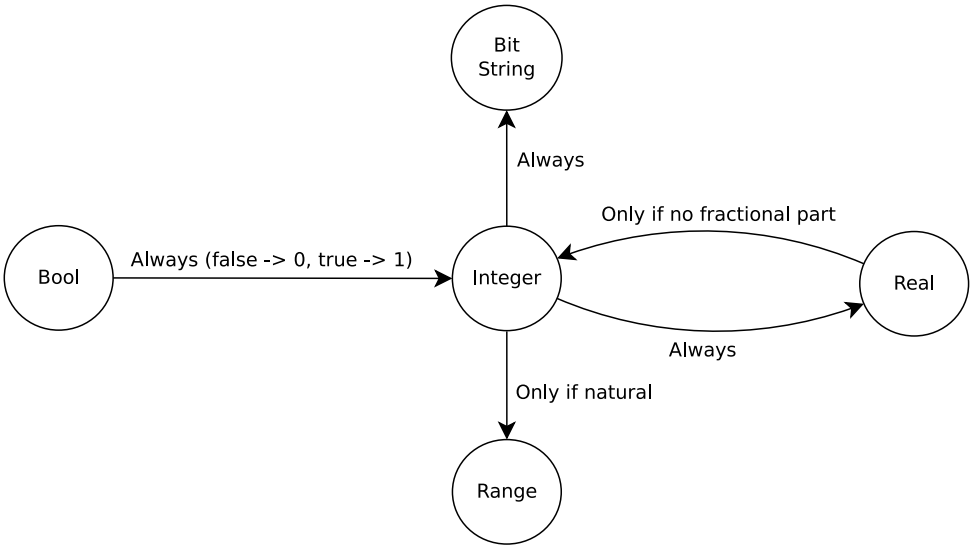
There are 7 data types in FBDL:

- bit string,
- bool,
- integer,
- range,
- 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 shall be resolved. In case of a type mismatch that cannot be resolved, an error must be reported by the compiler.

Conversion from bool to integer in expressions is implicit. Conversion from integer to real in expressions is implicit. Conversion from integer to range can be implicit if the interger value is natural. Conversion from real to integer can be implicit if there is no fractional part. If fractional part is present, then conversion from real to integer must be explicit and must be done by calling any function returning integer type, for example `ceil()`, `floor()`.

The below picture presents a graph of possible implicit conversions between different data types.



5.1 Bit string

The value of the bit string type is used for all **`*-value`** properties. It might be created explicitly using the bit string literal or it might be converted implicitly from the value of integer type. The only way to create a bit string value containing meta values is to explicitly use the bit string literal.

The below table presents unary negation operation results applied to possible bit string data type values.

Bit string unary bitwise negation

In Value	Out Value
`0`	`1`
`1`	`0`

-	-
U	U
W	W
X	X

Below tables present binary operation results applied to possible bit string data type values.

Bit string binary bitwise and (&) resolution

Operands	0	1	-	U	W	X	Z
0	0	0	0	U	0	X	0
1	0	1	1	U	1	X	1
-	0	1	-	U	W	X	Z
U	U	U	U	U	U	U	U
W	0	1	X	U	W	X	W
X	X	X	X	U	X	X	X
Z	0	1	X	U	W	X	Z

Bit string binary bitwise or (|) resolution

Operands	0	1	-	U	W	X	Z
0	0	1	0	U	0	X	0
1	1	1	1	U	1	X	1
-	0	1	-	U	W	X	Z
U	U	U	U	U	U	U	U
W	0	1	X	U	W	X	W
X	X	X	X	U	X	X	X
Z	0	1	X	U	W	X	Z

Bit string binary bitwise xor (^) resolution

Operands	0	1	-	U	W	X	Z
0	0	1	0	U	0	X	0
1	1	0	1	U	1	X	1
-	0	1	-	U	W	X	Z
U	U	U	U	U	U	U	U
W	0	1	X	U	W	X	W
X	X	X	X	U	X	X	X
Z	0	1	X	U	W	X	Z

5.2 Bool

The value of the bool type can be created explicitly using `true` or `false` literals. The value of the bool type shall be implicitly converted to the value of the integer type in places where the value of the integer type is required. The boolean `false` value shall be converted to the integer value 0. The boolean `true` value shall be converted to the 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.

5.3 Integer

The integer data type is always signed integer and must be at least 64 bits wide.

5.4 Real

The real data type is 64 bits IEEE 754 double precision floating-point type.

5.5 String

The string data type can only be created explicitly using a string literal. The string data type is only used for setting values of some properties, for example `access`.

5.6 Time

The time data type is only used for assigning value to the properties expressed in time. The value of time type can be created explicitly using the time literal. 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 the below property assignments are valid.

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