Functional Bus Description Language

Revision 0.0

2024-05-27

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 between 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.