WARSAW UNIVERSITY OF TECHNOLOGY

DISCIPLINE OF SCIENCE AUTOMATIC CONTROL, ELECTRONICS, ELECTRICAL ENGINEERING AND SPACE TECHNOLOGIES

FIELD OF SCIENCE ENGINEERING AND TECHNOLOGY

Ph.D. Thesis

Michał Kruszewski, M.Sc.

Functional Bus Description Language

Supervisor

Wojciech Zabołotny, Ph.D., D.Sc

WARSAW 2023

Abstract

Bus and register management is one of the crucial aspects of ASIC, SoC or FPGA based designs. The problems related to it are well known, and multiple tools or approaches are already trying to solve or mitigate them. However, all available solutions share the same register-centric paradigm. A user defines registers and then manually lays out the data into the registers. Such an approach has its limitations. A description does not contain information on data spanning multiple registers or data forming a broader context, procedure arguments, for example. It also does not contain information on the data purpose. As a result, the generated access code is low-level and usually needs an extra wrapper, which leaves room for potential human mistakes. For instance, it is the user's responsibility to guarantee proper access order to registers or to provide an atomic change of data wider than single register width.

The thesis proposes a new approach, the functionality-centric approach. In the functionality-centric approach user defines the data with the type of its functionality. The registers and bus hierarchy are later implicitly inferred. By defining the functionality of the data placed in the registers, it is possible to generate more access code, increase code robustness, improve system design readability, and shorten the implementation process.

The thesis includes the specification of the new domain-specific language (Functional Bus Description Language), the reasoning for some of the design decisions as well as some of the compiler implementation details.

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

Streszczenie

Zarządzanie magistralą oraz rejestrami jest jednym z kluczowych aspektów podczas projektowania układów ASIC, SoC lub systemów wykorzystujących układy FPGA. Problemy z tym związane są dobrze znane. Istnieje wiele narzędzi oraz sposobów postępowania, które starają się je rozwiązywać lub niwelować ich wpływ. Wszystkie dostępne rozwiązania cechuje jednak te same podejście do zagadnienia, są one zorientowane na rejestry. Użytkownik pierw definiuje rejestr, a dopiero w kolejnym kroku ręcznie rozmieszcza w nim dane. Takie podejście zawiera pewne ograniczenia. Opis rejestrów nie zawiera informacji na temat danych znajdujących się w więcej niż jednym rejestrze, czy na temat danych będących częścią jakiegoś szerszego kontekstu, jak np. argumenty procedur. Opis nie zawiera również informacji na temat funkcjonalności jakie poszczególne dane dostarczają. W rezultacie automatycznie wygenerowany kod jest niskopoziomowy i wymaga ręcznej implementacji kodu opakowującego. To z kolei przekłada się na pozostawienie miejsca na potencjalne ludzkie pomyłki. Przykładowo, to użytkownik odpowiedzialny jest za zapewnienie poprawnej kolejności dostępów do rejestrów, czy za zapewnienie atomowości zmian wartości danych, których szerokość przekracza szerokość pojedynczego rejestru.

W rozprawie zaprezentowano nowe podejście zorientowane na funkcjonalność danych. W podejściu tym użytkownik definiuje dane wraz z ich typem funkcjonalności. Na ich podstawie są następnie automatycznie generowane rejestry wraz z hierarchią magistrali. Definiowanie funkcjonalności danych pozwala na zwiększenie ilości kodu dostępowego generowanego automatycznie, i zmniejszenie ilości kodu pisanego ręcznie. To z kolei zwiększa odporność kodu na błędy, poprawia czytelność projektu i skraca czas spędzony na implementacji.

Praca obejmuje specyfikację jezyka specyficznego dla danej domeny (Język Opisu Funkcjonalnych Magistral), uzasadnienie niektórych decyzji projektowych oraz omówienie niektórych ze szczegołów implementacji kompilatora.

Słowa kluczowe: adresowanie rejestrów, automatyzacja projektowania, magistrala, generacja oprogramowania, generacja dokumentacji, hierarchiczny opis rejestrów, interfejs sterowania, język opisu sprzętu, języki programowania, magistrala, programowanie, projektowanie sprzętu, synteza rejestrów, systemy elektroniczne, utrzymanie kodu, weryfikacja projektu, zarządzanie systemem

Contents

List of abbreviations

AGWB Address Generator for Wishbone

AMBA ARM Advanced Microcontroller Bus Architecture

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

CBM Compressed Baryonic Matter

CDC Clock Domain Crossing

CPU Central Processing Unit

CMS Compact Muon Solenoid

DAQ Data Acquisition

DESY Deutsches Elektronen-Synchrotron

EDA Electronic Design Automation

EISA Extended Industry Standard Architecture

FBDL Functional Bus Description Language

FPGA Field Programmable Gate Array

GUI Graphical User Interface

HDL Hardware Description Language

HEP High Energy Physics

HLS High Level Synthesis

HTML HyperText Markup Language

IP Intellectual Property / Internet Protocol

ISA Industry Standard Architecture

JSON JavaScript Object Notation

LAN Local Area Network

LSB Least Significant Bit

MCA Micro Channel Architecture

MCU Microcontroller Unit

MMIO Memory Mapped Input Output

NoC Network on Chip

PCIe Peripheral Component Interconnect Express

POSIX Portable Operating System Interface for UNIX

SLR Super Logic Region

SoC System on Chip

STS Silicon Tracking System

SystemRDL System Register Description Language

TCP Transmission Control Protocol

TOML Tom's Obvious, Minimal Language

UART Universal Asynchronous Receiver-Transmitter

URL Uniform Resource Locator

UVVM Universal VHDL Verification Methodology

USB Universal Serial Bus

UVM Universal Verification Methodology

VESA Video Electronics Standards Association

VHDL Very High Speed Integrated Circuit Hardware Description Language

WAN Wide Area Network

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Preface

Context and motivation of the dissertation

Designing, implementing, and integrating FPGA-based designs with a software stack running on a traditional CPU or with a firmware stack running on an MCU poses a relatively complex technological, organizational, and methodical task. DAQ systems for HEP experiments, among military, medical, and digital entertainment systems, are good examples of areas where such tasks are omnipresent and inevitable.

The author of the dissertation, for four years, has been taking part in the design and implementation process of the gateware, firmware, and software for the DAQ system for the CBM [1] experiment that has been prepared at the GSI Helmholtzzentrum für Schwerionenforschung in Darmstadt [2].

Design environments for DAQ systems in HEP experiments are very peculiar. The whole design and implementation take relatively long, from a few to even a dozen or so years. The engineering teams are international. The educational background is varied. There are physicists, electronics engineers, computer science engineers, system administrators, etc. The spectrum of the age of the members is vast, ranging from first-year Ph.D. students to halftime retired workers. Most members participate in multiple projects or have academic duties, so the time they devote to a particular task is limited. During the development phase, there is also a rotation of the employees. As a whole system is extensive and complex and must work reliably, it is natural that the preliminary prototypes vary significantly from the final solutions. All of this leads to implementing the same or similar functionalities multiple times. For example, a change of programming language after the prototyping stage forces such reimplementation.

During the first two years of the studies, the author explored how to make such complex and multidimensional projects more manageable and verifiable. Trying to incorporate some industrial methodologies, such as UVM framework or formal verification, simply failed. There were at least several reasons for this. To name a few:

- Lack of free, open source tools or limited functionality of such tools. Paid commercial tools have expensive licenses.
- Too steep learning curve and lack of learning resources. The EDA tools appear to be inadequate for engineers who do not use them every day for eight hours. Instead

of focusing on the design and fundamental problems, one spends time learning how to use the EDA tools, and each of them has distinct user interface.

Throughout the work, it turned out that in such a diverse environment, there is another policy suited much better. Instead of incorporating cumbersome industrial standards that need expensive licenses, one can automatically generate as much gateware, firmware, and software as possible. As long as the description format, based upon which parts of the system are generated, is easily readable by a human, the work is moving forward surprisingly fast.

Based on this observation, the author has been looking for a way to enhance and extend existing generic methods and tools commonly used for gateware, firmware, and software code generation. During the work on the AGWB [3], and after using it for a few months, the author noticed that a relatively lot of code was still repeatedly implemented manually. That manually implemented code had some common characteristics and could be easily automatically generated. The only thing missing to generate it was the information on the functionality that must be served by a given data. That required shifting the accent from the register (register-centric approach) to the data, or more precisely to the functionality of the data (functionality-centric approach). After analysing state-of-the-art tools and approaches, the author concluded that there is actually no solution based on the data functionality paradigm. The author has decided that the idea is worth trying, and the FBDL realizes this idea.

Structure of the thesis

The thesis consists of 9 chapters and 4 additional appendices. Appendix D is the specification of the newly defined Functional Bus Description Language. It is advised to at least skim it before reading the dissertation and later come back to it while reading the chapter 5. The specification also includes definitions of some terms used in the thesis.

Chapter 1 introduces the bus and register management problem. It provides a simplified example that is used to present some of the subproblems and analyze how they are solved in the register-centric (typical) approach and functionality-centric (newly proposed) approach.

Chapter 2 briefly discusses on-chip interconnect architectures. It uses AMBA AXI and Wishbone buses to present two distinct bus control logics. It also discusses the NoC technology, a natural progression of traditional on-chip buses.

Chapter 3 is the prior art analysis. It includes only solutions following the register-centric paradigm. The author proposes a shift of paradigm to the functionality, and no solution

following this approach has been found.

Chapter 4 contains the definition of the thesis. Then, the aim and scope of the dissertation is described.

Chapter 5 serves as an extension to the FBDL specification. It discusses all supported functionalities, and unlike the specification, it focuses on answering the „why" questions instead of the „how" questions. It is recommended to read subsections of this chapter concurrently with the corresponding subsections of the FBDL specification (first specification, then dissertation) or to read the whole specification first.

Chapter 6 discusses the most common features present in the register-centric tools but absent in the FBDL. The focus is on reasoning why they are absent at the current stage of the language.

Chapter 7 describes the implementation of the compiler for the FBDL. As the comprehensive description would be relatively long and would include aspects irrelevant from the thesis point of view, the chapter describes only the overall structure and focuses on some general details that probably any FBDL compliant compiler will have to face.

Chapter 8 provides information on the project in which FBDL has been used. However, due to the proprietary nature of the project, no internal details are revealed.

Chapter 9 summarizes the advantages of describing a system bus using the functionality-centric approach instead of the register-centric.

The thesis has numerous code snippets and listings used as examples to illustrate problems better or explain solutions. The VHDL language has been chosen for the gateware, and the Python language has been chosen for the software. However, all presented concepts are programming language agnostic, so any language could be selected, and the reasoning would remain valid.

1 Introduction

Most ASIC, FPGA, or SoC designs, for sure the more complex ones, have some kind of internal bus. Such a bus is often referred to as a „system bus", „local bus", „on-chip bus", „interconnect bus" or „on-chip interconnect bus" (the last one is the most formal and probably the most appropriate). The main role of the bus is to provide an organized and structured manner for connecting independent modules within the chip. It also serves as some kind of gateway to access the internals of the gateware or hardware design from the firmware or software stack. Such access includes writing control signals, reading status signals, bi-directional data streaming, procedure triggering, interrupt signaling, etc. Figure 1.1 presents an example simplified structure of some SoC. Master modules are red, slave modules are yellow, and bus fabric components are blue.

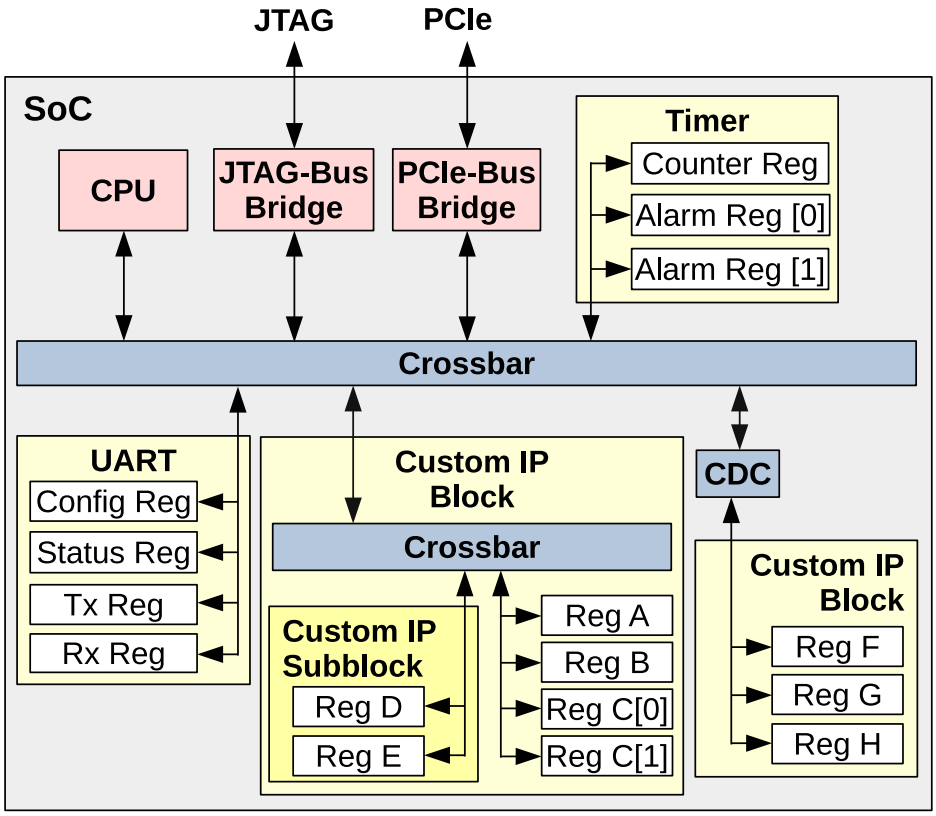


Figure 1.1: Example internal structure of some SoC design with bus.

A bus usually consists of an address bus, a data bus, and a control bus. The most popular on-chip buses used in FPGA designs are probably AXI [4] (which is part of the AMBA) and Wishbone [5].

If there is a bus in a design, then the bus needs to be managed. The bus management consists of the following logical elements:

1. Address space management. This includes:
    a) Assigning address ranges to the modules.
    b) Aligning address ranges according to the user's policy.
2. Bus fabric management. This includes:
    a) Description of the modules hierarchy.
    b) Generation of the bus fabric components (such as crossbars) according to the user-provided description.
3. Registers management. This includes:
    a) Ordering registers within the modules.
    b) Splitting long signals between multiple registers.
    c) Grouping short signals into a single register.
    d) Attributing additional functions to the registers, such as associated strobe or acknowledgment signals.

All of the bus management tasks can be done manually, in a semi-automated way, or in a fully automated way. The greater the automation, the less room for potential engineers' mistakes and the greater pace of the project development.

Managing the bus in a complex system is a well-known and non-trivial problem, especially in hardware-software co-design projects [6, 7, 8, 9]. Even though various approaches and implementations have already been proposed, there is still no solution that would make the bus management process fully automated. All available tools and standards either only support some of the logical elements of bus management or require users to do the register management manually. The register management is the most time-consuming and error-prone part of the bus management.

1.1 Example problem

The following section introduces an example to ease the reasoning. The example is also used to present the typical register-centric approach for managing registers and the new functionality-centric approach proposed in the thesis. It presents some, but not all, prob-

lems encountered in a register-centric approach that are eliminated in the newly proposed approach.

Let's assume there is a module implemented in the FPGA logic called the *Supervisor*. The Supervisor is capable of scheduling work to be done by some *Worker* modules. The Supervisor has its own 48 bits internal counter that can be reset. The Supervisor can pass data to Worker modules at programmed counter value. There are 24 workers, and the data passed to them is two 12 bits long vectors. The data might be passed to any set of workers. For simplicity, let's assume that the data passed to all the workers is the same. The Supervisor also has two additional status bits, informing whether it is currently programmed (the data is scheduled to be processed) and whether it has been programmed in the past. Programming in the past means that the Supervisor will not fire data passing to the Workers before counter overflow. The Supervisor can also be unprogrammed. Listing 1 shows the VHDL interface of the example Supervisor. Signals connected to the particular ports have analogous names without the `_i, _o` suffixes.

The example Supervisor must be controlled by the software running on a CPU. Listing 2 shows an example Python interface of the Supervisor.

Inside an FPGA, there is a 32 bits wide bus (this is the width of the data; the width of the address is irrelevant in this consideration). What bus it is and how it can be accessed from the software is irrelevant to the analysis. A proper interface for accessing the bus is provided via the `registers_handle` parameter.

1.2 Register-centric approach

In the register-centric approach, one has to take the following mandatory steps:

a) Identify control signals. In the case of the Supervisor, these are: `reset_counter, program, unprogram, programmed_counter_value, worker_data0, worker_data1, workers_mask`.

b) Identify status signals. In the case of the Supervisor, these are: `counter, programmed, programmed_in_past, workers_ready`.

c) Identify which control signals form a broader context. For instance, `worker_data0` does not make any sense when it is used alone. It is solely one of the procedure's parameters allowing for passing data to the workers. On the other hand, `unprogram` makes sense on its own.

d) Identify which status signals form a broader context. There is no such case in the

```
entity Supervisor is
   generic (WORKER_COUNT : positive := 24);
   port (
      clk_i : in std_logic;

      -- Supervisor control interface
      counter_o       : out std_logic_vector(47 downto 0);
      reset_counter_i : in  std_logic;
      -- Program procedure
      program_i                  : in std_logic;
      programmed_counter_value_i : in std_logic_vector(47 downto 0);
      worker_data0_i             : in std_logic_vector(11 downto 0);
      worker_data1_i             : in std_logic_vector(11 downto 0);
      -- Workers mask is set independently
      workers_mask_i : in  std_logic_vector(WORKER_COUNT-1 downto 0);
      -- Unprogram procedure
      unprogram_i : in std_logic;
      -- Status bits
      programmed_o         : out std_logic;
      programmed_in_past_o : out std_logic;
      workers_ready_o      : out std_logic_vector(WORKER_COUNT-1 downto 0);

      -- Interface to Workers
      workers_ready_i : in  std_logic_vector(WORKER_COUNT-1 downto 0);
      data_valid_o    : out std_logic_vector(WORKER_COUNT-1 downto 0);
      worker_data0_o  : out std_logic_vector(11 downto 0);
      worker_data1_o  : out std_logic_vector(11 downto 0)
   );
end entity;
```

Listing 1: Example Supervisor VHDL module interface.

example Supervisor.

e) Calculate the number of bits needed for control and status signals. The example Supervisor needs 82 status bits (`counter`, `programmed`, `programmed_in_past`, `workers_ready`) and 96 control bits (`programmed_counter_value`, `worker_data0`, `worker_data1`, `workers_mask`). Whether `reset_counter`, `program`, `unprogram` should be included is yet another question. As these are single-bit signals solely triggering some action, they can be implemented as registers or fields requiring explicit set and clear, or as register-associated signals triggered during register write. The second option is usually better as it provides lower latency. However, if the first option is chosen, then there are 99 control bits.

f) Identify control and status signals needing special handling. For example, in the

```
class Supervisor():
    def __init__(self, registers_handle):
        pass
    def read_counter(self):
        pass
    def reset_counter(self):
        pass
    def read_status_bits(self):
        pass
    def program(self, counter_value, worker_data0, worker_data1):
        pass
    def unprogram(self):
        pass
    def read_workers_ready(self):
        pass
    def set_workers(self, workers):
        pass
```

Listing 2: Example Supervisor Python software interface.

case of the Supervisor there is 48 bits long `counter` value. As the bus width is 32 bits, one needs to provide some mechanism for an atomic read of the counter value to keep the value integrity while reading the counter.

g) Manually decide the register layout. This step involves answering a lot of irrelevant questions. For example, how many registers are needed? Should lower bits of the counter value be placed in the first or the second status register? Should reading the first or the second register of the counter value trigger the atomic read? Should `programmed` and `programmed_in_past` be placed in separate registers or in one of the `counter` value registers to save some address space size? What should be the order of control signals within the control registers? The number of possible implementations is infinite.

Quite a lot of work, even for such a simple module. Moreover, the whole register structure must also be reflected in the software. Figure 1.2 shows a conceptual model of layers in a register-centric approach. The communication interface and interconnect layers are irrelevant in terms of the address space and registers management. Register-centric solutions focus on the module registers and bus fabric layers. They allow describing one or more of these layers and can auto-generate appropriate gateware, firmware, and software. However, these solutions ignore that some signals might need special handling or be a part of some broader context. For instance, a user has to implement atomic reads or writes himself. The same applies to the software responsible for triggering procedures

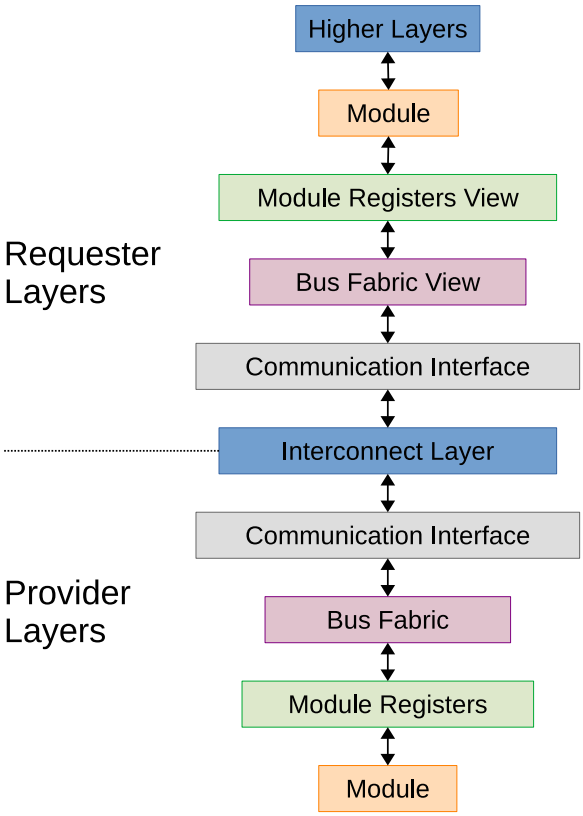


Figure 1.2: Conceptual stack of layers in the register-centric approach.

implemented in the gateware, consisting of multiple control registers. Such an approach is error-prone and leads to duplication of information. For example, the information that some signal needs atomic read is manually implemented in two places, in the firmware source code, and in the software source code.

Working manually on the register layout is also fragile to changes. In the example Supervisor module, there are 96 bits needed for the control signals if `reset_counter, program, unprogram` are implemented as strobe signals associated with given control registers. This is exactly three registers on a 32 bits wide bus. However, should `reset_counter, program, unprogram` be associated with registers storing some data, or maybe with virtual registers (registers with addresses but not storing any data)? What happens if more workers have to be added? The user has to manually add more control registers and adjust the firmware and software accordingly. Yet another question arises. Should the whole, longer `workers_mask` be moved to the new third control register, or maybe just the new extra bits?

Listing 3 shows an example implementation of the software handling Supervisor module in the case of a register-centric approach. It all has to be coded manually. What is worse, in

case of any register changes it also has to be adjusted manually. This is because available solutions are register-centric. They treat registers as a goal, not as a path to an actual goal, which is always the functionality of the data.

The register-centric approach gives much freedom and is highly flexible. On the other hand, it does not look at the registers from the broader context and is unaware of the semantics of the data stored in them. This implies micro-management of registers, generates a lot of irrelevant questions, and is relatively fragile to changes.

Listing 4 presents an example SystemRDL description for example Supervisor. SystemRDL is the only formally defined register-centric format. If there was a need to increase the number of workers above the bus width, then the description would need a relatively lot of adjustments. The register layout is described manually, so the `WORKER_COUNT` macro can no longer be used. Listing 5 presents the file difference that would have to be applied in such a case.

1.3 Functionality-centric approach

The thesis proposes a shift of paradigm leading to a different approach. It looks at the design and modules from the *functionality* point of view. It is the functionality of the data that is in the center. An engineer always thinks about the functionality a given module should serve. The whole register layout is automatically generated based on the declarative description of the provided functionalities. Figure 1.3 shows a conceptual model of layers in the functionality-centric approach. There is an extra data functionality layer compared to the register-centric approach. This is the core layer in this model. The module register layers are automatically generated based on the data functionality layer.

Looking at data from the functionality point of view allows for avoiding register micro-management. Having functionality embedded into the register data notation also helps to prevent information duplication. For example, atomic accesses or procedure calls can be easily automatically generated for both the requester and the provider. This removes a whole surface of potential human mistakes.

Listing 6 presents FBDL description for example Supervisor, and appendix A presents registerification results. If there was a need to change the number of workers, then it would be enough to change the `WORKER_COUNT` constant value, even if the new number was greater than the bus width. Listing 7 presents the file difference that would have to be applied in such a case. As the registerification process is carried out automatically by

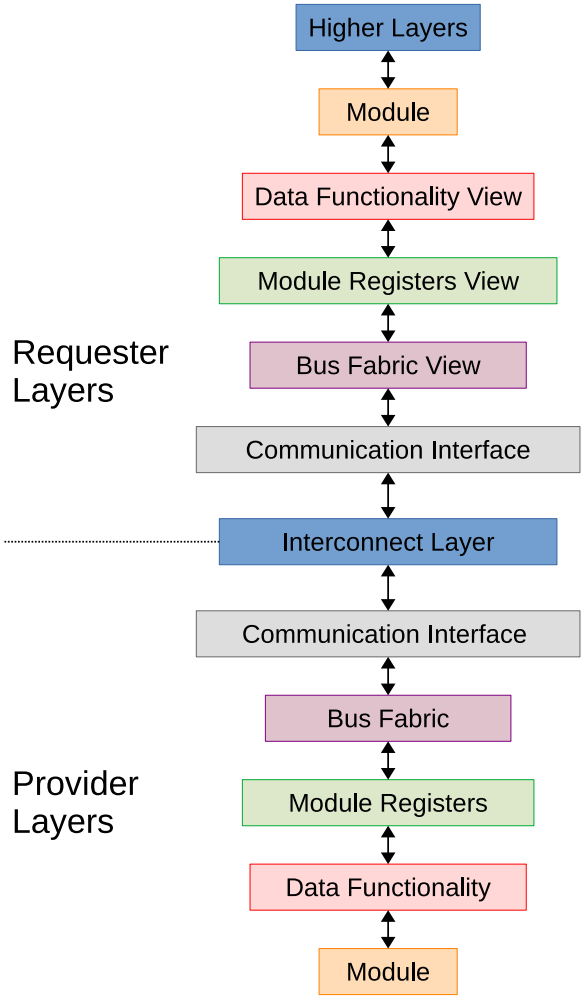


Figure 1.3: Conceptual stack of layers in the functionality-centric approach.

the compiler also the whole register layout is automatically adjusted. There is no need to adapt gateware, firmware or software code manually. As FBDL promotes safety by default, there is also no need to declare `Counter` status to be atomic explicitly. Any data wider than bus width has atomic access unless the user explicitly resigns from it.

```python
class Supervisor:
    def __init__(self, registers_handle):
        self.registers_handle = registers_handle

    def read_counter(self):
        """ To keep counter integrity and perform atomic read, the
            counter register 0 must be read as the first one. """
        counter = self.registers_handle.Counter0.read()
        counter |= self.registers_handle.Counter1.read() << 32
        return counter

    def reset_counter(self):
        self.registers_handle.Reset_Counter.write(0)

    def read_status_bits(self):
        """ Returns tuple (programmed, programmed_in_past). """
        status = self.registers_handle.Status.read()
        return status & 1, status & 2

    def program(self, counter_value, worker_data0, worker_data1):
        """ Program0 register has to be written as the last one, as it has
            strobe signal associated with it, which serves as the arm signal. """
        self.registers_handle.Program2.write((worker_data1 << 12) | worker_data0)
        self.registers_handle.Program1.write(counter_value >> 32)
        self.registers_handle.Program0.write(counter_value & 0xFFFFFFFF)

    def unprogram(self):
        self.registers_handle.Unprogram.write(0)

    def read_workers_ready(self):
        return self.registers_handle.Workers_Ready.read()

    def set_workers(self, workers):
        """ Enable given workers. Workers argument can be a worker number
            or a list of workers numbers. """
        if type(workers) == int:
            workers = [workers]
        mask = 0
        for w in workers:
            mask |= 1 << w
        self.registers_handle.Workers_Mask.write(mask)
```

Listing 3: Example Supervisor software interface implementation in the case of a register-centric approach.

```
addrmap Supervisor {
  name = "Supervisor";
  default regwidth = 32;

  `define WORKER_COUNT 24

  reg empty_strobe_reg_t {
    field {sw = w; hw = na; swacc;} dummy;
  };

  // Counter0 has to be read as the first one to
  // keep counter value integrity.
  reg { field { sw = r; hw = w; } data; } Counter0;
  reg {
    regwidth = 16;
    field {sw = r; hw = w;} data[16];
  } Counter1;
  empty_strobe_reg_t Reset_Counter;

  reg {
    field {fieldwidth = `WORKER_COUNT; sw = w; hw = r;} mask;
  } Workers_Mask;
  // Program0 must be written as the last one,
  // as the write triggers Program procedure.
  reg {
    field {sw = w; hw = r; swacc;} counter_value0;
  } Program0;
  reg {
    regwidth = 16;
    field {sw = w; hw = r;} counter_value1[16];
  } Program1;
  reg {
    field {sw = w; hw = r;} worker_data0[12];
    field {sw = w; hw = r;} worker_data1[12];
  } Program2;
  empty_strobe_reg_t Unprogram;

  reg {
    field {fieldwidth = `WORKER_COUNT; sw = r; hw = w;} mask;
  } Workers_Ready;
  reg {
    field {fieldwidth = 1; sw = r; hw = w;} programmed;
    field {fieldwidth = 1; sw = r; hw = w;} programmed_in_past;
  } Status;
};
```

Listing 4: Example Supervisor SystemRDL description.

```
5,6d4
<    `define WORKER_COUNT 24
<
19,20c17,21
<      field {fieldwidth = `WORKER_COUNT; sw = w; hw = r;} mask;
<    } Workers_Mask;
---
>      field {sw = w; hw = r;} mask;
>    } Workers_Mask0;
>    reg {
>      field {fieldwidth = 1; sw = w; hw = r;} mask;
>    } Workers_Mask1;
37,38c38,42
<      field {fieldwidth = `WORKER_COUNT; sw = r; hw = w;} mask;
<    } Workers_Ready;
---
>      field {sw = r; hw = w;} mask;
>    } Workers_Ready0;
>    reg {
>      field {fieldwidth = 1; sw = r; hw = w;} mask;
>    } Workers_Ready1;
```

Listing 5: Example Supervisor SystemRDL description change for workers count increase above the bus width.

```
Main bus
    Supervisor block
        const WORKER_COUNT = 24

        Counter status; width = 48
        Reset_Counter proc

        Workers_Mask mask; width = WORKER_COUNT
        Program proc
            counter_value    param; width = 48
            worker_data   [2]param; width = 12
        Unprogram proc

        Workers_Ready status; width = WORKER_COUNT
        type status_t status; width = 1; groups = "status"
        programmed         status_t
        programmed_in_past status_t
```

Listing 6: Example Supervisor FBDL description.

```
3c3
<               const WORKER_COUNT = 24
---
>               const WORKER_COUNT = 33
```

Listing 7: Example Supervisor FBDL description change for workers count increase above the bus width.

2 On-chip interconnect architectures

Probably every practical computing system ever created consists of independent components (there is at least some processing unit and a memory). In order to achieve synergy and serve desired functionality, these components must communicate with each other using a set of organized rules (communication protocols or standards). This network of connections is often referred to as system interconnect. The very first interconnect architectures were also called buses. The term „bus" originates from the computer, whose history can be traced back to 1946 [10]. This term is still in common use, although nowadays, bus protocols differ significantly from their ancestors. A bus, in general, is a common pathway through which information flows from one computer component to another. In the early days, computer components were relatively big, and all buses were physically made of copper wires, or later as traces on the printed circuit boards. Initially, those buses served four functions:

1. Data sharing - the primary purpose of every bus. Data processing is the core concept of every computing system. It would not be achievable without data transfer between system components.
2. Addressing - a bus had address lines. This allowed data to be sent to a particular system component to a specific memory location.
3. Clock distribution - a bus provided a system clock signal to synchronize the peripherals attached to it or even to clock the peripheral itself.
4. Power supplying - a bus supplied power to various peripherals connected to it.

The most popular computer expansion buses include ISA [11], EISA [12], MCA [13], VESA [14], SCSI [15], USB [16], and PCI/PCIe [17]. Most of them are not used anymore as they have been replaced with the USB and PCIe. With the advancement of technology, especially integrated circuits technology, it was possible to shrink components of computing systems to the sizes allowing placing multiple of them (or even the whole system) on a single chip. There was still a need to connect system components to enable communication between them. However, traditional microcomputer buses were fundamentally handicapped for use as a SoC interconneciton. This is because they were designed to drive long signal traces and connector system which are highly inductive and capacitive. In this regard, SoC is much simpler and faster. Furthermore, the SoC solutions have a rich set of interconnection resources. These do not exist in microcomputer

buses because they are limited by chips packaging and mechanical connectors. As the existing buses were not optimal for implementation on chips, the interconnect architectures started to be grouped into two classes, the off-chip interconnect architectures, and the on-chip interconnect architectures. The on-chip buses serve the same functions as the off-chip buses except the last one, the power supplying [18]. In the case of SoCs, the power is usually supplied separately via the chip backbone. The clock is also not always distributed, as a bus can be asynchronous [19], but this might also be valid in the case of off-chip buses. Examples of prevailing on-chip buses include ARM AMBA AXI [4], IBM CoreConnect [20], Intel Avalon [21], STMicroelectronics STBus [22], Opencores Wishbone [5], MARBLE (asynchronous) [23].

The following two sections briefly describe two on-chip bus standards, the AXI and the Wishbone. They have been chosen because:

1. they are omnipresent and popular,
2. they have different control logic.

The descriptions are brief because Wishbone revision B4 specification has 128 pages and AMBA AXI specification is 273 pages long, and the subsections' purpose is solely to introduce example bus logic.

2.1 AMBA AXI

The AMBA AXI protocol is copyrighted by the Arm company. Its first version was released in 2003, and its latest version 5 was released in March 2023. In 2021 the specification changed primary terminology, the Master term was replaced with the Manager term, and the Slave term was replaced with the Subordinate term. It is worth mentioning because almost all available materials, except the specification, and available IP cores still use the old terminology. AXI gained a lot of popularity probably because it became de facto the standard for connecting IP cores in FPGA designs utilizing AMD Xilinx or Intel chips. Both companies are the major programmable logic devices market vendors and both offer AXI interconnect cores as well as functional IP cores with AXI interface.

The AXI protocol defines five independent channels:

1. write request (AW),
2. write data (W),
3. write response (B),

4. read reqeust (AR),

5. read data (R).

Request channels carry control information that describes the nature of the data to be transferred. Having independent channels for write and read means that the master can simultaneously write and read the same slave. Write throughput is not limited by read transactions, and read throughput is not limited by write transactions. This is not true, for example, for the Wishbone bus.

The specification does not impose possible system interconnect topologies and only mentions the most popular ones:

1. shared request and data channels,

2. shared request channel and multiple data channels,

3. multilayer, with multiple request and data channels.

Figure 2.1 presents AXI channel architecture of writes. A single transaction might contain multiple transfers. Write transaction completion is signaled only for a complete transaction, not for each data transfer in a transaction.

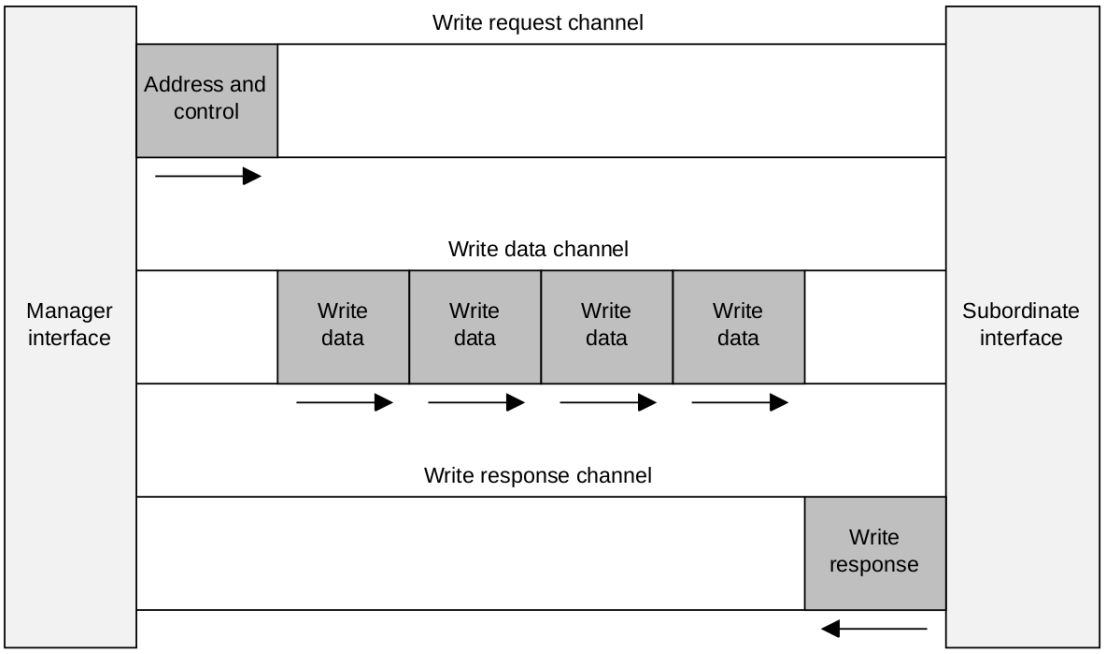


Figure 2.1: AXI channel architecture of writes [4].

Figure 2.2 shows the timing diagram for AXI single read transaction with single data transfer and a bare minimum number of interface signals. It is the simplest possible transaction with the minimum number of channels involved. The manager drives address

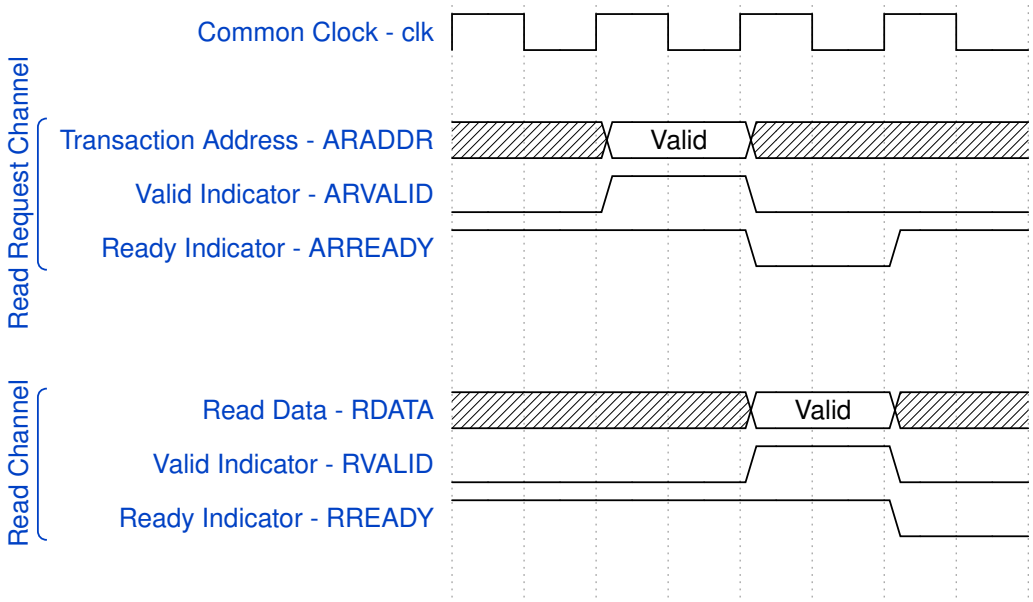


Figure 2.2: AXI single read transaction with single data transfer.

and valid signals in the read request channel and the ready signal in the read channel. The subordinate drives the ready signal in the read request channel and data and valid signals in the read channel. The side driving the ready signal can assert or deassert it anytime, even before valid signal assertion. This means handshaking in AXI can take as little as one clock cycle. A transfer occurs only when both the valid and ready signals are high. The side driving the valid signal must wait for ready assertion after it asserts the valid signal. A deadlock happens if the side driving the valid signal waits for the ready signal assertion before the valid signal assertion and the side driving the ready signal waits for the valid signal assertion before the ready assertion. To prevent such scenarios, the specification states that the valid signal source is not permitted to wait until the ready signal is asserted before asserting the valid signal. The specification forbids combinatorial paths between input and output signals, both on the manager and subordinate sides.

The AMBA AXI specification also defines the AXI-Lite version of the protocol. The AXI-Lite is a subset of AXI where all transactions have one data transfer. It is intended for communication with register-based components and simple memories when bursts of data transfer are not advantageous.

There is also AMBA AXI-Stream protocol defined in the separate specification [24]. AXI-Stream is a point-to-point protocol connecting a single Transmitter and a single Receiver. The terms Master/Manager and Slave/Subordinate are not used in this case, as the data always flows from the Transmitter to the Receiver. The specification of AXI-Stream describes how data is transferred but does not describe the meaning of the data. AXI-Stream is often used in data streaming applications, for example, video processing. Although de-

fined as a separate protocol, the AXI-Stream utilizes the same valid-ready handshaking approach as the standard AXI protocol.

2.2 Wishbone

Wishbone bus architecture was developed by Silicore Corporation. It was put into the public domain in August 2002 by OpenCores (an organization promoting open IP cores development). Wishbone versions till revision 4 were not copyrighted, and revision 4 is copyrighted to the OpenCores. Wishbone can be freely copied and distributed.

Wishbone supports various core interconnection means, including:

1. point-to-point,
2. shared bus,
3. crossbar switch,
4. data flow,
5. off chip.

The possible interconnections are presented in Figure 2.3.

Wishbone supports single read/write transactions, with possible pipelining (introduced in revision 4), block read/write transactions, and read-modify-write transactions. It also supports registered feedback transactions which allow for better throughput.

Figure 2.4 shows the timing diagram for a classic standard single read transaction with the bare minimum number of interface signals. It is the simplest possible transaction. However, it is enough to present how fundamentally different Wishbone control logic is from the AXI control logic. The transaction starts when the cycle signal is asserted by the master on the second clock rising edge. The master also drives the address bus, write enable and asserts the strobe signal to inform the slave that signals are valid and can be read. The slave drives data on the third clock rising edge and asserts the acknowledgment signal to inform the master that data is valid. The slave may wait before asserting the acknowledgment signal in order to throttle the transaction speed.

Compared to the AXI, the handshaking in Wishbone is related to the transaction as a whole. There is no separate handshaking for requests, data, and write response.

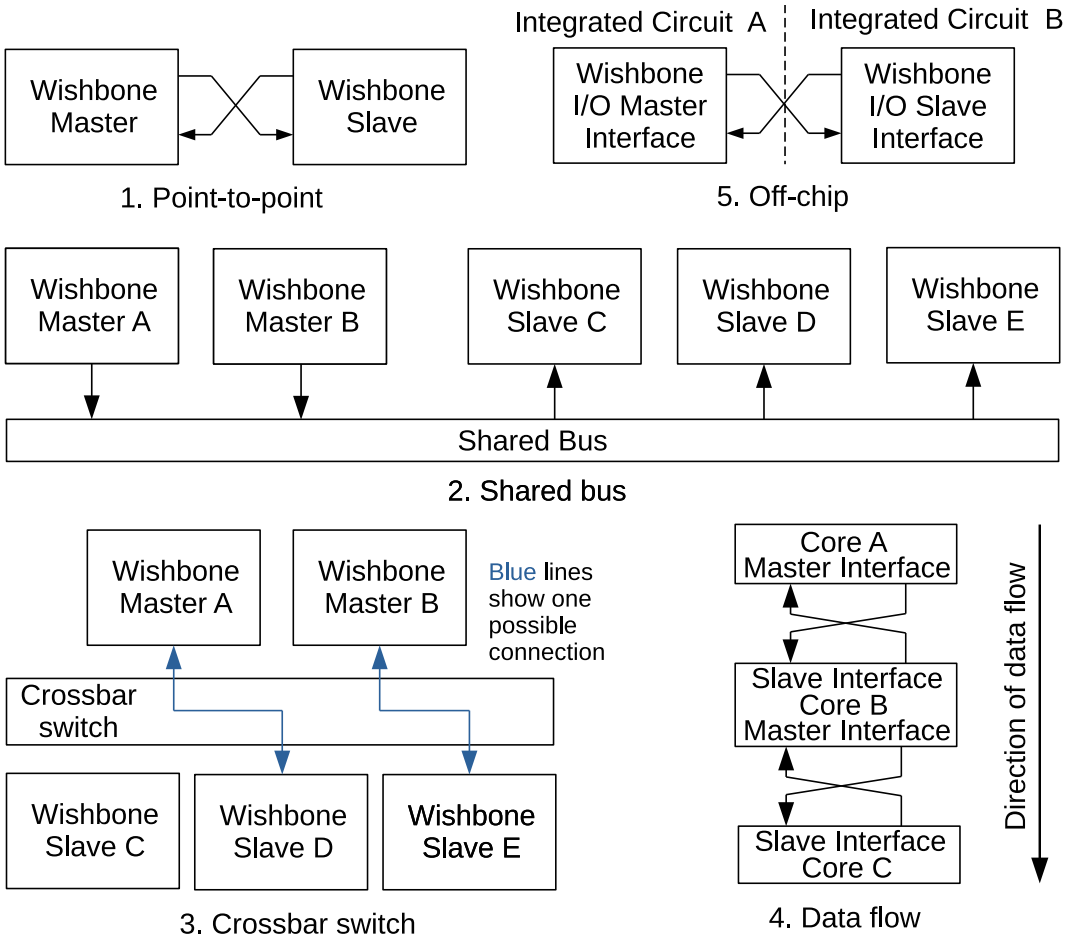


Figure 2.3: Possible Wishbone interconnections.

2.3 Network on Chip

The network on chip is an on-chip interconnect architecture trying to overcome the limits of the traditional bus architectures. The problem was observed and reported in the late 1990s, and was initially addressed in the early years of the 21st century [25, 26, 27, 28]. The most popular drawbacks of the traditional bus architectures that NoC tries to solve include:

1. Limited bandwidth shared by all attached units.
2. Decrease of the maximum frequency with the increase of the number of modules connected to the bus. Every module adds parasitic capacitance, therefore the electrical performance degrades with the increase of modules number.
3. IPs interface incompatibility. The 32 bits AXI Lite master will simply not work with the 64 bits Wishbone slave in a traditional bus architecture without an extra bridge. In the NoC approach each network node can have an individual interface

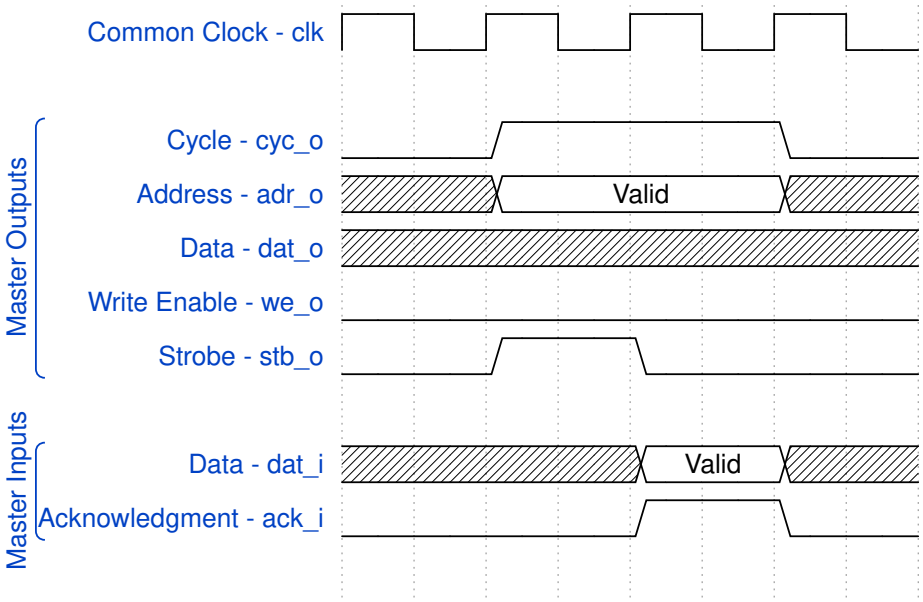


Figure 2.4: Wishbone classic standard single read transaction.

for local registers access.

4. Coupled transaction, transport, and physical activities. Changes to the bus physical implementation can have serious ripple effects upon the implementation of the higher-level bus behaviors. NoC distinguishes transaction, transport, and physical layers that can be adjusted or improved independently.

However, NoC is mainly used in cases of high bandwidth performance critical heterogeneous SoC applications. Even homogeneous designs focused on accelerating the processing of gigabytes or terabytes of data (usually implemented using HLS technique) do not use NoC, but rather different types of AXI interfaces depending on the nature and amount of data being exchanged between modules [29]. This is because NoC is not free of drawbacks. The most popular ones are:

1. Latency increase due to the internal network connections and routing algorithms.
2. Increased resource utilization compared to the traditional bus architectures.
3. Increased overall system complexity.

There are numerous different NoC topologies [30, 31, 32, 33, 34, 35]. The most popular ones include: ring, octagon, star, mesh, torus, folded torus, butterfly, binary tree, fat tree, cube, crossed cube, hypercube, reduced hypercube, reduced mesh and cluster-based hybrid, mesh connected ring, cmesh.

Although the NoC architecture was inspired by well-known computer networks such as LAN or WAN, it differs significantly from them. This is because the implementation of the

protocols used in these networks, such as IP [36] or TCP [37], would consume a relatively large amount of resources and would require significant buffering capabilities. NoC packet typically consists of a header and payload data. The header must include at least the address of the destination node, but it often also includes data length, data tags, and the address of the source node. How the data is routed via the network depends on the routing algorithm. Although the macro-level architecture of the NoC differs significantly from the traditional bus architecture, the packet data still has to be somehow distributed inside the module attached to the network via the network interface. There are two standard ways to achieve this. The first one is dataflow communication, and the second one is address space communication. This is exactly what traditional buses were designed for. So, in the end, the traditional bus architectures are still used within the NoC architectures. However, their scope is limited to the single network nodes. Figure 2.5 presents an example 12 nodes network on chip with the mesh topology.

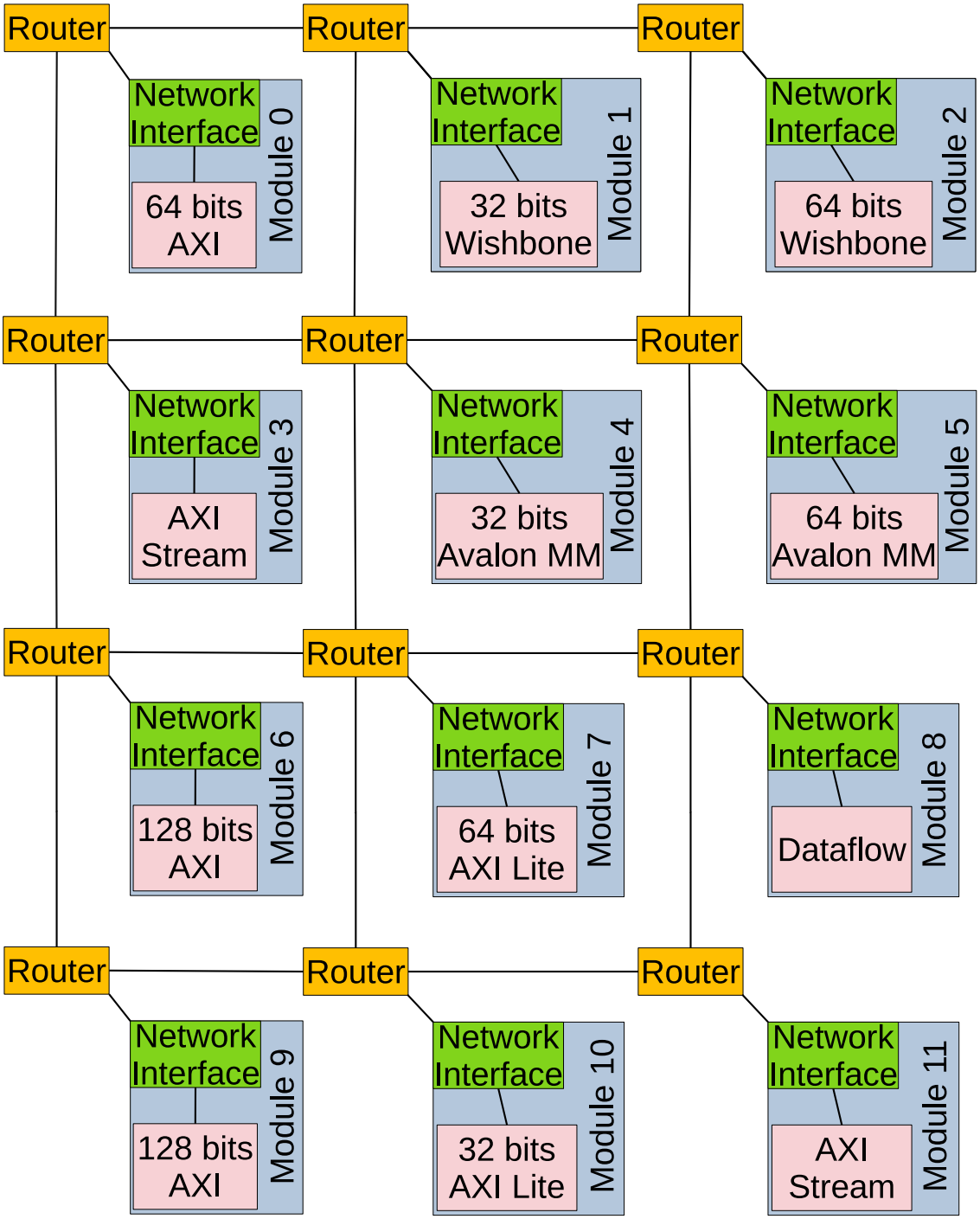


Figure 2.5: Example 12 nodes mesh network on chip.

3 Prior art

This chapter presents the current state of the art. The term „tool" is used for all solutions, although not all are strictly tools. Moreover, some are standalone entities, while others are a part of more extensive tools. Each tool has been designed and implemented by different teams. Although their main goal is the same, they sometimes put an accent on diverse areas. As a result, relative comparison is not always straightforward. This is why they are rather matched against a generic template. **Nonetheless, none of the available solutions offers a functional view of data placed in the registers. They are registers-centric.** The description of each tool is prepended with a table summarizing its capabilities. The order of analysis is alphabetical.

It is important to mention that all described tools and solutions are in continuous development, so some of their features might have changed, or new features might have been added since they were described. It is also worth mentioning that if tool T claims support for feature F or language L, then it might not be a full support, as all such tools are implemented in an incremental fashion. It does not indicate the weakness of the tools, but rather shows a pragmatic approach to the problem. There would be no technical progress in the described field if the tools were usable only when they were 100 % complete.

Table 3.1 presents the result of the review of existing solutions. Comparing bus and register management tool features is a challenging task. First of all, none of the register-centric tools, except SystemRDL, has formal specification. The implementation is the specification. What is more, most of the tools target only a limited set of hardware description or programming languages, and they are usually tailored to these languages. Comparing features of FBDL with register-centric tools is also not straightforward, as FBDL is functionality-centric and has a different paradigm. For example, some of the tools allow data value range constraining. However, it works only for data fitting a single register, whereas in FBDL, it works for data of any width. Partial support means that a given feature is available only to some extent. For example, tools utilizing YAML [38] format support parametrization achieved using YAML syntax. However, they do not provide any extra parametrization mechanism, and full design parametrization is not possible solely with YAML inheritance.

	airhdl	AGWB	AutoFPGA	Cheby	Corsair	FPGA Vendors	hdl_registers	II & CII	IP-XACT	Opentitan Register Tool	Register Wizard	RgGen	SystemRDL	vhdMMIO	wbgen2	FBDL
Register Requires Fields	Y	N	N	N	Y	U	Y	N	N	Y	N	Y	Y	N	Y	N
Bit-fields	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Hierarchy Description	N	Y	Y	Y	N	Y	N	Y	Y	N	N	N	Y	N	N	Y
Design Parametrization	P	Y	Y	P	P	P	N	Y	Y	N	N	P	Y	P	N	Y
Interrupts	Y	N	Y	N	U	Y	P	N	Y	Y	Y	P	Y	Y	Y	Y
Memory	Y	Y	Y	Y	N	Y	N	Y	Y	Y	N	Y	Y	N	Y	Y
Constants	N	Y	Y	N	Y	N	Y	Y	Y	Y	N	N	N	N	N	Y
Expressions	N	Y	Y	N	Y	N	Y	Y	Y	Y	N	Y	Y	N	N	Y
Enumeration Types	N	N	Y	Y	Y	N	N	Y	Y	Y	N	N	Y	N	N	N
Value Range Constraints	Y	N	Y	Y	Y	N	N	N	U	Y	N	Y	Y	N	Y	Y
Addressing Modes	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	DoC
Manual Addressing	Y	N	Y	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y	N	N
Package System	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y
Formally Specified	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	Y
Free and Open Source	N	Y	Y	Y	Y	N	Y	N	DoC	Y	Y	Y	DoC	Y	Y	Y
Actively Maintained	Y	Y	Y	Y	Y	Y	Y	U	Y	Y	N	Y	Y	N	N	Y

Table 3.1: Comparison of some of the features of the bus and register management tools (Y - yes, N - no, DoC - Depends on Compiler, P- Partial, U - Unclear).

3.1 airhdl

The airhdl [39] is a web-based AXI4 VHDL/SystemVerilog register generator tool. It also has a command line version, requiring Java runtime version 8 or higher, accepting register specification in JSON [40] format. It supports code generation for SystemVerilog, VHDL, C/C++, HTML [41] or Markdown documentaiton or transformation to IP-XACT XML [42] format. The tool is closed source, and any plan except the Free one is paid. The main website has a demo video based upon which it is clear that the tool follows the register-centric approach.

3.2 Address Generator for Wishbone

The AGWB [3, 43], the successor of addr_gen [44], facilitates the automated generation of the control system's HDL and software components based on the XML system description. It supports code generation for VHDL, C, Python, Forth, XML register map, and HTML for documentation.

Listing 8 presents an example AGWB registers description in XML format. This snippet is taken directly from the DAQ readout chain for the STS being prepared for the CBM experiment at GSI Darmstadt. The `hctsp_software_command_slot` block has three control registers with an extra strobe signal associated with the `control` register. What is not seen and can not be deduced from the description is the that all three control registers constitute a broader context. Namely, they are all used to pass arguments to the procedure sending commands to the set of front-end ASICs. None of the control registers makes sense without the remaining two registers. What is more, as the `control` control register has an associated strobe signal (`stb="1"`) it must be written as the last of the three registers. However, as the approach is register-centric, the correct write access order must be coded manually. It leaves room for the programmer's mistakes. If `control_frames` registers are written before the `control` register, the system „almost works". The first command reports CRC error. However, later commands are sent correctly with an extra one command delay, unless the set of destination front-end ASICs changes. In such cases valid commands are sent to the invalid set of ASICs, and no error is reported. These types of bugs can be hard and time-consuming to debug, as there is implicit state storage between commands in case of incorrect register write order. This kind of mistake happened to the author during the development and made him think that there must be a better way to describe data stored in the registers. Listing 9 shows a snippet of a Python method belonging to a Command class used to send that to a set of front-end ASICs. As the

access order has to be implemented manually, it is relatively easy to write `control` before `control_frames` by mistake.

```
<block name="hctsp_software_command_slot">
    <creg name="control" stb="1" default="0x0">
        <field name="chip_address"    width="4"/>
        <field name="downlink_mask"   width="12"/>
        <field name="group_mask"      width="8"/>
        <field name="sequence_number" width="4"/>
    </creg>
    <creg name="control_frame" reps="2" default="0x0">
        <field name="request_type"    width="2"/>
        <field name="request_payload" width="15"/>
        <field name="crc"             width="15"/>
    </creg>
</block>
```

Listing 8: Example AGWB description in XML format.

```
def send(self, handle):
    for i in range(0,2):
        handle.control_frame[i].writeb(
            (self.crcs[i] << 17) |
            (self.payloads[i] << 2) |
            self.request_types[i]
        )
    handle.control.writeb(
        (self.sequence_number << 24 ) |
        (self.group_mask << 16) |
        (self.downlink_mask << 4) |
        self.chip_address
    )
```

Listing 9: Snippet of Python method sending commands to the set of font-end ASICs.

3.3 AutoFPGA

AutoFPGA [45] is an FPGA design automation routine. AutoFPGA aims to take a series of bus component configuration files and compose a design consisting of the various bus components linked together in logic, having an appropriate bus interconnect, and more. AutoFPGA is much more than a register generation or bus management tool. It is more like a uniform framework for implementing FPGA designs. However, it is considered prior art in this dissertation because register and bus management aspects are a significant part.

AutoFPGA files used for design generation contains a lot more information than simply register definitions. Listing 10 presents a snippet of the AutoFPGA documentation regarding the registers description. Listing 11 presents a snippet, regarding register macros, of atuomatically generated `regdefs.h` file. This is a standard low level register-centric approach.

```
REGDEFS.H.INCLUDE  Placed at the top of the regdefs.h file
REGS.NOTE   A comment to be placed at the beginning of the register
            list for this peripheral
REGS.N      The number of registers this peripheral has.
            AutoFPGA will then look for keys of the form
            REGS.0 through REGS.(REGS.N-1).
REGS.0...?  Describes a register by name. The first value is the
            offset within the address space of this device.
            The second token is a string defining a C #def'd
            constant.  The third and subsequent tokens represent
            human readable names that may be associated with
            this register.
REGDEFS.H.DEFNS   Placed with other definitions within regdefs.h
REGDEFS.H.INSERT  Placed in regdefs.h following all of the
                  definitions
I may change this to the following notation, though:
REGSDEFS.NOTE
REGS.<name>.ADDR      # Offset within the peripheral
REGS.<name>.UNAME(s)  # User-readable name
REGS.<name>.DESC(ription for LaTeX)
```

Listing 10: AutoFPGA documentation on registers definition.

```
//
// Register address definitions, from @REGS.#d
//
#define R_BUSERR   0x00080000 // 00080000, wbregs names: BUSERR
#define R_FIXEDATA 0x00080004 // 00080004, wbregs names: FIXEDATA
#define R_PWRCOUNT 0x00080008 // 00080008, wbregs names: PWRCOUNT
#define R_RAWREG   0x0008000c // 0008000c, wbregs names: RAWREG
#define R_SIMHALT  0x00080010 // 00080010, wbregs names: SIMHALT
#define R_SPIO     0x00080014 // 00080014, wbregs names: SPIO
#define R_VERSION  0x00080018 // 00080018, wbregs names: VERSION
#define R_BKRAM    0x00100000 // 00100000, wbregs names: RAM
```

Listing 11: Snippet of `regdefs.h` file automatically generated by AutoFPGA regarding register macros.

3.4 Cheby

The Cheby [46, 47], the successor of the Cheburashka [48], aims at defining a file format to describe the hardware-software interface (the memory map), and a set of tools to generate

HDL, drivers and documentation from the files. It uses YAML as a register description file format.

Listing 12 presents an example Cheby registers description in YAML format. The Cheby generator is capable of generating a C++ library. The library provides a hierarchical interface over every memory node defined in a memory map. The library interface allows software developers to read or write to registers and their fields, having all low-level bit-shifting and masking operations done by the wrapper. This is a higher abstraction than addresses, masks, and shifts generation and implementing the access manually. However, there is no way to inform Cheby that a particular set of registers may form a broader context and that they must always be read or written as a whole in the correct order. The Cheby is a representative of a typical register-centric approach with abstracted access to a single register or bit field.

```
memory-map:
  bus: wb-32-be
  name: gpios
  x-hdl:
    busgroup: True
  children:
  - reg:
      name: inputs
      description: A register
      type: unsigned
      width: 32
      access: ro
  - reg:
      name: outputs
      type: unsigned
      width: 32
      access: rw
  - submap:
      name: gpios_axi4
      size: 0x40
      description: An AXI4-Lite bus
      interface: axi4-lite-32
```

Listing 12: Example Cheby registers description in YAML format.

3.5 Corsair

The Corsair [49] is a tool for creating and maintaining control and status register maps for HDL projects. The Corsair accepts JSON, YAML and plain text tables as input formats.

It is capable of generating files for Verilog, VHDL, C, Python, and documentation written in Markdown.

Listing 13 presents an example Corsair registers description in YAML format. Listing 14 presents generated C header file. This is a traditional, register-centric approach. An engineer describes registers at the lowest level and as a result gets information on addresses, masks, and shifts (LSB in this case). Later, this information is used in the manual implementation of the software accessing the data. Corsair also allows for code generation for Python. In this case, proper addressing, masking, and shifting are automatically generated. However, there is no way to define a broader context consisting of multiple registers.

3.6 Tools provided by FPGA vendors

Development environments provided by FPGA vendors offer some capabilities for bus and register management (for example, Block Designer - AMD Xilinx, Platform Designer - Intel). They allow for connecting master, slave, and bus fabric components using GUI tools. Figure 3.1 shows a simple system designed in Vivado Block Designer, containing blocks interconnected via the local AXI bus.

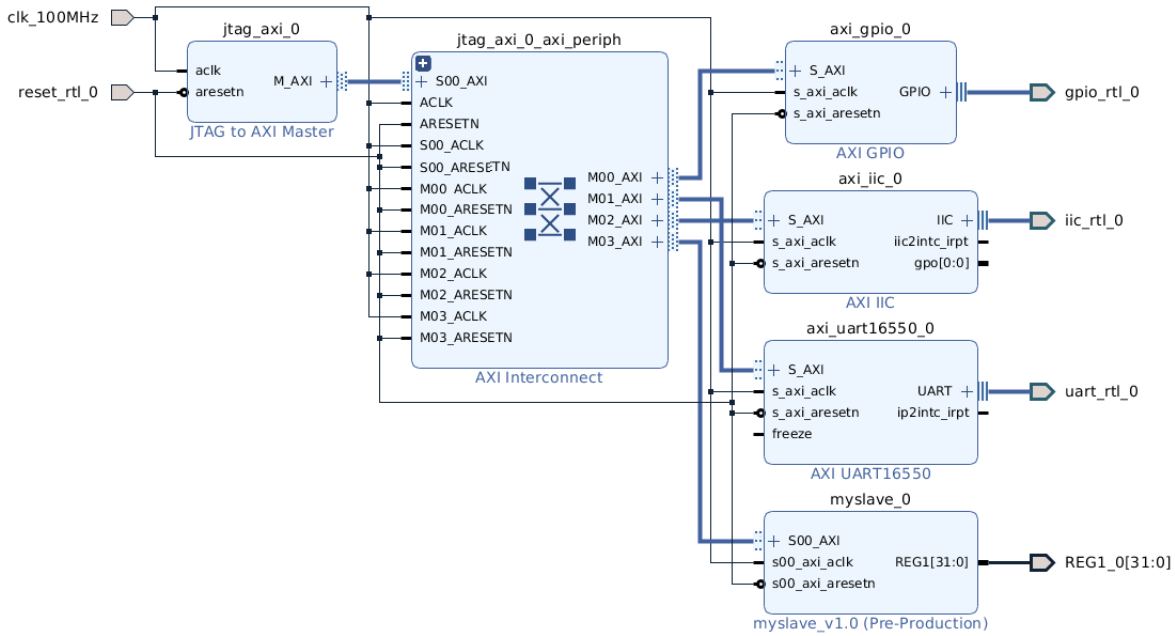


Figure 3.1: A simple design created using Block Designer in Xilinx Vivado environment [50].

Figure 3.2 shows the address table generated automatically by that tool. It is possible to adjust component address spaces manually. In the case of ready-to-use IP cores included

```yaml
regmap:
-   name: DATA
    description: Data register
    address: 4
    bitfields:
    -   name: FIFO
        description: Write to push value to TX FIFO, read to get data from RX FIFO
        reset: 0
        width: 8
        lsb: 0
        access: rw
        hardware: q
        enums: []
    -   name: FERR
        description: Frame error flag. Read to clear.
        reset: 0
        width: 1
        lsb: 16
        access: rolh
        hardware: i
        enums: []
-   name: STAT
    description: Status register
    address: 12
    bitfields:
    -   name: BUSY
        description: Transciever is busy
        reset: 0
        width: 1
        lsb: 2
        access: ro
        hardware: ie
        enums: []
    -   name: RXE
        description: RX FIFO is empty
        reset: 0
        width: 1
        lsb: 4
        access: ro
        hardware: i
        enums: []
```

Listing 13: Example Corsair register description in YAML format.

in the development environments, the register description is included in the core configuration file (vendor-specific format). The tools can generate device tree descriptions and

access codes, for example for Linux drivers. However, in the case of custom components, only the address space is reserved. The user still needs a custom mechanism for register management within the component.

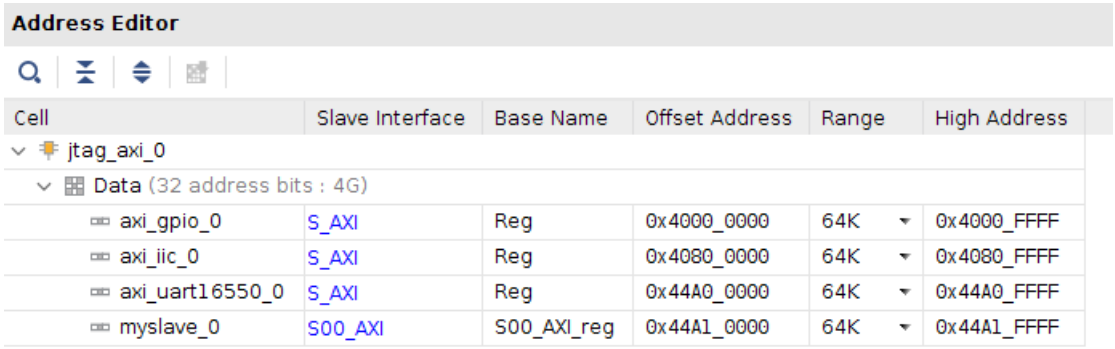


Figure 3.2: The address space allocation for the simple design from figure 3.1.

Unfortunately, it is difficult to manage designs using vendor EDA tools when the complexity of the system grows, especially when the number of blocks or nested subblocks is parameterized. Moreover, heavy reliance on GUI makes it incompatible with purely hdl-based or script-driven development flow. Opening the GUI application to apply changes is also relatively time-consuming compared to applying a change in a text file. A single change in a GUI widget often leads to multiple changes in project files. This makes tracking changes of the design using revision control system more complicated compared to the traditional approach in which configuration is done using textual files.

3.7 hdl_registers

The hdl_registers [51] project is an open-source HDL register generator. It is capable of generating files for C, C++, HTML (documentation), VHDL, and Python. Hdl_registers accepts register description in the TOML file format. It is also possible to work directly with the Python API without providing a TOML file.

Listing 15 presents an example hdl_registers registers description in TOML format. Listing 16 presents generated C header file. This is a typical, register-centric approach. Information on addresses, shifts, and masks is generated, and the user has to utilize it to write the access code.

However, the hdl_registers is also able to generate code with higher abstraction for C++ and Python. Listing 17 presents generated C++ header file. The higher abstraction is achieved by generating getters and setters for registers and bit fields. No need to use

address, shift, and mask values directly. However, this approach is still register-centric, as getters and setters are generated only for registers and all data fitting within a single register. If a counter width were equal to two registers, the user would have to manually glue read access by calling two getters. There is also no way to provide information on whether the read must provide atomicity in such a case. In the case of atomicity, it must be manually coded in HDL.

What is more, the generated interface is distinct for different targets (C vs C++). However, the nature of the data stored within the registers does not inherit from the language used to implement the access but from the functionality it serves. If the generated C++ code allows directly reading bit fields suiting a single register, why does the generated C code enforce the user to apply shifting and masking manually?

3.8 II & CII

The II (Internal Interface) [52] and CII (Component Internal Interface) are solutions developed for electronic systems created for CMS and DESY [53]. Although it is closed-source, its approach has been described in papers [54, 55]. The description in the papers does not allow for the reconstruction of the internal logic of the tool. However, based on the attached figures and description, it looks like CII approach is register-centric with abstracted away register width. A user is provided with the concept of records. A record has a type and width that can be greater than the width of the single register. Whether the access to the record is atomic is unclear based on the available information. The user does not define the functionality of the data placed in the record but the access rights.

3.9 IP-XACT

The IP-XACT [56] is neither a bus and register management tool nor a design framework. It is more like an interchangeable IP documentation format. The focus of the standard is to act as an electronic databook - its primary function is to „document what's there" [57]. However, it is mentioned as prior art as there were at least two [58, 59] tries to implement the bus and registers code generators from the IP-XACT registers description. IP-XACT uses XML file format for data representation. These XML files are usually highly unreadable as they are intended for machines. To make any use of them, special tools, such as Kactus2 [60], are needed. These are usually GUI programs with a friendly user interface using IP-XACT XML file format as an input/output file format.

3.10 Opentitan Register Tool

Opentitan [61, 62] is an open-source silicon Root of Trust project. As such, it has a subpart named the Register Tool [63] that can be used as a standalone tool. It uses Hjson (a syntax extension to JSON) as an input file format for the register description. It is capable of generating files for HTML documentation, standard JSON, Verilog, and C.

Listing 18 presents an example Opentitan register description. The Opentitan Register Tool can be used to generate C header files. The generated C header file contains information on registers addresses, bit field shifts, and masks and may have information on enumerated names and values. This is a typical register-centric approach. A developer has to use the address, mask, and shift information to implement firmware or software access code manually.

3.11 Register Wizard

The Register Wizard is a free tool from the Inventas (formerly Bitvis) company. It has been abandoned, but the company sends it on request [64]. The presentation links are also valid [65, 66]. It uses Model Description File format, which is actually a JSON format. It is capable of generating files for VHDL, C header, and documentation written in Office Open XML format. Listing 19 shows a register definition template from the Register Wizard documentation on defining registers and bit fields. This is a typical register-centric approach. The user describes particular registers, their addresses, access properties, internal bit fields, etc. The generated C header file includes information on addresses, masks, and shifts.

3.12 RgGen

RgGen [67] automatically generates source code related to configuration and status registers (CSR). RgGen is capable of generating files for SystemVerilog, VHDL, UVM, C, and register map documents written in Markdown.

What makes RgGen unique is the fact that register map specifications can be written in multiple formats, such as Ruby language API, YAML, JSON, TOML, Spreadsheet (XLSX, XLS, OSD, CSV), SiFive DUH (Design u Hardware) [68].

Listing 20 presents an example RgGen registers description in YAML format. Listing

21 presents generated C header file. This is a traditional, register-centric approach. An engineer describes registers at the lowest level and as a result gets information on addresses/offsets, masks, and widths. Later on, this information is used in the manual implementation of software accessing the data.

3.13 SystemRDL

The SystemRDL [69] is different from all other available solutions as it is the only one having an official specification. The SystemRDL is a language aimed at the detailed description of the registers. Version 2.0 supports the parameterization of components and the structure of the system. SystemRDL is by far the most advanced solution with the greatest number of features but also the most complex. Whether all of these features should be a part of the bus and register management tool is a separate topic. However the fact that most SystemRDL compilers do not implement all features makes the question at least partially justified. There are some closed-source paid [70, 71, 72] as well as open-source free SystemRDL compilers [73, 74, 75]. Listing 4 presents an example SystemRDL description.

The SystemRDL standard allows users to extend components with custom properties. The user-defined properties allow to add additional meaning to the data. This mechanism is quite flexible but also has some drawbacks. The first one is that user-defined properties are compiler specific. The second one is description verbosity, as SystemRDL is quite verbose even without extra custom properties. One reason for such a state might be that each register in SystemRDL must have at least one field, and registers without fields are not allowed.

3.14 vhdMMIO

VhdMMIO [76] is a tool to generate AXI4-Lite MMIO infrastructure based on YAML specification files. A single register file describes registers for a single AXI4-Lite slave and maps to a single VHDL entity. VhdMMIO is also capable of generating HTML for documentation. Listing 22 presents an example vhdMMIO registers description in YAML format.

VhdMMIO is distinct from all other register-centric tools. This is because vhdMMIO has the concept of registers/fields behavior (vhdMMIO uses the term *field* for the register and *subfield* for the field). This allows generating more gateware code automatically. However, as the behavior is bound to the particular field or register and not to the data,

it is impossible to describe broader data contexts occupying more than one register, such as procedures or streams, or even single data occupying more than one register, without explicitly defining particular registers. The user also has to assign addresses to particular fields explicitly. This makes vhdMMIO still a register-centric approach as the user thinks and acts in the following order: define register, then define data, then define the behavior of the data. Whereas in FBDL user thinks and acts in the following order: define data, then define the functionality of the data. All work related to the registers is then done automatically. The concept of a register is not even present in the thought flow.

3.15 wbgen2

Wbgen2 [77] is one of the first open-source tools for bus and register management. The slave description is prepared in the custom format and may contain registers, fields, interrupts, memory blocks, and FIFO. The wbgen2 is capable of generating the slave HDL code in VHDL or Verilog and C headers for integration. Additionally, it may generate the documentation for the created slave in Latex, Texinfo, or HTML. Wbgen2 does not support vectors of registers or blocks, or nested blocks. Listing 23 presents example registers description in wbgen2 specific format. The generated C code contains information on register and field addresses and masks.

```
#ifndef __REGS_H
#define __REGS_H
#define __I  volatile const // 'read only' permissions
#define __O  volatile       // 'write only' permissions
#define __IO volatile       // 'read / write' permissions

#include "stdint.h"
#define CSR_BASE_ADDR 0x0

#define CSR_DATA_ADDR 0x4
#define CSR_DATA_RESET 0x0
typedef struct { uint32_t FIFO : 8; uint32_t :16; uint32_t FERR : 1; } csr_data_t;
#define CSR_DATA_FIFO_WIDTH 8
#define CSR_DATA_FIFO_LSB 0
#define CSR_DATA_FIFO_MASK 0x4
#define CSR_DATA_FIFO_RESET 0x0
#define CSR_DATA_FERR_WIDTH 1
#define CSR_DATA_FERR_LSB 16
#define CSR_DATA_FERR_MASK 0x4
#define CSR_DATA_FERR_RESET 0x0

#define CSR_STAT_ADDR 0xc
#define CSR_STAT_RESET 0x0
typedef struct
    { uint32_t :2; uint32_t BUSY : 1; uint32_t :4; uint32_t RXE : 1; } csr_stat_t;
#define CSR_STAT_BUSY_WIDTH 1
#define CSR_STAT_BUSY_LSB 2
#define CSR_STAT_BUSY_MASK 0xc
#define CSR_STAT_BUSY_RESET 0x0
#define CSR_STAT_RXE_WIDTH 1
#define CSR_STAT_RXE_LSB 4
#define CSR_STAT_RXE_MASK 0xc
#define CSR_STAT_RXE_RESET 0x0

typedef struct {
    __IO uint32_t RESERVED0[1];
    union { __IO uint32_t DATA; __IO csr_data_t DATA_bf; };
    __IO uint32_t RESERVED1[1];
    union { __I uint32_t STAT; __I csr_stat_t STAT_bf; };
} csr_t;

#define CSR ((csr_t*)(CSR_BASE_ADDR))
#endif /* __REGS_H */
```

Listing 14: Example C header file generated using Corsair (comments removed for brevity).

```
[register.configuration]
mode = "r_w"
# This will allocate a bit field named "enable" in the "configuration" register.
[register.configuration.bit.enable]
default_value = "1"
# This will allocate a bit vector field named "data_tag" in the
# "configuration" register.
[register.configuration.bit_vector.data_tag]
width = 4
default_value = "0101"

[register.status]
mode = "r"
[register.status.bit.idle]
default_value = "1"
[register.status.bit.stalling]
description = "'1' if the module is currently being stalled."
[register.status.bit_vector.counter]
width = 8
```

Listing 15: Example hdl_registers description in TOML format.

```c
#ifndef EXMPL_REGS_H
#define EXMPL_REGS_H

#define EXMPL_NUM_REGS (2u)

typedef struct example_base_addresses_t {
  uint32_t read_address;
  uint32_t write_address;
} example_base_addresses_t;
typedef struct example_regs_t {
  uint32_t configuration;
  uint32_t status;
  example_base_addresses_t base_addresses[2];
} example_regs_t;

#define EXMPL_CONFIGURATION_INDEX (0u)
#define EXMPL_CONFIGURATION_ADDR (4u * EXMPL_CONFIGURATION_INDEX)

#define EXMPL_CONFIGURATION_ENABLE_SHIFT (0u)
#define EXMPL_CONFIGURATION_ENABLE_MASK (0b1u << 0u)
#define EXMPL_CONFIGURATION_ENABLE_MASK_INVERSE (~EXMPL_CONFIGURATION_ENABLE_MASK)

#define EXMPL_CONFIGURATION_DATA_TAG_SHIFT (1u)
#define EXMPL_CONFIGURATION_DATA_TAG_MASK (0b1111u << 1u)
#define EXMPL_CONFIGURATION_DATA_TAG_MASK_INVERSE (~EXMPL_CONFIGURATION_DATA_TAG_MASK)

#define EXMPL_STATUS_INDEX (1u)
#define EXMPL_STATUS_ADDR (4u * EXMPL_STATUS_INDEX)

#define EXMPL_STATUS_IDLE_SHIFT (0u)
#define EXMPL_STATUS_IDLE_MASK (0b1u << 0u)
#define EXMPL_STATUS_IDLE_MASK_INVERSE (~EXMPL_STATUS_IDLE_MASK)

#define EXMPL_STATUS_STALLING_SHIFT (1u)
#define EXMPL_STATUS_STALLING_MASK (0b1u << 1u)
#define EXMPL_STATUS_STALLING_MASK_INVERSE (~EXMPL_STATUS_STALLING_MASK)

#define EXMPL_STATUS_COUNTER_SHIFT (2u)
#define EXMPL_STATUS_COUNTER_MASK (0b11111111u << 2u)
#define EXMPL_STATUS_COUNTER_MASK_INVERSE (~EXMPL_STATUS_COUNTER_MASK)

#endif // EXMPL_REGS_H
```

Listing 16: Example C header file generated using hdl_registers (comments removed for brevity).

```cpp
#pragma once
#include <cassert>
#include <cstdint>
#include <cstdlib>

namespace fpga_regs {

class IExample {
public:
  static const size_t num_registers = 2uL;

  // Length of the "base_addresses" register array
  static const size_t base_addresses_array_length = 3uL;

  virtual ~IExample() { }

  virtual uint32_t get_configuration() const = 0;
  virtual void set_configuration(uint32_t register_value) const = 0;

  virtual uint32_t get_configuration_enable() const = 0;
  virtual uint32_t get_configuration_enable_from_value(
    uint32_t register_value) const = 0;
  virtual void set_configuration_enable(uint32_t field_value) const = 0;
  virtual uint32_t set_configuration_enable_from_value(
    uint32_t register_value, uint32_t field_value) const = 0;
  virtual uint32_t get_configuration_data_tag() const = 0;
  virtual uint32_t get_configuration_data_tag_from_value(
    uint32_t register_value) const = 0;
  virtual void set_configuration_data_tag(uint32_t field_value) const = 0;
  virtual uint32_t set_configuration_data_tag_from_value(
    uint32_t register_value, uint32_t field_value) const = 0;

  virtual uint32_t get_status() const = 0;
  virtual uint32_t get_status_idle() const = 0;
  virtual uint32_t get_status_idle_from_value(uint32_t register_value) const = 0;
  virtual uint32_t get_status_stalling() const = 0;
  virtual uint32_t get_status_stalling_from_value(uint32_t register_value) const = 0;
  virtual uint32_t get_status_counter() const = 0;
  virtual uint32_t get_status_counter_from_value(uint32_t register_value) const = 0;
};

} /* namespace fpga_regs */
```

Listing 17: Example C++ header file generated using hdl_registers (comments removed for brevity).

```
{ name: "REGA",
  desc: "Description of register",
  swaccess: "rw",
  resval: "42",
  fields: [
    { bits: "15:0",
      name: "RXS",
      desc: "Description of bit field",
    }
    { bits: "16",
      name: "ENRXS"
    }
}
```

Listing 18: Example Opentitan register description in Hjson format.

```
"registers": [{
    "name": "",
    "configuration": {},
    "address": "",
    "summary": [],
    "description": [],
    "width": ,
    "access": "",
    "signal": ""
    "reset": "",
    "location": "",
    "coreSignalProperties": {},
    "fields": [{
        "name": "",
        "position": "",
        "description": [],
        "access": "",
        "signal": "",
        "reset": "",
        "location": "",
        "coreSignalProperties": {}
    }]
}]
```

Listing 19: Snippet from the Register Wizard documentation on defining registers and bit fields.

```yaml
register_blocks:
  - name: block_0
    byte_size: 256
    registers:
    - name: register_0
      bit_fields:
      - {name: bit_field_0, bit_assignment: {width: 4}, type: rw  , initial_value: 0}
      - {name: bit_field_1, bit_assignment: {width: 2}, type: wrs , initial_value: 0}
      - {name: bit_field_2, bit_assignment: {width: 2}, type: rowo, initial_value: 0}
    - name: register_1
      bit_fields:
      - <<:
        - { bit_assignment: { lsb: 0, width: 1 }, type: rw, initial_value: 0 }
        - labels:
          - { name: foo, value: 0, comment: 'FOO value' }
          - { name: bar, value: 1, comment: 'BAR value' }
```

Listing 20: Example RgGen registers description in YAML format.

```c
#ifndef BLOCK_0_H
#define BLOCK_0_H
#include "stdint.h"
#define BLOCK_0_REGISTER_0_BIT_FIELD_0_BIT_WIDTH 4
#define BLOCK_0_REGISTER_0_BIT_FIELD_0_BIT_MASK 0xf
#define BLOCK_0_REGISTER_0_BIT_FIELD_0_BIT_OFFSET 0
#define BLOCK_0_REGISTER_0_BIT_FIELD_1_BIT_WIDTH 2
#define BLOCK_0_REGISTER_0_BIT_FIELD_1_BIT_MASK 0x3
#define BLOCK_0_REGISTER_0_BIT_FIELD_1_BIT_OFFSET 4
#define BLOCK_0_REGISTER_0_BIT_FIELD_2_BIT_WIDTH 2
#define BLOCK_0_REGISTER_0_BIT_FIELD_2_BIT_MASK 0x3
#define BLOCK_0_REGISTER_0_BIT_FIELD_2_BIT_OFFSET 6
#define BLOCK_0_REGISTER_0_BYTE_WIDTH 4
#define BLOCK_0_REGISTER_0_BYTE_SIZE 4
#define BLOCK_0_REGISTER_0_BYTE_OFFSET 0x0
#define BLOCK_0_REGISTER_1_BIT_WIDTH 1
#define BLOCK_0_REGISTER_1_BIT_MASK 0x1
#define BLOCK_0_REGISTER_1_BIT_OFFSET 0
#define BLOCK_0_REGISTER_1_BYTE_WIDTH 4
#define BLOCK_0_REGISTER_1_BYTE_SIZE 4
#define BLOCK_0_REGISTER_1_BYTE_OFFSET 0x4
#endif
```

Listing 21: Example C header file generated using RgGen.

```yaml
metadata:
  name: stream_monitor
  brief: monitors a number of streams.
features:
  bus-width: 32
  optimize: yes
entity:
  clock-name: axil_aclk
  reset-name: axil_aresetn
  reset-active: low
  bus-prefix: axil_
  bus-flatten: yes
interface:
  flatten: yes
fields:
  - repeat: 4 # <-- number of streams!
    stride: 5
    field-repeat: 1
    subfields:
      - address: 0
        name: ecnt
        doc: |
          Accumulates the number of elements transferred on the stream. Writing to
          the register subtracts the written value.
        behavior: custom
        interfaces:
         - input: valid
         - input: ready
         - input: count:8 # <-- width of count field!
         - input: dvalid
         - input: last
         - drive: ivalid
         - drive: iready
         - drive: itransfer
         - drive: ipacket
         - state: accum:32
      - address: 4
        name: vcnt
        behavior: internal-counter
        internal: ivalid
      - address: 8
        name: rcnt
        internal: iready
```

Listing 22: Example vhdMMIO registers description in YAML format.

```
peripheral {
  name = "GPIO Port";
  description = "A sample 32-bit general-purpose bidirectional I/O port.";
  hdl_entity = "wb_slave_gpio_port";
  prefix = "gpio";
  reg {
    name = "Pin direction register";
    description = "A register defining the direction of the GPIO potr pins.";
    prefix = "ddr";
    field {
      name = "Pin directions";
      description = "1 - OUTPUT, 0 - INPUT";
      type = SLV;
      size = 32;
      access_bus = READ_WRITE;
      access_dev = READ_ONLY;
    };
  };
  reg {
    name = "Pin input state register";
    description = "A register containing the current state of input pins.";
    prefix = "psr";
    field {
      name = "Pin input state";
      description = "Each bit reflects the state of corresponding GPIO port pin.";
      type = SLV;
      size = 32;
      access_bus = READ_ONLY;
      access_dev = WRITE_ONLY;
    };
  };
  reg {
    name = "Port output register";
    description = "Register containing the output pin state.";
    prefix = "pdr";
    field {
      name = "Port output value";
      description = "Writing '1' sets the corresponding GPIO pin to '1'";
      size = 32;
    };
  };
};
```

Listing 23: Example wbgen2 registers description in wbgen2 specific format.

4 Dissertation

4.1 Thesis

It is possible to infer the bus and register structure based on the description of the functionality of the data that shall be stored in the registers. Moreover, such an approach offers some significant advantages in most typical use cases compared to the classic approach in which register structure is described explicitly.

4.2 Aim and scope

The main aim of the dissertation is to design a language allowing to describe system bus and registers by defining functionality of the data. The work also includes the implementation of the proof of the concept compiler as well as the discussion of some general implementation details that any FBDL-compliant compiler will likely have to face.

5 Functionalities

It is recommended to read the subsections of this chapter concurrently with the corresponding subsections of the FBDL specification (first specification then dissertation) or to read the whole specification first. The specification is more focused on answering the „how" questions, whereas this dissertation is more focused on answering the „why" questions and describing the benefits of the newly proposed functionality-centric approach.

5.1 Block

The block functionality is mainly used to logically group or encapsulate functionalities. The block concept is not unique for FBDL approach as some of the register-centric approaches already had the same concept (for example, AGWB or SystemRDL). However, thanks to the type parametrization and type extending mechanism, it is easy to instantiate blocks with slightly different functionality. This is a common scenario in the case of the FPGAs with two SLRs [78]. The SLRs might have different numbers of available resources and might be connected to different hardware IP blocks. Let's suppose there are two SLRs, SLR0 and SLR1. SLR0 is connected to the PCIe, and there is a high throughput PCIe-AXI bridge in the SLR0. In case of any problems with the bridge, there might be a need to debug it. A side access channel is required for SLR0, hence it must have two master ports. What is more, it must have some extra configuration and status data compared to the SLR1. Listing 24 presents how such requirements can be easily satisfied in FBDL using type parametrization and type extending mechanisms.

5.2 Bus

The bus functionality represents the bus structure. The bus named `Main` is the default entry point for the description used for the code generation. A compiler is free to accept an argument allowing to change the root of the description from `Main` to any valid identifier. However, care is advised when choosing a naming convention for functionalities. Usually, a language has its preferred naming conventions. Some languages have multiple conventions (C/C++/VHDL). Some languages have only a single convention (Go/Python), but they are not formal, so there might be multiple in practice. As FBDL description might be

```
type SLR(masters_count=1) block
    masters = masters_count
    const PERIPHERAL_COUNT 1024
    C [PERIPHERAL_COUNT]config
    S [PERIPHERAL_COUNT]status; width = 14
    P proc
        p1 param;  width = 16
        p2 param;  width = 8
        r  return; width = 25
Main bus
    # SLR0 has 2 masters and is extended with some extra
    # config and status for high throughput PCIe-AXI bridge
    # configuration and debugging via low throughput
    # UART-AXI bridge.
    SLR0 SLR(2)
        PCIe_AXI_config config; width = 16
        PCIe_AXI_status status; width = 48; atomic = false
    SLR1 SLR
```

Listing 24: Example of type parametrization and type extending based on the block functionality.

(actually almost always is) compiled into multiple target languages, it is impossible to suit all naming conventions for given targets. Instead, it should be guaranteed that the given functionality name from the given `.fbd` file has the same name in all different target source files. It implies that the two most popular naming conventions (`camelCase`, `snake_case`) should be avoided for functionality instance names and for constants that should be accessible in target languages. Both `camelCase` and `snake_case` start with a lowercase letter. It imposes restrictions on how the target code might be implemented. For example, in Go, data types, fields, or functions starting with lowercase letters are not exported. A potential implementation would have to do one of the following:

1. Change the instance names so that the first letters are uppercase. The drawback is that the same instance would have at least two different names across all targets.
2. Generate extra functions allowing access to functionalities. For example, a function translating string into proper field value. This would imply extra performance overhead and more complex code.

The remaining naming conventions starting with uppercase letter are `PascalCase` and `Pascal_Snake_Case`. However, as some languages (VHDL, for example) are case insensitive and there is no way to enforce `PascalCase`, the `Pascal_Snake_Case`, and `SCREAMING_SNAKE_CASE` are strongly recommended. Broken `pascalcase` is really hard to read, especially after several hours of sitting in front of the computer screen.

5.3 Config

The config functionality is almost like a control register from the typical register-centric approach. Almost, because the config functionality abstracts away the limited width of the register.

Listing 25 shows an example description with a single config with a width equal to the register width in the RgGen. As RgGen does not support registers without bit fields, there is a need to type the `C` name twice. Most register-centric tools support registers without bit fields. Listing 26 shows an example description with a single config with a width equal to the register width in the FBDL. Listings 27 and 28 present example code writing the config. In case of config width not greater than the register width, the code is the same for the register-centric approach and for FBDL.

```
- register_block:
  - name: Main
  - registers:
    - name: C
      bit_fields:
      - { name: C, bit_assignment: { width: 32 }, type: rw }
```

Listing 25: Example config instantiation with width equal to the register width in the RgGen.

```
Main bus
    C config
```

Listing 26: Example config instantiation with width equal to the register width in the FBDL.

```
def do_something():
    value = prepare_value()
    Main.C.write(value)
```

Listing 27: Example config write utilizing the code generated by the register-centric approach compiler.

Listing 29 shows an example description with a single config with a width greater than the register width in the RgGen. Listing 30 shows an example description with a single config with a width greater than the register width in the FBDL. In this case, there is no need to adjust the code writing the config for FBDL. As any FBDL compiler is obliged to generate functionality write and read access code, the code from listing 28 is still valid. However, the register-centric approach code needs adjustments as an extra register has been added. Listing 31 presents adjusted code. It takes extra time to write the code, and there is a

```
def do_something():
    value = prepare_value()
    Main.C.write(value)
```

Listing 28: Example config write utilizing the code generated by the FBDL compiler.

room for possible mistakes. Firstly, the masks and shifts need to be applied to the value manually. Even if the masks and shifts are generated as constants/variables, there is still a possibility of typing an incorrect name. Secondly, if the config needs atomic access, then the registers must be read/written in the correct order. Thirdly, the atomicity must be manually coded at the HDL side. None of these is an issue in the FBDL, as everything is handled automatically by the compiler. This is the result of looking at the config as a functionality, not as a control register (the user cares about it as a whole, not as fragmented pieces).

```
- register_block:
  - name: Main
  - registers:
    - name: C1
      bit_fields:
      - { name: C1, bit_assignment: { width: 32 }, type: rw }
    - name: C2
      bit_fields:
      - { name: C2, bit_assignment: { width: 1 }, type: rw }
```

Listing 29: Example config instantiation with width greater than the register width in the RgGen.

```
Main bus
    C config; width = 33
```

Listing 30: Example config instantiation with width greater than the register width in the FBDL.

5.4 Irq

The irq functionality represents an interrupt handling. Whether interrupts should be considered as a part of a bus is a debatable topic. It has been decided that FBDL shall provide support for interrupts because of the following reasons:

1. Interrupts, in most cases, have associated registers informing about the interrupt source.

```
def write_C(value):
    Main.C1.write(value & 0xFFFFFFFF)
    Main.C2.write((value >> 32) & 0x1)
def do_something():
    value = prepare_value()
    write_C(value)
```

Listing 31: Example config write utilizing the code generated by the register-centric approach compiler - config wider than register (33 bits).

2. Interrupts, in most cases, have associated enable/mask registers allowing switching on or off particular interrupts.
3. Interrupt lines are frequently routed together with bus lines.

Although FBDL supports interrupts, the support is limited solely to interrupt handling. For example, there is no support for interrupts hierarchy (this feature is present, for instance, in SystemRDL). This is because the interrupts hierarchy is not related to the bus in any way, and it can be easily created at the provider side by properly connecting interrupt components. There is also no way to configure whether an interrupt is triggered by the high or low level or a rising or falling edge. As FBDL assumes positive logic, the high level is assumed for level-triggered interrupts, and the rising edge is assumed for edge-triggered interrupts. Low-level interrupts or falling edge interrupts can be easily handled by negating the signal at the provider side. Adding the distinction into the FBDL would unnecessarily complex the language and would create a second way for solving the same problem.

5.5 Mask

The mask functionality is very similar to the config functionality. From the provider's perspective, there is no difference between the mask and the config. However, there is a difference in the interface generated for the requester. The mask is bit-oriented, whereas the config is value oriented.

The mask has all the same advantages over the register-centric approach as the config has. There is also no need to add the `mask` prefix or suffix to the name to indicate to the user that particular data is a mask, as the type already indicates it. Additionally, it also has automatically generated means for bitwise operations. The interface must include ways for:

1. Setting (writing `1`) particular bits while simultaneously clearing remaining bits.

2. Clearing (writing `0`) particular bits while simultaneously setting remaining bits.
3. Setting (writing `1`) particular bits without changing the state of remaining bits.
4. Clearing (writing `0`) particular bits without changing the state of remaining bits.
5. Toggling particular bits without changing the state of remaining bits.

Appendix B presents a code that can be automatically bound to the data solely based on the distinct type for mask.

5.6 Memory

The memory functionality is used to directly connect and map an external memory to the generated bus address space. The memory does not have any valid inner functionalities. In SystemRDL, for example, within a memory it is possible to have virtual child instances representing a software view of the memory data. The FBDL takes a different approach in this case. As memory can be seen as a continuous area of storage elements, one can describe the layout of the data within the memory using a separate FBDL description file or even using one of the register-centric tools if it makes more sense in a particular case. An access interface used to access the data in the memory can then use the memory access methods generated for the primary FBDL description (the one having the memory functionality). The idea is presented in figure 5.1. Such an approach keeps the language smaller, more concise, and orthogonal.

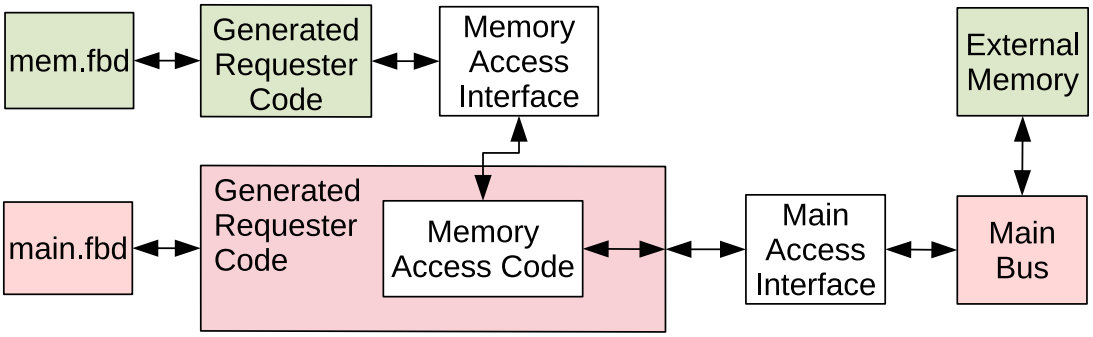


Figure 5.1: A possible access path to the external memory with separate FBDL description.

Memory can also be connected to the bus using the proc or stream functionality (technically, it is also possible using solely configs, but this method is verbose, vague, and impractical, so it has been omitted). Each of the five approaches (memory, two proc approaches, two stream approaches) has its advantages and disadvantages. Global advantage (+), global disadvantage (-), proc relative characteristic (•), stream relative

characteristic (*).

Memory:

+ The best potential throughput equal to the bus throughput.
+ No need for wrapper logic.
- To achieve the maximum throughput for block transactions, both the bus and the access interface must support true block transactions.
- Generated address space size increased by the memory address space size.

One proc:

+ Minimal generated address space size increase.
- The worst throughput limited by the requester-provider round-trip latency for each item access.
• The write access is additionality limited by the mandatory read of return data.

Two procs:

+ Minimal generated address space size increase.
• Needs more bits than one proc approach, as the memory address is repeated in the second proc.
- The worst throughput limited by the requester-provider round-trip latency for each item access.
• The write access is not additionality limited by the mandatory read of return data, as it is in the case of one proc approach.

Stream - common memory address in separate config:

+ Minimal generated address space size increase.
+ The throughput for block read and write can potentially equal the bus throughput.
- Suboptimal single read and write accesses because of additional memory address write to separate config.
- Needs more complex implementation as both the bus and the access interface must support true cyclic transactions to achieve maximum throughput.
- Needs wrapper logic if memory throughput is lower than the bus throughput for cyclic transactions.

Stream - downstream with its own memory address param.

+ Minimal generated address space size increase.

* Needs more bits than one stream with a common memory address, as the memory address must be placed for upstream in the config anyway.

+ The throughput for block read and write can potentially equal the bus throughput.

+ The throughput for random writes can potentially equal the bus throughput, as the memory address is the downstream param.

- Suboptimal single read access because of additional memory address write to separate config.

- Needs potentially the most complex implementation as both the bus and the access interface must support true cyclic block transactions to achieve maximum throughput.

- Needs wrapper logic if memory throughput is lower than the bus throughput for cyclic transactions.

Particular advantages or disadvantages of given approaches may not be valid if access to the memory is of read-only or write-only type. To make a satisfactory choice for a particular design, a user must take into account at least the following factors: required throughput, maximum overall address space size, type of memory access (read-write, read-only, write-only), type of memory transactions (will there be more single or block transactions), design simplicity. Listings 32, 33, 34, 35, 36 present example descriptions of five discussed external memory connections. The memory has a read-write access type, its size equals 65536 words, and the word with equals 16 bits. Depending on the requirements, it is also possible to mix some of the approaches. For example, if memory is written in blocks and writes require high throughput, but it is read in single transactions, then it is possible to use the stream for writes and proc for reads.

```
Main bus
    Mem memory
        size = 2 ** 16
        width = 16
```

Listing 32: FBDL external memory connection using memory functionality.

```
Main bus
    Access_Mem proc
        addr param;        width = 16
        data_in param;     width = 16
        read_write param; width = 1 # 0 - read, 1 - write
        # The delay depends on the clock frequency
        # and read latency.
        delay = 1 us
        data_out return; width = 16
```

Listing 33: FBDL external memory connection using one proc functionality.

```
Main bus
    Read_Mem proc
        addr param; width = 16
        delay = 1 us
        data return; width = 16
    Write_Mem proc
        addr param; width = 16
        data param; width = 16
```

Listing 34: FBDL external memory connection using two proc functionalities.

5.7 Param

The `param` functionality is an inner functionality of the `proc` and `stream` functionalities. The `param` functionality does not have the `default` property. This implies that `proc` or `stream` parameters cannot have default values, which further implies that functions or methods generated for the requester also do not have default values for parameters. It has been designed this way because not all programming languages support default values for function parameters (for example, C, Go, Rust). This could be worked around as the code for the requester is automatically generated anyway. However, in the end, it has been decided that adding support for the default value for the param functionality is not worth because of the following reasons:

1. It would add extra complexity to the FBDL compilers.
2. Programming languages without the support for default values for function parameters are doing well. There are even negative opinions on default values for function parameters. The argument behind these opinions is that they make code less readable and harder to analyze.
3. User can always implement wrapper functions in the target language.

```
Main bus
    addr config; width = 16
    Read_Mem stream
        data return; width = 16
    Write_Mem stream
        data param; width = 16
```

Listing 35: FBDL external memory connection using two stream functionalities with common address config.

```
Main bus
    addr config; width = 16
    Read_Mem stream
        data return; width = 16
    Write_Mem stream
        addr param; width = 16
        data param; width = 16
```

Listing 36: FBDL external memory connection using two stream functionalities with separate address in downstream.

5.8 Proc

The proc functionality is a concept not present in the register-centric approaches. It represents a procedure called by the requester and carried out by the provider. The proc functionality is a good representative presenting how the functional view on the data can significantly reduce the amount of manual work and increase the code robustness [79]. It is called proc (from procedure), and not, for example, func (from function), to highlight that this action has side effects and might take a non-negligible amount of time. In other words, it is not pure.

Listing 8 presents an example taken directly from the data acquisition design for the CBM experiment. Listing 9 presents Python code that had to be coded manually. Section 3.2 describes what is not optimal in this case in the register-centric approach. Listing 37 presents a description of the same block in FBDL format. Based on the description, it is already clear the inferred registers will be used for procedure call.

5.9 Return

The `return` functionality is an inner functionality of the `proc` and `stream` functionalities. It represents data returned by a procedure or streamed by an upstream. Technically, it was possible to add direction property to the `param` functionality, similar to the procedures

```
type HCTSP_Software_Command_Slot block
    Send proc
        chip_addr        param; width = 4
        downlink_mask    param; width = 12
        group_mask       param; width = 8
        sequence_number  param; width = 4

        request_type     [2]param; width = 2
        request_payload  [2]param; width = 15
        crc              [2]param; width = 15
```

Listing 37: HCTSP software command slot block description in FBDL format.

in the Ada language. However, `param` and `return` do not have the same properties. Making them distinct also makes the language design less fragile in case of potential future enhancements as it helps to avoid inter-property dependencies.

5.10 Static

The static functionality represents data placed at the provider side that shall never change. The register-centric approach usually achieves this using a status register driven by a fixed value. However, if it is impossible to mark the register as read-only for both sides, then it is not clear that the data inside the register never changes without any extra comment or code analysis. In FBDL, such constant data has its own type.

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. It is worth analyzing what is the typical difference between an id and a version. An id is usually data automatically added by a compiler, calculated using some hash function with input description being the hash function input. An id primary function is to be a description signature, upon which it is clear whether two or more descriptions are identical. A version is usually data manually added by an engineer to indicate what functionalities are supported by a given bus or block.

The FBDL specification does not require FBDL compilers to add any bus or block ids automatically. However, at least bus id is extremely useful in practice. It can be used to ensure that both requester and provider utilize the results of the compilation of the same bus description. Register with such id must be placed at a fixed, known address, usually at the beginning or at the end of the generated address space.

5.11 Status

The status functionality is almost like a status register from the typical register-centric approach. Almost, because the status functionality abstracts away the limited width of the register. All advantages of the config functionality (section 5.3) are also valid for the status functionality. The only difference between the config and the status functionalities is that in the case of the config, it is the requester that is the only writer, whereas in the case of the status, it is the provider that is the only writer.

5.12 Stream

The `stream` functionality represents a stream of data to a provider (downstream) or a stream of data from a provider (upstream).

Unlike `proc`, the `stream` functionality has only one associated signal at the provider side, the strobe signal. The `proc` has distinct call and exit signals. However, as the `stream` shall have only parameters (downstream) or only returns (upstream), having one associated signal is enough.

6 Absent features

The FBDL does not provide some of the popular capabilities present in some of the register-centric approach tools. This chapter lists the most common ones, and explains why they are absent. However, their absence does not mean that they will never be added. At the current stage their disadvantages are clear, but the potential advantages they might bring are vague.

6.1 Double side writable data

Double side writable data is a data that can be written by both the requester and the provider (FBDL specification nomenclature). In practice it means data can be written by both, the firmware/software side and the gateware/hardware side. This is possible in some of the register-centric tools. For example, SystemRDL refers to this aspect as to the software and hardware access properties. In the FBDL there is no functionality that would end up as a data that can be written by both the requester and the provider sides. There is always one side writing the data and zero, one or two sides reading the data (zero is possible, although it means that the functionality is unused). This can lead to increased address space size and resource utilization, however it cuts off all problems related with designing and debugging systems with multiple data writers. As the resulting increase of the resource utilization is relatively small (the number of required flip-flops is the same, only extra logic related with increased address space size is needed), and devices provide more and more resources every year, it has been decided that this tradeoff is worth to take. Allowing flip-flops to be written by two sources also increases the resource utilization, however it does not increase the address space size.

Single side write restriction does not mean that multiple requesters can not write the same `config` for example. This means that if the requester side can write some data, then the provider side must not write this data. The number of requesters allowed to write is unlimited.

Single side write restriction also does not mean that different data, writable by different sides, can not be placed in the same physical register (the same register address). Listings 38 and 39 show examples.

Config `C` and status `S` occupy exactly half of the register width. As the requester side is

the writer of config `C` and the reader of status `S`, and the provider side is the reader of config `C` and the writer of status `S`, both functionalities can be put into the same register without any overhead. The required address space size equals 1.

Procedure `P` has no data, so it needs only address for call triggerring. Status `ST` occupies whole register. As the requester side is the writer of procedure `P` and the reader of status `ST`, and the provider side is the reader of procedure `P` (it reads the call signal) and the writer of status `ST`, both functionalities can be put into the same register without any overhead. The required address space size equals 1.

```
Main bus
    C config; width = 16
    S status; width = 16
```

Listing 38: Example of `config` and `status` that can share register address.

```
Main bus
    P proc
    ST status
```

Listing 39: Example of `proc` and `status` that can share register address.

6.2 Enumeration type

The first issue with the enumeration type is that the FBDL description is not directly compiled into the machine code or synthesized into the digital logic. The FBDL description is transpiled. In other words, it is compiled to other programming or hardware description languages. However, those other languages do not share a common definition of the enumeration type. Let's analyze three, currently very popular, system programming languages:

1. C - enum type is a list of constant values.
2. Go - no support for any kind of enum type at all.
3. Rust - enum type is actually a union type or a sum type.

One of the goals of the FBDL is to easily add compiler back-ends for target languages. Extending FBDL with features peculiar for any single target language, or a subset of target languages, is against this rule. Usually, when speaking about enumeration type in the context of registers managing, a set of constant possible values is meant. This is already achievable in FBDL using constant definitions, listing 40. As constants are

bound to a scope, the values in generated files can also have limited scope. It is also already possible to limit the set of valid values for some functionalities using the `range` property.

```
# Global constants.
const E = 2.72
const PI = 3.14
const LN2 = 0.69
Main bus
    # Shorter form using multi constant definition.
    # Below constants are scoped only to the Main bus.
    const
        ZERO = 0
        ONE = 1
        TWO = 2
    # Range of possible values is limited for below config
    # from ZERO to TWO.
    c config; range = [ZERO, TWO]
```

Listing 40: Constraining value range using constants or `range` property.

The second issue with the enumeration type is the enumeration type values synchronization. This is more general issue, not related only with the register management tools. Let's suppose enumeration type is a list of constant values (the simplest enum definition). Figure 6.1 presents an example system design with three actors: firmware, gateware and software. The enumeration type definitions between actors must be consistent (the same values for corresponding options).

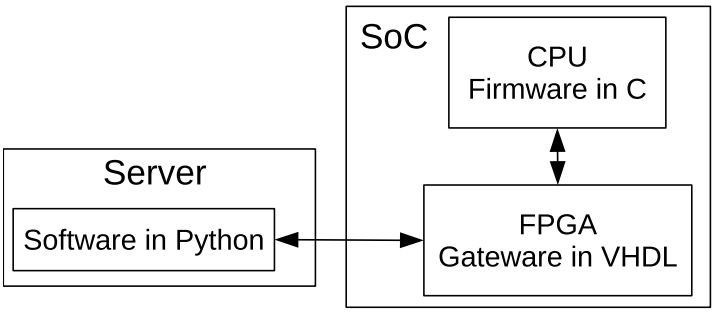


Figure 6.1: Example system with enumeration types synchronization issue.

There are at least three ways to approach the problem.

1. The FBDL is the source of the enumeration type definitions. The drawback of this approach is introducing internal depedency on the FBDL output inside firmware, gateware and software modules. The modules start to not only use the FBDL output to access or provide the functionalities, but also internally to implement its

own logic or data structures. For example, gateware module unit testbench requires type generation and no longer can be run in isolation.

2. There is a single source of enumeration type definitions, but it is not FBDL. This approach has three possible implementations, but all them require an extra tool for updating derived definitions.

   (a) Enumeration type definitions are derived from the software/firmware source code. The drawback is that often different languages are used for prototyping and for final implementation

   (b) Enumeration type definitions are derived from the gateware/hardware description. The hardware description language, once chosen, almost never changes during the project.

   (c) Enumeration type definitions are derived from the dedicated tool with its own syntax for definitions.

3. Enumeration type definitions are implemented manually for all languages. However, there is a tool (some kind of a sanitizer) capable of checking that all enumeration type definitions are coherent. As not all sources are always available in the repository the tool would have to support fetching sources via version control systems or accessing them via URL.

During the work on numerous projects the author has come across most of the mentioned approaches. As it is not clear that the approach with the register management tool being the source of the enumeration type definitions has advantages over other approaches it has been decided that adding support for enumeration type within the language at the current stage is not sufficiently justified.

6.3 Custom expression functions

The FBDL does not allow defining custom functions for expressions evaluation. This is possible with all tools providing programming language API for description definition, as in this case all features of the programming language are „inherited" and can be used without any limitations. This is very flexible mechanism, but it sometimes leads to abuses and bus/register management tool starts to be used as a general purpose design configuration tool storing information not related with the bus or registers. The FBDL's goal is to be a bus and register management tool, nothing less nothing more. However, the FBDL does contains built-in functions (listed in the specification) frequently used for bus or registers related calculations.

6.4 Manual addressing

Some of the register-centric tools allow manual register addressing. Manual addressing is setting register address explicitly. Placing some data at fixed address might be useful in case of bus identification or blocks versioning.

The FBDL does not allow manual addressing because of two reasons. The first one is that in FBDL user does not define registers, but data with its functionality type. This of course does not imply any implementation blockers for manual addressing support, as the address in such case could be the start address of the data. However, manual addressing simply does not fit into the FBDL paradigm. The second reason is that any decent compiler should automatically insert bus identification number at some fixed address. Placing single data, with unique value, at fixed address is enough to unambiguously identify address map. Based on this information the firmware or software can load appropriate address map code and access any data, for example blocks versions, even if its address differs between versions. In such a case supporting manual addressing does not solve any problems, but increases the complexity of registerification algorithms.

6.5 Custom attributes

SystemRDL allows for defining custom properties. Such mechanism can be useful for tuning the compiler behavior. On the other hand, it opens a space for inconsistency between compilers as they are free to ignore unknown custom properties. The FBDL does not support custom properties at the current stage, but it reserves a syntax and terminology. The term „attributes" will be used for custom properties if they are ever supported. Custom attributes will be assigned the same way the properties are assigned, but the attribute names will be prepended with the '@' (at sign) character, listing 41 presents an example.

```
Main bus
    @addressing-mode = "Compact"
```

Listing 41: Syntax reserved for custom attributes.

It is worth mentioning that compiler behavior can be tunable even without custom attributes using additional compiler parameters. The FBDL specification is also open to add more properties if their existence is justified enough.

7 Implementation

This chapter describes the implementation of the proof of the concept compiler for the FBDL. As the comprehensive description would be relatively long and would include aspects irrelevant from the thesis point of view, the chapter describes only the overall structure and focuses on some general details that probably any FBDL-compliant compiler will have to face.

The compiler has been divided into two parts, the front-end [80] and the back-end [81], both are publicly available. The front-end is responsible for reading FBDL description files, parsing them, instantiating functionalities, and carrying out the registerification process, all according to the FBDL specification. The back-end is responsible for taking the registerification result and generating the desired target code. The decision to divide the compiler into the front-end and back-end has been driven by two factors.

1. Regardless of the target, the parsing, instantiating, and registerification phases must be carried out for any compiler. However, what is later done with the registerification results for a particular target highly depends on the target itself. A Python interface with dynamic loading of address maps and asynchronous access has a completely different code structure than for example, C module with a statically compiled address map and synchronous access. The border between what is common and what depends on the target is quite straightforward, and splitting the compiler into the front-end and back-end feels quite natural.

2. If the compiler was monolithic and it was released with any restrictive license, for example, GPL-3.0, then it would not be possible to incorporate it into proprietary, closed-source programs directly. If the compiler was monolithic and it was released with any permissive license, for example, MIT, then anyone would be able to take it as is and fix bugs or implement improvements without even reporting it. The modular structure of the compiler is a compromise. Any changes applied by a third party to the front-end must be reported. However, it is still possible to write a closed-source back-end. In such a case, the closed-source back-end must call front-end as an external program and dump registerification result to a JSON file. The JSON file can then be read by the back-end.

Figure 7.1 presents the current (at the time of dissertation writing) structure of the implemented compiler. Input `.fbd` files are parsed in parallel by the parser module.

Both instantiation and registerification modules run internally in a sequential manner. Generators for different targets are run in parallel if a user asks for multiple outputs in a single call. What is more, if code for a given target is placed in multiple files, then the files are also generated in parallel.

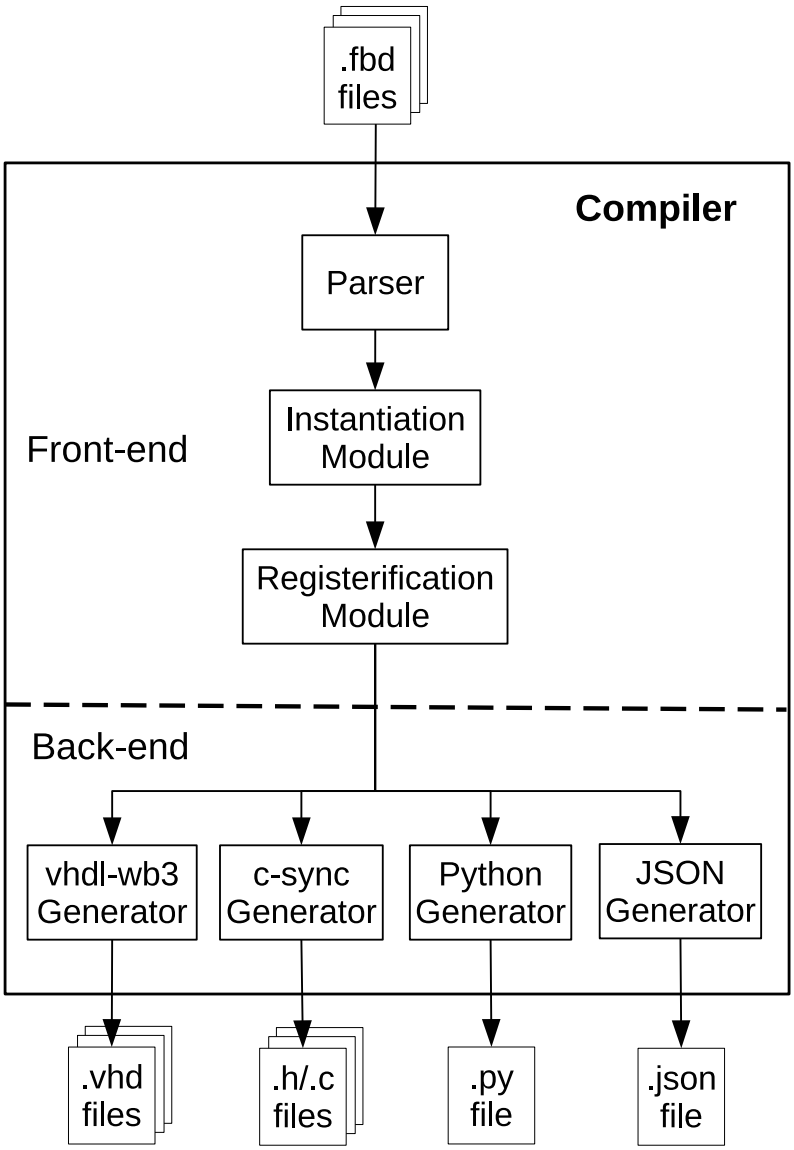


Figure 7.1: Current structure of the implemented compiler.

7.1 Front-end

The front-end of the compiler is responsible for processing everything defined in the FBDL specification except the access means, as they highly depend on the target. It is internally built of three stages: parsing, instantiation, and registerification.

7.1.1 Parsing

The parsing stage is responsible for building abstract syntax trees for description within files. As there are no inter-file dependencies during the parsing stage, it is easy to run this stage in parallel (true parallelism). The parser has been generated using the tree-sitter tool [82]. The defined grammar is available online [83]. Tree-sitter is a parser generator tool based on the GLR [84] parsing algorithm working most efficiently with a class of context-free grammars. It allows for very rapid prototyping, but it is not free of drawbacks. The main one is error handling. If the syntax provided by the user is not valid, then giving informative feedback to a user on what exactly is wrong requires relatively more work than in the case of a hand-written custom parser, or sometimes is even impossible.

7.1.2 Instantiation

The instantiation stage is responsible for instantiating functionalities starting from the `Main` bus description. As the type parametrization is resolved at this stage, it is not truly parallel. There are two possible approaches. The first one is to run this stage sequentially. The sequential approach is simpler to implement. The second one is to run the instantiation stage in parallel. The parallel approach is harder to implement. What is more, it requires more data copying internally, as each instantiation worker might have different values of type arguments in different scopes. The proof of the concept compiler implements the instantiation stage in the sequential manner. The whole compilation process is relatively short. A bus with up to 40 functionalities takes less than 10 ms to compile (both front-end + back-end) on a platform with Intel i7-8750H CPU. The gain from the parallel instantiation would not be worth the extra complexity added to the code.

7.1.3 Registerification

The registerification stage is responsible for putting functionalities into the actual registers. This stage includes assigning data bit masks, register addresses, block addresses, and masks, as well as access types. The registerification stage is relatively hard to be implemented in parallel, as it requires determinism. The registerification algorithms must be deterministic because, in the case of non-determinism, registerification of the same bus may lead to different register layout and performance. Although such behavior is not forbidden by the specification, it is highly impractical. What is more, the registerification stage has sequential nature. To optimize generated address space size, it is required to

put functionalities (if possible) into the gaps created during the registerification of other functionalities. This implies data dependency in the registerification algorithm.

Access types

During the registerification stage, it must be determined how the data of the functionality must be accessed. The access types are not defined in the specification, so each compiler is free to adopt its policy. For example, a compiler highly optimized for AXI byte addressing will probably implement different access types than some generic multi-target compiler supporting both byte and word addressing.

The implemented compiler has five access types:

1. Single Single
2. Single Continuous
3. Array Single
4. Array Continuous
5. Array Multiple

The Single Single access type is the simplest access type used for single data fitting a single register. Listing 42 presents a description with three data of Single Single access type, and figure 7.2 presents an example register layout. The Single Single access type requires address and mask (start bit and end bit) attributes to unambiguously describe how to access the data.

```
Main bus
    C  config; width = 12
    S0 status; width = 20
    S1 status
```

Listing 42: Example bus with three data of Single Single access type.

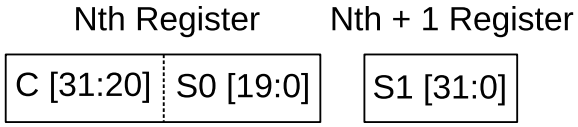


Figure 7.2: Example register layout of data of Single Single access type.

The Single Continuous access type is used to describe the access to data spanning multiple adjacent registers. Listing 43 presents a description with two data of Single Continuous

access type, and figure 7.3 presents an example register layout. The Single Continuous access type requires start address, start bit, and width attributes to describe how to access the data unambiguously.

```
Main bus
    S0 status; width = 87
    S1 status; width = 41
```

Listing 43: Example bus with two data of Single Continuous access type.

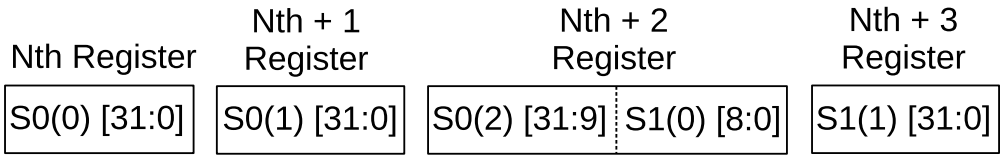


Figure 7.3: Example register layout of data of Single Continuous access type.

The Array Single access type is used to describe access to array data with a single element placed within a single register. Listing 44 presents a description with one array data of Array Single access type, and figure 7.4 presents an example register layout. The Array Single access type requires start address, mask (start bit and end bit), and elements count to unambiguously describe how to access the data.

```
Main bus
    S [3]status; width = 24
```

Listing 44: Example bus with one data of Array Single access type.

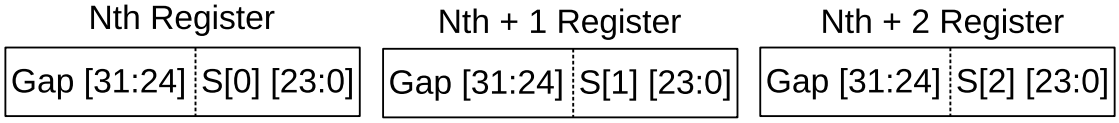


Figure 7.4: Example register layout of data of Array Single access type.

The Array Continuous access type is used to describe access to array data with elements placed adjacent to each other even if the gap in the register is narrower than the single element width. Listing 45 presents a description with two array data of Array Continuous access type, and figure 7.5 presents an example register layout. S0 is two-element array data with single element width greater than the register width. S1 is four-element array data with single element width smaller than the register width. The Array Continuous

```
Main bus
    S0 [2]status; width = 40
    S1 [4]status; width = 12
```

Listing 45: Example bus with two data of Array Continuous access type.

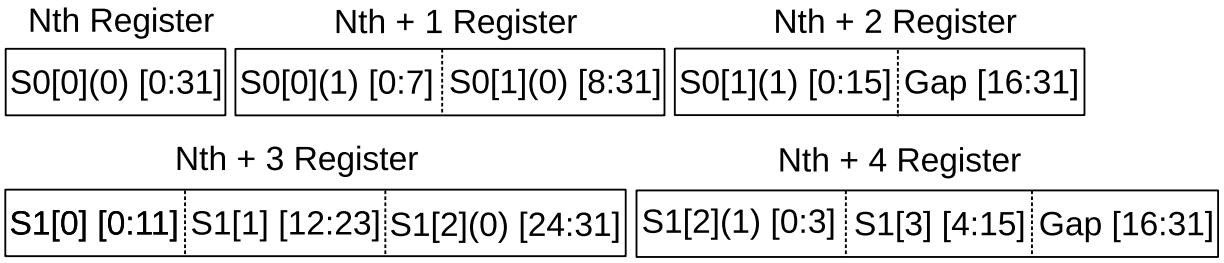


Figure 7.5: Example register layout of data of Array Continuous access type.

access type requires start address, start bit, and elements count to describe how to access the data unambiguously.

The Array Multiple access type is used to describe access to array data with multiple elements placed in one register. Listing 46 presents a description with two array data of Array Multiple access type, and figure 7.6 presents an example register layout. S0 is six-element array data with single element width being the divisor of the reigster width. S1 is five-element array data with single element width not being the divisor of the register width. The Array Multiple access type requires start address, start bit, element width, and elements count to describe how to access the data unambiguously.

```
Main bus
    S0 [6]status; width = 16
    S1 [5]status; width = 9
```

Listing 46: Example bus with two data of Array Multiple access type.

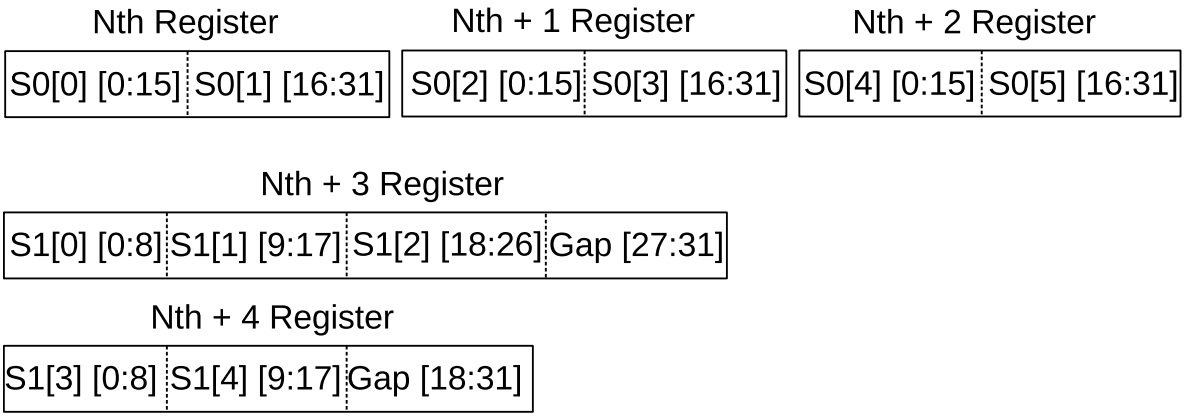


Figure 7.6: Example register layout of data of Array Multiple access type.

Registerification algorithm

The only registerification algorithm requirement imposed by the specification is determinism. A compiler must produce the same registerification result when run multiple times with exactly the same input and arguments. Everything else related to the registerification algorithm is up to a compiler. A single compiler may provide multiple registerification algorithms configurable, for example, via command line parameter.

Although a compiler has freedom in terms of the registerification algorithm, there are some general recommendations that, when followed, ease the implementation. The below recommendations work when functionalities are registerified one by one. That is, once picked, the functionality is ultimately registerified with its final hardware address. They might not be valid in the case of more sophisticated algorithms, for example, when procs, streams, and groups are first registerified internally and later organized in a sequence optimizing generated address space sizes. Some of the recommendations with greater indexes assume that some recommendations with lower indexes are met.

1. If the minimum number of registers for storing single functionality equals N ($N = ceil(functionality\ width/bus\ width)$), then this functionality should be placed into N registers. Theoretically, putting it into M ($M > N$) registers can save some address space if enough gaps exist. However, as the compiler knows nothing about the access interface during the compilation, an artificial increase of the number of registers needed for functionality can greatly increase the access time if the access interface does not support block transactions or if gaps are not placed in consecutive registers. A small address space size decrease is usually not worth an access time increase in such cases, as the round trip latency in some cases might be significant.
2. Proc and stream are encapsulated functionalities. Params and returns can always

be aligned to each other if params are not readable. The gaps are possible only at the edges. The call register (or downstream strobe register) must not have any external writable functionality such as config or mask as the write generates the call strobe. If proc params are readable, the exit register must not have any proc params. Moreover, the exit register must not have any functionality not belonging to the proc. This is because the read generates an exit pulse, and all functionalities in such a case are readable. As the specification does not impose whether proc params are readable, it depends on the compiler implementation. If the compiler does not allow param read, then it is safe to put proc params in the exit register. In such a case, the params might belong to the same proc or to another one, but all of them must belong to the same proc. To sum up, a gap after proc or stream registerification is created only when:

   (a) Proc has only params, or stream is downstream, and the sum of param widths is not multiples of the register width. Such a gap can be filled with functionality that is read-only, for example, static or status. If params cannot be read, then it is also safe to fill the gap with irq (if it is cleared on read) or proc with only returns or upstream. If params can be read, then it is also safe to fill the gap with returns if it will not create an exit or strobe register.

   (b) Proc exit register is pure, or stream is upstream, and params are not readable. In such case, it is safe to place proc or stream params in the exit register of another proc or strobe register of another stream.

3. Array functionalities should be registerified before single functionalities. It is easier to place single functionalities in the gaps created during array functionalities registerification than the reverse way.

4. Groups (functionalities belonging to groups) should be registerified before functionalities without groups. This is because groups impose relative placement of functionalities.

5. The order of groups registerification and the order of functionalities registerification within the groups are separate, orthogonal issues. The implementation should not introduce unnecessary dependency.

6. Single and array functionalities should be sorted before registerification. Wider functionalities should be registerified as the first ones. For example, lets consider bus description from listing 47. The proc `P` being encapsulated functionality is registerified as the first one. It leaves a 12-bits gap. If statuses are registerified in the appearance order, then 3 registers are needed. This is shown in figure 7.7.

```
Main bus
    P proc
        p param; width = 20
    S0 status; width = 4
    S1 status; width = 12
    S2 status; width = 28
```

Listing 47: Bus description presenting sorting effect on registerification result.

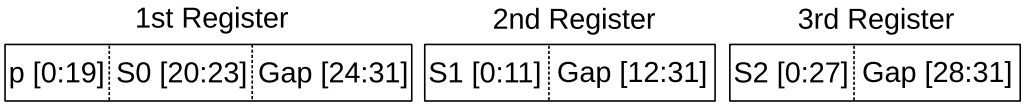


Figure 7.7: Register layout without functionality sorting.

However, if functionalities are first sorted in width decreasing order, then only 2 registers are needed. This is shown in figure 7.8.

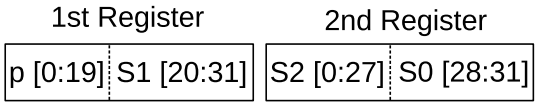


Figure 7.8: Register layout with functionality sorting.

This recommendation does not apply to single functionalities wider than the bus width. Such a case is more complicated as the optimal registerification depends also on the access atomicity. One possible implementation is to take the widest functionality and check if it can fulfill the last gap. If not, then simply registerify it starting from the next address. This approach is very simple to implement. However, it is not optimal in terms of the generated address space size.

7. Writable functionalities, such as config or mask, should be registerified before read-only functionalities, such as status and static. This is because read-only functionalities are very flexible. They can be placed in almost any gap. For example, let's consider bus description from listing 48. If statuses are registerified before configs,

```
Main bus
    S0 status; width = 16
    S1 status; width = 10
    C0 config; width = 16
    C1 config; width = 10
```

Listing 48: Bus description presenting registerification order change.

then 3 registers are required. This is shown in figure 7.9.

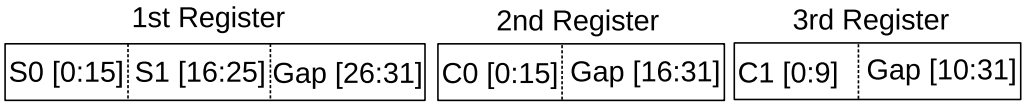


Figure 7.9: Register layout for status -> config order.

If configs are registerified before statuses, then 2 registers are required. This is shown in figure 7.10.

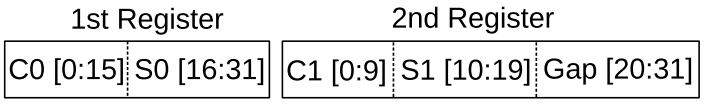


Figure 7.10: Register layout for config -> status order.

7.2 Back-end

The back-end of the compiler is responsible for generating code for a particular target. It must generate the means required to access the functionalities. As there is no inter-dependency between code generated for different targets, it is easy to run target code generation in parallel for multiple targets.

The architecture and design of the code generated for the target highly depend on the overall system requirements. Access to the data can be implemented in a synchronous or asynchronous fashion. Asynchronous code is conceptually harder to generate and use but potentially (if done right) improves system performance. The generated target code can load the address map statically or dynamically. Dynamic loading of address map is harder to implement but can be very useful when working with multiple versions of the same description or with various devices with entirely different buses. In the case of dynamic address map loading, there is no need to regenerate the target code and potentially recompile the code, as dynamic loading requires only the registerification results. The target language also is an important factor. Generating code for dynamic, weakly typed languages (e.g. Python, Perl, Lua) is generally a simpler task than generating code for compiled, strongly-typed languages (e.g. Ada, C, Rust).

Figure 7.11 presents a simplified connection scheme of a system utilizing FBDL. It shows two modules within the requester and the provider, but an even more elementary design with a single module is possible. However, what is more important is the fact that

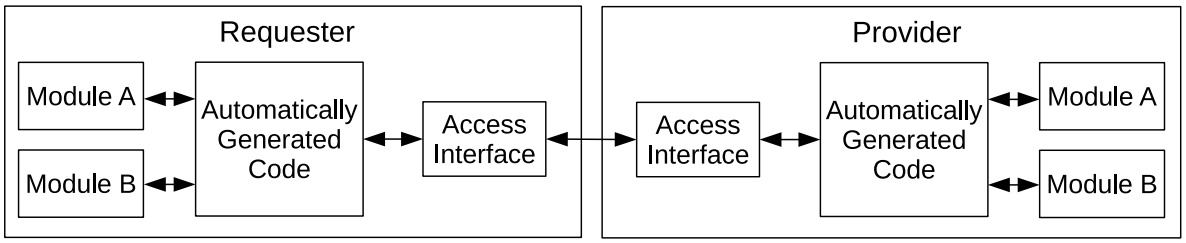


Figure 7.11: Simplified connection scheme of a system utilizing FBDL.

automatically generated code has to be connected with the access interface. The access interface code can also be generated by the compiler, but it not recommended approach because of at least two reasons. The first one is that FBDL does not specify anything about the access interface, so keeping it out of the compiler keeps the whole design architecture cleaner. The second one is that in case of extending the interface or replacing it with another one, for example, to improve performance, there is no need to regenerate the code or recompile the compiler.

In theory a functionally complete access interface requires only two functions:

1. read,
2. write.

However, single read and single write functions may not be sufficient in a system having rigid performance requirements. Frequently carried transactions are block read and block write, as well as accessing register with the same address multiple times (often called cyclic/fixed read/write or constant address read/write), for example, to read a FIFO. A slightly enhanced access interface should also provide distinct functions for:

1. block read,
2. block write,
3. cyclic read,
4. cyclic write.

In more complex systems, there also may be a need for vectored (scatter/gather) IO. In such a case, the access interface should also provide distinct functions for:

1. vectored read,
2. vectored write.

In the case of the most performance-demanding systems, there also might be a need for cyclic block (also called wrapped transactions) and cyclic vectored transactions.

In practice, a functionally complete access interface requires the following twelve functions:

1. `read` - single register read,
2. `write` - single register write,
3. `cread` - cyclic (fixed) read,
4. `cwrite` - cyclic (fixed) write,
5. `readb` - block read,
6. `writeb` - block write,
7. `creadb` - cyclic block read (wrapped read),
8. `cwriteb` - cyclic block write (wrapped write),
9. `readv` - vectored (scatter/gather) read,
10. `writev` - vectored (scatter/gather) write,
11. `creadv` - cyclic vectored (scatter/gather) read,
12. `cwritev` - cyclic vectored (scatter gather) write.

The list proposes names for particular transactions. As the names for vectored operations (`readv`, `writev`) are already defined in the POSIX standard, it makes sense to extend this naming convention further. This implies that the type of the operation is indicated by the single character suffix, `b` for block and `v` for vectored. Single read (`read`) and single write (`write`) do not have any suffixes, as this is common practice. Whether the transaction is cyclic is indicated by the single letter prefix (`c`).

The access interface does not have to provide all of the transactions, and even if it does, the last ten of them can be implemented on top of the `read` and `write`. In such a case, there will not be any performance gain, but the programming interface will be easier to use, as there will be no need to implement these functions manually. It is worth mentioning, that the performance of the access interface can be improved step by step only when necessary. For example, in the project's initial phase, the `readb`/`writeb` can be internally implemented as a loop of `read`/`write` calls. If, in a later phase, the performance of the block transactions becomes a bottleneck of the system, a true block access can be added to the interface internal implementation. The access interface can also be completely reworked or replaced at any phase of the project, and this will not result in any changes in the bus description. In other words, the bus description and the access interface are entirely independent.

There is also one more transaction type frequently found in access interfaces, the `rmw` (read-modify-write) transaction. The `rmw` is an atomic operation typically used to implement synchronization mechanisms or to reduce the round-trip latency. For example, if the provider supports `rmw` internally, the round-trip of remote access is cut by half, or even more if the requester does not care about the acknowledgment. Figure 7.12 presents sequence diagrams for `rmw` transaction without and with provider support for `rmw`. The last acknowledgment message may be ignored if the requester does not care whether the operation succeded or failed.

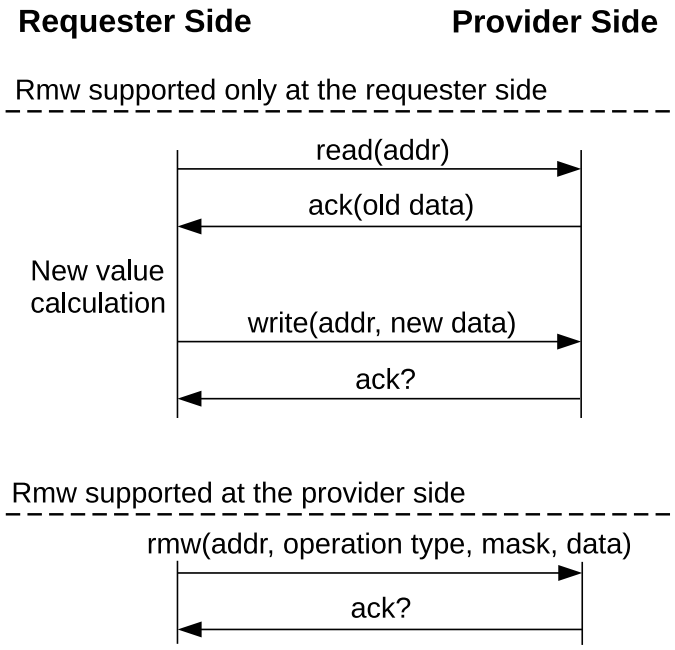


Figure 7.12: Sequence diagrams for `rmw` transaction without and with provider support for `rmw`.

The `rmw` transaction at the provider side can be implemented in two ways. In the first way, the `rmw` transaction is part of the bus protocol and is supported by the main bus master. This way provides the lowest possible latency for the `rmw`. In the second way, there is an extra, dedicated master offering `rmw` implemented as an FBDL procedure. Listing 49 presents an example RMW procedure described in FBDL. In real use cases, the widths of parameters depend on actual bus architecture. All remaining functionalities have been removed for brevity.

Figure 7.13 shows an example bus structure with an extra master providing `rmw` transaction support. Such a design has higher `rmw` transaction latency than a design with `rmw` transaction supported directly by the primary master, as the primary master has to first write `rmw` parameters in the extra master. However, the overall `rmw` latency is still much lower than in the case when the provider does not support `rmw` transaction at all.

```
Main bus
    RMW proc
        addr param
        operation_type param
        data param
        data_mask param
```

Listing 49: Example read-modify-write FBDL procedure.

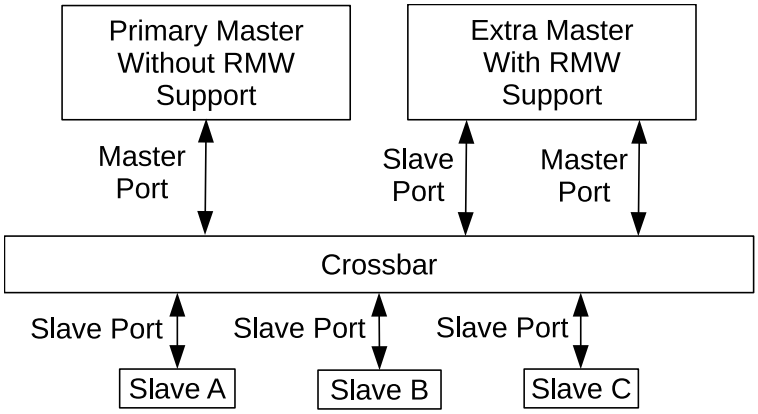


Figure 7.13: Example bus structure with extra master providing `rmw` transaction support.

8 Real use case

The implemented FBDL compiler has been used during the development of the delay generator module for femtosecond laser implemented as a part of the „Development of optical engine for rapid laser fabrication of transparent materials" (Eurostars-2) project carried out by the Fluence SP. Z O. O. The objective of the project was the development of a beam delivery module containing an optical Pancharatnam-Berry phase element and a laser equipped with precise pulse-on-demand synchronization for high-speed laser processing of transparent materials.

Due to the proprietary nature of the project, no internal details can be revealed. However, appendix C contains the statement from the Fluence company confirming the use of the FBDL compiler.

9 Summary

Describing system bus at the functional level using FBDL offers the following advantages compared to the typical register-centric approach:

1. Shorter development time, as more code can be automatically generated.
2. More readable and maintainable project structure. As FBDL is more strongly typed than the typical register-centric approach the description itself contains more information about the system. There is no need to read gateware, firmware, or software code to get to know that a certain set of registers form a broader context and are dependent (procedures and streams).
3. No space for invalid access order bugs. Code for writing parameters or reading returns of procedures and streams is automatically generated so the register with the associated strobe or acknowledgment signal is always accessed as the last one.
4. Less probability of non-atomic data access bugs. In FBDL access to any data is atomic by default. Any compiler claiming compliance with the FBDL specification must guarantee that the generated gateware or hardware provides atomic data access by default. Non-atomicity is an opt-out feature, achieved with explicit `atomic = false` property assignment.
5. Uniform data access interface across different target languages. The FBDL specification states what kind of accesses must be generated for particular functionalities. This eliminates scenarios where the generated C code provides information on addresses, masks, and shifts, but for example, the generated Python or C++ code abstracts this information by providing direct operations on registers and bit fields. The abstraction level of the code generated by the FBDL compiler is the same regardless of the target language and is always at the functionality level.

The FBDL may also be used in the case of on-chip connections utilizing the NoC technology. As each node of the network has to distribute data within its borders, the traditional bus architectures are still used for this purpose. In such a design, the FBDL may be used to describe the functionality of particular buses of nodes. The routing algorithm and access interfaces are then implemented independently and are only hooked to the code generated by an FBDL compiler.

Bibliography

[1] T. Ablyazimov, "Challenges in QCD matter physics –The scientific programme of the Compressed Baryonic Matter experiment at FAIR," *The European Physical Journal A*, vol. 53, no. 3, p. 60, Mar. 2017. [Online]. Available: https://doi.org/10.1140/epja/i2017-12248-y

[2] "Compressed Baryonic Matter experiment," Accessed: 2023-10-01. [Online]. Available: https://www.gsi.de//work/forschung/cbmnqm/cbm.htm

[3] W. M. Zabołotny, M. Gumiński, M. Kruszewski, and W. F. Müller, "Control and diagnostics system generator for complex fpga-based measurement systems," *Sensors*, vol. 21, no. 21, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/21/7378

[4] ARM, "AMBA AXI and ACE Protocol Specification," Accessed: 2021-06-03. [Online]. Available: https://developer.arm.com/documentation/ihi0022/latest/

[5] OpenCores, "WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores," Accessed: 2021-06-03. [Online]. Available: https://cdn.opencores.org/downloads/wbspec_b4.pdf

[6] G. De Michell and R. Gupta, "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349–365, Mar. 1997, Conference Name: Proceedings of the IEEE.

[7] J. Takalo, J. Kääriäinen, P. Parviainen, and T. Ihme, "Challenges of software-hardware co-design," *VTT WORKING PAPERS*, p. 49, 2008.

[8] J. Kokila, N. Ramasubramanian, and S. Indrajeet, "A survey of hardware and software co-design issues for system on chip design," in *Advanced Computing and Communication Technologies*, R. K. Choudhary, J. K. Mandal, N. Auluck, and H. A. Nagarajaram, Eds. Singapore: Springer Singapore, 2016, pp. 41–49.

[9] R. Ganesh, "Design issues in hardware/software co-design r. ganesh," 06 2020.

[10] F. Zhang, *High-speed Serial Buses in Embedded Systems*. Springer Singapore, 2020.

[11] Intel, "Isa bus specification and application notes," Accessed: 2023-03-14. [Online]. Available: https://archive.org/details/bitsavers_intelbusSpep89_3342148/mode/2up

[12] "Extended industry standard architecture (eisa) specification 3.1," Accessed: 2023-03-14. [Online]. Available: https://www.os2museum.com/files/docs/EISA_Specification-v3.1.pdf

[13] T. Time, "Micro channel bus tutorial," Accessed: 2023-03-14. [Online]. Available: https://github.com/schlae/mca-tutorial

[14] Infotel, "Vesa local bus 3486, 786 mini-board user's manual," Accessed: 2023-03-14. [Online]. Available: https://theretroweb.com/motherboard/manual/vlbus3486-61e8755ab7989037625802.pdf

[15] A. N. S. Institute, "Small computer system interface-2," Accessed: 2023-03-14. [Online]. Available: https://global.ihs.com/doc_detail.cfm?document_name=ANSI%20INCITS%20131&item_s_key=00009673&item_key_date=911231

[16] Hewlett-Packard, Intel, Microsoft, Renesas, ST-Ericsson, and T. Instruments, "Universal serial bus 3.1 specification," Accessed: 2023-03-14. [Online]. Available: https://manuais.iessanclemente.net/images/b/bc/USB_3_1_r1.0.pdf

[17] P. SIG, "Pci specifications," Accessed: 2023-03-14. [Online]. Available: https://pcisig.com/specifications

[18] S. Pasricha and N. Dutt, *On-Chip Communication Architectures*. Elsevier, 2008.

[19] J. Bainbridge, *Asynchronous System-on-Chip Interconnect*. Springer London, 2002.

[20] I. Microelectronics, "Coreconnect bus architecture," Accessed: 2023-03-14. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/user_guides/crcon_pb.pdf

[21] Intel, "Introduction to the avalon interface specifications," Accessed: 2023-03-14. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/introduction-to-the-interface-specifications.html

[22] STMicroelectronics, "Stbus communication system concepts and definitions," Accessed: 2023-03-14. [Online]. Available: https://www.st.com/resource/en/user_manual/cd00176920-stbus-communication-system-concepts-and-definitions-stmicroelectronics.pdf

[23] W. Bainbridge and S. Furber, "Marble: an asynchronous on-chip macrocell bus," *Microprocessors and Microsystems*, vol. 24, no. 4, pp. 213–222, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933100000752

[24] ARM, "Amba axi-stream protocol specification," Accessed: 2023-04-10. [Online]. Available: https://developer.arm.com/documentation/ihi0051/latest/

[25] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*, 2000, pp. 250–256.

[26] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 684–689.

[27] L. Benini and G. De Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[28] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, 2002, pp. 117–124.

[29] A. Xilinx, "Axi adapter interface protocols," Accessed: 2023-03-12. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/AXI-Adapter-Interface-Protocols

[30] J. Chen, P. Gillard, and C. Li, "Network-on-chip (noc) topologies and performance: A review," 2011.

[31] T. N. Kamal Reddy, A. K. Swain, J. K. Singh, and K. K. Mahapatra, "Performance assessment of different network-on-chip topologies," in *2014 2nd International Conference on Devices, Circuits and Systems (ICDCS)*, 2014, pp. 1–5.

[32] H. J. Mahanta, A. Biswas, and M. A. Hussain, "Networks on chip: The new trend of on-chip interconnection," in *2014 Fourth International Conference on Communication Systems and Network Technologies*, 2014, pp. 1050–1053.

[33] A. Kalita, K. Ray, A. Biswas, and M. A. Hussain, "A topology for network-on-chip," in *2016 International Conference on Information Communication and Embedded Systems (ICICES)*, 2016, pp. 1–7.

[34] A. Kumar, S. Tyagi, and C. K. Jha, "Performance analysis of network-on-chip topologies," *Journal of Information and Optimization Sciences*, vol. 38, no. 6, pp. 989–997, 2017. [Online]. Available: https://doi.org/10.1080/02522667.2017.1372145

[35] I. A. Alimi, R. K. Patel, O. Aboderin, A. M. Abdalla, R. A. Gbadamosi, N. J. Muga, A. N. Pinto, and A. L. Teixeira, "Network-on-chip topologies: Potentials,

technical challenges, recent advances and research direction," in *Network-on-Chip*, I. A. Alimi, O. Aboderin, N. J. Muga, and A. L. Teixeira, Eds. Rijeka: IntechOpen, 2021, ch. 3. [Online]. Available: https://doi.org/10.5772/intechopen.97262

[36] I. E. T. Force, "Internet Protocol," Internet Engineering Task Force, Request for Comments RFC 791, Sep. 1981, Num Pages: 51. [Online]. Available: https://datatracker.ietf.org/doc/rfc791

[37] ——, "Transmission Control Protocol," Internet Engineering Task Force, Request for Comments RFC 793, Sep. 1981, Num Pages: 91. [Online]. Available: https://datatracker.ietf.org/doc/rfc793

[38] O. Ben-Kiki, C. Evans, and I. dot Net, "Yaml specification index," Accessed: 2023-02-18. [Online]. Available: https://yaml.org/spec/

[39] noasic GmbH, "airhdl," Accessed: 2023-02-18. [Online]. Available: https://airhdl.com/#/

[40] "Introducing json," Accessed: 2023-02-18. [Online]. Available: https://www.json.org/json-en.html

[41] "Html standard," Accessed: 2023-02-18. [Online]. Available: https://html.spec.whatwg.org/

[42] W. W. W. Consortium, "Extensible markup language," Accessed: 2023-02-18. [Online]. Available: https://www.w3.org/TR/xml/

[43] W. M. Zabołotny, "Address generator for wishbone," Accessed: 2023-02-20. [Online]. Available: https://github.com/wzab/agwb

[44] ——, "Adr_gen - automatic address generator," Accessed: 2023-02-20. [Online]. Available: https://github.com/wzab/wzab-hdl-library/tree/master/addr_gen

[45] D. Gisselquist, "Autofpga," Accessed: 2023-02-21. [Online]. Available: https://github.com/ZipCPU/autofpga

[46] P. Plutecki, B. P. Bielawski, and A. Butterworth, "Code Generation Tools and Editor for Memory Maps," *Proceedings of the 17th International Conference on Accelerator and Large Experimental Physics Control Systems*, vol. ICALEPCS2019, pp. 4 pages, 0.730 MB, 2020, Artwork Size: 4 pages, 0.730 MB ISBN: 9783954502097 Medium: PDF Publisher: JACoW Publishing, Geneva, Switzerland. [Online]. Available: https://jacow.org/icalepcs2019/doi/JACoW-ICALEPCS2019-MOPHA115.html

[47] "cheby," Accessed: 2023-02-22. [Online]. Available: https://gitlab.cern.ch/be-cem-edl/common/cheby

[48] A. J. Rey, J. C. Molendijk, F. Dubouchet, A. Pashnin, A. Butterworth, M. Jaussi, and T. Levens, "Cheburashka: A tool for consistent memory map configuration across hardware and software," Oct. 2013, pp. 848–851.

[49] E. Bolnov, "Corsair," Accessed: 2023-02-24. [Online]. Available: https://github.com/esynr3z/corsair

[50] W. M. Zabołotny, M. Gumiński, and M. Kruszewski, "Automatic management of local bus address space in complex FPGA-implemented hierarchical systems," in *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2019*, R. S. Romaniuk and M. Linczuk, Eds., vol. 11176, International Society for Optics and Photonics. SPIE, 2019, p. 1117642. [Online]. Available: https://doi.org/10.1117/12.2536259

[51] L. Vik, "hdl_registers: An open-source HDL register generator fast enough to run in real time," Accessed: 2023-02-17. [Online]. Available: https://hdl-registers.com/index.html

[52] K. T. Pozniak, "I/O communication with FPGA circuits and hardware description standard for applications in HEP and FEL electronics." [Online]. Available: https://romaniuk.web.cern.ch/public-files/TESLA/tesla2005-22.pdf

[53] "Deutsches elektronen-synchrotron," Aug. 2012.

[54] P. Drabik and K. T. Pozniak, "Maintaining complex and distributed measurement systems with component internal interface framework," in *Proc. SPIE*, R. S. Romaniuk and K. S. Kulpa, Eds., vol. 7502, Wilga, Poland, Jun. 2009, p. 75022C. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.838155

[55] A. Zagoździńska, K. T. Poźniak, and P. K. Drabik, "Selected issues of the universal communication environment implementation for CII standard," R. S. Romaniuk, Ed., Wilga, Poland, Jun. 2011, p. 80080N. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.902748

[56] "1685-2014 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows." [Online]. Available: https://ieeexplore.ieee.org/document/6898803

[57] A. I.-X. W. Group, "Ip-xact user guide," Accessed: 2023-02-24. [Online]. Available: https://www.accellera.org/images/downloads/standards/ip-xact/IP-XACT_User_Guide_2018-02-16.pdf

[58] T. Timi, “rgen,” Accessed: 2023-02-24. [Online]. Available: https://github.com/tudortimi/rgen

[59] O. Balci, “Ipxact register map generator,” Accessed: 2023-02-24. [Online]. Available: https://github.com/legoritma/ipxact-register-generator

[60] A. Kamppi, L. Matilainen, J. M. Maatta, E. Salminen, T. D. Hamalainen, and M. Hannikainen, “Kactus2: Environment for embedded product development using ip-xact and mcapi,” in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, Aug 2011, pp. 262–265.

[61] F. Miller, “Root-of-trust-architekturen als open-source-hardware und deren zertifizierung: am beispiel von opentitan,” *Datenschutz und Datensicherheit*, vol. 44, no. 7, pp. 451–455, 2020.

[62] “Opentitan,” Accessed: 2023-02-22. [Online]. Available: https://docs.opentitan.org/

[63] “Opentitan register tool,” Accessed: 2023-02-22. [Online]. Available: https://docs.opentitan.org/doc/rm/register_tool/

[64] bitvis, “Register wizard abandoned,” Accessed: 2023-02-24. [Online]. Available: https://github.com/UVVM/UVVM/issues/200

[65] E. Tallaksen, “Auto-generate register related code and documentation - for free,” Accessed: 2023-02-24. [Online]. Available: https://www.linkedin.com/pulse/auto-generate-register-related-code-doc-free-espen-tallaksen/

[66] bitvis, “Verifying corner cases in a structured manner - using vhdl verification components,” Accessed: 2023-02-24. [Online]. Available: http://program.fpgaworld.com/2016/More_information/Bitvis___Verifying_CornerCases_Handout.pdf

[67] T. Ishitani, “Rggen,” Accessed: 2023-02-16. [Online]. Available: https://github.com/rggen/rggen

[68] A. Chapyzhenka, “Design hardware,” Accessed: 2023-02-16. [Online]. Available: https://github.com/sifive/duh

[69] accellera SYSTEMS INITIATIVE, “Systemrdl,” Accessed: 2023-02-18. [Online]. Available: https://www.accellera.org/downloads/standards/systemrdl

[70] AGNISYS, “Next generation systemrdl - using idesignspec for register implementation,” Accessed: 2023-02-16. [Online]. Available: https://www.agnisys.com/blog/next-generation-systemrdl-using-idesignspec-for-register-implementation

[71] eVision Systems, "Idesignspec," Accessed: 2023-02-16. [Online]. Available: https://evision-systems.com/linecard/agnisys/idesignspec/

[72] Semifore, "Csrcompiler," Accessed: 2023-02-16. [Online]. Available: http://semifore.com/csrcompiler/

[73] Juniper, "open-register-design-tool," Accessed: 2023-02-16. [Online]. Available: https://github.com/Juniper/open-register-design-tool

[74] S. open source community, "Free and open-source systemrdl tools," Accessed: 2023-02-16. [Online]. Available: https://github.com/SystemRDL

[75] M. Buechler, "desyrdl," Accessed: 2023-02-16. [Online]. Available: https://gitlab.desy.de/fpgafw/tools/desyrdl

[76] J. van Straten, "vhdmmio," Accessed: 2023-03-23. [Online]. Available: https://github.com/abs-tudelft/vhdmmio

[77] T. Włostowski, "Wishbone slave generator," Accessed: 2023-02-16. [Online]. Available: https://ohwr.org/project/wishbone-gen

[78] A. Xilinx, "Super logic region," Accessed: 2023-03-12. [Online]. Available: https://docs.xilinx.com/r/en-US/ug912-vivado-properties/SLR

[79] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[80] M. Kruszewski, "go-fbdl," Accessed: 2023-04-14. [Online]. Available: https://github.com/Functional-Bus-Description-Language/go-fbdl

[81] ——, "go-vfbdb," Accessed: 2023-04-14. [Online]. Available: https://github.com/Functional-Bus-Description-Language/go-vfbdb

[82] M. Brunsfeld, A. Hlynskyi, P. Thomson, J. Vera, P. Turnbull, T. Clem, D. Creager, A. Helwer, R. Rix, H. van Antwerpen, M. Davis, Ika, T.-A. Nguyễn, S. Brunk, N. Hasabnis, bfredl, M. Dong, M. Massicotte, J. Arnett, V. Panteleev, S. Kalt, K. Lampe, A. Pinkus, M. Schmitz, M. Krupcale, narpfel, S. Gallegos, V. Martí, and Edgar, "tree-sitter/tree-sitter: v0.20.8," Apr. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7798573

[83] M. Kruszewski, "tree-sitter-fbdl," Accessed: 2023-04-13. [Online]. Available: https://github.com/Functional-Bus-Description-Language/tree-sitter-fbdl

[84] M. Tomita, Ed., *Generalized LR Parsing*. Boston, MA: Springer US, 1991. [Online]. Available: http://link.springer.com/10.1007/978-1-4615-4034-2

List of Figures

List of Tables

Appendices

A Supervisor registerification results

Functionality addresses are relative addresses. Absolute addresses are obtained by adding block start address.


```
{
  "Name": "Main",
  "Doc": "",
  "IsArray": false,
  "Count": 1,
  "Masters": 1,
  "Reset": "",
  "Width": 32,
  "Sizes": { "BlockAligned": 32, "Compact": 10, "Own": 1 },
  "AddrSpace": { "Start": 0, "End": 31 },
  "BoolConsts": null,
  "BoolListConsts": null,
  "FloatConsts": null,
  "IntConsts": null,
  "IntListConsts": null,
  "StrConsts": null,
  "Configs": null,
  "Irqs": null,
  "Masks": null,
  "Memories": null,
  "Procs": null,
  "Statics": [
    {
      "Name": "ID",
      "Doc": "Bus identifier.",
      "IsArray": false,
      "Count": 1,
      "Groups": null,
      "InitValue": "x\"39a90380\"",
      "ReadValue": "",
      "ResetValue": "",
      "Width": 32,
      "Access": { "Strategy": "Single", "Addr": 0, "StartBit": 0, "EndBit": 31 }
    }
  ],
  "Statuses": null,
  "Streams": null,
  "Subblocks": [
    {
      "Name": "Supervisor",
      "Doc": "",
      "IsArray": false,
      "Count": 1,
      "Masters": 1,
      "Reset": "",
      "Width": 32,
```

```
"Sizes": { "BlockAligned": 16, "Compact": 9, "Own": 9 },
"AddrSpace": { "Start": 16, "End": 31 },
"BoolConsts": null,
"BoolListConsts": null,
"FloatConsts": null,
"IntConsts": { "WORKER_COUNT": 24 },
"IntListConsts": null,
"StrConsts": null,
"Configs": null,
"Irqs": null,
"Masks": [
  {
    "Name": "Workers_Mask",
    "Doc": "",
    "IsArray": false,
    "Count": 1,
    "Atomic": true,
    "Groups": null,
    "InitValue": "",
    "ReadValue": "",
    "ResetValue": "",
    "Width": 24,
    "Access": { "Strategy": "Single", "Addr": 5, "StartBit": 0, "EndBit": 23 }
  }
],
"Memories": null,
"Procs": [
  {
    "Name": "Reset_Counter",
    "Doc": "",
    "IsArray": false,
    "Count": 1,
    "Delay": null,
    "Params": null,
    "Returns": null,
    "CallAddr": 0,
    "ExitAddr": null
  },
  {
    "Name": "Program",
    "Doc": "",
    "IsArray": false,
    "Count": 1,
    "Delay": null,
    "Params": [
      {
        "Name": "counter_value",
        "Doc": "",
        "IsArray": false,
        "Count": 1,
        "Groups": null,
        "Range": null,
        "Width": 48,
        "Access": {
          "Strategy": "Continuous",
          "RegCount": 2, "StartAddr": 1, "StartBit": 0, "EndBit": 15
```

```
        }
      },
      {
        "Name": "worker_data",
        "Doc": "",
        "IsArray": true,
        "Count": 2,
        "Groups": null,
        "Range": null,
        "Width": 12,
        "Access": {
          "Strategy": "Continuous",
          "RegCount": 2, "ItemCount": 2, "ItemWidth": 12, "StartAddr": 2, "StartBit": 16
        }
      }
    ],
    "Returns": null,
    "CallAddr": 3,
    "ExitAddr": null
  },
  {
    "Name": "Unprogram",
    "Doc": "",
    "IsArray": false,
    "Count": 1,
    "Delay": null,
    "Params": null,
    "Returns": null,
    "CallAddr": 4,
    "ExitAddr": null
  }
],
"Statics": null,
"Statuses": [
  {
    "Name": "Counter",
    "Doc": "",
    "IsArray": false,
    "Count": 1,
    "Atomic": true,
    "Groups": null,
    "ReadValue": "",
    "Width": 48,
    "Access": {
        "Strategy": "Continuous", "RegCount": 2, "StartAddr": 6, "StartBit": 0, "EndBit": 15
    }
  },
  {
    "Name": "Workers_Ready",
    "Doc": "",
    "IsArray": false,
    "Count": 1,
    "Atomic": true,
    "Groups": null,
    "ReadValue": "",
    "Width": 24,
```

```
          "Access": { "Strategy": "Single", "Addr": 8, "StartBit": 0, "EndBit": 23 }
        },
        {
          "Name": "programmed",
          "Doc": "",
          "IsArray": false,
          "Count": 1,
          "Atomic": true,
          "Groups": [ "status" ],
          "ReadValue": "",
          "Width": 1,
          "Access": { "Strategy": "Single", "Addr": 8, "StartBit": 24, "EndBit": 24 }
        },
        {
          "Name": "programmed_in_past",
          "Doc": "",
          "IsArray": false,
          "Count": 1,
          "Atomic": true,
          "Groups": [ "status" ],
          "ReadValue": "",
          "Width": 1,
          "Access": { "Strategy": "Single", "Addr": 8, "StartBit": 25, "EndBit": 25 }
        }
      ],
      "Streams": null,
      "Subblocks": null
    }
  ]
}
```

B Python code for mask access

```
def calc_mask(m):
    """
    calc_mask calculates mask based on tuple (End Bit, Start Bit).
    The returned mask is shifted to the right.
    """
    return (((1 << (m[0] + 1)) - 1) ^ ((1 << m[1]) - 1)) >> m[1]

class SingleSingle:
    def __init__(self, iface, addr, mask):
        self.iface = iface
        self.addr = addr
        self.mask = calc_mask(mask)
        self.width = mask[0] - mask[1] + 1
        self.shift = mask[1]

    def read(self):
        return (self.iface.read(self.addr) >> self.shift) & self.mask

class MaskSingleSingle(SingleSingle):
    def __init__(self, iface, addr, mask):
        super().__init__(iface, addr, mask)

    def _bits_to_iterable(self, bits):
        if bits == None:
            return range(self.width)
        elif type(bits) == int:
            return (bits,)
        return bits

    def _assert_bits_in_range(self, bits):
        for b in bits:
            assert 0 <= b < self.width, "mask overrange"

    def _assert_bits_to_update(self, bits):
        if bits == None:
            raise Exception("bits to update cannot have None value")
        if type(bits).__name__ in ["list", "tuple", "range", "set"] and \
            len(bits) == 0:
            raise Exception("empty " + type(bits) + " of bits to update")
```

```
def set(self, bits=None):
    bits = self._bits_to_iterable(bits)
    self._assert_bits_in_range(bits)

    mask = 0
    for b in bits:
        mask |= 1 << b

    self.iface.write(self.addr, mask << self.shift)

def clear(self, bits=None):
    bits = self._bits_to_iterable(bits)
    self._assert_bits_in_range(bits)

    mask = self.mask
    for b in bits:
        mask ^= 1 << b

    self.iface.write(self.addr, mask << self.shift)

def toggle(self, bits=None):
    bits = self._bits_to_iterable(bits)
    self._assert_bits_in_range(bits)

    xor_mask = 0
    for b in bits:
        xor_mask |= 1 << b
    xor_mask <<= self.shift

    mask = self.iface.read(self.addr) ^ xor_mask
    self.iface.write(self.addr, mask)

def update_set(self, bits):
    self._assert_bits_to_update(bits)

    bits = self._bits_to_iterable(bits)
    self._assert_bits_in_range(bits)

    mask = 0
    for b in bits:
        mask |= 1 << b

    mask = self.iface.read(self.addr) | (mask << self.shift)
    self.iface.write(self.addr, mask)
```

```
def update_clear(self, bits):
    self._assert_bits_to_update(bits)

    bits = self._bits_to_iterable(bits)
    self._assert_bits_in_range(bits)

    mask = 2**BUS_WIDTH - 1
    for b in bits:
        mask ^= 1 << b

    mask = self.iface.read(self.addr) & (mask << self.shift)
    self.iface.write(self.addr, mask)
```

C Statement from the Fluence company

Warsaw, 09.06.2023

Fluence .technology
Fluence sp. z o.o.
ul. Kolejowa 5/7
01-217 Warszawa
NIP: 527-277-61-54
REGON: 365029156
KRS: 0000629831

Statement

The FBDL compiler has been used during the development of the delay generator module for femtosecond laser implemented as a part of the „Development of optical engine for rapid laser fabrication of transparent materials" (Eurostars-2) project carried out by the Fluence SP. Z O. O.

The functionality-centric approach resulted in shorter implementation time and increased system maintainability compared to the previous custom register-centric approach.

Project Manager

Członek Zarządu

dr Piotr Skibiński

D FBDL Specification

Functional Bus Description Language

Revision 1.0

13 June 2023

Abstract

This document is the official specification of the Functional Bus Description Language. Its main purpose is to define the syntax and semantics of the language. Functional Bus Description Language is a domain-specific language for bus and registers management. Its main characteristic is the shift of paradigm from the register-centric approach to the functionality-centric approach. In the register-centric approach user defines registers and then manually lays out the data into the registers. In the functionality-centric approach user defines the functionality of the data and the registers and bus hierarchy 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

Table of Contents

Participants

Michal 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 `stream`, the functionality is used to denote a broader context of encapsulated data.

gap
The gap term is used to refer to unused bits within 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 `stream` register 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 `.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_assignment
    { ; 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

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

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
```

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 low level to 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
        # Read_Frame reads single data frame
        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 }
```

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. There is only a 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, 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:     Blkblock
7:         C cfg_t
8:         Sstatus
```

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, const, doc, false, import, init-value, irq, mask, memory, param, proc, range, reset, read-value, reset-value, return, static, stream, true, 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' - unkown,
- '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

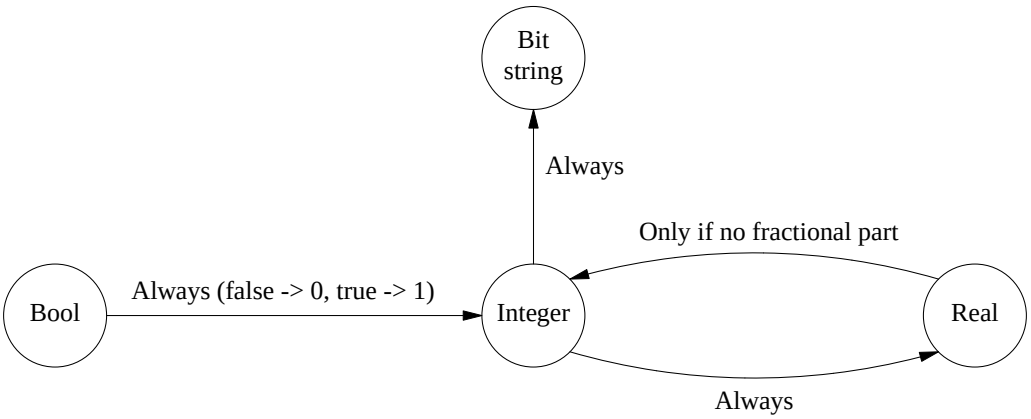
There are 6 data types in FBDL:

- 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 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 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

Z	Z

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 `groups`.

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
```

6. Expressions

An expression is a formula that defines the computation of a value by applying operators and functions to operands.

```
expression ::=
    bool_literal |
    integer_literal |
    real_literal |
    string_literal |
    bit_string_literal |
    time_literal |
    declared_identifier |
    qualified_identifier |
    unary_operation |
    binary_operation |
    function_call |
    subscript |
    parenthesized_expression |
    expression_list
```

The function call is used to call one of built-in functions.

```
function_call ::=
    declared_identifier( [ expression { , expression } ] )
```

The subscript is used to refer to a particular element from the expression list.

```
subscript ::= declared_identifier[ expression ]
```

The parenthesized expression may be used to explicitly set order of operations.

```
parenthesized_expression ::= ( expression )
```

The expression list may be used to create a list of expressions.

```
expression_list ::= [ [ expression { , expression } ] ]
```

6.1. Operators

6.1.1. Unary Operators

```
unary_operation ::= unary_operator expression

unary_operator ::= unary_arithmetic_operator | unary_bitwise_operator

unary_arithmetic_operator ::= -

unary_bitwise_operator ::= !
```

FBDL unary operators

Token	Operation	Operand Type	Result Type
-	Opposite	Integer Real	Integer Real
		Bool	Bool

!	Negation	Bit String Integer	Bit String Integer

6.1.2. Binary Operators

```
binary_operation ::= expression binary_operator expression

binary_operator ::=
    binary_arithmetic_operator |
    binary_comparison_operator |
    binary_logical_operator |
    binary_bitwise_operator

binary_arithmetic_operator ::= + | - | * | / | % | **

binary_comparison_operator ::= == | != | < | <= | > | >=

binary_logical_operator ::= && | ||

binary_bitwise_operator ::= << | >>
```

FBDL binary arithmetic operators

Token	Operation	Left Operand Type	Right Operand Type	Result Type
+	Addition	Integer	Integer	Integer
		Integer	Real	Real
		Real	Integer	Real
		Real	Real	Real
		Time	Time	Time
-	Subtraction	Integer	Integer	Integer
		Integer	Real	Real
		Real	Integer	Real
		Real	Real	Real
*	Multiplication	Integer	Integer	Integer
		Integer	Real	Real
		Real	Integer	Real
		Real	Real	Real
		Integer	Time	Time
		Time	Integer	Time
\	Division	Integer	Integer	Real
		Integer	Real	Real
		Integer	Real	Real
		Real	Real	Real
%	Remainder	Integer	Integer	Integer
**	Exponentiation	Integer	Integer	Real
		Integer	Real	Real
		Real	Integer	Real

FBDL binary comparison operators

Token	Operator	Left Operand Type	Right Operand Type	Result
==	Equality	Integer	Integer	Bool
		Integer	Real	Bool
		Real	Integer	Bool
		Real	Real	Bool
		Integer	Integer	Bool

!=	Nonequality	Integer Real Real	Real Integer Real	Bool Bool Bool
<	Less Than	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool
<=	Less Than or Equal	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool
>	Greater Than	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool
>=	Greater Than or Equal	Integer Integer Real Real	Integer Real Integer Real	Bool Bool Bool Bool

FBDL binary logical operators

Token	Operator	Left Operand Type	Right Operand Type	Result
&&	Short-circuiting logical AND	Bool	Bool	Bool
||	Short-circuiting logical OR	Bool	Bool	Bool

FBDL binary bitwise operators

Token	Operator	Left Operand Type	Right Operand Type	Result Type
<<	Left Shift	Integer	Integer	Integer
>>	Right Shift	Integer	Integer	Integer
&	And	Bit String Integer	Bit String Integer	Bit String Integer
|	Or	Bit String Integer	Bit String Integer	Bit String Integer
^	Xor	Bit String Integer	Bit String Integer	Bit String Integer

The bool data type is not valid operand type for the most of the binary operations. However, as there is the rule for implicit conversion from the bool data type to the integer data type, all operations accepting the integer operands work also for the bool operands.

6.2. 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`
The `abs` function returns the absolute value of `x`.

`bool``(x integer) bool`
The `bool` function 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`
The `ceil` function returns the least integer value greater than or equal to `x`.

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

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

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

`log``(x, b float) integer|float`
The `log` function returns the logarithm of `x` to the base `b`.

`u2``(x, w integer) integer`
The `u2` function returns two's complement representation of `x` as an integer assuming width `w`. For example `u2(-1, 8)`returns 255.

7. Functionalities

Functionalities are the core part of the FBDL. They define the capabilities of the provider. Each functionality is distinct and unambiguously defines the provider behavior and the interface that must be generated for the requester. There are following 12 functionalities:

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

7.1. Block

The `block` functionality is used to logically group or encapsulate functionalities. The `block` is usually used to separate functionalities related to 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:

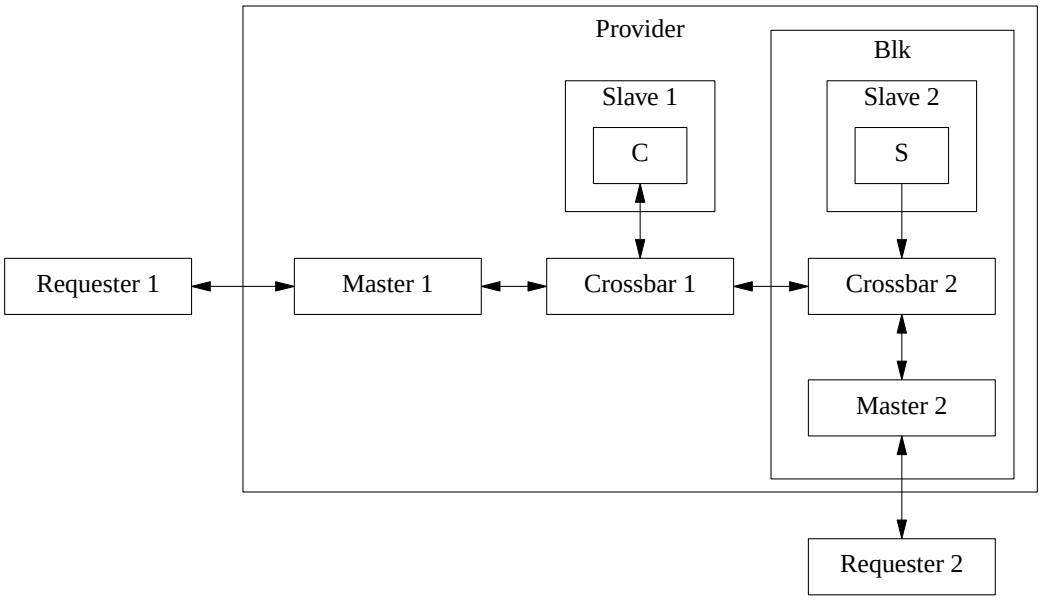
`masters` `integer (1)`
: The `masters` property defines the number of `block` masters.

`reset` `string (None)`
: The `reset` property defines the `block` reset type. By default the `block` has no reset. Valid values of the `reset` property are *`"Sync"`* for synchronous reset and *`"Async"`* for asynchronous reset.

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`.

7.2. Bus

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

The `bus` functionality has following properties:

`masters` `integer (1)`
: The `masters` property defines the number of `bus` masters.

`reset` `string (None)`
: The `reset` property defines the `bus` reset type. By default the bus has no reset. Valid values of the `reset` property are *`"Sync"`* for synchronous reset and *`"Async"`* for asynchronous reset.

`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.

7.3. Config

The `config` functionality represents configuration data. The configuration data is data that is automatically read by the provider from its registers. As the`config` is automatically read by the pro vider, there is no need for an additional signal associated with the config, indicating the 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.

`groups` `string | [string] (None)`
The `groups` property defines the groups the `config` belongs to. In case of a single group, the value can be a string. In case of multiple groups the value shall be a list of strings. Groups are thoroughly described in the grouping section.

`init-value` `bit string | integer (uninitialized)`
The `init-value` property defines the initial value of the `config`.

`range` `integer | [integer] (None)`
The `range` property defines the range of valid values. If the`range` v alue is of integer type then, the valid range is from 0 to the value, including the value. If the`range` v alue is an integer list, then it must have even number of elements. Odd elements specify lower bounds of the subranges and even elements specify upper bounds of the subranges. For instance, `range = [1, 3, 7, 8]` means that the valid values are: 1, 2, 3, 7 and 8. Range bound values shall not be negative. This is because the FBDL makes no assumptions on the negative values encoding. To accomplish negative range checks functions such as `u2` must be explicitly called. For example, following assignment limits the possible range from -16 to -8: `range = [u2(-8, 8), u2(-16, 8)]`. The`range` property shall not be e xplicitly set if the `width` property is already set. If the `range` property is not set, then the actual range implies from the `width` property. The code generated for the provider is not required to check or report if the value provided for the config write is within the valid range. The recommended way is to implement compiler parameter allowing enabling/disabling range check generation.

`read-value` `bit string | integer (None)`
The `read-value` property defines the value returned by the provider on the config read. If the `read-value` is not set, then the provider must return the actual value of the `config`.

`reset-value` `bit string | integer (None)`
The `reset-value` property defines the value of the config after the reset. If the `reset-value` is set, but a `bus` or `block` containing the `config` is not resettable (`reset = None`), then the compiler shall report an error.

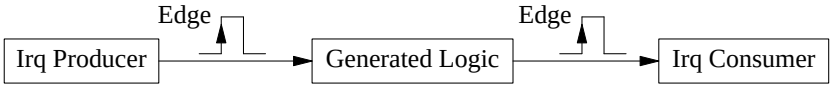
`width` `integer (bus width)`
The `width` property defines the bit width of the config. The `width` property shall not be explicitly set if the `range` property is already set.

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

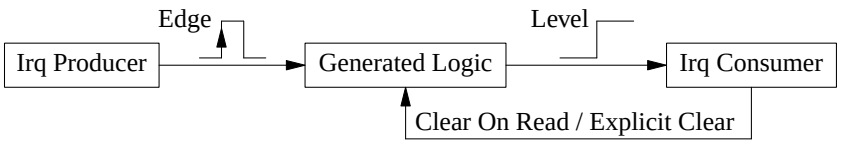
7.4. Irq

The `irq` functionality represents an interrupt handling. The `irq` functionality allows for automatic connection of the following interrupt producers (`in-trigger`) and consumers (`out-trigger`):

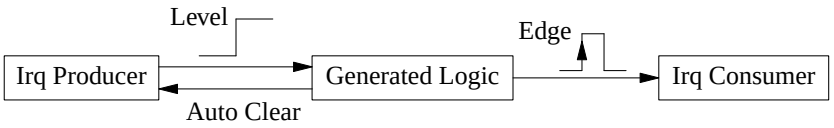
1) edge producer and edge sensitive consumer,



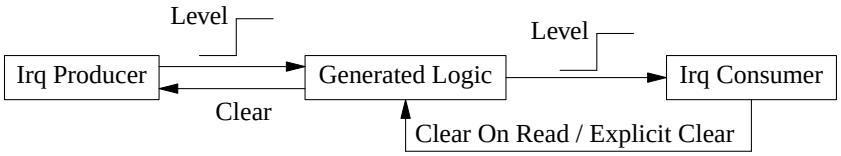
2) edge producer and level sensitive consumer,



3) level producer and edge sensitive consumer,



4) level producer and level sensitive consumer.



The `irq` functionality has following properties:

`add-enable` `bool (false)`
: The `add-enable` property defines whether an interrupt has associated enable bit in the interrupt enable register. The enable can be used to mask the interrupt.

`clear` `string ("Explicit")`
: The `clear` property defines how particular interrupt flag is cleared. The `clear` property is valid only in case of level-triggered interrupt consumer. If`clear` property is set for edge-triggered interrupt consumer a compiler shall shall report an error. Valid values are *`"Explicit"`* and *`"On Read"`*. The*`"Explicit"`* clear requires compiler to generate a means that must be explicitly used to clear the interrupt flag. The *`"On Read"`* clear requires the provider to clear the interrupt flag on each interrupt flag read.

`enable-init-value` `bit string | integer (uninitializd)`
: The `enable-init-value` property defines the initial value of the enable bit in the interrupt enable register. The value must not exceed one bit. If `add-enable` is `false` and `enable-init-value` is set, then a compiler must report an error.

`enable-reset-value` `bit string | integer (uninitializd)`
: The `enable-reset-value` property defines the value of the enable bit in the interrupt enable register after the reset. The value must not exceed one bit. If `add-enable` is `false` and `enable-reset-value` is set, then a compiler must report an error. If the`enable-reset-value` is set, b ut a `bus` or `block` containing the `irq` is not resettable (`reset = None`), then the compiler shall report an error.

`groups` `string | [string] (None)`
: The `groups` property defines the group for irq. Each `irq` must belong at most to one group. Interrupt groups are described in irq grouping subsection.

`in-trigger` `string ("Level")`
: The `in-trigger` property declares the interrupt producer type of trigger. Valid values are *`"Edge"`* and *`"Level"`*. It is up to the user to make sure declared trigger is coherent with the actual producer behavior. A mismatch may lead to incorrect behavior.

`out-trigger` `string ("Level")`
: The `out-trigger` property declares the interrupt consumer type of trigger. Valid values are *`"Edge"`* and *`"Level"`*. It is up to the user to make sure declared trigger is coherent with the actual consumer requirement. A mismatch may lead to incorrect behavior.

7.5. Mask

The `mask` functionality represents a bit mask. The mask is 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 v ery 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 with

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.

`init-value` `bit string | integer (uninitialized)`
: The `init-value` property defines the initial value of the `mask`.

`read-value` `bit string | integer (None)`
: The `read-value` property defines the value returned by the provider on the `mask` read. If the `read-value` is not set, then the provider must return the actual value of the mask.

`reset-value` `bit string | integer (None)`
: The `reset-value` property defines the value of the `mask` after the reset. If the `reset-value` is set, but a `bus` or `block` containing the `mask` is not resettable (`reset = None`), then the compiler shall report an error.

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

The code generated for the requester must provide means for setting, clearing and updating particular bits of the `mask`. The updating includes setting, clearing and toggling. The set differs from the update set. The set sets particular bits and simultaneously clears all remaining bits. The update set sets particular bits and keeps the value of the remaining bits. The clear differs from the update clear in an analogous way. The toggle always works on provided bits leaving the remaining bits untouched.

7.6. Memory

The `memory` functionality is used to directly connect and map an external memory to the generated bus address space. A memory can also be connected to the bus using the `proc` or `stream` functionality. However, using the `memory` functionality usually leads to greater throughput, but increases the size of the generated address space.

The `memory` functionality has following properties:

`access` `string (`*`"Read Write"`*`)`
: The `access` property declares the valid access permissions to the memory for the requester. Valid values of the `access` property are: *`"Read Write"`*, *`"Read Only"`*, *`"Write Only"`*.

`byte-write-enable` `bool (false)`
: The `byte-write-enable` property declares byte-enable writes, that update the memory on contents on a byte-to-byte basis. If the `byte-write-enable` property is explicitly set by a user, and a `memory` access is *`"Read Only"`*, then a compiler shall report an error.

`read-latency` `integer (obligatory if access supports read)`
: The `read-latency` property declares the read latency in the number of clock cycles. It is required, if a`memory` supports read access, to correcly implement read logic.

`size` `integer (obligatory)`
: The `size` property declares the `memory` size. The `size` is in the number of memory words with width equal to the `memory` `width` property value.

`width` `integer (bus width)`
: The `width` property declares the memory data width.

The code generated for the requester must provide means for single read/write and block read/write transactions. Whether access means for vectored (scatter-gather) transactions are automatically generated is up to the compiler. If memory is read-only or write-only, then an unsupported write or read access code is recommended not to be

generated.

7.7. Param

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

The `param` functionality has following properties:

`groups` `string | [string] (None)`
The groups property defines the groups the param belongs to. In case of a single group, the value can be a string. In case of multiple groups the value shall be a list of strings. Groups are thoroughly described in the grouping section.

`range` `integer | [integer] (None)`
The `range` property defines the range of valid values. The range property on`param` beha ves exactly the same as the `range` property on `config`.

`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
```

7.8. Proc

The `proc` functionality represents a procedure called by the requester and carried out by 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) by default has only the call signal. However, if an empty `proc` has the delay property set, then it has both the call signal and the exit signal. A `proc` having only parameters has by default only the call signal. However, if a proc having only parameters has the delay property set, then it also has the exit signal. A `proc` having only returns has by default only the exit signal. However, if a `proc` having only returns has the delay property set, then it also has the call signal. The existence or absence of call and exit signals is summarized in the below table.

Proc call and exit signals occurrence

Delay Set	Empty	Only Params	Only Returns	Params & Returns
No	call	call	exit	call & exit
Yes	call & exit	call & exit	call & exit	call & exit

The `proc` functionality has following properties:

`delay` `time (None)`
The `delay` property defines the time delay between parameters write end and returns read start.

The code generated for the requester must provide a mean for calling the procedure.

7.9. Return

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

The `return` functionality has following properties:

`groups` `string | [string] (None)`
: The `groups` property defines the groups the return belongs to. In case of a single group, the value can be a string. In case of multiple groups the value shall be a list of strings. Groups are thoroughly described in the grouping section.

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

The 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
```

7.10. Static

The `static` functionality represents data, placed at the provider side, that shall never change.

The `static` functionality has following properties:

`groups` `string | [string] (None)`
: The `groups` property defines the groups the `static` belongs to. In case of a single group, the value can be a string. In case of multiple groups the value shall be a list of strings. Groups are thoroughly described in the grouping section.

`init-value` `bit string | integer (obligatory)`
: The `init-value` property defines the initial value of the `static`.

`read-value` `bit string | integer (None)`
: The `read-value` property defines the value that must be returned by the provider on the `static` read after the first read. If the `read-value` property is set, then the actual value of the `static` can be read only once.

`reset-value` `bit string | integer (None)`
: The `reset-value` property defines the value of the static after the reset. If the `reset-value` is set, but a `bus` or `block` containing the `static` is not resettable (`reset = None`), then the compiler shall report an error. If both`read-value` and `reset-value` properties are set, then the `static` can be read one more time after the reset.

`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 = C8
    init-value = C113
    read-value = 0xFF
```

7.11. Status

The `status` represents data 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.

`groups` `string | [string] (None)`
: The `groups` property defines the groups the `status` belongs to. In case of a single group, the value can be a string. In case of multiple groups the value shall be a list of strings. Groups are thoroughly described in the grouping section.

`read-value` `bit string | integer (None)`
: The `read-value` property defines the value that must be returned by the provider on the `status` read after the first read. If the `read-value` property is set, then the actual value of 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 a mean for reading the `status`.

7.12. 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 `param` or `return`) is always a downstream. It is useful for triggering cyclic action with constant time interval. A downstream must not have any `return`. An upstream shall not have any `param`, and must have at least one `return`.

The `stream` functionality is very similar to the `proc` functionality, but they are not the same. There are two main differences. The first one is that the`stream` must not contain both `param` and `return`. The second one is that the code for the stream, generated for the requester, shall take into account the fact that access to the `stream` is multiple and 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` has associated strobe signal at the provider side. The strobe signal must be driven active for one clock cycle after all registers storing the parameters of a downstream have been written. It also must be driven active for one clock cycle after all registers storing the returns of an upstream have been read.

The `stream` functionality has following properties.

`delay` `time (None)`
: The `delay` property defines the time delay between writing/reading consecutive datasets for a downstream/upstream.

8. Parametrization

The FBDL provides the following three ways for description parametrization:

- constants,
- type definitions,
- types extending.

8.1. Constant

The constant represents a constant value. The value might be used in expression evaluations. The following code presents a bus description 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
```

8.2. Type definition

The type definition allows for defining custom functionalities. Any custom functionality resolves to one of the built-in functionalities. However, by defining custom functionality types it is possible to preset property values or to create easily parametrizable functionalities. The former leads to shorter descriptions and helps to avoid duplication.

```
type_definition ::=
    single_line_type_definition |
    multi_line_type_definition

single_line_type_definition ::=
    type
    identifier
    [ parameter_list ]
    declared_identifier | qualified identifier
    [ argument_list ]
    semicolon_and_property_assignments | newline

multi_line_type_definition ::=
    type
    identifier
    [ parameter_list ]
    declared_identifier | qualified identifier
    [ argument_list ]
    functionality_body

parameter_list ::= ( parameters )

parameters ::= parameter { , parameter }

parameter ::= identifier [ = expression ]
```

Parameters in the parameter list might have default values, but parameters with the default values must prepend parameters without default values in the parameter list.

```
argument_list ::= ( arguments )

arguments ::= argument { , argument  }

argument ::= [ declared_identifier = ] expression
```

Arguments in the argument list may be prepended with the parameter name. However, arguments with parameter names must prepend arguments without parameter names in the argument list.

The below snippet presents examples of type definitions.

```
    # Single line type definition
    type cfg_t(w = 10) config; width = w; groups = "configs"

    # Multi line type definition
    type blk_t(with_status = true, mask_count) block
        S [with_status]status
        M [mask_count]mask

    Main bus
        type irq_t irq; groups = "irq"
        I1 irq_t
        I2 irq_t

        C1 cfg_t
        C2 cfg_t(6)
        C3 cfg_t(width = 8)
```

```
    Blk1 blk_t(7)
    Blk2 blk_t(with_status = false, mask_count = 11)
```

8.3. Type extending

The type extending allows extending any custom defined type, either by instantiation or by defining a new type. This is mainly, 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.

9. Scope and visibility

9.1. 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 a 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 directory 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 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 package within the importing file. If the identifier is omitted, then the implicit identifier for the package is the last part of its path.

9.1.1. Package discovery

Each FBDL compiler is required to carry out the package auto-discovery procedure. The procedure must obey following rules.

1) If the compiler working directory contains a directory named `"fbd"`, then each of the `"fbd"` subdirectories is considered a package directory if it contains at least one file with the `".fbd"` extension. The name of the package is the same as the name of the subdirectory, unless it has `"fbd-"` prefix. In such a case, the prefix shall be removed from the package name. If the name of the subdirectory matches exactly the `"fbd-"` pattern, then a compiler must report an error on an invalid package name.
2) The compiler must recursively check all subdirectories of its working path (except the `"fbd"` directory in the working directory that is described in rule number 1). Each subdirectory with a name starting with the `"fbd-"` prefix is considered a package directory if it contains at least one file with the `".fbd"` extension. If the name of the subdirectory matches exactly the `"fbd-"` pattern, then a compiler must report an error on an in valid package name.

3) The compiler must recursively check all subdirectories of the paths defined in the `FBDPATH` environment variable. The variable may contain multiple paths separated by the ':' (colon) character. Each subdirectory with a name starting with the `"fbd-"` prefix is considered a package directory if it contains at least one file with the `".fbd"` extension. If the name of the subdirectory matches exactly the `"fbd-"` pattern, then a compiler must report an error on an invalid package name.

Compilers are also free to have their own parameters allowing to provide extra paths to look for packages. The below snippet presents a tree of example working directory.

```
|-- externals
|   `-- bar
|       |-- fbd-bar
|       |   `-- bar.fbd
|       `-- gw
|           `-- bar.vhd
|-- fbd
|   |-- fbd-pkg1
|   |   `-- a.fbd
|   |-- not-a-pkg
|   |   `-- c.txt
|   `-- pkg2
|       `-- b.fbd
|-- gw
|   |-- modules
|   |   |-- a.vhd
|   |   `-- b.vhd
|   `-- top.vhd
`-- sw
    `-- foo.py
```

In this case each FBDL compilant compiler must automatically discover following three packages:

- `bar` - path `"./externals/bar/fbd-bar"`,
- `pkg1` - path `"./fbd/fbd-pkg1"`,
- `pkg2` - path `"./fbd/pkg2"`.

9.2. Scope rules

The following elements define a new scope in the FBDL:

- package,
- type definition,
- functionality instantiation.

The 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 instantiation and in the `Blk` block instantiation. 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 a def ault value that equals 16. This is because 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.

10. Grouping

Grouping is a feature of the FBDL used to inform a compiler that particular functionalities might be accessed together, and their register location must meet additional constraints. This is achieved using the `groups` property. The following functionalities can be grouped: `config`, `irq`, `mask`, `static`, `status`. A functionality may belong to multiple groups (except `irq`), and groups must be registerified in the order they appear in the group lists. The following snippet presents three grouped configs.

```
Main bus
    type cfg_t; width = 8; groups = ["group"]
    A cfg_t
    B cfg_t
    C cfg_t
```

Any FBDL compliant compiler must place all three configs (`A`, `B`, `C`) in the same register.

10.1. Single register groups

The single register groups are groups of elements that fit a single register. The overall width of all functionalities is not greater than the single register width. In such a case, all functionalities must be placed in the same register. The specification does not impose any specific order of the functionalities within the register, and it is left to the compiler implementation. The following listing presents an example bus description with three single register groups.

```
Main bus
    C0 config; width = 16; groups = ["read_write_group"]
    M0 mask;   width = 15; groups = ["read_write_group"]

    C1  config; width = 16; groups = ["mixed_group"]
    S11 static; width = 8;  groups = ["mixed_group"]
    S12 status; width = 8;  groups = ["mixed_group"]

    S21 status; width = 4; groups = ["read_only_group"]
    S22 status; width = 7; groups = ["read_only_group"]
```

All functionalities of the *"read_write_group"* can be both read and written. The code generated by a compiler for the requester must provide means for reading/writing the whole group as well as for reading/writing particular functionalities of the group.

The *"mixed_group"* contains functionality that can be read and written (`C1`), as well as functionalities that can only be read (`S11`, `S12`). The code generated by a compiler for the requester must provide a means for reading all readable functionalities and writing all writable functionalities. It is valid even if the group has single readable or single writable functionality. The compiler must also generate means for reading/writing particular functionalities of the group. In the case of *"mixed_group"* this will result in two means doing exactly the same (writing the `C1` config). However, it is up to the user to decide which of the means should be used. If it makes sense, it is perfectly valid to use both of them in different contexts.

All functionalities of the *"read_only_group"* are read-only. In this case, the compiler must generate a mean only for reading the group. It must also generate means for reading particular functionalities.

10.2. Multi register groups

The multi register groups are groups with functionalities that overall width is greater than the width of a single register. The specification does not impose any order of functionalities or registers in such cases, and it is left to the compiler implementation. However, the compiler must not split functionalities narrower or equal to the register width into multiple registers. This implies that any functionality with a width not greater than the register width is always read or written using single read or write access. The following snippet presents a bus description with one multi register group.

```
Main bus
    C  config; width = 10; groups = ["group"]
    M  mask;   width = 10; groups = ["group"]
    SC static; width = 10; groups = ["group"]
    SS status; width = 10; groups = ["group"]
```

The compiler must generate code for the requester allowing to write all writable functionalities of the group as well as the code allowing reading all readable functionalities of the groups. It must also generate means for reading or writing particular functionalities.

There are multiple ways to place functionalities from the above example into registers. The following snippet presents one possible way.

```
         Nth register                Nth + 1 register
-----------------------------  ----------------------
|| C | M | SC | 2 bits gap ||  || SS | 22 bits gap ||
-----------------------------  ----------------------
```

However, the above arrangement might not be optimal if there is a need to read both SC and SS at the same time as it would require reading two registers not a single one. The below listing presents how to group elements within the group using subgroups.

```
Main bus
    C  config; width = 10; groups = ["csubgroup", "group"]
    M  mask;   width = 10; groups = ["csubgroup", "group"]
    SC static; width = 10; groups = ["ssubgroup", "group"]
    SS status; width = 10; groups = ["ssubgroup", "group"]
```

The set of possible functionalities placements within the registers is now limited as the groups are registerified in the order they appear. The below snippet shows a possible arrangement.

```
        Nth register               Nth + 1 register
-------------------------  ---------------------------
|| C | M | 12 bits gap ||  || SC | SS | 12 bits gap ||
-------------------------  ---------------------------
```

This time reading both SC and SS requires reading only one register, while reading the whole *"group"* still requires reading two registers.

10.3. Array groups

The array groups are groups with all functionalities being arrays. The groups do not necessarily have the same number of elements.

The code generated by a compiler, for an array group, for the requester must provide a means for writing an arbitrary number of elements for all writable functionalities starting from an arbitrary index. It must also provide a mean for reading an arbitrary number of elements for all readable functionalities starting from an arbitrary index.

The specification does not define what happens on access to the elements with an index greater than the length of some arrays. This is because some of the target languages support special data types indicating that the value is absent (for example, `None` - Python, `Option` - Rust), while others use for this purpose completely valid values (`0` - C, Go).

10.3.1. Single register array groups

The single register array groups are array groups with overall single elements width not greater than the width of a single register. The below listing presents an example bus description with a single register array group.

```
Main bus
    type cfg_t config; width = 8; groups = "group"
```

```
    A [1]cfg_t
    B [2]cfg_t
    C [3]cfg_t
    D [3]status; width = 8; groups = "group"
```

In the case of a single register array group all elements with corresponding indices must be placed in the same register. Elements with consecutive indexes must be placed in consecutive registers. The below snippet presents a possible arrangement of elements for the example bus.

```
          Nth register
------------------------------
|| D[0] | C[0] | B[0] | A[0] ||
------------------------------
          Nth + 1 register
-----------------------------------
|| D[1] | C[1] | B[1] | 8 bits gap ||
-----------------------------------
          Nth + 2 register
------------------------------
|| D[2] | C[2] | 16 bits gap ||
------------------------------
```

10.3.2. Multi register array groups

The single register array groups are array groups with overall single elements width greater than the width of a single register. The below listing presents an example bus description with a multi register array group.

```
Main bus
type cfg_t config; groups = "group"
    A [1]cfg_t; width = 16
    B [2]cfg_t; width = 12
    C [2]cfg_t; width = 12
```

In the case of multi register array group all elements with corresponding indices must be placed in consecutive registers. Also all elements with consecutive indexes must be placed in consecutive registers. Such a requirement guarantees that block access can always be used. The below snippet presents possible arrangement of elements for the example bus.

```
          Nth register                Nth + 1 register
-----------------------------   ------------------------
|| C[0] | B[0] | 8 bits gap ||   || A[0] | 16 bits gap ||
-----------------------------   ------------------------
       Nth + 2 register                Nth + 3 register
-----------------------------   ------------------------------
|| C[1] | B[1] | 8 bits gap ||   || C[2] | B[2] | 8 bits gap ||
-----------------------------   ------------------------------
```

10.4. Mixed groups

The mixed groups are groups with both single functionalities and array functionalities. The below listing presents an example bus description with a mixed group.

```
Main bus
    C config; width = 10; groups = "group"
    M mask;   width = 7;  groups = "group"
    S status; width = 8;  groups = "group"
```

```
CA [3]config; width = 10; groups = "group"
SA [3]config; width = 12; groups = "group"
```

In case of mixed groups array functionalities shall be registerified as the first ones assuming a pure array group. Single functionalities shall be later placed in the gaps created during array registerification. If there are no gaps, or gaps are not wide enough, then all reamining single functionalities shall be registerified as single register group or multi register group. If the gaps are wide enough to place single functionalities there, but for some reason it is not desired, then subgroup can be defined to group single functionalities of the mixed group as the first ones. The below snippet presents a possible arrangement of elements for the example bus.

```
      Nth register                 Nth + 1 register
----------------------   -----------------------------------
|| CA[0] | SA[0] | C ||   || CA[1] | SA[1] | M | 3 bits gap ||
----------------------   -----------------------------------
          Nth + 2 register
-----------------------------------
|| CA[2] | SA[2] | S | 2 bits gap ||
-----------------------------------
```

10.5. Virtual groups

Virtual groups are groups that name starts with the underscore ('_'), for example "*_group*". Virtual groups are used to group functionalities without generating the group interface for the requester code.

10.6. Registerification order

Groups must be registerified in the order they appear in the groups lists. A compiler must issue an error if the order of any groups is not the same in all groups lists. If the order is not unequivocal, then the compiler is free to choose the order. However, as the registerification results have to be deterministic and reproducible for a particular compiler, the order criterion has to be fixed in case of ambiguous order of groups. The most natural criteria are probably:

- Alphabetical order. Groups with ambiguous order are sorted alphabetically before registerification.
- Occurrence order. Groups with ambiguous order are registerified in parsing order. For example, if the order of groups *"b"* and *"a"* is ambiguous, and group *"b"* first occurrence is in line number 80, and group *"a"* first occurrence is in line number 120, then group *"b"* is registerified as the first one.

The order of groups might be used to prioritize the groups, so that access to some groups is more efficient than to the other groups. The below listing serves as an example of groups order used for optimizing access to a particular group.

```
Main bus
    C1 config; width = 20; groups = ["a"]
    C2 config; width = 12; groups = ["a", "b"]
    C3 config; width = 20; groups = ["b"]
```

As group *"a"* has higher priority than group *"b"* (its index is lower in the groups list for functionality `C2`), access to the group *"a"* will be more efficient, as functionalities `C1` and `C2` will be placed in the same register. A possible arrangement is presented in the below snippet.

```
Nth register     Nth + 1 register
-------------  ----------------------
|| C1 | C2 ||  || C3 | 12 bits gap ||
-------------  ----------------------
```

If the order of the groups in the groups list for functionality `C2` was reverse, then the access to the group *"b"* would be more efficient. A possible arrangement of functionalities in such a case could look as follows.

```
Nth register     Nth + 1 register
-------------  ----------------------
|| C2 | C3 ||  || C1 | 12 bits gap ||
-------------  ----------------------
```

The below listing presents a description of groups with ambiguous order.

```
Main bus
    C1 config; width = 10; groups = ["a", "b", "c"]
    C2 config; width = 10; groups = ["a", "d", "c"]
    C3 config; width = 10; groups = ["a", "b"]
    C4 config; width = 10; groups = ["a", "d"]
```

The order of groups *"b"* and *"d"* is not unequivocal. However, whether group *"b"* is registerified before the group *"d"* is not even important in this case, as the optimal structure is determined by three facts:

- both groups *"b"* and *"d"* are subgroups of group *"a"*,
- the intersection of groups *"b"* and *"d"* is an empty group,
- both groups *"b"* and *"d"* have higher priority than group *"c"*.

Possible arrangement of the functionalities is presented in the below snippet.

```
        Nth register              Nth + 1 register
-------------------------  -------------------------
|| C1 | C3 | 2 bits gap ||  || C2 | C4 | 2 bits gap ||
-------------------------  -------------------------
```

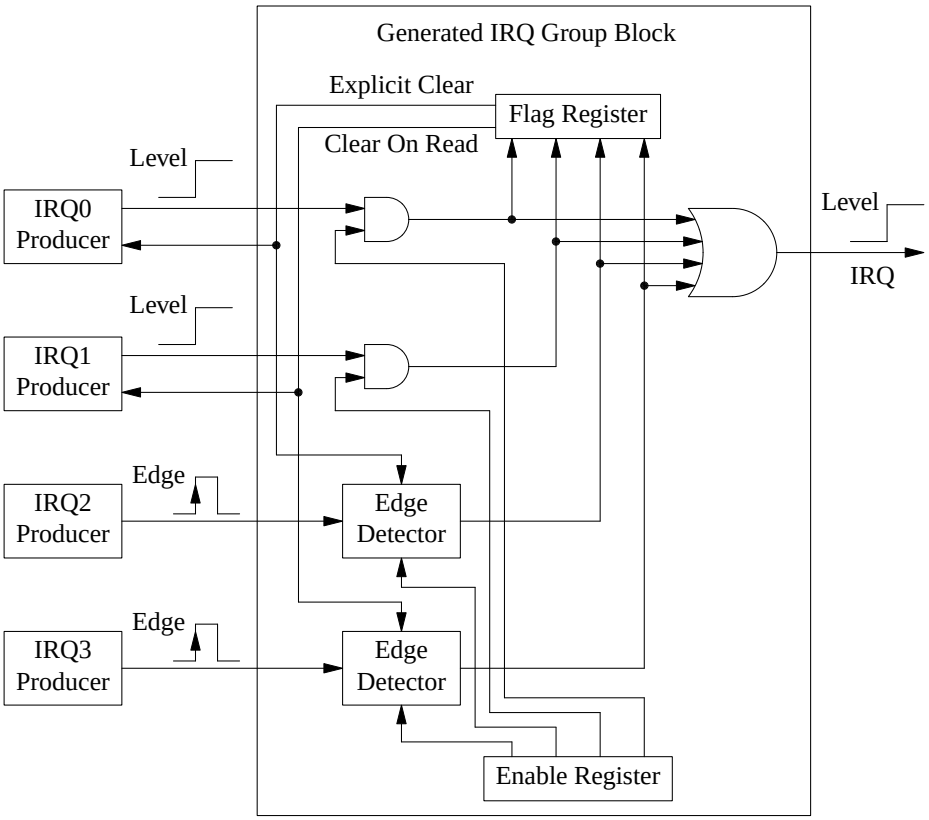
10.7. Irq groups

The irq groups are used for interrupt grouping. Grouped irqs have a common interrupt consumer signal. Each irq must belong at most to one group and each irq group must have at least two irqs. Irqs belonging to the same group might have different values of the producer trigger (`in-trigger`), but all of them must have the same value for the consumer trigger (`out-trigger`). In the case of level-triggered interrupt consumer the information on the interrupt source can be read from the interrupt group flag register.

The below snippet shows an example of an irq group for level-sensitive interrupt consumer.

```
Main bus
    type irq_t irq; add-enable = true; groups = "IRQ"
    IRQ0 irq_t
    IRQ1 irq_t; clear = "On Read"
    IRQ2 irq_t; in-trigger = "Edge"
    IRQ3 irq_t; in-trigger = "Edge"; clear = "On Read"
```

The picture below presents a possible logical block diagram of the irq group with level trigger for the interrupt consumer and enable register. The "Clear On Read" signal is driven on every Flag Register read. The "Explicit Clear" signal must be driven when the requester calls a means for clearing given interrupt flags. Probably the easiest form of the "Explicit Clear" implementation is clear on Flag Register write, where the clear bit-mask is the value of the data bus. The Flag Register is to some extent a virtual register, as it has an address, but it does not have any storage elements. The flag is stored in the interrupt producer in case of a level-triggered producer or in the Edge Detector in case of an edge-triggered producer.



10.8. Param and return groups

Param and return groups are used to group `proc` or `stream` parameters or returns. Such a kind of grouping may be necessary for performance optimizations, as the requester may store parameters or returns in a single list or in multiple distinct lists. Param and return groups help to avoid data reshuffling before or after the access. Param and return groups are independent. The below snippet presents a valid description with a single `proc` with one param and one return group.

```
Main bus
    P proc
        p1 param;  groups ="grp"
        p2 param;  groups ="grp"
        r1 return; groups = "grp"
        r2 return; groups = "grp"
```

Param and return groups may contain subgroups. Single param or return can belong to groups which sum is empty or is equal to one of the groups. The below snippet presents examples of two invalid and two valid parameters grouping.

```
Main bus
    # Param p2 belongs to group "b" and "c".
    # However, neither "b" is subgroup of "c"
    # nor "c" is subgroup of "b".
    Invalid1 proc
        p1 param; groups = ["a", "b"]
        p2 param; groups = ["a", "b", "c"]
        p3 param; groups = ["a", "c"]
```

```
Invalid2 proc
    p1 param; groups = "a"
    p2 param; groups = ["a", "b"]
    p3 param; groups = "b"

Valid1 proc
    p1 param; groups = "a"
    p2 param; groups = "a"
    p3 param; groups = "b"
    p4 param; groups = "b"

Valid2 proc
    p1 param; groups = ["a", "b", "c"]
    p2 param; groups = ["a", "b", "c"]
    p3 param; groups = ["a", "b"]
    p4 param; groups = "a"
```