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 2024

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 purpose of the data. 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 a single register width.

The thesis proposes a new approach, the functionality-centric approach. In the functionality-centric approach, the user defines the data with the type of its functionality. The registers and access code 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), presents an example of the advantages of the functionality-centric approach compared to the register-centric, and provides reasoning for some design decisions and some 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 kodem dostępowym. Definiowanie funkcjonalności danych pozwala na zwiększenie ilości kodu 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), opis korzyści wynikających z podejścia zorientowanego na funkcjonalność, uzasadnienie niektórych decyzji projektowych oraz omówienie niektórych ze szczegółów 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

AMD Advanced Micro Devices

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

BFM Bus Functional Model

CBM Compressed Baryonic Matter

CDC Clock Domain Crossing

CPU Central Processing Unit

CMS Compact Muon Solenoid

CSV Comma-Separated Values

DAQ Data Acquisition

DESY Deutsches Elektronen-Synchrotron

EDA Electronic Design Automation

EISA Extended Industry Standard Architecture

FBDL Functional Bus Description Language

FIFO First In, First Out

FPGA Field Programmable Gate Array

GPIO General-Purpose Input/Output

GUI Graphical User Interface

HDL Hardware Description Language

HEP High Energy Physics

HLS High Level Synthesis

HTML HyperText Markup Language

IBM International Business Machines Corporation

IO Input/Output

IP Intellectual Property / Internet Protocol

ISA Industry Standard Architecture

JSON JavaScript Object Notation

LAN Local Area Network

LSB Least Significant Bit

LUT Lookup Table

MCA Micro Channel Architecture

MCU Microcontroller Unit

MMIO Memory Mapped Input Output

NoC Network on Chip

OSD Open Software Description Data

PCI Peripheral Component Interconnect

PCIe Peripheral Component Interconnect Express

POSIX Portable Operating System Interface for UNIX

SCSI Small Computer Systems Interface

SLR Super Logic Region

SoC System on Chip

SPI Serial Peripheral Interface

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

XLS Excel Spreadsheet

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 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 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 members' ages 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 programming language change 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, each with a distinct user interface.

Throughout the work, it turned out that another policy is suited much better in such a diverse environment. 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 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 large amount of code was still repeatedly implemented manually. That manually implemented code had some common characteristics and could be easily automatically generated. The only thing that needed to be added to generate it was the information on the functionality that a given data must serve. 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 analyzing state-of-the-art tools and approaches, the author concluded that no solution is 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 10 chapters and 8 additional appendices. Appendix G 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 return to it while reading 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 paradigm shift 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 include aspects irrelevant from the thesis point of view, the chapter describes only the overall structure and focuses on some general details that any FBDL-compliant compiler will likely have to face.

Chapter 8 compares two descriptions of the same example system. One of the descriptions follows the register-centric approach, while the other follows the functionality-centric approach. Both descriptions have been tested using co-simulation testbenches. They have also been synthesized to compare resource utilization. The chapter contains multiple listings and waveforms presenting how the functionality-centric approach can decrease the probability of human mistakes and shorten the time required to implement the system.

Chapter 9 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 10 summarizes the advantages of describing system bus registers 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 primary role of the bus is to provide an organized and structured manner for connecting independent modules within the chip. It also serves as a gateway to access the gateware or hardware design internals 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 of a simplified structure of some SoC. Master modules are red, slave modules are yellow, and bus fabric components are blue.

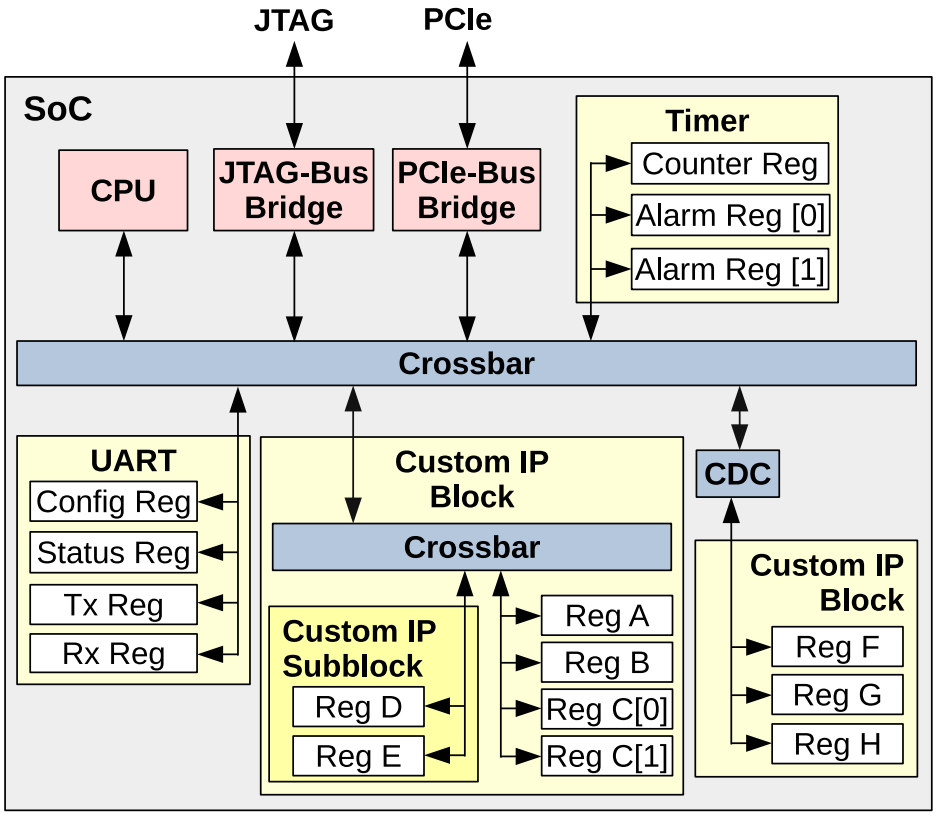


Figure 1.1: Example internal structure of some SoC design with bus.

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 bus and register management tasks can be done manually, semi-automated, or fully automated. The greater the automation, the less room for potential engineers' mistakes and the greater the 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, 10, 11]. 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 bus management's most time-consuming and error-prone part. What is more, when the register logic is not fully automatically generated, there is a need to verify the behavior of the registers. This is usually done in simulation by directed or randomized testbenches. However, [12] presents the benefits of doing register verification using formal methods, and [13] shows an example implementation of this idea.

1.1 Example problem

The following section introduces an example to ease the reasoning. The example also presents the typical register-centric approach for managing registers and the new functionality-centric approach proposed in the thesis. It presents some, but not all, problems encountered in a register-centric approach that are eliminated in the newly proposed approach.

Let us 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 a 48-bit 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-bit long vectors. The data might be passed to any set of workers. For simplicity, let us 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` and `_o` suffixes.

Inside an FPGA, is a 32-bit 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.

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

```vhdl
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 entity interface.

```python
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.

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 sense when 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 example Supervisor.

e) Calculate the number of bits required 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 requiring special handling. For example, in the case of the Supervisor, there is 48-bit 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.

It is 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 the register-centric approach.

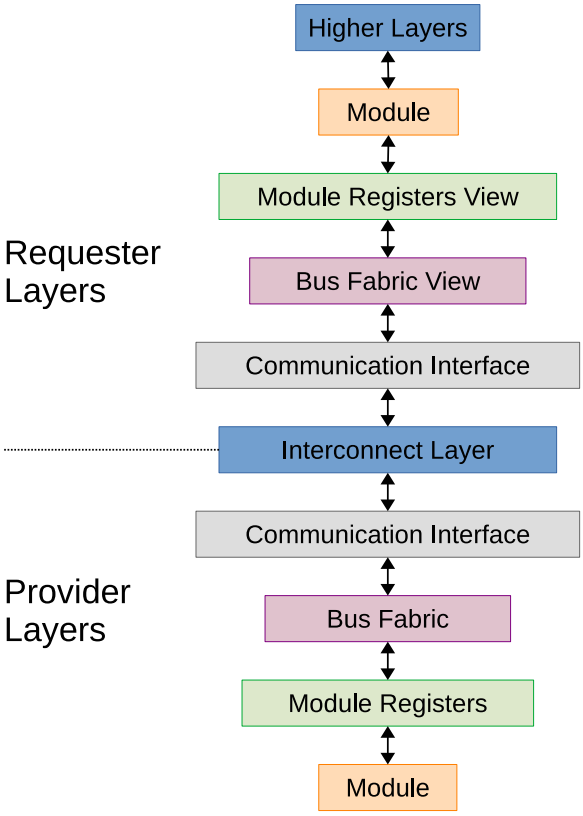


Figure 1.2: Conceptual stack of layers in the register-centric approach.

The communication interface and interconnect layers are irrelevant regarding the address space and register 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 the fact 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 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 the software source code.

Working manually on the register layout is also susceptible to changes. In the example Supervisor module, there are 96 bits needed for the control signals if `reset_counter`, `program`, and `unprogram` are implemented as strobe signals associated with given con-

trol registers. This is exactly three registers on a 32-bit 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.

```
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 for register-centric approach.

It all has to be coded manually. What is worse is that 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 stored data. This implies micro-management of registers, generates a lot of irrelevant questions, and is relatively susceptible to changes.

Listing 5 presents an example SystemRDL description for the example Supervisor. SystemRDL is the only formally defined register-centric format. If there were a need to increase the number of workers above the data 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 4 presents the file difference that would have to be applied in such a case.

```
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 4: Example Supervisor SystemRDL description change for worker count increase above the data bus width.

```
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 5: Example Supervisor SystemRDL description.

1.3 Functionality-centric approach

The thesis proposes a paradigm shift 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.

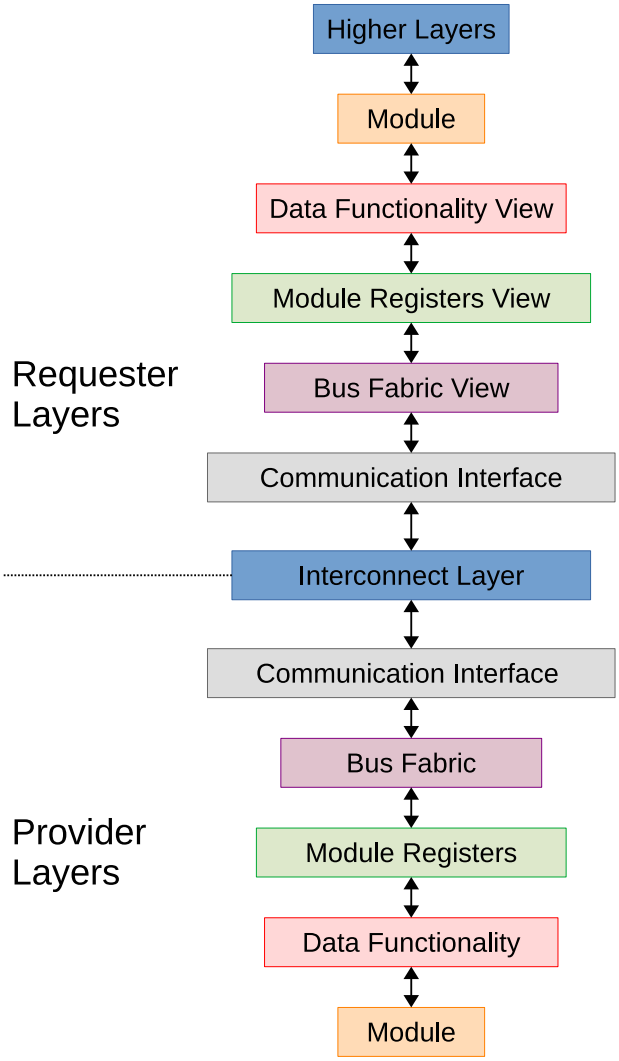


Figure 1.3: Conceptual stack of layers in the functionality-centric approach.

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 the example Supervisor, and appendix A presents registerification results. If there were 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 compiler carries out the registerification process, the whole register layout is automatically adjusted. There is no need to manually adapt gateware, firmware, or software code. As FBDL promotes safety by default, there is also no need to explicitly declare `Counter` status to be atomic. Any data wider than the data bus width has atomic access unless explicitly waived by the user.

```
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 worker 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 [14]. 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 [15], EISA [16], MCA [17], VESA [18], SCSI [19], USB [20], and PCI/PCIe [21]. 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 the placement of 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 interconnection. This is because they were designed to drive long signal traces and connector systems, 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 chip 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 supply [22]. 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 [23], 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 [24], Intel Avalon [25], STMicroelectronics STBus [26], Opencores Wishbone [5], MARBLE (asynchronous) [27].

The following 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 the Wishbone revision B4 specification has 128 pages and, the AMBA AXI specification is 273 pages long. 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 much 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 and functional IP cores with AXI interfaces.

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 the 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.

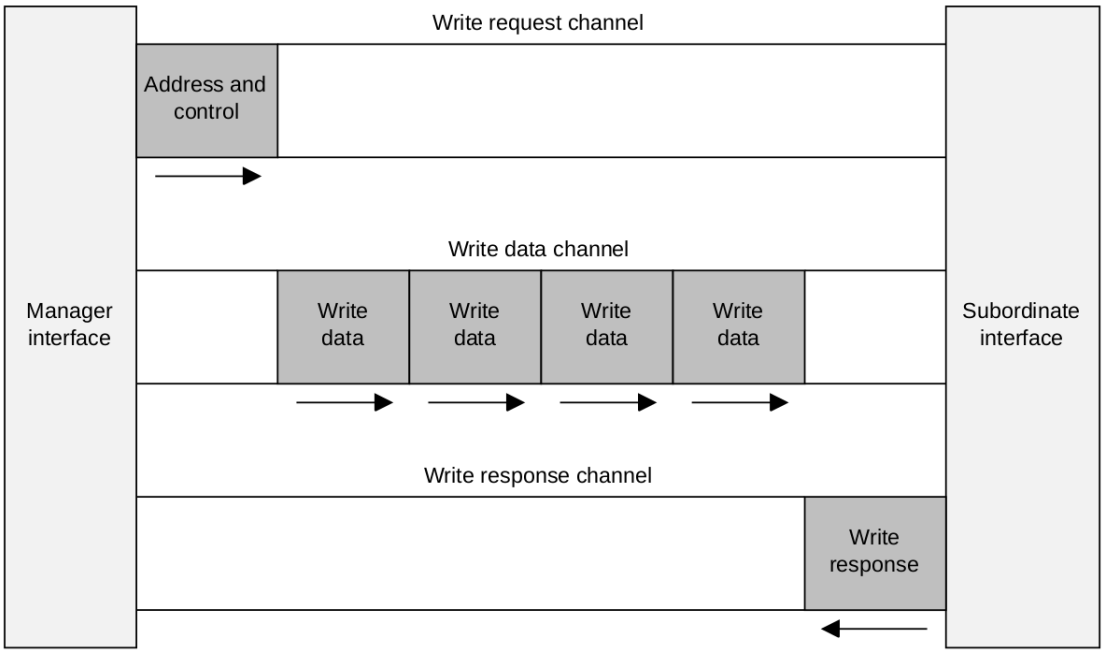


Figure 2.1: AXI channel architecture of writes [4].

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 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

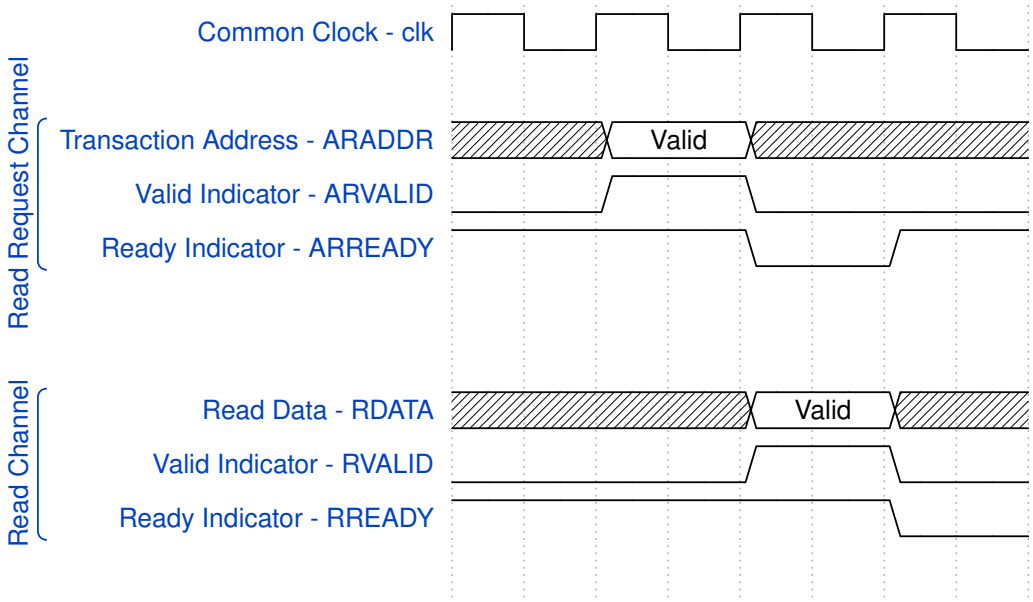


Figure 2.2: AXI single read transaction with single data transfer.

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 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 an AMBA AXI-Stream protocol defined in a separate specification [28]. 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 defined 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.

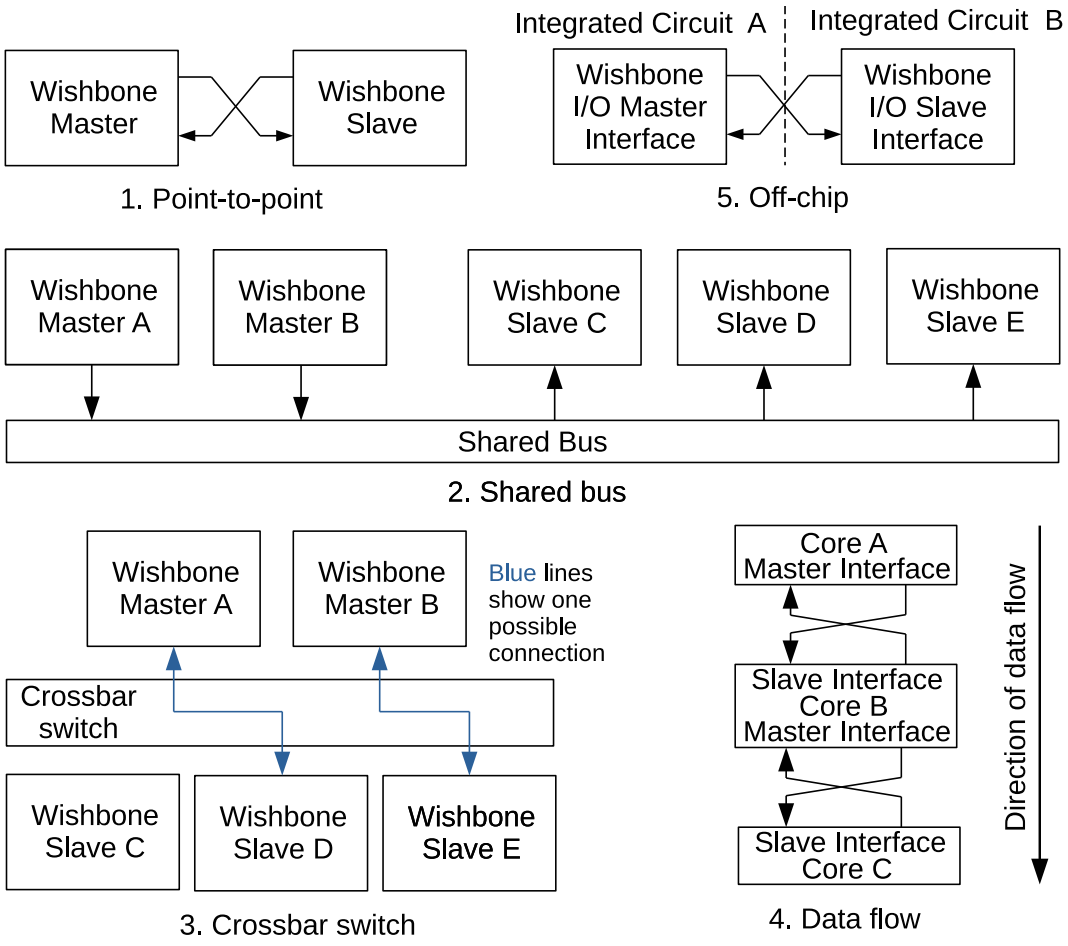


Figure 2.3: Possible Wishbone interconnections.

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

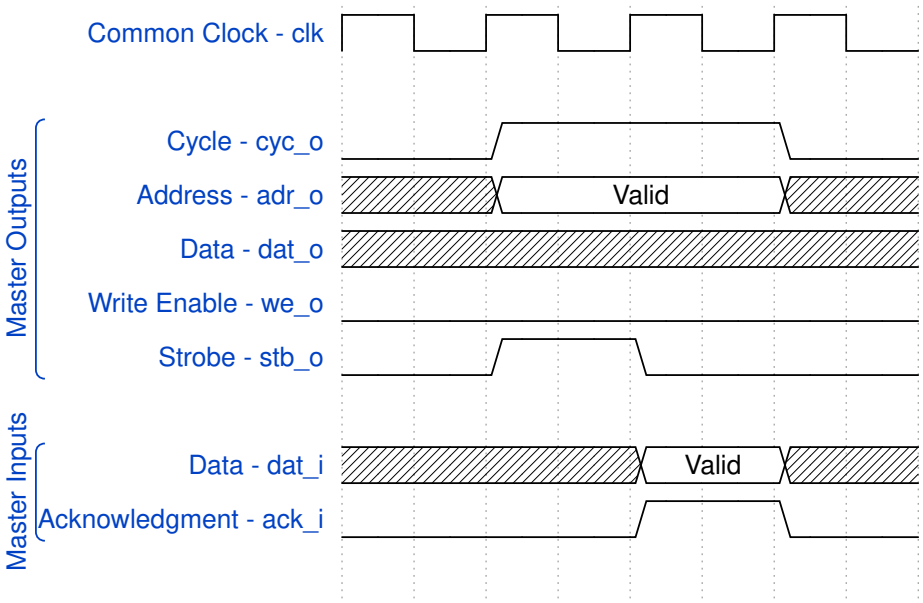


Figure 2.4: Wishbone classic standard single read transaction.

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 the data is valid. The slave may wait before asserting the acknowledgment signal 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.

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 [29, 30, 31, 32]. 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-bit AXI Lite master will simply not work with the 64-bit Wishbone slave in a traditional bus architecture without an extra bridge. In the NoC approach, each network node can have an individual interface for local register access.
4. Coupled transaction, transport, and physical activities. Changes to the bus physical implementation can have profound 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 high bandwidth performance critical heterogeneous SoC applications. Even homogeneous designs focused on accelerating the processing of gigabytes or terabytes of data (usually implemented using the 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 [33]. 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 [34, 35, 36, 37, 38, 39]. 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, and 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 implementing the protocols used in these networks, such as IP [40] or TCP [41], would consume a relatively large amount of resources and 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.

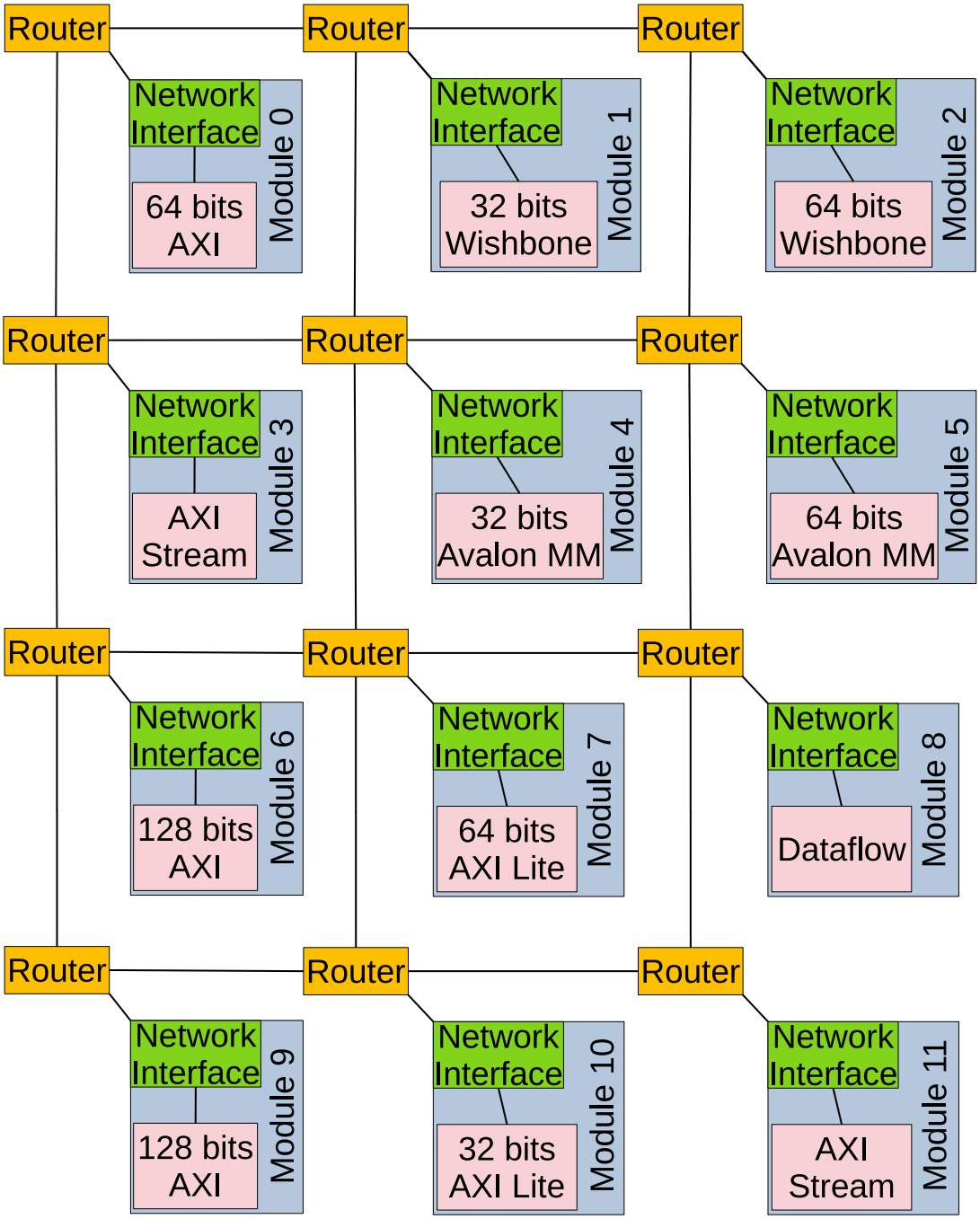


Figure 2.5: Example 12 nodes mesh network on chip.

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 frameworks. Each tool has been designed and implemented by different teams. Although their main goal is the same, they sometimes accentuate 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 order of analysis is alphabetical.

3.1 Existing tools

Register-centric approaches can be divided into two classes depending on the data they produce. The first class, as the output data, produces information on register addresses, masks, and bit shifts. The second class abstracts registers and bit fields as objects. The user does not explicitly use addresses, masks, and bit shifts but calls methods for reading and writing particular registers and bit fields. Instead of providing methods for reading and writing, some solutions prefer operators overloading, for example [42]. The second class is safer to use as it eliminates mistakes caused, for example, by applying bit shift of bit field A for bit field B.

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 full support, as all such tools are implemented incrementally. 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.

3.1.1 airhdl

The airhdl [43] 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 [44] format. It supports code generation for SystemVerilog, VHDL, C/C++, HTML [45] or Markdown documentaiton or transformation to IP-XACT XML [46] format. The tool is closed source, and any plan except the Free one is paid. The main website has a demo video upon which it is clear that the tool follows the register-centric approach. The user explicitly defines registers and bit fields. The generated C header file contains macros defining addresses, offsets, and masks.

3.1.2 Address Generator for Wishbone

The AGWB [3, 47], the successor of addr_gen [48], 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 register 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.

```
<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.

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 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. Which front-end ASICs receive the command depends on the values of

the `chip_address`, `downlink_mask`, and `group_mask`. None of the three 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.

Listing 9 shows the VHDL interface of the Software Command Slot entity. The `set_pending_i` port is connected to the strobe signal of the `control` register. The `clear_pending_i` port in the actual design is driven by the command consumer logic, but for co-simulation purposes, it was connected to the testbench register with an associated strobe signal. The definitions of `t_command` and `t_command_request` record types are not shown. However, all fields belonging to these types are presented in figures 3.1 and 3.2.

```
entity Software_Command_Slot is
   port (
      clk_40_i        : in std_logic;
      set_pending_i   : in std_logic;
      clear_pending_i : in std_logic;

      downlink_mask_i : in std_logic_vector(11 downto 0);
      group_mask_i    : in std_logic_vector(7 downto 0);
      command_i       : in t_command;

      command_request_o : out t_command_request := C_EMPTY_COMMAND_REQUEST
   );
end entity;
```

Listing 9: Software Command Slot VHDL entity interface.

Listing 10 shows the creation of write commands for Python co-simulation testbench. A single write command consists of two control frames. The first control frame contains the register address as the payload, and the second one contains data. The provided `sequence_number` is the sequence number of the first control frame, the second control frame within the command must have a sequence number increased by one compared to the first control frame.

```
write_commands = [
    Command(
        downlink_mask = 0x30,
        group_mask = 0x8,
        chip_address = 3,
        sequence_number = 0,
        request_types = (WRADDR, WRDATA),
        register_address = 0x4A,
        data = 0x31,
    ),
    Command(
        downlink_mask = 0x1D0,
        group_mask = 0xAB,
        chip_address = 0xF,
        sequence_number = 0xD,
        request_types = (WRADDR, WRDATA),
        register_address = 0xFF9,
        data = 0x75,
    ),
]
```

Listing 10: Snippet of Python code with write command objects creation for Software Command Slot.

Listing 11 presents the Python method with an invalid order of register writes. As the access order has to be implemented manually, it is relatively easy to write the `control` register before the `control_frame` registers by mistake. If the `control` register is written before `control_frame` registers, the system "almost works."

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

Listing 11: Python method sending command to the Software Command Slot - invalid write order.

Figure 3.1 presents waveforms for signals connected to the Software Command Slot entity ports in case of invalid write order. When the command request is marked as pending, the `typee`, `payload` and `crc` attributes of both control frames are not yet valid (waveforms). The behavior depends on when the command request consumer samples the data. If sampling happens after the `typee`, `payload`, and `crc` are updated, the system works correctly. However, if sampling happens before the update, then the system works incorrectly. The first command results in a 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. Such bugs can be complex 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 there must be a better way to describe data stored in the registers.

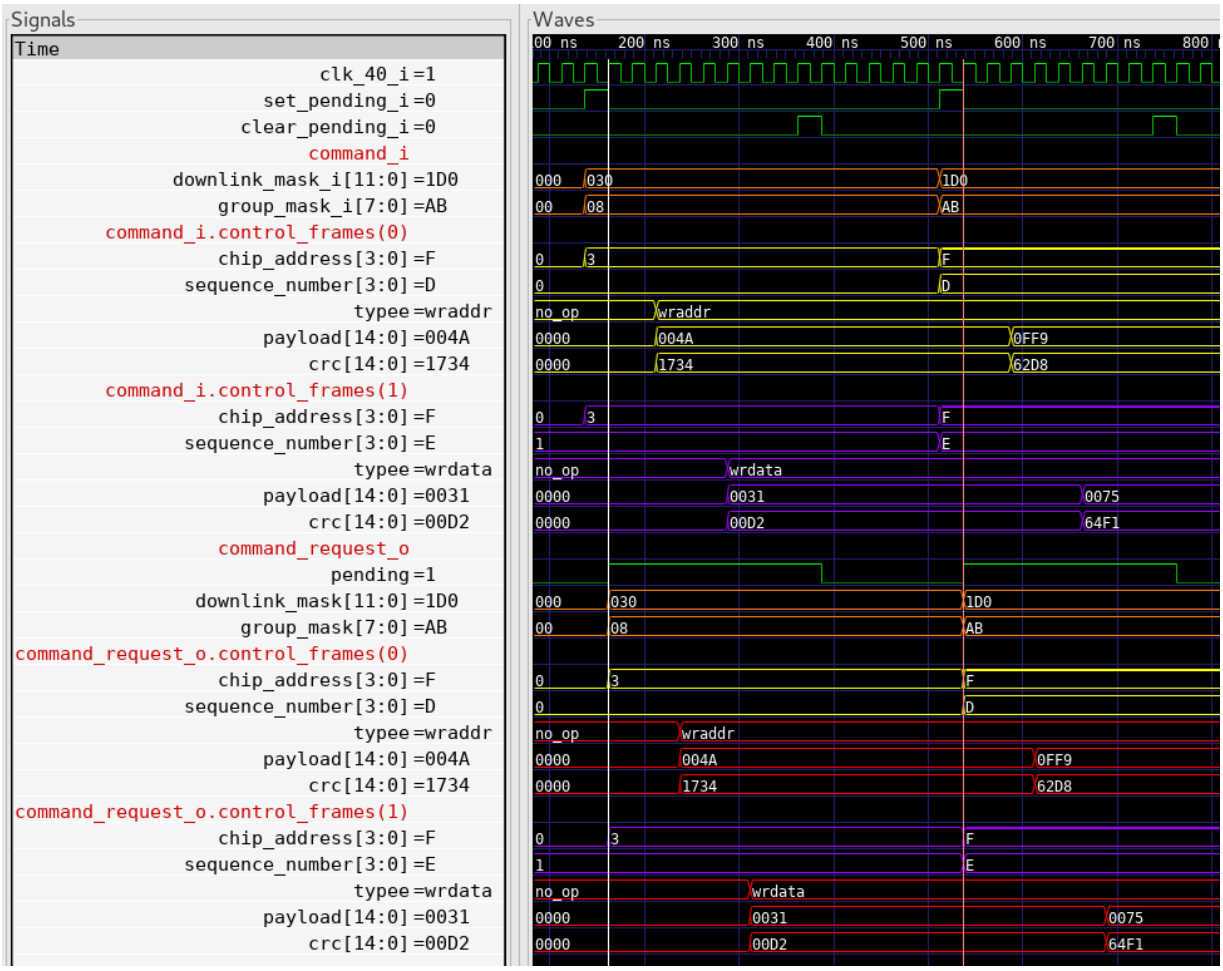


Figure 3.1: Software Command Slot entity port signal waveforms - invalid write order.

Listing 12 shows the Python method with a valid order of register writes, and figure 3.2 presents waveforms for signals connected to the Software Command Slot entity ports in case of valid write order. All attributes of both control frames are already valid when the command request pending signal is asserted. The result does not depend on the command consumer sampling time.

```
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 12: Python method sending command to the Software Command Slot - valid write order.

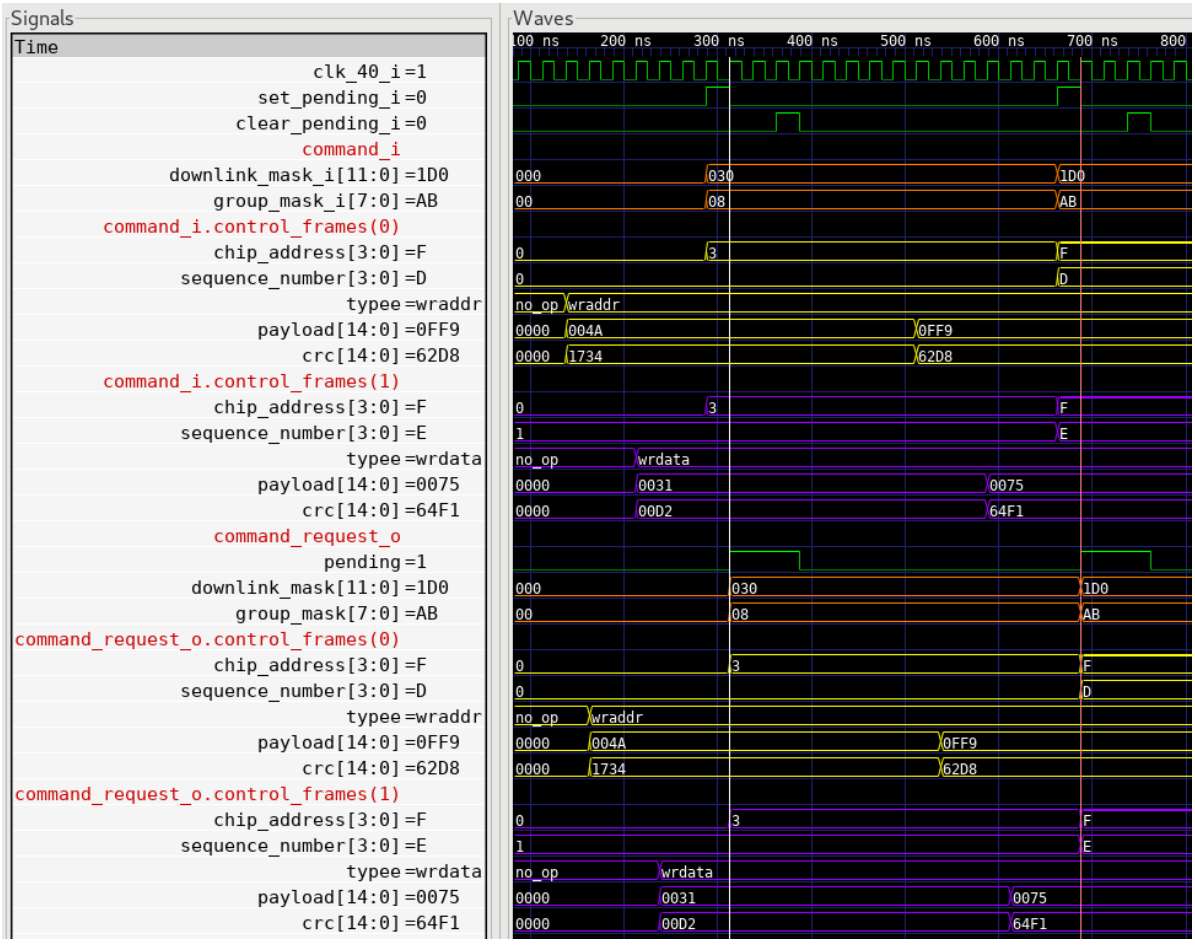


Figure 3.2: Software Command Slot entity port signal waveforms - valid write order.

3.1.3 AutoFPGA

AutoFPGA [49] 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 significant.

AutoFPGA files used for design generation contain much more information than register definitions. Listing 13 presents a snippet of the AutoFPGA documentation regarding the register description. Listing 14 presents a snippet regarding register macros of atuomatically generated `regdefs.h` file. This is a standard low-level register-centric approach. The user is provided with macros defining register addresses and must manually implement access methods.

```
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 13: AutoFPGA documentation on register 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 14: Snippet of `regdefs.h` file automatically generated by AutoFPGA.

3.1.4 Cheby

The Cheby [50, 51], the successor of the Cheburashka [52], aims to define 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 15 presents an example Cheby register description in YAML format. The user explicitly defines registers, providing their names, type, width, and access type. For example, register `inputs` represent 32 inputs of a GPIO. As inputs can only be read, the access type is defined as `ro` (read-only).

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 representative of a typical register-centric approach with abstracted access to a single register or bit field.

```yaml
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 15: Example Cheby registers description in YAML format.

3.1.5 Corsair

The Corsair [53] 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 16 presents an example Corsair register description in YAML format. Listing 17 presents the generated C header file. This is a traditional, register-centric approach. An engineer describes registers at the lowest level and 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.

```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 16: Example Corsair register description in YAML format.

```c
#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 17: Example C header file generated using Corsair (comments removed for brevity).

3.1.6 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 3.3 shows a simple system designed in Vivado Block Designer, containing blocks interconnected via the local AXI bus.

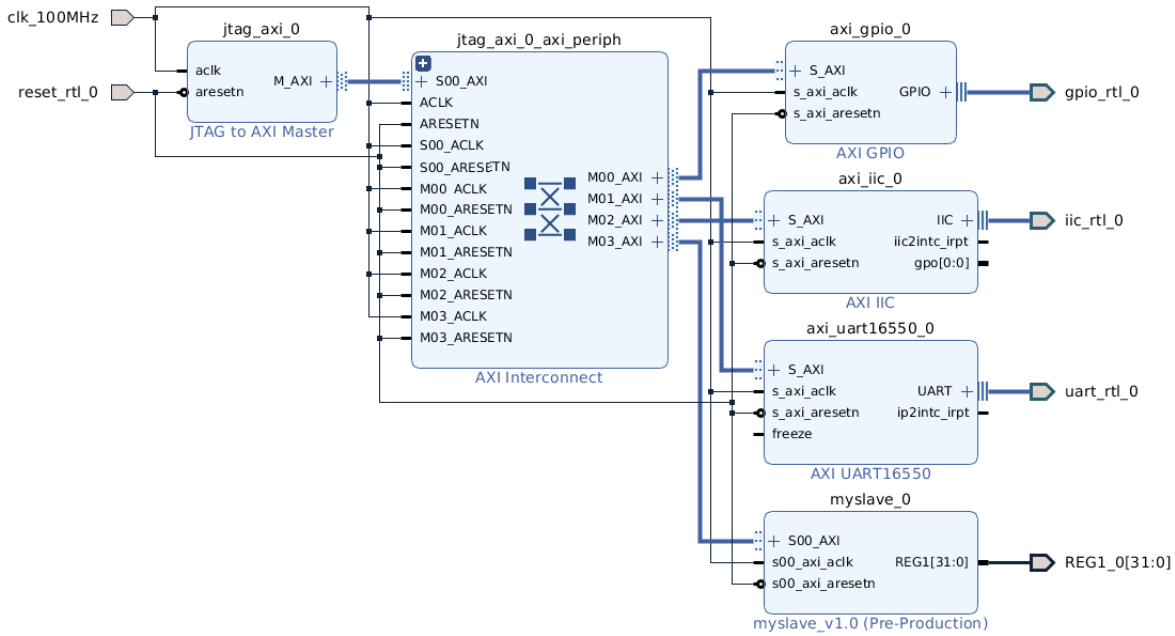


Figure 3.3: A simple design created using Block Designer in Xilinx Vivado environment [54].

Figure 3.4 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 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.

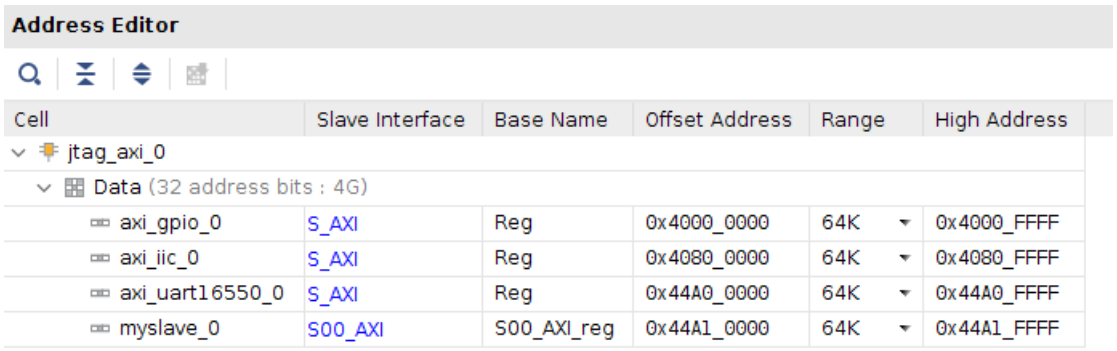


Figure 3.4: The address space allocation for the simple design from figure 3.3.

Unfortunately, managing designs using vendor EDA tools is challenging 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 in the design using a revision control system more complicated compared to the traditional approach in which configuration is done using textual files. An alternative approach is to use Tcl scripts for block design management [55]. While this approach eliminates the GUI approach disadvantages, it requires more user expertise.

3.1.7 hdl_registers

The hdl_registers [56] 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 18 presents an example register description in the TOML format used by the hdl_registers. The user explicitly defines registers and their mode, which is the access type. For example, the `configuration` register is defined with mode equal `r_w`. This implies that the register can be read and written. However, to store any data in the register, the user must also define bit fields belonging to the register. In the example description two bit fields are defined, `configuration.bit.enable` and `configuration.bit_vector.data_tag`.

Listing 19 presents the 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.

```toml
[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 18: Example hdl_registers description in TOML format.

Hdl_registers is also able to generate code with higher abstraction for C++ and Python. Listing 20 presents the generated C++ header file. The higher abstraction is achieved by generating getters and setters for registers and bit fields. There is 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?

```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 19: 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 20: Example C++ header file generated using hdl_registers (comments removed for brevity).

3.1.8 II & CII

The II (Internal Interface) [57] and CII (Component Internal Interface) are solutions developed for electronic systems created for CMS and DESY [58]. Although it is closed-source, its approach has been described in papers [59, 60]. The description in the papers does not allow for the reconstruction of the tool's internal logic. However, based on the attached figures and description, it looks like the 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.

3.1.9 IP-XACT

The IP-XACT [61] 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 is there" [62]. However, it is mentioned as prior art as there were at least two [63, 64] attempts to implement the bus and register code generators from the IP-XACT register 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 [65], are needed. These are usually GUI programs with a friendly user interface using IP-XACT XML file format as an input/output file format. A. Kamppi et al. proposed to extend IP-XACT with software features so that more firmware or software can be generated from the description [66].

3.1.10 Opentitan Register Tool

Opentitan [67, 68] is an open-source silicon Root of Trust project. As such, it has a subpart named the Register Tool [69] 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 21 presents an example Opentitan register description. The register is defined explicitly. The user must provide a register name, software access type (`swaccess`), and bit fields belonging to the register. The example description defines single register `REGA` with two bit fields `RXS` and `ENRXS`. As the software access type is `rw`, both bit fields can be read and written.

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

```
{
  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 21: Example Opentitan register description in Hjson format.

3.1.11 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 [70]. The presentation links are also valid [71, 72]. 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 22 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 addresses, masks, and bit-shift information.

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

Listing 22: Snippet from the Register Wizard documentation on defining registers and bit fields.

3.1.12 RgGen

RgGen [73] automatically generates source code related to configuration and status registers. 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) [74].

Listing 23 presents an example RgGen register description in YAML format. Listing 24 presents the 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.

```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 23: 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 24: Example C header file generated using RgGen.

3.1.13 SystemRDL

The SystemRDL [75] differs from all other available solutions as it is the only one with 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 [76, 77, 78] and open-source free SystemRDL compilers [79, 80, 81]. Listing 5 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.

3.1.14 vhdMMIO

VhdMMIO [82] 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 25 presents an example vhdMMIO register description in YAML format. The description defines a single control register with five bit fields (vhdMMIO uses the term *field* for the register and *subfield* for the field). In vhdMMIO, the user is required to provide addresses explicitly.

VhdMMIO is distinct from all other register-centric tools. This is because vhdMMIO has the concept of logical registers. Logical registers can be wider than data bus width and vhdMMIO is capable of generating atomic access hardware description. Unfortunately, atomic access is implemented in such a way that bus master must lock slave to access logical register sequentially and completely.

VhdMMIO also has the concept of registers/fields behavior. This allows generating more gateware description 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, such as procedures or streams. 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. Meanwhile, in FBDL, the 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 FBDL thought flow.

```yaml
metadata:
  mnemonic: SSP
  name: lpc1313_ssp
  doc: |
    This is mostly copypasted from the user manual of the SSP controller of
    the LPC1313 microcontroller (NXP UM10375) to serve as a real-world example
    of a register file description.
features:
  bus-width: 32
  optimize: yes
interface:
  flatten: yes
fields:
  - address: 0x0000
    register-name: CR0
    register-brief: SSP Control Register 0.
    register-doc: This register controls the basic operation of the SSP controller.
    behavior: control
    subfields:
      - bitrange: 3..0
        mnemonic: DSS
        name: data_size_select
      - bitrange: 5..4
        mnemonic: FRF
        name: frame_format
      - bitrange: 6
        mnemonic: CPOL
        name: clock_polarity
      - bitrange: 7
        mnemonic: CPHA
        name: clock_phase
      - bitrange: 15..8
        mnemonic: SCR
        name: prescaler_b
```

Listing 25: Example vhdMMIO register description in YAML format.

3.1.15 wbgen2

Wbgen2 [83] is one of the first open-source bus and register management tools. 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, blocks, or nested blocks. Listing 26 presents an example register description in the wbgen2-specific format. The generated C code contains information on register and field addresses and masks.

```
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 port 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 26: Example wbgen2 register description in wbgen2 specific format.

3.1.16 Others

Others also noticed the bus register management problem. For example, authors of [84] propose a custom text format for register description. The format defines register fields in one-to-one correspondence with those defined in the UVM register layer. Unfortunately, the tool is not publicly available, and the description of the tool is very short (2 pages). Authors of [85] propose to use Verilog attributes. The description of the approach is also very modest (2 pages). Both mentioned approaches are register-centric.

3.2 Summary

Table 3.1 summarizes capabilities of analyzed tools. The comparison table also includes FBDL just in case to satisfy the reader's curiosity.

Comparing bus and register management tool features is a challenging task. First, none of the register-centric tools, except SystemRDL, has formal specification. The implementation is the specification. What is more, most tools target only a limited set of hardware descriptions 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 [86] 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
Atomic Access Hardware Description	N	N	N	N	N	N	N	U	N	N	N	N	N	Y	N	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).

4 Dissertation

4.1 Thesis

It is possible to generate a hardware description of the bus register structure and software data access methods based on the description of the functionality of the data that shall be stored in the registers. Moreover, such an approach offers some advantages in certain practical 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 that allows the description of system bus registers by defining the functionality of the data. The work also includes the implementation of the proof of the concept compiler with a discussion of some general implementation details that any FBDL-compliant compiler will likely have to face and an example presenting the advantages of the functionality-centric approach in certain practical use cases.

5 Functionality types

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. In contrast, the dissertation focuses on answering the "why" questions and describing the benefits of the newly proposed functionality-centric approach.

5.1 Blackbox

The blackbox functionality is used to incorporate blocks implemented manually or generated by external tools (for example, the register-centric ones). The blackbox functionality has the following two rationales:

1. Entirely relying on the functionality-centric compiler's grouping algorithm may have disadvantages. Firstly, the compiler results might not always be optimal, for example, due to algorithm shortcomings. Secondly, automatic data grouping is challenging because of multiple constraints (described in subsection 7.1.3), so the algorithm might have bugs for some corner cases. There should be a way to bypass the automatic data placement in the registers to mitigate potential downsides.
2. A potential transition of a project already utilizing the register-centric approach to the FBDL would require extra work related to rewriting the register description to the data description, which might be time-consuming. The blackbox functionality allows for incorporating the hardware description generated by the register-centric tool and using the functionality-centric approach only for the new data.

5.2 Block

The block functionality is mainly used to logically group or encapsulate functionalities. The block concept is not unique to the 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 [87]. The SLRs might have different numbers of available resources and might be connected to different hardware IP blocks. Let us 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, it might need to be debugged. A side access channel is required for SLR0, hence it must have two master ports. Moreover, it must have some extra configuration and status data compared to the SLR1. Listing 27 presents how such requirements can be easily satisfied in FBDL using type parametrization and type extending mechanisms.

```
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 27: Example of type parametrization and type extending based on the block functionality.

5.3 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 that allows the change of 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 (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 target source files. It implies that the two most popular naming conventions (`camelCase`, `snake_case`) should be avoided for functionality instance names and constants 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 a proper field value. This would imply extra performance overhead and more complex code.

The remaining naming conventions starting with uppercase letters 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 hard to read, especially after several hours of sitting in front of the computer screen.

The FBDL bus description is bus-type agnostic. This implies that the actual bus type depends on the compiler support or, more precisely, on the generators provided by the compiler's back-end. However, a single compiler might support multiple bus types. Another implication is the fact that a bus description might contain only generic, common features such as:

1. number of masters (`masters` property),
2. bus reset type (`reset` property),
3. bus data width (`width` property),
4. the relative position of bus modules (`blackbox` and `block` functionalities).

Any bus-type specific parameter must be handled at the compiler level, for example, as a command line argument. Such an approach avoids unnecessary complexity in the language and compiler implementations. The language has fewer elements, and bus-type-specific compilers do not have to deal with extra logic that does not make sense for this particular bus type.

A bus description also does not contain information on the bus address width. The minimal bus address width (AW_{min}) implies from the address space size obtained as a registerification result. However, in the actual design, the bus address width always has some maximum value (AW_{max}). The following three scenarios are possible:

1. ($AW_{max} > AW_{min}$) In such a case, the designer has two options. The first one is to leave the upper bits of the address bus unconnected. The second one is to narrow the address bus so that $AW_{max} = AW_{min}$.
2. ($AW_{max} = AW_{min}$) In such a case, no action is required.

3. ($AW_{max} < AW_{min}$) In such a case, the designer has to increase the number of address bus bits. However, increasing the address bus width is not always possible or might require an unacceptable amount of time. An alternative approach is to limit the amount of data so that $AW_{max} = AW_{min}$.

The `bus` functionality has no property allowing to set the base address. However, FBDL-compliant compilers are allowed to accept the bus base address as a command line argument.

5.4 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 28 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 29 shows an example description with a single config with a width equal to the register width in the FBDL. Listings 30 and 31 present example code writing the config. In the case of config width not greater than the register width, the code is the same for the register-centric approach and FBDL.

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

Listing 28: Example config instantiation with width equal to the register width in the RgGen.

```
Main bus
    C config
```

Listing 29: Example config instantiation with width equal to the register width in the FBDL.

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

Listing 30: Example config write utilizing the code generated by the register-centric approach compiler.

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

Listing 31: Example config write utilizing the code generated by the FBDL compiler.

Listing 32 shows an example description with a single config with a width greater than the register width in the RgGen. Listing 33 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 31 is still valid. However, the register-centric approach code needs adjustments as an extra register has been added. Listing 34 presents adjusted code. It takes extra time to write the code, and there is room for possible mistakes. Firstly, the masks and shifts must be manually applied to the value. 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 on the HDL side. None of these is an issue in the FBDL, as everything is handled automatically by the compiler. This results from looking at the config as a functionality, not 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 32: Example config instantiation with width greater than the register width in the RgGen.

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

Listing 33: Example config instantiation with width greater than the register width in the FBDL.

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

5.5 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.
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 on the provider side by properly connecting interrupt components. There is also no way to configure whether the high or low level or a rising or falling edge triggers an interrupt. 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 complicate the language and create a second way to solve the same problem.

5.6 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 D (class `MaskSingleOneReg`) presents a code that can be automatically bound to the data solely based on the distinct type for mask.

5.7 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 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.

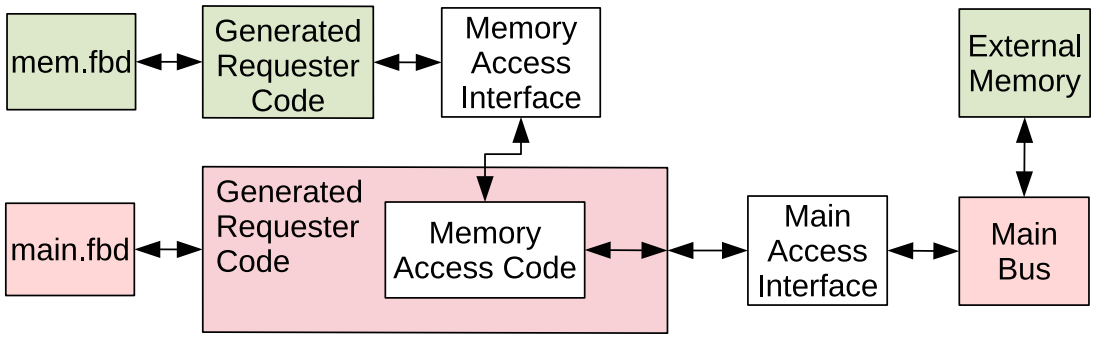


Figure 5.1: A possible access path to the external memory with separate FBDL description.

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 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 35, 36, 37, 38, and 39 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 35: 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 36: 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 37: FBDL external memory connection using two proc functionalities.

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

Listing 38: 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 39: FBDL external memory connection using two stream functionalities with separate address in downstream.

5.8 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 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. The user can always implement wrapper functions in the target language.

5.9 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 for presenting how the functional view of the data can significantly reduce the amount of manual work and increase the code robustness [88]. 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 12 presents Python code that had to be coded manually. Section 3.1.2 describes what is not optimal in the register-centric approach in this case. Listing 40 presents a description of the same block in FBDL format. Based on the description, it is already clear that the inferred registers will be used for the procedure call.

```
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 40: HCTSP software command slot block description in FBDL format.

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

5.11 Static

The static functionality represents data placed at the provider side that never changes. 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 type.

The static functionality may be used, for example, for versioning, bus id, bus generation timestamp, or storing secrets that shall be read only once. The typical difference between an id and a version is worth analyzing. 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's 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 id automatically. However, at least bus id is extremely useful in practice. It can be used to ensure that both requester and provider utilize the compilation results of the same bus description. Register with such id must be placed at a fixed, known address, usually at the beginning or the end of the generated address space.

5.12 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 5.4) 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, the requester is the only writer, whereas in the case of the status, the provider is the only writer.

5.13 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 Language 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 Two-writer data

Two-writer data (the term derived from the "two-writer register" term [89]) can be written by both the requester and the provider (FBDL specification nomenclature). In practice, both can write data, 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 the software and hardware access properties. In the FBDL, there is no functionality that would end up as 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 to designing and debugging systems with multiple data writers. As the resulting increase in resource utilization is relatively small (the number of required flip-flops is the same, only extra logic related to 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 resource utilization. However, it does not increase the address space size.

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

The one-writer 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 41 and 42 show examples.

Config `C` and status `S` occupy precisely 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 triggering. Status `ST` occupies the 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 overhead. The required address space size equals 1.

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

Listing 41: Example of `config` and `status` that can share register address.

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

Listing 42: 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 into other programming or hardware description languages. However, those other languages do not share a common definition of the enumeration type. Let us analyze three currently prevalent 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 add compiler back-ends for target languages easily. Extending FBDL with features peculiar to any target language or a subset of target languages is against this rule. Usually, when speaking about enumeration type in the context of register management, a set of constant possible values is meant This is already achievable in FBDL using constant definitions, listing 43. As constants are bound to a

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

```
# 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 43: Constraining value range using constants or `range` property.

The second issue with the enumeration type is the synchronization of enumeration type values. This is a more general issue, not related only to the register management tools. Let us suppose the 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).

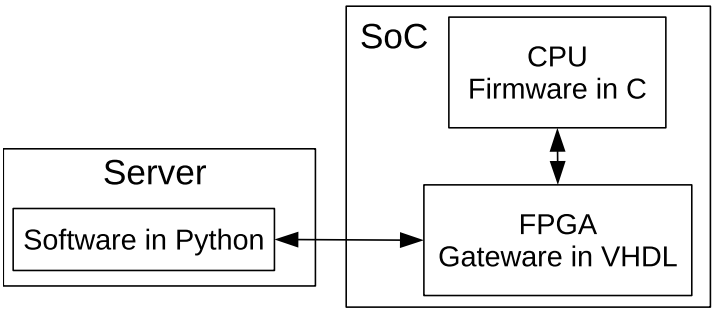


Figure 6.1: Example system with enumeration types synchronization issue.

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 an internal dependency on the FBDL output inside the 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, the gateware module unit testbench requires type generation and can no longer 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 of 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 different languages are often used for prototyping and final implementation

   (b) Enumeration type definitions are derived from the gateware/hardware description. The hardware description language, once chosen, rarely changes during the project.

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

3. Enumeration type definitions are implemented manually for all languages. However, a tool (some sanitizer) is 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.

While working on numerous projects, the author has encountered 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 expression evaluation. This is possible with all tools providing programming language API for description definition, as in this case, all programming language features are "inherited" and can be used without any limitations. This is a very flexible mechanism, but it sometimes leads to abuses. The bus/register management tool starts to be used as a general-purpose design configuration tool storing information unrelated to the bus or registers. The FBDL's goal is to be a bus and register management tool, nothing less, nothing more. However, the FBDL contains built-in functions (listed in the specification) frequently used for bus or register-related calculations.

6.4 Manual addressing

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

The FBDL does not allow manual addressing because of two reasons. The first one is that in FBDL, the 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 a case could be the start address of the data. However, manual addressing does not fit into the FBDL paradigm. The second reason is that any decent compiler should automatically insert a bus identification number at some fixed address. Placing single data with a unique value at a fixed address is enough to identify an address map unambiguously. Based on this information, the firmware or software can load the appropriate address map code and access any data, for example, block version, 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 a 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 syntax and terminology. The term “attributes” will be used for custom properties if 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 44 presents an example.

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

Listing 44: Syntax reserved for custom attributes.

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

7 Compiler 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 [90] and the back-end [91], both of which 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, any compiler must carry out the parsing, instantiating, and registerification phases. 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 an entirely different code structure than, for example, a 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 were monolithic and released with any restrictive license, such as GPL-3.0, it would not be possible to incorporate it directly into proprietary, closed-source programs. If the compiler were monolithic and released with any permissive license, for example, MIT, then anyone could take it as is and fix bugs or implement improvements without 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 the front-end as an external program and dump the registerification result into a JSON file. The back-end can then read the JSON file.

The compiler front-end supports all functionality types described in chapter 5. The compiler back-end supports all functionality types used in chapter 8.

Figure 7.1 presents the implemented compiler's current (at the time of dissertation writing) structure. 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. Moreover, if code for a given target is placed in multiple files, then the files are also generated in parallel.

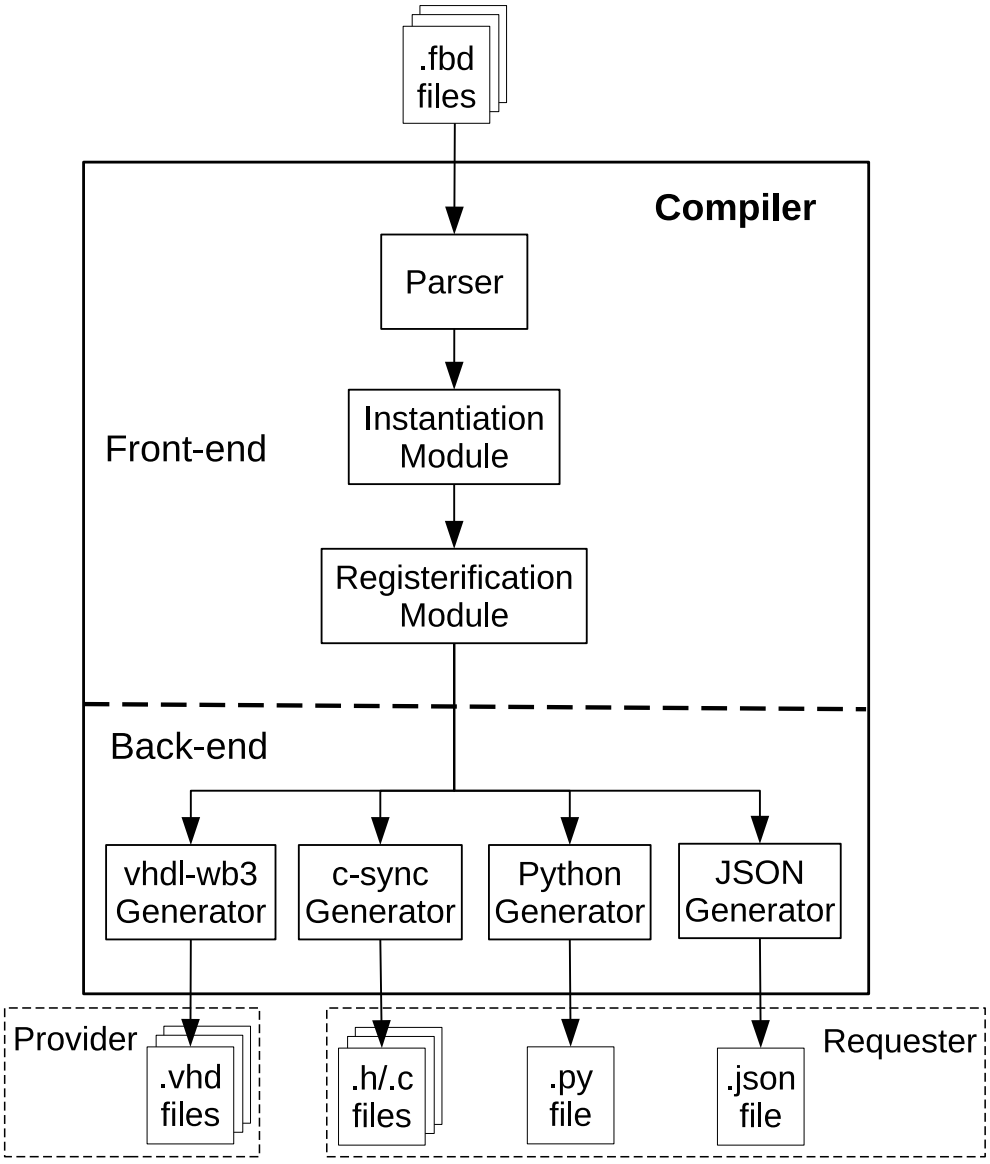


Figure 7.1: Current structure of the implemented compiler.

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 Description file 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, running this stage in parallel (true parallelism) is easy. The parser has been generated using the tree-sitter tool [92]. The defined grammar is available online [93]. Tree-sitter is a parser generator tool based on the GLR [94] parsing algorithm working most efficiently with a class of context-free grammars. It allows for rapid prototyping, but it is not free of drawbacks. The main one is error handling. Suppose the syntax provided by the user is not valid. In that case, giving informative feedback to a user on what exactly is wrong requires relatively more work than a hand-written custom parser, or sometimes is even impossible.

7.1.2 Functionality 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 more complex to implement. Moreover, 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 sequentially implements the instantiation stage. The whole compilation process is relatively short. A bus with up to 40 functionalities takes less than 10 ms to compile (front-end and 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 Functionality 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 complex to implement 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 the specification does not forbid such behavior, it is highly impractical. What is more, the registerification stage has a sequential nature. To optimize generated address space size, functionalities (if possible) must be put 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 six access types:

1. Single One Reg
2. Single N Regs
3. Array One In Reg
4. Array N Regs
5. Array N In Reg
6. Array N In Reg M In End Reg

The Single One Reg access type is the simplest access type used for single data fitting a single register. Listing 45 presents a description with three data of Single One Reg access type, and figure 7.2 presents an example register layout. The Single One Reg access type requires address and mask (start bit and end bit) attributes to describe unambiguously how to access the data.

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

Listing 45: Example bus with three data of Single One Reg access type.

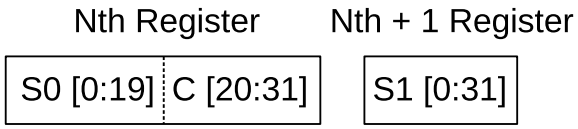


Figure 7.2: Example register layout of data of Single One Reg access type.

The Single N Regs access type is used to describe the access to data spanning multiple adjacent registers. Listing 46 presents a description with two data of Single N Regs access type, and figure 7.3 presents an example register layout. The Single N Regs access type requires start address, start bit, and width attributes to describe unambiguously how to access the data.

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

Listing 46: Example bus with two data of Single N Regs access type.

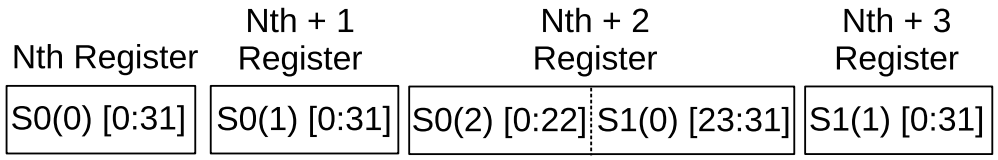


Figure 7.3: Example register layout of data of Single N Regs access type.

The Array One In Reg access type is used to describe access to array data with a single element placed within a single register. Listing 47 presents a description with one array data of Array One In Reg access type, and figure 7.4 presents an example register layout. The Array One In Reg access type requires start address, mask (start bit and end bit), and elements count to describe unambiguously how to access the data.

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

Listing 47: Example bus with one data of Array One In Reg access type.

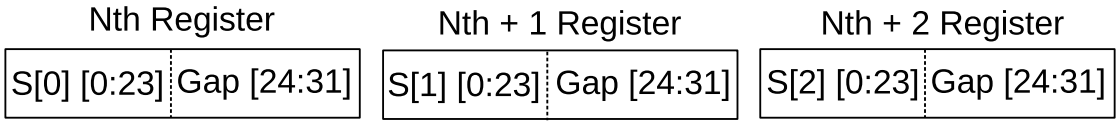


Figure 7.4: Example register layout of data of Array One In Reg access type.

The Array N Regs 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 48 presents a description with two array data of Array N Regs 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 N Regs access type requires start address, start bit, and elements count to describe unambiguously how to access the data.

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

Listing 48: Example bus with two data of Array N Regs access type.

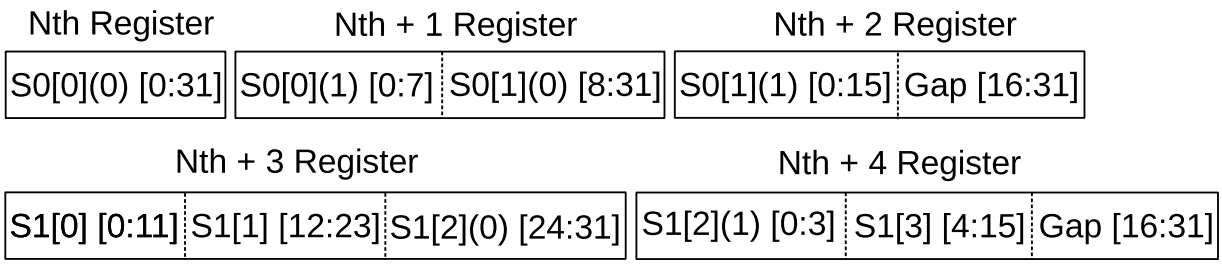


Figure 7.5: Example register layout of data of Array N Regs access type.

The Array N In Reg access type is used to describe access to array data with multiple elements placed in one register and with all registers having the same number of items. Listing 49 presents a description with two array data of Array N In Reg 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 four-element array data with single element width not being the divisor of the register width. The Array N In Reg access type requires start address, start bit, element width, and elements count to describe unambiguously how to access the data.

```
Main bus
    S0 [6]status; width = 16
    S1 [4]status; width = 14
```

Listing 49: Example bus with two data of Array N In Reg access type.

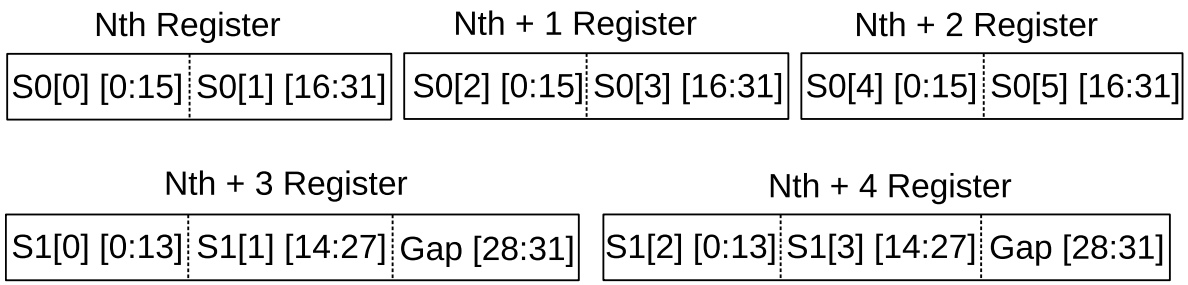


Figure 7.6: Example register layout of data of Array N In Reg access type.

The Array N In Reg M In End Reg access type is used to describe access to array data with multiple elements placed in one register and with all registers having the same number of items except the last one. Listing 50 presents a description with two array data of Array N In Reg M In End Reg access type, and figure 7.7 presents an example register layout. S0 is five-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 N In Reg M In End Reg access type requires start address, start bit, element width, and elements count to describe unambiguously how to access the data.

```
Main bus
    S0 [5]status; width = 16
    S1 [5]status; width = 10
```

Listing 50: Example bus with two data of Array N In Reg M In End Reg access type.

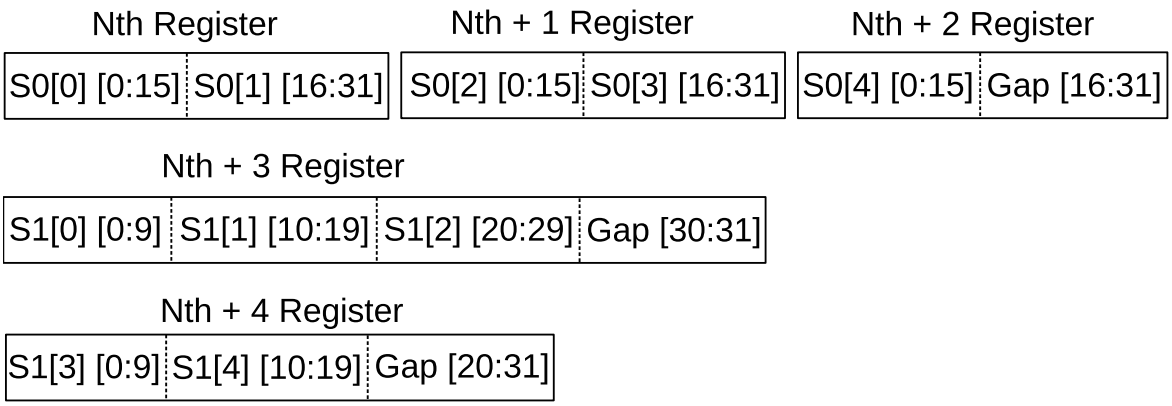


Figure 7.7: Example register layout of data of Array N In Reg M In End Reg access type.

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 that are configurable, for example, via a 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 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 ($N = ceil(functionality\ width/data\ bus\ width)$), 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, let us consider bus description from listing 51.

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

Listing 51: Bus description presenting sorting effect on registerification result.

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 required. This is shown in figure 7.8.

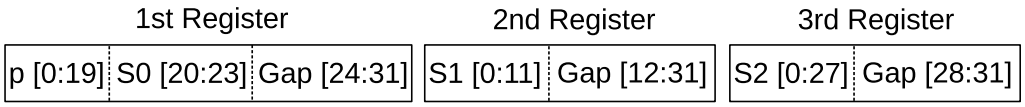


Figure 7.8: Register layout without functionality sorting.

However, if functionalities are first sorted in width decreasing order, then only 2 registers are needed. This is shown in figure 7.9.

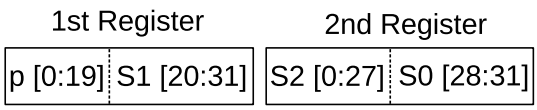


Figure 7.9: Register layout with functionality sorting.

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 us consider bus description from listing 52.

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

Listing 52: Bus description presenting registerification order change.

If statuses are registerified before configs, then 3 registers are required. This is shown in figure 7.10.

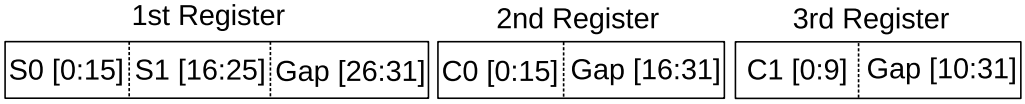


Figure 7.10: Register layout for status -> config order.

If configs are registerified before statuses, then 2 registers are required. This is shown in figure 7.11.

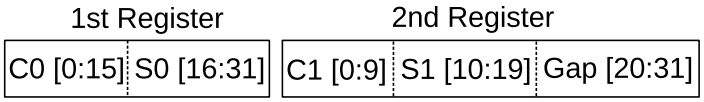


Figure 7.11: Register layout for config -> status order.

Address allocation

The FBDL specification does not enforce any particular addressing mode or address allocation algorithm. It is left to the compiler implementation. In contrast, SystemRDL formally defines three addressing modes: `compact`, `regalign`, and `fullalign`. A SystemRDL-compliant compiler should provide support for all of them. An FBDL-compliant compiler can be implemented with a fixed addressing mode, which eases the compiler implementation.

It is commonly known that assigning subsequent addresses to the registers and blocks without proper alignment results in suboptimal address decoders. This increases resource

utilization and the critical path length, lowering the maximum bus clock frequency. Therefore, the implemented proof of the concept compiler uses an address allocation algorithm oriented on the optimization of address decoders.

If a block requires B bits for internal addressing, then its overall address space is aligned to the 2^B boundary. That ensures that the access to the block may be easily decoded by a simple binary comparison of address bits B to $(address\ bus\ width - 1)$. This task can be easily performed by bus crossbars to which the blocks are connected.

The address allocation algorithm also minimizes the occupied address space by avoiding unnecessary fragmentation. This is achieved by applying the following rules:

1. For each block, the size of the required minimal block address space M is calculated as the sum of the required addresses for local registers and the address space size required for subblocks. The block address space size is rounded up to the nearest power of two: $S = 2^N$, where N is the smallest integer for which $2^N \geq M$. The S value is the size of the block address space. This step is performed recurrently, as the sizes of subblock address spaces are required to calculate the size of the parent block address space.
2. The local block registers are located at the beginning of the address space in each block. The subblocks are sorted according to their decreasing size and are placed starting from the end of the block's address space.
3. The final address map is built starting from the top block (`Main` bus) located at address 0 and traversing its subblocks.

The described address allocation procedure corresponds to the `regalign` addressing mode from the SystemRDL specification. It also explains why there is unused address space in the register map in Appendix C.

7.2 Back-end

The compiler's back-end is responsible for generating code for a particular target. It must generate the means required to access the functionalities. There is no inter-dependency between code generated for different targets, so 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 synchronously or asynchronously. 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. However, it can be beneficial when working with multiple versions of the same description or 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 7.12 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 that automatically generated code must be connected to the access interface. The compiler can also generate the access interface code, but it is not recommended for 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.

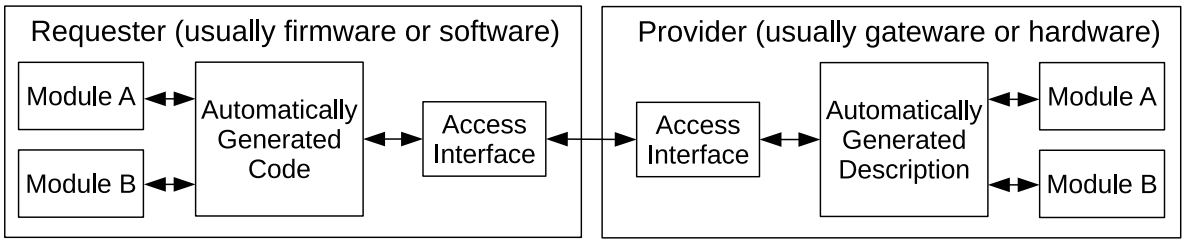


Figure 7.12: Simplified connection scheme of a system utilizing FBDL.

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, extending this naming convention further makes sense. 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 a 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 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 wholly 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 7.13 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.

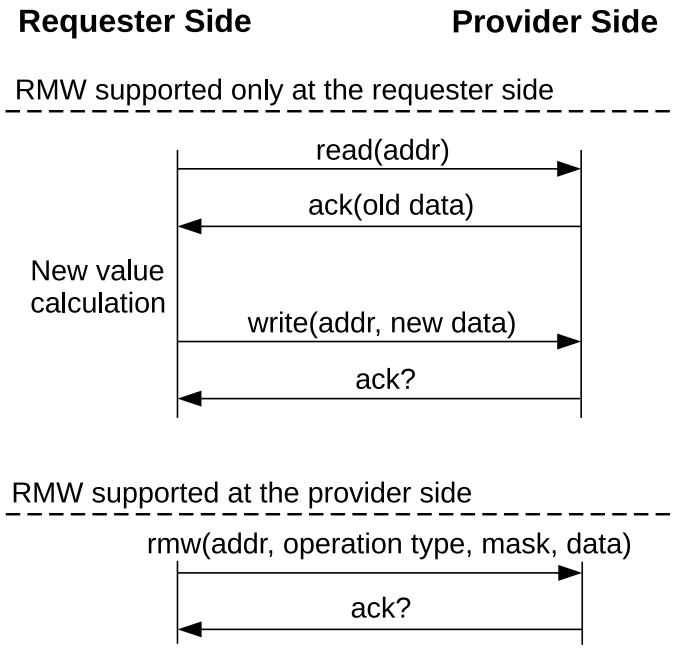


Figure 7.13: Sequence diagrams for `rmw` transaction without and with provider support for `rmw`.

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 primary 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 53 presents an example RMW procedure described in FBDL. In actual use cases, the widths of parameters depend on actual bus architecture. All remaining functionalities have been removed for brevity.

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

Listing 53: Example read-modify-write FBDL procedure.

Figure 7.14 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.

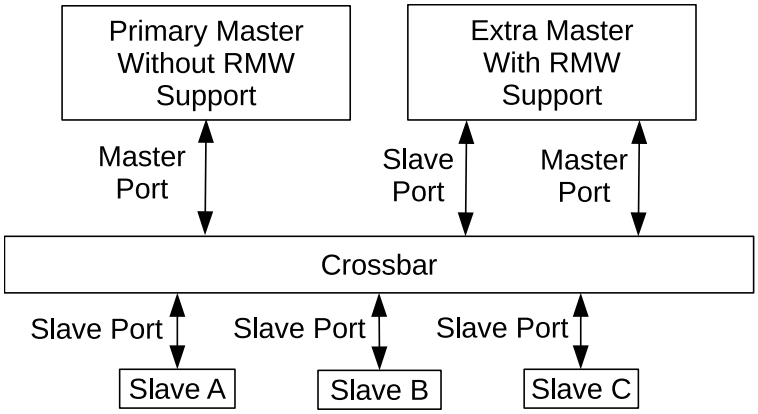


Figure 7.14: Example bus structure with extra master providing `rmw` transaction support.

8 Example design

This chapter presents two example descriptions of the same system. The first description uses the register-centric approach and utilizes the AGWB tool. The second description uses the functionality-centric approach and utilizes FBDL. The AGWB tool was chosen for the register-centric approach because of two reasons. The first one is that AGWB belongs to the class of register-centric tools abstracting registers and bit fields as objects, which makes it safer than the register-centric class providing users with addresses, masks, and bit shifts. The second reason is that both AGWB and FBDL use the same VHDL library for the Wishbone bus, which makes the analysis and comparison easier as bus-related signals share the same types.

Listing 54 presents the example bus description in the register-centric AGWB format, and listing 55 presents the same bus in the functionality-centric FBDL format. The first noticeable difference is the verbosity of the register-centric description (64 lines vs 30 lines). This is because the AGWB is based on the XML format, and FBDL is a domain-specific language. However, a functionality-centric approach can also be based on any popular format, such as YAML, JSON, or XML.

Lines of code at first seem like a good metric for code quality. However, the author of [95] provides arguments for desisting from using lines of code as a predictor of software quality.

```
<sysdef top="Main">
    <block name="Subblock_t">
        <!-- Add0, Add1 and Sum registers are part of the addition procedure. -->
        <creg name="Add0">
            <field name="A" width="20"/>
            <field name="B" width="10"/>
        </creg>
        <creg name="Add1" stb="1">
            <field name="C" width="8"/>
        </creg>
        <sreg name="Sum" width = "21"/>

        <!-- Add_Stream0 and Add_Stream1 are part of the addition stream. -->
        <creg name="Add_Stream0">
            <field name="A" width="20"/>
            <field name="B" width="10"/>
        </creg>
        <creg name="Add_Stream1" stb="1">
            <field name="C" width="8"/>
        </creg>
        <!-- Sum_Stream is part of the sum stream. -->
        <sreg name="Sum_Stream" ack="1" width="21"/>
    </block>

    <block name="Main">
        <creg name="C1" width="7"/>
        <creg name="C2" width="9"/>
        <creg name="C3" width="12"/>

        <sreg name="S1" width="7"/>
        <sreg name="S2" width="9"/>
        <sreg name="S3" width="12"/>

        <creg name="CA4" reps="2">
            <field name="Item0" width="8"/>
            <field name="Item1" width="8"/>
            <field name="Item2" width="8"/>
            <field name="Item3" width="8"/>
        </creg>
        <creg name="CA2">
            <field name="Item0" width="8"/>
            <field name="Item1" width="8"/>
        </creg>

        <sreg name="SA4" reps="2">
            <field name="Item0" width="8"/>
            <field name="Item1" width="8"/>
            <field name="Item2" width="8"/>
            <field name="Item3" width="8"/>
        </sreg>
        <sreg name="SA2">
            <field name="Item0" width="8"/>
            <field name="Item1" width="8"/>
        </sreg>

        <sreg name="Counter0" width="32"/>
        <sreg name="Counter1" width="1"/>

        <subblock name="Subblock" type="Subblock_t"/>

        <creg name="Mask" width="16"/>
        <sreg name="Version" width="3*8"/>
    </block>
</sysdef>
```

Listing 54: Example bus description in the register-centric AGWB format.

```
Main bus
    C1 config; width = 7
    C2 config; width = 9
    C3 config; width = 12

    S1 status; width = 7
    S2 status; width = 9
    S3 status; width = 12

    CA [10]config; width = 8
    SA [10]status; width = 8

    Counter status; width = 33

    Subblock block
        Add proc
            A param; width = 20
            B param; width = 10
            C param; width = 8
            Sum return; width = 21

        Add_Stream stream
            A param; width = 20
            B param; width = 10
            C param; width = 8
        Sum_Stream stream
            Sum return; width = 21

    Mask mask; width = 16
    Version static; width = 3*8; init-value = 0x010102
```

Listing 55: Example bus description in the functionality-centric FBDL format.

In listings 54 and 55, `C1`, `C2`, and `C3` represent control information, and `S1`, `S2`, and `S3` represent status information. Within the testbench design, `C1` is directly connected to `S1`, `C2` to `S2`, and `C3` to `S3`. In listing 55, `CA` denotes an array of control data, and `SA` denotes an array of status data. Listing 54 represents the same data as `CA4`, `CA2`, `SA4`, and `SA2` registers. Within the testbench design, the `CA` array is connected directly to the `SA` array. In listing 55, the `Counter` represents status data wider than the bus width. The same `Counter` is represented in listing 54 as registers `Counter0` and `Counter1`. The `Mask` data represents bit mask control data, and it is not connected in the testbench design as it only serves to present the difference between the software interface generated for bit mask handling. The `Version` data represents static information and is directly driven in the testbench design with a fixed value. Figure 8.1 presents the conceptual connection of the data in the testbench design.

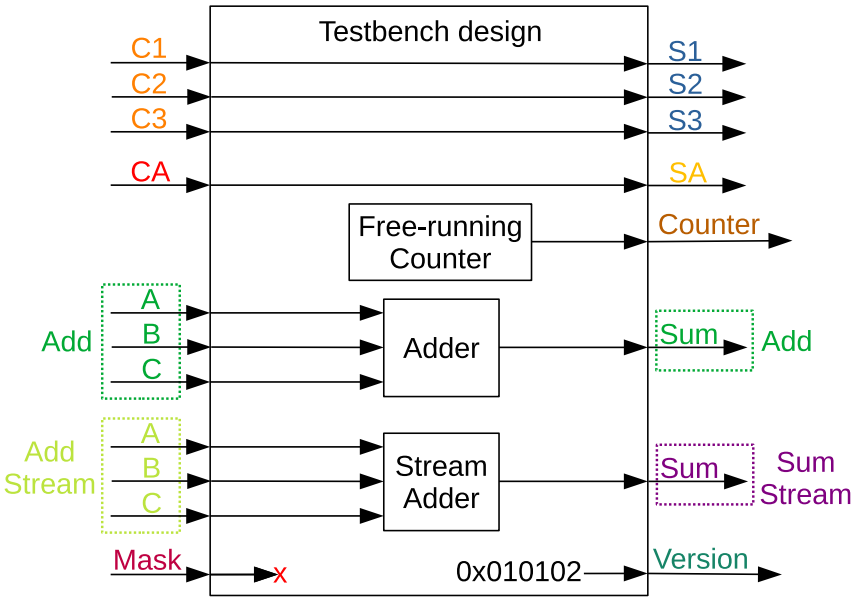


Figure 8.1: Conceptual connection of the data in the testbench design.

Both descriptions have been functionally verified in co-simulations. The repository [96] contains hardware descriptions and software codes used for co-simulations. It also contains all automatically generated files in the `autogen` directory so that readers do not have to install any application to view all relevant files. From the reader's point of view, the most important files are `tb_agwb.vhd`, `tb_fbdl.vhd`, `test_agwb.py`, `test_fbdl.py`, and all files placed in the `autogen` directory. All remaining files are dependency or script files related to the build and run automation and are irrelevant to the analysis. All listings in the chapter come from the repository.

Figure 8.2 presents the logical structure of the FBDL example design co-simulation. The logical structure of the AGWB example co-simulation looks almost the same. In the case of AGWB, the content of yellow blocks is replaced with files generated by the AGWB compiler, and files `test_fbdl.py` and `tb_fbdl.vhd` are replaced with `test_agwb.py` and `tb_agwb.vhd`.

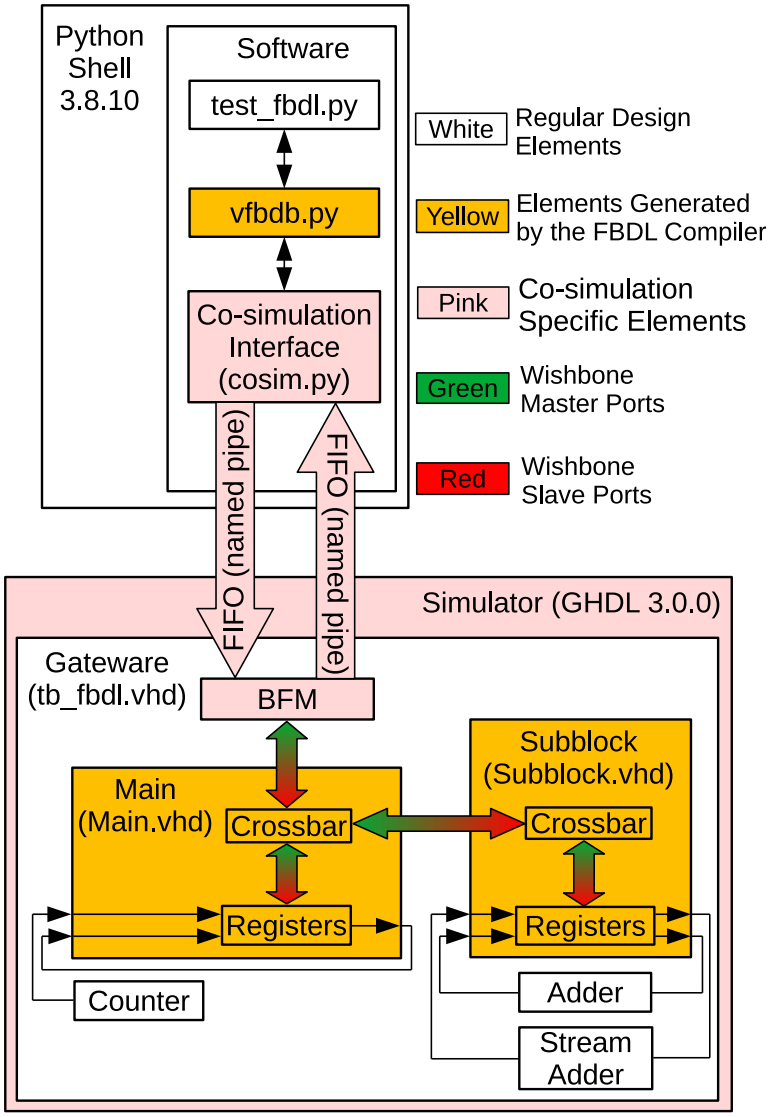


Figure 8.2: Logical structure of the FBDL exapmle design co-simulation.

Appendix B contains registerification results for the bus description from listing 55. This data is produced by the compiler front-end and utilized by the compiler back-end as input data for hardware/gateware description and firmware/software code generation. Appendix C, created manually based on the registerification results, presents the example design register map. Appendix D presents Python code (`vfbdb.py`) automatically generated by the FBDL compiler for the example design description. Python code snippets within subsections 8.1.1, 8.1.2, 8.1.3, 8.1.4, and 8.1.5 directly interact with the code from the appendix to access registers in the automatically generated hardware description. Appendix E contains the VHDL description generated by the FBDL compiler for the `Main` bus entity (`Main.vhd`), and Appendix F contains VHDL description generated for the `Subblock` entity (`Subblock.vhd`).

Listing 56 presents the VHDL interface of the `Main` entity generated by the register-centric AGWB, and listing 57 presents the VHDL interface of the `Main` entity generated by the functionality-centric FBDL compiler. The AGWB defines custom subtypes for the ports. However, these subtypes are simple `std_logic_vector` types, which is irrelevant to the analysis. The primary difference is that in the case of the register-centric approach, the user is provided with ports representing registers. However, in the case of the functionality-centric approach, the user is provided with ports representing data. In the example case, it is visible for `CA` and `SA` versus `CA4`, `CA4`, `SA4`, and `SA2`, as well as for the `Counter`, which is 33 bits wide and in the case of the register-centric approach, it must be divided into two registers manually (`Counter0` and `Counter1`).

```
entity Main is
  port (
    rst_n_i    : in std_logic;
    clk_sys_i  : in std_logic;

    slave_i : in t_wishbone_slave_in;
    slave_o : out t_wishbone_slave_out;

    Subblock_wb_m_o : out t_wishbone_master_out;
    Subblock_wb_m_i : in  t_wishbone_master_in;

    C1_o       : out t_C1;
    C2_o       : out t_C2;
    C3_o       : out t_C3;
    S1_i       : in  t_S1;
    S2_i       : in  t_S2;
    S3_i       : in  t_S3;
    CA4_o      : out ut_CA4_array(g_CA4_size - 1 downto 0);
    CA2_o      : out t_CA2;
    SA4_i      : in  ut_SA4_array(g_SA4_size - 1 downto 0);
    SA2_i      : in  t_SA2;
    Counter0_i : in  t_Counter0;
    Counter1_i : in  t_Counter1;
    Mask_o     : out t_Mask;
    Version_i  : in  t_Version
  );
end Main;
```

Listing 56: Interface of the VHDL `Main` entity generated by the register-centric AGWB.

```
entity Main is
  port (
    clk_i : in std_logic;
    rst_i : in std_logic;

    slave_i : in  t_wishbone_slave_in_array (1 - 1 downto 0);
    slave_o : out t_wishbone_slave_out_array(1 - 1 downto 0);

    Subblock_master_o : out t_wishbone_master_out_array(0 downto 0);
    Subblock_master_i : in  t_wishbone_master_in_array(0 downto 0);

    ID_o : out std_logic_vector(31 downto 0) := x"d2600e2f";

    C1_o      : buffer std_logic_vector(6 downto 0);
    C2_o      : buffer std_logic_vector(8 downto 0);
    C3_o      : buffer std_logic_vector(11 downto 0);
    S1_i      : in     std_logic_vector(6 downto 0);
    S2_i      : in     std_logic_vector(8 downto 0);
    S3_i      : in     std_logic_vector(11 downto 0);
    CA_o      : buffer slv_vector(9 downto 0)(7 downto 0);
    SA_i      : in     slv_vector(9 downto 0)(7 downto 0);
    Counter_i : in     std_logic_vector(32 downto 0);
    Mask_o    : buffer std_logic_vector(15 downto 0);
    Version_o : out    std_logic_vector(23 downto 0) := x"010102"
  );
end entity;
```

Listing 57: Interface of the VHDL `Main` entity generated by the FBDL compiler.

8.1 Functionality-centric approach advantages

The author described the advantages of the functionality-centric approach in [97]. However, the description in the thesis provides more details.

Before any comparisons, the author would like to introduce the "advantage classes" term. The term is not formal but helps to classify the advantages of the functionality-centric approach over a register-centric approach. The advantage class is a characteristic of the quality of the work. There are four advantage classes listed below in alphabetical order:

1. Maintainability (M) - indicates how easy it is to modify the system behavior,
2. Readability (R) - denotes the ease of understanding the system,
3. Safety (S) - represents the probability of human mistake,
4. Time (T) - represents the time required to implement, adjust, or correct the system.

Although the advantage classes are defined, the metrics for Maintainability, Readability, and Time classes are not introduced. This is because the classes are a bit fuzzy, and it is impossible to introduce objective metrics that cannot be questioned. Maintainability, readability, and time are also very subjective concepts. A proof of this fact might be the literature review prepared by authors of [98], who found that until the year 2013, there were about 13000 publications that used lines of code as one of the features for the code quality and maintainability prediction. The total number of publications trying to approach the software quality assessment problem was even greater, as there were also publications not including lines of code. FBDL users are assumed to evaluate solutions in advantage classes based on their expert knowledge, experience, and common sense.

The primary problem with applying objective numerical metrics for code quality assessment is the lack of common definitions. There is not even a single standard definition of readability in the computing domain. Authors of [99] define readability as "*a human judgment of how easy a text is to understand*", authors of [100] define readability as "*a property that influences how easily a given piece of code can be read and understood*", but, for example, authors of [101] define readability as "*the capability of the code that makes it readable and understandable for professionals.*"

There is also no consensus on what constitutes code readability. Most of the proposed models consider structural and visual aspects of code. However, for example, the authors of [102] propose to take into account also textual features. The way the code readability assessment problem is approached is also still evolving. Older methods utilized statistics and graph theory, but, for example, [103] and [104] propose using neural networks.

Some of the proposed advantage classes are also interdependent. For example, the author of [105] claims that source code readability is critical to the maintainability of a project, although it is not the only aspect that constitutes it. The readability also impacts the time required to adjust or correct system implementation in case of bugs or requirement changes.

The secondary problem with applying objective numerical metrics is that authors of publications rarely make implementations available. They describe details of their models, but the user has to implement them by himself. There are no ready-to-use programs that can be easily installed and used for free.

For the Security class, there is a binary metric because in the register-centric approach certain error scenarios are possible, while the functionality-centric approach inherently prevents them.

Titles of subsections 8.1.1, 8.1.2, 8.1.3, 8.1.4, and 8.1.5 are suffixed with letters indicating what advantages classes are brought by the functionality-centric approach compared to the register-centric approach. Within subsections, the author justifies why the functionality-centric approach is advantageous compared to the register-centric approach. Although the author thinks most advantage classes should be assigned to all the presented advantages, only the most significant ones have been chosen.

8.1.1 Automatic data placement (MT)

In the example design, `C1`, `C2`, and `C3` represent control information, and `S1`, `S2`, and `S3` represent status information. The first difference between the register-centric and functionality-centric approaches is that although they represent the same information, they are different entities. In the case of the register-centric approach, the information is represented as registers with proper types. In the case of the functionality-centric approach, the information is represented as data with proper types. The difference has substantial implications. In the case of the register-centric approach, the user must decide ahead on the data placement within registers. For example, in listing 54, `S1`, `S2`, and `S3` are placed in 3 separate registers. However, as `S1`, `S2`, and `S3` are read-only, and their total width is less than 32 bits, they could also be placed in one or two registers. Moreover, they can be placed in the registers with `C1`, `C2`, and `C3` or in a separate register. Even with just 6 data, there are numerous possible placements. In the case of the functionality-centric approach, the compiler is responsible for the data placement within the registers, which reduces development time.

Table 8.1 presents registerification results for single control and status data generated by the functionality-centric FBDL compiler. As can be seen, `S2` has been placed in the same register as `C2`, and `S3` has been placed in the same register as `C3`. The compiler has done this automatically to minimize the required address space. `S1` has been placed in the same register as the `Version`, which is static data (data that is never modified). `C1` has been placed in a separate register with address 6, and this is the only data placed in this register.

Data	Address	Bit range
C1	6	6 : 0
C2	5	8 : 0
C3	4	11 : 0
S1	8	30 : 24
S2	5	17 : 9
S3	4	23 : 12
Mask	7	15 : 0
Version	8	23 : 0

Table 8.1: Registerification results for single control and status data.

Now, let us consider what happens if the system requirements change and the user needs to change the width of some data. For example, both `C3` and `S3` shall now be 2 bits wide. In the case of the register-centric approach, the user must manually adjust the register layout. Depending on the scale of the change, it may be required to reshuffle the bit fields between registers. In the case of a width increase, the data might no longer fit the register width, and the user must manually define additional registers. In the case of a width decrease, the data will still fit the registers. However, the generated address space size might no longer be optimal. In the case of the functionality-centric approach, the data is automatically placed within the registers by the compiler, so the only change the user must introduce is the change of the value of the `width` property. Such an approach improves systems maintainability.

Table 8.2 shows registerification results generated by the FBDL compiler after changing the `C3` and `S3` width to 2. `C2`, `S2`, and `S3` are now placed in the same register. Some addresses are changed because the compiler has found a better register layout. The whole recompilation process takes milliseconds as it is done automatically by the computer application. Doing the same manually by the user would take seconds or even minutes for more complex adjustments.

Data	Address	Bit range
C1	5	6 : 0
C2	4	8 : 0
C3	6	1 : 0
S1	8	30 : 24
S2	4	17 : 9
S3	4	19 : 18
Mask	7	15 : 0
Version	8	23 : 0

Table 8.2: Registerification results for single control and status data after the `C3` and `S3` width change.

Listing 58 presents the VHDL description generated for `C1` access by the register-centric AGWB, and listing 59 presents the VHDL description generated for `C1` access by the functionality-centric FBDL. The difference is minor and irrelevant. The snippets are syntactically different mainly because of the custom types used by the AGWB. The access address also differs because AGWB and FBDL assign addresses to registers in different order. The description has the same semantics because the `C1` width is less than the data bus width. However, the access description generated for the gateware/hardware and the access code generated for the firmware/software has significant differences between register-centric and functionality-centric approaches when the data is an array, when the data is wider than the data bus or when the data forms broader context, what is presented in the corresponding subsections 8.1.2, 8.1.3 and 8.1.4.

```
if int_addr = std_logic_vector(to_unsigned(2, 5)) then
  int_regs_wb_m_i.dat <= (others => '0');
  int_regs_wb_m_i.dat(6 downto 0) <= std_logic_vector(int_C1_o);

  if int_regs_wb_m_o.we = '1' then
    int_C1_o <= std_logic_vector(int_regs_wb_m_o.dat(6 downto 0));
  end if;

  int_regs_wb_m_i.ack <= '1';
  int_regs_wb_m_i.err <= '0';
end if;
```

Listing 58: `C1` VHDL access description generated by the register-centric AGWB.

```
if 6 <= addr and addr <= 6 then
   if master_out.we = '1' then
      C1_o <= master_out.dat(6 downto 0);
   end if;

   master_in.dat(6 downto 0) <= C1_o;
   master_in.ack <= '1';
   master_in.err <= '0';
end if;
```

Listing 59: `C1` VHDL access description generated by the functionality-centric FBDL.

Listing 60 presents Python code for accessing the single data in the register-centric approach, and listing 61 presents Python code for accessing the single data in the functionality-centric approach. Within testbenches, `C1`, `C2`, and `C3` are connected directly to `S1`, `S2`, and `S3` to form a feedback loop in the hardware description. There is almost no difference in the software access code, except classes generated by the FBDL have additional attributes with the data width. However, the access code generated for the firmware/software significantly differs between register-centric and functionality-centric approaches when the data is wider than the data bus, which is presented in subsection 8.1.3.

```
def single_data_test(Main):
    print("Performing Single Data Test")

    r = randint(0, 2 ** 7 - 1)
    Main.C1.write(r)
    assert Main.C1.read() == r
    assert Main.S1.read() == r

    r = randint(0, 2 ** 9 - 1)
    Main.C2.write(r)
    assert Main.C2.read() == r
    assert Main.S2.read() == r

    r = randint(0, 2 ** 12 - 1)
    Main.C3.write(r)
    assert Main.C3.read() == r
    assert Main.S3.read() == r

    print("Single Data Test Passed")
```

Listing 60: Python code for testing access to single data in the register-centric AGWB.

```python
def single_data_test(Main):
    print("Performing Single Data Test")

    r = randint(0, 2 ** Main.C1.width - 1)
    Main.C1.write(r)
    assert Main.C1.read() == r
    assert Main.S1.read() == r

    r = randint(0, 2 ** Main.C2.width - 1)
    Main.C2.write(r)
    assert Main.C2.read() == r
    assert Main.S2.read() == r

    r = randint(0, 2 ** Main.C3.width - 1)
    Main.C3.write(r)
    assert Main.C3.read() == r
    assert Main.S3.read() == r

    print("Single Data Test Passed")
```

Listing 61: Python code for testing access to single data in the functionality-centric FBDL.

8.1.2 Automatic array handling (MRT)

In listing 55, `CA` denotes an array of control data, and `SA` denotes an array of status data. Listing 54 represents the same data as `CA4`, `CA2`, `SA4`, and `SA2` registers. Within testbench designs, the `CA` array is connected directly to the `SA` array to form a feedback loop in the hardware description.

The first difference is that in the register-centric approach, the user must manually lay out an array data in the registers. In the case of the functionality-centric approach, it is the compiler's responsibility. The same difference was presented in the section 8.1.1 for single data. However, the manual placement task is even more time-consuming in the case of array data. Depending on the data width and item count, the array might be represented as a replication of a single register or require an extra register containing a different number of items. The latter is in the example description, where 10 items of width 8 are placed within 3 registers with 4, 4, and 2 distribution. Moreover, not all register-centric tools allow bit field replication within a register. The user must define each bit field separately within the register. For example, in the case of 32 elements array with 1-bit data width, the user must explicitly define 32 bit fields.

The second important distinction between the register-centric and functionality-centric approaches regarding array handling is the generated firmware/software access code. Listing 62 presents Python code for accessing the array data in the register-centric approach,

and listing 63 presents Python code for accessing the array data in the functionality-centric approach. In the register-centric approach, the user must know the relationship between the data index and register and the bit field index. In other words, if the user wants to access data with index `D`, he must explicitly code access to register with index `R` and bit field with index `F`. In the case of the functionality-centric approach, the user operates on the array data, and all the index mapping is handled automatically by the code generated by the compiler. Instead of implicitly handling three indexes `D`, `R`, and `F`, the user only has to handle the `D` index.

```
def array_test(Main):
    print("Performing Array Test")

    data = []
    for _ in range(10):
        data.append(randint(0, 2 ** 8 - 1))

    for i in range(len(Main.CA4)):
        Main.CA4[i].Item0.write(data[0 + i * 4])
        Main.CA4[i].Item1.write(data[1 + i * 4])
        Main.CA4[i].Item2.write(data[2 + i * 4])
        Main.CA4[i].Item3.write(data[3 + i * 4])
    Main.CA2.Item0.write(data[8])
    Main.CA2.Item1.write(data[9])

    rdata = []
    for i in range(len(Main.CA4)):
        rdata.append(Main.CA4[i].Item0.read())
        rdata.append(Main.CA4[i].Item1.read())
        rdata.append(Main.CA4[i].Item2.read())
        rdata.append(Main.CA4[i].Item3.read())
    rdata.append(Main.CA2.Item0.read())
    rdata.append(Main.CA2.Item1.read())
    assert rdata == data, f"got {rdata}, want {data}"

    rdata = []
    for i in range(len(Main.SA4)):
        rdata.append(Main.SA4[i].Item0.read())
        rdata.append(Main.SA4[i].Item1.read())
        rdata.append(Main.SA4[i].Item2.read())
        rdata.append(Main.SA4[i].Item3.read())
    rdata.append(Main.SA2.Item0.read())
    rdata.append(Main.SA2.Item1.read())
    assert rdata == data, f"got {rdata}, want {data}"

    print("Array Test Passed")
```

Listing 62: Python code for testing access to array data in the register-centric AGWB.

```python
def array_test(Main):
    print("Performing Array Test")

    data = []
    for _ in range(len(Main.CA)):
        data.append(randint(0, 2 ** Main.CA.width - 1))
    Main.CA.write(data)

    rdata = Main.CA.read()
    assert rdata == data, f"got {rdata}, want {data}"

    rdata = Main.SA.read()
    assert rdata == data, f"got {rdata}, want {data}"

    print("Array Test Passed")
```

Listing 63: Python code for testing access to array data in the functionality-centric FBDL.

The FBDL compiler is able to handle arrays with a single element of any width. Listing 64 presents the VHDL access description generated for the `CA` array. As can be seen, 4 elements are placed in the register with address 1, another 4 elements are placed in the register with address 2, and the remaining 2 are placed in the register with address 3.

```
if 1 <= addr and addr <= 2 then
  for i in 0 to 3 loop
     if master_out.we = '1' then
        CA_o((addr-1)*4+i) <= master_out.dat(8*(i+1) + 0-1 downto 8*i + 0);
     end if;
     master_in.dat(8*(i+1) + 0-1 downto 8*i + 0) <= CA_o((addr-1)*4+i);
  end loop;

  master_in.ack <= '1';
  master_in.err <= '0';
end if;

if 3 <= addr and addr <= 3 then
  for i in 0 to 1 loop
     if master_out.we = '1' then
        CA_o(8+i) <= master_out.dat(8*(i+1) + 0-1 downto 8*i+0);
     end if;
     master_in.dat(8*(i+1) + 0-1 downto 8*i+0) <= CA_o(8+i);
  end loop;

  master_in.ack <= '1';
  master_in.err <= '0';
end if;
```

Listing 64: `CA` (size = 10, width = 8) array VHDL access description generated by the FBDL compiler.

Let us suppose that system requirements have changed, and `CA` shall now be an array of 30 elements with a width equal to 1 bit. Listing 65 presents the adjustment that has to be applied to the bus description, and listing 66 presents the VHDL access description generated for the new `CA` declaration. As can be seen, all the elements are now placed within a single register with address 1.

```
diff --git a/fbd/bus.fbd b/fbd/bus.fbd
index 3cdd74e..0a13424 100644
--- a/fbd/bus.fbd
+++ b/fbd/bus.fbd
@@ -7,7 +7,7 @@ Main bus
        S2 status; width = 9
        S3 status; width = 12

-       CA [10]config; width = 8
+       CA [30]config; width = 1
        SA [10]status; width = 8
```

Listing 65: Functional bus description diff for `CA` size change to 30 and width change to 1.

```
if 1 <= addr and addr <= 1 then
  for i in 0 to 29 loop
     if master_out.we = '1' then
        CA_o(i) <= master_out.dat(1*(i+1)+0-1 downto 1*i+0);
     end if;
     master_in.dat(1*(i+1)+0-1 downto 1*i+0) <= CA_o(i);
  end loop;

  master_in.ack <= '1';
  master_in.err <= '0';
end if;
```

Listing 66: `CA` (size = 30, width = 1) array VHDL access description generated by the FBDL compiler.

Let us suppose that system requirements have changed once more, and `CA` shall now be an array of 6 elements with a width equal to 21 bits. Listing 67 presents the adjustment that has to be applied to the bus description. Listing 68 presents the VHDL access description generated for the new `CA` definition. As can be seen, each item is now placed in a separate register and spans bits from 0 to 20. The first array register has address 1, and the last one has address 6.

```
diff --git a/fbd/bus.fbd b/fbd/bus.fbd
index 3cdd74e..0a13424 100644
--- a/fbd/bus.fbd
+++ b/fbd/bus.fbd
@@ -7,7 +7,7 @@ Main bus
        S2 status; width = 9
        S3 status; width = 12

-       CA [30]config; width = 1
+       CA [6]config; width = 21
        SA [10]status; width = 8
```

Listing 67: Functional bus description diff for `CA` size change to 6 and width change to 21.

```
if 1 <= addr and addr <= 6 then
  if master_out.we = '1' then
     CA_o(addr - 1) <= master_out.dat(20 downto 0);
  end if;
  master_in.dat(20 downto 0) <= CA_o(addr - 1);

  master_in.ack <= '1';
  master_in.err <= '0';
end if;
```

Listing 68: `CA` (size = 6, width = 21) array VHDL access description generated by the functionality-centric FBDL.

The FBDL compiler is also capable of handling access to arrays with elements wider than the data bus. Such an example is not presented as access atomicity is described in detail in subsection 8.1.3.

8.1.3 Access atomicity (MST)

In the example design, the `Counter` represents status data that is wider than the bus width. In the functionality-centric approach, each instantiated functionality has a bit width independent of the data bus width. In register-centric approaches (CII being the exception), the designer explicitly puts data into the registers. Hence, it cannot be defined as wider than the data bus width. For the data wider than the register width, the user must define multiple registers and partition the data into the registers. However, when the access to the data must be atomic, two additional issues arise:

1. Atomic data value change must be manually described in the HDL (vhdMMIO is the exception as it has the concept of logical registers and is capable of generating atomic access hardware description).

2. Correct access order to the registers must be manually implemented in the firmware/-software. The data is latched on reading the first register in the case of data reads

and updated on writing the last register in the case of data writes. Incorrect access order results in an invalid value if data changes during the transaction.

In the functionality-centric approach, the compiler automatically handles the additional issues related to access atomicity, as data is treated as an indivisible whole, not as a fragmented piece.

Figure 8.3 presents waveforms, and listing 69 presents the generated VHDL description for `Counter` access in the register-centric approach. Counter registers sampling times are marked with markers. During the first register read, the counter value equals `0x1FFFFFFFF` , and the value on the data bus equals `0xFFFFFFFF`. This is the expected value. Before the second read, the counter overflows. During the second read, the counter value equals `0x000000004`, and the value on the data bus equals `0x00000000`. The read value of the bit with index 33 is incorrect because during the first read, it equaled `'1'`, but it equaled `'0'` during the second read. The final `Counter` read value equals `0x0FFFFFFFF`, instead of `0x1FFFFFFFF`. This is two times less than expected. The problem occurs not only when values overflow but also when there is a change in the middle bits of a signal wider than the data bus. For example, if the `Counter` were 65 bits wide, the same problem with bit 33 would occur even though the `Counter` did not overflow.

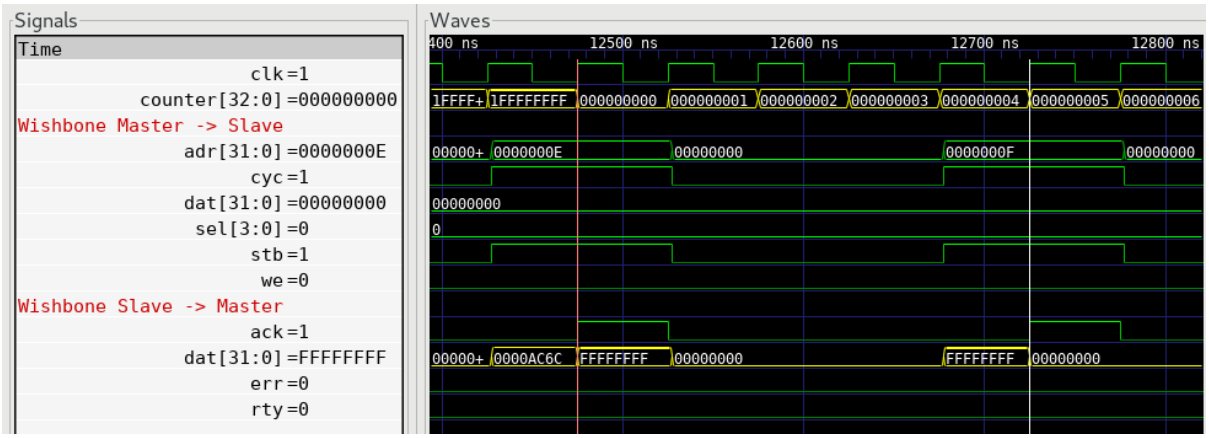


Figure 8.3: Counter non-atomic access issues in the register-centric approach.

```
if int_addr = std_logic_vector(to_unsigned(14 + i, 5)) then
  int_regs_wb_m_i.dat <= (others => '0');
  int_regs_wb_m_i.dat(31 downto 0) <= std_logic_vector(Counter0_i);
  int_regs_wb_m_i.ack <= '1';
  int_regs_wb_m_i.err <= '0';
end if;
if int_addr = std_logic_vector(to_unsigned(15 + i, 5)) then
  int_regs_wb_m_i.dat <= (others => '0');
  int_regs_wb_m_i.dat(0 downto 0) <= std_logic_vector(Counter1_i);
  int_regs_wb_m_i.ack <= '1';
  int_regs_wb_m_i.err <= '0';
end if;
```

Listing 69: `Counter` VHDL access description generated by the register-centric AGWB.

Figure 8.4 presents waveforms, and listing 70 presents the generated VHDL description for the `Counter` access in the functionality-centric approach. Counter registers sampling times are marked with markers. During the first register read, the counter value equals `0x1FFFFFFFF`, and the value on the data bus equals `0xFFFFFFFF`. This is the expected value. Before the second read, the counter overflows. During the second read, the counter value equals `0x000000005`. However, the value on the data bus equals `0x00000001`. The read value of the bit with index 33 equals `'1'` because this was the bit value when the first register read occurred. Compared to the register-centric approach, in the functionality-centric approach, the user is provided with the data, not the registers, and it is the compiler's responsibility to guarantee atomic data access.

The generated VHDL access description has an additional `Counter_atomic` signal latching the value of the remaining `Counter` bits when the read of the first register storing the `Counter` value happens. Such a register is often called a snapshot or shadow register, although, to the author's knowledge, there is no formal definition. When reading the upper bit, the data bus is driven from the snapshot register, not the `Counter` register.

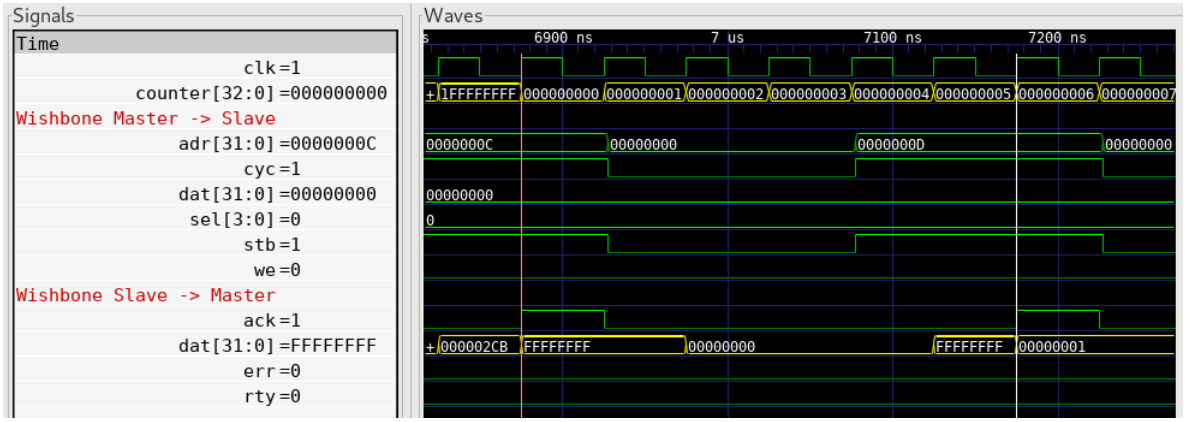


Figure 8.4: Counter atomic access in the functionality-centric approach.

```
-- Signal declaratin line taken from the architecture declarative part.
signal Counter_atomic : std_logic_vector(32 downto 32);
-- If statements taken from the process statement part.
if 12 <= addr and addr <= 12 then
  Counter_atomic(32 downto 32) <= Counter_i(32 downto 32);
  master_in.dat(31 downto 0) <= Counter_i(31 downto 0);
  master_in.ack <= '1';
  master_in.err <= '0';
end if;
if 13 <= addr and addr <= 13 then
  master_in.dat(0 downto 0) <= Counter_atomic(32 downto 32);
  master_in.ack <= '1';
  master_in.err <= '0';
end if;
```

Listing 70: `Counter` VHDL access description generated by the FBDL compiler.

In the register-centric approach, the user can provide access atomicity similarly. However, it must be done manually outside the automatically generated bus fabric description. This requires extra time and introduces room for potential mistakes, for example, when the user accidentally connects the wrong bits to the snapshot register. It is also required that the first read register has an associated read acknowledgment signal that triggers the snapshot register data latch.

Providing access atomicity consumes extra resources, and not all data wider than the data bus requires atomic access. In such a case, the FBDL compiler can be informed to discard access atomicity for particular data. The user can simply set the atomic property to false (`atomic = false`).

Listing 71 presents Python code for accessing `Counter` in the register-centric approach, and listing 72 presents Python code for accessing `Counter` in the functionality-centric approach. In the case of the register-centric approach, the user must manually implement a valid access order and recreate the value. For the example `Counter` , there are at least three possible mistakes: invalid register access order, invalid bit shift, and invalid order of arguments for the bitwise or operator (`|`). In the functionality-centric approach, the user simply reads the data. Everything related to the data read is handled automatically by the compiler.

```
print("Performing Counter Test")
cnt0 = Main.Counter0.read()
cnt1 = Main.Counter1.read()
cnt = (cnt1 << 32) | cnt0
assert cnt == 0x1FFFFFFFF
print("Counter Test Passed")
```

Listing 71: Python code for testing `Counter` access in the register-centric AGWB.

```
print("Performing Counter Test")
cnt = Main.Counter.read()
assert cnt == 0x1FFFFFFFF
print("Counter Test Passed")
```

Listing 72: Python code for testing `Counter` access in the functionality-centric FBDL.

8.1.4 Procedure and stream contexts (MRS)

In listing 54, registers `Add0`, `Add1`, and `Sum` represent a procedure. The procedure is a simple addition procedure with three summands: `A`, `B`, and `C`. The `Sum` register stores the operation result.

Describing procedures in the register-centric approach has the following drawbacks:

1. The user must manually place the parameter and return data in the registers. In the case of procedure data, atomic access is not required as all the argument data is latched using an additional strobe signal. Usually, the strobe pulse is generated when the last register storing procedures argument is written. No atomicity requirement means the argument data can be packed tightly in the registers to minimize the required address space. The bits of the single data can be split into two registers even if the data width is less than the data bus width. In the case of listing 54, 2 bits of `C` could be placed in the register `Add0`, and only the remaining 6 bits could be placed in the register `Add1`. This would not change the number of required registers in the example addition procedure. However, such an approach can reduce the number of required registers in the case of procedures with more parameters. Moreover, if there is enough space in the parameters' last register, the return data can be placed there. However, not all register-centric tools allow placing control and status data in the same register. In the case of procedure data change, the register-centric approach might require manual and time-consuming data reshuffling between registers.

2. Without additional comment, a user can only guess based on the register names that particular registers form a procedure context. Even with the comment, it may not be up to date, as it must be manually synced.

3. The user must provide correct parameter registers write order and return registers read order in firmware/software.

Listing 73 presents the interface of the VHDL `Subblock` entity generated by the register-centric AGWB. The user is provided with the addition procedure registers directly. There is no encapsulation of the procedure context.

```
entity Subblock_t is
  port (
    rst_n_i   : in std_logic;
    clk_sys_i : in std_logic;

    slave_i : in  t_wishbone_slave_in;
    slave_o : out t_wishbone_slave_out;

    Add0_o     : out t_Add0;
    Add1_o     : out t_Add1;
    Add1_o_stb : out std_logic;
    Sum_i      : in  t_Sum;

    Add_Stream0_o     : out t_Add_Stream0;
    Add_Stream1_o     : out t_Add_Stream1;
    Add_Stream1_o_stb : out std_logic;

    Sum_Stream_i     : in  t_Sum_Stream;
    Sum_Stream_i_ack : out std_logic
  );
end Subblock_t;
```

Listing 73: Interface of the VHDL `Subblock` entity generated by the AGWB.

Listing 74 presents Python code for testing the addition procedure in the register-centric approach in the co-simulation testbench. The `A` and `B` bit fields are written separately, meaning the register `Add0` is written twice. This can be avoided. However, it is the user's responsibility to create a valid value for the `Add0` register write. The single `Add0` write approach line is commented out.

```
def add_test(Main):
    print("Performing Add Test")

    a = randint(0, 2 ** 20 - 1)
    b = randint(0, 2 ** 10 - 1)
    c = randint(0, 2 ** 8 - 1)

    Main.Subblock.Add0.A.write(a)
    Main.Subblock.Add0.B.write(b)
    # Main.Subblock.Add0.write((b << 20) | a)
    Main.Subblock.Add1.C.write(c)
    assert Main.Subblock.Sum.read() == a + b + c

    print("Add Test Passed")
```

Listing 74: Python code for testing addition procedure in the register-centric approach.

The same addition procedure is presented in the listing 55. The functionality-centric procedure description is free of all the register-centric drawbacks. This is because the procedure context is defined explicitly using the `proc` functionality. Parameters and returns have their widths, but it is the compiler's responsibility to place them in registers.

The compiler is also responsible for generating the firmware/software method for calling the procedure, so there is no room for the incorrect access order mistake.

Listing 75 presents the interface of the VHDL `Subblock` entity generated by the functionality-centric FBDL. The procedure-related signals are encapsulated as record types. The `call` signal is driven high for one clock cycle every time the last parameter register is written. The `exitt` (‘`exit`’ is VHDL keyword) signal is driven high for one clock cycle every time the last return register is read. In the example system, the `exitt` signal is ignored, and the `call` signal is used to trigger the add procedure. The `Sum` is updated every time the `call` signal equals `'1'`.

```
-- Record type declarations taken from the Subblock_pkg.
type Add_out_t is record
   A : std_logic_vector(19 downto 0);
   B : std_logic_vector(9 downto 0);
   C : std_logic_vector(7 downto 0);
   call  : std_logic;
   exitt : std_logic;
end record;
type Add_in_t is record Sum : std_logic_vector(20 downto 0); end record;
type Add_Stream_t is record
   A : std_logic_vector(19 downto 0);
   B : std_logic_vector(9 downto 0);
   C : std_logic_vector(7 downto 0);
end record;
type Sum_Stream_t is record Sum : std_logic_vector(20 downto 0); end record;

entity Subblock is
port (
   clk_i : in std_logic;
   rst_i : in std_logic;

   slave_i : in  t_wishbone_slave_in_array (1 - 1 downto 0);
   slave_o : out t_wishbone_slave_out_array(1 - 1 downto 0);

   Add_o : out Add_out_t;
   Add_i : in  Add_in_t;

   Add_Stream_o     : out Add_Stream_t;
   Add_Stream_stb_o : out std_logic;

   Sum_Stream_i     : in  Sum_Stream_t;
   Sum_Stream_stb_o : out std_logic
);
end entity;
```

Listing 75: Interface of the VHDL `Subblock` entity generated by the FBDL.

Listing 76 presents Python code for testing the addition procedure in the co-simulation testbench. In the functionality-centric approach, a user is provided with a function that can be explicitly called. There is no need to write parameter registers manually to call the procedure.

```
def add_test(Main):
    print("Performing Add Test")
    a = randint(0, 2 ** 20 - 1)
    b = randint(0, 2 ** 10 - 1)
    c = randint(0, 2 ** 8 - 1)
    assert Main.Subblock.Add(a, b, c)[0] == a + b + c
    print("Add Test Passed")
```

Listing 76: Python code for testing add procedure in the functionality-centric approach.

Listing 77 presents the add procedure access description generated by the functionality-centric FBDL. The address signal (`addr`) is a relative subblock address, hence low address values such as 0 and 1. The compiler automatically put the lower 2 bits of the `C` parameter in the same register as the `A` and `B` parameters. The compiler also put the return `Sum` in the same register as the upper 6 bits of the `C` parameter. Figure 8.5 presents waveforms for the add procedure test. There are 3 bus transactions, 2 writes, and 1 read. The second write access has the same address as the read access. Yellow waveforms have decimal formatting. The result of the procedure is correct ($1045694 + 484 + 117 = 1046295$). It also may be noticed that the value of the `C` parameter changes twice on both writes.

```
if 0 <= addr and addr <= 0 then
  if master_out.we = '1' then
     Add_o.A <= master_out.dat(19 downto 0);
  end if;
  master_in.dat(19 downto 0) <= Add_o.A;
  if master_out.we = '1' then
     Add_o.B <= master_out.dat(29 downto 20);
  end if;
  master_in.dat(29 downto 20) <= Add_o.B;
  if master_out.we = '1' then
     Add_o.C(1 downto 0) <= master_out.dat(31 downto 30);
  end if;
  master_in.dat(31 downto 30) <= Add_o.C(1 downto 0);
  master_in.ack <= '1';
  master_in.err <= '0';
end if;
if 1 <= addr and addr <= 1 then
  if master_out.we = '1' then
     Add_o.C(7 downto 2) <= master_out.dat(5 downto 0);
  end if;
  master_in.dat(5 downto 0) <= Add_o.C(7 downto 2);
  master_in.dat(26 downto 6) <= Add_i.Sum;
  master_in.ack <= '1';
  master_in.err <= '0';
end if;
```

Listing 77: Add procedure access description generated by the functionality-centric FBDL.

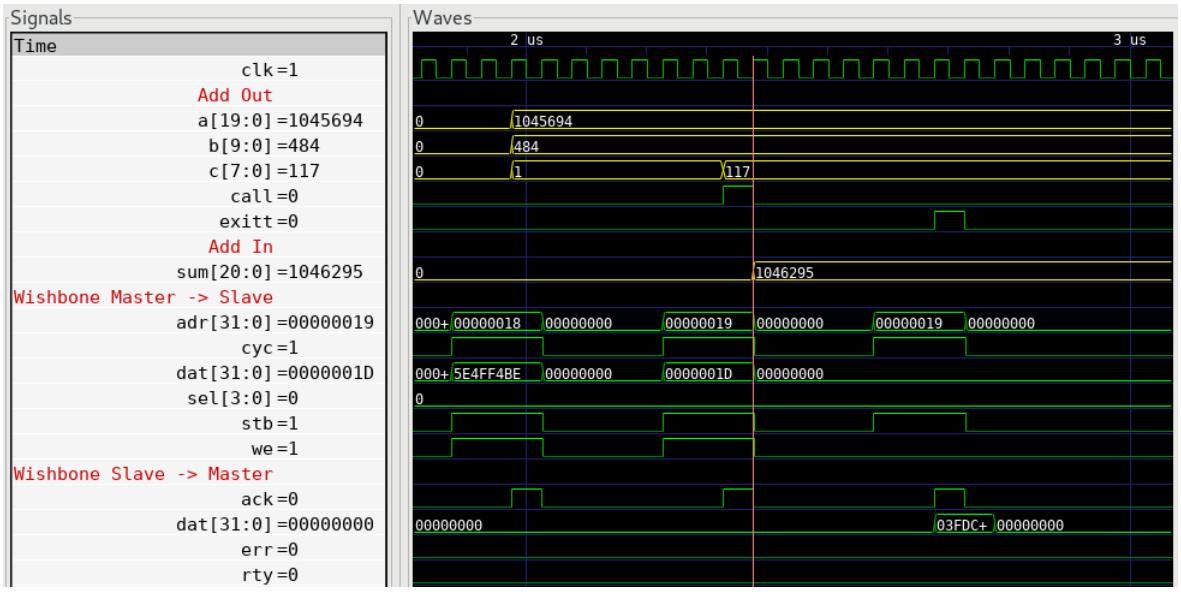


Figure 8.5: Add procedure waveforms for testbench utilizing description generated by the FBDL compiler.

In a practical design, registers often form not only procedure contexts but also stream contexts. A stream is very similar to a procedure. The first difference is that a stream is always unidirectional. It always has only parameters (downstream) or only returns (upstream). The second difference is in the way firmware/software calls a stream. An access to a stream is multiple, and an access to a procedure is single. The example design contains two streams `Add_Stream` downstream and `Sum_Stream` upstream. While the gateware/hardware description generated for streams is almost identical to the description generated for procedures, the generated firmware/software code has different API. Listing 78 presents Python code for testing addition streams in the co-simulation testbench. The first noticeable difference is that streams accept or return an array of datasets, whereas procedures accept and return a single dataset. The second difference is purely nomenclatural. Streams instead of being called, are written (downstream) or read (upstream).

```python
def streams_test(Main):
    print("Performing Streams Test")

    data = []
    for i in range(16):
        dataset = []
        dataset.append(randint(0, 2**Main.Subblock.Add_Stream.params[0]['Width']-1))
        dataset.append(randint(0, 2**Main.Subblock.Add_Stream.params[1]['Width']-1))
        dataset.append(randint(0, 2**Main.Subblock.Add_Stream.params[2]['Width']-1))
        data.append(dataset)
    Main.Subblock.Add_Stream.write(data)

    sums = Main.Subblock.Sum_Stream.read(16)
    for i, dataset in enumerate(data):
        got = sums[i][0]
        want = sum(dataset)
        assert got == want, f"{i}: got {got}, want {want}"

    print("Streams Test Passed")
```

Listing 78: Python code for testing addition streams in the functionality-centric approach.

The procedure and streams examples are contained within the subblock, both in the register-centric AGWB and functionality-centric FBDL approach. The goal is to present that in both approaches, describing the hierarchy of the modules is possible and is very similar and straightforward. The `block` functionality also allows defining the number of masters connected to the subblock via the `masters` property, which allows for connecting multiple bus or physical interfaces. The additional interfaces can be connected directly or via a custom bridge if a protocol translation is required.

8.1.5 Additional types (R)

In the register-centric approach, a user declares the register type. Most available tools offer control registers and status registers. A control register can be read and written from the firmware/software and read (sometimes also written) from the gateware/hardware. A status register can be read from the firmware/software, and read and written from the gateware/hardware. vhdMMIO is slightly different in this term as it has the concept of register behavior. The register behavior is an extension of the register type. However, this is still the register type, not the data type.

In the functionality-centric approach offered by the FBDL, a user declares the type of the data, not the register. This, in turn, allows for introducing additional types, increasing the amount of code that can be automatically generated, and improving the system's readability.

In listing 54, there is a control register named `Mask` and a status register named `Version`. The `Mask` is a bit mask, meaning the user will want to set, clear, and toggle particular bits of the value. The `Version` represents static information that never changes. However, in the register-centric approach, the user can only guess based on the names that `Mask` is a bit mask and `Version` is static. The description could have extra documentation comments explaining the purpose of the registers, but it would have to be manually kept up to date.

In listing 55, the same `Mask` and `Version` have distinct types. Based on the type, it is not only clear what functionality is served by the data, but it is also possible to generate access functions for firmware/software with the desired programming interface. In the case of the static `Version`, it is also possible to assign a value where the static data is defined, which improves readability. In most register-centric tools, the user has to check the gateware/hardware description to realize that the status register value never changes because it is driven by a constant signal. This is shown in listing 79, which presents an instantiation of the `Main` entity generated by the register-centric AGWB.

```
agwb_main : entity agwb.Main
port map (
  clk_sys_i => clk,
  rst_n_i => '1',
  slave_i => wb_ms,
  slave_o => wb_sm,
  Subblock_wb_m_o => subblock_wb_ms,
  Subblock_wb_m_i => subblock_wb_sm,

  C1_o => c1,
  C2_o => c2,
  C3_o => c3,

  S1_i => c1,
  S2_i => c2,
  S3_i => c3,

  CA4_o => ca4,
  CA2_o => ca2,

  SA4_i => sa4,
  SA2_i => sa2,

  Counter0_i => std_logic_vector(counter(31 downto 0)),
  Counter1_i => std_logic_vector(counter(32 downto 32)),

  Mask_o    => mask,
  Version_i => x"010102"
);
```

Listing 79: Instantiation of the VHDL `Main` entity generated by the register-centric AGWB.

Listing 80 presents Python code for testing access to the `Mask` data in the register-centric AGWB co-simulation testbench, and listing 81 presents Python code for the same test in the functionality-centric FBDL approach. In the case of the FBDL, it is possible to generate access methods operating directly on bits, which saves time and reduces the probability of human mistakes. In the case of the register-centric approach, it is impossible, as the compiler does not know the purpose of the data stored in the control register.

```
def mask_test(Main):
    print("Performing Mask Test")
    # Setting particular bits
    bits = [1, 3, 8, 15]
    mask = 0
    for b in bits:
        mask |= 1 << b
    Main.Mask.write(mask)
    for idx in range(16):
        val = mask & (1 << idx)
        if idx in bits:
            assert val == 1 << idx, f"bit {idx} not set"
        else:
            assert val == 0, f"bit {idx} set"
    # Toggling bits
    mask = Main.Mask.read()
    mask ^= (1 << 1)
    Main.Mask.write(mask)
    assert Main.Mask.read() & (1 << 1) == 0, "mask toggle didn't work"
    print("Mask Test Passed")
```

Listing 80: Python code for testing access to `Mask` data in the register-centric approach.

```
def mask_test(Main):
    print("Performing Mask Test")
    bits = [1, 3, 8, 15]
    Main.Mask.set(bits)
    mask = Main.Mask.read()
    for idx in range(Main.Mask.width):
        val = mask & (1 << idx)
        if idx in bits:
            assert val == 1 << idx, f"bit {idx} not set"
        else:
            assert val == 0, f"bit {idx} set"
    Main.Mask.toggle(1)
    assert Main.Mask.read() & (1 << 1) == 0, "mask toggle didn't work"
    print("Mask Test Passed")
```

Listing 81: Python code for testing access to `Mask` data in the functionality-centric FBDL.

8.2 Synthesis results

Hardware descriptions generated by the register-centric AGWB and the functionality-centric FBDL were synthesized to compare resource utilization and show that the functionality-centric approach is not only a theoretical concept but also something that works in practice.

A few adjustments have to be introduced to the logical design from figure 8.2 to make the example design work with real hardware. Namely, the co-simulation-specific elements (the pink ones) must be replaced with counterparts corresponding to the physical connection. The BFM was replaced with a custom SPI - Wishbone bridge, and GHDL simulator was replaced with the Cmod S7 development board [106]. FIFOs were replaced with physical cables and USB-SPI converter [107]. Python co-simulation interface was replaced with a custom interface implementation utilizing `busio` module from the Adafruit Blinka package [108]. It is worth mentioning that regardless of the run environment (co-simulation or actual hardware), elements generated by the FBDL compiler (the yellow ones) are the same. There is no need to regenerate files.

Vivado 2021.2 was used as the synthesis tool. The Synthesis Strategy was set to the `Default`, and the `-flatten_hierarchy` property was set to `none`. The target part was `XC7S25-1CSGA225C` (Spartan 7).

Figures 8.6 and 8.7 present post-synthesis resource utilization for the register-centric AGWB and functionality-centric FBDL. Table 8.3 presents the same information but in a more straightforward comparison format.

The number of utilized registers is the same for AGWB and FBDL. This is expected as the number of required registers depends on the total number of data bits, and both approaches describe the same data but use different approaches. The number of utilized registers is higher than the number of data bits because some registers are utilized for the bus logic. However, as the same bus and VHDL library are used in both examples, the number of registers utilized by the bus logic is the same.

The number of utilized LUTs is higher for the register-centric AGWB. In the case of the functionality-centric FBDL, the compiler automatically optimized the required address space size, resulting in simpler address decoding logic. The description generated by the register-centric AGWB effectively utilizes 22 addresses, and the description generated by the functionality-centric FBDL effectively utilizes 19 addresses.

Name	Slice LUTs (14600)	Slice Registers (29200)	F7 Muxes (7300)	F8 Muxes (3650)	Bonded IOB (100)	BUFGCTRL (32)
Top_AGWB	489	470	21	2	5	1
SPI_Wb_Bridge (SPI_Wb_Bridge)	143	133	5	2	0	0
agwb_subblock (Subblock_t)	95	113	0	0	0	0
xwb_crossbar_1 (xwb_crossbar__parameterized0)	34	0	0	0	0	0
agwb_main (Main)	202	162	16	0	0	0
xwb_crossbar_1 (xwb_crossbar)	94	4	0	0	0	0

Figure 8.6: Post-synthesis resource utilization for the register-centric AGWB.

Name	Slice LUTs (14600)	Slice Registers (29200)	F7 Muxes (7300)	F8 Muxes (3650)	Bonded IOB (100)	BUFGCTRL (32)
Top_FBDL	415	469	9	2	5	1
SPI_Wb_Bridge (SPI_Wb_Bridge)	145	132	5	2	0	0
vfbdb_main (Main)	150	162	4	0	0	0
crossbar (xwb_crossbar)	59	4	0	0	0	0
vfbdb_subblock (Subblock)	71	113	0	0	0	0
crossbar (xwb_crossbar__parameterized0)	35	1	0	0	0	0

Figure 8.7: Post-synthesis resource utilization for the functionality-centric FBDL.

		Main	**Subblock**
LUTs	**AGWB**	202	95
	FBDL	150	71
Registers	**AGWB**	162	113
	FBDL	162	113

Table 8.3: Post-synthesis resource utilization.

9 Real use case

The implemented FBDL compiler was 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 G contains the statement from the Fluence company confirming the use of the FBDL compiler.

10 Summary

Describing system bus registers at the functional level using FBDL offers the following advantages in certain practical use cases 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 contains more information about the system. There is no need to read gateware/hardware description or firmware/software code to know that particular 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.
4. Less probability of non-atomic data access bugs. In FBDL, access to any data is atomic by default. Any compiler compilant 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. However, 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 data functionality level.

The FBDL may also be used for on-chip connections utilizing the NoC technology. As each network node 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*, 53(3):60, March 2017. https://doi.org/10.1140/epja/i2017-12248-y.

[2] Compressed Baryonic Matter experiment. https://www.gsi.de//work/forschung/cbmnqm/cbm.htm. Accessed: 1 October 2023.

[3] Wojciech M. Zabołotny, Marek Gumiński, Michał Kruszewski, and Walter F.J. Müller. Control and Diagnostics System Generator for Complex FPGA-Based Measurement Systems. *Sensors*, 21(21), 2021. https://doi.org/10.3390/s21217378.

[4] ARM. AMBA AXI and ACE Protocol Specification. https://developer.arm.com/documentation/ihi0022/latest/. Accessed: 3 June 2021.

[5] OpenCores. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. https://cdn.opencores.org/downloads/wbspec_b4.pdf. Accessed: 3 June 2021.

[6] G. De Michell and R.K. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, March 1997. Conference Name: Proceedings of the IEEE, https://doi.org/10.1109/5.558708.

[7] A.A. Jerraya and W. Wolf. Hardware/software interface codesign for embedded systems. *Computer*, 38(2):63–69, 2005. https://doi.org/10.1109/MC.2005.61.

[8] Juha Takalo, Jukka Kääriäinen, Päivi Parviainen, and Tuomas Ihme. Challenges of software-hardware co-design. *VTT WORKING PAPERS*, page 49, 2008.

[9] J. Kokila, N. Ramasubramanian, and S. Indrajeet. A Survey of Hardware and Software Co-design Issues for System on Chip Design. In Ramesh K. Choudhary, Jyotsna Kumar Mandal, Nitin Auluck, and H A Nagarajaram, editors, *Advanced Computing and Communication Technologies*, pages 41–49, Singapore, 2016. Springer Singapore.

[10] Racha Ganesh. Design Issues in Hardware/Software Co-Design. *International Journal of Research in Electronics & Communication Technology*, June 2020.

[11] Luca Zulberti, Stefano Di Matteo, Pietro Nannipieri, Sergio Saponara, and Luca Fanucci. A Script-Based Cycle-True Verification Framework to Speed-Up Hardware

and Software Co-Design: Performance Evaluation on ECC Accelerator Use-Case. *Electronics*, 11(22), 2022.

[12] NamDo Kim, JunHyuk Park, Byeong Min, and Wesley Park. Register Verification: Do We Have Reliable Specification? In *Design and Verification Conference and Exhibition*, 2013.

[13] Haytham Saafan, M. Watheq El-Kharashi, and Ashraf Salem. Formal Based Methodology for Inferring Memory Mapped Registers. In *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 15–18, 2016. https://doi.org/10.1109/MTV.2016.11.

[14] Feng Zhang. *High-speed Serial Buses in Embedded Systems*. Springer Singapore, 2020. https://doi.org/10.1007/978-981-15-1868-3.

[15] Intel. ISA Bus Specification and Application Notes. https://archive.org/details/bitsavers_intelbusSpep89_3342148/mode/2up. Accessed: 14 March 2023.

[16] Extended Industry Standard Architecture (EISA) Specification 3.1. https://www.os2museum.com/files/docs/EISA_Specification-v3.1.pdf. Accessed: 14 March 2023.

[17] Tube Time. Micro Channel Bus Tutorial. https://github.com/schlae/mca-tutorial. Accessed: 14 March 2023.

[18] Infotel. VESA LOCAL BUS 3486, 786 MINI-BOARD USER'S MANUAL. https://theretroweb.com/motherboard/manual/vlbus3486-61e8755ab7989037625802.pdf. Accessed: 14 March 2023.

[19] American National Standards Institute. Small Computer System Interface-2. https://global.ihs.com/doc_detail.cfm?document_name=ANSI%20INCITS%20131&item_s_key=00009673&item_key_date=911231. Accessed: 14 March 2023.

[20] Hewlett-Packard, Intel, Microsoft, Renesas, ST-Ericsson, and Texas Instruments. Universal Serial Bus 3.1 Specification. https://manuais.iessanclemente.net/images/b/bc/USB_3_1_r1.0.pdf. Accessed: 14 March 2023.

[21] PCI SIG. PCI Specifications. https://pcisig.com/specifications. Accessed: 14 March 2023.

[22] S. Pasricha and N. Dutt. *On-Chip Communication Architectures*. Elsevier, 2008. https://doi.org/10.1016/B978-0-12-373892-9.X0001-1.

[23] John Bainbridge. *Asynchronous System-on-Chip Interconnect*. Springer London, 2002. https://doi.org/10.1007/978-1-4471-0189-5.

[24] IBM Microelectronics. CoreConnect Bus Architecture. https://www.xilinx.com/content/dam/xilinx/support/documents/user_guides/crcon_pb.pdf. Accessed: 14 March 2023.

[25] Intel. Introduction to the Avalon Interface Specifications. https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/introduction-to-the-interface-specifications.html. Accessed: 14 March 2023.

[26] STMicroelectronics. STBus communication system concepts and definitions. https://www.st.com/resource/en/user_manual/cd00176920-stbus-communication-system-concepts-and-definitions-stmicroelectronics.pdf. Accessed: 14 March 2023.

[27] W.J Bainbridge and S.B Furber. Marble: an asynchronous on-chip macrocell bus. *Microprocessors and Microsystems*, 24(4):213–222, 2000. https://doi.org/10.1016/S0141-9331(00)00075-2.

[28] ARM. AMBA AXI-Stream Protocol Specification. https://developer.arm.com/documentation/ihi0051/latest/. Accessed: 10 April 2023.

[29] 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)*, pages 250–256, 2000. https://doi.org/10.1109/DATE.2000.840047.

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

[31] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, 2002. https://doi.org/10.1109/2.976921.

[32] 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*, pages 117–124, 2002. https://doi.org/10.1109/ISVLSI.2002.1016885.

[33] AMD Xilinx. AXI Adapter Interface Protocols. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/AXI-Adapter-Interface-Protocols. Accessed: 12 March 2023.

[34] Jie Chen, Paul Gillard, and Cheng Li. Network-on-chip (noc) topologies and performance: A review. 2011.

[35] Tetala Neel Kamal Reddy, Ayas Kanta Swain, Jayant Kumar Singh, and Kamala Kanta Mahapatra. Performance assessment of different Network-on-Chip topologies. In *2014 2nd International Conference on Devices, Circuits and Systems (ICDCS)*, pages 1–5, 2014. https://doi.org/10.1109/ICDCSyst.2014.6926188.

[36] Hridoy Jyoti Mahanta, Abhijit Biswas, and Md. Anwar Hussain. Networks on chip: The new trend of on-chip interconnection. In *2014 Fourth International Conference on Communication Systems and Network Technologies*, pages 1050–1053, 2014. https://doi.org/10.1109/CSNT.2014.214.

[37] Alakesh Kalita, Kaushik Ray, Abhijit Biswas, and Md. Anwar Hussain. A topology for network-on-chip. In *2016 International Conference on Information Communication and Embedded Systems (ICICES)*, pages 1–7, 2016. https://doi.org/10.1109/ICICES.2016.7518838.

[38] Ana Kumar, Shivam Tyagi, and C. K. Jha. Performance analysis of network-on-chip topologies. *Journal of Information and Optimization Sciences*, 38(6):989–997, 2017. https://doi.org/10.1080/02522667.2017.1372145.

[39] Isiaka A. Alimi, Romil K. Patel, Oluyomi Aboderin, Abdelgader M. Abdalla, Ramoni A. Gbadamosi, Nelson J. Muga, Armando N. Pinto, and António L. Teixeira. Network-on-chip topologies: Potentials, technical challenges, recent advances and research direction. In Isiaka A. Alimi, Oluyomi Aboderin, Nelson J. Muga, and António L. Teixeira, editors, *Network-on-Chip*, chapter 3. IntechOpen, Rijeka, 2021. https://doi.org/10.5772/intechopen.97262.

[40] Internet Engineering Task Force. Internet Protocol. Request for Comments RFC 791, Internet Engineering Task Force, September 1981. Num Pages: 51, https://doi.org/10.17487/RFC0791.

[41] Internet Engineering Task Force. Transmission Control Protocol. Request for Comments RFC 793, Internet Engineering Task Force, September 1981. Num Pages: 91, https://doi.org/10.17487/RFC0793.

[42] Jonathan R. Engdahl and Dukki Chung. Device register classes for embedded systems. In *2011 11th International Conference on Control, Automation and Systems*, pages 773–778, 2011.

[43] noasic GmbH. airhdl. https://airhdl.com/#/. Accessed: 18 February 2023.

[44] Introducing JSON. https://www.json.org/json-en.html. Accessed: 18 February 2023.

[45] HTML Standard. https://html.spec.whatwg.org/. Accessed: 18 February 2023.

[46] World Wide Web Consortium. Extensible Markup Language. https://www.w3.org/TR/xml/. Accessed: 18 February 2023.

[47] Wojciech M. Zabołotny. Address Generator for Wishbone. https://github.com/wzab/agwb. Accessed: 20 February 2023.

[48] Wojciech M. Zabołotny. Adr_gen - Automatic Address Generator. https://github.com/wzab/wzab-hdl-library/tree/master/addr_gen. Accessed: 20 February 2023.

[49] Dan Gisselquist. AutoFPGA. https://github.com/ZipCPU/autofpga. Accessed: 21 February 2023.

[50] Przemyslaw Plutecki, Bartosz Przemyslaw Bielawski, and Andrew Butterworth. Code Generation Tools and Editor for Memory Maps. *Proceedings of the 17th International Conference on Accelerator and Large Experimental Physics Control Systems*, ICALEPCS2019:4 pages, 0.730 MB, 2020. Artwork Size: 4 pages, 0.730 MB ISBN: 9783954502097 Medium: PDF Publisher: JACoW Publishing, Geneva, Switzerland.

[51] cheby. https://gitlab.cern.ch/be-cem-edl/common/cheby. Accessed: 22 February 2023.

[52] Agustin James Rey, J. C. Molendijk, Frederic Dubouchet, A. Pashnin, Andrew Butterworth, Michael Jaussi, and T. Levens. Cheburashka: A Tool for Consistent Memory Map Configuration Across Hardware and Software. pages 848–851, October 2013.

[53] Evgeniy Bolnov. Corsair. https://github.com/esynr3z/corsair. Accessed: 24 February 2023.

[54] Wojciech M. Zabołotny, Marek Gumiński, and Michał Kruszewski. Automatic management of local bus address space in complex FPGA-implemented hierarchical systems. In Ryszard S. Romaniuk and Maciej Linczuk, editors, *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2019*, volume 11176, page 1117642. International Society for Optics and Photonics, SPIE, 2019. https://doi.org/10.1117/12.2536259.

[55] AMD. Exporting a Block Design to a Tcl Script in the IDE. https://docs.xilinx.com/r/en-US/ug994-vivado-ip-subsystems/Exporting-a-Block-Design-to-a-Tcl-Script-in-the-IDE. Accessed: 11 January 2024.

[56] Lukas Vik. hdl_registers: An open-source HDL register generator fast enough to run in real time. https://hdl-registers.com/index.html. Accessed: 17 February 2023.

[57] Krzysztof T Pozniak. I/O communication with FPGA circuits and hardware description standard for applications in HEP and FEL electronics. https://doi.org/10.3204/PUBDB-2018-03052.

[58] Deutsches Elektronen-Synchrotron. https://www.desy.de/, August 2012. Accessed: 10 April 2023.

[59] Pawel Drabik and Krzysztof T. Pozniak. Maintaining complex and distributed measurement systems with component internal interface framework. In Ryszard S. Romaniuk and Krzysztof S. Kulpa, editors, *Proc. SPIE*, volume 7502, page 75022C, Wilga, Poland, June 2009. https://doi.org/10.1117/12.838155.

[60] Agnieszka Zagoździńska, Krzysztof T. Poźniak, and Paweł K. Drabik. Selected issues of the universal communication environment implementation for CII standard. page 80080N, Wilga, Poland, June 2011. https://doi.org/10.1117/12.902748.

[61] 1685-2014 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows.

[62] Accellera IP-XACT Working Group. IP-XACT User Guide. https://www.accellera.org/images/downloads/standards/ip-xact/IP-XACT_User_Guide_2018-02-16.pdf. Accessed: 24 February 2023.

[63] Tudor Timi. rgen. https://github.com/tudortimi/rgen. Accessed: 24 February 2023.

[64] Onur Balci. IPXACT Register Map Generator. https://github.com/legoritma/ipxact-register-generator. Accessed: 23 February 2023.

[65] 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*, pages 262–265, August 2011. https://doi.org/10.1109/DSD.2011.36.

[66] Antti Kamppi, Lauri Matilainen, Joni-Matti Maatta, Erno Salminen, and Timo D. Hamalainen. Extending ip-xact to embedded system hw/sw integration. In *2013 International Symposium on System on Chip (SoC)*, pages 1–8, 2013. https://doi.org/10.1109/ISSoC.2013.6675264.

[67] Felix Miller. Root-of-Trust-Architekturen als Open-Source-Hardware und deren Zertifizierung: am Beispiel von OpenTitan. *Datenschutz und Datensicherheit*, 44(7):451–455, 2020.

[68] Opentitan. https://docs.opentitan.org/. Accessed: 22 February 2023.

[69] Opentitan Register Tool. https://docs.opentitan.org/doc/rm/register_tool/. Accessed: 22 February 2023.

[70] bitvis. Register Wizard Abandoned. https://github.com/UVVM/UVVM/issues/200. Accessed: 24 February 2023.

[71] Espen Tallaksen. Auto-generate register related code and documentation - for free. https://www.linkedin.com/pulse/auto-generate-register-related-code-doc-free-espen-tallaksen/. Accessed: 24 February 2023.

[72] bitvis. Verifying corner cases in a structured manner - using VHDL Verification Components. http://program.fpgaworld.com/2016/More_information/Bitvis__Verifying_CornerCases_Handout.pdf. Accessed: 24 February 2023.

[73] Taichi Ishitani. RgGen. https://github.com/rggen/rggen. Accessed: 16 February 2023.

[74] Aliaksei Chapyzhenka. Design u Hardware. https://github.com/sifive/duh. Accessed: 16 February 2023.

[75] Accellera Systems Initiative. SystemRDL. https://www.accellera.org/downloads/standards/systemrdl. Accessed: 18 February 2023.

[76] AGNISYS. Next Generation SystemRDL - Using IDesignSpec for register implementation. Accessed: 16 February 2023.

[77] eVision Systems. IDesignSpec. https://evision-systems.com/linecard/agnisys/idesignspec/. Accessed: 16 February 2023.

[78] Semifore. CSRCompiler. http://semifore.com/csrcompiler/. Accessed: 16 February 2023.

[79] Juniper. open-register-design-tool. https://github.com/Juniper/open-register-design-tool. Accessed: 16 February 2023.

[80] SystemRDL open source community. Free and open-source SystemRDL tools. https://github.com/SystemRDL. Accessed: 16 February 2023.

[81] Michael Buechler. desyrdl. https://gitlab.desy.de/fpgafw/tools/desyrdl. Accessed: 16 February 2023.

[82] Jeroen van Straten. vhdMMIO. https://github.com/abs-tudelft/vhdmmio. Accessed: 23 March 2023.

[83] Tomasz Włostowski. Wishbone slave generator. https://ohwr.org/project/wishbone-gen. Accessed: 16 February 2023.

[84] K. Poulos, K. Adaos, and G.P. Alexiou. Automated Generation of the Register Set of a SOC and its Verification Environment. In *Proceedings of the 18th Panhellenic Conference on Informatics*, PCI '14, page 1–2, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2645791.2645851.

[85] V. K. Vytla and L. Doolittle. Newad: A register map automation tool for Verilog. https://arxiv.org/abs/2305.09657/. Accessed: 20 January 2024.

[86] Oren Ben-Kiki, Clark Evans, and Ingy dot Net. YAML Specification Index. https://yaml.org/spec/. Accessed: 18 February 2023.

[87] AMD Xilinx. Super Logic Region. https://docs.xilinx.com/r/en-US/ug912-vivado-properties/SLR. Accessed: 12 March 2023.

[88] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990. https://doi.org/10.1109/IEEESTD.1990.101064.

[89] B. Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computers*, 37(12):1506–1514, 1988. https://doi.org/10.1109/12.9729.

[90] Michał Kruszewski. go-fbdl. https://github.com/Functional-Bus-Description-Language/go-fbdl. Accessed: 14 April 2023.

[91] Michał Kruszewski. go-vfbdb. https://github.com/Functional-Bus-Description-Language/go-vfbdb. Accessed: 14 April 2023.

[92] Max Brunsfeld, Andrew Hlynskyi, Patrick Thomson, Josh Vera, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Hendrik van Antwerpen, Michael Davis, Ika, Tuấn-Anh Nguyễn, Stafford Brunk, Niranjan Hasabnis, bfredl, Mingkai Dong, Matt Massicotte, Jonathan Arnett, Vladimir Panteleev, Steven Kalt, Kolja Lampe, Alex Pinkus, Mark Schmitz, Matthew Krupcale, narpfel, Santos Gallegos, Vicent Martí, and Edgar. tree-sitter/tree-sitter: v0.20.8, April 2023. https://doi.org/10.5281/zenodo.7798573.

[93] Michał Kruszewski. tree-sitter-fbdl. https://github.com/Functional-Bus-Description-Language/tree-sitter-fbdl. Accessed: 13 April 2023.

[94] Masaru Tomita, editor. *Generalized LR Parsing*. Springer US, Boston, MA, 1991. https://doi.org/10.1007/978-1-4615-4034-2.

[95] Vard Antinyan. Hypnotized by Lines of Code. *Computer*, 54(1):42–48, 2021. https://doi.org/10.1109/MC.2019.2943844.

[96] Michał Kruszewski. Register-centric vs functionality-centric example. https://github.com/Functional-Bus-Description-Language/Example. 2024-01-24.

[97] Michał Kruszewski . How shifting focus from register to data functionality can enhance register and bus management. *Electronics*, 13(4), 2024. https://doi.org/10.3390/electronics13040719.

[98] Danijel Radjenović, Marjan Hericko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55:1397–1418, August 2013. https://doi.org/10.1016/j.infsof.2013.02.009.

[99] Raymond P.L. Buse and Westley R. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010. https://doi.org/10.1109/TSE.2009.70.

[100] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 73–82, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/1985441.1985454.

[101] Aisha Batool, Muhammad Bilal Bashir, Muhammad Babar, Adnan Sohail, and Naveed Ejaz. Effect or Program Constructs on Code Readability and Predicting Code Readability Using Statistical Modeling. *Foundations of Computing and Decision Sciences*, 46(2):127–145, June 2021. https://doi.org/10.2478/fcds-2021-0009.

[102] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018. https://doi.org/10.1002/smr.1958.

[103] Qing Mi, Jacky Keung, Yan Xiao, Solomon Mensah, and Yujin Gao. Improving code readability classification using convolutional neural networks. *Information and Software Technology*, 104:60–71, 2018. https://doi.org/10.1016/j.infsof.2018.07.006.

[104] Qing Mi. Rank Learning-Based Code Readability Assessment with Siamese Neural Networks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3551349.3560440.

[105] K. K. Aggarwal, Yogendra Pratap Singh, and Jitender Kumar Chhabra. An integrated measure of software maintainability. *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*, pages 235–241, 2002.

[106] Digilent. Cmod S7 Breadboardable Spartan-7 FPGA Module. https://digilent.com/reference/programmable-logic/cmod-s7/start. Accessed: 20 January

2024.

[107] Adafruit. FT232H - USB converter for UART, SPI, I2C, GPIO - Adafruit 2264. https://botland.store/uart-rs232-rs485-converters/3003-ft232h-usb-converter-for-uart-spi-i2c-gpio--5904422354008.html. Accessed: 20 January 2024.

[108] Adafruit. Blinka. https://github.com/adafruit/Adafruit_Blinka. Accessed: 20 January 2024.

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,
  "Blackboxes": 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,
      "Blackboxes": 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 Example design registerification results

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

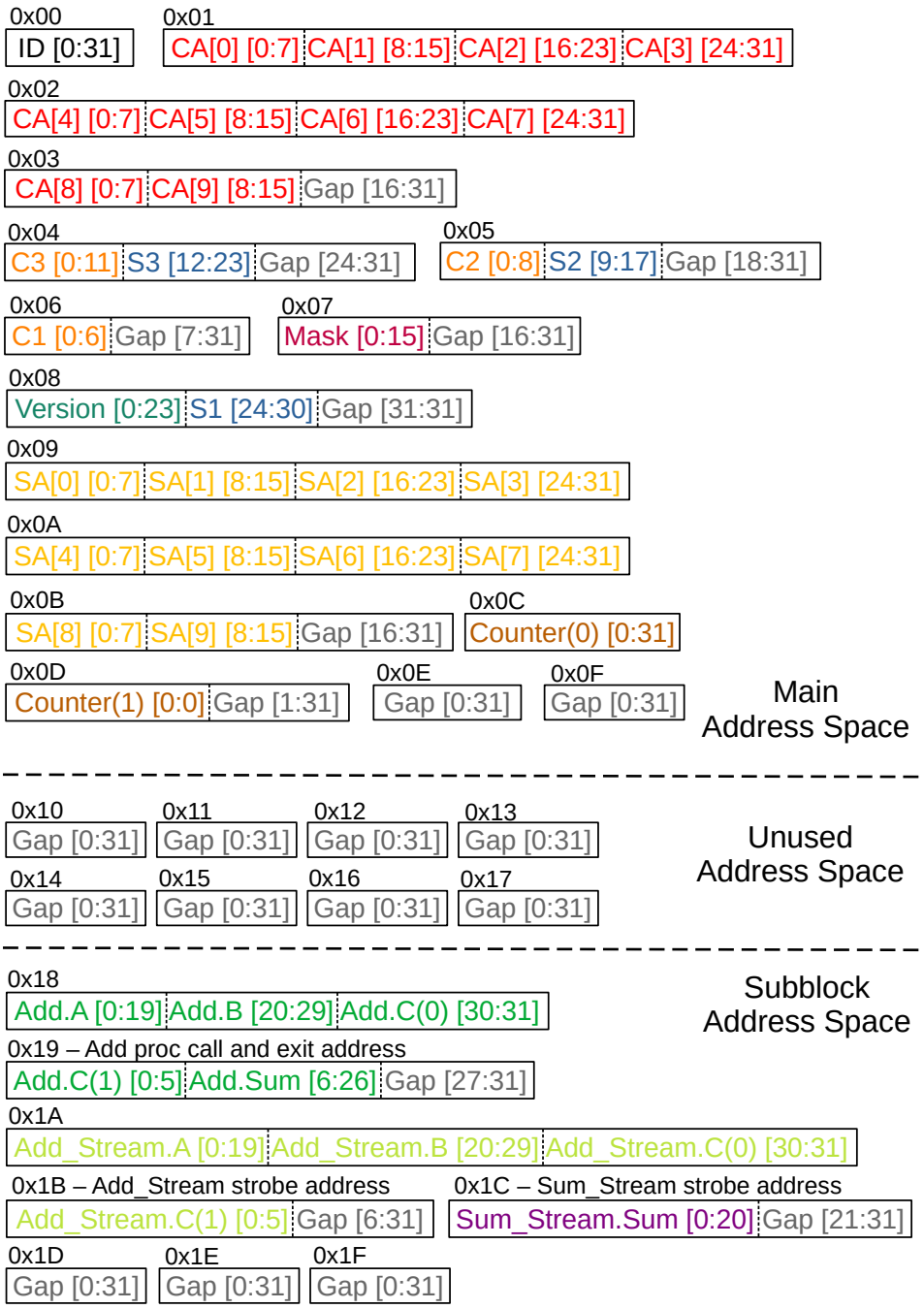

```
{
  "Name": "Main",
  "Doc": "",
  "IsArray": false,
  "Count": 1,
  "Masters": 1,
  "Reset": "",
  "Width": 32,
  "Sizes": { "BlockAligned": 32, "Compact": 19, "Own": 14 },
  "AddrSpace": { "Start": 0, "End": 31 },
  "Consts": {
    "Bools": null, "BoolLists": null, "Floats": null, "Ints": null, "IntLists": null, "Strings": null
  },
  "Configs": [ {
      "Name": "C1",
      "Doc": "",
      "IsArray": false,
      "Count": 1,
      "Atomic": true,
      "InitValue": "",
      "Groups": null,
      "Range": null,
      "ReadValue": "",
      "ResetValue": "",
      "Width": 7,
      "Access": { "Type": "SingleOneReg", "Addr": 6, "StartBit": 0, "EndBit": 6 }
    }, {
      "Name": "C2",
      "Doc": "",
      "IsArray": false,
      "Count": 1,
      "Atomic": true,
      "InitValue": "",
      "Groups": null,
      "Range": null,
      "ReadValue": "",
      "ResetValue": "",
      "Width": 9,
      "Access": { "Type": "SingleOneReg", "Addr": 5, "StartBit": 0, "EndBit": 8 }
    }, {
      "Name": "C3",
      "Doc": "",
      "IsArray": false,
      "Count": 1,
      "Atomic": true,
      "InitValue": "",
      "Groups": null,
      "Range": null,
      "ReadValue": "",
      "ResetValue": "",
      "Width": 12,
      "Access": { "Type": "SingleOneReg", "Addr": 4, "StartBit": 0, "EndBit": 11 }
    }, {
      "Name": "CA",
      "Doc": "",
      "IsArray": true,
      "Count": 10,
      "Atomic": true,
      "InitValue": "",
      "Groups": null,
      "Range": null,
      "ReadValue": "",
      "ResetValue": "",
```

```
        "Width": 8,
        "Access": {
          "Type": "ArrayNInRegMInEndReg", "RegCount": 3, "ItemCount": 10, "ItemWidth": 8,
          "ItemsInReg": 4, "ItemsInEndReg": 2, "StartAddr": 1, "StartBit": 0
        }
      }
    ],
    "Irqs": null,
    "Masks": [ {
        "Name": "Mask",
        "Doc": "",
        "IsArray": false,
        "Count": 1,
        "Atomic": true,
        "Groups": null,
        "InitValue": "",
        "ReadValue": "",
        "ResetValue": "",
        "Width": 16,
        "Access": { "Type": "SingleOneReg", "Addr": 7, "StartBit": 0, "EndBit": 15 }
      }
    ],
    "Memories": null,
    "Procs": null,
    "Statics": [ {
        "Name": "Version",
        "Doc": "",
        "IsArray": false,
        "Count": 1,
        "Groups": null,
        "InitValue": "x\"010102\"",
        "ReadValue": "",
        "ResetValue": "",
        "Width": 24,
        "Access": { "Type": "SingleOneReg", "Addr": 8, "StartBit": 0, "EndBit": 23 }
      }, {
        "Name": "ID",
        "Doc": "Bus identifier.",
        "IsArray": false,
        "Count": 1,
        "Groups": null,
        "InitValue": "x\"cacd0d6f\"",
        "ReadValue": "",
        "ResetValue": "",
        "Width": 32,
        "Access": { "Type": "SingleOneReg", "Addr": 0, "StartBit": 0, "EndBit": 31 }
      }
    ],
    "Statuses": [ {
        "Name": "S1",
        "Doc": "",
        "IsArray": false,
        "Count": 1,
        "Atomic": true,
        "Groups": null,
        "ReadValue": "",
        "Width": 7,
        "Access": { "Type": "SingleOneReg", "Addr": 8, "StartBit": 24, "EndBit": 30 }
      }, {
        "Name": "S2",
        "Doc": "",
        "IsArray": false,
        "Count": 1,
        "Atomic": true,
        "Groups": null,
        "ReadValue": "",
        "Width": 9,
        "Access": { "Type": "SingleOneReg", "Addr": 5, "StartBit": 9, "EndBit": 17 }
      }, {
        "Name": "S3",
        "Doc": "",
        "IsArray": false,
        "Count": 1,
        "Atomic": true,
        "Groups": null,
        "ReadValue": "",
        "Width": 12,
        "Access": { "Type": "SingleOneReg", "Addr": 4, "StartBit": 12, "EndBit": 23 }
      }, {
        "Name": "SA",
        "Doc": "",
```

```
      "IsArray": true,
      "Count": 10,
      "Atomic": true,
      "Groups": null,
      "ReadValue": "",
      "Width": 8,
      "Access": {
        "Type": "ArrayNInRegMInEndReg", "RegCount": 3, "ItemCount": 10, "ItemWidth": 8,
        "ItemsInReg": 4, "ItemsInEndReg": 2, "StartAddr": 9, "StartBit": 0
      }
    }, {
      "Name": "Counter",
      "Doc": "",
      "IsArray": false,
      "Count": 1,
      "Atomic": true,
      "Groups": null,
      "ReadValue": "",
      "Width": 33,
      "Access": { "Type": "SingleNRegs", "RegCount": 2, "StartAddr": 12, "StartBit": 0, "EndBit": 0 }
    }
  ],
  "Streams": null,
  "Subblocks": [ {
      "Name": "Subblock",
      "Doc": "",
      "IsArray": false,
      "Count": 1,
      "Masters": 1,
      "Reset": "",
      "Width": 32,
      "Sizes": { "BlockAligned": 8, "Compact": 5, "Own": 5 },
      "AddrSpace": { "Start": 24, "End": 31 },
      "Consts": {
        "Bools": null, "BoolLists": null, "Floats": null, "Ints": null, "IntLists": null, "Strings": null
      },
      "Configs": null,
      "Irqs": null,
      "Masks": null,
      "Memories": null,
      "Procs": [ {
          "Name": "Add",
          "Doc": "",
          "IsArray": false,
          "Count": 1,
          "Delay": null,
          "Params": [ {
              "Name": "A",
              "Doc": "",
              "IsArray": false,
              "Count": 1,
              "Groups": null,
              "Range": null,
              "Width": 20,
              "Access": { "Type": "SingleOneReg", "Addr": 0, "StartBit": 0, "EndBit": 19 }
            }, {
              "Name": "B",
              "Doc": "",
              "IsArray": false,
              "Count": 1,
              "Groups": null,
              "Range": null,
              "Width": 10,
              "Access": { "Type": "SingleOneReg", "Addr": 0, "StartBit": 20, "EndBit": 29 }
            }, {
              "Name": "C",
              "Doc": "",
              "IsArray": false,
              "Count": 1,
              "Groups": null,
              "Range": null,
              "Width": 8,
              "Access": {
                "Type": "SingleNRegs", "RegCount": 2, "StartAddr": 0,
                "StartBit": 30, "EndBit": 5
              }
            }
          ],
          "Returns": [ {
              "Name": "Sum",
              "Doc": "",
```

```
                "IsArray": false,
                "Count": 1,
                "Groups": null,
                "Width": 21,
                "Access": { "Type": "SingleOneReg", "Addr": 1, "StartBit": 6, "EndBit": 26 }
              }
            ],
            "CallAddr": 1,
            "ExitAddr": 1
          }
        ],
        "Statics": null,
        "Statuses": null,
        "Streams": [ {
            "Name": "Add_Stream",
            "Doc": "",
            "IsArray": false,
            "Count": 1,
            "Delay": null,
            "Params": [ {
                "Name": "A",
                "Doc": "",
                "IsArray": false,
                "Count": 1,
                "Groups": null,
                "Range": null,
                "Width": 20,
                "Access": { "Type": "SingleOneReg", "Addr": 2, "StartBit": 0, "EndBit": 19 }
              }, {
                "Name": "B",
                "Doc": "",
                "IsArray": false,
                "Count": 1,
                "Groups": null,
                "Range": null,
                "Width": 10,
                "Access": { "Type": "SingleOneReg", "Addr": 2, "StartBit": 20, "EndBit": 29 }
              }, {
                "Name": "C",
                "Doc": "",
                "IsArray": false,
                "Count": 1,
                "Groups": null,
                "Range": null,
                "Width": 8,
                "Access": {
                  "Type": "SingleNRegs", "RegCount": 2, "StartAddr": 2, "StartBit": 30, "EndBit": 5
                }
              }
            ],
            "Returns": null,
            "StbAddr": 3
          },
          {
            "Name": "Sum_Stream",
            "Doc": "",
            "IsArray": false,
            "Count": 1,
            "Delay": null,
            "Params": null,
            "Returns": [ {
                "Name": "Sum",
                "Doc": "",
                "IsArray": false,
                "Count": 1,
                "Groups": null,
                "Width": 21,
                "Access": { "Type": "SingleOneReg", "Addr": 4, "StartBit": 0, "EndBit": 20 }
              }
            ],
            "StbAddr": 4
          }
        ],
        "Subblocks": null
      }
    ]
  }
```

C Example design register map



D Python code automatically generated for the example design

```python
# This file has been automatically generated by the vfbdb tool.
# Do not edit it manually, unless you really know what you do.
# https://github.com/Functional-Bus-Description-Language/go-vfbdb

import math
import time

BUS_WIDTH = 32


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 _BufferIface:
    """
    _BufferIface is the internal interface used for reading/writing internal buffer
    (after reading)/(before writing) the target buffer. It is very useful
    as it allows treating proc or stream params/returns as configs/statuses.
    """

    def set_buf(self, buf):
        self.buf = buf

    def write(self, addr, data):
        self.buf[addr] = data

    def read(self, addr):
        return self.buf[addr]


def check_arg_values(params, *args):
    """
    check_arg_values checks that all arguments are in valid range and raises
    an exception if any argument is out of range.
    """
    for arg_idx, arg in enumerate(args):
        param = params[arg_idx]

        type = param['Access']['Type']

        if type.startswith("Single"):
            assert 0 <= arg < 2 ** param['Width'], "{} value overrange ({})".format(
                param['Name'], arg
            )
        elif type.startswith("Array"):
            assert (
                len(arg) == param['Access']['ItemCount']
            ), "invalid number of items ({}) for {} param, expecting {} items".format(
                len(arg), param['Name'], param['ItemCount']
            )

```

```
            for val_idx, v in enumerate(arg):
                assert (
                    0 <= v < 2 ** param['Width']
                ), "{}[{}] value overrange ({})".format(param['Name'], val_idx, v)
        else:
            raise Exception("invalid param access type {}".format(type))


def pack_params(params, *args):
    check_arg_values(params, *args)

    buf = []
    addr = None  # Current argument address
    data = 0

    for arg_idx, arg in enumerate(args):
        param = params[arg_idx]
        a = param['Access']

        if addr is None:
            addr = a['StartAddr']
        elif a['StartAddr'] > addr:
            buf.append(data)
            data = 0
            addr = a['StartAddr']

        if a['Type'] == 'SingleOneReg':
            data |= arg << a['StartBit']
        elif a['Type'] == 'SingleNRegs':
            for r in range(a['RegCount']):
                if r == 0:
                    data |= (arg & calc_mask((BUS_WIDTH - 1, a['StartBit']))) << a[
                        'StartBit'
                    ]
                    buf.append(data)
                    arg = arg >> (BUS_WIDTH - a['StartBit'])
                else:
                    addr += 1
                    data = arg & calc_mask((BUS_WIDTH, 0))
                    arg = arg >> BUS_WIDTH
                    if r < a['RegCount'] - 1:
                        buf.append(data)
                        data = 0
        elif a['Type'] == 'ArrayNRegs':
            start_bit = a['StartBit']
            for i, v in enumerate(arg):
                width = param['Width']
                # Number of registers ith argument from vector occupies.
                reg_count = (
                    int(math.ceil((width - (BUS_WIDTH - start_bit)) / BUS_WIDTH)) + 1
                )
                for _ in range(reg_count):
                    reg_width = width
                    if reg_width > BUS_WIDTH - start_bit:
                        reg_width = BUS_WIDTH - start_bit
                    data |= (v & ((1 << reg_width) - 1)) << start_bit
                    v >>= reg_width
                    start_bit = start_bit + reg_width
                    if start_bit >= BUS_WIDTH:
                        buf.append(data)
                        data = 0
                        start_bit %= BUS_WIDTH
                    width -= reg_width
        else:
            raise Exception("unhandled access type {}".format(a['Type']))

    buf.append(data)

```

```
    return buf


def create_mock_returns(buf_iface, start_addr, returns):
    """
    Create_mock_returns creates mock returns that can be used with internal software buffer.
    It is useful to be used with proc with returns and with upstram.
    """
    buf_size = 0
    rets = []
    for ret in returns:
        a = ret['Access']
        buf_size += a['RegCount']
        r = {}
        r['Name'] = ret['Name']

        if a['Type'] == 'SingleOneReg':
            r['Status'] = StatusSingleOneReg(
                buf_iface, a['StartAddr'] - start_addr, a['StartBit'], a['EndBit']
            )
        elif a['Type'] == 'SingleNRegs':
            r['Status'] = StatusSingleNRegs(
                buf_iface,
                a['StartAddr'] - start_addr,
                a['RegCount'],
                (BUS_WIDTH - 1, a['StartBit']),
                (a['EndBit'], 0),
            )
        else:
            raise Exception("unimplemented")

        rets.append(r)

    return buf_size, rets


class EmptyProc:
    def __init__(self, iface, call_addr, delay, exit_addr):
        self.iface = iface
        self.call_addr = call_addr
        self.delay = delay
        self.exit_addr = exit_addr

    def __call__(self):
        self.iface.write(self.call_addr, 0)
        if self.delay is not None:
            if self.delay != 0:
                time.sleep(self.delay)
            self.iface.read(self.exit_addr)


class ParamsProc:
    def __init__(self, iface, params_start_addr, params, delay, exit_addr):
        self.iface = iface
        self.params_start_addr = params_start_addr
        self.params = params
        self.delay = delay
        self.exit_addr = exit_addr

    def __call__(self, *args):
        assert len(args) == len(
            self.params
        ), "{}() takes {} arguments but {} were given".format(
            self.__name__, len(self.params), len(args)
        )

        buf = pack_params(self.params, *args)

```

```
193         if len(buf) == 1:
194             self.iface.write(self.params_start_addr, buf[0])
195         else:
196             self.iface.writeb(self.params_start_addr, buf)
197 
198         if self.delay is not None:
199             if self.delay != 0:
200                 time.sleep(self.delay)
201             self.iface.read(self.exit_addr)
202 
203 
204 class ReturnsProc:
205     def __init__(self, iface, returns_start_addr, returns, delay, call_addr):
206         self.iface = iface
207         self.returns_start_addr = returns_start_addr
208         self.delay = delay
209         self.call_addr = call_addr
210 
211         self.buf_iface = _BufferIface()
212         self.buf_size, self.returns = create_mock_returns(
213             self.buf_iface, returns_start_addr, returns
214         )
215 
216     def __call__(self):
217         if self.delay is not None:
218             self.iface.write(self.call_addr, 0)
219             if self.delay != 0:
220                 time.sleep(self.delay)
221 
222         if self.buf_size == 1:
223             buf = [self.iface.read(self.returns_start_addr)]
224         else:
225             buf = self.iface.readb(self.returns_start_addr, self.buf_size)
226 
227         self.buf_iface.set_buf(buf)
228         tup = []  # List to allow append but must be cast to tuple.
229 
230         for ret in self.returns:
231             tup.append(ret['Status'].read())
232 
233         return tuple(tup)
234 
235 
236 class ParamsAndReturnsProc:
237     def __init__(
238         self, iface, params_start_addr, params, returns_start_addr, returns, delay
239     ):
240         self.iface = iface
241 
242         self.params_start_addr = params_start_addr
243         self.params = params
244 
245         self.returns_start_addr = returns_start_addr
246         self.returns_buf_iface = _BufferIface()
247         self.returns_buf_size, self.returns = create_mock_returns(
248             self.returns_buf_iface, returns_start_addr, returns
249         )
250 
251         self.delay = delay
252 
253     def __call__(self, *args):
254         assert len(args) == len(
255             self.params
256         ), "{}() takes {} arguments but {} were given".format(
257             self.__name__, len(self.params), len(args)
258         )
259 
260         params_buf = pack_params(self.params, *args)
```

```
261             if len(params_buf) == 1:
262                 self.iface.write(self.params_start_addr, params_buf[0])
263             else:
264                 self.iface.writeb(self.params_start_addr, params_buf)
265 
266             if self.delay is not None:
267                 if self.delay != 0:
268                     time.sleep(self.delay)
269 
270             if self.returns_buf_size == 1:
271                 returns_buf = [self.iface.read(self.returns_start_addr)]
272             else:
273                 returns_buf = self.iface.readb(
274                     self.returns_start_addr, self.returns_buf_size
275                 )
276             self.returns_buf_iface.set_buf(returns_buf)
277             tup = []  # List to allow append but must be cast to tuple.
278             for ret in self.returns:
279                 tup.append(ret['Status'].read())
280 
281             return tuple(tup)
282 
283 
284     class Static:
285         def __init__(self, value):
286             self._value = value
287 
288         @property
289         def value(self):
290             return self._value
291 
292         @value.setter
293         def value(self, v):
294             raise Exception(f"cannot set value of static element")
295 
296 
297     class StatusSingleOneReg:
298         def __init__(self, iface, addr, start_bit, end_bit):
299             self.iface = iface
300 
301             self.addr = addr
302             self.start_bit = start_bit
303 
304             self.mask = calc_mask((end_bit, start_bit))
305             self.width = end_bit - start_bit + 1
306 
307         def read(self):
308             return (self.iface.read(self.addr) >> self.start_bit) & self.mask
309 
310 
311     class StaticSingleOneReg(Static, StatusSingleOneReg):
312         def __init__(self, iface, addr, start_bit, end_bit, value):
313             Static.__init__(self, value)
314             StatusSingleOneReg.__init__(self, iface, addr, start_bit, end_bit)
315 
316 
317     class ConfigSingleOneReg(StatusSingleOneReg):
318         def __init__(self, iface, addr, start_bit, end_bit):
319             super().__init__(iface, addr, start_bit, end_bit)
320 
321         def write(self, data):
322             assert 0 <= data < 2 ** self.width, "value overrange ({})".format(data)
323             self.iface.write(self.addr, data << self.start_bit)
324 
325 
326     class MaskSingleOneReg(StatusSingleOneReg):
327         def __init__(self, iface, addr, start_bit, end_bit):
328             super().__init__(iface, addr, start_bit, end_bit)
```

```
329
330     def _bits_to_iterable(self, bits):
331         if bits == None:
332             return range(self.width)
333         elif type(bits) == int:
334             return (bits,)
335         return bits
336
337     def _assert_bits_in_range(self, bits):
338         for b in bits:
339             assert 0 <= b < self.width, "mask overrange"
340
341     def _assert_bits_to_update(self, bits):
342         if bits == None:
343             raise Exception("bits to update cannot have None value")
344         if type(bits).__name__ in ["list", "tuple", "range", "set"] and len(bits) == 0:
345             raise Exception("empty " + type(bits) + " of bits to update")
346
347     def set(self, bits=None):
348         bits = self._bits_to_iterable(bits)
349         self._assert_bits_in_range(bits)
350
351         mask = 0
352         for b in bits:
353             mask |= 1 << b
354
355         self.iface.write(self.addr, mask << self.start_bit)
356
357     def clear(self, bits=None):
358         bits = self._bits_to_iterable(bits)
359         self._assert_bits_in_range(bits)
360
361         mask = self.mask
362         for b in bits:
363             mask ^= 1 << b
364
365         self.iface.write(self.addr, mask << self.start_bit)
366
367     def toggle(self, bits=None):
368         bits = self._bits_to_iterable(bits)
369         self._assert_bits_in_range(bits)
370
371         xor_mask = 0
372         for b in bits:
373             xor_mask |= 1 << b
374         xor_mask <<= self.start_bit
375
376         mask = self.iface.read(self.addr) ^ xor_mask
377         self.iface.write(self.addr, mask)
378
379     def update_set(self, bits):
380         self._assert_bits_to_update(bits)
381
382         bits = self._bits_to_iterable(bits)
383         self._assert_bits_in_range(bits)
384
385         mask = 0
386         for b in bits:
387             mask |= 1 << b
388
389         mask = self.iface.read(self.addr) | (mask << self.start_bit)
390         self.iface.write(self.addr, mask)
391
392     def update_clear(self, bits):
393         self._assert_bits_to_update(bits)
394
395         bits = self._bits_to_iterable(bits)
396         self._assert_bits_in_range(bits)
```

```
397
398         mask = 2 ** BUS_WIDTH - 1
399         for b in bits:
400             mask ^= 1 << b
401
402         mask = self.iface.read(self.addr) & (mask << self.start_bit)
403         self.iface.write(self.addr, mask)
404
405
406 class StatusSingleNRegs:
407     def __init__(self, iface, start_addr, reg_count, start_mask, end_mask):
408         self.iface = iface
409         self.addrs = list(range(start_addr, start_addr + reg_count))
410         self.width = 0
411         self.masks = []
412         self.reg_shifts = []
413         self.data_shifts = []
414
415         for i in range(reg_count):
416             if i == 0:
417                 self.masks.append(calc_mask(start_mask))
418                 self.reg_shifts.append(start_mask[1])
419                 self.data_shifts.append(0)
420                 self.width += start_mask[0] - start_mask[1] + 1
421             else:
422                 self.reg_shifts.append(0)
423                 self.data_shifts.append(self.width)
424                 if i == reg_count - 1:
425                     self.masks.append(calc_mask(end_mask))
426                     self.width += end_mask[0] - end_mask[1] + 1
427                 else:
428                     self.masks.append(calc_mask((BUS_WIDTH - 1, 0)))
429                     self.width += BUS_WIDTH
430
431     def read(self):
432         data = 0
433         for i, a in enumerate(self.addrs):
434             data |= (
435                 (self.iface.read(a) >> self.reg_shifts[i]) & self.masks[i]
436             ) << self.data_shifts[i]
437         return data
438
439
440 class ConfigSingleNRegs(StatusSingleNRegs):
441     def __init__(self, iface, start_addr, reg_count, start_mask, end_mask):
442         super().__init__(iface, start_addr, reg_count, start_mask, end_mask)
443
444     def write(self, data):
445         assert 0 <= data < 2 ** self.width, "value overrange ({})".format(data)
446         for i, a in enumerate(self.addrs):
447             self.iface.write(
448                 a, ((data >> self.data_shifts[i]) & self.masks[i]) << self.reg_shifts[i]
449             )
450
451
452 class StaticSingleNRegs(Static, StatusSingleNRegs):
453     def __init__(self, iface, start_addr, reg_count, start_mask, end_mask, value):
454         Static.__init__(self, value)
455         StatusSingleNRegs.__init__(
456             self, iface, start_addr, reg_count, start_mask, end_mask
457         )
458
459
460 class StatusArrayOneReg:
461     def __init__(self, iface, addr, start_bit, width, item_count):
462         self.iface = iface
463         self.addr = addr
464         self.start_bit = start_bit
```

```
465         self.width = width
466         self.item_count = item_count
467 
468     def __len__(self):
469         return self.item_count
470 
471     def read(self, idx=None):
472         reg = self.iface.read(self.addr)
473         mask = (1 << self.width) - 1
474 
475         if type(idx) == int:
476             assert 0 <= idx < self.item_count
477             shift = self.start_bit + self.width * idx
478             return (reg >> shift) & mask
479         elif idx is None:
480             idx = tuple(range(0, self.item_count))
481 
482         for i in idx:
483             assert 0 <= i < self.item_count
484 
485         data = []
486         for i in idx:
487             shift = self.start_bit + self.width * i
488             data.append((reg >> shift) & mask)
489 
490         return data
491 
492 
493 class ConfigArrayOneReg(StatusArrayOneReg):
494     def __init__(self, iface, addr, start_bit, width, item_count):
495         super().__init__(iface, addr, start_bit, width, item_count)
496 
497     def write(self, data, offset=0):
498         """ offset - elements index offset, applied also when data is dictionary """
499         assert 0 <= len(data) <= self.item_count, f"invalid data len {len(data)}"
500 
501         val = 0
502         mask = 0
503 
504         if type(data) == dict:
505             for i, v in data.items():
506                 assert type(i) == int, f'invalid index type {type(i)}'
507                 assert i >= 0, f"negative index {i}"
508                 assert i + offset < self.item_count, f"index overrange {i}"
509                 assert (
510                     0 <= v < 2 ** self.width
511                 ), f"data out of range, index {i}, value {v}"
512                 shift = self.start_bit + (i + offset) * self.width
513                 val |= v << shift
514                 mask |= (2 ** self.width - 1) << shift
515         else:
516             assert len(data) + offset <= self.item_count
517 
518             for i, v in enumerate(data):
519                 assert (
520                     0 <= v < 2 ** self.width
521                 ), f"data out of range, index {i}, value {v}"
522                 shift = self.start_bit + (i + offset) * self.width
523                 val |= v << shift
524                 mask |= 2 ** self.width - 1 << shift
525 
526         if len(data) == self.item_count:
527             self.iface.write(self.addr, val)
528         else:
529             self.iface.rmw(self.addr, val, mask)
530 
531 
532 class StatusArrayOneInReg:
```

```
533     def __init__(self, iface, addr, mask, item_count):
534         self.iface = iface
535         self.addr = addr
536         self.mask = calc_mask(mask)
537         self.shift = mask[1]
538         self.width = mask[0] - mask[1] + 1
539         self.item_count = item_count
540 
541     def __len__(self):
542         return self.item_count
543 
544     def read(self, idx=None):
545         if idx is None:
546             idx = tuple(range(0, self.item_count))
547             if self.item_count == 1:
548                 return (self.iface.read(self.addr) >> self.shift) & self.mask
549             else:
550                 buf = self.iface.readb(self.addr, self.item_count)
551                 return [(data >> self.shift) & self.mask for data in buf]
552         elif type(idx) == int:
553             assert 0 <= idx < self.item_count
554             return (self.iface.read(self.addr + idx) >> self.shift) & self.mask
555         else:
556             for i in idx:
557                 assert 0 <= i < self.item_count
558             return [
559                 (self.iface.read(self.addr + i) >> self.shift) & self.mask for i in idx
560             ]
561 
562 
563 class ConfigArrayOneInReg(StatusArrayOneInReg):
564     def __init__(self, iface, addr, mask, item_count):
565         super().__init__(iface, addr, mask, item_count)
566 
567     def write(self, data, offset=0):
568         """ offset - elements index offset, applied also when data is dictionary """
569         assert 0 <= len(data) <= self.item_count, f"invalid data len {len(data)}"
570 
571         if type(data) == dict:
572             idxs = sorted(data.keys())
573             for idx in idxs:
574                 self.iface.write(self.addr + offset + idx, data[idx] << self.shift)
575         else:
576             assert len(data) + offset <= self.item_count
577 
578             if len(data) == 1:
579                 self.iface.write(self.addr + offset, data[0] << self.shift)
580             else:
581                 buf = []
582                 for d in data:
583                     buf.append(d << self.shift)
584                 self.iface.writeb(self.addr + offset, buf)
585 
586 
587 class StatusArrayNInReg:
588     def __init__(self, iface, addr, start_bit, width, item_count, items_in_reg):
589         self.iface = iface
590         self.addr = addr
591         self.start_bit = start_bit
592         self.width = width
593         self.item_count = item_count
594         self.items_in_reg = items_in_reg
595         self.reg_count = math.ceil(item_count / self.items_in_reg)
596 
597     def __len__(self):
598         return self.item_count
599 
600     def read(self, idx=None):
```

```
601         mask = (1 << self.width) - 1
602
603         if idx is None:
604             idx = tuple(range(0, self.item_count))
605             reg_idx = tuple(range(self.reg_count))
606         elif type(idx) == int:
607             assert 0 <= idx < self.item_count
608             reg_idx = idx // self.items_in_reg
609             shift = self.start_bit + self.width * (idx % self.items_in_reg)
610             return (self.iface.read(self.addr + reg_idx) >> shift) & mask
611         else:
612             reg_idx = set()
613             for i in idx:
614                 assert 0 <= i < self.item_count
615                 reg_idx.add(i // self.items_in_reg)
616
617         reg_data = {reg_i: self.iface.read(self.addr + reg_i) for reg_i in reg_idx}
618
619         data = []
620         for i in idx:
621             shift = self.start_bit + self.width * (i % self.items_in_reg)
622             data.append((reg_data[i // self.items_in_reg] >> shift) & mask)
623
624         return data
625
626
627 class ConfigArrayNInReg(StatusArrayNInReg):
628     def __init__(self, iface, addr, start_bit, width, item_count, items_in_reg):
629         super().__init__(iface, addr, start_bit, width, item_count, items_in_reg)
630
631     def write(self, data, offset=0):
632         """ offset - elements index offset, applied also when data is dictionary """
633         assert 0 <= len(data) <= self.item_count, f"invalid data len {len(data)}"
634
635         regs = dict()
636
637         def add_to_regs(idx, val):
638             idx = idx + offset
639             assert idx <= self.item_count, f"index overrange {idx + offset}"
640             reg_idx = idx // self.items_in_reg
641             if reg_idx not in regs:
642                 regs[reg_idx] = [0, 0]  # [value, mask]
643             shift = self.start_bit + (idx % self.items_in_reg) * self.width
644             regs[reg_idx][0] |= val << shift
645             regs[reg_idx][1] |= (2 ** self.width - 1) << shift
646
647         if type(data) == dict:
648             for idx, val in data.items():
649                 add_to_regs(idx, val)
650         else:
651             for idx, val in enumerate(data):
652                 add_to_regs(idx, val)
653
654         reg_idxs = sorted(regs.keys())
655         for idx in reg_idxs:
656             self.iface.rmw(self.addr + idx, regs[idx][0], regs[idx][1])
657
658
659 class StatusArrayNInRegMInEndReg(StatusArrayNInReg):
660     def __init__(self, iface, addr, start_bit, width, item_count, items_in_reg):
661         super().__init__(iface, addr, start_bit, width, item_count, items_in_reg)
662
663
664 class ConfigArrayNInRegMInEndReg(ConfigArrayNInReg):
665     def __init__(self, iface, addr, start_bit, width, item_count, items_in_reg):
666         super().__init__(iface, addr, start_bit, width, item_count, items_in_reg)
667
668
```

```
class StatusArrayOneInNRegs:
    def __init__(
        self, iface, addr, width, item_count, regs_per_item, reg_count, end_bit
    ):
        self.iface = iface

        self.addr = addr
        self.width = width
        self.item_count = item_count

        self.regs_per_item = regs_per_item
        self.reg_count = reg_count
        self.last_reg_mask = calc_mask((end_bit, 0))

    def __len__(self):
        return self.item_count

    def _regs_to_data(self, buf):
        assert len(buf) == self.regs_per_item
        data = 0
        for i, bite in enumerate(buf):
            if i == len(buf) - 1:
                data |= (bite & self.last_reg_mask) << (i * BUS_WIDTH)
            else:
                data |= bite << (i * BUS_WIDTH)
        return data

    def read(self, idx=None):
        if idx is None:
            buf = self.iface.readb(self.addr, self.reg_count)
            data = []
            for i in range(self.item_count):
                data.append(
                    self._regs_to_data(
                        buf[i * self.regs_per_item : (i + 1) * self.regs_per_item]
                    )
                )
            return data
        elif type(idx) == int:
            assert 0 <= idx < self.item_count
            buf = self.iface.readb(
                self.addr + idx * self.regs_per_item, self.regs_per_item
            )
            return self._regs_to_data(buf)
        else:
            data = []
            for i in idx:
                assert 0 <= i < self.item_count
                buf = self.iface.readb(
                    self.addr + i * self.regs_per_item, self.regs_per_item
                )
                data.append(self._regs_to_data(buf))
            return data


class Upstream:
    def __init__(self, iface, addr, delay, returns):
        self.iface = iface
        self.addr = addr
        self.delay = delay
        self.buf_iface = _BufferIface()
        self.buf_size, self.returns = create_mock_returns(self.buf_iface, addr, returns)

    def read(self, n):
        """
        Read the stream n times.
        Read returns a tuple of tuples.
        Non grouped returns are returned as values within tuple.
```

```
        """
        if self.buf_size == 1:
            read_data = [[x] for x in self.iface.cread(self.addr, n)]
        else:
            read_data = self.iface.creadb(self.addr, self.buf_size, n)

        data = []
        for buf in read_data:
            self.buf_iface.set_buf(buf)
            tup = []  # List to allow append but must be cast to tuple.

            for ret in self.returns:
                tup.append(ret['Status'].read())

            data.append(tuple(tup))

        return tuple(data)


class Downstream:
    def __init__(self, iface, addr, delay, params):
        self.iface = iface
        self.addr = addr
        self.params = params
        self.delay = delay

    def write(self, data):
        wbuf = []  # Write buffer
        args_in_one_reg = False  # All arguments occupy one register

        for args in data:
            assert len(args) == len(
                self.params
            ), f"invalid number of arguments {len(args)}, want {len(self.params)}"

            buf = pack_params(self.params, *args)
            if len(buf) == 1:
                args_in_one_reg = True
                wbuf.append(buf[0])
            else:
                wbuf.append(buf)

        if self.delay is None:
            if args_in_one_reg:
                self.iface.cwrite(self.addr, wbuf)
            else:
                self.iface.cwriteb(self.addr, wbuf)
        else:
            for i, val in enumerate(wbuf):
                if args_in_one_reg:
                    self.iface.write(self.addr, val)
                else:
                    self.iface.writeb(self.addr, buf)

                if i < len(wbuf) - 1:
                    time.sleep(self.delay)


class Main:
    def __init__(self, iface):
        self.iface = iface
        self.Version = StaticSingleOneReg(iface, 8, 0, 23, 0b00000000100000001000000010)
        self.ID = StaticSingleOneReg(
            iface, 0, 0, 31, 0b01100101011001101000011010110111 1
        )
        self.S1 = StatusSingleOneReg(iface, 8, 24, 30)
        self.S2 = StatusSingleOneReg(iface, 5, 9, 17)
        self.S3 = StatusSingleOneReg(iface, 4, 12, 23)
```

```
        self.SA = StatusArrayNInRegMInEndReg(iface, 9, 0, 8, 10, 4)
        self.Counter = StatusSingleNRegs(iface, 12, 2, (31, 0), (0, 0))
        self.C1 = ConfigSingleOneReg(iface, 6, 0, 6)
        self.C2 = ConfigSingleOneReg(iface, 5, 0, 8)
        self.C3 = ConfigSingleOneReg(iface, 4, 0, 11)
        self.CA = ConfigArrayNInReg(iface, 1, 0, 8, 10, 4)
        self.Mask = MaskSingleOneReg(iface, 7, 0, 15)
        self.Subblock = self.SubblockClass(self.iface)

    class SubblockClass:
        def __init__(self, iface):
            self.iface = iface
            self.Add = ParamsAndReturnsProc(
                iface,
                24,
                [
                    { 'Name': 'A', 'Width': 20, 'Access': {
                            'StartAddr': 24,
                            'StartBit': 0,
                            'EndBit': 19,
                            'RegCount': 1,
                            'Type': 'SingleOneReg', }, },
                    { 'Name': 'B', 'Width': 10, 'Access': {
                            'StartAddr': 24,
                            'StartBit': 20,
                            'EndBit': 29,
                            'RegCount': 1,
                            'Type': 'SingleOneReg', }, },
                    { 'Name': 'C', 'Width': 8, 'Access': {
                            'StartAddr': 24,
                            'StartBit': 30,
                            'EndBit': 5,
                            'RegCount': 2,
                            'Type': 'SingleNRegs', }, },
                ],
                25,
                [
                    { 'Name': 'Sum', 'Access': {
                            'StartAddr': 25,
                            'StartBit': 6,
                            'EndBit': 26,
                            'RegCount': 1,
                            'Type': 'SingleOneReg', }, }
                ],
                None,
            )
            self.Add_Stream = Downstream(
                iface,
                26,
                None,
                [
                    { 'Name': 'A', 'Width': 20, 'Access': {
                            'StartAddr': 26,
                            'StartBit': 0,
                            'EndBit': 19,
                            'RegCount': 1,
                            'Type': 'SingleOneReg', }, },
                    {
                        'Name': 'B', 'Width': 10, 'Access': {
                            'StartAddr': 26,
                            'StartBit': 20,
                            'EndBit': 29,
                            'RegCount': 1,
                            'Type': 'SingleOneReg', }, },
                    { 'Name': 'C', 'Width': 8, 'Access': {
                            'StartAddr': 26,
                            'StartBit': 30,
                            'EndBit': 5,
```

```
                                'RegCount': 2,
                                'Type': 'SingleNRegs', }, },
            ],
        )
        self.Sum_Stream = Upstream(
            iface,
            28,
            None,
            [
                { 'Name': 'Sum', 'Access': {
                        'StartAddr': 28,
                        'StartBit': 0,
                        'EndBit': 20,
                        'RegCount': 1,
                        'Type': 'SingleOneReg', }, }
            ],
        )
```

E VHDL Main entity description generated for the example design

```vhdl
-- This file has been automatically generated by the vfbdb tool.
-- Do not edit it manually, unless you really know what you do.
-- https://github.com/Functional-Bus-Description-Language/go-vfbdb
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
library ltypes;
   use ltypes.types.all;
library work;
   use work.wb3.all;
package Main_pkg is
end package;
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
library general_cores;
   use general_cores.wishbone_pkg.all;
library ltypes;
   use ltypes.types.all;
library work;
   use work.wb3.all;
   use work.Main_pkg.all;

entity Main is
generic ( G_REGISTERED : boolean := true );
port (
   clk_i : in std_logic;
   rst_i : in std_logic;
   slave_i : in  t_wishbone_slave_in_array (1 - 1 downto 0);
   slave_o : out t_wishbone_slave_out_array(1 - 1 downto 0);
   Subblock_master_o : out t_wishbone_master_out_array(0 downto 0);
   Subblock_master_i : in  t_wishbone_master_in_array(0 downto 0);
   Version_o : out std_logic_vector(23 downto 0) := x"010102";
   ID_o : out std_logic_vector(31 downto 0) := x"cacd0d6f";
   S1_i : in std_logic_vector(6 downto 0);
   S2_i : in std_logic_vector(8 downto 0);
   S3_i : in std_logic_vector(11 downto 0);
   SA_i : in slv_vector(9 downto 0)(7 downto 0);
   Counter_i : in std_logic_vector(32 downto 0);
   C1_o : buffer std_logic_vector(6 downto 0);
   C2_o : buffer std_logic_vector(8 downto 0);
   C3_o : buffer std_logic_vector(11 downto 0);
   CA_o : buffer slv_vector(9 downto 0)(7 downto 0);
   Mask_o : buffer std_logic_vector(15 downto 0)
);
end entity;
architecture rtl of Main is
constant C_ADDRESSES : t_wishbone_address_array(1 downto 0) :=
   (0 => "00000000000000000000000000000000", 1 => "00000000000000000000000000011000");
constant C_MASKS     : t_wishbone_address_array(1 downto 0) :=
   (0 => "00000000000000000000000000010000", 1 => "00000000000000000000000000011000");
signal master_out : t_wishbone_master_out;
signal master_in  : t_wishbone_master_in;
signal Counter_atomic : std_logic_vector(32 downto 32);
begin
crossbar: entity general_cores.xwb_crossbar
```

```
generic map (
   G_NUM_MASTERS => 1,
   G_NUM_SLAVES  => 1 + 1,
   G_REGISTERED  => G_REGISTERED,
   G_ADDRESS     => C_ADDRESSES,
   G_MASK        => C_MASKS
) port map (
   clk_sys_i   => clk_i,
   rst_n_i     => not rst_i,
   slave_i     => slave_i,
   slave_o     => slave_o,
   master_i(0) => master_in,
   master_i(1) => Subblock_master_i(0),
   master_o(0) => master_out,
   master_o(1) => Subblock_master_o(0)
);
register_access : process (clk_i) is
variable addr : natural range 0 to 14 - 1;
begin
if rising_edge(clk_i) then
-- Normal operation.
master_in.rty <= '0';
master_in.ack <= '0';
master_in.err <= '0';
transfer : if
   master_out.cyc = '1'
   and master_out.stb = '1'
   and master_in.err = '0'
   and master_in.rty = '0'
   and master_in.ack = '0'
then
   addr := to_integer(unsigned(master_out.adr(4 - 1 downto 0)));
   -- First assume there is some kind of error.
   -- For example internal address is invalid or there is a try to write status.
   master_in.err <= '1';
   -- '0' for security reasons, '-' can lead to the information leak.
   master_in.dat <= (others => '0');
   master_in.ack <= '0';
   -- Registers Access
   if 0 <= addr and addr <= 0 then
      master_in.dat(31 downto 0) <= x"cacd0d6f"; -- ID
      master_in.ack <= '1';
      master_in.err <= '0';
   end if;
   if 1 <= addr and addr <= 2 then
      for i in 0 to 3 loop
         if master_out.we = '1' then
            CA_o((addr-1)*4+i) <= master_out.dat(8*(i+1) + 0-1 downto 8*i + 0);
         end if;
         master_in.dat(8*(i+1) + 0-1 downto 8*i + 0) <= CA_o((addr-1)*4+i);
      end loop;
      master_in.ack <= '1';
      master_in.err <= '0';
   end if;
   if 3 <= addr and addr <= 3 then
      for i in 0 to 1 loop
         if master_out.we = '1' then
            CA_o(8+i) <= master_out.dat(8*(i+1) + 0-1 downto 8*i+0);
         end if;
         master_in.dat(8*(i+1) + 0-1 downto 8*i+0) <= CA_o(8+i);
      end loop;
      master_in.ack <= '1';
      master_in.err <= '0';
   end if;
   if 4 <= addr and addr <= 4 then
      master_in.dat(23 downto 12) <= S3_i;
      if master_out.we = '1' then
         C3_o <= master_out.dat(11 downto 0);
```

```
125         end if;
126         master_in.dat(11 downto 0) <= C3_o;
127         master_in.ack <= '1';
128         master_in.err <= '0';
129      end if;
130      if 5 <= addr and addr <= 5 then
131         master_in.dat(17 downto 9) <= S2_i;
132         if master_out.we = '1' then
133            C2_o <= master_out.dat(8 downto 0);
134         end if;
135         master_in.dat(8 downto 0) <= C2_o;
136         master_in.ack <= '1';
137         master_in.err <= '0';
138      end if;
139      if 6 <= addr and addr <= 6 then
140         if master_out.we = '1' then
141            C1_o <= master_out.dat(6 downto 0);
142         end if;
143         master_in.dat(6 downto 0) <= C1_o;
144         master_in.ack <= '1';
145         master_in.err <= '0';
146      end if;
147      if 7 <= addr and addr <= 7 then
148         if master_out.we = '1' then
149            Mask_o <= master_out.dat(15 downto 0);
150         end if;
151         master_in.dat(15 downto 0) <= Mask_o;
152         master_in.ack <= '1';
153         master_in.err <= '0';
154      end if;
155      if 8 <= addr and addr <= 8 then
156         master_in.dat(23 downto 0) <= x"010102"; -- Version
157         master_in.dat(30 downto 24) <= S1_i;
158         master_in.ack <= '1';
159         master_in.err <= '0';
160      end if;
161      if 9 <= addr and addr <= 10 then
162         for i in 0 to 3 loop
163            master_in.dat(8*(i+1) + 0-1 downto 8*i + 0) <= SA_i((addr-9)*4+i);
164         end loop;
165         master_in.ack <= '1';
166         master_in.err <= '0';
167      end if;
168      if 11 <= addr and addr <= 11 then
169         for i in 0 to 1 loop
170            master_in.dat(8*(i+1) + 0-1 downto 8*i+0) <= SA_i(8+i);
171         end loop;
172         master_in.ack <= '1';
173         master_in.err <= '0';
174      end if;
175      if 12 <= addr and addr <= 12 then
176         Counter_atomic(32 downto 32) <= Counter_i(32 downto 32);
177         master_in.dat(31 downto 0) <= Counter_i(31 downto 0);
178         master_in.ack <= '1';
179         master_in.err <= '0';
180      end if;
181      if 13 <= addr and addr <= 13 then
182         master_in.dat(0 downto 0) <= Counter_atomic(32 downto 32);
183         master_in.ack <= '1';
184         master_in.err <= '0';
185      end if;
186   end if transfer;
187   if rst_i = '1' then
188      master_in <= C_DUMMY_WB_MASTER_IN;
189   end if;
190   end if;
191   end process register_access;
192   end architecture;
```

F VHDL Subblock entity description generated for the example design

```
1  -- This file has been automatically generated by the vfbdb tool.
2  -- Do not edit it manually, unless you really know what you do.
3  -- https://github.com/Functional-Bus-Description-Language/go-vfbdb
4  library ieee;
5     use ieee.std_logic_1164.all;
6     use ieee.numeric_std.all;
7  library ltypes;
8     use ltypes.types.all;
9  library work;
10    use work.wb3.all;
11
12 package Subblock_pkg is
13 type Add_out_t is record
14    A : std_logic_vector(19 downto 0);
15    B : std_logic_vector(9 downto 0);
16    C : std_logic_vector(7 downto 0);
17    call : std_logic;
18    exitt : std_logic;
19 end record;
20 type Add_in_t is record
21    Sum : std_logic_vector(20 downto 0);
22 end record;
23 type Add_Stream_t is record
24    A : std_logic_vector(19 downto 0);
25    B : std_logic_vector(9 downto 0);
26    C : std_logic_vector(7 downto 0);
27 end record;
28 type Sum_Stream_t is record
29    Sum : std_logic_vector(20 downto 0);
30 end record;
31 end package;
32
33 library ieee;
34    use ieee.std_logic_1164.all;
35    use ieee.numeric_std.all;
36 library general_cores;
37    use general_cores.wishbone_pkg.all;
38 library ltypes;
39    use ltypes.types.all;
40 library work;
41    use work.wb3.all;
42    use work.Subblock_pkg.all;
43
44 entity Subblock is
45 generic ( G_REGISTERED : boolean := true );
46 port (
47    clk_i : in std_logic;
48    rst_i : in std_logic;
49    slave_i : in  t_wishbone_slave_in_array (1 - 1 downto 0);
50    slave_o : out t_wishbone_slave_out_array(1 - 1 downto 0);
51    Add_o : out Add_out_t;
52    Add_i : in Add_in_t;
53    Add_Stream_o : out Add_Stream_t;
54    Add_Stream_stb_o : out std_logic;
55    Sum_Stream_i : in Sum_Stream_t;
56    Sum_Stream_stb_o : out std_logic
```

```
57  );
58  end entity;
59  architecture rtl of Subblock is
60  constant C_ADDRESSES : t_wishbone_address_array(0 downto 0) :=
61     (0 => "00000000000000000000000000000000");
62  constant C_MASKS     : t_wishbone_address_array(0 downto 0) :=
63     (0 => "00000000000000000000000000000000");
64  signal master_out : t_wishbone_master_out;
65  signal master_in  : t_wishbone_master_in;
66  begin
67  crossbar: entity general_cores.xwb_crossbar
68  generic map (
69     G_NUM_MASTERS => 1,
70     G_NUM_SLAVES  => 0 + 1,
71     G_REGISTERED  => G_REGISTERED,
72     G_ADDRESS     => C_ADDRESSES,
73     G_MASK        => C_MASKS
74  ) port map (
75     clk_sys_i   => clk_i,
76     rst_n_i     => not rst_i,
77     slave_i     => slave_i,
78     slave_o     => slave_o,
79     master_i(0) => master_in,
80     master_o(0) => master_out
81  );
82
83  register_access : process (clk_i) is
84  variable addr : natural range 0 to 5 - 1;
85  begin
86  if rising_edge(clk_i) then
87
88  -- Normal operation.
89  master_in.rty <= '0';
90  master_in.ack <= '0';
91  master_in.err <= '0';
92
93  -- Procs Calls Clear
94  Add_o.call <= '0';
95  -- Procs Exits Clear
96  Add_o.exitt <= '0';
97  -- Stream Strobes Clear
98  Add_Stream_stb_o <= '0';
99  Sum_Stream_stb_o <= '0';
100
101 transfer : if
102    master_out.cyc = '1'
103    and master_out.stb = '1'
104    and master_in.err = '0'
105    and master_in.rty = '0'
106    and master_in.ack = '0'
107 then
108    addr := to_integer(unsigned(master_out.adr(3 - 1 downto 0)));
109    -- First assume there is some kind of error.
110    -- For example internal address is invalid or there is a try to write status.
111    master_in.err <= '1';
112    -- '0' for security reasons, '-' can lead to the information leak.
113    master_in.dat <= (others => '0');
114    master_in.ack <= '0';
115    -- Registers Access
116    if 0 <= addr and addr <= 0 then
117       if master_out.we = '1' then
118          Add_o.A <= master_out.dat(19 downto 0);
119       end if;
120       master_in.dat(19 downto 0) <= Add_o.A;
121       if master_out.we = '1' then
122          Add_o.B <= master_out.dat(29 downto 20);
123       end if;
124       master_in.dat(29 downto 20) <= Add_o.B;
```

```
125         if master_out.we = '1' then
126            Add_o.C(1 downto 0) <= master_out.dat(31 downto 30);
127         end if;
128         master_in.dat(31 downto 30) <= Add_o.C(1 downto 0);
129         master_in.ack <= '1';
130         master_in.err <= '0';
131      end if;
132      if 1 <= addr and addr <= 1 then
133         if master_out.we = '1' then
134            Add_o.C(7 downto 2) <= master_out.dat(5 downto 0);
135         end if;
136         master_in.dat(5 downto 0) <= Add_o.C(7 downto 2);
137         master_in.dat(26 downto 6) <= Add_i.Sum;
138         master_in.ack <= '1';
139         master_in.err <= '0';
140      end if;
141      if 2 <= addr and addr <= 2 then
142         if master_out.we = '1' then
143            Add_Stream_o.A <= master_out.dat(19 downto 0);
144         end if;
145         if master_out.we = '1' then
146            Add_Stream_o.B <= master_out.dat(29 downto 20);
147         end if;
148         if master_out.we = '1' then
149            Add_Stream_o.C(1 downto 0) <= master_out.dat(31 downto 30);
150         end if;
151         master_in.ack <= '1';
152         master_in.err <= '0';
153      end if;
154      if 3 <= addr and addr <= 3 then
155         if master_out.we = '1' then
156            Add_Stream_o.C(7 downto 2) <= master_out.dat(5 downto 0);
157         end if;
158         master_in.ack <= '1';
159         master_in.err <= '0';
160      end if;
161      if 4 <= addr and addr <= 4 then
162         master_in.dat(20 downto 0) <= Sum_Stream_i.Sum;
163         master_in.ack <= '1';
164         master_in.err <= '0';
165      end if;
166      Add_call : if addr = 1 then
167         if master_out.we = '1' then
168            Add_o.call <= '1';
169         end if;
170      end if;
171      Add_exit : if addr = 1 then
172         if master_out.we = '0' then
173            Add_o.exitt <= '1';
174         end if;
175      end if;
176      Add_Stream_stb : if addr = 3 then
177         if master_out.we = '1' then
178            Add_Stream_stb_o <= '1';
179         end if;
180      end if;
181      Sum_Stream_stb : if addr = 4 then
182         if master_out.we = '0' then
183            Sum_Stream_stb_o <= '1';
184         end if;
185      end if;
186   end if transfer;
187   if rst_i = '1' then
188      master_in <= C_DUMMY_WB_MASTER_IN;
189   end if;
190   end if;
191   end process register_access;
192   end architecture;
```

G 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

H FBDL Specification

Functional Bus Description Language

Revision 1.0

26 January 2024

Abstract

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

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

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

array_marker ::= [ expression ]

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, config, const, doc, false, import, init-value, irq, mask, memory, param, proc, range, reset, read-value, reset-value, return, static, stream, true, type, in-trigger, out-**

trigger.

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

4.5. Literals

4.5.1. Bool literals

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

4.5.2. Number literals

```
underscore ::= _

zero_digit ::= 0

non_zero_decimal_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

decimal_digit ::= zero_digit | non_zero_decimal_digit

binary_base ::= 0B | 0b

binary_digit ::= 0 | 1

octal_base ::= 0O | 0o

octal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hex_base ::= 0X | 0x

hex_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
    A | a | B | b | C | c | D | d | E | e | F | f
```

4.5.3. Integer literals

```
integer_literal ::=
    binary_literal |
    octal_literal |
    decimal_literal |
    hex_literal

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

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

decimal_literal ::= non_zero_decimal_digit {[underscore] decimal_digit}

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

4.5.4. Real literals

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

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

4.5.5. String literals

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

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

4.5.6. Bit string literals

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

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

The meta characters have following meaning:

- '-' - don't care,
- 'U' - uninitialized,
- 'W' - weak unknown,
- 'X' - 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

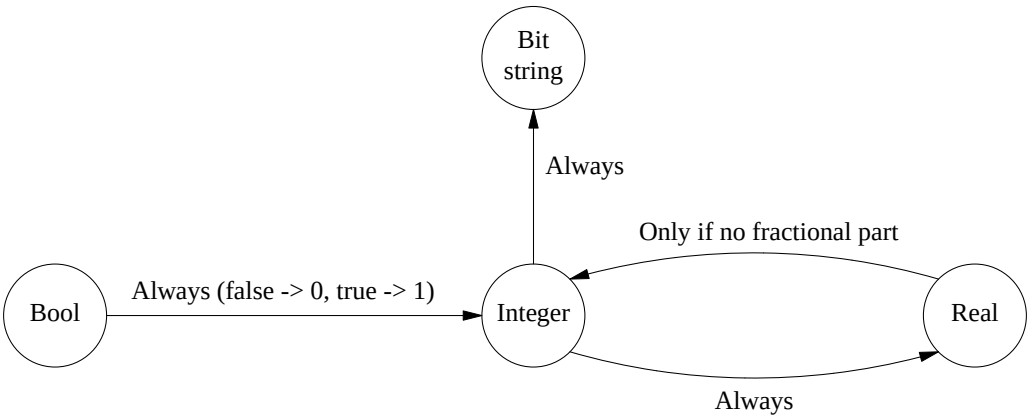
There are 6 data types in FBDL:

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

7.1. Blackbox

The `blackbox` functionality is used to incorporate blocks implemented manually or generated by external tools. For example, a user may want to explicitly manage some particular registers' layouts. In such a case, a register-centric tool might be used, and the generated block can be incorporated into the wrapping functionality-centric description using the `blackbox` functionality.

The `blackbox` functionality has following properties:

`size` `integer (obligatory)`
: The `size` property defines size of the `blackbox` in the number of words with width equal to the `width` property value of the block in which `blackbox` is defined.

The code generated for the requester should not provide any means for accessing the `blackbox`. The code generated for the provider must provide a means to connect the `blackbox` to the remaining part of the bus generated by an FBDL compiler.

7.2. 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:

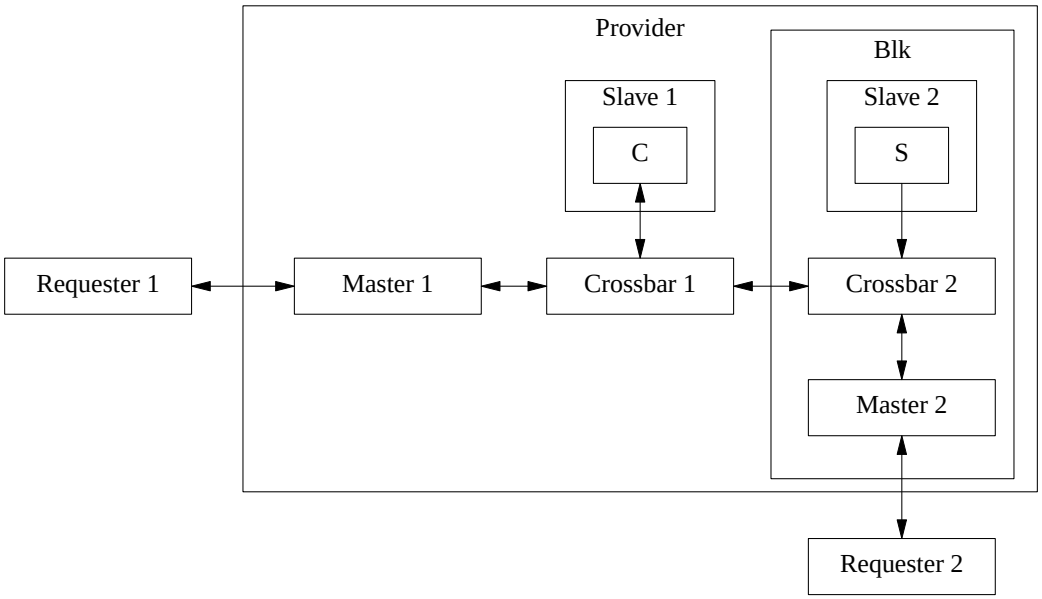
`masters` `integer (1)`
: The `masters` property defines the number of `block` masters.

`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`.

7.3. 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.

7.4. 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.

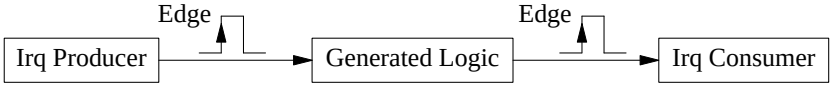
`width` `integer (bus width)`
: The `width` property defines the bit width of the config. The `width` property shall not be explicitly set if the `range` property is already set.

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

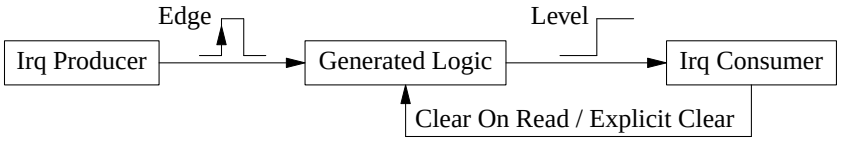
7.5. Irq

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`):

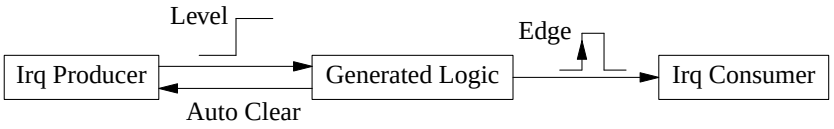
1) edge producer and edge sensitive consumer,



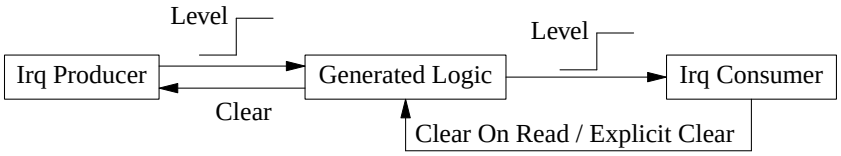
2) edge producer and level sensitive consumer,



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.

7.6. 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.

7.7. 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.

7.8. 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 onparam 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
```

7.9. Proc

The proc functionality represents a procedure called by the requester and carried out by the provider. Theproc 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.

7.10. 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
```

7.11. 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
```

7.12. 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`.

7.13. Stream

The `stream` functionality represents a stream of data to a provider (downstream), or a stream of data from a provider (upstream). An empty stream (stream without any `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 ]
    [ array_marker ]
    declared_identifier | qualified identifier
    [ argument_list ]
    semicolon_and_property_assignments | newline

multi_line_type_definition ::=
    type
    identifier
    [ parameter_list ]
    [ array_marker ]
    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 ||
-------------------------  -------------------------
```

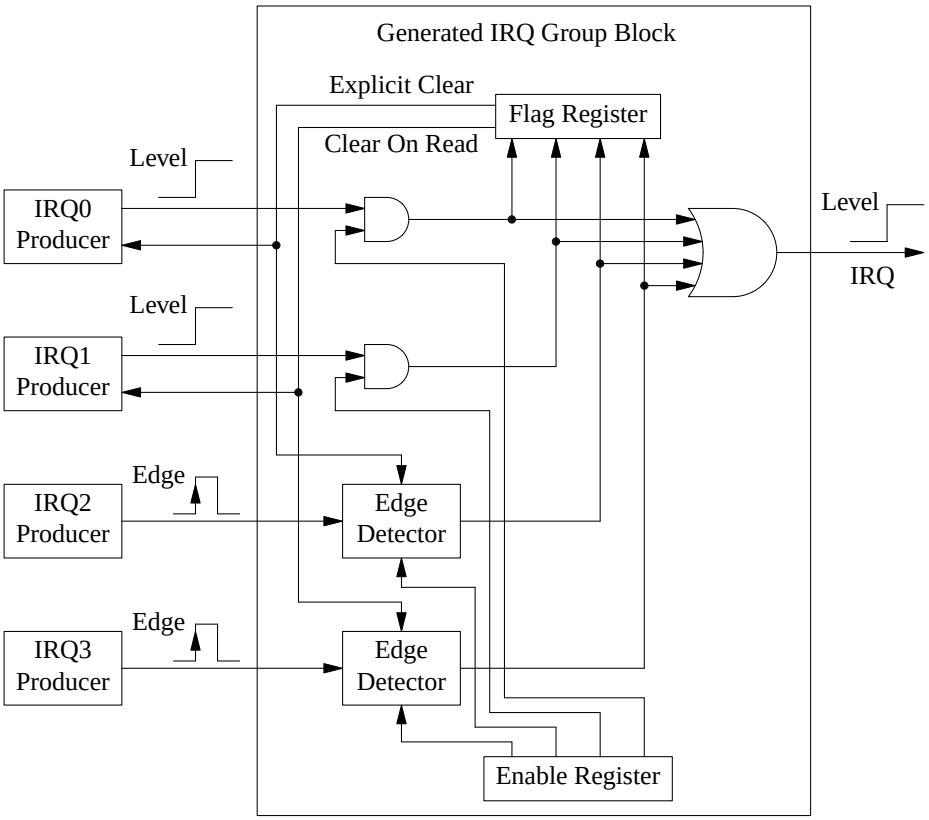
10.7. Irq groups

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"
```