 doi:10.20944/preprints202110.0098.v1

Article

Control and Diagnostics System Generator for Complex FPGA-based Measurement Systems

Wojciech M. Zabołotny [1,*], Marek Gumiński [1], Michał Kruszewski [1] and Walter F.J. Müller [2]

[1] Warsaw University of Technology, Faculty of Electronics and Information Technology, Institute of Electronic Systems, Nowowiejska 15/19, 00-65 Warszawa, Poland
[2] GSI-Helmholtzzentrum für Schwerionenforschung GmbH, Planckstraße 1, 64291 Darmstadt, Germany
* Correspondence: wzab@ise.pw.edu.pl

Abstract: FPGA-based data acquisition and processing systems play an important role in modern high-speed, multichannel measurement systems, especially in High-Energy and Plasma Physics. Such FPGA-based systems require an extended control and diagnostics part corresponding to the complexity of the controlled system. Managing the complex structure of registers while keeping the tight coupling between hardware and software is a tedious and potentially error-prone process. Various existing solutions aimed at helping that task do not perfectly match all specific requirements of that application area. The paper presents a new solution based on the XML system description, facilitating the automated generation of the control system's HDL code and software components and enabling easy integration with the control software. The emphasis is put on reusability, ease of maintenance in case of system modification, easy detection of mistakes, and the possibility of use in modern FPGAs. The presented system has been successfully used in data acquisition and preprocessing projects in High-Energy Physics experiments. It enables easy creation and modification of the control system definition and convenient access to the control and diagnostic blocks. The presented system is an open-source solution and may be adopted by the user for particular needs.

Keywords: FPGA, Wishbone, Control interface, VHDL, System management, System diagnostics

1. Introduction

Modern measurement systems, especially those used in high-energy physics or plasma physics experiments, require complex data acquisition and concentration systems to collect data from multiple input channels efficiently. They often require high-speed data processing to reduce the volume of the received data stream, such as via selection of interesting events, aggregation, or compression [1–3]. Those systems typically use sensors connected via frontend electronics boards (FEBs) using specialized high-speed interfaces [4–6]. Additionally, perfect synchronization of data streams received in different channels is required [7]. The programmable devices - Field Programmable Gate Arrays (FPGA) are usually used to provide those functionalities [8–14]. Their big advantage is high flexibility, enabling significant changes to the communication protocols or data processing algorithms without modifying the underlying hardware.

1.1. General properties of FPGA-implemented data acquisition and preprocessing systems

Such complex systems must be implemented in a modular way where standard and well-tested basic blocks are used to implement more advanced functionalities. This pattern is repeated over the functional levels, resulting in a multilayer, hierarchical block structure.

Reuse of standard blocks in various systems and for various purposes imposes their parameterization. The complexity of the design may change during development and debugging. For example, a simpler version with a limited number of input channels

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

may be used at the beginning of the development to help to detect, isolate and fix bugs. Therefore, the number and structure of higher-level blocks should also be parameterized.

The data processed by such a measurement system may have a complex structure. The HDL language used for implementation should be capable of handling such data efficiently. They can be described with VHDL records [15] or with SystemVerilog structures. However, VHDL seems to be better suited for the implementation of such complex and parameterized systems. Its strict type checking provides good mistakes and errors detection when the system is modified.

To summarize, the data acquisition and preprocessing system is a complex, parameterized, multilayer hierarchical design, significantly relying on advanced features of the VHDL language. The main functionalities, data collection and preprocessing, are usually implemented as a pipelined datapath, optimized for maximal throughput and minimal latency.

Except for the datapath, such a system also requires an efficient control and diagnostics layer. Its purpose is to configure the data processing part, control it, and receive information about its operation (for example, the warnings and errors, the performance metrics, etc.).

The Control and Diagnostics (C&D) system is usually decoupled from the datapath[1] and uses a separate control bus to access the registers.

The read and write operations on the registers are performed by the software running on a CPU[2]. Therefore, an essential feature of the C&D-system is to provide convenient access from the software to selected control and diagnostics registers. The control and diagnostic operations usually use random access to the registers using read, write, and read-modify-write operations. Long block transfers are used relatively rarely.

The typical structure of such an FPGA-implemented measurement system is shown in Figure 1.

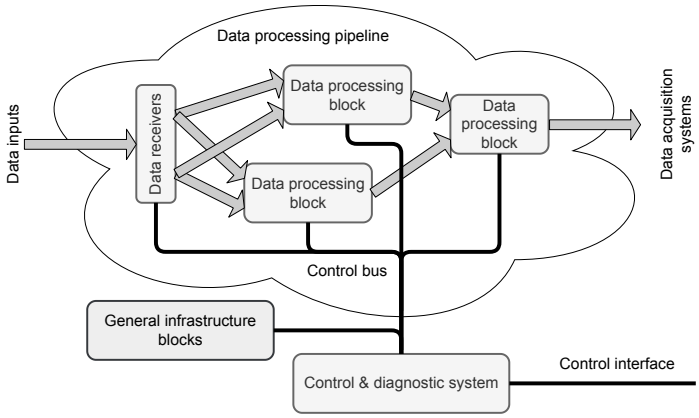

Figure 1. The general block diagram of the data acquisition and preprocessing system implemented in FPGA. The acquired data are concentrated and preprocessed in the pipelined datapath before sending to the central data acquisition (DAQ) system. The Control and Diagnostics (C&D) system uses an independent bus to access the registers located in different parts of the data processing pipeline and general infrastructure blocks.

1.2. Requirements for the Control and Diagnostics (C&D) system

The described properties of the data acquisition and preprocessing systems are the basis for formulating requirements for the C&D-system.

[1] Certain information about the system's operation and processing parameters may also be added to the output data stream as metadata. That may help in further data analysis.

[2] The CPU may be implemented inside the FPGA (so-called soft CPU, like Xilinx Microblaze or Forth CPU - J1B) or an external device communicating with FPGA via an appropriate interface.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

1.2.1. Selection of the control bus

Currently, the most popular internal buses for FPGA designs are the Wishbone [16] and AXI4 [17]. The selection of the control bus depends on multiple criteria and must consider the specific requirements of the control and diagnostic system. Wishbone is a relatively old standard – version B3 [18] was released in 2002, and version B4 [19] in 2010. It is open and is very popular in the open-source world. There are many implementations of Wishbone masters and slaves [20–22]. It offers various modes of operation, where the "classic standard single read/write mode" enables very simple implementation of the slaves. The Wishbone bus is also compatible with the IPbus [23] solution, used to control FPGA-based boards via Ethernet.

The AXI4 bus offers excellent block transfer performance but requires a more complex implementation of the bus slaves [24]. It also offers a simplified "AXI-Lite" [25] version suited explicitly for accessing the memory-mapped registers, but it is still more complex than Wishbone [26].

Because the control and diagnostics system mainly utilizes random accesses to the registers, and the resource use by the C&D-system should be minimized, the Wishbone bus seems to be the right solution. That decision is also supported by the broad availability of Wishbone-compatible slave blocks. The 32-bit width of data and address buses should be sufficient. The synthesis tools should optimize unused address lines if the C&D-system utilizes a smaller subset of the address space.

1.2.2. Requirements for the registers

Exchange of the control and diagnostic information is achieved by reading and writing registers. Two main types of registers may be defined: the control registers (available for reading and writing) and the status registers (that can be only read)[3].

If not specified otherwise, the registers have a width (number of bits) equal to the width of the data bus in the control system. However, if the control parameter or the status value to be transferred via the register has a smaller width, it is desirable that the width of the register could be limited accordingly.

On the other hand, sometimes, it is useful to combine multiple low-width control parameters or status values into a single register. That gives a possibility to read or write them simultaneously and also reduces resource consumption. For that purpose, the C&D-system should support "bitfields" described by their width and position of their least significant bit in the register.

1.2.3. Support for the hierarchical parameterized design

As described in section 1.1, the data acquisition and processing system usually has a multilayer hierarchical structure. That affects the requirements for the control system. The hierarchy may be extended horizontally - there may be multiple identical functional blocks or multiple identical control or status registers. The C&D-system should support vectors of blocks and vectors of registers.

The multilayer hierarchy assumes that the blocks may be nested, and the C&D-system system should also reflect it. In that case, the connections to the nested blocks must be easy to create and maintain.

The parameterization functionality described in section 1.1 requires that the lengths of the mentioned vectors and the presence of the particular blocks and registers should be defined by modifiable parameters.

[3] In fact, it could be possible to implement also "write-only" registers. They should be backed by the "shadow registers" implemented in the software and holding the last written value. The usage of "write-only" registers may further limit resource consumption. However, they do not allow verification of the written value – a helpful feature in the debugging mode. Therefore, we decided not to implement them.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

1.3. Need for C&D-system generator

The structure of the control and diagnostic system is tightly coupled with the structure of the measurement system. That means that usually, it must be created for the particular system and evolve together with it.

An address in the address space of the control bus must be assigned to all registers. To simplify the addressing, the vectors of registers should occupy the contiguous areas in the address space.

Similarly, each block must have assigned its private area in the bus address space, and the blocks belonging to a vector should occupy the consecutive areas.

The proper alignment of blocks simplifies the address decoding and enables reusing resources between address decoders. In the optimal solution, the base address of each block should be aligned to the 2^N boundary, where 2^N is the smallest power of 2 that may fit all the addresses belonging to the block. Such alignment allows connecting the block's internal address decoder only to address bits 0 to $N-1$.

The above means that the **address map** of the C&D-system may require modification after each change of the system's structure or even after changing the parameters describing that structure. Therefore, an automated system for address assignment is necessary.

The assigned addresses must also be somehow passed to the C&D software. So we need a **C&D-system generator** capable of assigning the addresses and generating the address map in a format legible for hardware synthesis tools and software environments.

It is also desirable that the generator implements the registers and the control bus infrastructure, minimizing the effort needed to integrate the generated C&D-system with the rest of the data acquisition and processing system.

2. Possible existing solutions for C&D-system generation

Generation of C&D-systems for FPGA-implemented systems is not a new problem. It has been investigated for many years, and many such solutions have been developed.

2.1. SystemRDL

The most advanced system related to the generation of the C&D-systems is SystemRDL [27]. The SystemRDL is a language aimed at the detailed description of the registers. It tries to cover all possible aspects of register structure and behavior, including descriptions of an arbitrarily complex hierarchy of blocks and registers. The last version supports the parameterization of components and the structure of the system. SystemRDL is well designed and mature but also very complex. When generating a C&D-system, it is necessary to use a special tool to translate the SystemRDL description into the output format - the HDL implementation or software source supporting the communication. Unfortunately, currently, there are only a few such tools available. Agnisys offers a commercial solution [28], but it is closed-source and cannot be modified by the user. There is an open-source **ordt** from Juniper [29], but it does not generate VHDL code. It does not support the Wishbone bus either. There is a whole set of SystemRDL related tools developed in the GitLab repository [30]. Unfortunately, up to now, there is no tool capable of generating the VHDL output.

2.2. Internal Interface and Component Internal Interface

The Internal Interface (II) was developed by Krzysztof Pozniak and others for electronic systems created for CMS and DESY [31–33]. It has been later extended with object functionalities, forming the Component Internal Interface (CII) [34,35]. II initially was using a VME-like interface as a local FPGA bus. In the CII version, it has been supplemented with a possibility to control the Wishbone bus. The CII-implemented C&D-systems may work with software written in C++, Java, and Matlab. II/CII supports complex data structures, like arrays of arbitrary lengths, vectors of bits of arbitrary lengths, etc. However, that flexibility has its price: a high complexity of the interface,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

significant resource consumption, and lower maximum clock frequency. Another disadvantage of the CII is the high complexity of the description of the registers and the fact that it is a closed solution that cannot be used in Open Source projects.

2.3. Address generator for IPbus

During the development of the Data Processing Boards (DPB) for the CBM experiment [36], our team faced the situation where manual allocation of register addresses for IPbus-connected C&D-system became inefficient due to increase of complexity of the developed firmware.

The **addr_gen** [37,38] system was proposed to cure that problem. It accepts the description of the blocks and registers in Python language. It also supports the hierarchy of blocks and vectors of registers and blocks.

The system's output is two VHDL packages defining the constants with parameters and a complex VHDL record with addresses of particular blocks, subblocks, and registers. Integration with software is supported by generating a Python dictionary with the list of assigned addresses and, for C++, the IPbus XML Address Map [39]. Comparing to the requirements formulated in section 1.2, it has the following deficiencies:

- It does not support bitfields.
- It does not generate the HDL code for accessing the registers.

The user's responsibility is to prepare the HDL code providing the read and write access to the registers. The practical usage of **addr_gen** in a few versions of the DPB firmware has exposed additional deficiencies:

- Assigning subsequent addresses to the registers and blocks without alignment resulted in suboptimal address decoders. That increased resource consumption and the critical path length, resulting in lowering of maximum bus clock frequency.
- Handling all blocks and registers in a single component appeared to be extremely inconvenient. Routing of signals connected to those registers between the blocks and through the multiple hierarchy levels was messy and error-prone. Based on that, it was stated that registers should be located near to the place where the connected signals are used. Therefore the control bus infrastructure should be distributed between the blocks.

2.4. Wishbone slave generator

One of the simplest C&D-system generators is the **wbgen2** application known as the **Wishbone slave generator** [40]. It supports the Wishbone local bus. The slave description is prepared in the C-like format and may contain registers, memory blocks, and FIFOs. The **wbgen2** generates the slave HDL code in VHDL or Verilog and C headers for integration with the software. Additionally, it may generate the documentation for the created slave in LaTeX, Texinfo, or HTML. Unfortunately, it does not support vectors of registers or blocks, neither nested blocks. The **wbgen2** is a free and Open Source project. In fact, it was an inspiration for the development of the system described in this paper.

2.5. Other Open Source tools

Other Open Source tools include:

- **Opentitan register tool** [41], which unfortunately does not generate VHDL;
- **hdlregs** [42], which supports VHDL generation and even AXI4-Lite converter, but does not support a hierarchy of blocks;
- **rggen** [43], which has a limited support to hierarchical designs (only one level) and does not support Wishbone (only APB and AXI4-Lite buses);
- **rgen** [44], written in old Python 2.7 and supporting only IP-XACT input format; supports Wishbone, but only the old B3 version;

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

- **cheby** [45], which supports VHDL output, Wishbone bus, and hierarchy but does not support parameterization of the description.

2.6. Final decision

The review of existing solutions leads to the statement that none of them may be used directly to implement the system fulfilling the formulated requirements. Additionally, the introductory review of the code has shown that adding the needed functionality may be difficult.

Therefore, the development of the new one was started[4].

3. Development of the C&D-system generator

The main task of the proposed C&D-system is the allocation of addresses for registers connected to the Wishbone bus, and the generation of appropriate HDL and software sources. Therefore the system was named "**A**ddress **G**enerator for **W**ish**b**one", in short **AGWB**. The version of the system from June 2019 is described in [46]. Since this time, the system has been significantly improved based on experiences from its use.

3.1. The C&D-system hardware structure

The general block structure of the C&D-system based on the formulated requirements is shown in Figure 2.

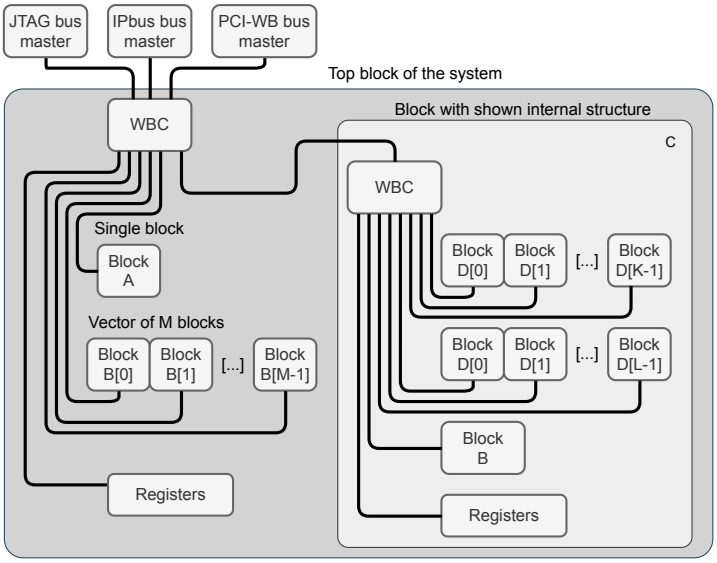

Figure 2. The simplified block diagram of the AGWB-based C&D-system in the FPGA. The "WBC" blocks are Wishbone crossbars.

The core part is the Wishbone crossbar[5]. The crossbar may be controlled by a single or multiple masters (see section 3.2.1). The crossbar provides multiple child node buses – one for each connected slave. Each child node bus handles a certain exclusive segment of the overall address space. Allocation of those segments is handled by the address allocation algorithm, described in section 3.3. The organization of the child node buses is shown in Figure 3. The implementation heavily relies on the VHDL records. Each Wishbone bus consists of two records – the first one for signals transmitted from master to slave and the second one for signals transmitted from slave to master. Another VHDL type is created to describe a vector of Wishbone buses.

[4] Please note that the development of AGWB was started in 2018, while the above review was done now. In fact some of the listed tools were much less mature in 2018.

[5] The **xwb_crossbar** component from the General cores [22] library is used as the crossbar in each block.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

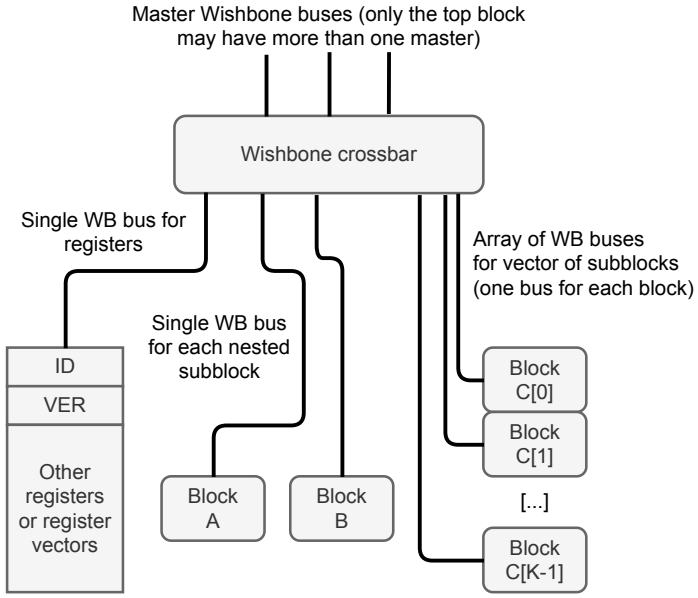

Figure 3. Allocation of Wishbone buses for registers and child nodes (subblocks).

A special case is the Wishbone bus controlling the registers. It is handled by an automatically generated process supporting reading and writing registers. The registers are always located at the beginning of the address segment occupied by the block[6]. . The first two registers are always present and have a special meaning. They allow the software to verify if it communicates with the right C&D-system in the correct version. The first register is always the block ID [7]. The second register is always the block version (VER) [8]. Other registers are placed after the VER register[9].

The described functionalities are implemented in the automatically generated "AGWB local node" (ALN) shown in Figure 4.

[6] It is possible to reserve certain area below registers, using the **reserved** attribute described in section 3.2.2

[7] The value of block ID is calculated as a CRC32 value of its name.

[8] The value of VER is calculated as a CRC32 value of the combined XML configuration file used to generate the C&D-system. In designs using **variants**, the VER value for each block may be different (see section 3.6).

[9] It is possible to place other special registers at addresses 4–7, using the **testdev_ena** attribute described in section 3.2.2.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

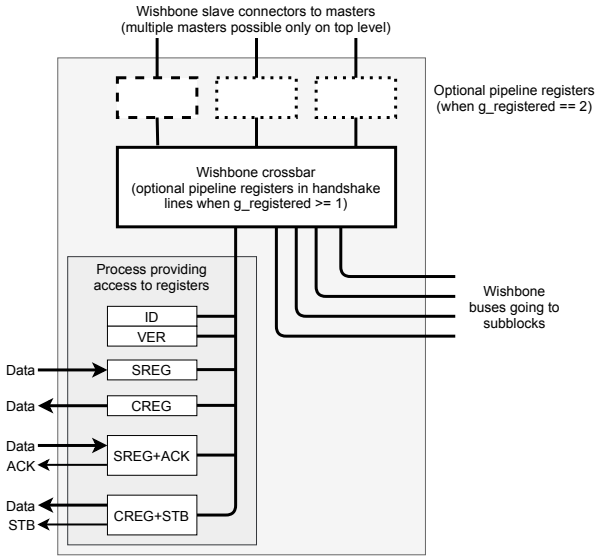

Figure 4. The structure of the "AGWB local node" (ALN) automatically generated by AGWB. ALN should be placed by the user in the design, as described in section 3.4.

The ALN should be instantiated by the user in the design. Except for the VHDL code of ALN, AGWB also generates necessary VHDL packages supporting that instantiation. They are described more thoroughly in section 3.4.1.

The structure of the generated C&D-system is defined by the user in the system definition, described in the next section.

3.2. AGWB format for C&D-system definition

The C&D-system in AGWB is described with XML. The XML is often considered obsolete and bloated compared to newer alternatives like JSON or YAML. However, it distinguishes between attributes and elements. That significantly facilitates description of the C&D-systems. The XML format is defined with the RelaxNG schema, which allows automated detection of a significant part of errors[10].

The AGWB XML format uses a non-standard extension - the **include** element described in section 3.2.1 enabling splitting the whole system description into parts that may be reused in different projects. Only the "combined" file obtained by inserting the included files may be checked for compliance with the schema.

The AGWB XML format has been designed to make it legible for a human and easy to edit. An example of the system description is shown in Listing 1. The definition of a nested block that may be included in the system definition is shown in Listing 2.

[10] The schema is also converted to RNC and DTD formats. Due to the bug in the library used to check compliance with RelaxNG and RNC schema, the DTD version is used for the checking.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1


```
<sysdef top="MAIN" masters="2">

<constant name="NEXTERNS" val="4" />
<constant name="LINK_NR_BITS" val="5" />
<constant name="LINK_NR" val="(1 &lt;&lt; LINK_NR_BITS)-1" />

<include path="block1.xml"/>

<block name="MAIN" reserved="1024" >
  <blackbox name="I2C" type="I2C_CTRL" addrbits="3" reps="8;4" />
  <subblock name="LINKS" type="SYS1" reps="LINK_NR + 1"  />
  <blackbox name="BRAM" type="WB_BRAM" addrbits="12" desc="Block RAM"/>
  <creg name="CTRL" desc="Control register in the main block">
    <field name="LINK_SELECT" width="LINK_NR_BITS" default="7" />
    <field name="COUNT_MODE" width="4" default="2"/>
    <field name="COUNT_RESET" width="1" trigger="1"/>
    <field name="PLL_RESET" width="1" trigger="1"/>
  </creg>
  <creg name="TEST_OUT" width="17" reps="3" default="23" stb="1" />
  <sreg name="TEST_IN" width="16" reps="4" ack="1" />
</block>

</sysdef>
```

Listing 1. Example definition of the C&D-system. The example exposes various features of AGWB, which are explained in the next sections. Please note how constants are defined and used in expressions and attributes. The definition of **LINK_NR** constant exposes the limitations of XML – the expression "`1<<LINK_NR_BITS`" had to be coded as "`1 << LINK_NR_BITS`". The number of repetitions of block "I2C" is defined as a colon-separated list. That is associated with *variants* functionality described in section 3.6.

```
<block name="SYS1" aggr_outs="1">
  <creg name="CTRL" desc="Control register" stb="1">
    <field name="START" width="1" trigger="1" desc="Start the operation"/>
    <field name="SPEED" width="4" default="-1" type="signed"
           desc="Transmission speed"/>
    <field name="STOP" width="1" trigger="1" desc="Stop the operation" />
  </creg>
  <sreg name="STATUS" desc="Status register" ack="1">
    <field name="RX_AV" width="1" desc="Received data available"/>
    <field name="TX_RDY" width="1" desc="May accept data to transmit"/>
    <field name="TX_DONE" width="1" desc="All data transmitted"/>
    <field name="TX_ERROR" width="2" desc="TX error code"/>
    <field name="RX_ERROR" width="4" desc="RX error code"/>
  </sreg>
  <sreg name="RXD" desc="Received data register" ack="1"/>
  <creg name="TXD" desc="Transmit data registers" stb="1" default="0x0"/>
</block>
```

Listing 2. Example definition of the nested block. Please note that this is not a valid AGWB XML description. It must be included in the system definition to form the valid AGWB XML description.

The possible XML elements that may be used in the AGWB system definition are explained in the following sections.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

3.2.1. **sysdef** element

The main element in the AGWB XML file is **sysdef**. It may contain three types of child elements:

- **block** that corresponds to the block in 1.2,
- **constant** that defines a constant parameter, which may be used in further definitions. The constants are exported to the generated system descriptions. The constant element must have two attributes. The **name** attribute defines the unique name of the constant. The **val** attribute defines the value of the constant. It may be defined as a number or as a Python expression using numbers and previously defined constants. The constant may also have the **desc** attribute containing the description stored in the generated documentation.
- **include** that defines another XML file (its path should be specified in the **path** attribute of the element), which content should be inserted instead of the **include** element. The **include** functionality is essential for the reuse of definitions in different projects. The user may prepare a library of parameterized definitions of blocks and include them in descriptions of various projects. The user may also put the parameters, defining the system's structure into a separate file. That enables the separation of user-modifiable constants from the rest of the description, which should not be changed by the user.

The **sysdef** element must have a **top** attribute that should be set to the name of the block being the top of the design. In many applications, the C&D-system may have a few bus masters. For example the Wishbone bus may be controlled via an IPbus interface or via a JTAG-based controller, which is slower but enables debugging when the network connection is not available yet. Another master may be a soft CPU included in the design, which should perform the system's initial configuration, making other interfaces operable. If the system has more than one master, the **sysdef** should have an optional **masters** attribute set to the number of masters.

3.2.2. **block** element

The **block** element defines a block of the C&D-system system. Usually, the block of a C&D-system is located in a block of the data processing system. The block may contain instances of other blocks – they are defined by **subblock** child elements. The block may also contain control and/or status registers – they are defined by **creg** and **sreg** child elements. It is possible to include into the block a slave that is not generated by AGWB - it is defined by the **blackbox** child node. Each **block** element must have a **name** attribute that defines a unique name of the block. It may also have a few optional attributes:

- **aggr_ins** and **aggr_outs**, if present and set to a non-zero value, define that the signals associated with status and control registers, respectively, are not exposed individually at a port of the generated block. Instead they are encapsulated in a VHDL record. That simplifies the routing of those signals if they should be used in another part of the design. In most cases, that feature should not be necessary for correctly designed systems.
- **reserved** attribute defines how many words should be reserved at the beginning of the address space assigned for the block. Such reserved areas will be excluded from the address allocation scheme used by AGWB. Usually, the reservation should be made only in the top block.
- **testdev_ena** attribute if present, and set to a non-zero value, switches on the generation of a simple **test device** at the beginning of the address space assigned to the block. Each block always has two status registers - ID (block identifier at offset 0) and VER (block version at offset 1) placed right after the reserved area (or at the beginning of the block address space when no area is reserved). The **test device** consists of four additional registers:

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

- **test_rw** register at offset 4, which may be written and read (it should return the last written value).
- **test_wo** register at offset 5, which may be only written, an attempt to read generates the bus error.
- **test_ro** register at offset 6, which may be only read (it should return the last value written to **test_wo**). An attempt to write to that register results in a bus error.
- **test_tout** register at offset 7. Any access to that register results in a bus timeout.

The test device provides means to test the correct operation of the C&D bus reliably. Usually, it should be instantiated only in the top block. However, if the C&D-system contains parts operating with different clocks and connected via clock-domain-crossing (CDC) blocks, it may be reasonable to istantiate them in one block in each of those parts.

- **desc** attribute should be set to the textual description of the block. The value of that element is written to the generated documentation of the C&D-system.
- **ignore** attribute says that this particular block, together with its contents, should be ignored when generating the address map for the particular backend. Currently, that functionality is used only for the Forth backend. The J1B soft CPU [47,48] using the Forth backend has limited memory, and a huge address map may overflow it. This functionality enables excluding the blocks not used during the initialization from the address map generated for Forth CPU.

3.2.3. The **subblock** element

That element may appear inside of the **block** definition. It describes the instance of another block. The name of the instance is defined by the obligatory **name** attribute. Another obligatory **type** attribute must contain the name of the instantiated block. The optional **desc** and **ignore** attributes have the same function as in the **block** element. The **reps** attribute, if present, enforces the implementation of the vector of instances. Even if the value of the attribute is equal to one, a one-element vector is generated. If the value of the attribute is zero, the vector of blocks will not be instantiated. The **used** attribute enables conditional instantiation of the single instance of the subblock. If the value of the attribute is zero, the instance is not created. Otherwise, it is.

3.2.4. The **blackbox** element

As stated in section 3.2.3, that element defines the instance of the block that is not generated by AGWB. It reserves the area in the address space defined by the value of the **addrbits** attribute. The size of that area is 2^{addrbits} words. The area is also properly aligned. The optional **xmlpath** attribute defines the name of the internal address map of the blackbox, defined by the user. If this attribute is not set, the name of the address map is created based on the **type** attribute, as **agwb_TYPE_address.xml**. Other attributes have the same meaning as in the **subblock** element.

3.2.5. The elements for registers description

AGWB uses two XML elements to describe the registers – **creg** for control registers and **sreg** for status registers. Because they are similar, they may be described together.

The register may be split into bitfields. In that case, it contains **field** child elements (see section 3.2.7).

The register definition must contain the **name** attribute which defines its name. The attributes: **ignore**, **desc**, **reps**, and **used** have the same meaning in registers, as in previously described elements. The attribute **type** sets the type of the register. It can have value "std_logic_vector" (default), "signed" or "unsigned". For each register also the type of the associated signal is created. By default, it has the form **t_REGISTERNAME**. That solution, however, may sometimes lead to namespace collisions. Therefore, the user may enforce another name of the generated type using the **stype** attribute. The **width**

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

attribute allows the user to limit the width of the register. The **mode** attribute is specific to the IPbus backend, and its value is transparently passed to the generated XML IPbus address map.

The status registers may have the optional **ack** attribute. If it is set to a non-zero value, an additional *acknowledge* signal will be generated. Whenever the register is read, it will be asserted for one bus clock period.

Similarly, the control registers may have the optional **stb** attribute. If it is set to a non-zero value, an additional *strobe* signal will be generated. Whenever the register is written, it will be asserted for one bus clock period.

Those two signals are especially useful if the associated registers are connected to a FIFO block. The additional signals may be used as read and write strobes.

The last attribute specific to control registers is the **default** attribute. It defines the register's *default* value, which is set after the FPGA is reprogrammed or after the reset signal is asserted.

3.2.6. Handling of registers with aggregation of inputs or outputs

As mentioned in section 3.2.2, combining the signals associated with control and/or status registers into a single record is possible.

If the **aggr_ins** attribute is set to the non-zero value, a single input **regs_in** port of type **t_BLOCKNAME_in_regs** is generated. The type **t_BLOCKNAME_in_regs** is generated as a record containing the elements with the same names as the status registers and the same types as those registers. If any status registers have the **ack** attribute active, there is also an output port **ack_regs_o** generated of type **t_BLOCKNAME_ack_regs**. The type **t_BLOCKNAME_ack_regs** contains elements with the same names as the associated registers. The *std_logic* elements are created for single registers, and for vectors of registers, the *std_logic_vector* elements are created.

If the **aggr_outs** attribute is set to the non-zero value, the **regs_out** port of type **t_BLOCKNAME_out_regs** is generated. The type **t_BLOCKNAME_out_regs** is generated as a record containing the elements with the same names as the control registers and the same types as those registers. For control registers with non-zero **stb** attribute, there are additional **REGISTERNAME_stb** elements created in the record. Their type is *std_logic* for single registers and *std_logic_vector* for vectors of registers.

3.2.7. The **field** element

The **field** elements define bitfields inside of registers. Their names are defined with the obligatory **name** attributes. The second obligatory attribute is **width** describing the number of bits in the bitfield. The sum of widths of all fields in the register must not exceed 32. The **field** element may also have the optional **default** (only for bitfield in a control register), **desc**, **ignore**, and **type** attributes with the meaning described earlier.

The bitfield defined in the control register may have a **trigger** attribute. If it is present and set to the non-zero value, all bits in that bitfield, when written with '1', remain asserted only for a single period of the bus clock. They are always read as zeroes. Such bitfields are dedicated to triggering actions in the hardware.

If the register is split into bitfields, its width is automatically set to the sum of its fields' widths.

3.3. Address allocation algorithms

The address allocation in AGWB is designed to support the optimization of address decoders. Therefore each block is placed in the address space so that the certain number of lower address bits (let us denote them as M) is used for internal addressing (selection of registers or subblocks), while the rest $K = 32 - M$ are used to select that block (so they are decoded by the Wishbone crossbar to which it is connected). The second goal is to minimize the occupied address space by avoiding unnecessary fragmentation.

The address allocation works in the following way:

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

- For each block, the number of used addresses L is calculated as the sum of sizes of its registers and subblocks. The number of addresses is rounded up to the nearest power of two: $P = 2^N$, where N is the smallest integer for which $2^N \geq L$. The P value is the size of the block. That step may be performed recurrently, as the size of subblocks is needed to calculate the size of the parent block.
- In each block, the registers are located at the beginning of the address space[11] (so **ID** and **VER** have well-defined locations). The subblocks are sorted according to their decreasing size and are placed starting from the end of the block's address space.
- The final address map is built starting from the top block located at address 0 and traversing its subblocks.

Such an algorithm ensures that each block's base address is properly aligned and warrants minimal address space fragmentation.

3.4. Integration of AGWB-generated part with user logic

The C&D-system components generated by the AGWB must be properly embedded in the user logic. The user must implement the appropriate connection of the master Wishbone buses to the masters and route the child node Wishbone buses to the nested blocks. The simplified diagram of the necessary connection is shown in Figure 5.

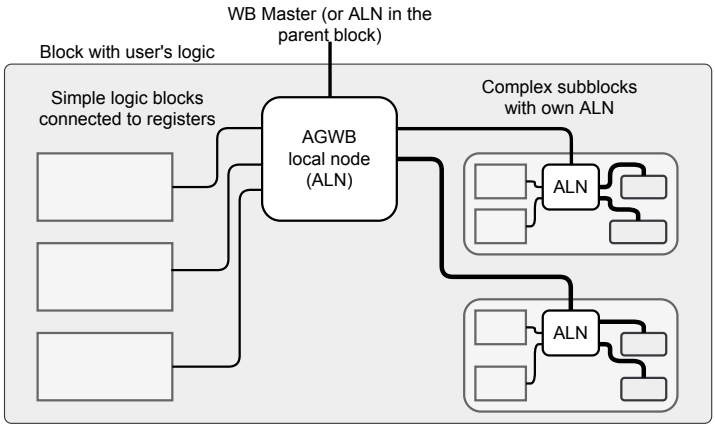

Figure 5. Block diagram of the single block, showing how the signals generated in the AGWB local node (ALN) should be connected to the user's logic consisting of simple logic blocks connected to registers and complex subblocks with their own ALN. The structure of the ALN is shown in Figure 4.

The AGWB has been designed in a way that simplifies the integration. The signals associated with registers may be of type **std_logic_vector**, **signed**, or **unsigned**, as defined with the **type** attribute described in section 3.2.5. AGWB creates special types defined for individual registers (**t_REGISTERNAME** for individual registers and **t_REGISTERNAME_array** for vectors of registers), which may be used to define necessary signals. For registers with bitfields, the appropriate record types are created. The usage of generated types minimizes the risk of mistakes.

Connections between the blocks are intended to be done with standard Wishbone buses. The appropriate types **t_wishbone_slave_in**, **t_wishbone_slave_out**, **t_wishbone_master_in**, and **t_wishbone_master_out** are provided by the library General cores [22].

For vectors of blocks, arrays of the Wishbone buses are used. The related types **t_wishbone_slave_in_array**, **t_wishbone_slave_out_array**,

[11] If there is a reserved area at the beginning, defined with the **reserved** attribute (see section 3.2.2), it will be placed before the registers.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

t_wishbone_master_in_array, and **t_wishbone_master_out_array** are delivered by the same library.

3.4.1. VHDL files generated by AGWB

To support the integration of the generated C&D-system with the user-provided logic, AGWB generates for each block the following files:

- the **BLOCKNAME.vhd** providing the AGWB local node (ALN) code,
- the **BLOCKNAME_pkg.vhd** providing the **BLOCKNAME_pkg** package containing the types and constants specific to that block,
- for the top block only, the **BLOCKNAME_const_pkg.vhd** containing the constants defined in the system definition XML file.

The block-specific constants belong to the respective packages. That enables avoiding name conflicts, which may, for example, occur when registers with the same name (and hence of the type with the same name) are used in multiple blocks.

3.4.2. Use of C&D-system in designs using multiple clock domains

Different data processing blocks may work with different clock frequencies in more complex data acquisition and processing systems. If the Wishbone bus working with the same clock is used across the whole design, reading and writing the registers driven by another clock requires proper synchronization. Doing it at the register level would be very resource-consuming. A better approach may be to create the Wishbone bus segments operating at different clock frequencies and connect them via an appropriate clock-domain-crossing (CDC) block. AGWB offers a dedicated CDC block optimized for use in the C&D-systems.

However, if the blocks working with different clocks are scattered throughout the design, there are two approaches for splitting the control bus.

The first approach preserves the logical organization of the control tree and uses CDC whenever the part of the block is operating with another clock. That approach is shown in Figure 6.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

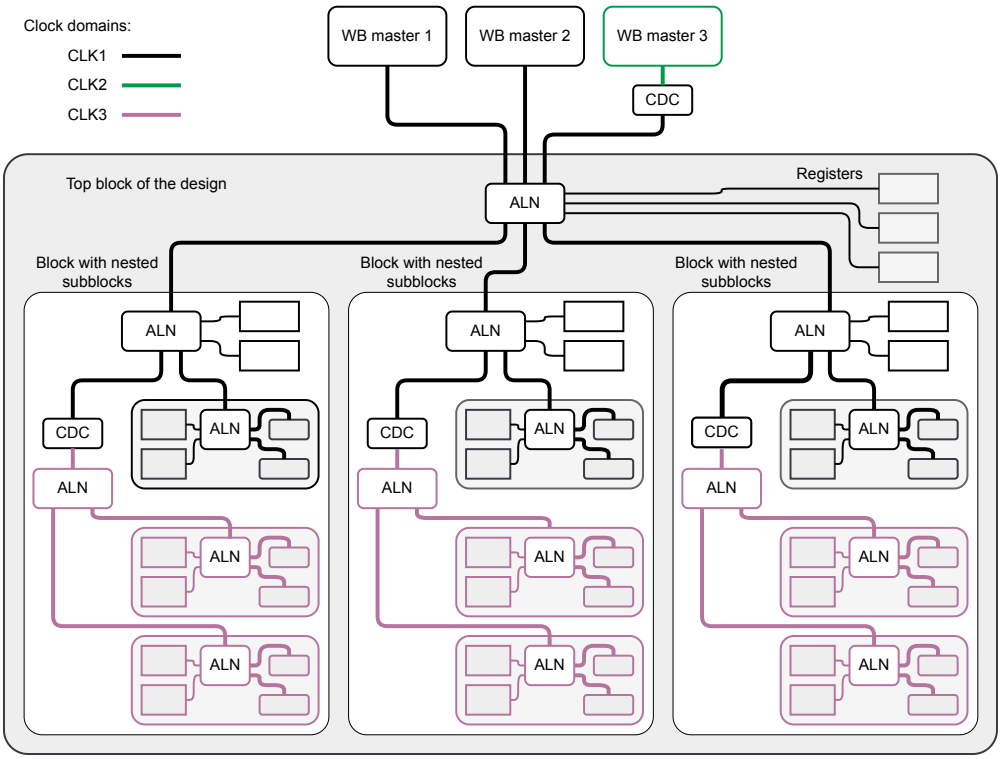

Figure 6. Block diagram of a system with multiple clock domains. One of the WB masters works with another clock domain. Each of the top block subblocks has a part that works with another clock. In this approach, each block is controlled via a single Wishbone bus. For the part working with another clock, the internal CDC is implemented.

The advantage is that each block may use a coherent area in the address space. The cost is that the number of CDC blocks is high. This approach may lead to connecting a few CDC in series in certain topologies, which results in a significant slow-down of register accesses (each synchronization stage consumes a certain number of clock cycles during the transaction).

Another approach is shown in Figure 7.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

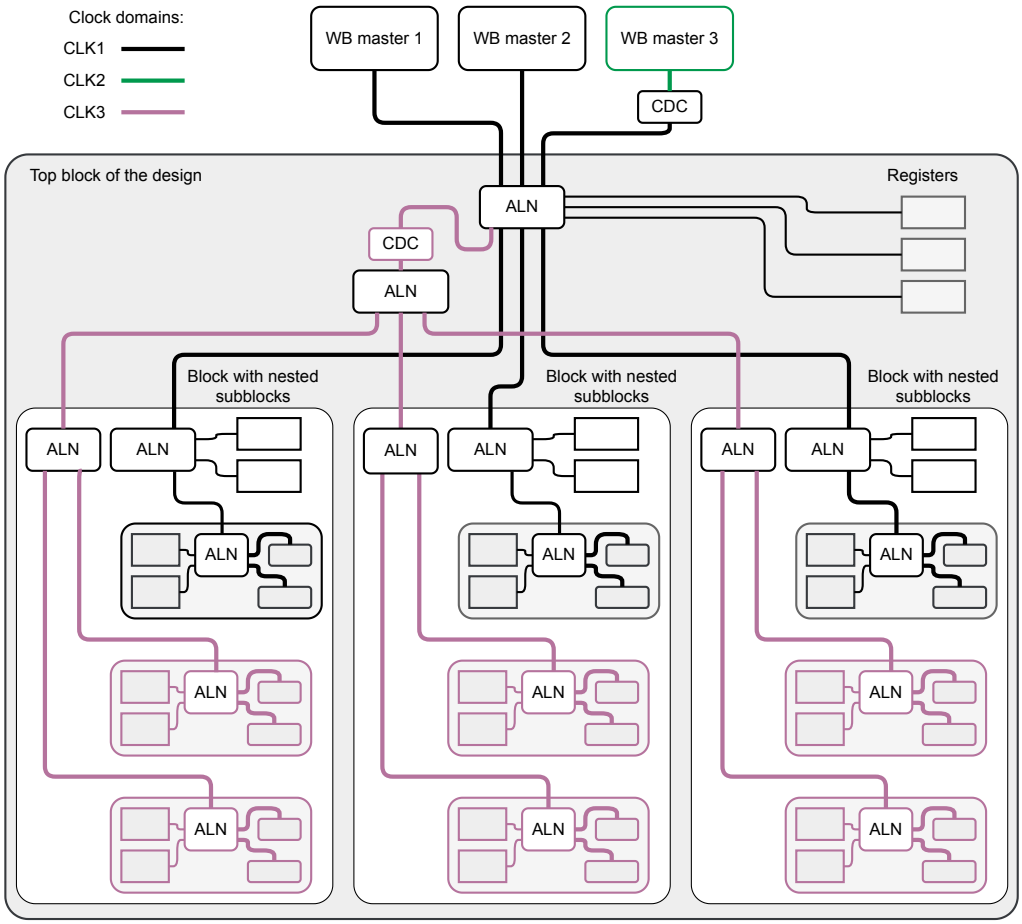

Figure 7. Block diagram of a system with multiple clock domains. One of the WB masters works with another clock domain. Each of the top block subblocks has a part that works with another clock. In this approach, for two clocks used in the Wishbone slaves, the separate buses are created at the top level. This split of control bus needs only one additional CDC, but two buses must be routed through the design, and the address space of each subblock is split into two areas.

In this approach, the buses working with different clocks are independently routed through the design. That keeps the minimal possible number of CDC blocks but increases resource consumption for routing the two independent buses. Additionally, the block may occupy two or more separate areas in the control address space, complicating the software's writing.

The selection of the right approach is the responsibility of the user. Sometimes the best solution could be using the first approach in certain parts of the design, and the second in others.

3.5. Integration of C&D-system with software

The hardware part of the C&D-system connected to the user's logic must be integrated with the software. That requires a few main points:

- The software must be informed about the structure of the hierarchy, addresses and properties of the individual registers (obligatory);
- There may be a hierarchy of objects created that reflects the hierarchy of blocks and registers in the software (optional)

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

The method to fulfill those requirements may depend on the language in which the software should be written.

Currently, there are a few languages supported, which are handled by the appropriate backend routines generating the necessary files.

3.5.1. IPbus backend

The AGWB development was started for the systems controlled via the IPbus interface [23]. Therefore the generation of the address map in the XML format suitable for IPbus [39] was an initial solution. The address tables generated by the IPbus backend may be used by Python software and by C++ software. However, the IPbus address table format has a significant disadvantage – it does not support vectors of registers nor vectors of blocks. Each element of the vector must be specified individually in the address table.

The advantage of the IPbus address table is that its modification does not require recompilation of the software. For example, the same compiled software may support hardware with different versions of the firmware. It is sufficient to load the appropriate version of the address table.

3.5.2. AMAP XML backend

The AMAP XML format has been created to avoid the limitations of the IPbus format while preserving its advantages. It has been extended with support for vectors of registers and vectors of blocks. Therefore, the nodes may have two additional attributes:

- **nelems** - describing the number of elements in the vector,
- **elemoffs** - the distance in the address space between the base addresses of the consecutive elements of the vector.

Additionally, it introduces different XML elements for different types of nodes (the IPbus XML address table keeps everything in an element **node**). The block definition is stored in the element **module**. The subblock instances are stored in the elements **block**. The register definitions are stored in the elements **register**.

The software using the AMAP XML, similarly to the IPbus software, should allow loading or reloading the definition of the system. Therefore the modification of the address map does not enforce recompilation of the software.

The AMAP XML format supports **variants** (described in section 3.6).

3.5.3. Native Python backend

The native Python backend gives the best integration with the Python language. The hierarchy of blocks and registers is fully reflected in the hierarchy of classes. The structure of the C&D-system is reflected in the tree of objects created when the user imports the generated Python code and accesses the register.

The native Python mode is very flexible. It may use different connections to the Wishbone bus. For elementary access, the user must only define a virtual interface with the following methods:

- `read(self, address)` that returns a 32-bit value from the register at the particular address,
- `write(self, address, value)` that writes a 32-bit value to the register at the particular address,

AGWB may be used with network-based control interfaces, which offer high throughput, but high round-trip latency. It may significantly affect the performance of the read-modify-write (RMW) operations, often used in control algorithms. Performing the RMW operation in hardware and aggregating multiple bit-fields operations into a single

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

RMW operation may cure that problem[12]. The native Python backend may use those optimizations if the following methods are implemented:

- `writeb(self, address, value)` – only schedules a write (unless the operation list is full or until **dispatch** is called),
- `readb(self, address)` – returns the "Callable" object. Calling the returned object returns the value (possibly executing **dispatch** if necessary),
- `write_masked(self, address, mask, value)` – executes the read-modify-write operation defined as follows `X:= (X and ~mask) | (value and mask)`.
- `writeb_masked(self, address, mask, value, more=False)` – prepares the read-modify-write operation defined as follows `X:= (X and ~mask) | (value and mask)`. When **more** is set to "True" it blocks the immediate scheduling of the operation. Then multiple writes **to fields located in the same register** are accumulated. The last call must have **more** set to False - it schedules the resulting read-modify-write command.
- `dispatch()` – executes the accumulated list of operations (the list may be automatically executed if it grows to its maximal allowed length).

The native Python mode is ideal for interactive debugging that involves the software running on a PC. The user may directly access the registers. The Python introspection and reflection functionalities are available in the interactive mode. The complex Python routines also may be executed.

3.5.4. C header backend

For software written in C, AGWB generates a few C headers. The constants defined for the whole system are placed into the **agwb_TOPBLOCK_const.h** file. The constants are generated as numerical values, but they are accompanied by a comment explaining how that value was calculated in Python (e.g.,: `#define NSEL_MAX 31 // (1 << NSEL_BITS)-1`).

For each block, there is a header **agwb_BLOCKNAME.h** generated. This header contains the following content:

- Constants with the block ID and VER values (see section 3.1).
- Definition of the type associated with the structure consisting of
  - **uint32_t** fields (corresponding to registers),
  - similar structures describing the nested blocks,
  - **uint32_t** fillers (corresponding to unused parts of the address space).
- Definitions of the inline functions for reading or writing the values of the bitfields.

A necessity to use special functions instead of standard C bitfields results from the fact that the C standard does not define how the bitfields are placed in words. Therefore, another portable method was finally implemented after initial attempts to encode the bitfields using the standard C mechanism. The access functions are defined as follows:

- `inline uint32_t agwb_BLOCK_REG_FIELD_get(uint32_t * ptr)`
- `void agwb_BLOCK_REG_FIELD_set(uint32_t * ptr, uint32_t val)`

The C backend is generally dedicated to writing the C drivers (especially the kernel drivers) for systems where the FPGA-implemented register is directly mapped into the CPU address space. Typical use cases are the SoC or MPSoC with FPGA part connected directly to the AXI bus of the CPU (in that case, an AXI to Wishbone bridge is needed) or system with FPGA connected via PCIe interface (in that case also an appropriate bridge PCIe-AXI-WB is needed).

The advantage of the C backend is fast and direct access to the C&D-system registers. Of course, the user must take care of all particular hardware platform features, like operations reordering, access synchronization, etc.

[12] A good example may be an Ethernet-based IPbus [23] interface which implements both mentioned optimizations.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

The disadvantage of the C backend is that it requires recompilation of the software whenever the design is modified.

3.5.5. Forth backend

The Forth backend enables control of the AGWB-generated system from the J1B Forth CPU [47,49,50]. It is a simple synthesizable CPU able to execute the programs written in Forth [51] language.

The Forth language supports very efficient interactive work, so it is a good tool for interactive debugging. It requires only a console connection[13] to the FPGA with firmware containing J1B. During the interactive work, the programmer still may define procedures (called "words" in Forth) and create fairly complex algorithms. It is also possible to put the defined procedures into the FPGA configuration bitstream, enabling automated execution of a certain word after the system starts. Therefore, J1B may be a perfect tool for the initialization of the board.

The J1B CPU is optimized for low resource consumption and, therefore, it has limited code memory. In the case of a complex AGWB-generated C&D-system, its memory may get filled just with the names of blocks and registers. For initialization, usually, only a small subset of the registers is needed. The unused blocks and registers may have the "ignore" attribute set to "forth" to be excluded from the generated Forth software (see section 3.2.2).

The "Swapforth" version of Forth used by J1B [52] offers a possibility to clean and reload word definitions. When generating the Forth code for J1B, it is possible to define a special **marker** word before the AGWB definitions.

```
marker del_agwb
```

Executing that word cleans its definition and all further definitions, enabling the user to load another set of word definitions[14].

The Forth code generated for the system definition from Listings 1 and 2 is shown in Listing 3.

[13] Usually, Forth debugging uses the UART, but it is also possible to provide a console via JTAG, SPI, or another interface.
[14] Of course, the **del_agwb** marker should be recreated before.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1


```
: /%NEXTERNS $4 ; \ 4
: /%LINK_NR_BITS $5 ; \ 5
: /%LINK_NR $1f ; \ (1 << LINK_NR_BITS)-1
: // $0 ;
: //_ID // $400 + ;
: //_VER // $401 + ;
$89bd20d0 constant //_ID_VAL
$a81ec322 constant //_VER_VAL
: //_CTRL // $402 + ;
: //_CTRL.LINK_SELECT //_CTRL $1f $0 ;
: //_CTRL.COUNT_MODE //_CTRL $1e0 $5 ;
: //_CTRL.COUNT_RESET //_CTRL $200 $9 ;
: //_CTRL.PLL_RESET //_CTRL $400 $a ;
: //#TEST_OUT // + $403 + ;
: //#TEST_IN // + $406 + ;
: //#I2C // $ec0 + swap $8 * + ;
: //#LINKS // $f00 + swap $8 * + ;
: //#LINKS_ID //#LINKS $0 + ;
: //#LINKS_VER //#LINKS $1 + ;
$5bd964c2 constant //#LINKS_ID_VAL
$b0efef2 constant //#LINKS_VER_VAL
: //#LINKS_CTRL //#LINKS $2 + ;
: //#LINKS_CTRL.START //#LINKS_CTRL $1 $0 ;
: //#LINKS_CTRL.SPEED //#LINKS_CTRL $1e $1 ;
: //#LINKS_CTRL.STOP //#LINKS_CTRL $20 $5 ;
: //#LINKS_STATUS //#LINKS $3 + ;
: //#LINKS_STATUS.RX_AV //#LINKS_STATUS $1 $0 ;
: //#LINKS_STATUS.TX_RDY //#LINKS_STATUS $2 $1 ;
: //#LINKS_STATUS.TX_DONE //#LINKS_STATUS $4 $2 ;
: //#LINKS_STATUS.TX_ERROR //#LINKS_STATUS $18 $3 ;
: //#LINKS_STATUS.RX_ERROR //#LINKS_STATUS $1e0 $5 ;
: //#LINKS_RXD //#LINKS $4 + ;
: //#LINKS_TXD //#LINKS $5 + ;
: //_BRAM // $1000 + ;
```

Listing 3. The Forth code generated by AGWB. The names of words are based on the names of block instances and registers. The initial "//" enables avoiding collisions with standard Swapforth words. The names include block instance(s), register (separated with "_"), and optionally bitfield (separated with "."). The use of "#" instead of "_" as a separator informs that the object is a part of a vector of blocks or registers. The element number (or numbers if it is a vector of registers in an element of a blocks' vector) must be put on the stack before executing such word.

The defined words push to the data stack the address of the accessed register, which can be later on read with **wb** or written with **wb!**. When accessing bitfields, the register address and the bitfield parameters are pushed to the stack. Afterward, the bitfield may be read with **wb@** or written with **bf!**.

3.6. Support for special types of designs

The AGWB assumes that the instances of the block are identical. It was a conscious design decision that simplifies the allocation of addresses and generation of code. That is why no user parameters are passed to the **subblock** element. Therefore the size allocated for the block in the address space is always the same.

However, there may be situations where not all resources provided by certain instances of the block are needed.

An example may be the usage of multigigabit transceivers (MGT). They are often grouped in banks containing four transceivers, which share certain infrastructure. Therefore, it makes sense to create an MGT controller block in C&D-system that controls four

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

MGTs. Let us assume that our FPGA has four such banks, but we need to utilize two MGTs for other purposes and require another controller. So we need three instances of MGT controller that control four MGTs, and one instance which controls only two MGTs.

AGWB handles such cases at the level of integration of the AGWB-generates C&D-system with the user logic. In theory, we could only not connect two MGTs to the last instance of the MGT controller. Unfortunately, that approach does not warrant that unused resources will be optimized out during the firmware synthesis[15].

3.6.1. precise customization

To avoid it, AGWB offers a **precise customization** functionality, enabling the developer to control which registers or subblocks and how many of them are synthesized. The generated ALN code uses special generics of form **g_REGISTERNAME_size** or **g_SUBBLOCKNAME_size**. Those generics may be set to 0 (excluding the related object from synthesis) or 1 (including it) for single blocks or registers. For vectors of blocks and registers, the generics may be set to 0 (excluding the whole vector from synthesis) or to any value between 1 and the maximum size (defined with the **reps** attribute described in section 3.2.3). That maximum size is defined as the constant **c_REGISTERNAME_size** or **c_SUBBLOCKNAME_size** in the generated VHDL package. The generics defined in the same package have default values set to those constants to keep using the AGWB simple when the user does not want to utilize precise customization. The user may omit the generic during instantiation, and it will be set to its maximum value.

Using precise customization also affects the definitions of types for signals.

Therefore, except for types described in section 3.4 like **t_REGISTERNAME_array**, there are also declared unconstrained types **ut_REGISTERNAME_array**.

Of course, the above modifications affect the hardware part of the generated C&D-system. The user's responsibility is to ensure that the C&D software is aware of the irregularities handled by precise customization.

3.6.2. Design **variants**

Usage of AGWB with big Xilinx FPGAs that use Stacked Silicon Interconnect (SSI) technology [53] and are divided into multiple Super Logic Regions (SLRs)[54] have exposed yet another need for handling irregularities in the AGWB-described C&D-system.

The communication between SLRs requires special Super Long Lines (SLL) [55], a scarce resource. Therefore, if the FPGA has two SLRs[16], where each is connected to the host via a separate PCIe interface, the optimal solution may be to implement two similar data acquisition and processing subsystems – one in each SLR.

Certain blocks, however, have to be implemented only in one SLR and communicate with another SLR via SLLs. That introduces irregularities of another kind than those described in section 3.6.1. There, the irregularities appeared in the same C&D-system. Here, we need to put two different versions of the C&D-system into the same FPGA. The first possibility seemed to be the use of **include** functionality of AGWB. Two top-level XMLs could be created – one for each SLR. Each should include an SLR-dependent set of constants and then the standard description of the system. Of course the generated files should be put into other directories for each SLR. Unfortunately, such a solution is not possible due to VHDL limitations. To have two different versions of the generated VHDL code, one should place each of them into another VHDL library. The code integrating the user logic with the generated code should then select the appropriate library. Unfortunately, up to now, the VHDL does not support selecting the library by generics, passing the library name as a generic, or creating an alias to the library.

[15] The control registers are storage elements, so they remain synthesized even if they are not connected to any user logic. Similarly, the CDC blocks must be protected against certain optimizations, resulting in keeping them even if they are not used.

[16] That is the case of the xcku115-flvf1924 FPGA available in the CRI boards used in the CBM experiments.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

The finally implemented and working solution extends the precise customization described in section 3.6.1 and is called **variants**.

The user may define a few **variants** of the design, and specify the values of certain attributes for each variant individually. That is achieved by defining a colon-separated list of possible values instead of a single value, like below:

```
<blackbox name="I2C" type="I2C_CTRL" addrbits="3" reps="8;6;4" />
```

The above description defines three variants of the design, where variant 0 has 8 I2C controllers, variant 1 – 6, and variant 2 – 4 controllers.

Of course, all variant-dependent attributes must define the same number of variants. The following description will result in an error:

```
<blackbox name="I2C" type="I2C_CTRL" addrbits="3" reps="8;6;4" />
<blackbox name="SPI" type="SPI_CTRL" addrbits="3" used="1;0" />
```

because the first line defines three variants, while the second only two.

When using variants, the additional constants **v_REGISTERNAME_size** or **v_SUBBLOCKNAME_size** are defined in the generated package, which are the integer arrays, storing the size of the object in each variant.

Currently, only two software backends support **variants**:

- native Python (see section 3.5.3),
- AMAP XML (see section 3.5.2).

If the block is variant-dependent and the C&D-system is supposed to be used with the variants-aware software, the user should set the VER value for the variant-dependent value (calculated as a CRC32 of the generated AMAP XML description of that block). The following setting of generic provides that during the instantiation of the block:

```
 g_ver_id         => v_BLOCKNAME_ver_id(variant_number),.
```

The number of the variant should be propagated throughout the design as a dedicated integer generic.

The example of the instantiation of the variant-dependent block is shown in Listing 4.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1


```
  [...]

  -- LINKS are a vector of blocks so they use a vector of Wishbone buses
  signal LINKS_wb_m_o : t_wishbone_master_out_array(v_LINKS_size(var_nr)-1 downto 0);
  signal LINKS_wb_m_i : t_wishbone_master_in_array(v_LINKS_size(var_nr)-1 downto 0);
  -- EXTHUGE is a single block present in only one variant
  signal EXTHUGE_wb_m_o : t_wishbone_master_out;
  signal EXTHUGE_wb_m_i : t_wishbone_master_in;
  -- TEST_IN is a variant-dependent vector
  signal TEST_IN_i : ut_TEST_IN_array(v_TEST_IN_size(var_nr)-1 downto 0)
         := (others => (others => '0'));
  -- The below signals are related to the variant-independent blocks and registers
  signal EXTERN_wb_m_o : t_wishbone_master_out_array((C_NEXTERNS-1) downto 0);
  signal EXTERN_wb_m_i : t_wishbone_master_in_array((C_NEXTERNS-1) downto 0);
  signal CDC_wb_m_o : t_wishbone_master_out_array((C_NEXTERNS-1) downto 0);
  signal CDC_wb_m_i : t_wishbone_master_in_array((C_NEXTERNS-1) downto 0);
  signal CTRL_o : t_CTRL;
  signal TEST_OUT_o : t_TEST_OUT_array := (others => (others => '0'));

  [...]

  MAIN_1 : entity agwb.MAIN
    generic map(
      g_ver_id => v_MAIN_ver_id(var_nr),
      g_LINKS_size => v_LINKS_size(var_nr),
      g_EXTHUGE_size => v_EXTHUGE_size(var_nr),
      g_registered => 2
)
    port map (
      slave_i        => wb_s_in,
      slave_o        => wb_s_out,
      LINKS_wb_m_o   => LINKS_wb_m_o,
      LINKS_wb_m_i   => LINKS_wb_m_i,
      EXTHUGE_wb_m_o => EXTHUGE_wb_m_o,
      EXTHUGE_wb_m_i => EXTHUGE_wb_m_i,
      EXTERN_wb_m_o  => CDC_wb_m_o,
      EXTERN_wb_m_i  => CDC_wb_m_i,
      CTRL_o         => CTRL_o,
      TEST_IN_i      => TEST_IN_i,
      TEST_OUT_o     => TEST_OUT_o,
      rst_n_i        => rst_n_i,
      clk_sys_i      => clk_sys_i);

  [...]
```

Listing 4. Example of instantiation of the variant-dependent ALN.

3.7. Usage of the AGWB C&D-system generator

Processing of the AGWB C&D-system description and generating of the output files is performed by executing the **addr_gen_wb.py** script with the following command-line options:

- **-h, –help** – show help message and exit,
- **–infile INFILE** – Input file path,
- **–hdl HDL** – destination directory for VHDL output files,
- **–ipbus IPBUS** – destination directory for IPbus output files,
- **–amapxml AMAPXML** – destination directory for AMAP XML output files,
- **–header HEADER** – destination directory for C headers,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2021 doi:10.20944/preprints202110.0098.v1

- **–fs FS** – destination directory for Forth output files,
- **–python PYTHON** – destination directory for deprecated Python output files (not described in this article, cannot be used together with –pythondca),
- **–pythondca PYTHONDCA** – destination directory for the native Python files (cannot be used together with –python),
- **–html HTML** – destination directory for HTML documentation destination,
- **–fusesoc** generate FuseSoc .core file in the current directory,
- **–fusesoc_vlnv FUSESOC_VLNV** – FuseSoc VLNV tag,
- **–eprj** – generate the VEXTPROJ .eprj file in the VHDL output directory.

The AGWB system may be used with the FuseSoc [56] or VEXTPROJ [57] environments for FPGA firmware synthesis.

4. Results - practical applications

Usage of AGWB requires connection to a compatible hardware interface, supporting control of the Wishbone bus from the software. Up to now, the following hardware interfaces have been prepared and tested.

- The simplified **cbus.py** interface supporting only read and write commands. It enables testing of AGWB-generated C&D-system in simulation with GHDL.
- Connection to the IPbus master, with minimal signals adjustments (as described in [58]). The IPbus may be used with AGWB in two ways:
  - using the standard C++ and Python libraries designed for IPbus (with the code generated by the IPbus backend in section 3.5.1),
  - using only the "client" object offered by the IPbus library and its access procedure. This method works with the native Python backend (see section 3.5.3).
- The interface controlling Wishbone bus via JTAG interface [59].
- The PCIe interface based on Xilinx "AXI Bridge for PCI ExpressGen3 Subsystem v3.0" [60] and simple AXI-Lite to Wishbone bridge. The solution is available in the project [61] (branch "agwb").
- The dedicated PCIe to Wishbone bridge developed for the CBM experiment [62]. That interface allows operation in "native Python" mode and in "AMAP XML" mode.
- A specialized GBT-SC [63] interface which, when used together with the GBT-FPGA [4] core, enables control of the Wishbone bus via the GBT [64] link.

Up to now, the AGWB has been successfully used in a few experimental projects [50,61, 65] and four serious projects used for CBM and BM@N experiments:

- the DPB firmware [37],
- the GBTX Emulator (GBTxEMU) [66,67],
- the SMX tester [62],
- the CRI firmware [62].

Intensive use of AGWB in real projects, and feedback received from other developers and users contributed to introducing new functionalities and eliminating bugs. The DPB firmware [37] was the first practical project using AGWB. In the GBTX Emulator (GBTxEMU) [66], AGWB is used with three different masters - the IPbus, the J1B, and a project-specific GBT IC master. This project enabled testing and has proven the correctness of the multi-master functionality. It also required the use of the CDC block. The CRI firmware [62] project was the most demanding for AGWB up to now. The software uses extensive hardware detection and testing functions, which require reliable support for bus errors and timeouts. Its development and usage resulted in significant improvement and intensive testing of the CDC block and the introduction of the built-in test device. The CRI firmware is used in the big FPGAs consisting of two SLRs resulting in introducing the "variants" functionality described in section 3.6. The designs prepared for the CBM experiment share significant parts of the C&D-system. Therefore,

their development was a good test of the reusability of fragments of the AGWB system description.

5. Discussion and conclusions

The AGWB was created simultaneously as the description format and as a conversion tool. Therefore all proposed functionalities were immediately analyzed concerning the viability of implementation. A good example was adding the support for precise customization and variants. This approach resulted in a good balance between the functionality available to the user, low complexity of implementation, and simplicity of description.

The AGWB appeared to be a useful tool for generating C&D-systems. The syntax of the C&D-system description is simple and may be easily edited in any text editor.

Good support for parameterization of the design has been confirmed in practical use, such as changing the number of components for simplified debugging versions. It has been confirmed that properly used AGWB indeed minimizes the effort needed to adjust the user logic to the modified parameters.

The possibility to include the fragments of XML files facilitates sharing of the definitions between different C&D-system systems.

Adding new registers and blocks is relatively simple. The workflow is friendly for text-based tools, end hence for version control systems.

Of course, AGWB has its limitations resulting from the compromise between functionality and simplicity. It does not offer the versatility and completeness of SystemRDL, but is a small and consistent solution.

It is a fully open-source solution. The source code is available in the GitHub repository [68]. The LGPL V2 license allows the user to modify it for his or her needs, and the system's simplicity should facilitate such modifications.

The code generated by the AGWB may be freely used and distributed by the user, but it relies on the General cores library and uses the components licensed under Solderpad Hardware License, Version 2.0 [69].

5.1. Future plans

The AGWB system has been developed as a tool supporting currently developed systems. Its gradual evolution resulted in a structure where analysis of the system description is somehow interconnected with the output generation. In future versions, it may be advantageous to improve that separation by formalizing the internal representation of the generated system. That may urge users to create their own software or hardware backends.

The end users suggested improvements to add support for local constants and pass user arguments to block instances (subblocks). Such extensions may increase the usability of AGWB, but they must be carefully tested regarding the possibility of implementation. For example, making the block's address space size dependent on the user-provided parameters would significantly complicate the address allocation algorithm and the generated ALN code.

It is expected that the role of SystemRDL will increase in the future. Therefore, it should be investigated if it is possible to support conversion between the AGWB description and a certain well-defined subset of SystemRDL.

Author Contributions: Conceptualization, W.Z.; methodology, all authors; software, W.Z., M.G. and M.K.; validation, all authors; investigation, all authors; writing—original draft preparation, W.Z.; writing—review and editing, all authors ; visualization, W.Z.; supervision, W.Z. The percentage contribution of the authors is: W.Z. – 60%, M.G. – 15%, M.K. – 15%, W.M. – 10%. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: The work has been partially supported by the statutory funds of the Institute of Electronic Systems and partially by the Facility for Antiproton and Ion Research (FAIR).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Engel, H.; Alt, T.; Kebschull, U.; ALICE Collaboration. FPGA based data processing in the ALICE High Level Trigger in LHC Run 2. *Journal of Physics: Conference Series* **2017**, *898*, 032018. doi:10.1088/1742-6596/898/3/032018.
2. Li, Q.; Amar-Youcef, S.; Doering, D.; Deveaux, M.; Fröhlich, I.; Koziel, M.; Krebs, E.; Linnik, B.; Michel, J.; Milanovic, B.; Müntz, C.; Stroth, J.; Tischler, T. A FPGA-based Cluster Finder for CMOS Monolithic Active Pixel Sensors of the MIMOSA-26 Family. *Journal of Physics: Conference Series* **2014**, *513*, 022021. doi:10.1088/1742-6596/513/2/022021.
3. Zabołotny, W.M.; Czarski, T.; Chernyshova, M.; Czyrkowski, H.; Dąbrowski, R.; Dominik, W.; Jakubowska, K.; Karpiński, L.; Kasprowicz, G.; Kierzkowski, K.; Kudła, I.M.; Poźniak, K.; Rzadkiewicz, J.; Sałapa, Z.; Scholz, M. Optimization of FPGA processing of GEM detector signal. Proc. SPIE; Romaniuk, R.S., Ed.; , 2011; Vol. 8008, pp. 80080F–80080F–9. doi:10.1117/12.905427.
4. Marin, M.B.; Baron, S.; Feger, S.; Leitao, P.; Lupu, E.; Soos, C.; Vichoudis, P.; Wyllie, K. The GBT-FPGA core: features and challenges. *Journal of Instrumentation* **2015**, *10*, C03021–C03021. doi:10.1088/1748-0221/10/03/C03021.
5. Kasinski, K.; Szczygiel, R.; Zabolotny, W.; Lehnert, J.; Schmidt, C.; Müller, W. A protocol for hit and control synchronous transfer for the front-end electronics at the CBM experiment. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **2016**, *835*, 66–73. doi:10.1016/j.nima.2016.08.005.
6. Serial Interface for Data Converters JESD204C. https://www.jedec.org/system/files/docs/JESD204C.pdf. [Online; accessed 3-October-2021].
7. Choe, H.; Gorfman, S.; Heidbrink, S.; Pietsch, U.; Vogt, M.; Winter, J.; Ziolkowski, M. Multichannel FPGA-Based Data-Acquisition-System for Time-Resolved Synchrotron Radiation Experiments. *IEEE Transactions on Nuclear Science* **2017**, *64*, 1320–1326. doi:10.1109/TNS.2017.2655366.
8. García, G.; Jara, C.; Pomares, J.; Alabdo, A.; Poggi, L.; Torres, F. A Survey on FPGA-Based Sensor Systems: Towards Intelligent and Reconfigurable Low-Power Sensors for Computer Vision, Control and Signal Processing. *Sensors* **2014**, *14*, 6247–6278. doi:10.3390/s140406247.
9. Bai, Y.; Bodlak, M.; Frolov, V.; Jary, V.; Huber, S.; Konorov, I.; Levit, D.; Novy, J.; Salach, R.; Steffen, D.; Virius, M.; Paul, S. FPGA based event building and data acqiusition system for the COMPASS experiment. 2015 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC); IEEE: San Diego, CA, USA, 2015; pp. 1–2. doi:10.1109/NSSMIC.2015.7581844.
10. Khedkar, A.A.; Khade, R. High speed FPGA-based data acquisition system. *Microprocessors and Microsystems* **2017**, *49*, 87–94. doi:10.1016/j.micpro.2016.11.006.
11. Bai, Y.; Gaisbauer, D.; Huber, S.; Konorov, I.; Levit, D.; Steffen, D.; Paul, S. Intelligent FPGA Data Acquisition Framework. *IEEE Transactions on Nuclear Science* **2017**, *64*, 1219–1224. doi:10.1109/TNS.2017.2708510.
12. Shu, S.; Wang, L.; Liu, D.; Chen, M.; Zhang, Y.; Luo, J.; Ji, F. A high-speed data acquisition system based on FPGA for tokamak. *Review of Scientific Instruments* **2018**, *89*, 10K120. doi:10.1063/1.5035364.
13. Herrero, R.; Carpeno, A.; Esquembri, S.; Ruiz, M.; Barrera, E. FPGA-Based Solutions for Analog Data Acquisition and Processing Integrated in Area Detector Using FlexRIO Technology. *IEEE Transactions on Nuclear Science* **2018**, *65*, 781–787. doi:10.1109/TNS.2017.2782827.
14. Zabołotny, W.M.; Kasprowicz, G.; Poźniak, K.; Chernyshova, M.; Czarski, T.; Gąska, M.; Kolasiński, P.; Krawczyk, R.; Linczuk, P.; Wojeński, A. FPGA and Embedded Systems Based Fast Data Acquisition and Processing for GEM Detectors. *Journal of Fusion Energy* **2018**. doi:10.1007/s10894-018-0181-2.
15. Zabołotny, W.M.; Bartkiewicz, D.; Bluj, M.; Buńkowski, K.; Byszuk, A.; Doroba, K.; Górski, M.; Kalinowski, A.; Kierzkowski, K.; Konecki, M.; Królikowski, J.; Okliński, W.; Olszewski, M.; Poźniak, K. FPGA implementation of overlap MTF trigger: preliminary study. Proc. SPIE; Romaniuk, R.S., Ed.; , 2014; Vol. 9290, p. 929025. doi:10.1117/12.2073380.
16. SoC Interconnection: WISHBONE. https://opencores.org/howto/wishbone. [Online; accessed 3-October-2021].
17. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification. [Online; accessed 3-October-2021].
18. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. https://cdn.opencores.org/downloads/wbspec_b3.pdf. [Online; accessed 3-October-2021].
19. Wishbone B4, WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. https://cdn.opencores.org/downloads/wbspec_b4.pdf. [Online; accessed 3-October-2021].
20. LibreCores Project List. https://www.librecores.org/project/list. [Online; accessed 3-October-2021].
21. Projects OpenCores. https://opencores.org/projects. [Online; accessed 3-October-2021].
22. Open Hardware Wishbone modules. https://ohwr.org/project/general-cores/tree/master/modules/wishbone. [Online; accessed 3-October-2021].
23. Larrea, C.G.; Harder, K.; Newbold, D.; Sankey, D.; Rose, A.; Thea, A.; Williams, T. IPbus: a flexible Ethernet-based control system for xTCA hardware. *Journal of Instrumentation* **2015**, *10*, C02019–C02019. doi:10.1088/1748-0221/10/02/C02019.

24. Building the perfect AXI4 slave. https://zipcpu.com/blog/2019/05/29/demoaxi.html. [Online; accessed 3-October-2021].
25. AMBA AXI4-Lite Interface Specification. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI4-Lite-Interface-Specification. [Online; accessed 3-October-2021].
26. Buidilng an AXI-Lite slave the easy way. https://zipcpu.com/blog/2020/03/08/easyaxil.html. [Online; accessed 3-October-2021].
27. SystemRDL 2.0 Register Description Language. https://www.accellera.org/images/downloads/standards/systemrdl/SystemRDL_2.0_Jan2018.pdf. [Online; accessed 3-October-2021].
28. UVM Register Model Generator, SystemRDL Compiler, IP-XACT Compiler. https://www.agnisys.com/products/idesignspec-uvm-register-generator/. [Online; accessed 3-October-2021].
29. open-register-design-tool. https://github.com/Juniper/open-register-design-tool. [Online; accessed 3-October-2021].
30. SystemRDL - Free & open-source SystemRDL tools. https://github.com/systemrdl. [Online; accessed 3-October-2021].
31. Pozniak, K.T.; Bartoszek, M.; Pietrusinski, M. Internal interface for RPC muon trigger electronics at CMS experiment. 2004, pp. 269–282. doi:10.1117/12.568902.
32. Koprek, W.; Kaleta, P.; Szewinski, J.; Pozniak, K.T.; Czarski, T.; Romaniuk, R.S. Software layer for FPGA-based TESLA cavity control system; , 2005; p. 32. doi:10.1117/12.610572.
33. Poźniak, K.T. Internal Interface I/O communication with FPGA circuits and hardware description standard for applications in HEP and FEL electronics ver. 1.0, 2005. Available from the website https://flash.desy.de/reports_publications/tesla_reports/tesla_reports_2005/ acessed 18-August-2021.
34. Drabik, P.; Pozniak, K.T. Maintaining complex and distributed measurement systems with component internal interface framework. Proc. SPIE; Romaniuk, R.S.; Kulpa, K.S., Eds.; , 2009; Vol. 7502, p. 75022C. doi:10.1117/12.838155.
35. Zagoździńska, A.; Poźniak, K.T.; Drabik, P.K. Selected issues of the universal communication environment implementation for CII standard; , 2011; p. 80080N. doi:10.1117/12.902748.
36. Zabołotny, W.M.; Byszuk, A.; Emschermann, D.; Gumiński, M.; Kasprowicz, G.; Lehnert, J.; Loizeau, P.A.; Mueller, W.; Pozniak, K.T.; Romaniuk, R. GBT oriented firmware for Data Processing Boards for CBM. Proceedings of Topical Workshop on Electronics for Particle Physics — PoS(TWEPP2018); Sissa Medialab: Antwerp, Belgium, 2019; p. 067. doi:10.22323/1.343.0067.
37. Zabolotny, W.; Byszuk, A.; Guminski, M.; Kasprowicz, G.; Michal, K.; Pozniak, K.; Romaniuk, R. DPB/CRI firmware development. In *CBM Progress Report 2018*; Friese, V.; Selyuzhenkov, I., Eds.; GSI Helmholtzzentrum fuer Schwerionenforschung, GSI, Darmstadt: Darmstadt, Germany, 2019; p. 140. doi:10.15120/GSI-2019-01018.
38. Adr_gen - automatic address generator. https://github.com/wzab/wzab-hdl-library/tree/master/addr_gen. [Online; accessed 3-October-2021].
39. uHAL quick tutorial, creating an address table. https://ipbus.web.cern.ch/doc/user/html/software/uhalQuickTutorial.html#creating-an-address-table. [Online; accessed 3-October-2021].
40. Wishbone slave generator. https://ohwr.org/projects/wishbone-gen. [Online; accessed 3-October-2021].
41. Register Tool. https://docs.opentitan.org/doc/rm/register_tool/. [Online; accessed 3-October-2021].
42. hdlregs by adrianf0 - A Python-based HDL register file generator. https://www.librecores.org/adrianf0/hdlregs. [Online; accessed 3-October-2021].
43. rggen by taichi-ishitani - Code generation tool for control/status registers in a SoC design. https://www.librecores.org/taichi-ishitani/rggen. [Online; accessed 3-October-2021].
44. Register Generator by tudor-timi – IP-XACT based register generator. https://www.librecores.org/tudor-timi/rgen. [Online; accessed 3-October-2021].
45. Cheby. https://gitlab.cern.ch/cohtdrivers/cheby/-/wikis/home. [Online; accessed 3-October-2021].
46. Zabolotny, W.M.; Gumiński, M.; Kruszewski, M. Automatic management of local bus address space in complex FPGA-implemented hierarchical systems. Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2019; Romaniuk, R.S.; Linczuk, M., Eds.; SPIE: Wilga, Poland, 2019; p. 23. doi:10.1117/12.2536259.
47. Bowman, J. The J1 Forth CPU. https://www.excamera.com/sphinx/fpga-j1.html, 2010. [Online; accessed 3-October-2021].
48. Bowman, J. The J1B source code. https://github.com/jamesbowman/swapforth/blob/master/j1b/verilog/j1.v, 2010. [Online; accessed 3-October-2021].
49. Forth based system for AFCK board initialization and diagnostics. https://github.com/wzab/AFCK_J1B_FORTH. [Online; accessed 3-October-2021].
50. Forth based system for KCU116 board initialization and diagnostics. https://gitlab.com/WZab/kcu116_j1b. [Online; accessed 3-October-2021].
51. Brodie, L. Thinking Forth. http://thinking-forth.sourceforge.net. [Online; accessed 23-October-2021].
52. Bowman, J. Swapforth. https://github.com/jamesbowman/swapforth. [Online; accessed 3-October-2021].
53. Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency. [Online; accessed 3-October-2021].
54. Large FPGA Methodology Guide Including Stacked Silicon Interconnect (SSI) Technology. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf. [Online; accessed 3-October-2021].
55. Large FPGA Methodology Guide Including Stacked Silicon Interconnect (SSI) Technology. [Online; accessed 3-October-2021].
56. FuseSoC. https://github.com/olofk/fusesoc. [Online; accessed 3-October-2021].

57. Zabołotny, W.M. Version control friendly project management system for FPGA designs. Proceedings of SPIE; Romaniuk, R.S., Ed. SPIE, 2016, p. 1003146. doi:10.1117/12.2247944.
58. On-chip bus. https://ipbus.web.cern.ch/doc/user/html/firmware/bus.html. [Online; accessed 3-October-2021].
59. JTAG to Wishbone bridge. https://github.com/wzab/wzab-hdl-library/tree/master/jtag2wb. [Online; accessed 3-October-2021].
60. AXI PCI Express (PCIe) Gen 3 Subsystem. https://www.xilinx.com/products/intellectual-property/axi_pcie_gen3.html. [Online; accessed 3-October-2021].
61. Versatile-DMA1. https://gitlab.com/WZab/versatile-dma1. [Online; accessed 3-October-2021].
62. Zabolotny, W.; Guminski, M.; Michal, K.; Miedzik, P.; Pozniak, K.; Romaniuk, R. FPGA-related development for CBM DAQ. In *CBM Progress Report 2020*; Senger, P.; Friese, V., Eds.; GSI Helmholtzzentrum fuer Schwerionenforschung, GSI, Darmstadt: Darmstadt, Germany, 2021; pp. 149–150. doi:10.15120/GSI-2021-00421.
63. GBT-SC module for FPGA. https://gitlab.cern.ch/gbtsc-fpga-support/gbt-sc. [Online; accessed 3-October-2021].
64. Moreira, P.; Marchioro, A.; Kloukinas, K. The GBT, a proposed architecture for Multi-Gb/s data transmission in high energy physics. Proceedings of the Topical Workshop on Electronics for Particle Physics, TWEPP 2007; , 2007; pp. 332–336.
65. AFCK_J1B. https://gitlab.com/WZab/afck_j1b. [Online; accessed 3-October-2021].
66. Zabołotny, W.; Byszuk, A.; Gumiński, M.; Kasprowicz, G.; Kruszewski, M.; Poźniak, K.; Romaniuk, R.; Emschermann, D.; Schmidt, C. GBTX Emulation for BM@N/MPD Data Acquisition Systems. *Acta Physica Polonica B Proceedings Supplement* **2021**, *14*, 555. doi:10.5506/APhysPolBSupp.14.555.
67. Zabołotny, W.M.; Byszuk, A.P.; Dementev, D.; Emschermann, D.; Gumiński, M.; Kruszewski, M.; Miedzik, P.; Poźniak, K.; Romaniuk, R.; Schmidt, C.J.; Shitenkov, M. GBTX emulator for development and special versions of GBT-based readout chains. *arXiv:2109.11591 [physics]* **2021**. arXiv: 2109.11591.
68. AGWB (Address Generator for Wishbone) - register access for hierarchical Wishbone connected systems. https://github.com/wzab/agwb. [Online; accessed 3-October-2021].
69. Solderpad Hardware Licence v2.0. http://solderpad.org/licenses/SHL-2.0/. [Online; accessed 3-October-2021].