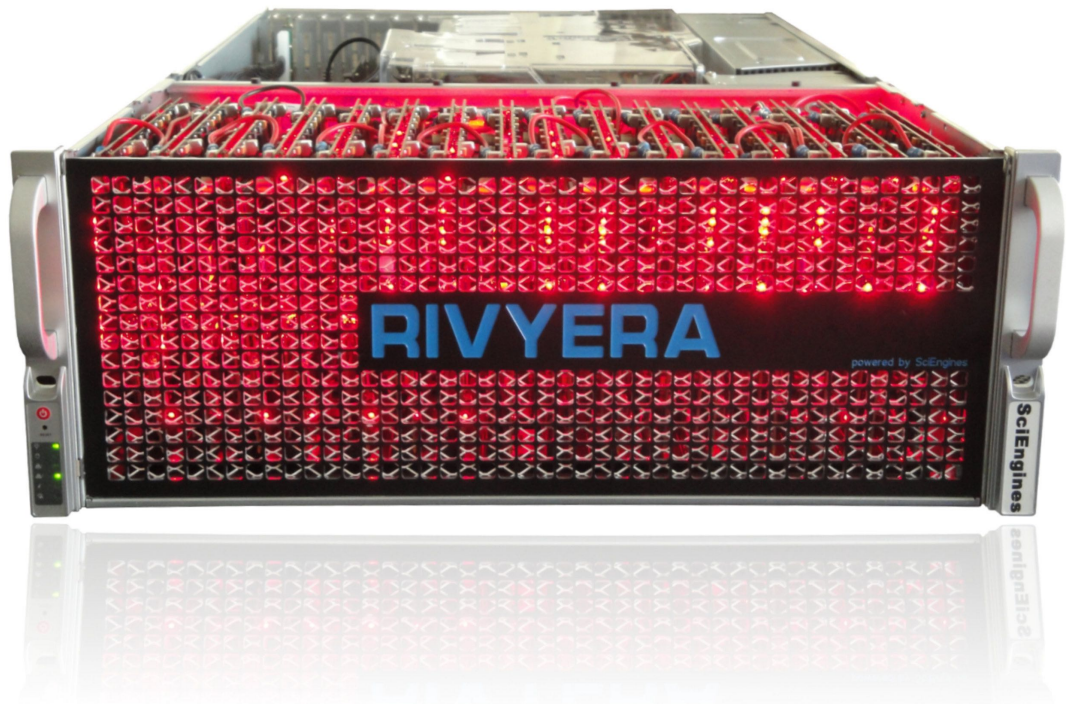# SciEngines
massively parallel computing

RIVYERA API

SciEngines RIVYERA Machine-API Documentation

Development User Guide, Machine-API (VHDL)

Version 1.95.00

Released version

SciEngines RIVYERA Machine-API Documentation

Development User Guide, Machine-API (VHDL)
Version 1.95.00

SciEngines GmbH

SciEngines GmbH
Am-Kiel-Kanal 2
24106 Kiel
Germany

Public

Released version

Prepared for   SciEngines GmbH, Am-Kiel-Kanal 2, 24106 Kiel, Germany
     Under   No Contract Number

Abstract: This introduction offers a brief overview of the SciEngines RIVYERA computer. It describes the physical and structural details from the programmers' point of view.

The main purpose of the RIVYERA API is to interface with single and multiple FPGAs in a massively parallel architecture as simply and easily as possible. We intended to provide an infrastructure for your FPGA designs which allows to leverage the benefits of a massively parallel architecture without raising the complexity of your design.
Therefore, we provide a simple interface hiding the idiosyncratic implementation details of the physical layers while permitting a high-level view of your RIVYERA computer.

Contents

Figures and Tables

Figures

Tables

1 General

1.1 Basic Information

This introduction offers a brief overview of the SciEngines RIVYERA computer. It describes the physical and structural details from the programmers' point of view.

The main purpose of the RIVYERA API is to interface with single and multiple FPGAs in a massively parallel architecture as simply and easily as possible. We intend to provide an infrastructure for your FPGA designs which allows you to leverage the benefits of a massively parallel architecture without raising the complexity of your design.
Therefore, we provide a simple interface hiding the idiosyncratic implementation details of the physical layers while permitting a high-level view of your RIVYERA computer.

1.1.1 General ideas of parallel programming

Traditionally, software has been written for serial computation. There are two historic reasons for serial computation concepts: one is that thinking in a **serial**, causal way is easy for most humans, the other is that computers started mechanically. Still during the early 1980s, the most common way to input data or programs was via punched tape or magnetic tape drives. Most of today's computers are **von Neumann architectures**. Named after the Hungarian mathematician John von Neumann who first stated the general requirements for an electronic computer in his 1945 papers. Since then, virtually all computers have followed this basic design, which differed from earlier computers programmed through '*hard wiring*'. Standard CPUs are designed to provide a good instruction mixture for almost all commonly used algorithms. Therefore, for a class of target algorithms they cannot be as effective as possible in terms of design freedom. Most software is intended to be run on such general purpose computers having one single central processing unit (*CPU*). A problem is split into a discrete series of instructions, each instruction is executed one after the other and only a single instruction may be executed at a time.

The SciEngines approach follows a massively parallelized architectural concept. It provides a large number of Field Programmable Gate Arrays (*FPGAs*), which are able to implement a huge number of individual processing elements. In the simplest case, **FPGA parallel computing** is the simultaneous use of multiple resources like processing elements to solve large computational problems. The RIVYERA API allows to interface hundreds of such processing elements per FPGA. To solve a complex task, it is split into discrete parts that can be solved concurrently. Each part is computed in its own processing element. Unlike a classical CPU, the discrete parts are further split to a series of instructions which are executed in highly problem-optimized dedicated hardware. This hardware task is coded in the hardware description language VHDL. The instructions from each part are executed simultaneously on different processing elements and FPGAs.

General computational problems usually demonstrate characteristics such as the ability to be split into discrete pieces of work that can be solved simultaneously and execute multiple program instructions at any moment in time. Therefore, problems are solved in less time with SciEngines RIVYERA than with a single computational resource like a CPU.

1.1.2 Concept of using SciEngines RIVYERA

To efficiently use SciEngines RIVYERA, the computational problem or algorithm is split in two general parts (see figure 1). One part is the strict software or frontend part which remains on the integrated host PC inside the RIVYERA computer. The other part is the core algorithm

which is accelerated by using the FPGAs on a single RIVYERA computer or even on multiple RIVYERA computers. The FPGAs programmable by the user are referred to as *UserFPGAs*.
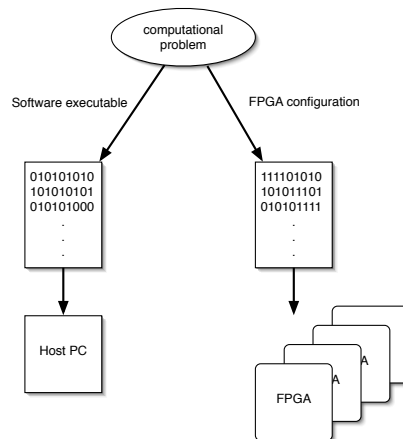
Figure 1. Partitioning of a problem into host- and machine-parts

In general, the software part could be seen as a frontend for the user or as a data interface to provide the resources for the FPGA accelerated parts. Also, simple pre- or post-computations are ideal for this part. The RIVYERA Host-API offers a rich set of interface functions which can be easily used by existing code.

CAUTION

In a massively parallel architecture the **flow control** should always be a point to think about. To achieve the best speedup, the flow control should be done **within the Machine-API**, e.g. by designing a special FPGA entity. Compared to FPGA architectures, PC architectures react much slower, because incoming events always have to be analyzed by schedulers, memory managers and other OS components. Therefore, the programmer always adds an artificial delay when allowing the FPGAs to wait for a PC reaction. Flow control in your PC software using the Host-API is still fast and quick to implement but might not result in the speedup your design is capable of.

The second part implements the acceleration, flow control and multiple processing elements to solve the computational problem. The RIVYERA Machine-API offers useful functions which easily allows you to implement the key parts of the algorithm.

To create the host part and the machine part of your application, different software tools are useful. On the host side, high level languages such as C or C++ and even Java are addressed by the RIVYERA Host-API. In order to design efficient processing elements, VHDL or Verilog is recommended. Implementations using cross-language compilers like SystemC are possible, but will most likely not result in the expected speedups.

In order to move any suitable computational problem to the RIVYERA computer, the computational problem should be partitioned into the two mentioned parts (see figure 2). For the integrated frontend on the host PC, the usage of any suitable compiler and development environment will create adequate results. The recommended tools are Eclipse for the IDE and the Gnu C Compiler (*gcc*) or any comparable Unix based compiler in order to create executable code on the integrated RIVYERA Host PC [1]. Machines shipped with Unix based operating systems, like Linux, usually provide a pre-installed gcc or equivalent compiler. All available RIVYERA computers provide templates for several programming languages like C/C++ or Java.

---

[1] RIVYERA API has been tested with Linux/gcc. Other compilers may work but are not officially supported.
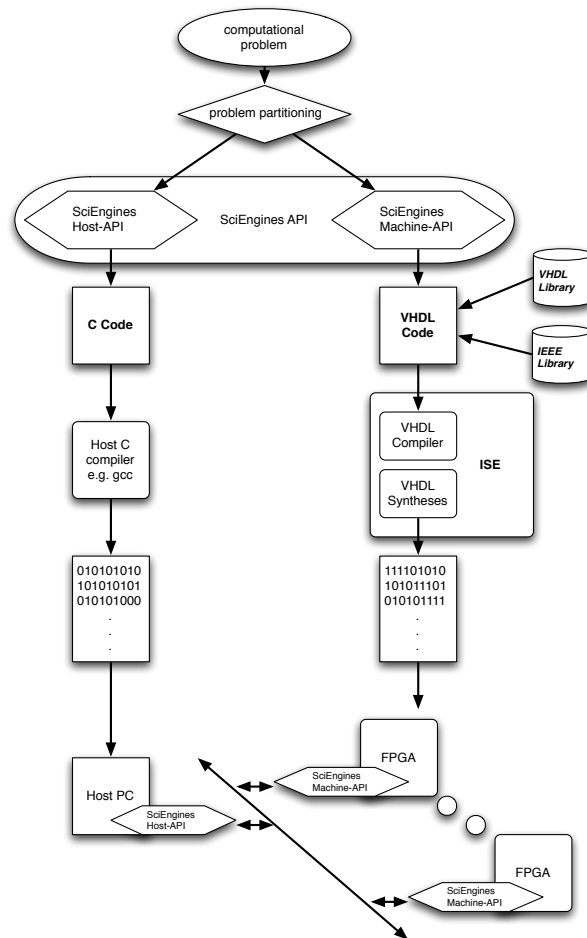
Figure 2. Design flow for multi-component software systems

For FPGA design and implementation, the recommended development environments for the differing RIVYERA architectures are:

- `RIVYERA S6-LX150`: XILINX® ISE® development environment.

- `RIVYERA X-32G1`: INTEL® QUARTUS® development environment.

Most third party compilers and IDEs might work as there are no other templates included except the ones provided for ISE® and QUARTUS®. Using the RIVYERA Machine-API allows simple interfacing of your VHDL-implemented processing elements.

1.1.3   API version information

The SciEngines API follows a simple versioning scheme. All API versions are denoted `aa.bb.cc s` with the symbols as follows.

- `aa`: **Major API version**
  Major API version changes indicate that the complete code structure will have to be changed if migrating. A changing Major version often indicate complete restructurings of the APIs code and therefore have a very long interval.

- `bb`: **Minor API version**
  A change in the API minor version will be triggered by new features.

- `cc`: **API Service Pack** (sometimes abbreviated with *SP*)
  The API Service Pack will increase if there have been bug fixes.

- `s`: **API revision string**
  The revision string can be an arbitrary string annotating the version. For example, "*RC1*" as a revision string may indicate that this is the *first release candidate* of a new API version.

Within this scheme, there is one specific caveat: All versions with $bb \geq 90$ are pre-release versions of a higher major version. For example, API 1.90.00 was the first alpha version of API 2.00.00.

1.1.4 RIVYERA API Addressing Scheme

The addressing scheme in the RIVYERA API is straightforward. Every single data word travels through the machine containing two addresses. One of these (the so called *target*) contains information where it should be sent to, the other one (so called *source*) tells the receiver where this word originated. Each address is built from multiple components which will be explained below.

Physical Address Components

To gain highest possible flexibility, every FPGA in the whole RIVYERA is uniquely identifiable and can therefore be addressed individually. The addressing scheme contains two physical fields: *Slot* and *FPGA address*. These fields are derived from the physical machine structure. Every RIVYERA computer physically consists of one or more FPGA Cards, each of which is plugged into a backplane slot. All plugged cards are numbered from index `0` to index `CARD_COUNT-1`, retaining their physical order. The index of each card is called its slot index. Multiple FPGAs may reside on each card. Similar to the cards in one system, the FPGAs are numbered in order, starting at index 0 as well. However, all FPGAs on one card share the same slot index. Using both the slot and FPGA index, every FPGA may be addressed uniquely throughout a whole RIVYERA computer.

Address Wildcards

Physical Address Components may be replaced by wildcards, such as `ADDR_SLOT_ALL` or `ADDR_FPGA_ALL`. Using these wildcards, it is possible to create broadcast- or very simple multicast-addresses. For example `slot=ADDR_SLOT_ALL, fpga=0` refers to the first FPGA on all cards, whereas `slot=0, fpga=ADDR_FPGA_ALL` selects all FPGAs on slot 0. `slot=ADDR_SLOT_ALL, fpga=ADDR_FPGA_ALL` of course selects every FPGA on every slot.

Virtual Address Components

The addressing scheme is completed by two more fields: *command* and *register*. Both fields do not have any physical means but are only useful for communication. The *command* field may contain one of *read* or *write*. *Write* commands do not imply a dedicated behavior on the FPGA side, whereas *read* commands assume a proper answer. Please see section 2.5.1 (Responding to Read Requests) in the VHDL-documentation for more information. The *register* address field **MAY** be used to create multiple data streams. It can be considered as a stream identifier. As both sent and received words always contain information about their source and target register the user can leverage a very powerful feature to create and design his very own data-flows. A very common way to use the *register* field is to employ different types of streams for each *register*. For example, consider an FPGA design which has two calculation cores which have to be fed with independent data. In this example, it would make sense to use register 0 for core 1 and register 1 for core 2. Please note that using multiple registers does not affect communication bandwidth.

Target Addresses

A target address specifies where a given data word is to be delivered to and how the target shall interpret the incoming word. For example, incoming words with `api_i_tgt_cmd_out = CMD_WR` tells the target FPGA that the sender does not expect an answer. Whenever

`api_i_tgt_cmd_out = CMD_RD` your user logic is expected to send a number of words specified in `api_i_data_out` back to the sender.

Please note that as a receiver, you will not see the target slot and FPGA fields of an incoming word, because these are given implicitly by data receipt.

Source Addresses

Source addresses contain information about the source of an incoming data word. While a source's slot and FPGA information is straightforward, the *command* and *register* fields are more complex to understand. In general, both *source command* and *source register* do not have to be taken into account. Whenever the user FPGA receives data from the host interface, the *source command* will be `CMD_WR` and the *source register* will be set to `0x0`. However, you are free to implement designs that effectively use these fields within inter-FPGA communication, for example to tell the receiver to send responds to a defined target address.

1.2   RIVYERA API Structure

In the RIVYERA architecture all data uses the same transport channel and in order to maintain the correctness of order, data frames are not allowed to overtake each other. These specific features have to be kept in mind when designing your code for RIVYERA.

1.2.1   RIVYERA API Register Paradigm
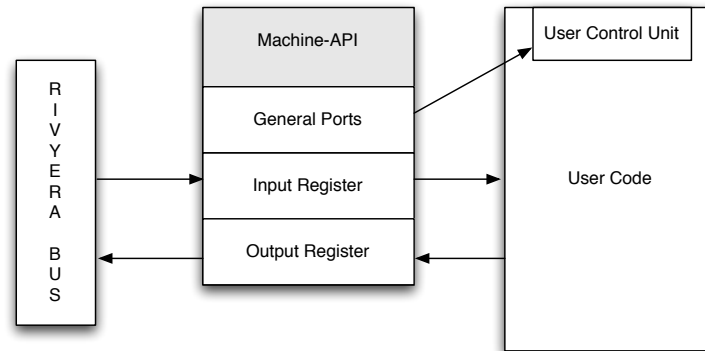
Figure 3. VHDL-API taking care of user design's I/O

Figure 3 shows the block diagram of one example of an FPGA design. The host interface provided by the Machine-API is instantiated once and connects to an addressed FPGA. This design paradigm will be modeled by the Machine-API and, accordingly, by the Host-API.

- Input Register

   The SciEngines RIVYERA API enables the user to send and receive streamed data to and from an FPGA. Using this mechanism, it is possible to send data from host to one or multiple FPGAs as well as transfer data between FPGAs and send data from FPGAs to the host. A stream consists of individual 64 bit data words which are transferred in order. This means: words written earlier to an FPGA arrive earlier than words which are written later.

- Output Register

   The SciEngines RIVYERA API provides a single register which can be used to send data. Whenever the user wants to send data to either the host PC or any other (possibly multiple) FPGA(s), he may provide data to this output register.

Both Input and Output Register are realized as BlockRAM FIFOs.

1.2.2   RIVYERA API Routing Strategies

SciEngines API will support multiple routing schemes, so the RIVYERA can be adapted according to each user's needs. Currently, the only supported routing scheme is Smart Routing. All routing strategies are strictly deterministic. Therefore, every sent word takes exactly the same path through the RIVYERA, depending on its physical source and target address. SciEngines API does not avoid links with high traffic.

Smart Routing

The Smart Routing strategy, which is enabled by default, will determine the shortest route through the RIVYERA for every sent word. It will make full usage of the machine's architecture with its card-to-card shortcuts.

Broadcasted transfers will automatically be spread in both communication directions to reduce the worst-case latency. The following illustrations show one FPGA card with 8 FPGAs. The sender of a word is always colored in bright green, whereas the links that are used to pass a word are highlighted red. Please note that exactly the same routing method applies to FPGA cards with different numbers of FPGAs.

Figure 4 depicts the route of a word written to all FPGAs by the Host application. The host-connected Service FPGA duplicates the word and sends it to its User FPGAs using both ring directions. All FPGAs but numbers 3 and 4 do both: forwarding the incoming word to their successors and forwarding it to the internal user User Logic. The FPGAs 3 and 4 forward the word to their own user logic, but do <u>not</u> forward it to the next FPGA. Therefore, no FPGA gets the word twice.
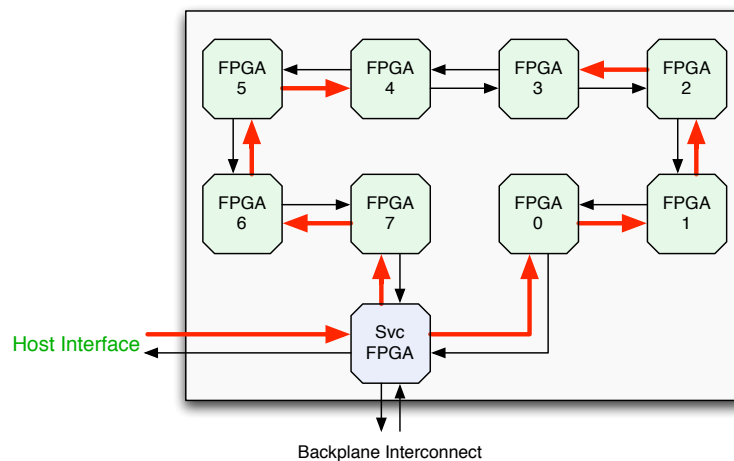
Figure 4. Routing of a host-initiated write

The same principle of routing applies for FPGA $\leftrightarrow$ FPGA transfers as shown in Figure 5. If an FPGA issues a broadcast, then it is broadcasted in both directions and it is assured by the API that no FPGA gets the same word twice.
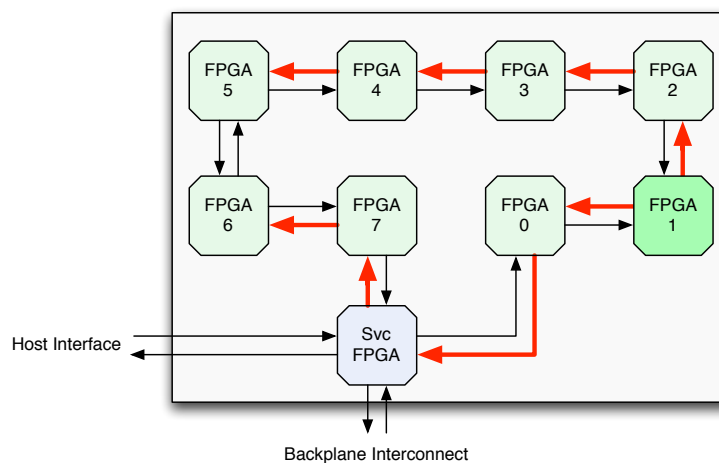
Figure 5. Routing of an FPGA-initiated write

1.3   VHDL API Introduction

The following sections will give a more detailed overview regarding the HDL-API-Component.

1.3.1   Introduction

The SciEngines RIVYERA API provides easy access to the communication features that the RIVYERA is capable of. The handling of the I/O registers is similar to the handling of Xilinx® FIFO components. However, there are some slight differences in usage and behaviour of SciEngines API components.

Features
The SciEngines RIVYERA API is a precompiled netlist (softmacro) and provides the following features:

- Complete handling of all physical I/O-Layers and routing procedures

- Bidirectional (Full-Duplex) communication throughout the whole RIVYERA machine

- Fully asynchronous input and output register

- Support of reading and writing from and to every FPGA on the entire machine

- Up to 400 MB/s useable communication bandwidth per interconnect[2]

The VHDL-part of the SciEngines RIVYERA API comes within a single, precompiled entity which has to be instantiated by designs running on RIVYERA machines.

1.3.2   API instantiation and HDL design flow

All the functionality of the SciEngines RIVYERA API is shipped within one single block. You can consider it as a black box, that handles all the FPGAs I/O Pins and provides the interface described in this document to your user code's side. This black box is called a "Macro". Thinking in terms of VHDL it is simply an entity that is capable of all the things necessary to operate the RIVYERA. Figure 6 shows the general HDL-Design-Flow.

On its way to a programmable bitfile, every sourcecode has to pass the following steps:

- Synthesis
  Within synthesis, the sourcecode is synthesized and transformed into an RTL Netlist. The resulting netlist is often referred to as a Softmacro.

- Translation
  The general netlist is translated into a Vendor specific netlist.

- Map
  In Map phase, the netlist is mapped to the available logic on the target chip.

- Place & Route
  In Place & Route (*PAR*), the gates are placed and routes are determined. Placed & Routed Macros (Hardmacros) are of course excluded from PAR and are only taken in as given blocks.

Because the HDL-API-Module comes as a softmacro, it is first opened in the translation step. The SciEngines API module ships in a `.edf` format and will be translated to `.ngo` format in the translation step. Please note that if your API Module changes on the file system, Xilinx tools DO NOT update the translated netlist (`.ngo`) by themselves. The only way to force Xilinx tools to update the translated netlist is to remove the `SciEngines_API.ngo` file in

---

[2]Depending on available device
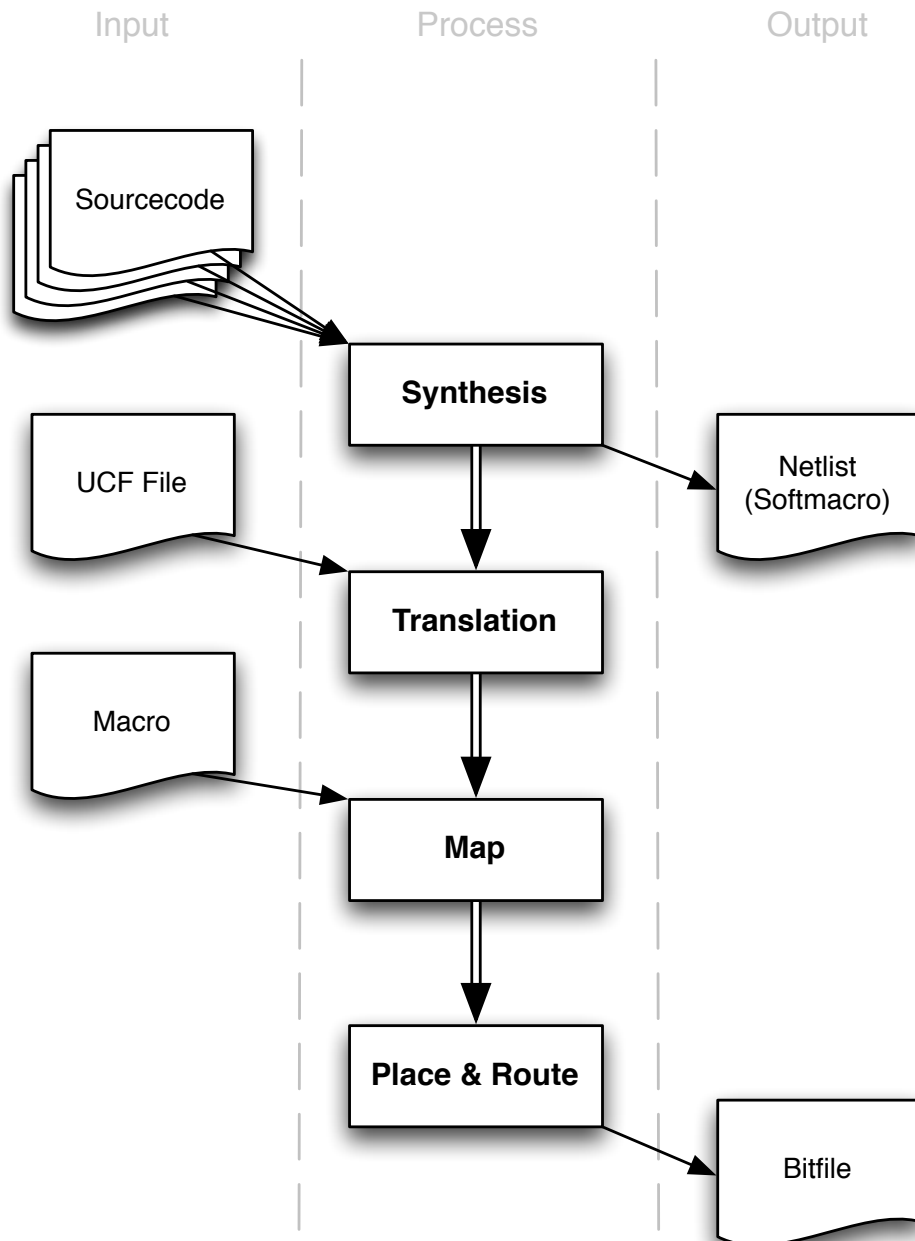
Figure 6. General HDL Design Flow

the project directory manually.

After your design passes all the design steps, a programming file (referred to as *bitfile*) is generated. SciEngines API supports either binary bitfiles (`.bit`) or ASCII bitfiles (`.rbt`). In order to not accidentally configure the FPGAs with a wrong bitfile, SciEngines API checks the User ID contained in the bitfile which is depending on the active API version. Table 1 shows an overview of valid User IDs (X denotes a *Don't Care*). This UserID check has been introduced in the first alpha of the SciEngines API 2 (Version 1.90.00), so if you are using API 1 (below version 1.90.00) you can ignore the UserID setting. Projects generated with the SciEngines ProjectCompass will automatically set the correct User ID. If you want to create a project manually, you can set the User ID with the `bitgen -g UserID:` switch or graphically in the *Confguration Options* of the *Generate Programming File* properties.

1.3.3 Functional Description

API Version	User ID
<1.90.00	0xXXXXXXXX
<2.90.00	0x10000000

Table 1. Overview of version related Bitfile User IDs

The SciEngines RIVYERA API module handles the RIVYERA Bus traffic and provides communication features to the user's FPGA design.

The SciEngines RIVYERA API provides a 64-bit input register. The register will buffer all incoming data. This data will only be discarded if the user's FPGA design acknowledges its receipt. When the FPGA has data to send, the 64-bit output register may be used.

General Ports

As well as the ports used for communication, RIVYERA API contains additional ports for clocking, reset and additional information. Table 2 shows all general ports of the API-Component including a short description. All of these general RIVERA API ports are running at 100 MHz. Therefore, `api_clk_out`, provides a 100 MHz clock. RIVYERA API uses two of the FPGA's DCMs, so you are free to use all the others to create different clock domains. The clock output of the SciEngines API may be directly connected to another DCM (please select *No Buffer* as clock input Source when creating the DCM).

Note that after powerup, SciEngines RIVYERA API will need some time to initiate itself, so `api_rst_out` will be high initially. Whenever `api_rst_out` is low, your design may safely run. As long as `api_rst_out` is asserted, it is not safe to use any information contained in the general ports, because they may change during initialization.

The ports `api_self_slot` and `api_self_fpga` contain information about the FPGA's address.

The ports `api_*_contr_out` contain information about all controllers next to the FPGA's slot. If the slot of the FPGA has an interface to the host-PC, `api_self_contr_out` is asserted. `api_next_contr_out` provides the next controller'slot index, which is the first card with a higher slot index and a host-interface. If there is no next controller, then `api_next_contr_out` points to the previous controller if `api_self_contr_out` is not asserted or to its own slot index, otherwise. According to the next controller information, `api_prev_contr_out` provides the first card with a lower slot index and a host-interface. If there is no such card, it points to the next controller if `api_self_contr_out` is not asserted or to its own slot index, otherwise. If `api_self_contr_out` is asserted (meaning that the FPGA's card has a host-interface) and there is no other controller, both `api_prev_contr_out` and `api_next_contr_out` point to its own slot index.

The port *api_led_in* can be used to drive the LEDs connected to the FPGA.

Bit(s)	Name	Direction	Reset Value	Description
0	api_clk_out	Output	0	**Clock.** The API's clock output.
0	api_rst_out	Output	1	**Reset.** Reset output needed to reset the user's design.
1-0	api_led_in	Input	00	**LED.** Input to enable/disable the User-FPGA's LEDs. Tied to 0 if not set.
0	api_self_contr_out	Output	0	**Controller Flag.** Indicating wether there is a host interface connected to this slot or not.
9-0	api_next_contr_out	Output	0x0	**Next Controller Adress.** The slot address where the next host interface is located at.
9-0	api_prev_contr_out	Output	0x0	**Previous Controller Adress.** The slot address where the previous host interface is located at.
9-0	api_self_slot_out	Output	0x0	**Slot Address.** The FPGA's slot address.
4-0	api_self_fpga_out	Output	0x0	**FPGA Address.** The FPGA's FPGA address.

Table 2. General ports of the API-Component

CAUTION

`api_rst_out` should not be used as a global reset for all instantiated logic without design consideration. If you want to reset your logic according to `api_rst_out`, you have two options to consider:

- Adding a timing ignore attribute (`TIG`) to `api_rst_out`. Consideration should be given that the design might not start synchronously, meaning that some components may sense `user_rst_out` low earlier than other. This is simply because the design tools do not analyze `api_rst_out` anymore regarding its timing aspects, so the signal may have different runtime from its source to each component.

- Build a FlipFlop tree to distribute the signal to all your logic avoiding a huge fan-out. Building a FlipFlop tree is the more difficult way to handle a global reset, but the safer way, as well. With building a FlipFlop chain, you add an artificial delay to the signal, allowing it to reach every component at the same time. This, of course, makes your code start later than `api_rst_out` might indicate, but `api_rst_out=0` only means that your design *may* start to run, but not that it has to.

Input register

The input register is used for incoming data transfer. Its behaviour is similar to that of *First-Word-Fall-Through* FIFOs. For all incoming data words, the user's design has to acknowledge its receipt, so the API can make sure that the user's design does not miss any data.
All ports of the input register are synchronous to `api_i_clk_in`. This port **MUST** be connected in order to be able to receive data. It can be conencted to any desired clock, as long as timing closure can be reached. If unsure, simply connect `api_i_clk_in` to `api_clk_out`. This will operate the input register at the same clock as the general ports.
The presence of data in the Input register will be signalled by unasserted `api_i_empty_out`. As long as `api_i_empty_out` is not asserted, all output ports will provide valid data. `api_i_empty_out` will stay asserted as long as the Input register does not contain more than one word.
Once `api_i_empty_out` is low and the next incoming word is desired, `api_i_rd_en` should be asserted for one clock cycle. This leads to discard of the currently present word and presents the next word from the FIFO, if any. Please note that the input register has a delay of two clock cycles. If `api_i_rd_en_in` is asserted, the register content WILL NOT CHANGE in the very NEXT CLOCK CYCLE. Simple Implementations - as the given example code below - will therefore only sample the register's content if `api_i_rd_en_in = '0'` AND `api_i_empty_in = '0'` to make sure that the register is given time to update its content. For a performance optimization, please consider the use of `api_i_am_empty_in`.

Figure 7 illustrates the receipt of a read request from the controller interface at slot zero, followed by a write to register 2 by fpga 4 on slot 2.
The first word (`api_i_data_out = D1`) arrives at the Input register, indicated by `api_i_empty_out`

Bit(s)	Name	Direction	Reset Value	Description
0	api_i_clk_in	Input	-	**Clock.** All input register ports will by synchronous to this clock. It MUST be connected.
9-0	api_i_src_slot_out	Output	0x0	**Source Slot Address.** The slot address of the communication source
4-0	api_i_src_fpga_out	Output	0x0	**Source FPGA Address.** The FPGA address of the communication source
5-0	api_i_src_reg_out	Output	0x0	**Source Register Address.** The register address of the communication source
0	api_i_src_cmd_out	Output	0	**Source Command.** The source's command.
5-0	api_i_tgt_reg_out	Output	0x0	**Target Register Address.** The register address this word was targetted at communication source.
0	api_i_tgt_cmd_out	Output	0	**Target Command.** The command to be performed.
63-0	api_i_data_out	Output	0x0	**Data.** The data to be read.
0	api_i_empty_out	Output	0	**Empty.** This flag is indicating that there is no input available.
0	api_i_am_empty_out	Output	0	**Almost Empty.** This flag is indicating that there is only one more word of input available.
0	api_i_rd_en_in	Input	0	**Acknowledge.** This control signal acknowledges that the user core is aware of the new data and indicates that the data register can be freed.

Table 3. Input Register ports of the API-Component

= '0'. Because there is only one word inside the register, api_i_am_empty_out stay asserted. This changes in the very next clock cycle, because the second word arrives at the register. It now contains two words and therefore api_i_am_empty_out gets asserted. Two clock cycles after api_i_rd_en_in is triggered, the first word is discarded and the second word is provided at the interface. According to its definition, api_i_am_empty_out transits to high again, because the presented word is the only word present in the Input register at this time. After the second word gets read, too, api_i_empty_out gets asserted, indicating the there is no valid data present.
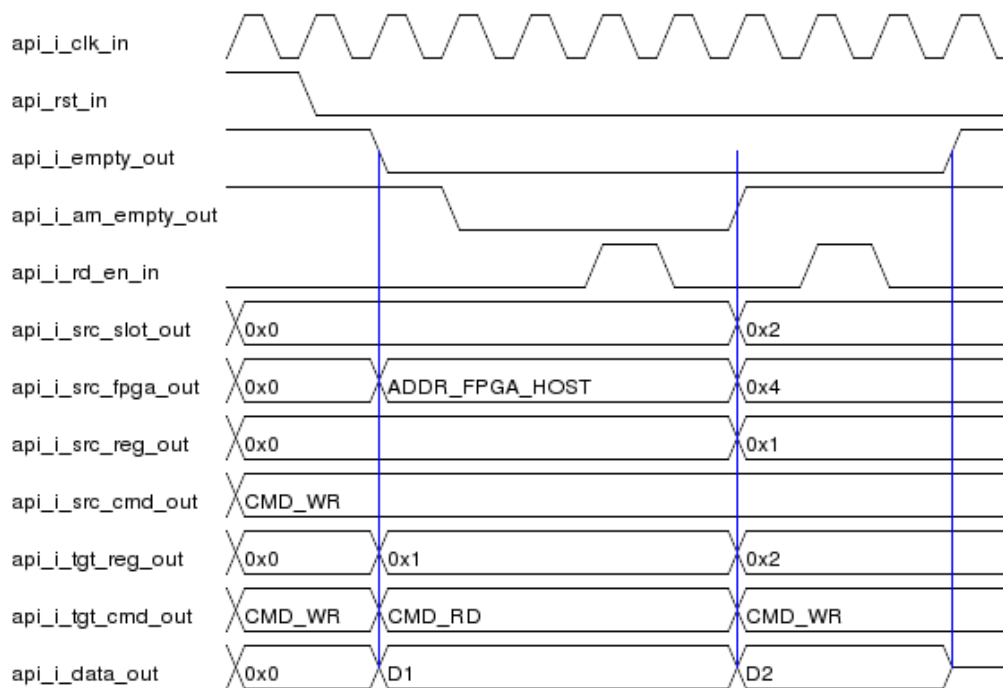
Figure 7. Input Register Timings

Output Register

If the FPGA needs to send data to any other FPGA or to the host, the output register has to be used. All ports of the output register are synchronous to api_o_clk_in. This port **MUST** be connected in order to be able to send data. It can be connected to any desired clock, as long as timing closure can be reached. If unsure, simply connect api_o_clk_in to api_clk_out. This will operate the output register at the same clock as the general ports.

Bit(s)	Name	Direction	Reset Value	Description
0	api_o_clk_in	Input	-	**Clock.** All output register ports will by synchronous to this clock. It MUST be connected.
9-0	api_o_tgt_slot_in	Input	0x0	**Target Slot Address.** The slot address of the communication target.
4-0	api_o_tgt_fpga_in	Input	0x0	**Target FPGA Address.** The FPGA address of the communication target.
5-0	api_o_tgt_reg_in	Input	0x0	**Target Register Address.** The register address of the communication target.
0	api_o_tgt_cmd_in	Input	0	**Source Command.** The command to be performed at the target.
5-0	api_o_src_reg_in	Input	0x0	**Source Register Address.** The register address of the communication source.
0	api_o_src_cmd_in	Input	0	**Target Command.** The command that an appropriate answer should be directed to.
63-0	api_o_data_in	Input	0x0	**Data.** The data to be sent.
0	api_o_wr_en_in	Input	0	**Write.** Initiates the send process.
0	api_o_rfd_out	Output	0	**Ready for data.** Flag for indicating if the API core is ready for data.

Table 4. Output Register ports of the API-Component

Whenever `api_o_rfd_out` is high, you may pass data to this register. Additionally, it is able to write data in the very next clock cycle, when `api_o_rfd_out` changes from high to low, because it is designed as an inverted *almost full flag*.

The usage of the output register is straightforward. After providing the data to `api_o_data_in`, target command to `api_o_tgt_cmd_in`, target address to `api_o_tgt_*_in`, source command and source register to `api_o_src_*_in`, everything is sent by setting `api_o_wr_en_in` to '1' for one clock cycle. Be sure to set `api_o_tgt_cmd_in` to the desired command.

Figure 8 illustrates the process of sending data to slot 1, FPGA 3, register 2 from register 5 with respect to the `api_o_rfd_in` flag.

Once `api_o_rfd_out` gets asserted, it is allowed to strobe the `api_o_wr_en_out` signal to send the data presented to the interface. In the example, the words `D1`, `D2`, `D3` and `D4` are sent within the first assertion period of `api_o_rfd_in`.
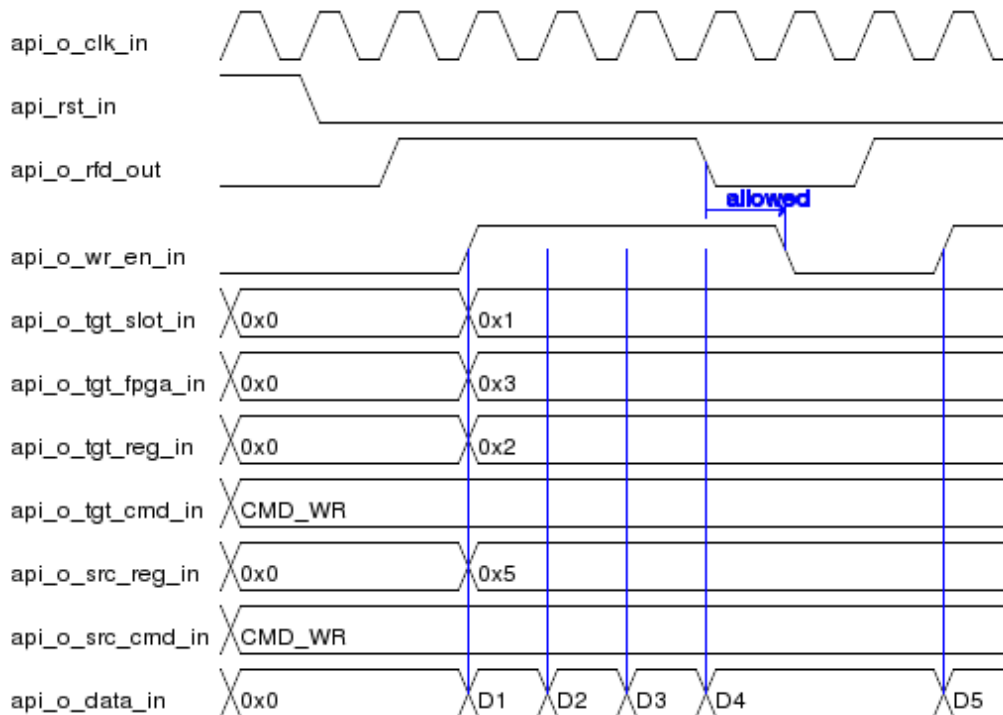

Figure 8. Output Register Timings

1.3.4 General Notes

The SciEngines API can handle much of the communication complexity, but not all of it. Hence your code also has to provide an appropriate functionality. Consider an incoming read request (`CMD_RD`): As soon as some component addresses your design, it is up to your code to react and send back an appropriate response.

Responding to Read Requests

Read requests are a special case in SciEngines RIVYERA API as they need to be serviced by your design. A read request is sent whenever a component waits for incoming data. Every read request (`CMD_RD`) has to be answered by sending a write request (`CMD_WR`) with the read request's source as new target and the read request's target as source. Therefore you need to set `api_o_src_reg_in` to the input target register `api_i_tgt_reg_out` and set `api_o_tgt_reg_in` to the input source register `api_i_src_reg_out`. It is easy to imagine that you have to *claim* that you respond as register `api_o_src_reg_in`, so when register 2 should be read and the response is expected to be written to register 5, then (`api_i_new_out = '1'`, `api_i_tgt_reg_out = "000010"`, `api_i_src_reg_out = "000101"` and `api_i_tgt_cmd_out = CMD_RD`) are set and you *claim* to respond as register 2 and therefore set `api_o_src_reg_in <= "000010"`, `api_o_tgt_reg_in <= "000101"`.
Whenever a read request occurs, `api_i_data_out` will contain the number of 64 bit words expected to be sent by your design, so when `api_i_data_out` equals `0x3`, your design is expected to send three words.

Initiating Read Requests

When your design initiates a read request, the request's target will react by writing data to the input source register that was specified in `api_o_src_reg_in` while sending the read request.

The number of words expected by the target to be returned needs to be put to `api_o_data_in` (e.g. when the target has to return three words, `api_o_data_in` is set to `0x3`).
Note: You can not request to read data from the host.

Host Data Transfers

In order to address the host system, simply set `api_o_tgt_fpga_in` to `ADDR_FPGA_HOST`. Additionally, you need to specify the correct target slot. Note that the card located in the target slot *MUST* be connected to a Host Controller. You will find the addresses of the surrounding controllers in `api_prev_contr_out` and `api_next_contr_out`. In general the UserFPGAs are reacting to host read requests, which occur at the input register with `api_i_tgt_cmd_out = CMD_RD`. If so, you can simply write back to the request's source slot. If this is not the case, and your design needs to write data to the host without being asked for it, please refer section Autonomous Writes.

Autonomous Writes

There might be some cases in which the FPGAs need to communicate with the host software without being requested to. For convenience, these FPGA write actions will be called

autonomous writes. Whenever your design needs to write data to the host without being asked for it, you need to specify FPGA `ADDR_FPGA_HOST` as target. Please note that in the current API Version you will not be able to use a wildcard to address the very next controller but you have to set the target controller's slot address by yourself. This can be done either by specifying a specific slot with an active PC connection or you may write data targeted for slot `ADDR_SLOT_ALL` to send the data to the very next controller. In the case of transfers to the controller, (ADDR_SLOT_ALL) does NOT denote a real broadcast but will be replaced by the next controller's slot. See the Host-API documentation for how to handle autonomous FPGA writes.

1.3.5   Example Code

Reading and writing to registers is straightforward since it does not differ from standard components provided by Xilinx®. In order to help you become familiar with the machine and to allow you to start programming designs right away, a brief introduction to the behaviour of the RIVYERA API components will be given.

Reading an Input Register

The most common case in communication will be reading incoming data from an input register. As we described briefly in section 1.3.3, you have to confirm the reading of every data package, otherwise the input register will block any further traffic.

```vhdl
                       Listing 1.1. Example code of how to read an input register
-- Register to store the source slot for later computation
signal src_slot : seSlotAddr_type := (others => '0');
-- Register to store the source command
signal src_cmd  : seCmd_type       := CMD_WR;
-- Register to store the target command
signal tgt_cmd  : seCmd_type       := CMD_WR;
-- Register to store the payload for later computation
signal data     : seData_type      := (others => '0');

input_proc : process
begin
   wait until api_clk_in = '1' and api_clk_in'event;

   -- Only proceed if new data available
   if api_i_empty_out = '0' and api_i_rd_en_in = '0' then
      -- Store data's source and command word
      src_slot <= api_i_src_slot_out;
      src_cmd  <= api_i_src_cmd_out;
      tgt_cmd  <= api_i_tgt_cmd_out;

      -- Store data if it is meant to be written.
      if api_i_tgt_cmd_out = CMD_WR then
         -- Store data for calculation
         data    <= api_i_data_out;
      end if;

   -- Confirm the receipt of data.
   api_i_rd_en_in <= '1';
   else
      -- If no data present, do not read.
      api_i_rd_en_in <= '0';
   end if;
end process;
```

Sending Data

In order to send data you have to be aware that data may only be sent if `api_o_rfd_out` is asserted. Otherwise any change to the port's signals will have no effect. After setting the target's address (including slot-, fpga- and register address), the desired command and the payload, the whole frame is written by setting `api_o_wr_en_in` to `'1'`. The API is ready for the next data word as soon as `api_o_rfd_out` is high again.

Listing 1.2. Example code of how to send data

```
output_proc : process
begin
   wait until api_clk_in = '1' and api_clk_in'event;

   -------- Set informations ---------
   -- Specify command (here: write command)
   api_o_tgt_cmd_in  <= CMD_WR;
   -- Set Address to Slot 3, FPGA 2, input register 4
   api_o_tgt_slot_in <= "0000000010";
   api_o_tgt_fpga_in <= "00001";
   api_o_tgt_reg_in  <= "000011";
   -- Send from register 1
   api_o_src_reg_in  <= "000001";
   api_o_src_cmd_in  <= CMD_WR;
   -- Set payload
   api_o_data_in     <= (others => '0');

   ---------- Try to write -----------
   -- Only write if API is ready.
   api_o_wr_en_in    <= api_o_rfd_out;
end process;
```

1.4 Namespace Documentation

1.4.1 sciengines_api_types Namespace Reference

This package contains all the types and constants used for the SciEngines RIVYERA API.

1.5 Class Documentation

1.5.1 sciengines_api_types Package Reference

This package contains all the types and constants used for the SciEngines RIVYERA API.

Constants

- LENGTH_ADDR_SLOT **positive:= 10**

    The length of a slot address.
- LENGTH_ADDR_FPGA **positive:= 5**

    The length of an FPGA address.
- LENGTH_ADDR_REG **positive:= 6**

    The length of a register address.
- LENGTH_ADDR **positive:= 21**

    The overall address length.
- LENGTH_CMD **positive:= 1**

    The length of command words.

- LENGTH_DATA **positive:= 64**

    The length of the payload.
- LENGTH_HW_REV **positive:= 8**

    The length of the hardware revision vector.

Data types for single words

Types

- **seBusFlag_type is array(natural range<>) of seFlag_type**

    Data type used for multiple flags.

Subtypes

- seFlag_type **std_logic**

    Data type used for single flags.
- seSlotAddr_type **std_logic_vector(LENGTH_ADDR_SLOT - 1 downto 0)**

    Data type for slot addresses.
- seFpgaAddr_type **std_logic_vector(LENGTH_ADDR_FPGA - 1 downto 0)**

    Data type for FPGA addresses.
- seRegAddr_type **std_logic_vector(LENGTH_ADDR_REG - 1 downto 0)**

    Data type for register addresses.
- seCmd_type **std_logic_vector(LENGTH_CMD - 1 downto 0)**

    Data type for command words.
- seData_type **std_logic_vector(LENGTH_DATA - 1 downto 0)**

    Data type for payload.
- seHwRev_type **std_logic_vector(LENGTH_HW_REV - 1 downto 0)**

    Data type for hardware revision information.

Addressing wildcards

Constants

- ADDR_SLOT_ALL **seSlotAddr_type :=(others=>' 1 ')**

    Slot wildcard.
- ADDR_FPGA_ALL **seFpgaAddr_type :=(others=>' 1 ')**

    FPGA wildcard.
- ADDR_FPGA_HOST **seFpgaAddr_type :=(0 =>' 0 ',others=>' 1 ')**

    Host FPGA constant.
- ADDR_REG_EOT **seRegAddr_type :=(others=>' 1 ')**

    EOT Register constant.

Command types

Constants

- CMD_RD **seCmd_type :=" 0 "**

    Read command.
- CMD_WR **seCmd_type :=" 1 "**

    Write command.

RIVYERA X32 Clocking definitions

Constants

- PLL_DEFAULT_M **pll_reconf_data_t :=(isOddDivision=>' 0 ',isBypass=>' 0 ',high-Div=>x" 03 ",lowDiv=>x" 03 ")**

  Default value for PLL M value (Multiply 6)

- PLL_DEFAULT_N **pll_reconf_data_t :=(isOddDivision=>' 0 ',isBypass=>' 1 ',high-Div=>x" 00 ",lowDiv=>x" 00 ")**

  Default value for PLL N value (Divide 1 / Bypass)

- PLL_DEFAULT_C **pll_reconf_data_t :=(isOddDivision=>' 0 ',isBypass=>' 0 ',high-Div=>x" 03 ",lowDiv=>x" 03 ")**

  Default valud for all PLL C valued (Divide 6)

Types

- **record: pll_reconf_data_tOddDivision:std_logic**
  isBypass:std_logic
  highDiv:std_logic_vector(7 downto 0)
  lowDiv:std_logic_vector(7 downto 0)

  Data type for PLL reconfiguration data.

- **pll_reconf_data_arr_tisarray(naturalrange<>)ofpll_reconf_data_t**

  Array of PLL reconiguration data.

- **record: pll_reconf_data_userm:pll_reconf_data_t**
  n:pll_reconf_data_t
  c:pll_reconf_data_arr_t (8 downto 0)

  Record containing all User PLL reconfiguration values.

- **record: pll_reconf_data_api_iom:pll_reconf_data_t**
  n:pll_reconf_data_t
  c:pll_reconf_data_t

  Record containing all IO PLL reconfiguration values.

FPGA types

Constants

- FPGA_none **seFpgaType_type :=" 0000 "**

  No or unknown FPGA.

- FPGA_xc3s1000_4ft256 **seFpgaType_type :=" 0001 "**

  Xilinx Spartan 3 1000, Speed Grade -4, Package FT256.

- FPGA_xc3s1500_4fg676 **seFpgaType_type :=" 0010 "**

  Xilinx Spartan 3 1500, Speed Grade -4, Package FG676.

- FPGA_xc3s5000_4fg676 **seFpgaType_type :=" 0011 "**

  Xilinx Spartan 3 5000, Speed Grade -4, Package FG676.

- FPGA_xc6slx75_3fg484 **seFpgaType_type :=" 0100 "**

  Xilinx Spartan 6 LX75, Speed Grade -3, Package FG484.

- FPGA_xc6slx150_3fg676 **seFpgaType_type :=" 0101 "**

  Xilinx Spartan 6 LX150, Speed Grade -3, Package FG676.

- FPGA_xc4vsx35_10ff668 **seFpgaType_type :=" 0110 "**

  Xilinx Virtex 4 SX35, Speed Grade -10, Package FF668.

- FPGA_10ax115h4f34e3sg **seFpgaType_type :=" 0111 "**

  Intel Arria10 GX 1150, Speed Grade -3, Package F34.

- FPGA_10ax032h4f34e3sg **seFpgaType_type :=" 1000 "**

  Intel Arria10 GX 320, Speed Grade -3, Package F34.

Subtypes

- seFpgaType_type **std_logic_vector(3 downto 0)**

  Datatype used for FPGA types.