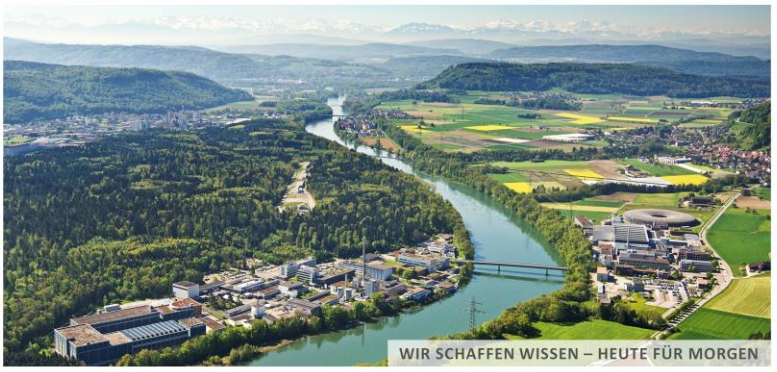
PAUL SCHERRER INSTITUT

PSI



Oliver Bründler :: FPGA Engineer :: Paul Scherrer Institut

psi_common

FPGA Library

30.11.2018

PAUL SCHERRER INSTITUT

Agenda

- Why using Libraries?
- Concepts in PSI Libraries
- psi_common Overview
- Conclusion

PAUL SCHERRER INSTITUT

Agenda

- **Why using Libraries?**
- Concepts in PSI Libraries
- psi_common Overview
- Conclusion

PAUL SCHERRER INSTITUT PSI

Why using Libraries?

+ Faster

+ Proven

+ Well Known

- Not everything covered

+ Benefit from Improvements

Page 4

Libraries are like Lego Bricks

Imagine you have to build a little doll-house. You may tend to use LEGO instead of building the doll-house from scratch for several reasons:

- It is way faster to stick together a bunch of LEGO bricks that create the doll-house from scratch out of wood or something similar.
  (reusing existing code saves time)
- LEGO bricks are well proven over time and all early issues like wear-out are resolved.
  (after some time, library elements will be bugfree)
- During the time LEGO exists, most commonly used features were implemented as bricks (e.g. wheels, steering wheels, minifigures). So most of your needs are covered.
  (libraries will cover most of your non-application specific needs)
- Everybody knows how to use LEGO-bricks. Your child will even be able to add new features to the doll-house because it knows LEGO and the main concepts of using LEGO stay the same, independently of the context.
  (code and interfaces from different users will look similar)

Of course using LEGO bricks also has its drawbacks:

- Not each and every special feature may be available, so you are a bit limited by the bricks available.

Fortunately in the world of firmware and software development, you are free to easily your own «bricks» if something is not available off-the-shelf.

PAUL SCHERRER INSTITUT PSI

Why using Libraries?

Goals behind the Libraries

- Not covering ***everything*** in libraries …
  - … but covering ***commonly used code***
- Not being bug-free ***initially***…
  - … but fixing reported bugs ***gradually*** and ***only once***
  - This requires thorough verification environment
- Not ***being*** complete …
  - … but ***becoming*** complete
  - Adding elements to libraries when implemented for projects
- Not ***replacing*** the developer …
  - … but allowing the developer to ***focus on the application***

Goals of the Libraries

The intention behind the libraries discussed is not to replace handwritten VHDL code in general or to cover each and every detail of FPGA applications. The main goal is to reduce the time spent on developing and debugging the same code again and again in different projects or sections.

The libraries are normal code and not magic, so they may contain bugs but in contrast to project specific code, bugs fixed once are fixed for all projects using the libraries. As a result the Libraries will gradually become bug-free. The same applies to new features: The libraries will not contain all features from the first day on but features will be added when something reusable is required for a project.

In the end the libraries shall allow the developer to save time on simple and repetitive work and spend this time on improving the actual application. Fortunately that is the attractive part of engineering work everybody wants to spend time on ☺

PAUL SCHERRER INSTITUT PSI

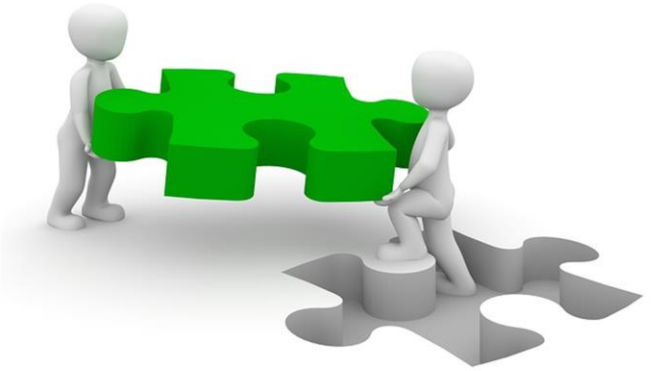
Why using Libraries?

Participation

It's all about giving and taking…

- Using code from the libraries
- Add reusable code to the libraries
- Improve documentation
- Report bugs
- Request features

The libraries discussed are ***open source***
→Invite your friends
→More users = more benefits



Participation

Libraries live from giving and taking. Of course the creators tend to give more than others at the beginning but in the long-run, it is in the interest of every user to give something back so all users can profit from each other.

Of course everybody is free to just use the libraries without supplying anything back. That *everybody* in cludes all FPGA developers in the world with internet access since these libraries are open-source and available through the *paulscherrerinstitute* account on GitHub. So you can invite your friends, you are free to share the libraries with collaborators such as other institutes and you can even continue using them if you should once leave PSI.

However, at some point it would be fair to give something back. The most obvious way of doing so is to add your own code to the libraries. But there are other, less obvious alternatives: Just improving points in the documentation that were not clear to you as user will help others. Reporting bugs is immediately in your interest, since you will get bugfixes back. But in the long-run bug reports help keeping the libraries bug-free and hence are an important contribution. The same applies to feature requests. If you ask for a feature, you may get it for free (or some agreement is made with the PSI FPGA development group) but additionally it will be available to all users in the future.

PAUL SCHERRER INSTITUT

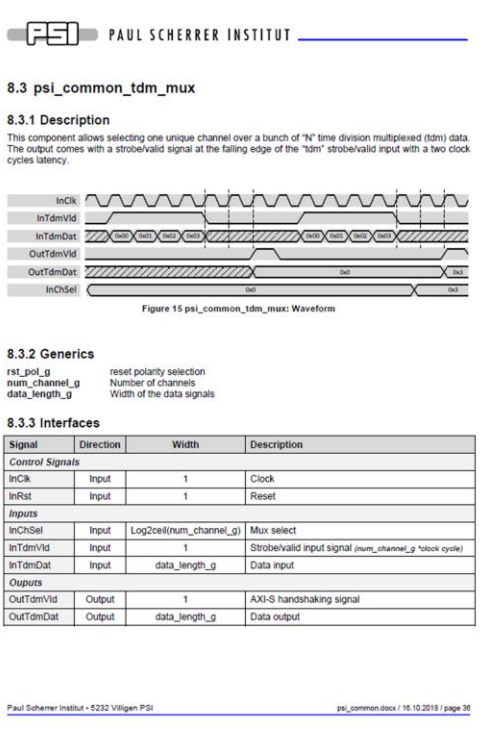
Agenda

- Why using Libraries?
- **Concepts in PSI Libraries**
- psi_common Overview
- Conclusion

Documentation

- Documentation for each entity
- As little as possible but as much as required
- Usually 1-2 pages for simple elements

- Yes, it means «effort» …
- … but do you want to use a library without documentation?



PAUL SCHERRER INSTITUT

8.3 psi_common_tdm_mux

8.3.1 Description

This component allows selecting one unique channel over a bunch of "N" time division multiplexed (tdm) data. The output comes with a strobe/valid signal at the falling edge of the "tdm" strobe/valid input with a two clock cycles latency.

Figure 15 psi_common_tdm_mux: Waveform

8.3.2 Generics

rst_pol_g	reset polarity selection
num_channel_g	Number of channels
data_length_g	Width of the data signals

8.3.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
InClk	Input	1	Clock
InRst	Input	1	Reset
Inputs			
InChSel	Input	Log2ceil(num_channel_g)	Mux select
InTdmVld	Input	1	Strobe/valid input signal *(num_channel_g *clock cycle)*
InTdmDat	Input	data_length_g	Data input
Ouputs			
OutTdmVld	Output	1	AXI-S handshaking signal
OutTdmDat	Output	data_length_g	Data output

Paul Scherrer Institut - 5232 Villigen PSI psi_common.docx / 16.10.2018 / page 36

Page 8

Documentation

Nobody wants to use a library without any documentation. Therefore each GP library element is documented. The documentation is pragmatic: Only the most important points, especially the interfaces and the meaning of all parameters and ports, are described. Usually this leads to one to two pages of documentation for simple library elements. Depending on the library element more information such as the architecture for more complex elements is given.

Of course this means you have to write a little documentation if you add something to the library. But always keep in mind that you expect some documentation for library elements provided by others too. The time required for writing this minimal documentation is negligible compared to the time saved when using the library.

Concepts in PSI Libraries

GIT Setup

- One Library <-> one GIT repo
- No submodules in libraries
  - To avoid nested submodules
  - Relative links used for dependencies
    → Directory structure must match
  - Dependencies clearly documented

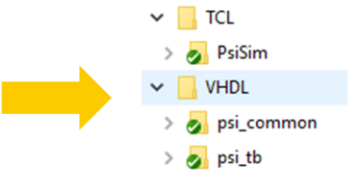
- Link:
  https://github.com/paulscherrerinstitute
  search for the tag «*fpga*»

Dependencies

The required folder structure looks as

Alternatively the repository psi_fpga_a
correct folder structure.

- TCL
  - PsiSim (2.0.0 or higher)
- VHDL
  - psi_common
  - psi_tb (2.0.0 or higher)



GIT Setup

Each library is provided as one separate GIT repository. If libraries depend on each other, links between libraries are made using relative paths (and not using submodules). This allows avoiding the usage of submodules and nested submodules (hard to understand for many users). As a result, the location of libraries, to which a dependency exists, must be known. Therefore the required folder structure is described in each library together with the version requirements.

This approach also ensures that each library is only required once (and not several times in different versions because of dependencies). Since VHDL in general and especially the vendor tools are quite prone to name-clashes, this is an important point.

PAUL SCHERRER INSTITUT PSI

Concepts in PSI Libraries

Keep Clean

- Self-checking test-benches for each entity (mandatory)
- Regression test scripts
  - Modelsim & GHDL
- Automated build server
  - Jenkins
  - Regression tests on pushes to master
  - E-Mail notifications on errors

Keep Clean

Keeping things clean in the long-run requires two main ingredients: Discipline and the correct tools for cleaning. This applies to ovens as well as firmware libraries. If an oven is cleaned regularly with the right tools it will stay clean. Once an oven is left uncleaned for a long time, it will never get clean unless very high cleanup effort is invested.

For the libraries the tools used are described below.

Self-checking test-benches
Each entity has a fully automated test-bench. This allows checking easily if something was broken after applying changes or fixing bugs. It also allows checking behavior easily if there are any assumptions about misbehavior of a library element under some circumstances. These test-benches are the heart of good code quality, so they are mandatory for code that is added to the library. This is where the «discipline» part comes into play: Don't write code with proper test-benches.
Additionally to their usage for testing, the test-benches also serve as last resort if, a user needs more information about the behavior of an entity than available from the documentation. Because the test-benches are fully automated, it is easy to run them and just see what is going on the interfaces.

Regression test scripts
Before releasing a new version of the libraries or after changing a core element used in many places (e.g. a RAM), all test-benches must be ran and checked. To avoid repetitive work, a script that automatically runs all test-benches and checks if errors occurred is required. Such a script is in place and adding test-benches to it is a matter of only a handful of simple TCL lines.

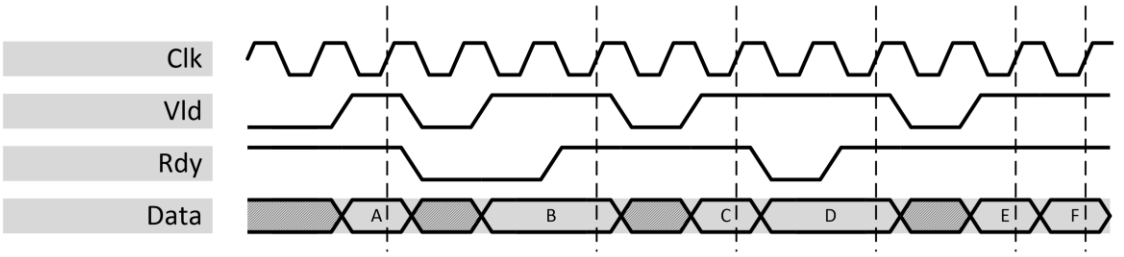
By the way: the regression test scripting framework is a separate library but this will be covered in another presentation on another day.

Automated build server
We all make errors. But we do not all want to suffer from an erroneous commit somebody made to the library. To avoid this, a build server automatically runs the regression test scripts whenever somebody pushes code to the master branch of a library. If errors occur, the maintainer is informed via e-mail so he can take actions.

→A lot of effort was spent to make the libraries safe to use and ensure good code quality!

AXI-S Interfaces

- AXI-S handshaking is used wherever streaming data exchange is required
- Connection of library elements without glue-logic

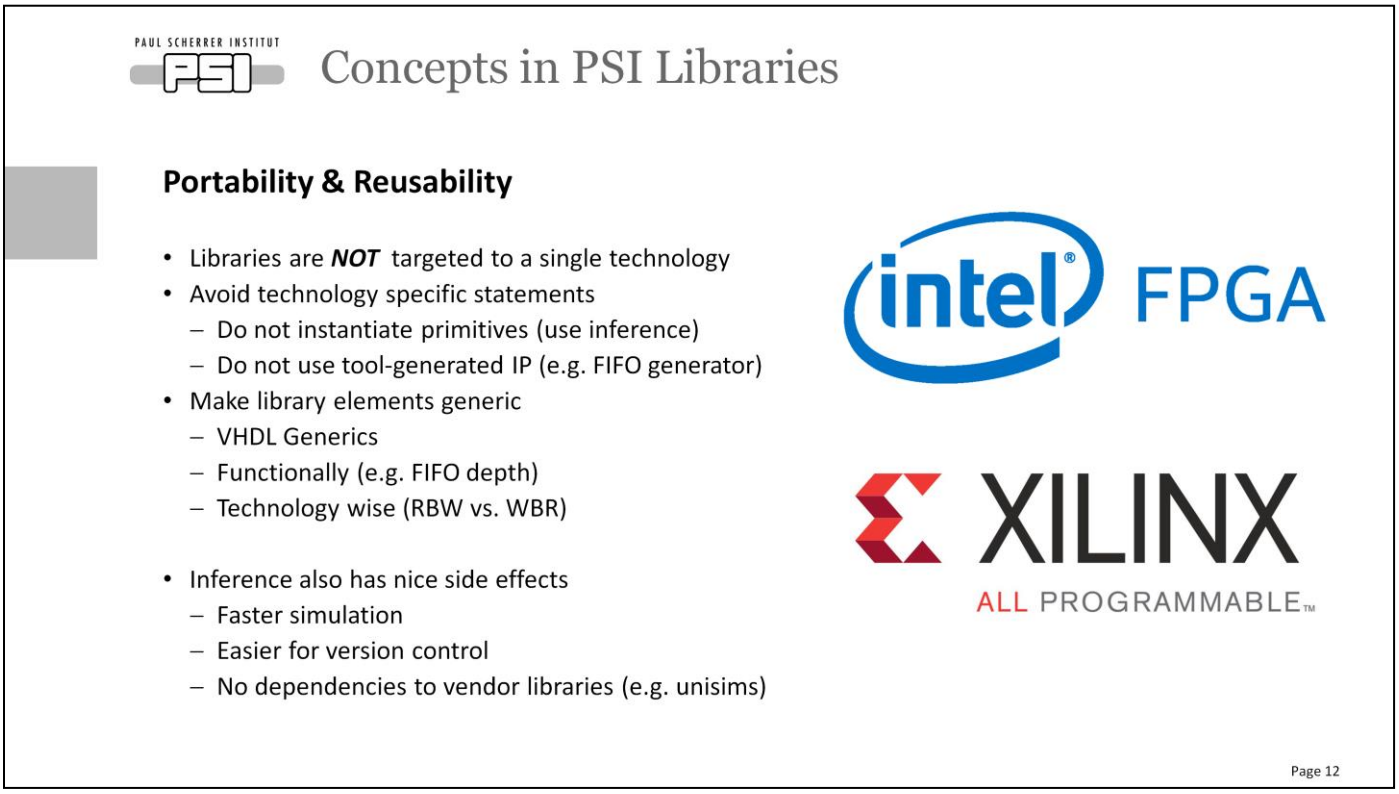


AXI-S Interfaces

Wherever streaming data exchange is required, the library components implement AXI-S handshaking signals. This definitions prevents the requirement for glue-logic to connect different entities from the libraries.

AXI-S has many optional signals. Every library element of course only implements what makes sense in its context.

AXI-S is the most common industry standard for handshaking signals, so it should be known to all FPGA developers. Also is AXI-S well thought to cover all requirements and well defined.



Portability

All the libraries are not targeted to a single technology (and also not to a single vendor). So the usage of any technology specific statements like primitive instantiations or the use of tool-generated IP such as FIFO generator from Xilinx is prohibited. Inference from pure VHDL code shall be used instead. Fortunately the library already contains technology independent and strongly parametrizable implementations of the most commonly used elements such as FIFOs, RAMs and clock-crossings. As a result, these entities can be used for future library elements instead of having to check the exact VHDL syntax that leads to correct inference every time.

Inference also brings some other benefits with it: It usually simulates way faster than the vendor provided primitives and it is simpler for version control, because all functionality is in the VHDL and not in any tool-generated files or libraries provided by the vendors (e.g. unisims from Xilinx). Of course inference can have some drawbacks in certain cases. For example FIFOs are realized with BRAMs and LUTs for the logic instead of using the FIFO logic built-in into BRAMs in some technologies. This is a tradeoff between portability and optimization and given the fact that the number of LUTs for a FIFO is small, it is acceptable in most cases. If ultimate optimization is required at some places, it is still possible to use primitives at these places while using the library elements for the rest of the project.

If you add code do the library, make it parametrizable. For example a «512x16 FIFO» does not really make a good library element. In case of a FIFO, width and depth shall be configurable. The same applies to parameters that are required for efficient inference (e.g. for RAMs the behavior «read-before-write» or «write-before-read»).

Comment BO82:
I personally worked with inference based VHDL code for over 9 years on FPGAs of Altera and Xilinx and I could not see any major problems with RAM or Multiplier inference. So in my eyes this technology is proven and stable.

PAUL SCHERRER INSTITUT

Agenda

- Why using Libraries?
- Concepts in PSI Libraries
- **psi_common Overview**
- Conclusion

PAUL SCHERRER INSTITUT PSI

psi_common

Overview

- Basic logic elements
- Commonly used functionality
- No domain specific things (e.g. DSP things like filters)

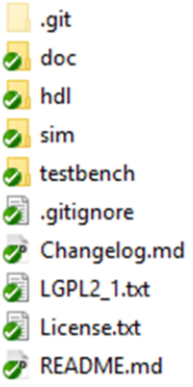
Overview

The *psi_common* library contains basic logic elements (similar to what is available in primitive libraries of the vendors). Additionally commonly used functionality that is not specific to any domain or application is stored in this library.

The library shall explicitly not contain any application specific or vendor specific code. For example a FIR filter clearly goes into the DSP domain, so it shall be stored in such a library (in this case *psi_fix*). If we would start doing advanced motor control in FPGAs and produce many motor control specific library elements, a separate library shall be opened for these entities. The reason behind this approach is that the *psi_common* library is the most basic one and required in almost every project. One should not be forced to checkout hundreds of domain specific entities, just to use a FIFO.

Content

- Packages (array definitions, logic functions, mathematical functions)
- RAMs (true dual port, simple dual port, single port)
- FIFOs (synchronous, asynchronous)
- Clock crossings
- Arbiters (round-robin, priority)
- Pipeline stages with back-pressure handling
- Efficient delay implementation (using SRLs or BRAMs)
- Others
  - SPI master
  - Clock frequency measurement
  - Pulse shaper
  - Strobe generator
  - …



Example «Sync FIFO»: Comparison with Primitive

Primitive

```
fifo_wr_data                <= X"000000000000" & "000000" & mem_addr(11 downto  2);
fifo_1024x10_inst: FIFO36E1
generic map
(
   INIT                     => X"0000000000000000",
   SRVAL                    => X"0000000000000000",
   FIFO_MODE                => "FIFO36",
   DATA_WIDTH               => 18,
   EN_SYN                   => FALSE,
   FIRST_WORD_FALL_THROUGH  => TRUE,
   DO_REG                   => 1,
   SIM_DEVICE               => "7SERIES"
)
port map
(
   RST                      => fifo_rst,
   RSTREG                   => fifo_rst,
   WRCLK                    => s00_axi_aclk,
   WREN                     => fifo_wr,
   DI                       => fifo_wr_data,
   DIP                      => LOW8,
   FULL                     => fifo_full,
   RDCLK                    => s00_axi_aclk,
   RDEN                     => fifo_rd,
   REGCE                    => HIGH,
   EMPTY                    => fifo_empty,
   DO                       => fifo_rd_data,
   INJECTDBITERR            => LOW,
   INJECTSBITERR            => LOW,
);
```

psi_common

```
fifo_1024x10_inst: entity work.psi_common_sync_fifo
   generic map (
      Width_g        => 10,
      Depth_g        => 1024
   )
   port map (
      Clk            => s00_axi_aclk,
      Rst            => fifo_rst,
      InVld          => fifo_wr,
      InData         => mem_addr(11 downto  2),
      OutRdy         => fifo_rd,
      OutData        => fifo_rd_data,
      Full           => fifo_full,
      Empty          => fifo_empty
   );
```


Example «Sync FIFO»: Comparison with Primitive

Primitives are representations of circuits available in the FPGA. They are very low level so they do not abstract away any details. In this example this becomes clearly visible at some points:

- The data signals (DI/DO) are 64-bits wide because that's the maximum supported width of the FIFO. As a result zero padding is required.
- The primitive optimally supports party bits (DIP/DOP). If these bits are used as normal data bits (e.g. for 18-bit FIFOs), data must be split to DI and DIP.
- Some parameters are not related to the «functionality» but to the «implementation». For example FIFO_MODE must be set according to the documentation, so the user has to check what mode is required.

As a result, for readability and compact code, VHDL library elements are usually better.

Another point is the configurability. Primitives are only configurable in the bounds of what the actual resources on the chip support. In the case of the FIFO, the depth and width of the FIFO is limited by the constraints of one single RAMB36 block. Of course multiple primitives can be chained (usually there are specific ports for that) but it must be done manually and changing the depth or width can become cumbersome in such cases. In the case of VHDL library elements, such physical limitations do not exist (inference does connect multiple RAM blocks together automatically).

On the other hand, the primitives of course allow ultimate control. Each and every pin of the actual resource on the chip can be connected, the resource type to use is exactly known and the tools cannot take any wrong decisions. The primitive also does not need any additional LUTs but can use the built-in FIFO logic.

Comment BO82:
In my personal opinion, the more abstract and easier to use library element is preferable as long

as there is no specific reason to go for the primitive. Exactly as in software development, high-level languages are preferred and assembler is only used as last resort for special cases where high-level languages are not sufficient.

PAUL SCHERRER INSTITUT

Agenda

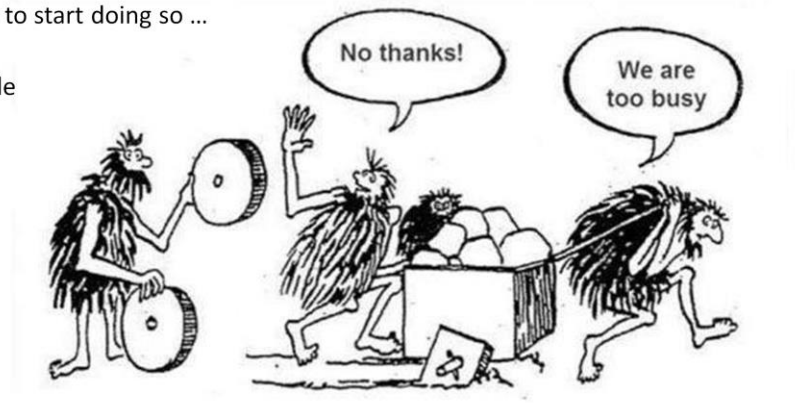
- Why using Libraries?
- Concepts in PSI Libraries
- psi_common Overview
- **Conclusion**

PAUL SCHERRER INSTITUT PSI

Conclusion

Let's share code and work on Libraries together!

- Don't be too busy to start doing so …
- Be curious!
- Support is available



Conclusion

In general the libraries will save time for every user. If you decide to contribute, you may invest some time in making your code parametrizable, document it properly and write clean test-benches for it. However, if everybody decides to spend this time on making code reusable, the code-base will grow quickly and everybody will get more out of the library than he put into it.

Comment BO82:
I worked with such libraries for 8 years at Enclustra and in my eyes they were a key benefit over competitors. So the concept of sharing code in well maintained libraries is not new but proven in reality.
Let me know if you want to start working with the library, I will happily give you and introduction and help you integrating the libraries into your project.