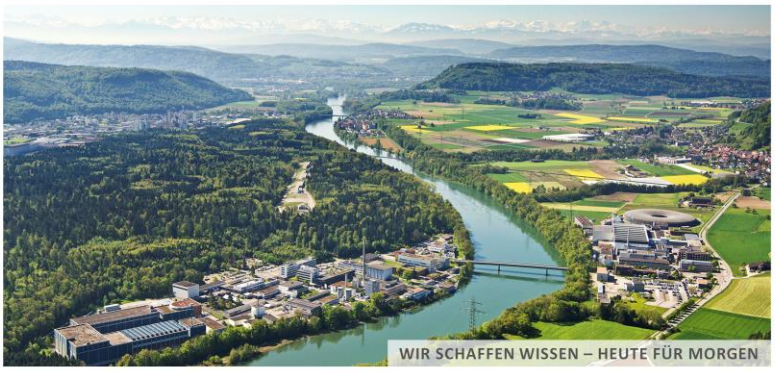
PAUL SCHERRER INSTITUT

PSI



Oliver Bründler :: FPGA Engineer :: Paul Scherrer Institut

psi_fix
FPGA Library

30.11.2018

PAUL SCHERRER INSTITUT

Agenda

- Why using Libraries?
- Concepts in PSI Libraries
- Binary Fixed-Point Numbers
- Bit-True Models
- psi_fix
- Conclusion

PAUL SCHERRER INSTITUT

Agenda

- **Why using Libraries?**
- Concepts in PSI Libraries
- Binary Fixed-Point Numbers
- Bit-True Models
- psi_fix
- 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

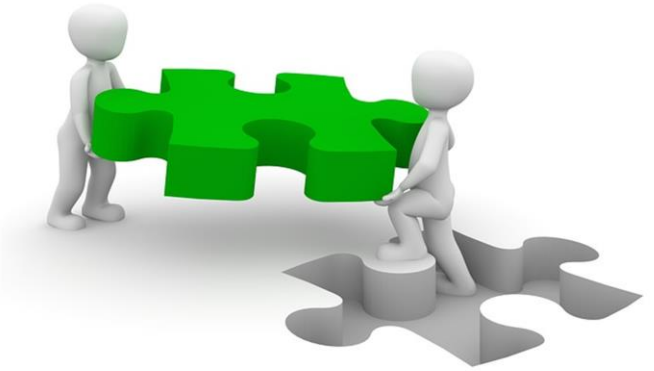
Why using Libraries?

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

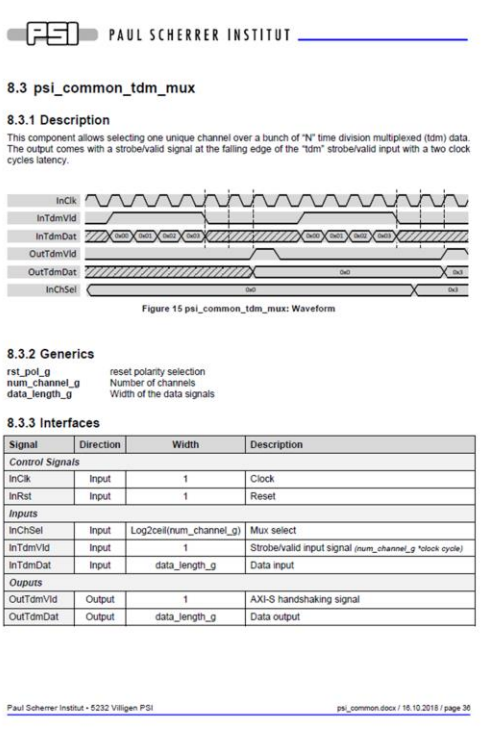
Agenda

- Why using Libraries?
- **Concepts in PSI Libraries**
- Binary Fixed-Point Numbers
- Bit-True Models
- psi_fix
- 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

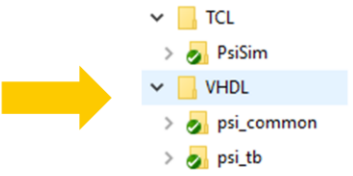
- Link:
  https://github.com/paulscherrerinstitute
  search for the tag «*fpga*»

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.

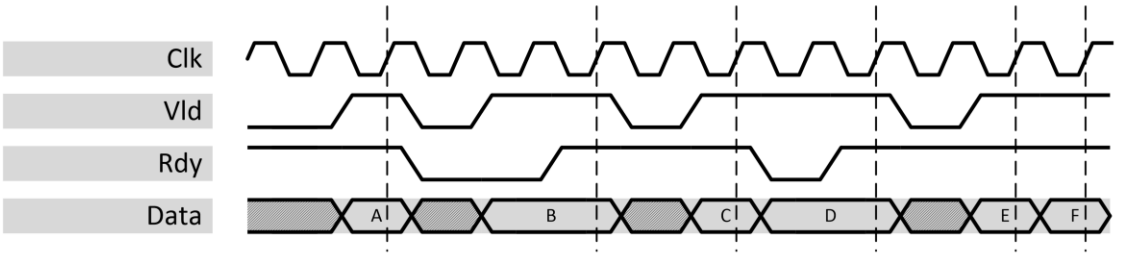
By the way: the regression test scripting framework is a separate library but this will be covered in another presentation on another day.

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

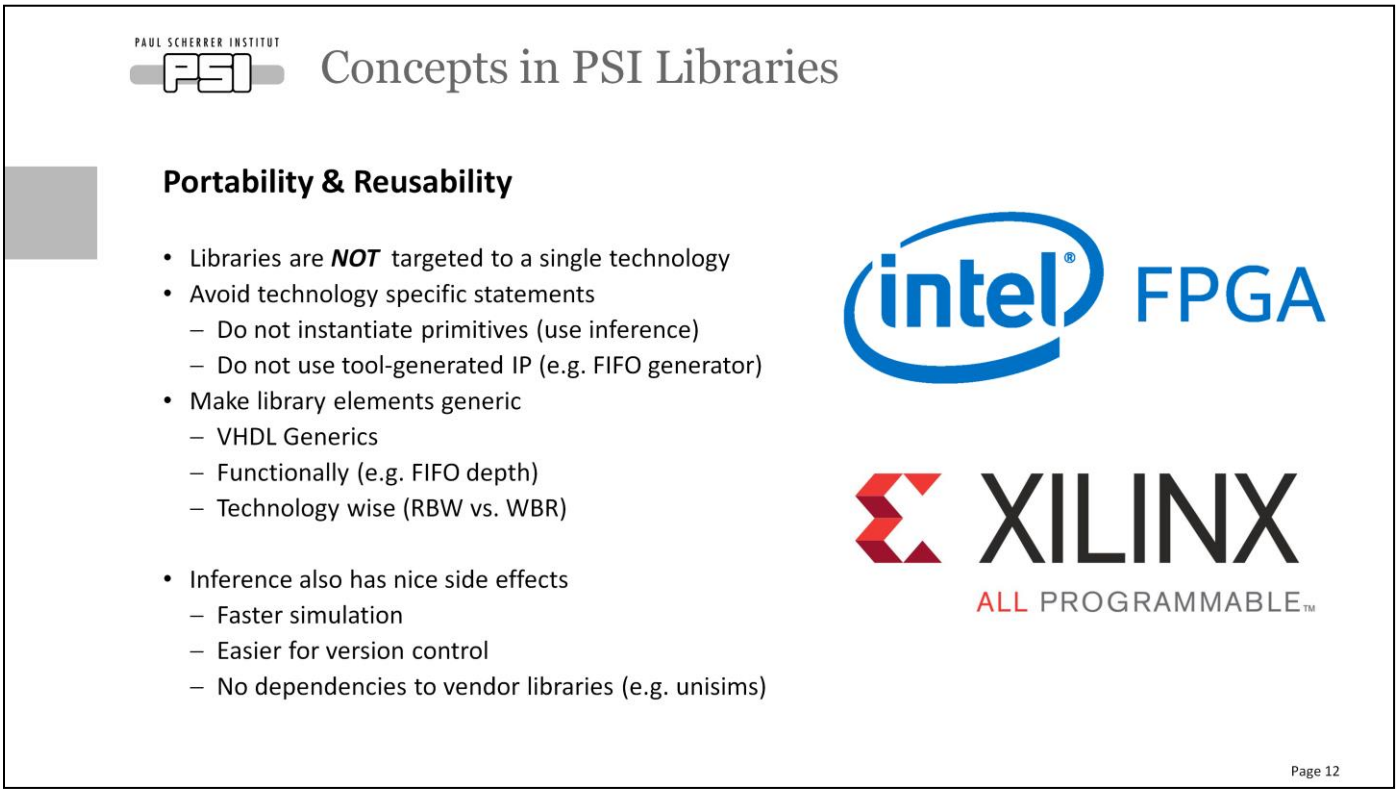


AXI-S Interfaces

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
- **Binary Fixed-Point Numbers**
- Bit-True Models
- psi_fix
- Conclusion

Binary Fixed-Point Numbers

The Binary Fixed-Point System

- How does the decimal system work?
  - Factor of 10 between each digit
  - 10 Values per digit (0 … 9)
  - Decimal point between two digits to mark 10^0 → 12.25
  - Range: $10^{<\text{Digits to the left of the decimal point}>}$
  - Resolution: $10^{-<\text{Digits to the right of the decimal point}>}$
  - Digit values: … - 100 - 10 - 1 - . - 1/10 - 1/100 - …
- How does the binary system work?
  - Factor of 2 between each digit
  - 2 Values per digit (0 or 1)
  - Binary point between two digits to mark 2^0 → 1100.01
  - Range: $2^{<\text{Bits to the left of the binary point}>}$
  - Resolution: $2^{-<\text{Bits to the right of the binary point}>}$
  - Bit values: … - 4 - 2 - 1 - . - 1/2 - 1/4 – 1/8 …

The Binary Fixed-Point System

Binary fixed-point numbers work exactly the same way as the well known decimal numbers, just with a factor 2 wherever we are used to a factor 10. Exactly as the decimal numbers, they allow representing fractional numbers.

PAUL SCHERRER INSTITUT PSI

Binary Fixed-Point Numbers

Why using Binary Fixed-Point Numbers?

- Because it is the natural behavior of digital numbers
  - More efficient, fast, easy to implement
- Example: Distance between 0 and 5 m
  - Could be given as fixed-point number (0, 3, 10)
    - Range 0 … ~8 m
    - Resolution 0.976 mm (1/1024 m)
  - Alternatively as «integer multiple of mm»
    - Range 0 … ~8 m
    - Resolution 1 mm
  - Multiplication between two such distances
    - Fixed-point: (A x B) >> 10 → Shift is for free in logic
    - Alternative: (A x B)/1000 → Division is expensive

Why using Binary Fixed-Point Numbers?

Binary fixed-point numbers represent the natural behavior of a digital system using 2's complement numbers. In contrast to forcing a digital system to somehow work in the decimal system by adding scaling at many places, all scaling can be done by some shifts in the binary fixed-point system. Shifts come for free in FPGA implementation, so there is a significant advantage over using the decimal system for FPGA implementations.

Binary Fixed-Point Numbers

What about Float?

- Benefits
  - No hassle with number formats
  - High dynamic range
- Drawbacks
  - Less resource efficient (but acceptable today)
  - More latency
  - Not natural to VHDL – More code
  - Rounding errors
    - Limited performance (e.g. FIR filters)
    - Some concepts fail (e.g. CIC, moving average)

10'000'000 + 1 = ?

What about Float?

With today's large FPGAs, couldn't all problems be solved by just using floating point operations?

Well, certainly not all problems. But certainly there is justification for floating point numbers in some cases. If numbers with very high dynamic range needs to be passed into or out of an FPGA design efficiently, floating point numbers are the way to go. Of course they also remove the requirement to think about number formats in detail but this is dangerous. Not thinking about number formats often also means not thinking about rounding effects.

Drawbacks of floating point implementations are more resource usage and more latency. Both can be acceptable in many cases. But floating point leads also to more complex code since a floating-point addition usually means instantiating a (vendor-specific) IP core. For sure a floating point addition is not done on a single VHDL line as it is the case for fixed-point numbers.

However, far more important than the inconveniences above is the performance drawback of single-precision floating-point numbers. Due to rounding errors, the performance of many signal processing elements such as FIR filters is degraded. Some concepts that rely on exact calculations do not work at all in floating-point. For example CIC filters and moving averages start drifting away if rounding errors occur. Double precision floating point numbers would ease some of these effects but not all of them and at the same time it increases latency and resource usage.

So floating-point can be used but it is not the answer to all problems.

PAUL SCHERRER INSTITUT

Agenda

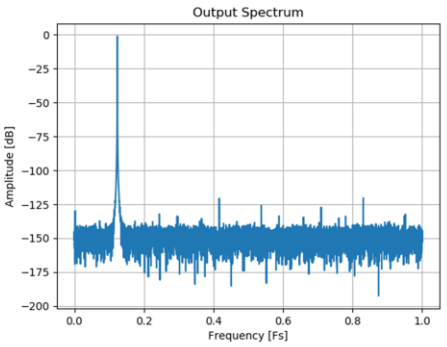
- Why using Libraries?
- Concepts in PSI Libraries
- Binary Fixed-Point Numbers
- **Bit-True Models**
- psi_fix
- Conclusion

PAUL SCHERRER INSTITUT PSI

Bit-True Models

What is a Bit-True Model?

- A model of an algorithm or component (e.g. a DDS)
  - Usually in a more productive language
  - Usually in a environment with powerful tools for performance analysis (MATLAB, Python, …)
- Produces ***exactly*** the same results
  - REALLY exactly!
- Can be used to evaluate performance of an algorithm
- Only numeric behavior is modeled (timing is not)



What is a Bit-True Model

A bit-true model does the exactly same computation than the actual implementation but in a more convenient language/environment. Therefore a bit-true model is usually created in a fraction of the time required for the actual. To see all effects including rounding-, wrapping- and saturation-effects, the model implements 100% the same behavior - Like the term «bit-true» says: down to the last bit.

For FPGA firmware usually languages like Python (incl. NumPy and SciPy) and MATLAB are used. Models written in these languages execute way faster than VHDL simulations, so larger datasets can be simulated. These languages also allow generating stimuli signals easily and doing advanced analysis on outputs (e.g. doing spectrum analysis and nice plots). As a result, such a setup is ideally suited to check the performance of an algorithm or parts of it prior to implementation.

For example the noise created by a DDS (direct digital synthesizer) can be analyzed exactly prior to implementation. Such an analysis is shown in the figure on the slide. The analysis can be executed at exactly the frequency generated in the target system, so any unwanted effects such as harmonics would become visible quickly. Since only the DDS is simulated in this case, there are no unwanted effects from the measurement setup or any other circuitry involved.

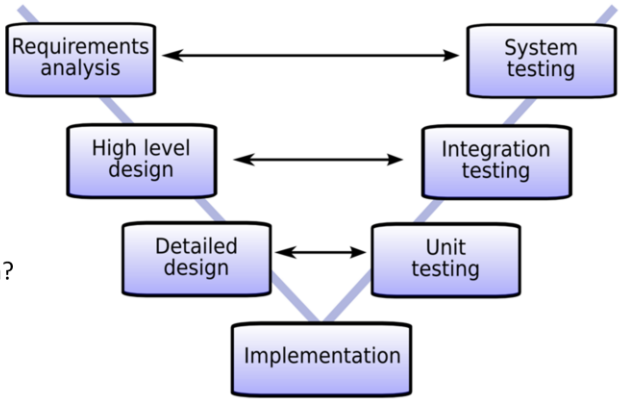
Of course such bit-true models also have limitations: They only cover the numeric behavior but not the timing. Also do they only model the processing but not the interaction with other parts of the system (e.g. configuration of the processing through a register-bank).

PAUL SCHERRER INSTITUT PSI

Bit-True Models

Why using a Bit-True Model?

- When do you know if your algorithm achieves the required performance?
- How do you test individual parts of your DSP chain?



Why using a Bit-True Model?

Let's have a look at the implementation of a DSP project without bit-true models. Usually the architecture of the system is defined on engineering decisions. These are usually based on experience and analytic calculations (e.g. based on ideal frequency responses of different parts of the system). After that, the system is implemented in VHDL and synthesized. Maybe some effort is invested in timing closure before the design can be brought up on hardware. After that, the overall system performance can be measured. If the results are satisfactory, everything is fine.

What if the results are not satisfactory? Is In this case usually many iterations with an instrumented design (including on-chip logic analyzers) are required until the source of the problem is found. Each iteration includes synthesis and possibly timing-closure. Additionally bugfixes may have significant impact on the architecture and implementation, so they tend to be time-consuming.

With a bit-true model, the numeric performance of each part of the algorithm as well as the performance of the full algorithm can be evaluated during architecture and module design. So most iterations happen during that phase without all the implementation hassle. It is also possible to evaluate the exact parameters required (e.g. CORDIC resolution and iterations to achieve a certain precision for a given kind signal). As a result, the quality of the signal processing can be checked before hardware is available. This leads to less debugging on hardware and hence a shorter overall development time.

Another point worth mentioning is testing of single parts of the DSP chain. With a bit-true model, the test is simple: If the same input is applied to the model and the VHDL implementation (in simulation), the outputs must exactly match. Without a bit-true model, it is very hard to see if for example a DDS «works correctly» since analysis of the phase noise in VHDL is not very handy.

PAUL SCHERRER INSTITUT

Agenda

- Why using Libraries?
- Concepts in PSI Libraries
- Binary Fixed-Point Numbers
- Bit-True Models
- **psi_fix**
- Conclusion

PAUL SCHERRER INSTITUT PSI psi_fix

ENCLUSTRA FPGA SOLUTIONS

What is psi_fix?

- **A remedy for fixed-point design related headache!**

- A VHDL Package (based on *en_cl_fix* from Enclustra GmbH*)*
  - Handles format conversions incl. rounding and saturation
  - Functions for all common arithmetic operations (Add, Mult, Resize, …)
- A VHDL Library
  - Contains commonly used entities (FIR, CIC, function approximation, …)
  - Strongly parametrizable (also number formats)
- A Python Library
  - Contains Python bit-true models of all functions in the VHDL package
  - Contains bit-true models of all entities that exist in VHDL
  - Makes the conversion between VHDL and bittrue Python model easy
  - Based on the well known *NumPy* package for fast processing of arrays
  - Interface to MATLAB exists (usable for MATLAB users)

Whati s psi_fix?

The main goals of *psi_fix* are:
- Making implementation of fixed-point designs in VHDL easy
- Produce more readable and maintainable code
- Easy conversion between VHDL implementation and Python bit-true model
- Provide VHDL implementation and bit-true models of commonly used elements

The first three goals are fulfilled by a VHDL package that covers all the fixed-point related shifting and format changes behind function calls. The same functionality with the same syntax is also provided as Python package, so writing the same things in Python and VHDL is easy. Both packages (VHDL and python) are based on the *en_cl_fix* package that is provided as open-source code by Enclustra GmbH.

The last goal is fulfilled by a library of standard components such as FIR filters, CIC decimators and CORDIC implementations based on the packages described above. Each unit comes with VHDL implementation, bit-true model and a test-bench that checks if the behavior of these matches exactly.

All Python code is based on the *NumPy* package to support fast processing of arrays and for MATLAB users, an interface to MATLAB was implemented (so the Python models can be called from MATLAB).

psi_fix Number Formats

- psi_fix Format: (S,I,F)
  - S: Sign bit (yes/no)
  - I: Number of integer bits → Range
  - F: Number of fractional bits → Resolution
- Examples:

Number Format	Range	Bit Pattern	Example Int	Example Bits
[1,2,1]	-4 ... +3.5	sii.f	-2.5	101.1
[1,2,2]	-4 ... +3.75	sii.ff	-2.5	101.10
[0,4,0]	0 ... 15	iiii.	5	0101.
[0,4,2]	0 ... 15.75	iiii.ff	5.25	0101.01
[1,4,-2]	-16 ... 12	sii--.	-8	110--.
[1,-2,4]	-0.25 ... +0.1875	s.--ff	0.125	0.--10

Unused bits may be omitted

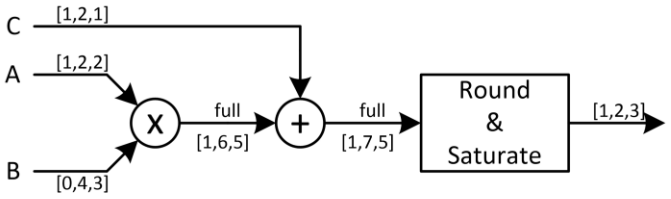
psi_fix Number Formats

In *psi_fix* number formats are denoted as a tuple of three numbers. The first one (0 or 1) says if the number is signed. The second one denotes the number of integer bits (bits to the left of the binary point) that defines the range of representable values. The third number gives the the number of fractional bits (bits to the right of the binary point) that defines the resolution of representable values.

Some examples are given on the slide.

Why a VHDL Package is required

- **Fixed-point VHDL code tends to be …**
  - **… unnecessarily long**
  - **… unreadable (hidden dependencies)**
  - **… hard to maintain (changing many magic numbers)**
  - **… not reusable (number formats are fixed)**
- The presence of fixed-point libraries for C, MATLAB, C++, … shows that this problem really exists and is not limited to VHDL



Why a VHDL Package is required

Implementing fixed-point processing is possible in pure VHDL without any package, so why do we need one?

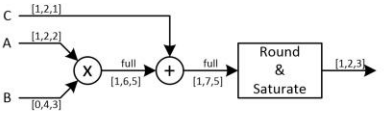
The point is that fixed-point code written in pure VHDL tends to be very unreadable and unnecessarily long. Additionally many things usually get hard-coded so the code is not re-usable. These drawbacks lead to hardly maintainable code. This will be demonstrated based on a simple example on the next slides.

The example consists of a simple multiplication, an addition and rounding and saturation at the output.

PAUL SCHERRER INSTITUT

psi_fix

Traditional VHDL Implementation




```
signal a : std_logic_vector(4 downto 0);          -- Format: (1,2,2)
signal b : std_logic_vector(6 downto 0);          -- Format: (0,4,3)
signal c : std_logic_vector(3 downto 0);          -- Format: (1,2,1)

signal m : std_logic_vector(12 downto 0);         -- Format: (1,6,5) --> magic
signal sum : std_logic_vector(12 downto 0);       -- Format: (1,7,5) --> magic
signal rnd_add : std_logic_vector(12 downto 0);  -- Format: (1,7,5) --> magic
signal rnd : std_logic_vector(10 downto 0);       -- Format: (1,7,3) --> magic
signal sat : std_logic_vector(5 downto 0);        -- Format: (1,2,3)

...

m <= std_logic_vector(signed(a)*signed('0' & b));
sum <= std_logic_vector(signed(m) + signed(c & "0000"));    -- magic number "0000"
rnd_add <= std_logic_vector(signed(sum) + 4);               -- magic number 4
rnd <= rnd_add(12 downto 2);                                -- magic range (12 downto 2)
if signed(rnd) > 31 then                                    -- magic number 31
   sat <= "011111";                                         -- magic number "011111"
elsif signed(rnd) < -32 then                                -- magic number -32
   sat <= "100000";                                         -- magic number "100000"
else
   sat <= rnd(5 downto 0);                                  -- magic range (5 downto 0)
end if;
```

Resource	Utilization
LUT	56

Traditional VHDL Implementation

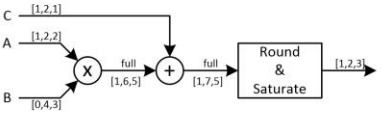
The traditional VHDL implementation works but is not really maintainable. It contains many magic numbers for ranges of signals, shifts and comparisons. If one just looks at this code, it is quite hard to recognize what is implemented here.

Another important point is that quite a lot of code-lines are required to describe a relatively simple circuit.

PAUL SCHERRER INSTITUT

psi_fix

psi_fix based VHDL Implementation




```
constant a_fmt : PsiFixFmt_t := (1,2,2);
constant b_fmt : PsiFixFmt_t := (0,4,3);
constant c_fmt : PsiFixFmt_t := (1,2,1);
constant s_fmt : PsiFixFmt_t := (1,2,3);

-- Automatic scaling of internal formats
constant m_fmt : PsiFixFmt_t := (max(a_fmt.S, b_fmt.S), a_fmt.I+b_fmt.I, a_fmt.F+b_fmt.F);

-- Automatic scaling of word-widths
signal a : std_logic_vector(PsiFixSize(a_fmt)-1 downto 0);
signal b : std_logic_vector(PsiFixSize(b_fmt)-1 downto 0);
signal c : std_logic_vector(PsiFixSize(c_fmt)-1 downto 0);
signal m : std_logic_vector(PsiFixSize(m_fmt)-1 downto 0);
signal s : std_logic_vector(PsiFixSize(s_fmt)-1 downto 0);

...

-- Short and readable functional code
m <= PsiFixMult(a, a_fmt, b, b_fmt, m_fmt);
s <= PsiFixAdd(m, m_fmt, c, c_fmt, s_fmt, PsiFixRound, PsiFixSat);
```

Resource	Utilization
LUT	52

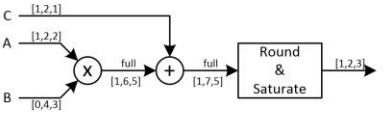
psi_fix based VHDL Implementation

The psi_fix based VHDL implementation is much more readable. The actual functionality is described on two clearly readable lines and there are no magic numbers. Number format are clearly readable and even scale automatically (*m_fmt* scales with the input formats). So the code is not only readable but also easily maintainable, even if number formats change.

Automatic scaling of number formats is a key requirement for creating configurable library elements.

This looks good in code, but what about synthesis? Both code-pieces synthesize to exactly the same logic. *en_cl_fix* as well as *psi_fix* are written with synthesis in mind. Formats are handled as constants that have no direct connection to the datapath, so no additional logic is created. Because number formats are constant, all unused if/else clauses (like unused rounding) can be fully optimized out. This is proven over years of usage for *en_cl_fix* and *psi_fix* also exists for more than a year now without showing any suspicious results.

psi_fix based Python Model

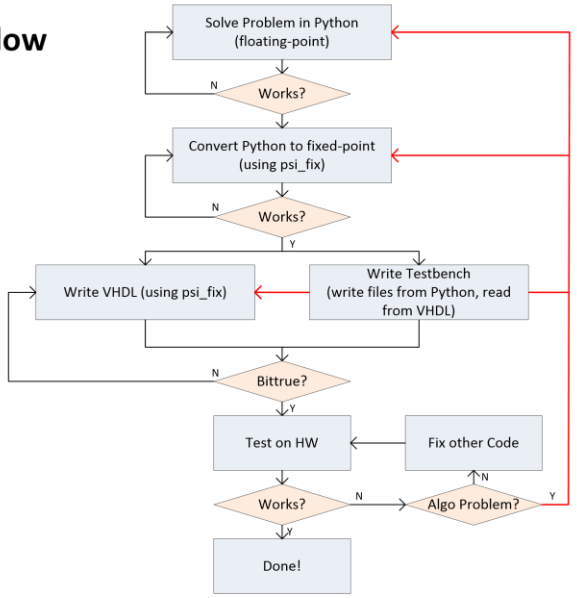



```
from psi_fix_pkg import *
import numpy as np

a_fmt = PsiFixFmt(1,2,2)
b_fmt = PsiFixFmt(0,4,3)
c_fmt = PsiFixFmt(1,2,1)
s_fmt = PsiFixFmt(1,2,3)

#Automatic scaling of internal formats
m_fmt = PsiFixFmt(max(a_fmt.S, b_fmt.S), a_fmt.I+b_fmt.I, a_fmt.F+b_fmt.F)


def Model(a : np.ndarray, b : np.ndarray, c : np.ndarray) -> np.ndarray:
    m <= PsiFixMult(a, a_fmt, b, b_fmt, m_fmt);
    s <= PsiFixAdd(m, m_fmt, c, c_fmt, s_fmt, PsiFixRound, PsiFixSat);
    return s
```


psi_fix based Python Model

The exactly same functions as for VHDL are provided in Python. As a result, the bit-true Python code looks very similar to the VHDL code. Independently of which one was first, converting the VHDL code into Python or vice versa is straight-forward.

Development Flow



Development Flow

When implementing fixed-point processing with *psi_fix*, it is usually easiest to develop the general concept of the algorithm in pure Python. At this stage floating-point and all Python libraries can be used (e.g. SciPy). The only goal is defining the algorithm in general.

The floating point algorithm is then converted to fixed point (using the *psi_fix* Python package). After the conversion, the performance of the algorithm is tested again to see if rounding, saturation and other fixed-point related behavior is not affecting performance.

If performance is fine, the fixed-point model is converted to VHDL. In parallel, a test-bench is created. The test-bench consists of a Python script that creates stimuli and injects them into the bit-true model and a VHDL test-bench that compares the output of the Python model to the VHDL implementation when injecting the same stimuli. To do so, the Python script writes stimuli and response to files that can be read by the VHDL test-bench. As a result the VHDL test-bench becomes trivial.

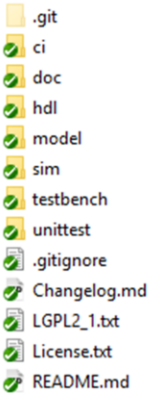
The test-procedure is always the same, so it is very easy to copy all code from an existing test-bench and slightly modify it according to the requirements of the new entity.

Once the VHDL code is bit-true, it can be implemented on the FPGA (including timing closure) and tested on hardware. Anything else than correct behavior according to the simulations would be a big surprise at this point.

PAUL SCHERRER INSTITUT PSI psi_fix

Content

- CIC (decim., interpol., single channel, multi-channel)
- CORDIC implementations (rotating, vectoring)
- DDS
- Modulator / Demodulator
- FIR filters
- Linear function approximation (code generator)
- Moving average
- Binary divider
- Complex number operations (absolute, multiplication, add/sub)
- Code generators for LUTs and constant-packages
- … more to come



Content

The functionality of most elements in *psi_fix* is pretty self-explaining. However, some of them are not that obvious, so they get described below.

For all types of filters (CIC/FIR), different implementations are available. Single-channel, multi-channel with parallel or TDM (time-division-multiplexed) handling of channels, etc. A naming scheme is defined, in order to have unified entity names for different implementation flavors of the same filter logic.

The linear function approximation is implemented as code generator in python. So a function can be described in Python and a piecewise-linear approximation is generated automatically. This allows easily approximating function such as sqrt(). For standard functions such as sin or sqrt, the approximations are checked in into the library but more project-specific approximations can be generated easily.

The code generators for LUTs and VHDL packages containing constants allow easily passing parameters from Python simulations to VHDL without manually updating them in some VHDL files which is error prone.

PAUL SCHERRER INSTITUT

Agenda

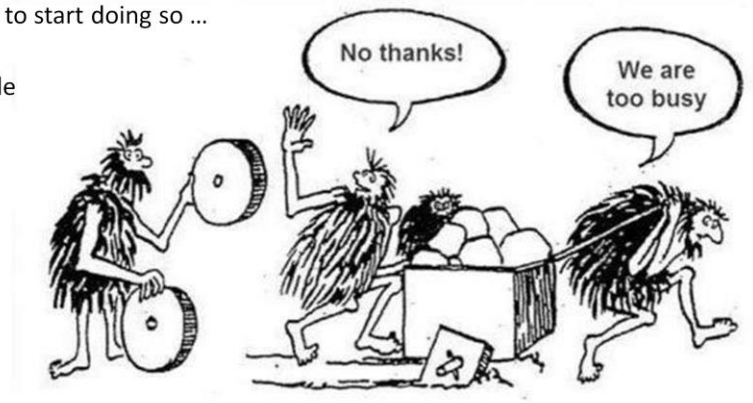
- Why using Libraries?
- Concepts in PSI Libraries
- Binary Fixed-Point Numbers
- Bit-True Models
- psi_fix
- **Conclusion**

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.