Java2VHDL

Dr. Hartmut Schorrig
www.vishia.org

2023-04-12

Table of Contents

1 Manual how to read, examples and support

To study concepts and approaches start with the following chapter 2 Motivation and the following 3. Java2VHDL - approaches

To use as user's guide see chapter 4. Java2VHDL - User's guide

Last not least an example 'BlinkingLed' to build your own first example is given start with chapter 5. The example Blinking LED, view to Java sources in respect to the FPGA description, and following.

Example code and link

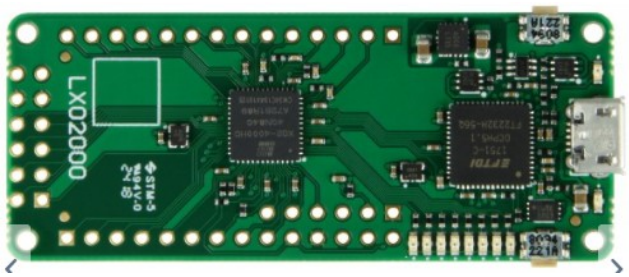
You find a home page of Java2Vhdl on

https://vishia.org/Fpga

Also there, you find a zip file,

currently https://vishia.org/Fpga/deploy/J2Vhdl_Workbox-2023-04-12.zip

and some more in https://vishia.org/Fpga/deploy



This zip file contains the infra structure to work with own projects. It contains also a "`exmpl_vishiaJ2Vhdl_BlinkingLed`" which is a real working example using a Lattice FPGA test board from https://shop.trenz-electronic.de/de/TEL0001-02-LXO2000-mit-Lattice-XO2-4000-On-Board-USB/JTAG-2-5-x-6-15-cm. This example is elaborately explained in chapter 5. The example Blinking LED, view to Java sources in respect to the FPGA description from page 92 and 6. The example Blinking LED, view to Java sources in respect to test on Java level from page 106.

This zip file contains all sources which are used here to explain how to do in chapter 4. Java2VHDL - User's guide from page 44

located in the zip file in `src\vishiaFpga\java\org\vishia\fpga\tmpl_J2Vhdl`.

The translator can be started calling `src\vishiaFpga\makeScripts\genTmpl_J2Vhdl.bat` ,

the result to compare is contained in `src\vishiaFpga\makeScripts\genTmpl_J2Vhdl.bat`.

The translator itself needs a Java JRE, tested on Oracle's Java-8 and also with OpenJDK.The jar files for the translator itself are hosted on

https://vishia.org/Java/deploy

To get the files follow 4.3. Tools necessary for Java to Vhdl translation and test support page 48, supported by the zip files.

2. Motivation

VHDL was originally created in the 1980th for timing simulations of the behavior of ASICs See https://en.wikipedia.org/wiki/VHDL: *In 1983, VHDL was originally developed at the behest of the U.S. Department of Defense in order to document the behavior of the ASICs that supplier companies were including in equipment. …* (requested on 2022-05-25)

As a second approach it was later used to determine the content of FPGAs (input for routing).

In the earlier time of FPGA development, till mid of 1990th, usual the timing behavior was evaluated after the routing process, either by timing simulation, or by examination all delays of paths. With some manually settings (placing, special lines manually routed) the routing is repeated till all is met. For that also the VHDL was proper.

From approximately mid of 1990th usage of "*timing constraints*" become more importance. With timing constraints the router can decide by itself for using routing resources. It means constrains + logical description is sufficient to get a proper FPGA content. The importance of timing simulation is no longer given, at least for most of the FPGA content. Only for clock synchronization approaches or asynchronous parts of logic it is necessary.

The FPGA tool suites support the formulation of timing constraints usual in a special kind. VHDL has not an meaningful contribution for that.

Also the simulation of hardware designs needs often special tools. Test approaches with comprehensive functional tests are a special topic.

For all that reasons another way to formulate hardware design content inclusively timing constraints was searched.

This documentation offers the idea to formulate hardware design with timing constraints and elaborately possible functional tests, in Java language. The design is oriented to a *one clock system* as synchronous state machines. The source code in Java which describes that and which is used for the functional tests is then translated to VHDL for the routing process. It means, the full FPGA content can be developed with functional simulation outside of FPGA tool suites. From the FPGA tools so much as necessary can be used, for example only the routine process with timing report (to check whether all is met), or for example also a timing simulation for specific signals outside the functional logic (synchronization with more as one clock or such one). That is all possible of course, but firstly the synchronous part of the design can be planned and tested completely only using a Java development environment (such as Eclipse). This supports also elaborately usage of test possibilities (input signal preparation, evaluation, test control etc.).

The better structure of modularity in Java (with ObjectOriented approach, referencing, interface technology with simple MOC replacement on test) was not in focus on the motivation phase. It was the result of developing following Java approaches.

As translation from Java to VHDL the ZBNF parser from the author proven since 2008 was used, also proven here. (https://vishia.org/ZBNF/index.html). The Java sources are read in, broken down into syntactic parts and then the VHDL is generated, whereby all cross-connections, e.g. from the interface call to the implementation instructions, are also evaluated. Using this translation approach was planned from beginning. A detail: An expression (conditional, assignment etc.) is dispersed in its parts and presented in the Revers Polish Notation. That clarifies and solves all precedense rules. From that RPN presentation the VHDL expression is generated, which has partially other precedense rules. The experience to do this had been available from other translation projects for many years.

3. Java2VHDL - approaches

This chapter describes which approaches are present by the Java2VHDL system before there are concrete application instructions. For the concrete application instruction see next main chapter4. Java2VHDL - User's guide from page 44 or the particular description of 5. The example Blinking LED, view to Java sources in respect to the FPGA description from page 92

3.1. Writing hardware logic in Java, principles

Java is one of the most known and used programming languages. In opposite to for example C++ Java is a save language. The commonly small programming mistakes by the users do never cause a non predictable behavior, they are obviously.

There are two different approaches for Fpga content:

a) Pure hardware logic

b) Using FPGA as platform instead a microcontroller: A microcontroller can react in step times down to ~ 10 µs. 50 µs are often usual. For faster especially controlling algorithm with step times of ~ 1 µs the usage of FPGAs becomes more and more familiar, because the FPGA have enough space, are no more so expensive with much space and the tool suites are more and more proper.

The approach b) is not handled here. They are other tools such as https://hdl.github.io/awesome/items/synthesijer/ or programming FPGA in C++ (known since ~ 2005) which fulfill this approaches.

This work is related to pure hardware logic.

Writing hardware logic in Java requires knowledge about hardware in FPGA. On the first hand one should familiar with the internal structures of logic blocks in your FPGA and also with such things as floorplanner, physical view, timing reports. On the other hand you should familiar with VHDL as hardware description language. The VHDL is used as bridge between the Java hardware sources and the place&route tools of the FPGA.

The benefit of writing hardware logic in Java is: You can organize elaborately functional tests. If your logic is clocked and all time constrains are met, the behavior in the hardware is the same as the functional behavior in Java. And the functional behavior is the complex one thing, not the details of behavior of the implemented design in the FPGA also only simulated. It is sufficient to simulate the functionality in Java, check matching all timing constrains, and the test with the real hardware FPGA.

A second benefit is: In Java a better module and sub module organization is possible, including the advantages of Object Oriented assembling of the modules als using the interface concept. The last one may be importance, if you want to vary your solution for different test approaches, or also for different implementations with the same sources. Last not least the amount of different sources for variants are reduced, reuse is more supported as with the module concept of VHDL.

And the last benefit: Java is often proper known from a large spread of persons. The tool support is well. That are detail advantages.

3.1.1. Principle of functional simulation of synchronous state machines

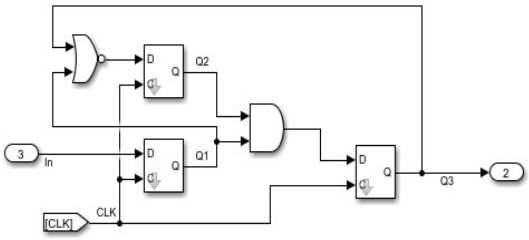
Typically, the bulk of the content of an FPGA should be a synchronous design. It means, all input signals are gathered by only one clock, and then furthermore used only clock-synchronous.

Secondary hint: Using several clocks for a FPGA design is often not the best choice. If you have different time ranges for signals, it is better to use one fast clock and several '*clock enable*' strategies. This is especially supported here.

All new states are based on the given states with logical relations, which are then adopted as current states with the clock edge.

```
Qn+1 = fn( Qn )
```

Whereby the input signals are also part of the `Q` (means all FlipFlop states). The `fn` (function) is a only logical relation.



To do the same in any sequential programming language, firstly all new states should be calculated newly from the current states without using the new states itself for calcualtion. That should be done for the whole functionality (not only module per module), because elsewhere new states from already calculated modules may be used by mistake. This new states are the representation of the signal levels on the D inputs of the FlipFlops. For that, `step()` operations are written in Java for each module and for the whole content.

Then the clock edge comes in hardware, in Java software it is the copy process to declare the new calculated states as current ones. That is done in Java by `update()` operations for each module and for all.

After that, output values can be calculated from the new current states. Alternatively the output values can also be calculated as the D inputs firstly as new output values, and then copied to the current state of outputs, they are clocked outputs, also done by `update()`.

Java: Principle for step and update for some modules

```
mdl1.step(time);
mdl2.step(time);
mdl1.update();
mdl2.update();
```

Step and update operations looks like:

Java: One simple step operation in a module

```
public void step ( int time ) {
  this.record_d = new Record(time, this.record, inputs);
}
```

It means it creates a new instance and calls the constructor using the previos states and inputs. The new instance is referenced first in the variable which presents the 'D inputs of Flipflops'.

Java: One simple update operation in a module

```
public void update ( ) {
  this.record = this.record_d;
}
```

Now all states are available for the next `step()` and for outputs.

As you see in mnemonic, a module contains some `RECORD` in VHDL. The constructor is translated to a VHDL-`PROCESS`.

Inside the inner class for a 'RECORD' it is recommended to declare all variables as´`final`. So it cannot be forgotten to set all, and it is clarified that all RECORD variables are set unique. The

template for a constructor looks like:

Java: One simple constructor for a VHDL-PROCESS

```
public void RecordType ( int time, RecordType z, RefType inputs ) {
  if(inputs.module.ce) {
    this.varA = inputs.input && inputs.otherModule.state;
  } else {
    this.varA = z.varA;
  }
}
```

The `RecordType` contains here only a variable `final boolean varA;`. It is set as combinatoric if a '*clock Enable*' comes from another module (for this example) and it is hold (not changed) if … ce is false. The …state from the other module is of course the state of the last step after update. The z represents the own state of the last step for the whole RECORD of that PROCESS.

3.2 Timing relations

The time to build the combinatoric for the D-inputs of the flipflops in the FPGA should be lesser than the time between the clock edges. Then the logical functionality is also the real functionality.

If delays are longer, the behavior is undefined. That is not admissible. It is controlled by the timing constraints.

But not all paths should consider the minimal time between the clock edges. For example you can have a fast clock signal of 200 MHz (5 ns). Some paths should switch in this speed. But not all paths can met the constraint of 5 ns. If the logic is complex, it is a too much requirement. Usage of different clocks for different logic parts is really not a good idea, it requires additional effort for synchronization between the systems. There is a better solution.

Usual the FlipFlops in the Logic Block of a FPGA have a CE input. It is Clock Enable. If this input is 0, then the signal on the D input has no effect. The FF does not change its state. It means, you can free a FF for switching with a longer period though the clock frequency is high. Only the CE signal should be provided in the required speed for the clock period. The other signals to build the logic need only the CE period respectively the time between two CE='1'.

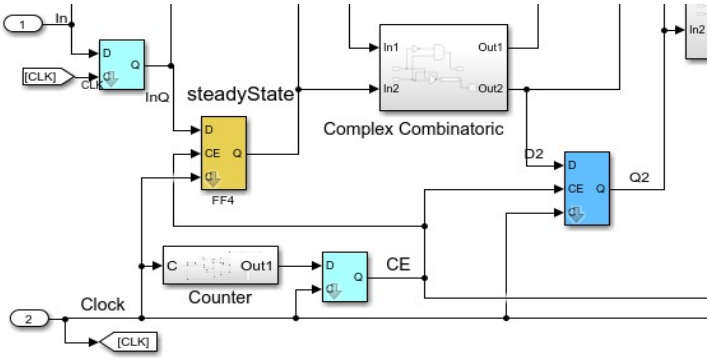


Figure 1: CE on Flipflops

You see the clock in the last track, each rising edge changes the FF state. But the CE signal controls that only each 5th clock edge force switch. In this example the CE is built by a counter which outputs CE exactly after each 5th clock edge. But you see also that CE is delayed. It does not come immediately after the rising clock edge, it does come a little bit later and goes later. The importance is, that CE should be stable on the rising clock, with a minimal necessary setup time. CE can also be delayed, but lesser as one clock period.

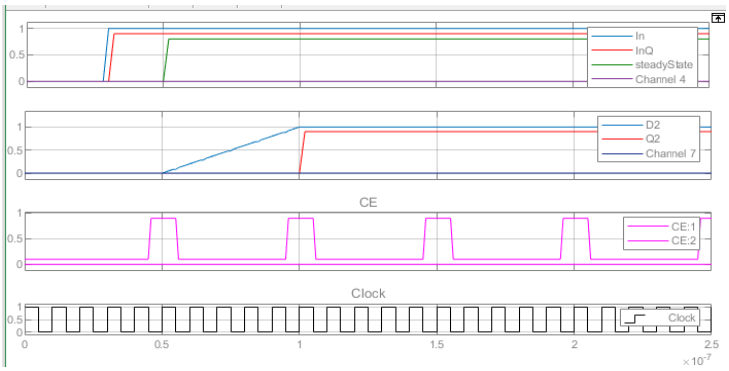


Figure 2: CE in scope

In this example the red signal in the first track switches after any rising clock edge, not guarded with CE. But the green signal switches with clock and CE. The next track shows a long delayed signal. It is presented here as ramp, because for simulation Simulink (© Mathworks) is used (not VHDL and a FPGA tool suite). But before the next CE guarded clock edge, the signal arrives its necessary state, and that is sufficient. So the next CE-enabled clock edge gets the state of the delayed signal without failure and produces the next dark green output. This output can also in turn have a delay to its next output, feeding the D input of a FF etc. pp.

In VHDL such behavior can be written in the following form:

```
PROCESS(CLK, CE) BEGIN
  IF rising_edge(CLK) THEN
    IF CE='1' THEN
      Q2 <= Q1 AND someOtherLogic;
    END IF;
```

```
  END IF;
END PROCESS
```

In VHDL it is determined that nothing should be done if CE is not ='1'.

As ideal case CE is an output of a FlipFlop. This output is immediately routed to the CE input of all using FF. For routing a stable clock net with high fan out is used.

But the router decides by itself how the CE input of the FF in the FPGA are used. See the next slightly changed example:

```
PROCESS(CLK, CE) BEGIN
  IF rising_edge(CLK) THEN
    IF CE='1' THEN
      IF(steadyState='1') THEN
        Q2 <= Q1 AND someOtherLogic;
      END IF;
    END IF;
  END IF;
END PROCESS
```

There is another signal as IF condition, here named as steadyState. Then the routing process can assemble this signal also to build the CE for the FF inputs. For example if you have a register of 16 or more FF instead the single Q2 and Q1, it saves a lot of routing resources to do so.

Right side see which may be produced by the router, drawn with ordinary gatter logic. In the real FPGA LUTs (Look Up Tables) are used instead gatter, of course.

How this is related to writing logic in Java ?

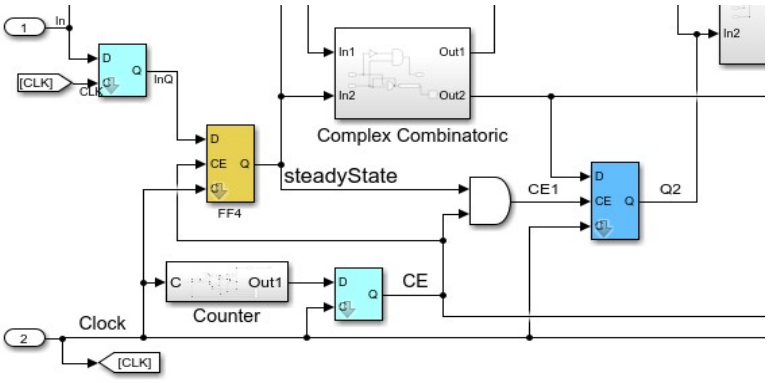


Figure 3: CE combinatoric on FlipFlops

The goal is, determine timeing constrains (see next chapter) for some paths. In Java you can measure the number of clocks between changing signals, respectively the number of clocks how long is a given signal in the steady state. It is very simple. If the signal (variable) is written newly, similar the current time is also stored. The time is a simple incremented int value (32 bit, sufficient for 1 billion clock steps). Also a long value may be usable, but seems to be not necessary.

If the signal is used for combinatoric, the current time is compared with the stored time of the input signals. Yet an assertion or warning message can be built if this time difference is lesser than expected.

Details how to do are described in chapter 4.12 Checking time between FF groups page 86.

3.2.3. Be careful because of glitches in logic

Because the CE is built fastly which each clock edge, it should have a less delay. All FF which should have such an fast delay can be assembled to a FF group, and for that group a constraint can be written (here for Lattice Diamond FPGA tool):

```
MAXDELAY FROM GROUP "FFfast" 4.5 ns;
```

But maybe that the steadyState signal does not switch frequently. It switches only also with the CE signal as shown in the schema above. Then it can have a longer delay because the fast switching CE signal itself determines the used CE1 for CE. It means the FF for steadyState may be a member of the group for CE-driven FF and can be have a longer delay:

```
MAXDELAY FROM GROUP "FF_CE" 23.0 ns;
```

Now the router can use a longer path for the steadyState signal to build a CE signal for other FF. The longer delay is possible because this signal does not switch independent, it switches only with CE itself.

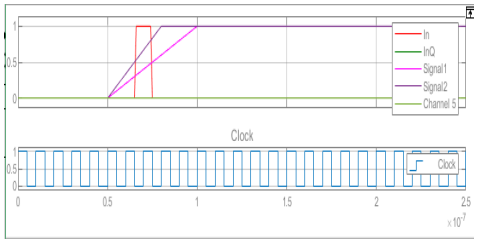
It is not expected that the logic for the CE1 with the longer delayed signal steadyState or some more other signals produces hazards or glitches because the CE itself is also used and it has the short delay. It determines the output of the LUT to build the CE1 to 0 so long CE='0'.

But you should take care that really one signal with a short path is always member of such an IF construct. If you write in VHDL

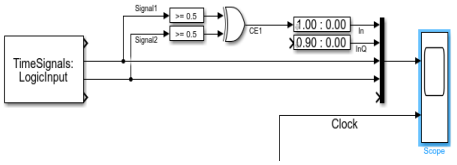
```
PROCESS(CLK, CE) BEGIN
  IF rising_edge(CLK) THEN
    IF (signal1 XOR signal2)='1' THEN
      Q2 <= Q1 AND someOtherLogic;
    END IF;
  END IF;
END PROCESS
```

then the router can also use CE to control switching the FF Q2.

If the signal1 and signal2 constrain a longer delay as the clock period then it is possible that a faulty intermediate state can occur where the signals are detected as different (XOR delivers '1') though the signals are not different after the delay. Then the CE can be '1' by mistake during a clock edge. It means this is a bad formulation in VHDL. At least one fast signal should be used as AND logic.



Only for info: In Simulink this is produced by the following schema:



The TimeSignals S-Function-Block is programmed with the given input pattern:

```
%time:0.000000000: LogicInput.signal1 = 0, LogicInput.signal2 = 0;
%time:0.000000050: LogicInput.signal1 = 0, LogicInput.signal2 = 0;
%time:0.000000100: LogicInput.signal1 => 1.0;
%time:0.000000080: LogicInput.signal2 => 1.0;
```

See also https://vishia.org/smlk/html/SmlkTimeSignals/SmlkTimeSignals.html. .

The signals itself are float with ramps, they are converted by the comparison block to boolean, with that approach a delay can be simulated. Because the first signal1 reaches the 1-level earlier than the signal2, temporary the '1' value is built by the XOR. That is a glitch. If the clock edge comes during the glitch, it is effective and causes a faulty behavior.

Note that the Simulink is not used for FPGA design for this example. It is only used to get simulation results of this hardware. I have used Simulink because it is my familiar simulation environment not only for FPGA, but more for controlling approaches. It should be possible for everybody to use him/here likely tool to get results. The results should be comparable and transferable between the tools.

Automatic check on Java level

The above shown situation can come also from a Java formulated design. It should be possible to check such constructs statically on Java level, yet not clarified. The check should test whether at least one signal is in AND constellation for the whole condition, and this signal should have the less constrain for delay.

3.3. Data types in Java for Fpga design in VHDL

VHDL distinguishes between the `BIT` type which has two states `'1'` and `'0'` (similar as boolean in Java) and adequate also for the `STD_LOGIC` type which may have also only the both values `'1'` and `'0'` if no more is used on the one hand …

And on the other hand VHDL knows a boolean type which is strongly used in `IF THEN` statements.

In VHDL there is no automatic conversion between both.

The typical boolean operators `AND`, `OR` are valid for all three, the boolean type and also for `BIT` and `STD_LOGIC` but of course with different results. The result follows the inputs, a mix is not possible. This may be one of the "*safety*" of the VHDL language, strongly distinguish, but it can be seen also as difficult and confusing.

In Java the situation is clarified with a boolean type which is also strongly and safe:

`boolean` in Java has two states, `true` and `false`.

Better than in C language the `true` representation is strongly defined.

More strongly as in C and C++ language a `if` statement needs a `boolean` expression. All other is faulty.

For the Java representation of a Fpga design with view to VHDL the following is defined:

Without a specific annotatio a `boolean` variable represents a `SIGNAL` of `BIT` type. The values `true` and `false` are associated to `'1'` and `'0'`. Negative logic is not supported as language feature. It is a feature of the user's semantic.

A `BIT_VECTOR` is represented by a `int` or `long` value (with up to 64 bit). This is better to test in Java as a `boolean` vector, because `BIT VECTOR` is often used as register values, also for shift and compare operations.

To access one bit of such an vector, there is a special access operation `Fpga.getBit(vector, 7)` where `7` is a bit number. The result of this operation in Java is `boolean` which is a `BIT` type.

Exact the same can be done with a `STD_LOGIC_VECTOR`, also represented in Java with `int` or `long`, distinguished only by the annotation `@Fpga.BITVECTOR` or `@Fpga.STDVECTOR`

Yet numerical values are not supported, done in the (near) future. The types `short` and `int` for 16 and 32 bit bit width without additional annotations should be used for 16- and 32 bit with algorithm (long also for 64 bit).

3.3.1. boolean expression and SIGNAL types with view to VHDL

In VHDL, if you have a boolean operation with a BIT or also STD_LOGIC type, you can write:

```
result <= (a AND NOT b) OR c;
```

If you do the same for an IF construct you must write:

```
IF (a='1' AND b='0') OR c='1' THEN
```

It means the operands should fistly converted to a boolean one. But it is also possible to write:

```
IF ((a AND NOT b) OR c) ='1' THEN
```

Here the expression is calculated as bit and then converted to boolean as last action for the `IF` usage.

But if you have a mix of `BIT` and `STD_LOGIC` it is not easy to write. If `b1` .. `b3` are BIT types and `s1` .. `s3` are `STD_LOGIC` you can write

```
b3 <= b1 AND b2;
s3 <= s1 AND s3;
```

But you cannot write a mix of both.

```
s3 = a1 AND b1;
```

For that it is necessary to write:

```
IF(a1='1' AND b1='1') THEN s3 = '1'; ELSE s3 = '0'; END IF;
```

Now the check is done on compilation level as boolean and the results are set. For the routing process and the logic in FPGA it is exactly the same if there are no tristate or wired lines or such one, or better: If the `STD_LOGIC` signals have in any case only the values `'0'` and `'1'`. It is a property of the language VHDL, which regards that a `STD_LOGIC` can or may have also other values then `'1'` and `'0'` as possibility. The given line does not give an information about that.

Hence it is a little bit complicated for the code generation. The Java2VHDL translator regards that stuff by using converting operations on VHDL side so far as possible.

3.3.2. Use cases of STD_LOGIC definitions

The `STD_LOGIC` was introduced in VHDL firstly to support timing simulation with the values `'U'`, `'X'` etc. That is unrelated for the Java2Vhdl, because in Java only the functional simulation is done which does not need `'U'` and `'X'`, a maybe timing simulation is done only in pure VHDL with the specific tools. But it may be important: If Simulation should be done with VHDL then all SIGNALs should be defined as `STD_LOGIC` and not as `BIT`.

Secondly, also important for Java2Vhdl, the Tristate and a wired or or wired and (pullup, pulldown lines) in the FPGA hardware can only defined with `STD_LOGIC`. For that the type

```
@Fpga.STDLOGIC char mySignal;
```

supports definition of a data type with the 9 possible states. They are given as characters for the char variable and adequate used for Java simulation.

The other possibility is to define

```
@Fpga.STDLOGIC boolean mySignal;
```

In this case this variable is translated to `STD_LOGIC`, but only the two states `'1'` and `'0'` are supported.

3.4. Modularity, with Object Oriented Approach

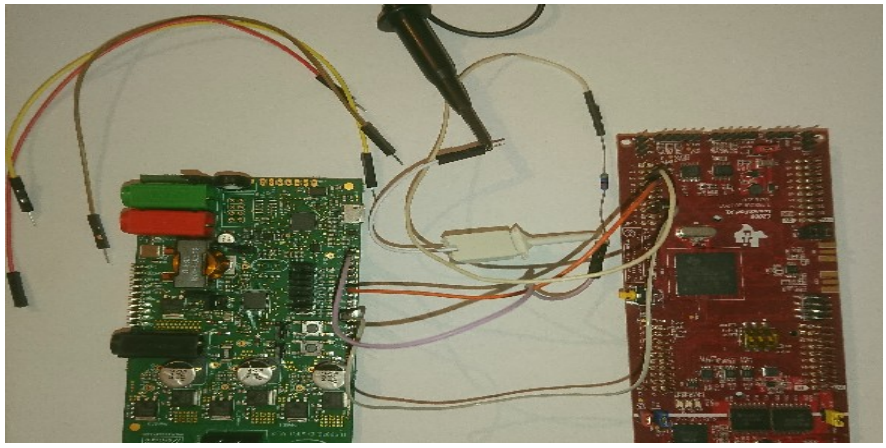
Of course, Java is an Object Oriented Language. It is important for ordinary software programming. The important common features for Object Orientation and their relations to hardware description approaches are shown in the following sub chapter.

3.4.1. Modularity in classic VHDL

VHDL was developed in a time as Object Orientation was not familiar. And of course, is Object Orientation proper for hardware approaches? Not from the eyes of the 1980th. But from the eyes of hardware test approaches and translation possibilities of the 2020th!

- The modularity of VHDL is classic, it is a dataflow approach:
- Define an interface to a module, this is in the module the `ENTITY PORT` definition part.
- This definition part should be repeated on usage as `COMPONENT` definition with the same content (a disadvantage, In C header files for that are used).
- On usage secondly in a `PORT MAP` signals should be connected. The connection of two modules needs extra signals.

For that the writing and maintain effort is high. This is especially a problem on refactoring. The importance of refactoring in modern software technology is some more higher as in the 1980th, related with the topics agile program development, complexity of solutions.



The image right side is only an illustration. It shows how two cards with simple plugs can be connected in a simple way. You are responsible to all by yourself, and you can do what ever you want, in this example have a resistor in the line. This is similar as wiring of modules in VHDL. You need an effort, but you can insert your own logic on module plugging level between the modules.

For hardware description in Java a similar approach is possible using an `In` and `Out` inner class for the input and output signals. This is necessary if one module should translated as one module to VHDL.

3.4.2. ObjectOriented approaches and their mapping for VHDL

- OO: Some related data are combined in a class.
- VHDL: This is also familiar in VHDL, using a `TYPE RECORD` and its instances. Secondly the classic modularity with one VHDL file per module fullfils this approach.
- OO: Some operations are related to the data, the operations which works with the data.
- VHDL: it is possible to build `RECORD`s, and `PROCESS`es, which touches only data of exactly one record instance. It means the `RECORD` definitions are oriented to the `PROCESS` definitions. The `PROCESS` is the operation or "*method*" related to the data. That is possible, not necessary in VHDL but may be seen as recommended. But other than in Object Oriented writing style the `PROCESS` can be related only to one `RECORD` instance and not to the `TYPE RECORD`, to the type as in Object Orientation. For the Java2VHDL translator the operation is defined type related of course (because of other reason in the constructor). The translator generates the `PROCESS`es from that constructor code, but instance related. In the normal software Object Orientation the instance relation is made on run time via the arguments of the method (in particular the instance pointer this, but also via other referenced instances as arguments). The Java2Vhdl translator resolves this arguments and builds `PROCESS`es for each instance as result of translation. It can be seen that the realization of the VHDL design is associated with a runtime, while the description in Java is the source for compilation.
- OO: Encapsulation of data. Data can be designated as private or protected. In both cases the data cannot be accessed from any other module, only from the own one ore derived ones in case of protected. The package private designation in Java (without public keyword) is a furthermore possibility similar to the friend designation in C++: Some modules can be associated to access there data one together, but the access is not possible from modules outside of this group.

  This feature is not firstly for protection, is supports a proper modularity without too much data dependencies.
- VHDL: In a module in an own VHDL file the data are also encapsulated. But the Java2VHDL translator supports also a ***flattened*** style with one VHDL file for some modules and some `TYPE RECORD` inside. But because of encapsulation of the data in Java, the access to the records are also sorted.
- OO: ***aggregated modules***: Any module can have references to other modules. In UML (Unified Modelling Language) this references are designated as composition (own sub module), aggregation (hard referenced other module, not changable) and association (changeable). But the access to properties of the other module are organized. Only public or package private members can be accessed, see topic above. It is able to clarify whether an access can be done only via access operaions ("getter") or also to public data. The data can be final especially in Java to forbid writing, changing of the data in another module. This is an essential idea to decrease to much functional dependencies between modules.
- VHDL: In the classic VHDL only the data flow via the Interface data (`ENTITY PORT`) is possible. This is not an aggregation concept, it is a dataflow concept.
- But if a `TYPE RECORD` is seen as a module, in a flattened design (one file for more modules), then the access to the data to other modules is immediately possible. Last not least, for the deployed design in the FPGA it is the same as using more as one VHDL file with ports. But from the view point of the VHDL source it is maybe more obviosly and more simple to read what`s happen. This is one of the basic idea of the flattened design of the generated VHDL file.
  From view point of the FPGA content description in Java, Aggregations are used, see chapter 3.2.3 More possibilities with Java2VHDL: References (aggregations) in Object Orientation

kind. Via the aggregations either public data can be accessed immediately, or the better approach is using the interface concept, see 3.4.4. Interface technology in Java for VHDL page 23. This is possible via modules in one flattened design.

- OO: There is the possibility to use derived types from an abstract type: The thinking should not be: Have any type, and derive it. The better thinking way is: Have some different types, how to find common properties and their embodiment in an abstract type of all these different types. It means the derivation is the secons approach, the abstraction is the first one. This is done because dealing with common properties (abstraction) is a good idea.
- OO: Related to Java programming languate, firstly the interface concept is one of the abstraction: Defining of a comman access possibility. The interface is implemented then in any type (class).
- OO: This feature of Object Orientation is very important for flexibility and testing.

  A module can be connected to different similar but not necessary equal other types (aggregation in UML).

  For testing an aggregated module can be replaced by a stub or mock which fullfils the interface for test approaches, to execute module-related (unit) tests.
- VHDL: This is an interesting topic. Also in VHDL a module / unit test may be seen as important. The test can be done in a reduced real desing in an FPGA, and also on software level. Exactly this topic is realized in the Java2Vhdl translator. See chapter 3.4.4. Interface technology in Java for VHDL page 23.

3.4.3. References (aggregations) in Object Orientation kind

The following example code snippet comes from the example project in the linked zip file on start of this document:

The modules in Java can be implemented in a flattend form in VHDL. Then each inner class from one Java file (module) is one TYPE RECORD definition in VHDL. The top level VHDL file or also a module can have some such RECORD TYPE definitions.

Example start of an inner class in a module MyModule.java:

Java: inner class for a PROCESS in VHDL

```
//fpga/exmpl/modules/ClockDivider.java
  /**Inner PROCESS class builds a TYPEDEF RECORD in VHDL and a PROCESS for each in...
   * Note: Need public because here the interface technology is not used (negative...
   * Compare with {@link Reset.Q}
   */
  @Fpga.VHDL_PROCESS public static final class Q{

    @Fpga.STDVECTOR(4) final int ct;

    /**This is the variable of the record accessed from outside.
     * Note: Need public because here the interface technology is not used (negati...
     * Compare with {@link Reset.Q#res}
     */
    public final boolean ce;

    /**Time of the latest set operation of any of the variables. */
    public int time;
```

Builds the record in VHDL:

```
TYPE ClockDivider_Q_REC IS RECORD
  ct : STD_LOGIC_VECTOR(3 DOWNTO 0);
  ce : BIT;
END RECORD ClockDivider_Q_REC;
```

The RECORD is instantiated, here with one instance, but also more as one instances are possible:

```
SIGNAL ce_Q : ClockDivider_Q_REC;
```

In VHDL from inside any PROCESS a variable of another module (RECORD instance) can be accessed, because of the flattened form with RECORDS.

```
ct_Q_PRC: PROCESS ( clk )
BEGIN IF(clk'event AND clK='1') THEN

  IF ce_Q.ce='1' THEN
```

That is possible because of the flattened property. This is simple also for the routing process for FPGA tools. The signal ce in the record instance ce_Q is immediately accessed. But:

The idea working flattened is bad for the real sources. It means that is not an approach for the Java sources. But instead, the Java sources can work with aggregations. The adequate line in Java for the process of the other module looks like:

Java: constructor for a PROCESS in VHDL usning ce

```
//fpga/exmpl/modules/BlinkingLedCt.java
    @Fpga.VHDL_PROCESS Q(int time, Q z, Ref ref, Modules modules) {
      Fpga.checkTime(time, ref.clkDiv.q.time, 1);  // for the ce signal, constrain...
      if(modules.ct_clkDiv.q.ce) {
```

- In Java the access is not done immediately to the variable in the other class. Instead a reference ref.clkDiv is used.
- Whereby ref is a module specific instance which holds all references, clkDiv is the reference

to the other module.

- The name after ref.clkDiv is not related to Vhdl. It is a local name in the module class in Java, not related to the really used instance.
- This is due to modularity. A module should not know which concrete instance of another module is used. This is not to be clarified in the module. It must be clarified in the higher-level programming which uses the modules.

How it is related: Any module can contain a Ref class with a Ref ref instance, here:

Java: Reference usage

```
//fpga/exmpl/modules/BlinkingLedCt.java
  private static class Ref {

    /**Common module for save creation of a reset signal. */
    final Reset_ifc reset;

    final BlinkingLedCfg_ifc cfg;

    /**Specific module for clock pre-division. */
    final ClockDivider clkDiv;

    Ref(Reset_ifc reset, BlinkingLedCfg_ifc cfg, ClockDivider clkDiv) {
      this.reset = reset;
      this.cfg = cfg;
      this.clkDiv = clkDiv;
    }
  }

  private Ref ref;
```

That are aggregations in UML wording (Unified Modeling Language for Object Orientated software technology).

As you see the name of the referenced module clkDiv is a private name, not the name of an existing instance in the parent module. This is per se unknown, not determined which modules and also which type of modules are used.

The constructor of the Ref have to be called in the constructor of the module, because ref is final:

Java: constructor and init operation for the Ref class

```
//fpga/exmpl/modules/BlinkingLedCt.java
  /**Module constructor with public access to instantiate.
   * <br>
   * Note: The arguments should have the exact same name and type as in the {@link...
   *
   * @param reset module provide the reset signal on power on and as input
   * @param clkDiv module provide a clock enable signal: {@link ClockDivider.Q#ce}
   */
  public BlinkingLedCt ( Reset_ifc reset, BlinkingLedCfg_ifc cfg, ClockDivider clk...
    this.ref = new Ref(reset, cfg, clkDiv);
    this.modules = new Modules(this.ref, this);
  }

  /**Non parameterized constructor if the aggregations are not existing yet. Use {...
   * for aggregation. */
  public BlinkingLedCt () {}
  /**The init operation should be used instead the parameterized constructor with ...
   * The modules should be known each other. Then only one module can be instantia...
   * The other module can only be instantiated firstly without aggregations, then ...
   * <br>
   * Note: The arguments should have the exact same name and type as in the {@link...
   * @param reset aggregation to the reset module.
   * @param cfg aggregation to the configuration
   * @param clkDiv aggregation to the clock divider.
```

```
   */
  public void init ( Reset_ifc reset, BlinkingLedCfg_ifc cfg, ClockDivider clkDiv) {
    this.ref = new Ref(reset, cfg, clkDiv);
    this.modules = new Modules(this.ref, this);
  }
```

Here it is important that the name of the arguments in the constructor is the same as the name of the aggregation reference in the Ref class. That is not a problem and also a good style. It is necessary for the translation to VHDL.

Now, in the top level in Java all modules should be instantiated. The Java-toplevel decides which modules are used. For that there is a Modules inner class:

Java: Module class defines the used modules with their relations

```
//fpga/exmpl/modules/BlinkingLedCt.java
  /**The modules which are part of this Fpga for test. */
  public class Modules {

    /**The i/o pins of the top level FPGA should have exact this name ioPins. */
    public BlinkingLed_FpgaInOutput ioPins = new BlinkingLed_FpgaInOutput();

    /**Build a reset signal high active for reset. Initial or also with the reset_...
     * This module is immediately connected to one of the inputFpga pins
     * via specific interface, see constructor argument type.
     */
    public final Reset res = new Reset(this.ioPins.reset_Inpin);

    public final Test_Combinatoric_BlinkingLed vhdl_Combinatoric = new Test_Combin...

    public final BlinkingLedCt ct = new BlinkingLedCt(this.res, BlinkingLed_Fpga.t...

    public final ClockDivider ce = new ClockDivider(this.res, this.ct);

    Modules ( ) {
      //aggregate the module afterwards
      this.ct.init(this.res, BlinkingLed_Fpga.this.blinkingLedCfg, this.ce);    //...
    }
  }

  public final Modules modules;
```

Here three modules are defined which are used in the top level of the FPGA. In the constructor of the modules the aggregations to the other modules are defined. Note that also a module.init(…) can be used beside the constructor. This is necessary if circular dependencies are needed. For the Java2VHDL translation both can be used, arguments of the constructor and arguments in the associated init(…), not shown here.

The name of the modules (the composite reference name in Java) build the name of the RECORD instances in VHDL:

```
SIGNAL ce_Q : ClockDivider_Q_REC;
SIGNAL ct_Q : BlinkingLedCt_Q_REC;
SIGNAL res_Q : Reset_Q_REC;
```

On translation from Java to VHDL an index (TreeMap in Java) is built for each module which associates the intern name of a reference to a module with the real used module. This index is internally used for translation, but also reported in the report file (option -rep:path/to/file.txt)

```
== Module: ct
  localName         | accessed module     {@link J2Vhdl_ModuleInstance#idxAggregat...
--------------------+----------------
  cfg               | BlinkingLed_Fpga : BlinkingLed_Fpga
  clkDiv            | ce : ClockDivider
  reset             | res : Reset
--------------------+----------------
```

Now, while generating the `PROCESS` for the module ct, this association index is used to assign the `clkDiv` in Java with ce as name of the `RECORD` instance in VHDL.

How is this index built:

- The name of the instance in Java given as actual argument of the constructor respectively the init(…) operation BlinkingLedCt as this.ce. this. is only for Java internals, ce is used.
- The name of the formal argument of the constructor or init(…) of BlinkingLedCt is clkDiv. It is also gathered.
- This both names are stored in the index shown above.
- The name of the actual argument of the Ref constructor and also the name of the reference itself should be the same: clkDiv. This is necessary as style guide. It should not be a problem. Test and translation of this stuff may be also possible, but a non necessary effort.

So the Object orientated writing style in Java is translated to a flattened immediately access to the correct `SIGNAL` (usual a Flipflop, a register etc.).

3.4.4. Interface technology in Java for VHDL

The last chapter has shown using Object Orientation with aggregations and their association to instances. That is necessary for flexibility in module usage (which combination) and also for the test of modules

There is one more approach: using interface technology.

Basics of interface technology

In the chapter above it is able to associate, which module is used. But the access inside the module is inflexible. The variable name of the module is immediately used, in the example q.ce.

Two things should be free to do:

- Changing of internal names in a module for further development. Using modules should not be refactored. That is also the basic idea of private encapsulation in the Object Orientation.
- Using another module should be possible with similar properties for using but another internal design This idea is the abstraction and inheritance in Object Orientation. It is like in daily life. You need a car to drive. Which car is not important. All cars are basically similar, there are exactly equal in the required properties.
- The last point is also for test: You can replace a module with a test bed emulation of the original module to make an independent module or unit test. The test bed replacement has another inner structure.

Look to another part of the example:

Java: interface usage

```
//fpga/exmpl/modules/BlinkingLedCt.java
        if(ref.reset.res(time, 20)) {          // interface access to assigned her...
          this.ct = ref.cfg.time_BlinkingLed();
```

This is a snippet from the BlinkingLedCt PROCESS in Java: A reset information is requested. If reset is given, then the counter ct is set to its reload value.

But it is not designated here from where reset is coming and how it is built, and also from where the reload value is coming and how it is built. This should be clarified outside of the module. The module only needs connection for it.

The translated VHDL design looks like:

VHDL: generated code of interface usage

```
      IF (res_Q.res)='1' THEN
          ct_Q.ct <= TO_STDLOGICVECTOR(BlinkingLed_Fpga_time_BlinkingLed);
```

Here the relations to the reset signal and the reload value is full clarified: A signal from another RECORD instance (another module) is used for reset, and the reload value is defined as constant in the VHDL code above:

VHDL: constant definition due to interface operations

```
CONSTANT BlinkingLed_Fpga_onDuration_BlinkingLed : INTEGER := 10;
CONSTANT BlinkingLed_Fpga_time_BlinkingLed : BIT_VECTOR(7 DOWNTO 0) := x"64";
```

But in another usage configuration of the same module, without change of the Java code of the module, the reload value can for example come from another Signal vector as for example

VHDL: other result of interface usage

```
          ct_Q.ct <= otherRecord.reloadVal;
```

References with interface type

Explained from hardware view: A simple signal or variable in Java with a dedicated type, BIT or STD_LOGIC_VECTOR or such in VHDL or boolean or int in Java is like a cable with a standard plug, which's properties are general defined, not specialized. Whereas an interface is a cable with a plug of a certain design which can only be plugged into the corresponding counterpart. The conditions on the interface are well defined. But it is not defined how the implementing device behind the plug works.

The invocation of the interface relation uses the referenced modules ref.reset and ref.cfg for this both values from the chapter above.

Look on the definition of all references for input values from outside of the BlinkingLedCt module:

Java: Reference class definition

```
//fpga/exmpl/modules/BlinkingLedCt.java
  private static class Ref {

    /**Common module for save creation of a reset signal. */
    final Reset_ifc reset;

    final BlinkingLedCfg_ifc cfg;

    /**Specific module for clock pre-division. */
    final ClockDivider clkDiv;

    Ref(Reset_ifc reset, BlinkingLedCfg_ifc cfg, ClockDivider clkDiv) {
      this.reset = reset;
      this.cfg = cfg;
      this.clkDiv = clkDiv;
    }
  }

  private Ref ref;
```

As you see, the type of the ref.reset is Reset_ifc, as also the type of ref.cfg is BlinkingLedCfg_ifc. Look firstly to the defintion of the Reset_ifc:

Java: Definition of Reset_ifc:

```
//fpga/exmpl/stdmodules/Reset_ifc.java
package org.vishia.fpga.stdmodules;

public interface Reset_ifc {

  /**Returns true for reset. false for normal operation.
   * @param time current time for the access
   * @param max check whether the time of the accessed signal was latest set to (t...
   * @return true then reset active, false: normal operation.
   */
  public boolean res ( int time, int max);

}
```

Using of interfaces is very proven in Java (as also another Object Orientated Languages), and it is not so complex to translate it to VHDL, if some sensible simple style guides are additional regarded.

It means, to get an information about the reset state as boolean value in Java or BIT in VHDL, the reset() operation is called in the implementing module. That is the formal form of the plug. How this function is implemented - depends on the plugged module.

The association for ref.reset is set on construction. For this example the init(…) operation is responsible to plug:

Java: Reference class definition

```
//fpga/exmpl/modules/BlinkingLedCt.java
  /**The init operation should be used instead the parameterized constructor with ...
   * The modules should be known each other. Then only one module can be instantia...
   * The other module can only be instantiated firstly without aggregations, then ...
   * <br>
   * Note: The arguments should have the exact same name and type as in the {@link...
   * @param reset aggregation to the reset module.
   * @param cfg aggregation to the configuration
   * @param clkDiv aggregation to the clock divider.
   */
  public void init ( Reset_ifc reset, BlinkingLedCfg_ifc cfg, ClockDivider clkDiv) {
    this.ref = new Ref(reset, cfg, clkDiv);
    this.modules = new Modules(this.ref, this);
  }
```

This operation creates the Ref instance and sets the references with the outside given reference to the implementor, the supplier for this signal with the interface type. Alternatively this can be also done also with the constructor of the module.

The value for the supplier of the reset signal in form of the Reset_ifc comes from another module. The connection is done in the Moduls class on top level:

Java: Module definition and relations

```
//fpga/exmpl/modules/BlinkingLedCt.java
  /**The modules which are part of this Fpga for test. */
  public class Modules {

    /**The i/o pins of the top level FPGA should have exact this name ioPins. */
    public BlinkingLed_FpgaInOutput ioPins = new BlinkingLed_FpgaInOutput();

    /**Build a reset signal high active for reset. Initial or also with the reset_...
     * This module is immediately connected to one of the inputFpga pins
     * via specific interface, see constructor argument type.
     */
    public final Reset res = new Reset(this.ioPins.reset_Inpin);

    public final Test_Combinatoric_BlinkingLed vhdl_Combinatoric = new Test_Combin...

    public final BlinkingLedCt ct = new BlinkingLedCt(this.res, BlinkingLed_Fpga.t...

    public final ClockDivider ce = new ClockDivider(this.res, this.ct);

    Modules ( ) {
      //aggregate the module afterwards
      this.ct.init(this.res, BlinkingLed_Fpga.this.blinkingLedCfg, this.ce);    //...
    }
  }

  public final Modules modules;
```

Here you see the invocation of the init(…) operation with the first argument `this.res. It is the reference to the Reset module which supplies the interface. Adequate it is done for the others. The implementor of the BlinkingLedCfg_ifc is here assembled in the environment class denoted by BlinkingLed_Fpga.this and their in blinkingLedCfg, see following chapter 3.2.4.5 Interface access instances for stubs (replacement for non existing module outputs). You see also here that the references are fulfilled for example for the Reset module by construction, here with reference to the pin of the Input interface on the FPGA.

Implementation of an Interface from the whole module

If you have a simple module, as here the Reset which has only one task: Deliver the reset signal, this module can/should immediately implement this interface. The image right shows a module, it has only one task, supply

voltage 5 V, one interface, an USB plug, no more.

The Reset module in Java implements the interface on module level:

Java: Implementation of Reset.reset():

```
//fpga/exmpl/stdmodules/Reset.java
public class Reset implements FpgaModule_ifc, Reset_ifc {
  .....
  @Override public boolean res ( int time, int max) { return this.q.res; }
```

It's very simple: It accesses the variable res in its own inner class q which is the PROCESS class for the reset functionality. This is the detail of this Reset module.

The Java2Vhdl translator evaluates this term in the context of this module which is given as reference. Hence the result is the already above shown VHDL code:

VHDL: generated code of interface usage

```
      IF (res_Q.res)='1' THEN
          ct_Q.ct <= TO_STDLOGICVECTOR(BlinkingLed_Fpga_time_BlinkingLed);
```

Generally, only the term of the return statement is evaluated. If the interface operation contains more statements, it is usable for simulation on Java level. For the ordenary software execution of interfaces concepts in Java as also other programming languages the interface operation can also change data and do anywhich else. But this "full freedom to do whatever you want" is also criticized in some software writing guidelines. Normally it is a good style to prevent or forbid changes in another software module on only access operations. Changes should be done with (also maybe interface-) operations which are named set…, do…, process… or exec…or such else. For that things the processes in VHDL are responsible. It means execution routines cannot be invoked in the Java context for VHDL hardware descriptions.

But the return expression can be more complex, for example can contain logical combinations, access to bit ranges, comparison and all what is possible also in other expressions. This expression is generated inside the accessing PROCESS. If the same interface operation is used more as one time in different contexts, this operations are generated on any access. It is similar as execution of the operation in the Java runtime. Also there any access executes the statements again.

It means, the PROCESS of the module can prepare proper signals for simple access if it is expectable that this signals are used more as one time. On the other hand the optimizer while routing can accomplish the aggregation of multiple equal accesses. How these accesses are to be designed is up to the Java description of the VHDL developers.

Now, any module can implement this Reset_ifc. It means for testing, or for another design, you can use another module for the reset signal without change of the inner structure of the using module. The Java2Vhdl translator gets the correct access.

Interface agents or access instances

If a module has more interface connections, especially connections of the same type more as one, then it is not optimal or not possible for the same interface type to implement the interface with the module as a whole. For that Java offers an interesting possibility: Implementing the interface in instances of anonymous inner classes. The inner class has only the task to implement the interface with the possibility to access to the environment (outer) class which is the module. For that it can be seen as agent which is responsible to access to inner details of the module from outside, without the necessity to regard this access in the module itself. On the other hand it can be designated as access point.

Java: Example for interface access or agent

```
//fpga/exmplBlinkingLed/fpgatop/BlinkingLed_FpgaInOutput
package org.vishia.fpga.exmplBlinkingLed.fpgatop;

import org.vishia.fpga.Fpga;
import org.vishia.fpga.stdmodules.Reset_Inpin_ifc;

public class BlinkingLed_FpgaInOutput {
.....
  /**Get the reset pin as referenced interface access from a module.
   * Using the {@link org.vishia.fpga.stdmodules.Reset} may be seen as recommended...
   */
  @Fpga.IfcAccess public Reset_Inpin_ifc reset_Inpin = new Reset_Inpin_ifc () {
    @Override public boolean reset_Pin() { return BlinkingLed_FpgaInOutput.this.in...
  };
```

Such an anonymous class instance or agent is shown here for the access to the reset pin itself. The pin is located in a static class Input definition in this module with a final Input input = new Input() instance. This is used to generate the PORT definition in VHDL, but this is not the point of interest here.

For the port definition, especially, there may be a lot of pins. A flexibility is necessary because the same FPGA functionality may be necessary to implement in different environments, different card types etc. For that the interface access to the port pins is a very good idea and reduces adaption effort and error possibilities because of pin confusion.

Hence, the port definition is associated to an own module (which can be replaced for different applications without replacing and adaption of other modules). This module contains:

```
public class MyPortsVersionXY_FPGA {
  public static class Input {
    //.... input pins
  }

  public static class Output {
    //.... output pins
  }

  public final Input input = new Input();     //instantiation
  public final Output output = new Output();

  //some access operations:
  @Fpga.IfcAccess public Type_Inpin_ifc access_Inpin = new Type_Inpin_ifc () {
    @Override public boolean access_Pin() { .... }
  };
```

Now this module can be connected with dedicated access instances to any module which needs an input pin. More as that: A module which needs primary an input pin can also be connected not immediately to the pin, but to another module may be with a filter functionality. This other module should only offer also such an interface access agent with the same interface type.

Of course this Port module is only an example. The interface access agents can be used anywhere for more complex applications.

Interface access instances for stubs (replacement for non existing module outputs)

Especially for test, but also for variants of a FPGA design sometimes a functionality is not necessary. Without adaption of the module, the inputs can be connected to constant signals driving '0' or '1'. The place and route will be remove this unnecessary parts. But on source level an adaption should not be necessary.

Adequate can be done if some constant values are necessary. The constant values should not be set in a module itself to preserve flexibility, it should be delivered from outside. It’s the same story: a module should not be modified and adapted for a specific use, but the functionality should be defined externally with appropriate connections.

For the Blinking Led example this is done with timing parameters:

Java: interface usage

```
//fpga/exmpl/modules/BlinkingLedCt.java
        if(ref.reset.res(time, 20)) {          // interface access to assigned her...
          this.ct = ref.cfg.time_BlinkingLed();
```

The request is adequate, a value should be accesses. It is not first intendet that this is a constant value. It can be also a signal which contains this reload value as dynamic information.

Especially for tests sometimes signals are terminated by fix values, if this functionality is not in focus yet. The signals should be fulfilled, of course, but inputs may be always false or '0' for this test condition or for more simple usage of a module. The last hint is also important: A module can contain more functionality as necessary for the amount of usages. If inputs are terminated with constants and outputs are not used, then the routing process will remove this unnecessary parts. The sources can contain it, without disadvantages. It means also that a module should not be prepared or adapted for specific usage situations. It can be uses "as is", and the outer signals and connections determine which is really implemented in the FPGA.

For that all reasons a second use cases and translation goal for interface access is supported by the Java2Vhdl converter: If the return expression delivers a constant value, then a CONSTANT definition in VHDL is created and used.

Follow the example: The interface to access the time_BlinkingLed() is implemented here in the main or top level file also with an access instance:

Java: interface implementation for a constant

```
//fpga/exmpl/modules/Fpga_BlinkingLed.java

  /**Provides the used possibility for configuration values.
   */
  @Fpga.IfcAccess BlinkingLedCfg_ifc blinkingLedCfg = new BlinkingLedCfg_ifc ( ) {

    @Override @Fpga.BITVECTOR(8) public int time_BlinkingLed() {
      return 0x64;
    }

    @Override public int onDuration_BlinkingLed() {
      return 10;
    }

    @Override
    public int time() { return 0; }  // set from beginning

  };
```

The essence is: The return … expression contains only one term, it is a constant. If this situation is detected, this constant value is generated in VHDL with the name of the module and the name of interface operation, it`s unique. The result is (already shown above):

VHDL: constant definition due to interface operations

```
CONSTANT BlinkingLed_Fpga_onDuration_BlinkingLed : INTEGER := 10;
CONSTANT BlinkingLed_Fpga_time_BlinkingLed : BIT_VECTOR(7 DOWNTO 0) := x"64";
```

The advantage of building a constant element instead the immediately constant value is: It is obvious in VHDL. Elsewhere only the constant value will be written there as evaluation of the expression, and the back tracking to the Java code is difficult.

If for instance inputs are terminated by a constant to prevent usage, the generated VHDL code contains for instance:

VHDL: constants for input termination example

```
CONSTANT Toplevel_Fpga_inputX : BIT := '0';
 ....
```

```
PROCESS ...
  IF Toplevel_Fpga_inputX = '1' THEN
    ...
```

In VHDL you see that the input of the module is used, but the expression is never true. Hence, the router removes this part, proper for the use case.

The idea to implement such termination interfaces on the top level comes from the idea, that the top level, or the whole design decides about usages of details of some modules. Another module as stub is not necessary. It simplifies the design. But of course a specific module can also be used for that.

How does the interface technology works for Java2Vhdl

As you have seen in the chapters above: The references and interface operations are full resolved to simple accesses to the internals of another module. Here a question can be asked: Why is it so complicated to run the programs with virtual operations? The interface usage in Java is the same as virtual or overridden operations with late linkage in C++.

The answer is: If references are not full clarified from beginning, the mechanism of the virtual table is necessary. This is the common approach for software execution. But if references are clarified from beginning in the construction phase, and they are never changed, then the execution level can use the simple immediate access. It may be also in the same kind for a Java just-in-time (JIT) compiler: This is the translation from Java byte code to machine code on loading a class. After loading and preparing the immediate machine code is executed. Thats why Java execution needs more startup time, but then it runs fast. If references are organized as final in the constructor, the JIT can optimize it. Whereas, C++ has not such an JIT compiler, and should use the virtual operation call anyway.

Now, for the FPGA usage, the references are also clarified from beginning, with the adequate specific constructs also using the init(…). Hence it can be resolved on translation time.

There are two things to considerate: The refernced module and the interface operation.

- On translation of each module the interface operations (only in the first level classes of each file, not in inner classes) are gathered. The return statement is searched and evaluated.
- Either it is a simple constant, then the constant is stored in the idxConstDef:
- All constant definitions: J2Vhdl_FpgaData.html#idxConstDef
- and the const definition is stored in
- constant definition of an interface operation: J2Vhdl_ModuleType.IfcConstExpr.html#constVal. Or if it is not a simple constant the expression is stored in
- expression of an interface operation: J2Vhdl_ModuleType.IfcConstExpr.html#expr

Both alternatives are then stored module-type-related in

interface operation per module type: J2Vhdl_ModuleType.html#idxIfcExpr

On translation this table is reported in the -rep:reportfile.txt for this example as:

```
== J2Vhdl_ModuleType: Reset
  ifcOperation()    | access    {@link J2Vhdl_ModuleType#idxIfcOperation}
--------------------+----------------
  reset()           | this.q..??refres @;
--------------------+----------------
```

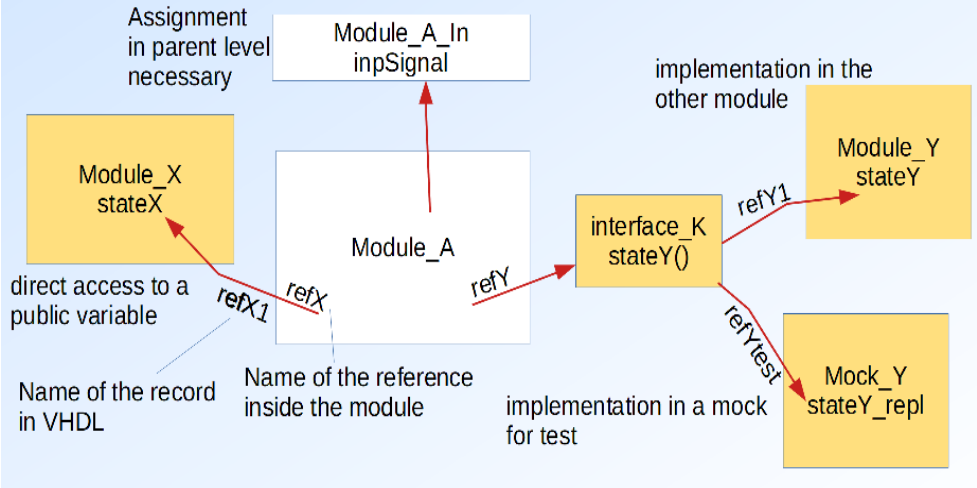
If an interface operation is detected as part of an expression, then the reference is dissolved, detecting the type of the reference, and the interface operation is searched (binary search) in that module type related index.

Then either the constant definition is immediately used, or the given expression is evaluated as

any other expression too, whereby the module context is switched.

3.4.5. Overview modularity

The next image should show the variants in a common way (unrelated to the example), also with public immediately access in the modularity:



- The Module_A_In in this image is an inner In class adequate to the VHDL approach using PORT. It is possible.
- Left side an immediately access to a inner variable in Java is shown. The variable should be public. The disadvantage for that is, that the white Module_A should know the defintiion of Module_X. That is possible and also familiar in Java. It is bad for module test, because on testing also an instance of the ModuleX should be present in the test environment, exact this module or a replacement but with the same package path, which is terrible for source maintaining (version management).
- But also for that immediately usage the inner aggregation refX is translated to the ref1 on usage. It means the type should be known, but the instance is determined outside. This is important if more as one instance for one type is existing.
- Right side the interface usage is shown. The interface operation should have the same name as the accessed variable. That is not necessary from the view point of Java. It is necessary for the Java2Vhdl translation. But it is possible.
- You can connect different implementation modules also on FPGA level in VHDL. It means the variable stateY can exist in different records types. Only the name is determined.
- Now the advantage for testing is shown. That is only done in Java, not on VHDL level. That's why the test implementation of the interface_K can use any implementation, should not access exact one variable.
- Note that also for the module implementation for Vhdl translation the access operation here for stateY can contain more as the simple access to the variable, for example log informations, adaption, a debug break point. That is all possible on Java level, not translated to VHDL. For VHDL anytime the simple access to the record signal with the same name as the access operation is translated.

3.5. Including existing VHDL files

There are two reasons to include a given VHDL file in the generated sources:

a) Existing reused modules

b) Often the tools for FPGA design have features to generate VHDL files for specific features, for example for a RAM block.

VHDL is the standard for FPGA design (beside Verilog), hence VHDL is necessary to include.

For simulation on Java level, the functionality of the VHDL code should be emulated. For example for a RAM module this should be simple. You may also only simulate only the basically functionality to fulfill the interface to the module.

See handling in chapter 4.8.3. Included VHDL modules page 60 and 4.9 Java source for an emulated VHDL module page 68

3.6. State machines, enum

State machines are familiar both in software and in hardware. In software solutions often the so named "Harel-Statecharts" are used. (https://en.wikipedia.org/wiki/David_Harel), which defines nested and parallel states with history. That's a part of the "Unified Modeling Language" (UML): https://en.wikipedia.org/wiki/UML_state_machine, definition by omg.org: http://www.omg.org/spec/UML/. For Software state machines often events are responsible to switch the states.

But all these practical things are not substantiated in the Java core definition. It is possible in Java, but with specific classes.

Hardware state machines are often a little bit simpler. Especially there are not event driven, the system clock switches because of conditions. But the idea of nesting and parallel states is and should also possible and practicable. Also the idea of the event may be possible, presented by an additional condition (the event bit). This means that thinking in terms of states should not distinguish too much between software and hardware.

A basis for this is the enum definition for states and the enum type for the state variable. Exactly this is implemented in the current version of Java2Vhdl.

3.6.1. State variable with 1-of-n decoding

A state variable is basically a numeric value. For 5 states you need 3 bits to code numbers between 1 to 5 as presentation of the state. But another coding is better:

Use 5 bits if you have 5 states, the state variable is 1-to-5 decoded (one bit from 5 is set).

```
00000 invalid state
00001 state A
00010 state B
00100 state C
01000 state D
10000 state E
```

With this coding schema only one bit is need to detect a specific state, and this is a lesser effort in routing, for the FPGA hardware resources. The number of FlipFlops in a FPGA are usual enough, the scare resource is often the lines for routing and the look up tables (LUT) for combinatoric.

Only the quest of the invalid state (after reset) needs testing of all 5 bits, or more for more states. In the current version of the Java2VHDL translator, the set of a state influences all state bits. Most of them are set from 0 to 0, only the elapsed state bit is set from 1 to 0 and the new state bit is set from 0 to 1. Maybe the router can improve this situation by optimizing. Setting all state bits clarifies a possible mistake if more as one bit is set.

3.6.2. enum definition in Java

Known from C/++ language, a enum is only a definition of an integer constant value by a symbol, and the enum type assures only acceptance of these enum constant definitions. In Java the enum definition is a little bit more powerful:

- Java creates an enum Object with some values as constant.
- The enum value in an enum variable is the reference to one of these enum constant objects. This needs 8 byte for a pointer, in opposite to C/++ where for example 2 Bytes for an int16_t enum base type are sufficient. But Java runs only on powerful processors, not an disadvantage.

The first point offers usage of some more properties of the enum constant.

Java: enum definition

```
/**States for xyz*/
public enum MyState {
  Undef  (0b00000, -1, '0'),
  /**State A */
  A  (0b00001, 0, 'A'),
  B  (0b00010, 1, 'B'),
  C  (0b00100, 2, 'C'),
  D  (0b01000, 3, 'D'),
  E  (0b10000, 4, 'E'),
  ;
  public final int _val_;
  public final char show;
  final int bit;
  /**Constructor for the enum value
   * @param value The internal used Vhdl value
   * @param bit the bit number for VHDL translation, following the value
   * @param show character to show the state in println
   */
  MyState(int value, int bit, char show) {
    this._val_ = value; this.bit = bit; this.show = show;
  }
}
```

This is a strikingly example. The state names A..E are of course any identifier, but you have a namespace inside the enum definition (clashs are prevented, better than in C/++). The fields value and bit are required. The field show is proper usable for output of test results, see some programming examples. The state Undef is the undefined state after reset.

3.6.3. state variable as enum

The state variable in Java has this enum type. Concluding, Java assures only set with a valid state value. For the VHDL conversion it should be a bit vector (not a STD_LOGIC_VECTOR, not supported in the current version). A BIT_VECTOR is sufficient for usage.

Java: state variable definition inside a process class:

```
  @Fpga.BITVECTOR(5) final MyState state;
```

The number of bits should follow the enum definition, elsewhere mistakes in the generated VHDL code are resulting.

3.6.4. query state variables

The query is usal written as:

Java: state variable query:

```
if( z.state == MyState.Undef ) {
  //.....
else if( z.state == MyState.A ) {
  //.....
else if( z.state == MyState.B && otherSignal ) {
  //.....
```

As you see, the content of the state variable is compared to the given state constant value. Usage of the identifier of the enum type definition is necessary in Java, it clarifies name clashing and increases obviousness and readability. Usage of switch … case is possible and may be also recommended in Java, but it is not yet supported in the Java2Vhdl translator. As given in the third query line the query of the state can be also combined with the logic relation to other signals (variables), which are the conditions for state usage. This is better possible in an if (… construct.

The Java2Vhdl translator detects a definition of a state constant with a bit value >=0, then only the query of the state bit is produced in VHDL. For this example the result in VHDL is:

VHDL: state variable query:

```
IF(module_Prc.state = Module_MyState_Undef) THEN
  --.....
ELSE IF ( module_Prc.state(0)='1') THEN
  --....
ELSE IF ( module_Prc.otherSignal AND module_Prc.state(1))='1' THEN
  --....
```

The first comparison of state used the state value 0b00000 which is not marked with a valid bit, hence the Java2VHDL translator generates a full comparison of the state vector with the constant value which is defined as

VHDL: state constant definition

```
CONSTANT Module_MyState_Undef : BIT_VECTOR(4 DOWNTO 0) :=  "00000";
```

The second comparison knows from the Java enum definition, bit 0 is associated, and generates the simple access to this bit value. The third query combines the BIT access to the state BIT_VECTOR bit with the BIT variable otherSignal as BIT-AND and converts outside of the paranthesis to the necessary boolean value for the IF query.

Java: state variable query:

```
myBitVariable = (z.state == MyState.C) & otherBitVariable;
```

This is a combinatoric from a state query and a boolean, it is translated very simple to:

VHDL: state variable query:

```
module_Prc.myBitVariable <= module_Prc.state(2) AND module_Prc.otherBitVariable;
```

It means it is only a BIT logical combination.

3.6.5. set state variables

To switch the state only the new state should be set:

Java: state variable set:

```
    if(condition){
      this.state = MyState.C;
```

This is immediately translated to

VHDL: state variable query:

```
    IF condition THEN
      module.Prc.state <= Module_MyState_C;
```

It means all state bits are determined. The optimizer of the FPGA routing process may detect not changed bits.

3.6.6. Nested and parallel states

As mentioned in the introduction, Harel-Statemachines have nested and/or parallel states. Especially nested states helps get overview, it follows the practical requirements.

In Java level for VHDL any sub state in a state should have its own state variable. Entry in a nested state needs set two state variables. Query of the composite state (the outer of nesting) queries and set only the outer state variable, the inner state remains its value. This is helpful for the entry to the "history state", to the given remained inner state back again.

These things are all able to do on user level. A translation between UML diagrams with given Harel state charts to Java and then to VHDL may be nice and possible, but not presented here. See links in german:

https://vishia.org/StMn/pdf/StatemProgr_de_2020-01-12.pdf

https://vishia.org/StMn/pdf/EventQueue_de_2020-02-20.pdf

3.7. Test in Java

The Test of the logic is a very important part.

General a test should be done in two categories:

- Test under exact defined conditions expecting dedicated results. This is important for two situations:
  - Test of features which are assumed in requirements.
  - Repeated tests after changes to fast and automatic clarify, all is ok.
- Test under accidently conditions. This tests are important to study the behavior independent of planned tests and exactly defined requirements.

  It is possible that some conditions are not exactly defined, but that definitions should be intrinsinc necessary.

Furthermore, the tests should be done in three situations:

- Test of the ready to use logic in the routed FPGA under several conditions (several input signals), of course in both categories as above presented, planned tests and accident tests (sometimes denoted by "white noise tests").
- Test of the whole logic on Java level with dedicated test cases, or maybe also for accident inputs. Whereas a random value generator may be used in software. But of course the accident situations follows only the programmed randomizations, this is not a "white noise test".
- Test of modules ("unit test"). Usual for that only planned tests are determined. The behavior of a module should follow exactly situations, this follows the denotation "design by contract". The contract of the behavior of a module should be well defined. A module should be manageable (rather than a complex system).

These are pure basics about tests that are generally applicable.

3.7.1 step and update operations

The content of the step(int time) and update() operations are not used for the VHDL translation. Essential for the VHDL translation are only the existence of the inner classes designated by the annotation `@Fpga.VHDL_PROCESS` in the modules and the instantiation of the modules in a class Modules in the top level file and also possible in modules for sub modules.

It means the (manually programmed) content of this routines should follow the existing module instances and process classes. Then only the behavior of the test is identically with the original FPGA behavior. It means, intrinsic, this operations should be generated also for Java level. But this is not done yet.

The step routine prepares the states before the next clock. The update routine is the clock, it is the manifestation for the next state.

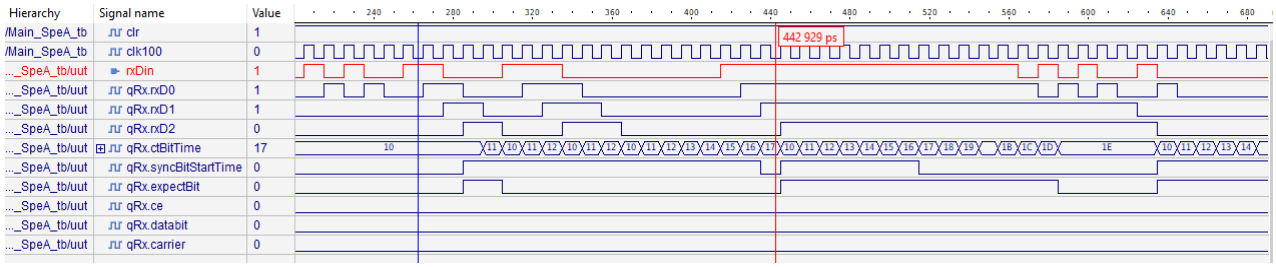
3.7.2 Input signals for test simulation in Java

If you are testing the whole FPGA design usual the top level file has a Input and Output inner class with the designated input and output instance which presents the pins of the FPGA. Hence should only set the elements of the input with the appropriate values. This input values can come from simple test bed algorithm, or from more complex algorithm maybe also fet from values in tables, the test cases.

If you are testing a module usual the module has interface connections. You should satisfy this interface requirements by implementations in the test bed, for the appropriate input signals.

3.7.3 Output signals for manually evaluation of the test results

With given tools of FPGA simulation a graphical output is usual for instance:



Of course an adequate approach is possible for Java simulation because Java has graphical possibilities, it can be programmed in a specific way, or given routines in libraries of from other tools are usable. But is this the best one?

The very simple solution presented below may be also satisfactory:

```
ct.ctLow__________:0001 0000 ffff 61a7 ...0001 0000 ffff 61a7 ...0001 0000 ffff 61...
ct.ct_____________:63   63   63   62   62      62   62   61   61      61   61   60...
ct.time___________:                   '5.001.20             '10.001.40          ...
io.ledA___________:_____________________________________________________________AA...
io.ledB___________:_____________________________________________________________...
=====================================================================
ok Test_BlinkingLed
```

This is the content of a simple text file, able to view with any normal stupid or better a powerful text editor, for example https://notepad-plus-plus.org/ or http://www.jedit.org/. The original lines produces by the Blinking_Led example has a length of ~5000 character, no problem for viewing, shifting and copy also a part to the clipboard using the "rectangular selection" mode in Jedit or the "column selection mode" on Nodepad++.

The lines are time lines. Each column position in each line presents the same time, more exact the start column of a longer information. Of course you should use a monospace font in the editor.

The resolution of such an output can be one system clock per column or character, or also condensed, for example a dedicated CE (clock enable) occurrence per column.

Because you can use proper characters to show a state, this is more obviously as only simple lines. Especially you can assemble more signals in one line using proper characters. For that look on a snippet of another example:

```
rx2.rxCE__________:___________----f---------f---------f--=======e=========e=====...
Time.time_________:^``8.54      |       |       |      ^``9.00       |       |      ...
```

This first line shows three signals: * The 'f' or 'e' is a special clock enable signal, always one clock period wide. * The _ is used for the idle state, then also no ce should be occur. * The --- is one specific state, in this state a ce may occur as shown for the one clock period. * The === and + is each another state, also with occurrence of ce. The letter for the occurrence of the ce signal is also

changed for better visibility.

To build this output, you should override the operation addSignals in the derived class of org.vishia.fpga.testutil.TestSignalRecorder in the following form:

Java: Example to build a character for output

```
char ce;
if(thism.qrx.ce) { ce = thism.qrx.dataState ? 'e' : 'f'; }
else if(thism.qrx.stateA && thism.qrx.stateB) { ce = '='; }
else if(thism.qrx.stateA) { ce = '-'; }
else if(thism.qrx.stateC) { ce = '+'; }  //(not all is full decoded)
else { ce = '_';}
this.sbCe.append(ce);
```

You can use more mnemonic character for the states, and also for short signals in the state. The only one challenge is, a good documentation of the used characters. You get a very compressed representation with more expressiveness than many lines.

The second line shows the time, which is equidistant here. This signal is created using a instance of

org.vishia.fpga.testutil.TestSignalRecorder.Time

Using this class, the time is always written on start of line, and then in a proper distance with 5- and 10 divisions.

The time in the line may not be equidistant, depending of the output routines. This is an advantage. You can show an incident in detail zoomed in time, whereas non interesting time spreads are condensed. The example shows it. The trigger for writing details is here the ctLow with the value 0x0001. It is shortly before zero crossing. The ct, the higher count is shown, its value changes if the ctLow reaches ffff and the reload value is also set if ffff is reached before, and not as expectable, if 0000 is reached before. This is the written logic. The time is also presented, only on end of this zoomed spread because it needs upto 13 character positions. The time is a millisecond value, proper readable till 10 ns resolution. As you can see the period of one count step of ct is not 5.00000 ms as expected than 5.00020 ms, 200 ns longer. This is an effect which is better able to see with this numeric time information than in a diagram with measurement.

The algorithm in Java to sort the output for this information is not complicated:

Java: Output routine for signals

```
//fpga/exmpl/modules/BlinkingLedCt.java
    @Override public int addSignals ( int time, int lenCurr, boolean bAdd ) throws...
      BlinkingLedCt thism = BlinkingLedCt.this;
      int zCurr = this.sbCt.length(); // current length for this time
      int zAdd = 0;                   // >0 then position of new length for this time
      if(thism.ref.clkDiv.q.ce) {       // because the own states switches only wi...
        if(thism.q.ctLow == 1) {        // on this condition
          this.wrCt = 5;              // switch on, write 5 steps info
        }
        if(--this.wrCt >0) {          // if one of the 5 infos shouls be written:
          StringFunctions_C.appendHex(this.sbCtLow, thism.q.ctLow,4).append(' '); ...
          StringFunctions_C.appendHex(this.sbCt, thism.q.ct,2);                    ...
          if(checkLen(this.sbtime, zCurr)) {      // add the time information if h...
            StringFunctions_C.appendIntPict(this.sbtime, time, "33'331.111.11");  ...
          }
          zAdd = this.sbCtLow.length();  //length of buffers for new time determin...
        }
        else if(this.wrCt ==0) {         // end of the 5 steps, append .... as sep...
          this.sbCtLow.append("..... ");
          zAdd = this.sbCtLow.length();  //length of buffers for new time determin...
        }
      }// if ce
```

```
      return zAdd;       // will be used in TestSignalRecorderSet.addSignals(zAdd)...
    }//addSignals
```

Generally for this example, adding a signal is only done if clkDiv.q.ce is true, because: This module changes only its state with this ce clock enable. It saves calculation time if this is quest firstly. Then the trigger for the output is mdl.q.ctLow == 1. With this condition the next 5 ce times are presented.

Because 5 characters are appended to sbCtLow, this is the longest appendix, all lines are elongated to this legth, as you can see in the ouput. It means this time uses 5 column positions in the output, in all output lines.

You should not see the effort to program the output, you should see the advantage to design your necessary output for complex signals and test cases. Programming in Java is much easier and safer than, for example, in the C++ language or in certain scripting languages often used for simulation tools. You can make extensive use of auto-completion during editing, you will get immediately writing mistake messages (syntax errors) and hints for correction.

This horizontal output as also a vertical output, the time is continued in the lines than in columns, is organized by two classes in the package org.vishia.fpga.testutil. It is explained in detail in Java2Vhdl_TestOutput.html and for the Blinking Led example in chapter Java2Vhdl_ToolsAndExample.html#JavaSrcTest

3.7.4. Test of modules or the whole design on Java level

In Java programming also the environment, the "test bed" should be written. In this test bed the inputs are determined and the outputs are gathered maybe in lists appropriate to states and times. This results can be compared with expected results in a very more proper way because Java is a safe and familiar language.

You can use several test environment approaches. It is recommended to use the simple org.vishia.util.TestOrg, see ../../../Java/docuSrcJava_vishiaBase/org/vishia/util/TestOrg.html

Last not least the link to the so named StimuliSelector should be also offered here:

../../../StimuliSel/html/StimuliSel.html

This tool is not used yet for the FPGA test, but it is possible. It is used yet for Simulink tests, for algorithm tests in C/++, see ../../../emc/html/TestOrg/testStrategie_en.html#testStrategies and more.

3.8 Writing style of logic - data assignment versus situation thinking

It is a general question, orientation to data assignment or situation evaluation. What is meant by this?

- Situation evaluation: Programming is a familiar idea of program flow. The typical construct is if … then … else. "If a situation is given, then do this and that". This is also the prevailing mindset for VHDL and for the programming style of the 1980th (before Object Orientation became really familiar).
- Orientation to data assign: The real truth of programming is: Influencing data. That is also one of the ideas of Object Orientation. The objects are the data. Program flow and operation is only a way to influence the data. The quest is the state of data. The flow is only a mediator for action.

In graphical models such as Simulink or Labview immediately a data flow is modeled. This is a data flow between function blocks, another level, nor related here. This data flow concept seems to be in opposite to the modeling with UML, which shows not a flow but data relations especially in Class- and Object Diagrams.

Another topic is functional programming. This is more oriented to data assign because all is result of a function, and a function is the result of the input data. This is mentioned here only as additional information.

Look on examples for Situation evaluation versus Orientation to data assign:

3.8.1 Style: Situation evaluation, program flow

```
if(condition) {
  data = changed;
} else if(other_condition) {
  data = other_input;
  other_data = changed;
}
```

Here it is not clarified whether the data are changed anywhere other too, as well as the other_data are changed in one of the given situation. The change of data are not well described. But the reaction of situations (test a condition) is well described. Usual the condition is only tested one time.

3.8.2. Style data assignment orientation (data flow)

```
if(condition) {
  data = changed;
else if(other_condition) {
  data = other_input;
} else {
  //comment: data remain unchanged.
}
if(other_condition) {
  other_data = changed;
} else if(specific_condition) {
  other_data = specific;
} else {
  //other_data unchanged in all other situation
}
```

Here we have two program blocks, because two different data are handled. The data are not related, except one other_condition influences both. But the conditions are secondary. Each program block for one data should be complete for this data.

The disadvantage of data assign orientation: Usual it needs more code. The test of conditions are

programmed more as one time. For all data extra. That needs also more runtime for the program in a controller (if the compiler does not optimize).

The advantage of data assign orientation is: The program clearly shows, how data are changed. If one looks on the part for one data specification, it is complete. The data will not be changed anywhere else.

In practice, both approaches are often combined. Fundamental situation as "clear", "reset" are programmed with the situation approach: "what's happen on clear:…". But for details the data orientation should be better.

The argument of longer execution time is not applicable for the hardware design if complex combinatoric are stored in intermediate variable. If that is not done, the router may optimize the combinatoric too. Only the readability of the code is decisive.

3.8.3. Ternary or condition operator in Java: condition ? a : b

Now, for thinking to data orientation, in Java (as also in C/++ the so named ternary or condition operator can be used:

```
data = condition ? changed : other_input;
```

This is the simplest form with two variants. It can be more complex:

```
data = condition ? changed
     : other_condition ? other_input
     : data;
```

This is the same effect as the if in the examples above. But it is well obviously. Also the else branch is exactly determined. The compiler of software languages will remove the unnecessary assignment for the unchanged data. For this situation it is well documented that the data are used unchanged.

In Java you can use a `final` keyword for such data:

```
final int data = condition ? changed
               : other_condition ? other_input
               : old_data;
```

This is similar as https://en.wikipedia.org/wiki/Functional_programming functional programming. With the final keyword the compiler has more capabilities to optimize.

And now, for hardware design:

The new state of a FlipFlop or a FF group is complete determined by such an final functional construct. The logic is obvious at a glance.

3.8.4. Solutions for pure VHDL

As mentioned above, VHDL was created in a time (1980th) where structured programming was familiar. Additional to the known if … then … else … end if; VHDL knows:

```
'case 'selection is when choice ⇒ statements
```

This is also typical for situation thinking. The statements can contain any assignment. The selection describes which situation is checked.

```
assignment ⇐ value1 when condition1
value2 when condition2
else value else;
```

This is exactly the behavior for the data assignment thinking. Only one variable is under consideration. It is set under several conditions.

This `when … else` statement was introduced also for using in processes (behavioral programming) with VHDL-2008. It is ideal for this case. But unfortunately VHDL-2008 doesn't seem to be considerate by all tools and all thinking:

https://hardwarecoder.com/qa/73/vhdl-when-else (2023-05-21)

4. Try to only use when else outside a process even though it's supported in VHDL-2008. Why? Because in 2020, some synthesis tools still have some bugs compiling VHDL-2008. Perhaps in the future this won't be an issue. Plus, if you have to support an older FPGA using older tools, you won't have 2008 as an option. Keep these issues in mind when you code. There are better options than when else if you need to have it in a process, like using case instead.

It means it is not able to use for a tool independent translation from Java.

3.8.5. Java2VHDL for condition operator

Hence, conclusion, in processes only `IF` can be used for translation of the Java condition operator.

```
//Java:
data = condition ? changed
     : other_condition ? other_input
     : data;
```

is translated to:

```
IF condition THEN data <= changed;
ELSIF other_conditon THEN data <= other_input;
ELSE data <= data;
END IF;
```

whereas the last line can be removed. This is the same behavior as written with the conditional operator in Java. What is obviously: The same left side term for the data assignment (`data <=`) is written for each branch. But this is not an disadvantage. It is readable, and the router and simulation tools with VHDL can proper deal with it. Only the readability of the source is a little bit not optimal because one line may contain a writing mistake. But because it is generated code - no problem.

Unfortunately outside of a process `IF THEN ELSE` cannot be used, here the `WHEN` construct is used for translation of such an conditional expression.

The same problem occurs on conversion of a bit in `BIT` logic from `STD_LOGIC`:

```
 -- in a PROCESS:
 IF myBitValue = '1' THEN myStdValue <= '1';
 ELSE myStdValue <= '0';
 END IF;
```

and outside of a PROCESS

```
 myStdValue <= '1' WHEN myBitValue = '1' ELSE '0';
```

3.8.6. Multiplexer in hardware design, problem of WHEN ELSE

In Java it is very simple to write in an expression:

```
boolean q = a & (b ? c : d) | e;
```

In this case `c` and `d` is multiplexed by selecting with `b`. The rest is boolean logic.

In VHDL theoretically a

```
c WHEN b='1' ELSE d
```

is existing. It works outside of a process. But this construct doesn't seem to be considerate by all tools and all thinking:

https://hardwarecoder.com/qa/73/vhdl-when-else (2023-05-21)

4. Try to only use when else outside a process even though it's supported in VHDL-2008. Why? Because in 2020, some synthesis tools still have some bugs compiling VHDL-2008. Perhaps in the future this won't be an issue. Plus, if you have to support an older FPGA using older tools, you won't have 2008 as an option. Keep these issues in mind when you code. There are better options

than when else if you need to have it in a process, like using case instead.

It means, the simple expression for a multiplexer, which is familiar in hardware, is not possible for VHDL.

But instead, VHDL likes to see IF constructs:

```
IF b='1' THEN t = c; ELSE t = d; END IF;
```

This works also in processes, it is a basic, supported, known. But what is the disadvantage: As part of an expression we need a temporary variable. And this part of expression should be extracted as extra statement.

```
PROCESS
-- Java: boolean q = a & (b ? c : d) | e;

SIGNAL b_sel : BIT;  --process variable
BEGIN
IF b = '1' THEN b_sel := c;
ELSE b_sel := d; END IF;
q <= (a AND b_sel ) OR e;
```

This complicates the readability. But we have VHDL, the best and safe language for hardware.

3.8.7. Programming in loops

Also VHDL knows loops. If you follow https://vhdlguide.readthedocs.io/en/latest/vhdl/behave.html#problem-with-loops then loops should not be used.

In software two types of loops should be well distinguished:

Classic loops for repeated execution till a condition is met. It is typical a while loop, or repeat - until.

- Loops only to execute an algorithm for all given instances. That is not really a execution in the loop, it is only written as loop in software to process all given instances. This can be done also parallel and also sequentially. But because the instances are contained in a so named container the loop works for all member of the container. This is typically a foreach loop. In Java it is written as:

```
//Java
for( Type element : container ) {
  //do for all element.
}
```

  Such constructs are parallelizable also in software, for example distributed on several cores of a processor. Why: Because all operations which are executed in a loop one after another are independent. It means the order of execution is not important.

- This scheme can now be used for hardware designs for parallelization. Of course the FPGA should have enough resources for the task.

  It is also interesting that each element in a container can have a different derived type. It means really, different operations are executed (via virtual operations). Now, thinking in hardware: You have a planned container with elements, the elements are a little bit different and you need a design for this elements:

  a) You write code for each element extra.

  b) You use a for-each loop. But because the type of the elements are only known in runtime (depends on other program parts which may be not in focus) such a VHDL code as generated parallel code can be built also especially in run time. The executed statements are not executed, instead they produce VHDL code.

This is another approach, currently not supported by the Java2VHDL concept, maybe done in future.

4. Java2VHDL - User’s guide

Code templates and outputs

4.1. Working tree organization for sources and tools

Look in the given example Example1_BlinkingLed.zip. The content of this file may be but need not be the template for the file tree organization.

Generally the idea of the https://www.vishia.org/SwEng/html/srcFileTree.html is used, a file tree similar as the familiar known maven or gradle file tree, whereas maven or gradle itself is not used here.

```
Path/to/myWorkingTree
 +-src/      all sources should be versioned
 +-tools/    tools loadable from internet
 +-build/    output directory for build outputs (may be in RAM disk, or temp location)
```

The `tools` and `build` sub directory are created with batch files (Windows-oriented) or adequate shell scripts. The empty `build` directory will be created and removed by the above shown `+clean.bat` and `+clean_mkLinkBuild.bat`.

The example contains some more files and directories on root level, but this files are really only for the simple example. It should be assembled with adequate content in a user project inside the src tree.

```
Path/to/myWorkingTree
 +- +clean.bat              batch helper file to clean all
 +- +clean_mkLinkBuild.bat  batch helper file to create the build directory
 +- +gen_Vhdl_Example1.bat  batch file for start generation
```

The last files calls a file inside the source tree at adequate positions where ist shoud be versioned..

The following files are given, to load the tools from internet, see also https://www.vishia.org/SwEng/html/srcFileTree.html#libsTools and the following chapter 4.3. Tools necessary for Java to Vhdl translation and test support page 48

```
Path/to/myWorkingTree
 +-src/load_tools/
        +-+loadTools.bat     script to create the tools directory and load tools
        +-tools.bom          so named "bill of material" to determine which tools
        +-vishiaMinisys.jar  a simple java executable to execute the load
```

The `src` working tree is organized in the following form:

```
src
 +-vishiaFpga/                       The necessary component for all
 |  +-java/                          contains Java sources
 |  +-makeScripts/                   make for general thinks:
 |     +-genTmpl_Java2Vhdl.bat       generation file for the template files
 |     +-+genjavadoc_vishiaFpga.sh   shell script to generate Javadoc
 |  +-genVHDL_cmp/                   contains the template generation result to use
 |  +-.filelist                      For versioning, contains time stamps (missed in git)
 |  +-.git                           reference to the git archive for this component
 |  +-.gitignore
 +-exmpl_vishiaJ2Vhdl_BlinkingLed/   A user project, here the example
 |  +-java/                          The Java sources
 |  +-lattice/                       A lattice project
 |  +-makeScripts/                   dto
 |  +-genVHDL_cmp/                   The lattice project uses the VHDL from here ...
 +-load_tools/        helper for the tools, not in maven concept.
```

Below `src` there are components. In original *gradle* tree it is `main` and `test` to separate between the product relevant files in `main` and test files. But *gradle* has another approach for components, load it temporary, and that is not desired here.

The next level below the component designation is the kind of source files (as in *gradle*), here Java files or some VHDL files, or `makeScripts` or whatever else as also `cpp` for C and C++ files, maybe necessary for other parts of the whole project or just lattice for a *Lattice Diamond* project.

Inside the Java components you have the familiar *Java package tree*, starting in this case all with `org/vishia`. The Java package tree is familiar since the first Java development in the 1990th. It is a world wide unique deterministic of packages using the revers internet address as first members. Hence all parts which are developed related to the https://www.vishia.org web page (Java related parts) are denoted in the `org/vishia/…` package tree. For your own you should use your web presence as start path such as `com/siemens/department/…` if you are from the Siemens company or `com/bosch/department/…` if you are working in the Bosch company or whatever else, as usual in your company. This should be only understand as hint or notice, may or may not be important.

4.2. The platform to edit the Java sources for VHDL

It is recommended to use Eclipse, but also another IDE is possible as your choice.

You can also use any text editor to view the sources.

To compile and run the example independent of an IDE you can use the batch file compilation. But you need he javac compiler (part of JDK, Java Development Kit, https://www.oracle.com/java/technologies/downloads/. or also for OpenJDK for example in https://www.azul.com/java-alternative-vendors

Look at `src/exmpl_vishiaJ2Vhdl_BlinkingLed/java/_make/+makejar_exmplBlinkingLedFpga.sh`.

The compile result will be written in the `build` folder. From there it can be run starting `src/test/bat/test_Example1_BlinkingLed.bat`. This is for your experience.

4.3. Tools necessary for Java to Vhdl translation and test support

The necessary tools for Java to VHDL translation are really less. It is only jar files to work with Java.

Java itself should be familiar for usage. This examples and tool files are related to the long term provided Java-8 version from Oracle, but also some open source Java may usable.

After loading the Java files from the internet via clicking on src/load_tools/+loadTools.bat you get the following files to work:

```
2022-05-23  14:14               502 +loadTools.bat
2022-05-23  14:48               584 tools.bom
2022-05-23  14:49         1.500.128 vishiaBase.jar
2022-01-24  20:25            81.128 vishiaMinisys.jar
2022-05-23  14:49            56.282 vishiaVhdlConv.jar
               5 Datei(en),      1.638.624 Bytes
```

If you look on src/load_tools/tools.bom you see the following:

```
#Format: filename.jar@URL ?!MD5=checksum

#The minisys is part of the git archive because it is need to load the other jars,...
vishiaMinisys.jar@https://www.vishia.org/Java/deploy/vishiaMinisys-2022-05-31.jar ...

#It is need for the organization of the generation.
vishiaBase.jar@https://www.vishia.org/Java/deploy/vishiaBase-2022-05-31.jar  ?!MD5...

##Special tool for Java2Vhdl
vishiaVhdlConv.jar@https://www.vishia.org/Java/deploy/vishiaVhdlConv-2022-05-31.ja...
```

This textual file is executed by the Java class `org.vishia.minisys.GetWebfile` which is contained in the here also registered `vishiaMinisys.jar`. It contains the internet location for the jar file, the destination file name and a MD5 checksum. You can do this actions also manually, build and compare the check sum. The files are able to view and load in the given location, this is https://www.vishia.org/Java/deploy. You find also the source files beside the jar files with the same name, only with the extension `-source.zip` All is open source, you can study the algorithm, and also compile it newly. The source-zip archive contains a _make directory. You should only place all depending jar files or sources (that is here `srcJava_vishiaBase`) side beside. Depending jar files should be placed in a tools directory beside:

After newly translation you get the same jar files with exactly the same binary content and hence the same check sum. This is the approach of reproducible build, see also https://www.vishia.org/Java/source+build/reproducibleJar.html and also https://reproducible-builds.org/reports/2020-03/. It means you can both check the correctness of the MD5 check sum and check whether the sources are really valid for the given binary.

As you see, the `VhdlConv` itself is only a small file consist of a few Java classes. No more is necessary. But the basics, independent of the VHDL approach, the Parser, text generator etc. are all contained in the `vishiaBase.jar`. But this file has also only 1.5 MByte. The other used tools are only the Java-8 system from Oracle. No other tools and executable are used. Nothing is stored in any temporary or home/user locations. Getting the core of the job done usually doesn't require sprawling tools.

4.4. The component srcJava_vishiaFpga

This component contains some Java files. They are necessary in a user's project for test and for using annotations and call specific operations. It means this component should be used as source file tree. It is located for the example.zip in:

```
src
 +-vishiaFpga
   +-genVHDL_cmp/                    comparison which should be generated from tmpl_J2Vhdl
   +-makeScripts/
   +-java/
      +-srcJava_vishiaFpga/
         +-org/vishia/fpga/
                        +-stdmodules/*.java     useable in the design
                        +-testutil/*.java       useable for test on Java level
                        +-tmpl_J2Vhdl/*.java    template files for projects
                        +-Fpga.java             define some standard operations and...
                        +-FpgaModule_ifc.java   the essential module interface
```

You don't need (must not) change the content of these files, only use it. It is also versioned (yet TODO Github)

4.5. The translation Java to VHDL

This is only the start of a command line execution, for the example:

```
java -cp tools/vishiaBase.jar;tools/vishiaVhdlConv.jar org.vishia.java2Vhdl.Java2Vhdl
 ... -sdir:src/exmpl_vishiaJ2Vhdl_BlinkingLed/java
 ... -sdir:src/main/java/srcJava_vishiaFpga
 ... org.vishia.fpga.exmplBlinkingLed.fpgatop.BlinkingLed_Fpga
 ... -o:build/BlinkingLed_Fpga.vhd -tmp:build/ -rep:build/BlinkingLed2Vhdl_report.txt
```

Note, the ` … ` on line start means continue of the line before. This is a very long line because of the arguments, not obviosly. Therefore a better solution is possible, given in the example:

```
java -cp tools/vishiaBase.jar;tools/vishiaVhdlConv.jar org.vishia.java2Vhdl.Java2Vhdl
 ... --@%0:convArgs
::convArgs ##argument label. Space after the label, trim trailing spaces and comme...
::-sdir:src/exmpl_vishiaJ2Vhdl_BlinkingLed/java            ##source dirs from current
::-sdir:src/main/java/srcJava_vishiaFpga
::-top:org.vishia.fpga.exmplBlinkingLed.fpgatop.BlinkingLed_Fpga   ##top level fil...
::-o:build/BlinkingLed_Fpga.vhd                              ##output
::-oc:build/ BlinkingLed_Fpga.lpfx                           ##part of contraint
::-tmp:build/
::-parseData                                                 ## The java data tre...
::-parseResult                                               ## The parse result ...
::---parseLog                                                ## an elaborately pa...
::-rep:build/BlinkingLed2Vhdl_report.txt                     ##report with meta i...
pause
```

General with the argument --@path/to/argfile some arguments can be read from a file. Whereas each line of the file is one argument. That makes it also possible to use white spaces in arguments without quotation marks. This feature is contained in org.vishia.util.Arguments. But often an extra file for that is not nice. That's why the same file as the command file is used, given with the %0 in Windows batchfile syntax. With a given label after the argfile path after a colon :convArgs the argument processor searches this label in the argument file, also after up to 5 comment characters till the 5th position. The same comment characters are tested in the following lines to regard which are argument lines. That lines should be all commented lines for the batch script, starting with ::. Then the arguments are written after, one in each line. The first line without these comment character, here the pause line is then the termination of argument lines. The arguments can be commented with the given ## as comment characters after the label. One space between label and argument comment characters forces removing trailing spaces in the line, which is often sensible but not at all. Hence it can be controlled here.

With this argument designation the arguments are well readable.

A short explanation of the arguments comes if the converter is started without arguments:

```
Java2Vhdl made by HSchorrig, 2022-02-16 - 2023-04-01
 see www.vishia.org/Fpga/html/Vhdl/Java2Vhdl_ToolsAndExample.html
-i:path/to/template.vhd  ...optional, if given, read this file to insert
-o:path/to/output.vhd
-oc:path/to/constraint.ext
-top:pkg.path.VhdlTopModule ... the top level java file (without .java, as class path)
-sdir:path/to/srcJava  ... able to use more as one
-sl ... optional, if given, remark src and line
-parseData ... optional, if given, writes the parser java data tree
-pd ... optional, same as -parseData
-parseResult ... optional, if given, writes the parser result
-pr ... optional, same as -parseResult
-parseLog ... optional only with -parseResult, writes an elaborately parser log file
-pl ... optional, same as -parseLog
-tmp:path/to/dirTmp for log and result
-rep:path/to/fileReport.txt   ... optional
```

This is of course only a short description, with the link to this document.

- The -i:path/to/template.vhd can be used if only a part of the VHDL file should be generated, the frame is given with this file. The generated parts are firstly the TYPE … RECORD definitions and the SIGNAL ….:_REC instances, ` and secondly the PROCESS . The given file should contain labels in the following form:

```
  ...start of the file, with heading, ENTITY, Ports
ARCHITECTURE BEHAVIORAL OF ....

-- INSERT Java2Vhdl
  ... This parts are replaced by the new generated one TYPE ... RECORD definitions
  ... and SIGNAL ....:_REC` instances
-- END Java2Vhdl
  ... further content, SIGNAL and COMPONENT definiton, especially the
BEGIN
  ... and more given content
-- INSERT Java2Vhdl
  ... This parts are replaced by the new generated processes
-- END Java2Vhdl
  ... finishing content
```

- `-o:path/to/output.vhd` is also used if -i:… is given. It means the -i:… file will not be replaced, only read. It may be recommended to generate a new file first to a temporary location in the file system, and then compare because of changes, at least replace.
- `-oc:path/to/constraint.ext`: If given some information about time cell groups are written into it, see 4.12.2 Association between PROCESS variables and time GROUPs page 88.
- `-top:pkg.path.VhdlTopModule` This is the class path with package path of the top level Java class for the FPGA design. Usual this class contains a class Modules inner class to determine all other sub modules.
- `-sdir:path/to/srcJava` This argument can be given more as one (usual) as search path for the Java files. It contains the directory where the Java package path starts (with org/…), not the directory of the Java file itself.
- `-sl` means "source line". If given then in the generated -o:… file the source file and the line of the Java source for the appropriate generated VHDL line is written as ---path/to/src: line. This helps to associate generated lines and Java source lines. However, using this feature makes it a little bit difficult to compare a newly created file with the previous version because often the lines are shifted in the source, hence only all the line numbers are changed. It makes really changes lesser obviously. It may be recommended to generate both versions, with and without this option, and store both as second source, without line numbers for a simple version comparison and with line numbers to search assiciations with the Java sources.
- `-tmp`:path/to/dirTmp It is possible to output intermediate files for parsing results etc. especially during development, not used in the compiled version.
- `-rep`:path/to/fileReport.txt This is an interesting report file about modules, interfaces, variables and should be stored beside the VHDL output file.

4.6. Java source for top level FPGA class

See also the example chapter [topclass].

1. Java: class for the FPGA top level

```
package com.company.department.project;

import org.vishia.fpga.FpgaModule_ifc;
import org.vishia.fpga.stdmodules.Reset;

class MyFpgaTop implements FpgaModule_ifc {
  public class Modules {                                                  // (1)
    public final MyFpgaIo ioPins = new MyFpgaIo();                        // (2)
    public final Reset reset = new Reset(this.ioPins.resetInPin);         // (3)
    final ModuleXY moduleXY = new ModuleXY();                             // (4)
    Modules ( ) {
      this.moduleXY.init(this.reset, this.ioPins.specificIfcAccess);      // (5)
    }
  }
  public final Modules modules = new Modules();                           // (6)

  public MyFpgaTop() { }                                                  // (7)

  @Override public void reset ( ) {                                       // (8)
    this.modules.reset.reset();
    this.modules.moduleXY.reset ();
  }

  public void input ( ) {                                                 // (9)
    this.modules.moduleXY.in.varx = this.modules.ioPins.input.in_Pin;
  }

  @Override public void step ( int time ) {                               // (10)
    this.modules.reset.step(time);
    this.modules.moduleXY.step(time);
  }

  @Override public void update ( ) {                                      // (11)
    this.modules.reset.update();
    this.modules.moduleXY.update();
  }
  public void output ( ) {                                                // (12)
    this.modules.moduleXy.output();
    this.modules.ioPins.output.out_Pin = this.modules.moduleXy.getValxy();
  }
}
```

(1) The class for the top level FPGA should contain an inner `class Modules` which defines the used modules for the Java2VHDL translation. On translating this top level VHDL file as given argument of translator `-top:...`, all module classes will be detected and also translated, also for sub modules inside the modules named here.

(2) One of the module should be named `ioPins`. This module is recognized as the IO pin description of the whole FPGA. See next chapter

(3) The modules are defined by the chapter 4.8. Java sources for Modules page 56. Note: You can have more as one module (instance name) with the same type (class name). Any module instance will be placed in the FPGA. But of course the java file of the class is only parsed once.

(4), (5) The modules can be either instantiate by the default constructor, then they should have a init(…) call. Or they can instantiate with initial arguments without init(). he arguments are used form the here named modules itself. See also chapter 4.8.1. Connections and inner modules,

page 56.

`(6)` In the modules can be simple initialized, because the top level has no other connections.

`(7)` The constructor of the top level is also simple and empty.

`(8)` The reset() should call all reset() from the modules. This is not used for Java2VHDL, but important for simulation.

`(9)` The `input()` and also the `output()` operation is evaluated for Java2VHDL. Its assignments generates assign statements in VHDL. The input() should fill variables in a possible in sub-Instance in a module, see chapter 4.8.1. Connections and inner modules, page 56 `(2)`.

`(10)`, `(11)` The overridden operation `step()` as well as `update()` should call all adequate operations for all modules. It is not used for the Java2VHDL translation, but of course for the test.

`(12)` The `output()` operation is evaluated for Java2VHDL. Its assignments generates assign statements in VHDL especially to set the output pins of the FPGA.

Module connections

The top level Java file contains `modules`, but also a module Java file (see chapter 4.8. Java sources for Modules page 56) can contain sub modules. This (sub-) modules are named in the `class Modules` as inner class of the top level class and also possible as inner class of a Module.

The (sub-) modules are connected together. But other as in classic VHDL, where the modules are built from different VHDL files, all of this modules are combined in a large VHDL file. It is not so large, it can be overviewed. The Java-modules builds `TYPE … RECORD` and `SIGNAL` definitions in VHDL. Hence it is simple to access for one process in other `SIGNAL` structures, and also the logic is more simple to map to the implmenting FPGA (view in Floorplanner etc) as using the module structure in original VHDL with some VHDL files with its `PORT` structures and `PORT MAPPING`

How Modules are connected with references:

This is described as concept also in chapter 3.4.3. References (aggregations) in Object Orientation kind page 19. Also the following chapter 3.4.4. Interface technology in Java for VHDL is meaningfully. The arguments for the references in the modules constructors and init() are always access operations with the interface type. That enables the simple exchange of one module by another module with other content and definitions, but the same type of access operations. This is important for flexible unit tests, as known in the Object Oriented programming, also usable here.

The other kind of module connections are the assignment of variables. This is near the classic VHDL approach. The in and out instances in the modules java classes are similar a PORT definition in a VHDL file. But other than in modular VHDL this instances are also mapped to a `TYPE ...In_REC RECORD` and `SIGNAL ..._In : ...In_REC;` definition which is more simple.

But for modules which are included as real VHDL module this assignments builds the necessary `PORT_MAP` assignments. See chapter 4.8.3. Included VHDL modules page 60.

4.7. Java source for Pin definition FPGA class

See also the example chapter 5.2 The FPGA pin description file.

The Java class for the FPGA pin description is that class in the top level *.java file, which is detected in the sub class Modules definition as module with the name ioPins:

Java: Modules class in top level contains ioPins definition of the FPGA

```
  /**The modules which are part of this Fpga for test. */
  public class Modules {

    /**The i/o pins of the top level FPGA should have exact this name ioPins. */
    public BlinkingLed_FpgaInOutput ioPins = new BlinkingLed_FpgaInOutput();
```

From the appropriate class the inner classes static class Input and static class Output are recognized. The elements in this class are the pins. The pins cannot be vectors, only boolean and char for BIT and STD_LOGIC with the appropriate annotations:

Java: class for IO pins of the FPGA

```
class MyFpgaIo implements FpgaModule_ifc {
  public static class Input {
    public boolean reset_Pin = true;
    public char tristatePin = 'L';
  }
  public static class Output {
    public char tristatePin;
    public boolean testout;
  }

  public final Input input = new Input();
  public final Output output = new Output();

  ... some access operations

  public class TestSignals extends TestSignalRecorder {
    ... }
}
```

This is the template for an IO pin class. The reset()[} step(int time){} and update(){} operations should formally contained here also, but can be left empty. It is nothing todo with the pins, also for Java-Test. The testSignals can be filled for the Java test.

The access operations are the same concept as in modules, it helps also to exchange modules in the system, using the interface concept especially with anonymous interface implementations, see also 3.2.4.4 Interface agents or access instances

second page

```
ENTITY FpgaTop_SpeA IS
PORT (
  clk: IN BIT;                                    (1)
```

(1) The VHDL translator defines automatically a signal clk as BIT. It is used in the VHDL processes as `PROCESS ( clk ) … BEGIN IF(clk'event AND clK='1') THEN …` . The clk itself is not used in Java, because it is the calculation clock between all `step(time)` and `update()`.

4.8. Java sources for Modules

See also chapter 5.3 A module file.

A Java source of a module should contain all the parts in the following sub chapters. Whereby the `@Fpga.LINK_VHDL_MODULE` inner class for included VHDL module (chapter4.8.3. Included VHDL modules page 60) is of course optional, only if such is necessary. Also the number of `@Fpga.VHDL_PROCESS` inner process classes (chapter 4.8.2. Inner class for records and process, page 58) can be vary, maybe omitted if the module has not states, only logical relations.

Last not least the number of @Fpga.IfcAccess anonymous interface implementations (chapter 4.8.5 Interface access agents in Modules page 63) depends on the necessary access agents from other modules.

4.8.1. Connections and inner modules

2. Java: class for a FPGA module, references and sub modules

```
public final class ModuleXY implements FpgaModule_ifc { //                    (1)
  public static class In {                                                    (2)
    boolean in1;
    @BITVECTOR(8) inVal;
  }
  public final In in;
  public static class Out {                                                   (3)
    boolean out1;
    @STDVECTOR(8) outVal;
  }
  public final Out out;
  //
  static class Ref {  // This class contains references to other modules      (4)
    Reset_ifc reset;
    OtherModule_ifc mdlXx;

    Ref ( Reset_ifc reset, OtherModule_ifc mdlXx ){                             (5)
      this.reset = reset; this.mdlXx = mdlXx;
    }
  }
  private Ref ref;                                                            (6)
  //
  static class Modules {        // This class contains sub modules            (7)
    final ModuleXY moduleXy = new ModuleXy();                                 (8)
    Modules ( Ref ref, MyModule thism ) {                                     (9)
      this.moduleXy.init(ref.reset);
    }
  }
  Modules modules;                                                            (10)
```

(1) This is a module with sub modules and references.

(2) A `class In` is possible, but often not recommended. It contains signals which should be set from outer assignments. The `class In` builds a `TYPE MyModule_In_REC IS RECORD` and per module instance a `SIGNAL module_In : MyModule_In_REC` in the VHDL file, which's elements are set in other modules. The usage of `class In` is unnecessary if consequently the Object Oriented approach is used, see `class Ref`.

(3) A `class Out` is possible, adequate to class In, to access output values immediately. It builds also an own `TYPE MyModule_Out_REC IS RECORD` and an instance per module. If one VHDL file for the module is generated, it is the interface of this VHDL module generated adequate to the top level.

If the module is used flattened in the enclosing module, then specific Records for this `In` and `Out` inner classes are created. The values are gotten and taken there. But the enclosing

module is responsible to set content to In and put content from out.

(4) The `class Ref` is usual necessary, for the idea of references to other modules usual known in ObjectOrientation programming. This references are *aggregations* in UML slang. Aggregations are relations to other modules given on startup, that is proper.

The name `Ref` need ot be used for this role, detect by Java2VHDL. The references to other modules can be especially the interfaces, not the instance types. Especially different signals or signal groups as access to another module can be references by the concept of chapter: 3.4.4. Interface technology in Java for VHDL page 23. It means connections (single signals or signal groups) are used to connect the module, not referencing any other module as a whole. This increases the number of connections, but also the flexibility.

For the implementation in VHDL the references are translated to direct accesses to the `RECORD` signals from other modules flattened in FPGA. For the implementation level that are simple accesses to other FlipFlops etc. It means, this Object Orientated references are dissolved first in the flattend VHDL, and second by the implementation. But from view point of modularity in the Java sources they are flexible. The references are connected outside on instantiation of the module, see chapter 4.6. Java source for top level FPGA class page 52

(5) The constructor of `Ref` gets all references to the real implementing modules. It is called either on the `init(...)` of on the construction of the module, see 4.8.3 Constructor and init for a module page 62. The names of arguments for the `Ref(…, argument, …)` have to be the same as the names of the references `variables` itself in the `class Ref{ ... }`. Also the same names used for the module constructor arguments and for the module's `init(… arguments)` operation needs to be exactly the same as in the `class Ref{ … }`. The Java2VHDL translator gets this names in the argument lists and fulfills it with the accesses given in call of the module. Inside the inner classes for `@Fpga.VHDL_PROCESS .. class` (next chapter 4.8.2. Inner class for records and process) the names of the access to `ref.name...` are used. This identifier names are used as key to search the access in a Map (index), to fulfill it with the expression which is given on initialization of a module. It is very simple. Use the same names. Note that the argument name, for this example `mdlXx` is well distinguish from the class-(instance-) variable with the same name `mdlXx` which is addressed using the `this.reference`.

(6) The instance of `ref` and also `modules` `(10)` is set in construction of the module , see chapter 4.8.3 Constructor and init for a module page 62 (1) .. (4).

(7) The `class Modules` is only necessary if the module has sub modules. The identifier `Modules` and `modules` should be written as shown here, detect by Java2VHDL. Note: In UML slang the sub modules are *compositions*.

(8) Inside the `Modules` class any other modules can be defined and initialized as sub modules of this modules. This sub modules are defined in extra module classes of course. Initializing is possible either with the constructor of the sub module called in the constructor of `Modules` or see (10).

(9) The constructor of the `Modules` for the sub modules is called in the constructor of this module class, see 4.8.3 Constructor and init for a module page 62. Initializing of the sub modules is also possible via the `init(…)` operation.

(10) The `modules` are initialized in the ctor of this module, see 4.8.3 Constructor and init for a module page 62). Other as initializing in the top level (see 4.6. Java source for top level FPGA class page 52) the initializing of the sub modules may need references which are given as arguments of the module's constructor from other modules outside (as direct associations from sub modules to outside). May be there are the same as also stored in `Ref ref` of this module.

4.8.2. Inner class for records and process

3. Java: class for a FPGA module, PROCESS classes

```
@Fpga.VHDL_PROCESS public static class Q_CE0 {              // (1)
  public final CeTime_ifc time_;                            // (2)
  final boolean var1;                                       // (3)
  @Fpga.STDVECTOR(16) public final int val;
  Q_CE0() {                                                 // (4)
    this.time_ = null;
    this.var1 = false;
    this.val = 0x0000;
  }
  @Fpga.VHDL_PROCESS  Q_CE0(int time, Q_CE0 z, Ref ref, ModuleXY thism) {    // (5)
    this.time_ = thism.srcCE0;                              // (6)
    if(thism.srcCE0.ce()) {                                 // (7)
      if(ref.reset.res(time, 1)){                           // (8)
        this.var1 = false;
        this.val = 0x1234;                                  // (9)
      } else {
        this.val = z.val + 1;
        this.var1 = (z.val == 0x0010);
      }
    } else throw new IllegalStateException();               // (10)
  }
}
protected Q_CE0 q_CE0 = new Q_CE0();                        // (11)
private Q_CE0 q_CE0_d = q_CE0;                              // (12)
```

See also example in chapter 5.3.3 Inner static classes in a module which builds a TYPE RECORD and PROCESS in VHDL

(1) A PROCESS class is a inner @Fpga.VHDL_PROCESS static class, here named only '[J]Q' for the first class of the module building Q outputs of Flipflops, usable also a more comprehensive name. It describes a PROCESS in VHDL and the associated RECORD variables.

Note that the inner class for the Process should be static. It means this inner class is independent of the environment, the module class. To access the environment class an argument thism should be used, see (5).

(2) The variable `time_` won't be used immediately for Java2VHDL translation. It is for generation and checking timing constraints, see 4.12.2 Association between PROCESS variables and time GROUPs page 88. It refers a `CeTime_ifc` instance, or `null` after initialization.

(3) Some variables are defined (should be `final`) which builds in VHDL a member of the `TYPE MyModule_Q_REC`

(4) The default constructor is only for Java test. It should initialize all variables as the hardware in the FPGA does, usual with 0 (hardware reset).

(5) The parameterized constructor marked with `@Fpga.VHDL_PROCESS` should be called in the `step(…)` routine and describes (is translated) to a `PROCESS` in VHDL. As seen from the hardware functionality, it describes how the D-inputs (`this…`) are calculated as combinatorics from the Q outputs of the own Flipflops (`z….`) and from other signals (`ref…` , `thism…` , `modules...`).

Dn+1 = fn(Qn, Inputs)

The constructor of the process can/should get the following arguments, the names are obligate, the order in the argument list is free, but `time` and `z` should be named first. These names are recognized during VHDL translation and used for access within the statements.

- `time`: The currently central time of this step for timing assertions.

`z`: This is the current state (or the state before) of this process variables Qn. z is as in Z-Transformation controlling theory.

`thism`: The reference to the own module, the environment class instance, to access immediately states from other RECORD variables from the other processes of the own module

`ref`: Access to the Ref ref class for access to other modules (aggregated).

`modules`: Access to the own sub modules defined in the Modules inner class.

`(6)` Assignment of any implementation of the `CeTime_if` determines the association of the time `GROUP` which is delivered from `CeTime_if#timeGroupName()` , and allows access to the last time of this time Group. This assignment is not used for generating the VHDL file, but for generating of the constraint file

`(7)` This is a typical condition where a '*clock enable*' signal is used, here from another process in the same module referenced via `thism.srcCE0` It is adequate translated to VHDL and adequate implemented (using the CE input of Flipflops) in the hardware.Because it is the first level `if(…)`, it is related to the whole `PROCESS` functionality. It has two tasks:

- The resulting access value is generated in the VHDL code. It is here `IF (mdl1_PCE.ce)='1' THEN`. Hence it is the common *clock enable* in this process.
- For timing constraints it clarifies that this `SIGNAL … RECORD` is member of the same time `GROUP` as the accessed `CeTime_ifc`. See 4.12 Checking time between FF groups page 86.

`(8)` Constructions as `if(…` are translated to VHDL with given rules, see chapter 4.10 Statements in Java and their translation to VHDL page .70

`(9)` All variables of the own class, in VHDL it is the built `TYPE … RECORD`, should be set in any branch. The `final` declaration of the variables helps. You cannot forget or set a variable twice.

Note: You need use `this.` to mark the variables as class (instance) variables of the own class. In Java (as also in C/++) normally writing `this.` is optional, can be omitted, but may be seen as recommended. But for the Java2VHDL translation it is necessary. Note that in the early years of enthusiasm for the new class-oriented operations it was recommended to leave the `thiz→` designations for variables defined on class level. But meanwhile auto completion and "*be explicitely*" is more important.

`(10)` This `else` branch is the *do nothing* branch. In VHDL it is omitted. But in Java it must be present, because the variable value should be set, of course from the state before (from the Q output of the own Flipflops) writing `this.var1 = z.var1;`. That is the here not shown possibility. Here it is presumed that in the step(…) operation the constructor is only called in condition of `thism.srcCE0.ce()`. It saves calculation time. Hence the else branch is never entered. That is tested, instead set of all variables with there `z…` values. That saves writing effort.

`(11)` The **name** of the instance should be exact the **same as the PROCESS class name**, only written with **lower case as first character**. The instance of this class presents the `SIGNAL … _REC` in VHDL. It can be accessed also from other processes in this module, from test preparation and for any tests and in debugging. This instance may be `public` for a more simple access, but should be intrinsically `protected`. For access, from other modules it is preferred using interface accesses

`(12)` The `…_d` instance presents the D-inputs from Flipflops, see following `step(…)` and `update()`., see 4.8.4 reset, step, update and output in a module page 63. It should be any time private. The reset-initialization should follow the `(11)` as shown in pattern.

4.8.3. Included VHDL modules

There are two reasons to include a given VHDL file in the generated sources:

• a) Reuse existing VHDL-defined modules

• b) Often the tools for FPGA design have features to generate VHDL files for specific features, for example for a RAM block.

For simulation on Java level, the functionality of the VHDL code should be emulated. How to write the emulation for a given VHDL module: See 4.9 Java source for an emulated VHDL module page 68.

4. VHDL module mentioned in Modules

```
static class Modules {        // This class contains sub modules
  final VhdlModuleXy moduleXy = new VhdlModuleXy();                    (1)
  Modules ( ... ) { … }
}
Modules modules;
```

(1) First, the included VHDL module must exist in the inner modules class, see 4.8.1. Connections and inner modules, page 56. This is also valid for the Top level Java class 4.6. Java source for top level FPGA class 52: This module has no references to connect. The name is part of the instance name of the VHDL module in this module (name of the `PORT MAP`). The type is the name of the corresponding Java class to emulate this module, see 4.9 Java source for an emulated VHDL module page 68.

5. Inner class describes the inclusion for a VHDL module (PORT MAP)

```
@Fpga.LINK_VHDL_MODULE private static final class Vhdlink_moduleXY {    (2)
  boolean final int inDataXy;                                           (3)
  @Fpga.STDVECTOR(7) final int outDataXy;
  Vhdlink_moduleXY() {                                                  (4)
    this.inDataXy = false;
    this.outDataXy = 0;
  }
  @Fpga.LINK_VHDL_MODULE Vhdlink_moduleXY ( int time                    (5)
      , VhdlModuleXy vhdlMdl                                            (6)
      , Ref ref, Modules modules, MyModule thism) {                     (7)
    this.inDataXy = ref.refXy.signal && thism.processXy.signal;         (8)
    vhdlMdl.input.Clock = Fpga.clk;                                     (9)
    vhdlMdl.input.inpXz = thism.processXy.vectorX2;                     (10)
    vhdlMdl.input.inDataX9 = inDataXy;                                  (11)
    vhdlMdl.step(time)                                                  (12)
    vhdlMdl.update();
    vhdlMdl.output.outDataXy = vhdlMdl,output.outDataX5;                (13)
  }
}
```

(2) Any instance of a included VHDL module needs such an inner class (in sub modules, also in the top level class file). It must be marked with `@Fpga.LINK_VHDL_MODULE`. The name of this inner class should be start with `Vhdlik_`. Then it is continued exact with the name of the instance in Modules (1). This is the coherence for the VHDL translation.

If you have more as one `PORT MAP` with the same included VHDL file, this should be done more as one time. This inner class is the adequate for the VHDL `PORT MAP`.

(3) For any output signal of your VHDL module (there in the `ENTITY` definition) and for all input signals which are not given as simple signal you need an inner final variable in your inner class. After Java2VHDL this variables builds a `TYPE MyModule_Vhdlink_moduleX_REC IS RECORD ...`.with a `SIGNAL myModule_Vhdlink_moduleX : MyModule_Vhdlink:moduleX_REC;`

(4) The default constructor in Java is only necessary for Java compilation and test.

(5) The efffective constructor is marked with `@Fpga.LINK_VHDL_MODULE`. As also for inner Process classes it has a `int time` argument.

(6) The second argument have to be named `vhdlMdl`, detected by Java2VHDL. The type of this argument, here `VhdlModuleXy` is the same type as the vhdl module in (1). It is the Java class name for the emulation class of the VHDL module, see 4.9 Java source for an emulated VHDL module page 68.

(7) All other arguments are adequate PROCESS inner classes. You may need the `Ref ref` for access to other modules, the `Modules modules` to access to other inner modules, and `thism` to access to other RECORDs of this environment module, or maybe it is the top level class.

(8) Now, in the implementation of the constructor, first the intermediate variables should be assigned with any expression accessing the environment.

(9) A clock input of the VHDL PORT MAP should be connected. The `Fpga.clk` is the access to use the signal `clk`, automatically defined in the `ENTITY` of the Top level FPGA.

(10) .The inputs of the VHDL module should be set. This is for the test execution in Java, as well as for the `PORT_MAP` assignment in VHDL.

(11) If an input port can be assign with a simple assignment of a variable, it should be done without an additional intermediate variable.

(12) `step(time);` and `update();` are especially for Java test. But it should be written between input settings and output getting.

(13) The assignment from the output to the output variable of this RECORD builds the `PORT_MAP` assignment in VHDL for the output variable.

The value of the output can now gotten from the SIGNAL of the RECORD of (3).

6. VHDL output for this template of inclusion for a VHDL module (PORT_MAP):

```
TYPE MyModule_Vhdlink_moduleX_REC IS RECORD
  inDataXy : BIT;
  outDataXy : STD_LOGIC_VECTOR(6 DOWNTO 0);
END RECORD MyModule_Vhdlink_moduleX_REC;                          (14)
…

SIGNAL myModule_Vhdlink_moduleX : MyModule_Vhdlink:moduleX_REC;   (15)
…

-- The external VHDL file RAM_SpiRamSel is included here.
-- Assignments for VHDL instance inputs:
myModule_Vhdlink_moduleX.inDataXy <= otherModuleXy.signal AND …   (16)

myModule_Vhdlink_moduleXy_vhdlMdl: VhdlModuleXy                   (17)
PORT MAP(
  Clock =>  TO_STDULOGIC(clk) ,
  inpXz =>  TO_STDLOGICVECTOR(myModule_processXy.vectorX2) ,
  inDataX9 =>  myModule_Vhdlink_moduleX.InDataXy ,
  outDataX5 =>  myModule_Vhdlink_moduleX.outDataXy
 );  --PORT MAP VhdlModuleXy
```

(14), (15) Definition of necessary intermediate variables in RECORD

(16) Assignment of the input variable(s)

(17) PORT MAP definition of the included module.

4.8.3 Constructor and init for a module

Some of this content is relevant for the Java2VHDL translation, other only for the test in Java. Follow the explanation.

7. .Java: class for a FPGA module, constructor and init

```
public MyModule ( Reset_ifc reset, OtherModule_ifc mdlXx ) {                    <1>
  this.ref = new Ref(reset, mdlXx);                                             <2>
  this.modules = new Modules( ref, this);
}

public MyModule ( ) {}        // use init to initialize                         <3>

public void init ( Reset_ifc reset, OtherModule_ifc mdlXx ) {                   <4>
  this.ref = new Ref(reset, mdlXx);
  this.modules = new Modules( ref, this);
}
```

<1> There are two variants of construction. <3> and <4> shows the other one. This constructor gets all references to other modules. They are **aggregations** as explained in . ructor of the module in Java can get all references, and

<2> should initialize the ref and also sub modules modules if existing.

<3> Or there is a argument less ctor, paired with the

<4> `init(...)` operation. This variant is necessary to use if the order of modules is not exactly a tree, but the modules have their specific cross connections. Both is the same for the generated VHDL file, or also for Java, it is a known problem on initializing order.

It may be seen as recommended to try initializing with the constructor, where the modules are in a tree organization, and only use the `init(...)` for some cross depending modules. But it is also proper to use generally `init(...)`. That is a common discussion of modularity, not a topic of Java2VHDL.

4.8.4 reset, step, update and output in a module

8. Template for reset, step, update, output in a module

```
public void reset ( ) { // call of the empty ctor for all process inner classes
  this.q = this.q_d = new Q();     // to set hardware reset values              <5>
  this.modules.xy.reset();
}

public void input ( ) { // optional sets input records of sub modules           <6>
  this.modules.xy.in.var = ref.mdlXx.q.var;
}

public void step ( int time ) {    // calculates the D-states (pre states)     <7>
  this.pCE_d = new PCE(time, this.pCE, this.ref);
  if(this.srcCE0.ce()) {
    this.q_CE0_d = new Q_CE0(time, this.q_CE0, this.ref, this);
  }
  if(this.srcCE0.ce()) {
    this.val_CE7_d = new Val_CE7(time, this.val_CE7, this.ref, this);
  }
  this.modules.submdlXz.step(time);
}

public void update ( ) {           // Activates the Q-states (Flipflop outputs)
  this.q = this.q_d;               // this is the clock edge in hardware       <8>
  this.modules.xy.update();
}

public void output ( ) {   // optional, but necessary for top level           <9>
  this.out = this.q.var && this.modules.xy.out.var;
}
```

<5> The `reset()` operation should be set the state of the FPGA as it is after power on, especially after repeated tests. Do not base on states before! This is done simply by calling all the standard constructors of the processes, which should be programmed to force the power-on-reset states. The content of `reset()` is not relevant for Java2VHDL translation, but for Java-test. Do also initialize the `…_d` reference!

<6> The `input()` operation is only necessary if any sub module has an `In` sub class that should be set with information from the environment (from `ref`). The content of `input()`, all assignments, are evaluated by the Java2VHDL translator.

<7> The `step(int time)` operation should call all constructors of the processes in the shown kind. Also (as in all other operations too `step()` of sub modules should be called. The content of `step()` is not relevant for Java2VHDL translation. It should be written in the unique kind to ensure tests.

<8> The `update()` should be also written in the unique kind for tests, not relevant for the Java2VHDL translation.

<9> The content of `output()`, all assignments are relevant for Java2VHDL translation. It should get the values from processes which should be placed manually to outputs, especially in the top level class for the output pins of the FPGA. Generally the processes (`step(int time)` and `update()` informations from inputs are taken usual via `ref`, but only the own state is set. It means the `output()` is really relevant for the output pins.

4.8.5 Interface access agents in Modules

An interface access point is an anonymous interface implementation in Java which accesses

internal data in the module in any process for data access interfacing to other modules.

9. Example for an interface access point

```
@Fpga.IfcAccess Bit_ifc getValxy = new Bit_ifc() {                    //(1)
  @Override public boolean getBit () {                               //(2)
    return ModuleXY.this.q.var1;                                     //(3)
  }
};
```

(1) The `@Fpga.IfcAccess` is necessary as marker for this interface to detect for java2Vhdl.

The type of this anonymous interface implementation is the type of the necessary interface, here `Bit_ifc` which is defined in `import org.vishia.fpga.stdmodules;`

The name of the implementation, here `getValuexy`, is the access name and should be used as actual argument of any other module's constructor or `init(…)` which needs this access as aggregation.

The `new Bit_ifc() {` is necessary in Java syntax to create the instance.

(2) The `getBit()` operation is defined in the interface, hence it have to be implemented here.

(3) The implementation should (only) contain `return` following an expression. All other statements are ignored for Java2Vhdl. Especially it is possible to write either specific statements for test or als to check the timing constrains, see Error: Reference source not found page Error: Reference source not found.

The expression after `return` is evaluated as access to the appropriate data element.

Writing `ModuleXyY.this.` is necessary. In Java commonly `this.` can be omitted because the Java translator detects automatically the variable defined in the environment class. But writing the qualified access may be also recommended for normal Java programming. For the Java2Vhdl translator it is necessary.

`q` is the instance name of a process inner class in this module, which builds a RECORD in VHDL with the name `MyModule_Q_REC` (with the type Q) and a `SIGNAL mdl1_Q : ModuleXY_Q_REC`; with the given module instance, here `mdl1`.

In a calling environment, here written as `this.modules.mdl1.getValxy.getBit()` in the top level class, it produces the code `mdl1_Q.var1`, it is resolved to the simple access to the variable `var1` in the given instance `mdl1_Q`.

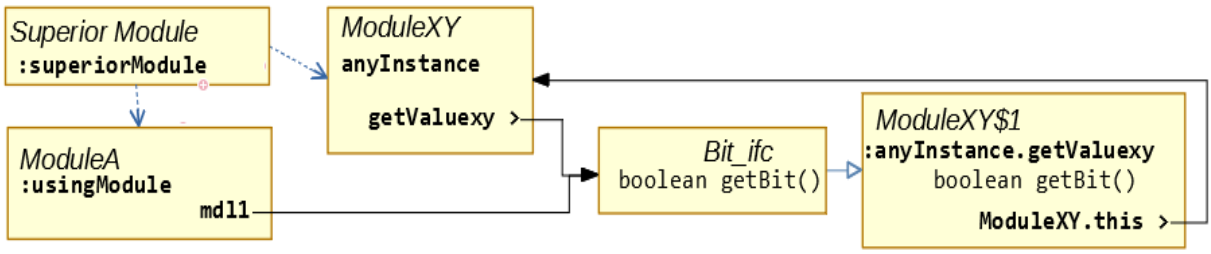


Figure 4: Interface Access UML diagram

The elaborately diagram shows the relation in UML from view of the Java code:

- The aggregation `mdl1` inside the using `ModuleA` refers the interface access instance of `ModuleXY$1` via the given `Bit_ifc` type. This instance was created before inside the `ModuleXY`.
- The not named but referenced instance of `ModuleXY$1` (in heap), which implements the `Bit_ifc`, refers its environment class `ModuleXY` with the `ModuleXY.this` pointer. Hence it knows the whole instance of `ModuleXY` and can access all members, also private ones because it is an inner class. Hence it can return in its implementation a value inside `ModuleXY`,

which is here `q.var1`.

- In the initialization phase the constructor of of the top level or the superior module constructs first the `ModuleXY` instance together with the instance of its inner class `ModuleXY$1`. Then, with knowledge of this instance (dependency, dotted line), it delivers the reference to the proper `ModuleXY$1` to the constructor of the using `ModuleA`. That constructor sets the `mdl1` reference, hence knows the implemenation.

That are the full cohesion in Java shown with UML approaches. It is a little bit sophisticated for practice.

A more simple short diagram shows the same cohesion for practical use:

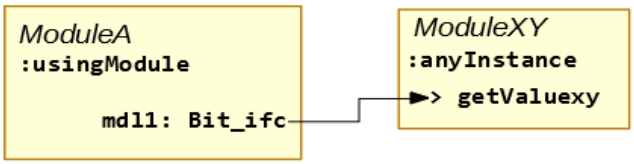


Figure 5: Interface Access Block diagram

The type of the interface is noted with the reference `mdl1: Bit_ifc`, after colon. The aggregation goes to the variable `getValuexy` inside the class `ModuleXY`. Because the aggregation line is not connected to the class, it is connected to the inner reference `getValuexy`, the type of the class `ModuleXY` should not be known. It should only be known by wiring the aggregations in the initialization phase, which is shown in the UML figure above with the dotted line of dependency. Here the dependency should not be elaborately shown, it is clarified by showing the target from `mdl1` inside the named type and instance.

Note that this is a diagram type, which is not defined in the UML, but more simple. It comes from practice of Function Block programming.

4.8.6 Implementation of module interfaces

Not mentioned in chapter 4.8.1. Connections and inner modules, page 56, the module itself can implement another interface for access. It is similar of the interface access agents, but associated to the whole module. Then the module should only implement the requested interface operations. The implementation should be only marked with the recommended `@override` annotation. That is sufficient for the Java2Vhdl translator.

The implementation of interfaces of the whole module assumes, that the interface is unique for the module. It is not possible to implement the same interface type, for example the basically Bit_ifc, more as one. That's why it is limited. The interface access agents are the more universal approach.

10. Any specific interface for a module

```
public interface Specific_ifc {
  @Fpga.BITVECTOR(16) int cmd ( );
  boolean cmdMaster ( );
}
```

11. Implementation of the specific interface inside the module

```
class MyModule implements FpgaModule_ifc, Specific_ifc {
  …..
  @override @Fpga.BITVECTOR(16) int cmd ( ) { return this.process.valx; }
  …
```

Similar as in the chapter above described, only the expression after return delivers the result.

4.8.6. TestSignalRecorder in a module for Java based test

Each module may contribute to test outputs. The test outputs of the module is included automatically, if the next shown inner classes are contained. It means for different test approaches this content inside a module should be changed. But currently changing a module is not the best approach for development of a whole component. That's why, it's better to include a flexibility (amount of test outputs, select specific signals) to reuse the module inclusively this test output in different situations.

It may be also possible to access specific inner variables of the module immediately from outside written test operations. That is possible if the inner PROCESS classes of the module are `public`. This is a worse organization because modularity is disregarded. But it is a workaround to fast output till now non determined variables. The better and following work to do is, think about how it is possible to do in a more universal form as intrinsic test access property of the module, for the future.

A module class can contain:

12. Java: class for a FPGA module, Test support

```
public class TestSignals extends TestSignalRecorder {    // (1)
  StringBuilder sbxy = new StringBuilder();               // (2)

  public TestSignals(String moduleName) {                 // (3)
    super(moduleName);
  }

  @Override public void registerLines ( ) {               // (4)
    super.clean();
    registerLine(this.sbxy, "xy"); // for this signal(s)
  }

  @Override public int addSignals ( int time, int lenCurr, boolean bAdd )  (5)
  throws IOException {
    MyModule thism = MyModule.this;   // only for simple access
    this.sbxy.append(thism.q.var1 ? '1' : '_');                          // (6)
    return this.sbxy.length();                                           // (7)
  }

  @Override public void endSignals ( int lenCurr) throws IOException {  // (8)
    this.pos = lenCurr;
    if(this.sbs !=null) {
    for(StringBuilder sb : this.sbs) {
      while(sb.length() < lenCurr) { sb.append(' '); }
    } }
  }

}
```

(1) This is a non static class, it should access to the environment class. The org.vishia.fpga.testutil.TestSignalRecorder is the base class for entries in the test signal outputs using the org.vishia.fpga.testutil.TestSignalRecorderSet. The organization of the test output is described in chapter 4.10.3 The TestSignalRecorderSet to record test signals from modules

(2) Each test signal output line is accumulated in a StringBuilder instance, need to be intialized here. Of course you can have more as one line per module, it means more as one StringBuilder.

(3) The constructor for this class is called as described in 4.10.3 The TestSignalRecorderSet to record test signals from modules. The `moduleName` is given from outside, because it is possible to have more instances of this module. The constructor can have more arguments for flexiblity of the test signal recording, choice of test signals etc.

(4) The modul's `TestSignalRecorder` should register its line in the output if given signal names. The line is designated in the output with the `moduleName_signalName`. Note that because of more arguments on the constructor **<3>** it is possible to adapt the registrations of lines.

(5) This operation **`addSignals`** is called from the **`TestSignalRecorderSet#addSignals(time)`**, which is called outside, see 4.11.3. The TestSignalRecorderSet to record test signals from modules 83 The arguments are

* **`time`**: the current simulation time step, possible to decide about output.
* **`lenCurr`**: The numbers of character in the other StringBuilder till now. For example it is possible to add a longer string if another module has also add a longer string for this test. It is necessary to have the same length of all lines to hold it synchronous in the column meaning to the time and to all other signals. It is recommended to add characters to get the same length. The length is not influenced by the common organization. So it is possible to save space to use it for next steps. But all in all, the positions in all StringBuilders should be aligned.
* **`bAdd`** this is true if any StringBuilder before has added somewhat. It means also that the length of the own StringBuilder should be lesser than `lenCurr`, but it can be use simple as signal to add also. It is possible that modules are related in their behavior. For example register first a module which decides to add or not (for example using a specific signal as event for recording), and all other follow.

(6) You can also add more signals in only one line using different characters. Then it is able to distinguish. It is also possible to add any hex value for a vector or such, be carefully for the length.

(7) The return value should be `0` if nothing is added, but should be the length of the longest own StringBuilder (all may have the equal length) if anything was added.

(8) This operation is optional, the default implementation is shown. You can decide add some more information, if any of `TestSignalRecorder` in the other modules has added more. This opeation is called after all `addSignals(…)` of all modules are called.

4.9 Java source for an emulated VHDL module

There are two reasons to include a given VHDL file in the generated sources:

• a) Reuse existing VHDL-defined modules

• b) Often the tools for FPGA design have features to generate VHDL files for specific features, for example for a RAM block.

For simulation on Java level, the functionality of the VHDL code should be emulated. For example for a RAM module this may be simple. Also for more sophisticated behavior of the original VHDL module you may only simulate a basically functionality to fulfill the interface to the module.

13. Template for the Java class definition of an emulated VHDL module:

```
@Fpga.VHDL_MODULE ( vhdlEntity = "ModuleXy" ) public class VhdlModuleXy     (1)
  implements FpgaModule_ifc {

  public static class Input {                                               (2)
    public @Fpga.STD_LOGIC boolean Clock;                                   (3)
    public @Fpga.STD_LOGIC boolean ClockEn;
    public @Fpga.STD_LOGIC boolean Reset;
    public @Fpga.STDVECTOR(7) int inDataXz;
    public @Fpga.STD_LOGIC boolean inDataX9;
  }
  public static class Output {
    public @Fpga.STDVECTOR(8) int outDataXy;
  }
  public Input input = new Input();
  public Output output = new Output();

  /**Example, RAM with 9 addresses, 8 data emulated ... */
  byte[] content = new byte[512];                                           (4)

  @Override public void reset ( ) {                                         (5)
    //todo may clean the RAM
  }

  @Override public void step ( int time ) {
    ...
  }

  @Override public void update () {
    this.output.outDataXy = this.content[this.input.inDataXz & 0x7f] & 0xff;
  }
}
```

(1) The class should marked with the annotation `@Fpga.VHDL_MODULE (...)` to detect, it is specific to translate. The given argument value after `vhdlEntity` is the file name of the VHDL module. It may be similar or the same as the java class/file name.

(2) This class should have an inner class `Input` as well as `Output`. This builds the interface to the VHDL module generated in the `COMPONENT … PORT` definition. The same rules as for 4.7. Java source for Pin definition FPGA class page 54 are valid. `INOUT` signals should be defined with the same name in both `Input` and `Output` classes. For Java they are different variables. For VHDL they can be get anyway, and set in a specific way usual with tristate properties using a `char` in Java.

This Input and Output class elements should exactly follow the `ENTITY … PORT ( … )` definition in the given VHDL module, in names, types and the order.

(3) The VHDL module may have of course any clock input which should be wired usual with `Fpga.clk`, see 4.8.3. Included VHDL modules page 60. Also any clock enable and reset should be existing in the VHDL module which should be wired.

(4) The class now may have logic to emulate. The template shows here only, how simple it is to emulate a RAM.

(5) The **reset()**, **step(time)** and **update()** are only for Java execution (emulation). The **update()** is immediately called after **step**, hence it is not important whether the code is written in **step()** or **update()**. The output values are stored in the execution of the calling environment as prepared values (D-Inputs) in the **xyz_d** reference, so that correct clocked usage is met.

The original VHDL file may have its head information as following:

```
entity ModuleXy is
    port (
        Clock: in  std_logic;
        ClockEn: in  std_logic;
        Reset: in  std_logic;
        inDataXz: in  std_logic_vector(6 downto 0);
        inDazaX9: in std_logic;
        outDataXy: out  std_logic_vector(7 downto 0));
end ModuleXy;
```

4.10 Statements in Java and their translation to VHDL

Generally the statements for Java2VHDL are only located in the constructor of a `@Fpga.VHDL_PROCESS static final class MyProcess {…}`. and in the `input()` and `output()` operations of a module and the top level.

4.10.1 Variable definitions

Firstly regard differences "*what is a variable*" in sequential languages and in VHDL. In sequential languages of course any variable is a storage in memory located either in a class (in heap) or in the stack for temporary values.

In VHDL a variable which is set inside a `PROCESS` is represented by one or some FlipFlops, stored similar as in sequential languages. But the usual known difference is: The variable is not set immediately after the assignment, the new value is only valid after the clock, this is on the next entry to the process. To considered this fact,:you should never use `this.value` inside an expression (on right side). This is yet not checked by Java2VHDL translation, should be done (RFC1).

In VHDL a variable which is set outside of a `PROCESS` has no representation in the FPGA. It is a placeholder or such as a macro in C language (`#define xy expression`). This is also true for locale variables defined in a process.

A variable definition can be done:

- As final variable in a `@Fpga.VHDL_PROCESS public static class` . This variable is represented in the FPGA with one or some FlipFlops.
- As local variable in the `@Fpga.VHDL_PROCESS` constructor body. This builds in VHDL a `VARIABLE` definition inside the VHDL `PROCESS` without representation in the FPGA, it is temporary.
- As variable inside an inner `class In` or `class Out`. This variables are generated to a `TYPE ... RECORD` definition and the appropriate `SIGNAL` definition. This assignment to this variables should be done only in the `input()` and `output()` operations in Java. In the FPGA there are not presented, it is temporary.
- As variable inside `class Input` or `class Output` of the Pin description class file. This variables builds the `ENTITY … PORT` definition of the VHDL file.

Variables on class level are not regarded for Java2VHDL. They can be defined for additional calculations for tests. Hence, variables outside of a `TYPE ... RECORD` (simple variables in VHDL) are never generated.

The Java variable type determines the VHDL type:

```
boolean bitValue;                          (1)
char stdValue                              (2)
@Fpga.STD_LOGIC boolean stdValue2;         (3)
@Fpga.BITVECTOR(16) int bitVector;         (4)
@Fpga.STDVECTOR(7) int stdVector;          (5)
@Fpga.BITVECTOR(5) Enumtype enumval;       (6)
```

```
bitValue: BIT;
stdValue : STD_LOGIC;
stdValue2 : STD_LOGIC;
bitVector : BIT_VECTOR(15 DOWNTO 0);
stdVector : STD_LOGIC_VECTOR(6 DOWNTO 0);
enumVal : BIT_VECTOR(4 DOWNTO 0);
```

(1) A simple boolean with the values true and false is always a simple BIT variable.

(2) A STD_LOGIC variable which needs also specific states as Tristate are represented in Java with a char. The value of the char should be `'0'` `'1'` or `'Z'`, The other values of STD_LOGIC are non presented in Java2VHDL, because only a functional simulation is done. The other values are important in VHDL for timing simulation.

(3) boolean can also be translated to STD_LOGIC, but then only the values `'0'` `'1'` are usable.

(4) A BIT_VECTOR is presented by an int with at least 32 bit or a long with 64 bit. But only bits

starting right side with 0 are supported also in VHDL DOWNTO 0. In Java the int value is not automatically mask if you have overflows on addition or shift operations. You should mask if necessary in an expression with a value starting with m_ (“mask”). This operation is not translated to VHDL, not necessary. Using specific operations for shift and select bits do the necessary mask, so that the int value is always clean for test in Java (follows the VHDL behavior).

(5) It is the same for `STD_LOGIC_VECTOR`. Note that some operations can be done in VHDL only with `STD_LOGIC_VECTOR`. Hence you should use it. On assignment and access also an automatic conversion is done.

(6) An enum value in Java is interesting especially for state machines. See todo.

4.10.2 Assignments

Assignments which are translated to assignments in VHDL are contained in:

- the `@Fpga.VHDL_PROCESS` constructor body. There you should only assign the own inner class variable written with `this.name =` or the local defined variables. You shoud never assign other variables than the own class variables. This follows the rule in VHDL: Each variabe which is assigned in a `PROCESS` should not assigned any where other again. The rule, only assign the own class variable, helps for clarity. The variables which are handled in one `PROCESS` are combined in one `RECORD` in VHDL.
- the `input()` operation of the Java module. There values to an `In in` instance in a sub module should be set, typically written as `this.modules.subModuleXy.in.variableXz = …`. It is the non referencing variant for value connection). The destination for this assignment in VHDL is the appropriate `SIGNAL … RECORD` variable. They are not presented in the FPGA, it is temporary.
- the `output()` operations of the Java module. The destination variable should be located either in the own `Out out` instance for the non referencing variant for value connection. Or it can be also located in the `Output output` instance of the `ioPins` module of the top level class. That are immediately the output pins of the FPGA. This is true especially for the top level class as well as also in module classes. Inner module classes should know a reference to the `ioPins` in there `Ref` inner class (`ioPins` should be referenced). This should be done only for specific main modules of a FPGA, not commonly.

4.10.3 Expressions, Operations

Expressions are either the right side of an assignment statement, or it is used as boolean expression for conditions.

First considerate the type of an expression. It depends first on the type of the operands. But operators change the type, as known in all programming languages, also in VHDL, also in Java.

In VHDL especial it is distinguish between a BIT type of value, a STD_LOGIC or the boolean type which is used in conditions in VHDL. In Java all three are the same, it are the boolean type. Some adaptions are done, see next.

Operators and preference

In java there can be used:

For the example, it is set:

14. Java, Example Variables for expression:

```
  @Fpga.VHDL_PROCESS public static class Val {             // (1)
    final boolean bit1, bit2, bit3;                                  // (2)
    final @Fpga.STD_LOGIC boolean lg1, lg2, lg3;                        // (2)
    @Fpga.STDVECTOR(16) public final int val1, val2, val3;
    @Fpga.BITVECTOR(8) public final int bvec1, bvec2;
```

15. Simple boolean expressions in Java:

```
      this.out.out1 = this.val.bit1  & this.val.bit2;
      this.out.out3 = this.val.bit3 && this.val.bit2;
      this.out.out3 = this.val.lg1 && this.val.bit2;
      this.out.olg1 = this.val.lg1 && this.val.bit2;
      this.out.out1 = this.val.bit1 & this.val.bit2 | this.val.bit3;
```

16. The result after translation is:

```
mdl1_Out.out1 <=  mdl1_Val.bit1 AND mdl1_Val.bit2 ;
mdl1_Out.out3 <=  mdl1_Val.bit3 AND mdl1_Val.bit2 ;
mdl1_Out.out3 <= TO_BIT(mdl1_Val.lg1 AND TO_STDULOGIC(mdl1_Val.bit2) );
mdl1_Out.olg1 <=  mdl1_Val.lg1 AND TO_STDULOGIC(mdl1_Val.bit2) ;
mdl1_Out.out1 <=   (mdl1_Val.bit1 AND mdl1_Val.bit2 )  OR mdl1_Val.bit3 ;
```

- `&` and `&&` can both used for java `boolean`, it is `AND` in VHDL, same as for `|` AND `||`, it's `OR`.
- Conversion between `BIT` and `STD_LOGIC` are done in the expression.
- Preference is corrected by (). In VHDL `AND` is not preferred to `OR`, in Java `&` is preferred to `|`.

17. Comparison expressions in Java to set BIT and STD_LOGIC boolean:

```
this.out.out5 = this.val.bvec1 == 0x45;
this.out.out6 = this.val.val1 < this.val.val2
            && (this.val.bvec1 & this.val.val1) > 0x0005;
```

The result after translation is

```
mdl1_Out.out5  <=  '1' WHEN mdl1_Val.bvec1 = x"45"  ELSE '0';
mdl1_Out.out6  <=  '1' WHEN mdl1_Val.val1 < mdl1_Val.val2
  AND  ( ( (mdl1_Val.bvec1 AND TO_BITVECTOR(mdl1_Val.val1) ) )  > x"0005" )   ELSE '0';
```

- The comparison results in a boolean type in VHDL. That cannot assigned nor to `BIT` nor to `STD_VALUE`. Hence, a `WHEN .. ELSE` construct is translated to VHDL. Inside a PROCESS the `WHEN .. ELSE` is not admissible. For that a `IF .. ELSE` results after translation.
- `&` in Java with int variables which are different type are possible, an automatic conversion is resulted. Same is also for the other bit operations.

18. Numeric operations in Java 2 VHDL

```
      this.out.oVal1 = this.val.val1 + this.val.val2;
      this.out.oVal2 = this.val.val1 + this.val.bvec2;
      this.out.oVal3 = this.val.bvec1 - this.val.bvec2;                RFC2
```

```
mdl1_Out.oVal1 <=  mdl1_Val.val1 + mdl1_Val.val2 ;
mdl1_Out.oVal2 <=  mdl1_Val.val1 + TO_STDLOGICVECTOR(mdl1_Val.bvec2) ;
mdl1_Out.oVal3 <= TO_STDLOGICVECTOR(mdl1_Val.bvec1 - mdl1_Val.bvec2 );  ERROR
```

- \+ and – are admissible in VHDL only with `STD_LOGIC_VECTOR`. The last line is a mistake yet in the Java2VHDL (RFC), but you should use a `@Fpga.STDVECTOR` for the input. The conversion in the second line is done because the expression starts with a `STD_LOGIC_VECTOR`-type. For that the second operand is converted to the current expression type.
- `*` and `/` are not supported. An FPGA does not support more complex arithmetic by standard features. For multiplication you may include a multiplication module.
- But as RFC it should be simple to translate a multiplication with a simple less constant value and a division by a power of two to the proper + and shift (or better bit select) operations. They can be routed in a simple way. For example `val * 5/8` should be a proper requested operation which can be translated. Whether also `val * 0.625f` wíll be correct translated – may be also possible.

19. Shift operations

```
      this.out.bVal2 = this.val.bvec2 >> 2;
      this.out.bVal3 = (this.val.val1 << 3) + 5;                    RFC2
```

```
mdl1_Out.bVal2 <=  mdl1_Val.bvec2 SRL 2 ;
mdl1_Out.bVal3 <= TO_BITVECTOR( ( (mdl1_Val.val1 SLL 3 ) )  + 5 );   ERROR
```

- Also shift operations are translated. But VHDL has less problems with itself, shift operations works only for `BIT_VECTOR` and numeric operations works only for `STD_LOGIC_VECTOR`. But this should be clarified in the future from Java2VHDL, with necessary conversions.

4.10.4 Operands

For left side Operands, the variable to assign, see chapter 4.10.2 Assignments page 71

Operands in expressions can be gotten in java with the following references:

20. Operand access in a constructor of a process

```
@Fpga.VHDL_PROCESS  Val(int time, Val z, Ref ref, ModuleXY thism) {        // (4)
  boolean localVar1 = ref.ioPins.input.reset_Pin;                          // (5)
  this.bit1 = z.bit1 & localVar1 & ref.mdlXx.getSpecValue();               // (6)
  this.bit2 = thism.pCE.ce;                                                // (7)
  this.val1 = ref.mdlXx.getConstValue();                                   // (8)
```

produces the following VHDL lines

```
mdl1_Val_PRC: PROCESS ( clk )                                              - (4)
  VARIABLE localVar1 : BIT;                                                - (5)
BEGIN IF(clk'event AND clK='1') THEN
  localVar1 := reset_Pin;                                                    (5)
  mdl1_Val.bit1 <=  mdl1_Val.bit1 AND localVar1  AND module2_Qx.bit1 ;       (6)
  mdl1_Val.bit2 <=  mdl1_PCE.ce;                                             (7)
  mdl1_Val.val1 <= TO_STDLOGICVECTOR(OtherModuleXz_getConstValue);           (8)
```

(4) The following examples for access are inside a constructor describing a PROCESS

(5) Delaration of a local process variable as local variable in constructor. It is not final because also in VHDL it can be assign in several branches. The last seen assignment is the used value, adequate to Java. The initial assignment is only the initial one.

Since you use ref…. here, the resolution of the access is created for VHDL.

(6) The access to z…. is the value of the last state. In VHDL it is intrinsic to access the last state in a PROCESS, hence nothing specifics is written there.

In a processes constructor you should never access `this.var` because it is the value of the next step. In VHDL adequate `z.var` is used. The simulation will be faulty using `this.val`.

The next term uses ref… hence it is resolved to the implementation of the interface access in Java.

(7) This example uses via thism… the access to another inner class value, which is another RECORD. It is adequate resolved and translated.

(8) The getConstValue() is implemented return 0x3456, a constant value. Hence, a variable is defined in VHDL:

```
CONSTANT OtherModuleXz_getConstValue : BIT_VECTOR(15 DOWNTO 0) := x"3456";
```

This variable is accessed. But because it is a simple value, it is typed with the BIT_VECTOR. Hence the adaption to the assigned variable type is generated automatically.

21. Access in output() or input()

```
TODO
```

22. Operands for constants

```
this.ct = 0b1001;     // set ct to 9                    (1)
this.ct = 9;          //faulty
this.val = 0x0000;                                     (2)
this.val = 0;         // faulty
this.bit1 = true;                                      (3)
this.olg1 = 'Z';                                       (4);
this.bit2 = this.olg1 == 'Z';                          (5)
```

(1) For bit or STD_LOGIC_VECTOR constants you can write the value is binary constant. The number of digits should match exact the number of bits in the vector.

Note: This is in version 2023-03. RFC2 is, detect destination type and convert automatically from the numeric given value.

(2) If the number of bits is able to divide with 4, you can also use the hexa representation, but here also with the exact number of digits.

(3) Assignment of true and false to `BIT` or `STD_LOGIC` in Java presented as `boolean` is proper.

(4) For `STD_LOGIC` presented as `char` in Java you can use the known character which are also proper for VHDL: '1' '0' 'H' 'L' 'Z' If you use 'U' 'X' 'W' '-' then you should think about what it means for your Java test. All character values are translated to VHDL.

(5) Writing a comparison with a constant value, here especially for a char represented STD_VALUE produces a true or false in VHDL, as written. It is cast to the destination type if necessary..

4.10.5 Special operations for bit vectors

VHDL knows immediately access to bits written as

```
… myVector(13 DOWNTO 6) ...
```

This accesses to the defined bits and it presents a vector here with 8 bits.

In Java this might be written as

```
… ((myVector >>6) & 0xff)
```

delivering the same result. But this might be not proper obviously and it is also not simple to correct translate. It is not obviously what is ment. Because it may be also translated to

```
… ((myVector LSR 6) AND x"ff") ...
```

which presents a vector of the original length of `myVector`.

It is better to use operations in Java with a dedicated semantic. They are:

23. Bit vector operations

`Fpga.getBit(vector, 5)` (1)	`vector(5)`
`Fpga.getBits(vector, 13, 6)` (2)	`vector(13 DOWNTO 6)`
`Fpga.getBitsShl(vector,15, false)` (3)	`vector(14 DOWNTO 0) & ('0')`
`Fpga.getBitsShr(true, 15, vector)` (4)	`('1') & vector(15 DOWNTO 1)`
`Fpga.concatBits(16, vec1, 8, vec2)` (5)	`vec & vec2`

In the examples above the numbers should be numbers, but the number itself is exampleness.

(1) Access to one given bit in a vector, resulting type is BIT or STD_LOGIC

(2) Access to some bits, left and right bit is given. Resulting same type with new length.

(3) Shift a vector 1 bit to left, with given number of bits and given bit0

(4) Shift a vector 1 bit to right, with given right bit and the number of bits shifted

(5) Concatenation of any vectors. This is a combination of bit selection and concatenation:

24. concatBit variations

```
Fpga.concatBits(16, this.out.oVec81, 8, this.out.oVec82);                                   // (1)
Fpga.concatBits(16, this.out.oVec81, 8, Fpga.getBits(this.out.oVec82, 6,1));                // (2)
Fpga.concatBits(16, this.out.oVec81, 10, this.out.oVec82, 6, this.out.oVec83);              // (3)
Fpga.concatBits(16, 0b000, 13, this.out.oVec81, 5, this.out.oVal4);                         // (4)
Fpga.concatBits(16, this.out.oVec81, 5, this.out.oVal4);                                    // (5)

Fpga.concatBits(16, 0b000, 13, this.out.oVec81, 5, this.out.oVal4);                 (4)
Fpga.concatBits(16, this.out.oVec81, 5, Fpga.getBits(this.out.oVec82, 6,1));        (3)
Fpga.concatBits(16, this.out.oVec81, 6, Fpga.getBits(this.out.oVec82, 7,2));
```

```
mdl1_Out.oVec81 & mdl1_Out.oVec82                                                           (1)
mdl1_Out.oVec81 & "00" & mdl1_Out.oVec82(6 DOWNTO 1)                                        (2)
mdl1_Out.oVec81(5 DOWNTO 0) & mdl1_Out.oVec82(3 DOWNTO 0) & mdl1_Out.oVec83(5 DOWNTO 0)     (3)
"000" & mdl1_Out.oVec81 & TO_BITVECTOR(mdl1_Out.oVal4(4 DOWNTO 0))                          (4)
"000" & mdl1_Out.oVec81 & TO_BITVECTOR(mdl1_Out.oVal4(4 DOWNTO 0))                          (5)



mdl1_Out.oVec81 & "00" & mdl1_Out.oVec82(6 DOWNTO 1));                              (1)
"000" & mdl1_Out.oVec81 & TO_BITVECTOR(mdl1_Out.oVal4(4 DOWNTO 0)));                (2)
"000" & mdl1_Out.oVec81 & mdl1_Out.oVec82(6 DOWNTO 1)(4 DOWNTO 0));                 (3)
"00" & mdl1_Out.oVec81 & mdl1_Out.oVec82(7 DOWNTO 2));                  (4)
```

The 1th, 3th etc. value is a simple number of bit positions. The next argument is a vector. If the vector is longer than the difference to the next bit position, a sub vector is built, right aligned. If the vector is shorter, some 0 bits are included.

`(1)` This is a simple form, two vectors are concatenated with the original length. The length should

be known in Java.

(2) A vector can be built also from a getBits() access to sub bits. In this example it is combined with padding 0-buits, to accomplish the 8th position.

(3) This is a combination to access sub bits from three vectors. The bit positions 16, 10 and 6 determines the selection of sub bits in the vectors, but it is right aligned.

(4) Th 3th Vector is shortened because it has lesser bits, the 2th vector has exact its 8 bits, and left side it is manually padded with 0. This is obviously.

(5) It produces the same result because of automatic padding. The user should decide

(2) This example shortens the 3th vector. The left vector is added manually with a constant to padding.

Secondly the 3th vector is of BIT_VECTOR type whereas the 2th vector is a STD_LOGIC_VECTOR type. The type of the expression follows the first given vector from left. To fulfill concatenation the right vector is converted.

(3) This line results in an error in VHDL, the problem is shorten in combination with an bit select operation. You should select the correct number of bits as done in (4).

(4) todo more concise examples

4.11. Test organization on Java level

One of the benefit of the Java2VHDL approach is, it is possible to make elaborately functional tests on Java level to evaluate the logic in the FPGA.

It is common known that the effort for test is often two times more as the functionality for the product itself. So it can be seen also here. Why is it so?

The test can be done of course in practical usage. It should be do so. Then you see practical effects of the functionality, some stuff may be worse. But then you must think about why, the effort is high, and maybe your costumer is not happy.

Hence, the test conditions should be simulated before the product is ready, the tests can be done in software on the desk of the developer, and the advantage is: The developer can look to details inside the logic. The effort to simulate the outer environment may be high to get expressive test results.

The test environment can be free programmed in Java using all capabilities, for example reading test stimuli data from Excel sheets or other result files.

4.11.1. General execution order for java execution of the FPGA functionality

The Java execution starts in the `static void main(String args[]){…}`

You can provide some arguments from a outside test environment which calls the Java program for example with different stimuli for a repeated test after changes, but you can also organize different tests in Java itself in the `main(…)` routine.

It means, you can call different sub routines for test and calling the FPGA execution.

In this different sub routines, or only in the one `main(…)`, some instances of classes which simulates the environment, can be instantiated and initialized.

If you do it simple, you should stimulate only the inputs of the only one FPGA with simple signals. If the test is done more elaborately (in progress of development) you may have more as one FPGA to test co-working, you should emulate the behavior of a embedded control adapted to the FPGA. Note that you can also organize co-working with the original embedded control software maybe in C language, with connection via Socket interface or shared memory: The embedded software may also work in PC, can have a socket interface (not complicated on PC simulation), and socket connection in Java is also simple. So you get signals from the embedded control with original functionality to use for your FPGA test.

Java: Test main(…)

```
public static void main ( String args) {
  MyTest thiz = new MyTest(args);                          // (1)
  TestOrg test = new TestOrg("Test_MyTest_All", 3, args); // (2)
  try {
    thiz.executeTest_A(test);                              // (3)
  }
  catch(Exception exc) {                                   // (4)
      System.out.println(exc.getMessage());
      test.exception(exc);                                 // (5)
  }
  test.finish();                                           // (6)
}
```

(1) This schema shows execute more tests with one given argument set `args`, maybe used or not. The instance of the main test class is initilaized.

(2) The ../../../Java/docuSrcJava_vishiaBase/org/vishia/util/TestOrg.html is a meanwhile proven concept a little bit similar as Google test but it is better.

(3) Here you can execute more as one routine for each test one after another.

(4) This is only a general catch for unexpected errors.

(5) The `test-exception(..)` writes an exception info and the message, that the test was aborted.

(6) The `test.finish()` now closes the test organization and writes end information.

But see now one execution operation. It shows the simplest way is, only stimulate inputs with simple functionality.

Java: one test routine calls the FPGA step

```
executeTest_A(TestOrg testParent) {
  TestOrg test = new TestOrg("testXy", 6, testParent);        // (1)
  MyFpga fpga = new MyFpga();                                  // (2)
  fpga.reset();                                                // (3)
  TestEnvironment testEnv = new TestEnvironment(...);          // (4)
  testEnv.reset(fpga);
  TestSignalRecorderSet outClk = new TestSignalRecorderSet();
  initTestSignalRecording()                                    // (5)
  // initialize some pins as delivered from hardware from begin  (6)
  fpga.modules.ioPins.input.reset_Pin = false;   // lo active from begin
  int time = 0;
  while(time < 10000) { // does here 10000 clock edges          // (7)
    if(this.time == 50) {
      fpga.ioPins.input.reset_Pin = true;  // hi passive reset  (8)
    }
    testEnv.setFpgaPins(fpga, time);                          // (9)
    fpga.step(time);                    // Simulation of FPGA (10)
    fpga.update();
    fpga.output();
    testEnv.step(time, fpga);  // simulation of the test environment (11)
    signalRec.addSignals(time);                                // (12)
  } //while
  // time test is finished
  signalRec.output();                                          // (13)
  testEvaluation(test, signalRec);                             // (14)
  test.finish();                                               // (15)
}
```

(1) For the test evaluation build a sub **`TestOrg`** instance, closed with <15>

(2) The FPGA top level is initialized, see chapter 4.6 Java source for top level FPGA class. Alternatively you can instantiate the FPGA (or more as one) outside, reuse it and call

(3) `fpga.reset()` to bring the FPGA after other tests back in the reset state, it is adequate switch off power and switch on again to start the test with a clean FPGA. Note that stimulate the reset pin, see **<6>** and **<8>** may not force a clean FPGA, because it depends on the FPGA content (VHDL) itself.

(4) Also, allocate or initialize your specific test environment which may be simple for the first test. This can now influence FPGA pins as initial state.

(5) The **`TestSignalRecorderSet`** handling is explained in 4.10.3 The TestSignalRecorderSet to record test signals from modules

(6) This is a simple possibiity to determine signal pins on begin of the test, here the low active reset is set to low, as a reset logic may do on power on or with a outer button.

(7) The **`int time`** counts the clock steps and helps to formulate stimulis signals and information for test signal outputs. It is also used to determine the test after a time. Of course the test can be determined depending form some signals from the FPGA or from the test environment.

(8) The FPGA pins can be set immediately simple depending from the time, here done for the rest pin set it high inactive after a dedicated time …

(9) Or the FPGA pins can be determined from an operation in the test environment class which can be more comprehensive.

(10) **`step(time)`**, **`update()`** and **`output()`** for the top level of the FPGA should be executed in each time step. It is the simulation of the FPGA functionality.

(11) The adequate is done for the environment simulation. Note that this can be also written as

FPGA logic if it is hardware. But is can be more simple if it should only produce inputs for the FPGA. Possible here also, use measurements from real behavior for the input of the FPGA. But also the dependence from the state of the FPGA may be important.

(12) Test signal recording is called in any step. It may be to save run time in Java that this is called only after some steps, for example if signals should be recorded lesser, for example only if a '***clock enable***' signal inside the FPGA is active. It may also depend on test cases which signal recorder is used if you have prepared more as one. Often for general tests all signals in any clock edge should be gathered. But if the general logic works, only lesser signals are gathered.

(13) After end the test run in the `while { … }` loop the test signals may be output to a file to visit it manually. Because there are lines with time in columns, the output cannot be done during the running test, only on end. But for debugging you can visit the content of the StringBuilder. <14> A routine for test evaluation may analyze the content of the `StringBuilder` of the signal recorder, either to immediately compare with a pattern, or to compare the significance of the content. It should write information in the `TestOrg` instance.

(14) This is the `TestOrg#finish()` which writes some information to console or to a file for evaluation of the test result.

4.11.2. Execution order inside the FPGA for the test

Lets have first a remark to execution time.

The execution of Java is very fast. Tests with more as 1’200’000’000 steps are done for 12 seconds test run with a clock of 100 MHz (10 ns). It works in the simulation only a few minutes. This is not primary expectable, because there are 1200000000 calls of `new` for the process instances of some modules. But generally, `new` is a cheap operation if the memory is sufficient. The memory is sufficient, if you need for example 3 kByte for your data, you need only 3 GByte for one million steps. Then the garbage collector can work to clean up, but all instances are less and one after another, so that also the garbage collector has not to much to do. It need milliseconds, no more.

You can save calculation time, if you regard, that some processes works only with a *clock enable* signal. Also in the FPGA 10 ns are less, and the *clock enable* simply the timing. If you know the functionality of the ***clock enable***, you can use it to call `step(…)` not in any simulation step, instead for example only any 10th time, for specific processes only which runs with a dedicated *clock enable* .

That can all regarded in Java level if the simulations grows and get slowly. But you should do carefully such things, think about the functionality, to avoid differences between the really functionality in the FPGA hardware and your test result.

One remark additionally. Of course the tests are only functional tests without regarding the timing in the FPGA. But the timing should be met anyway, see chapter 3.1.2 Timing relations and

The operations for step() and update() as well as reset() and output() are also presented in the chapter Error: Reference source not found page Error: Reference source not found. They are important especially for the test.

The `step( int time) {…}` operation can also contain time tests of signals to check the timing constraints, see chapter

4.11.3. The TestSignalRecorderSet to record test signals from modules

The org.vishia.fpga.testutil.TestSignalRecorderSet is a container which holds all org.vishia.fpga.testutil.TestSignalRecorder from all modules. But it should be initialized with the modules. See also 4.11.1. General execution order for java execution of the FPGA functionality, page 79, there the code template for '*one test routine calls the FPGA step*':

Java: initializing of the TestSignals for the whole simulation.

```
void initTestSignalRecording ( ) {
  this.outClk = new TestSignalRecorderSet();             // This is to show all te...
  this.outClk.registerRecorder(sim.fpgaA.modules.moduleXy.new TestSignalsFrame("A-...
  this.outClk.registerRecorder(sim.fpgaA.modules.moduleXy.new TestSignalsAll("A-Xy...
  this.outClk.registerRecorder(sim.fpgaA.modules.moduleAb.new TestSignals("A-Ab"));
}
```

This is only a simple example. The FPGA may (only) have two modules. The non static inner classes derivend from `TestSignalRecorder` are instantiated here, with the `.new` operator given with the module instance reference as it is necessary to initialize non static inner classes from outside. The initialized classes get the reference to this named outer class which is necessary by this `outerIntance.new(…)` construct, may be not anytime known by occasionally Java programmers.

In this example it is also demonstrated that the `moduleXy` has two inner classes for `TestSignalRecorder` with different signals. Yet it can depend on test conditions which recorder is used, or both as shown here. The argument of the `TestSignalRecorder` constructor is the module name as it is written before the internal signal names, this is determined also here.

See Error: Reference source not found page Error: Reference source not found

4.11.4. Evaluation of the recorder test signals

After test the StringBuilder in the TestSignalRecorder are filled with information.

Firstly you can visit it manually, writing in a File as mentioned in 4.11.1. General execution order for java execution of the FPGA functionality page 79 point `(13)`. That is important to look on new functionality and study the correctness.

But if you want to run repeated tests only for checking, nothing was wrong by made changes, the manually comparison is expensive and error-prone. You don't may see important faulties.

That's why for such approaches automatically comparison and automatically test executions are proper.

The simplest for is, compare with the last given results – should be the same. But some times the outputs depends from not relevant input changes. The stupid comparison returns errors, whereby only maybe a little bit other timing is given which is not so relevant. Of course because of that the comparison pattern can be adapted, after elaborately study of the correctness. That is costly.

Sometimes only a sequence should be checked: Comes an output because of an input in a given time span or not. Then more sophisticated analyzes of the depending output lines can be done.

For example you have different automatically tested input stimuli, with different width of signals, and your test result should be: The outputs should follow the width. You do not test pattern of outputs with pattern of inputs, you should test functional requirements from inputs to outputs. For example, output should follow input with given delay depending of a parameter.

Maybe also, you don't want to compare all detailed signals, only want to see deterministic outputs.

But all that is proper able to program on Java level.

The template for doing checks with the output is:

```
Test test = new Test
    TestOrg test = new TestOrg("testName", 6, testParent);
    … execute the functionality
    boolean bOk = true;
    for(int ix = 0; ix < this.txData.length; ++ix) {
      bOk &= (this.rxData[ix] == this.txData[ix]);// compare tx with rx for own test
    }
    test.expect(bOk, 7, "rxData[] should be equal as txData[]");
    test.finish();
```

This example shows the check of a result of a complex simulation with two FPGAs and also a simulated SPI interface of a controller (SPI = Serial Peripheral Interface which is usual given on controller chips and often used for FPGA communication). The test is met if the received data of the other FPGA are equal to the transmitted ones, whereby different conditions were be given. Details are not tested here. It is an end test.

Details may be manually viewed, or tested with the adequate effort.

4.12 Checking time between FF groups

Timing constraints are essential for routing because the place & route tool of the FPGA implementation should know which signals can have a longer path. Usual not all path can met the time between two clock edges, if you have for example a FPGA system clock of 100 or 200 MHz.

The timing constraints are also checked on Simulation in Java.

4.12.1 How to set timing constraints for place and route tool

For ***Lattice Diamond*** you can write in the lpf file (constraint file) the following information about timing:

25. Timing constraints for Lattice Diamond

```
PERIOD PORT "clk" 10.000000 ns ;                          (1)
DEFINE CELL GROUP "Ce0"                                   (2)
"*CE0.*"                  (3)
"data*"
;
DEFINE CELL GROUP "Ce7"                                   (2)
"*CE7.*";                 (3)

MULTICYCLE FROM GROUP "Ce0" TO GROUP "Ce0" 10.0 X;        (4)
MULTICYCLE FROM GROUP "Ce7" TO GROUP "Ce7" 10.0 X;
MULTICYCLE FROM GROUP "Ce0" TO GROUP "Ce7"  7.0 X;
MULTICYCLE FROM GROUP "Ce7" TO GROUP "Ce0"  3.0 X;
```

(1) Here first the clock period is defined.

(2) A `CELL GROUP` is a group of Flipflops which switches with the same clock enable signal. The name follows here the clock enable signal, here `Ce0`.

(3) The members of a group are names of the registers able to see in the *Netlist View*. The names are built after the routing process as names of the **implementation** in the flattened view to the real FPGA content. This names can be written with wildcards, of course, important for register groups.

This names follow more or lesser the names in the VHDL sources. If you have modular VHDL sources, the names are sometimes not very well predictable. But for the flattened VHDL file, which is produces by Java2VHDL, the name follows simply the SIGNAL names of the TYPE … RECORD definitions. This SIGNAL names are translated from Java2Vhdl using the module name in the Top level, maybe nested module names and then the name of the inner Process class type. This are your names in Java. Hence they are predicted.

If you have your own naming conventions, for example using the suffix “CE0” for all SIGNAL records, which are determined in PROCESSes by a “CE0” *clock enable*, then you can use this convention for a simple build of the group. But this is a special solution, not applicable for large designs with different modules. It presumes a regularity of name schemes, which may not be possible in any case.

The better solution is using the automatic generated part of the constraints from the Java2Vhdl converter, which lists all Flipflop groups following the detected clock enable signals with the Interface usage `CeTime_ifc`, see following chapters.

(4) This line clarifies, that the time between the Flipflops of this GROUP is the given factor longer than the clock PERIOD. Because the clock enable signal of all FlipFlops of this GROUP comes with this period, it is proper.

Unfortunately, the names for Cell groups should be existing names in the routed design. It would be better, the original source names may be usable. It means VHDL should have a language

feature to associated flipflop groups.

The Java2VHDL translator has the advantage, that the names of the routed ”CELL” can follow simply the SIGNAL names of RECORD in VHDL, if the VHDL is the top level module. Because the translating from Java2VHDL is done in a flattened way, s*ome modules which are aggregated and hold in different module files in Java are translated to only one top level (or sub level) VHDL file,* it is usual possible to predict the names of the Cells in the routing design. Hence, the association from PROCESS class variables to the time cell groups can be done by automatically, see next chapter.

But unfortunately, if the router discardes some FlipFlops because their outputs are not used (optimization), the translator produces a CELL GROUP name which is not existing in the routed design. Then the Diamond tool ignores the whole cell group, only visible as warning on routing logging. This is a disadvantage or the router and can be fixed there. “*Non existing signal names should not disable the whole CELL GROUP built with them*”. This is tested with Lattice Diamond tool 3.12.0.240.2.

It means the tuning between the automatically generated DEFINE CELL GROUP with the used plf file should be made manually, with checking whether all names are existing in the net list. That is not a sophisticated work, and also, it can help to detect unexpected optimized blocks. If you have such one, you should mark it on Java level.

RFC3 for Java2VHDL: An annotation is necessary to mark PROCESS classes which should not be translated to VHDL, for example because there are used only for specific simulation situations.

See chapter 7. Requests for Change (RFC) for the Java2Vhdl tool page 114

The output produced in the file given with option `-oc:path/to/constraint.ext`: looks like

```
DEFINE CELL GROUP "mdl1_CE0"
"ioPins_Output_data.*"
"mdl1_Q_CE0.*"
;
DEFINE CELL GROUP "mdl1_CE7"
"mdl1_Val_CE7.*"
;
```

It should be merged in the used constraint file.

4.12.2 Association between PROCESS variables and time GROUPs

The next pattern is related to chapter 4.8.2. Inner class for records and process page 58:

It is a good decision if all values in a `@Fpga.PROCESS` inner class switches with only one *clock enable*. Then all Flipflops can be assigned to one time `GROUP` in constraints.

26. Start of `PROCESS static class` with `time_` variable

```
@Fpga.VHDL_PROCESS public static class Q_CE0 {            // (1)
  public final CeTime_ifc time_;                          // (2)
```

`(2)` The process class should contain this variable. It should be set in the constructor with the source of the clock enable signal `(5)`, `(6)`:

27. Start of `PROCESS` constructor with `ce()` condition and time `GROUP` selection

```
  @Fpga.VHDL_PROCESS  Q_CE0(int time, Q_CE0 z, Ref ref, ModuleXY thism) {    // (5)
    this.time_ = thism.srcCE0;                    // (6)
    if(thism.srcCE0.ce()) {                       // (7)
      .. ..
    } else {
      // nor more relevant statements to change process variables  (8)
    }
```

`(7)` This is the essential statement to associated this `PROCESS` class to the time `GROUP` which is defined with the `CeTime_ifc`. The Java2Vhdl translator detects, that the reference `thism.srcCE0` is type of this interface. The implementing instance contains information about the time `GROUP`.

`(8)` The if...else should be the only one branch which sets the process variables with new values. The `else` branch should only contain `this.varx = z.varxy;` which has no contribution for the variables in VHDL. Or it contains a `throw new IllegalStateException();` as shown in chapter 4.8.2. Inner class for records and process page 58.

Now have a look to a PROCESS class which defines the CE signals:

28. Example for build to ce signals

```
@Fpga.VHDL_PROCESS public static class PCE {
  final boolean ce, ce7;
  final @Fpga.STDVECTOR(4) int ct;
  final int time_ce, time_ce7;
  PCE(){
    this.ce = this.ce7 = false; this.ct = 0;
    this.time_ce = this.time_ce7 = 0;
  }
  @Fpga.VHDL_PROCESS PCE(int time, PCE z, Ref ref, ModuleXY thism) {
    if(Fpga.getBits(z.ct, 3,2) == 0b11) {
      this.ct = 0b1000;            // counts 8..0,-1, 10 times.
      this.ce = true;              // (1)
      this.time_ce = Fpga.checkTime(time, z.time_ce, thism.srcCE0.period());  // (2)
    } else {
      this.ct = z.ct -1;
      this.time_ce = z.time_ce;
      this.ce = false;
    }
    if(z.ct == 0b0010) {
      this.ce7 = true;
      this.time_ce7 = Fpga.checkTime(time, z.time_ce, thism.srcCE7.period());  // checks the...
    } else {
      this.ce7 = false;
      this.time_ce7 = z.time_ce7;
    }
  }
}
protected PCE pCE = new PCE(); private PCE pCE_d;
```

It is only an example. The two clock *enable signals* comes from a counter which counts from 8 down to -1. It is also possible and sensitive, that a clock enable is related to an input signal which has a significant period. A filter (an *input clock enable synchronization*) may clarify that some aberration may shorten or elongate the clock enable period, but in a related kind. The clock enable period for a signal which comes for example from a 10 MHz input can vary then between 9 .. 11 system clock ticks of 100 MHz.

The more important parts for timing constraints are the access agents to the clock enable. For that the interface `org.vishia.fpga.stdmodules.CeTime_ifc` is given:

29. Access to clock enable with time definitions:

```
  @Fpga.IfcAccess public CeTime_ifc srcCE0 = new CeTime_ifc() {                    // (1)
    @Override public boolean ce () { return ModuleXY.this.pCE.ce; }               // (2)

    @Override public int time () {                                                // (3)
      return ModuleXY.this.pCE.time_ce;
    }
    @Override public String timeGroupName () { return "CE0"; }                    // (4)

    @Override public int period () { return 10; }                                 // (5)
  };
```

`(1)` This anonymous interface implementation of `CeTime_ifc` is detected by Java2VHDL, can be located in any module, and produces the constraints.

`(2)` The access operation `ce()` as part of an expression is accepted by the java2Vhdl translator and returns the value determined by its `return` statement.

`(3)` The `time()` operation can be used to get the time of the last setting of the appropriate clock enable signal. This is necessary for checking the time on Java simulation. The time can be stored as shown in 28. Example for build to ce signals left side on marker `(2)`.

`(4)` The `timeGroupName()`, its return string, is evaluated by the Java2Vhdl translator to for the constraint file generation.

`(5)` The return value of `period()` should be a constant literal, the minimal number of clock steps between two active `ce()` signals of the accessed instance. It is used both for the constraint file and as argument for time checks in Java, see `(2)` in 28. Example for build to ce signals .

The Java2Vhdl translator does the following:

- Time groups are built with all found interface access agents of type `CeTime_ifc` in all modules,. The name of the time group is the module name, following the value of the `timeGroupName`, It means if you have more as one module with the same type, of course it builds different time groups.
- All processes which uses the `ce()` of any `CeTime_ifc` access as exclusively if condition for all process variables are associated to this time group.
- All processes which are not exclusively use one `ce()`, are not associated to time groups. It means, you should sort your processes. Each process should associated to exact one clock enable which is delivered from such a `CeTime_ifc`. Then you have a proper order for time Groups and constraints.

See also the common explanation in chapter 3.2 Timing relations

4.12.3 Check of timing between Flipflops in Java

If you know your signals, then it is sufficient without any test. But better is, also test it.

The basic for the test is the `time` argument of all step routines and PROCESS constructors. With the current time, which counts up on any clock edge, you can test the last time of set an accessed, used signal, and check the minimal difference. If you have met all sensitive situations in test then you have a proper tested evidence.

The real delay in the time groups can be checked while building this signals itself:

```
    if(...
       this.ce = true;              // (1)
       this.time_ce = Fpga.checkTime(time, z.time_ce, thism.srcCE0.period());  // (2)
```

The `Fpga.checkTime(…)` operation takes the current time and a second time, builds the difference and compares it with the given min difference. The output is the first argument `time`, which allows writing comparison and assignment in one line. The `z.time_ce` is the last time before activation of ce, the current will be set in `this.time_ce`. The comparison is done with the same value which is stored in the access agent for ce. This is done any time on new build of ce.

The delay between the time groups can be tested one time per step in the main loop, if the main loop knows all relevent time groups:

```
  @Override public void step ( int time ) {                                  // (10)
    this.modules.mdl1.srcCE0.checkTime(time, this.modules.mdl1.srcCE7, 3);
    this.modules.mdl1.srcCE7.checkTime(time, this.modules.mdl1.srcCE0, 7);
```

Here manually determined with 3, it checks that any `ce()` from the srcCE0 comes minimal 3 clocks after `srcCe7.ce()`. This is the proper value for the constraint between this groups. It is not automatically translated in the current version of Java2Vhdl (2023-04) but can manually taken.

Additionally, any signal can be tested with the `Fpga.checkTime(…)` operation. The time can be stored in a variable time_ in any class, it is not used for Java2Vhdl translation, only for Java test. This may be interesting in sophisticated situations.

5. The example Blinking LED, view to Java sources in respect to the FPGA description

This is a study example or template for your own. The sources are located for the exmaple.zip in:

- ../../deploy/Example1_BlinkingLed-2022-05-26.zip A zip file containing an example and link to the tool base.
- **Note: This content should be updated to the current version.**

```
src
 +-main
   +-java
      +-srcJava_FpgaExmplBlinkingLed
         +-org/vishia/fpga/exmplBlinkingLed    This should be your own package pat...
            +-fpgatop/*.java
            +-modules/*.java
            +-test/*.java                      only for test on Java level, may be...
```

The content of the files are partially described already in the approach document / chapter 3 Java2VHDL - approaches This chapter describes it from the view of the template for your own.

5.1. The top level FPGA java file

All following code snippets comes from the `main/java/srcJava_FpgaExmpl…/fpgatop/BlinkingLed_Fpga.java`.

5.1.1. Package and class definition, import

In Java always the name and path of the file itself should match to the package declaration and name of the only one `public` class inside the file: The Java file starts with the package declaration. The package names and also the appropriate directories on the file system must be written starting with a lower case character.

Java: top level class definition

```
package org.vishia.fpga.exmplBlinkingLed.fpgatop;

import org.vishia.fpga.stdmodules.Reset;
    ......

public class BlinkingLed_Fpga implements FpgaModule_ifc {
```

The `import` statements name all used classes from other packages. It is possible to use an asterisk to select all Files in the package. But then the dependencies are not well documented. In this case anyway all files in the 'exmplBlinkingLed.modules' package should be part of, then it is ok.

The module class should implement the `FpgaModule_ifc`. That is all necessary. This interface defines:

Java: FpgaModule_ifc definition as basic interface

```
/**This interface should unify a module for FPGA.
 * The here defined operations are necessary especially for test in Java.
 * Not necessary for VHDL translation.
 * @author Hartmut Schorrig www.vishia.org
 *
 */
public interface FpgaModule_ifc {
```

```
  /**Creates an initial state as after hardware reset.
   * Usual the default ctors should be called here.
   * The operation should be called first on start of simulation.
     * Especially necessary on reusing of a given instance (without new
construction)
   * for several tests.
   * Pattern: <pre>
   * void reset ( ) {
   *   this.my = new My();
   * }</pre>
   */
  void reset ( );

  /**This operation should prepare all D-inputs of flipflops. It is the creation
o...
   * Pattern: <pre>
   * void step ( int time) {
   *   this.my_d = new My(time, this.my, this, ref);
   * }</pre>
   *
   * @param time
   */
  void step ( int time);

  /**This operation should update the Q-Outputs of flipflop from D
   * and can also output signals to ports.
   * Pattern: <pre>
   * void update ( ) {
   *   this.my = this.my_d;
   * }</pre>
   */
  void update ( );


}
```

5.1.2. The modules in the top level

The class definition continues with the

Java: used modules in the top level

```
  /**The modules which are part of this Fpga for test. */
  public class Modules {

    /**The i/o pins of the top level FPGA should have exact this name ioPins. */
    public BlinkingLed_FpgaInOutput ioPins = new BlinkingLed_FpgaInOutput();

     /**Build a reset signal high active for reset. Initial or also with the
reset_...
     * This module is immediately connected to one of the inputFpga pins
     * via specific interface, see constructor argument type.
     */
    public final Reset res = new Reset(this.ioPins.reset_Inpin);

        public final Test_Combinatoric_BlinkingLed vhdl_Combinatoric = new
Test_Combin...

          public final BlinkingLedCt ct = new BlinkingLedCt(this.res,
BlinkingLed_Fpga.t...

    public final ClockDivider ce = new ClockDivider(this.res, this.ct);
```

```
    Modules ( ) {
      //aggregate the module afterwards
                  this.ct.init(this.res,   BlinkingLed_Fpga.this.blinkingLedCfg,
this.ce);    //...
    }
  }

  public final Modules modules;
```

The modules are aggregated together as described in Java2Vhdl Approaches, chapter More possibilities with Java2VHDL: References (aggregations) in Object Orientation kind.

Because you can generate a ***Javadoc*** it is recommended to comment all modules in the given style (here not all are commented). But also a non commented style is sufficient because you can use cross referencing in the IDE.

The modules should be connected immediately here, either on instantiation with a parameterized constructor, or with the `init(…)` operation in the constructor of modules. It depends on circular referencing whether the immediately referenced instantiation can be done. That is more simple. But using `init(…)` has more flexibility.

Firstly in this class the `ioPins` are defined. This is done in an extra class, see next chapter.

The writing style with the explicitly `this` is recommended. `this` is also implicitly accept (can be omitted), but then the relations are worse documented. The `BlinkingLed_Fpga.this` writing style is necessary for the translator. `BlinkingLed_Fpga.this` is the reference to the environment class. Java can automatically detect this relation, it checks whether the following identifier is able to find either locally, or in the own class, or in all environment classes. This is more error prone because of confusion in identifier usage. Hence the dedicated writing style `EnvironmentClass.this` prevents the confusion. For the Java2Vhdl translator it is also more simple.

5.1.3. step(…) and update() operations

Following the both routines are defined:

Java: top level class step update

```
  @Override
  public void step(int time) {
     this.vhdlink_vhdl_Combinatoric = new Vhdlink_vhdl_Combinatoric(time, this,
thi...
    this.modules.res.step(time);
    this.modules.ce.step(time);
    this.modules.ct.step(time);

  }


  @Override
  public void update() {
    this.modules.res.update();
    this.modules.ce.update();
    this.modules.ct.update();
  }
```

See also Java2Vhdl_Approaches.html#stepupd

Both routines should call all `step(time)` and `update()` of all sub modules. The `time` comes from the simulation environment useble for time checking, see Java2Vhdl_Approaches.html#timeCnstrn and for signal output, see Java2Vhdl_Approaches.html#testOutp.

5.1.4. interface agents in the top level

The so named ***interface agents*** are anonymous class definitions as interface implementation to access data in this module. They are usable for referencing.

Java: top level class interface agent/access

```
  /**Provides the used possibility for configuration values.
   */
  @Fpga.IfcAccess BlinkingLedCfg_ifc blinkingLedCfg = new BlinkingLedCfg_ifc ( )
{

    @Override @Fpga.BITVECTOR(8) public int time_BlinkingLed() {
      return 0x64;
    }

    @Override public int onDuration_BlinkingLed() {
      return 10;
    }

    @Override
    public int time() { return 0; }  // set from beginning

  };
```

In the top level they should be used either for stubs instead not implemented modules, for test designs or variations, but also for parameter of modules which should be determined in the top level. This is shown above. The interface access implements a …`Cfg`… interface for configuration parameters. It is used as reference for the `ct(…, BlinkingLed_Fpga.this.blinkingLedCfg,…` module, see chapter 5.1.2 The modules in the top level

5.1.5. test output in the top level

The top level does not need test output if it has no own PROCESS sub classes. The output preparation for the main level can be done immediately in the test environment, see chapter 6.1.9 Test output preparation for the main level. This has no meaning for the Java to VHDL translation, writing this stuff to the test java file unburdens this file for translation.

5.2. The FPGA pin description file

It is a good idea to separate the Pin description source file from the top level file, because different inner FPGA designs can use the same pinning. This is typical if you have a hardware board with a given layout, but the content of the FPGA should be varied. Then you need define the pinning only one time for the given layout.

Furthermore it is also possible to have more as one top level FPGA file for different parts, but you should have only one file for the pinning, because this is board layout related.

5.2.1. How to designate the ioPins file

The pin description file should be designated as `ioPins` in the `Modules` class of a top level:

Java: ioPins definition in the top level

```
  /**The modules which are part of this Fpga for test. */
  public class Modules {

    /**The i/o pins of the top level FPGA should have exact this name ioPins. */
    public BlinkingLed_FpgaInOutput ioPins = new BlinkingLed_FpgaInOutput();
```

See also chapter 5.1.2 The modules in the top level. The designated class file is searched and

translated especially for IO pinning.

The ioPins file starts due to Java conventions with

Java: BlinkingLed_FpgaInOutput class definition head

```
package org.vishia.fpga.exmplBlinkingLed.fpgatop;

import org.vishia.fpga.Fpga;
import org.vishia.fpga.stdmodules.Reset_Inpin_ifc;

public class BlinkingLed_FpgaInOutput {
```

5.2.2. Input and Output inner classes

For this example the pin classes are short, for more pins it is a little bit more, very simple:

Java: BlinkingLed_FpgaInOutput Input and Output

```
  public static class Input {

    /**A low active reset pin, usual also the PROGR pin.
     * Because of the specific function the access should only be done with the
he...
     * For test it can be accessed in a test access class in the same package.*/
    boolean reset_Pin;
  }


  public static class Output {

     /**Ordinary output pins, use public in responsible to top level design
sources...
    public boolean led1, led2, led3, led4, led5, led6, led7, led8;
  }

  /**This instances are final and public accessible.
   * The inputs should be used in the step operations as given on start of D-
calcu...
   */
  public final Input input = new Input();

  /**The outputs should be set in the update() operation of the top level with
the...
  public final Output output = new Output();
```

- If the same pin should be used both as input and output, then the same pin name should be defined in the `Input` and in the `Output` inner class.
- If a pin should have tristate character (often necessary on input/output switch), it should be designated as `char` type and set with `'0'`, `'1'` and `'Z'`. (TODO for the 2022-05 version).
- The Input pins can be declared as ***package private*** (without designation) if an interface access is defined here also to set the pins. See next chapter. Elsewhere they can be designated as `public` for simple set accesses for tests.
- Output pins should be declared as `public` for immediately access for test and to set from the different top level FPGA java file maybe in different packages.

  The output pins are written immediately in the `update()` operation on the top level:

  Java: top level class using interface implementation and simple access operation for output

5.2.3. Interface access to the Input pins

The input pins are usual accessed via interface references from view of a module. Hence the Java2Vhdl_Approaches.html#IfcAccess should be defined in a proper way for all input pins. For specific pins a specific interface also with more as one bit can be used. For common ordinary pins the following simple interface is sufficient:

Java: Bit_ifc

```
package org.vishia.fpga.stdmodules;

import org.vishia.fpga.Fpga;

/**This is a very common interface only for one bit for any usage.
 * For some reason an input is necessary and an output is offered,
  * but the connection from the input to the output is determined not by a
module,
 * It is determined by the interconnection of modules.
 * In such cases a specialized interface should not be used.
 * The connection should be proper for any plug.
 * The responsibility of the correctness of the connection is associated to the
in...
 * <br>
  * The implementor should have the annotation <code>@Fpga.IfcAccess</code>,
example:
 * <pre>
  public @Fpga.IfcAccess Bit_ifc txReqMaster = new Bit_ifc ( ) {
    @Override public boolean getBit() {
      return SpiMaster.this.spiM.stateCmd;
    }
  };
 * </pre>
 * @author Hartmut Schorrig
 *
 */
public interface Bit_ifc {

  boolean getBit ( );
}
```

In this example only the access to the only one `reset_Pin` is necessary

Java: BlinkingLed_FpgaInOutput Interface access

```
  /**Get the reset pin as referenced interface access from a module.
      * Using the {@link org.vishia.fpga.stdmodules.Reset} may be seen as
recommended...
   */
  @Fpga.IfcAccess public Reset_Inpin_ifc reset_Inpin = new Reset_Inpin_ifc () {
                  @Override public boolean reset_Pin() { return
BlinkingLed_FpgaInOutput.this.in...
  };
```

This uses the `Reset_Inpin_ifc` as used in the accessing module `org.vishia.fpga.stdmodules.Reset`, see the `Modules` definition in the top level file.

To set this inpin a set operation is necessary because of the non public property of the pin variable. It is a good decision for overview (divide and conquer) to write this set operation in an extra file, but in the same package:

Java: BlinkingLed_IoAcc for test, the whole file:

```
package org.vishia.fpga.exmplBlinkingLed.fpgatop;

/**This class is only for test to access Pins in {@link
```

```
BlinkingLed_FpgaInOutput}
 * @author Hartmut Schorrig
 *
 */
public class BlinkingLed_IoAcc {

   /**Operation to set the reset Inpin, with the hint that this pin is low
active.
   * @param val the immediately pin value, true for high, inactive.  */
   public static void setLowactive_reset_Inpin(BlinkingLed_Fpga fpga, boolean
val) {
    fpga.modules.ioPins.input.reset_Pin = val;
  }

}
```

5.3. A module file

All following code snippets comes from the `exmpl_vishiaJ2Vhdl_BlinkingLed/java/ org/vishia/fpga/modules/BlinkingLedCt.java`.

5.3.1. Package and class definition, import and module interface

The principles in Java are already explained in 5.1.1 Package and class definition, import.

Java: module class definition

```
package org.vishia.fpga.exmplBlinkingLed.modules;

import org.vishia.fpga.Fpga;
import org.vishia.fpga.FpgaModule_ifc;
import org.vishia.fpga.stdmodules.Bit_ifc;
import org.vishia.fpga.stdmodules.Reset_ifc;
import org.vishia.fpga.testutil.TestSignalRecorder;
import org.vishia.util.StringFunctions_C;

import java.io.IOException;

public final class BlinkingLedCt implements FpgaModule_ifc, BlinkingLed_ifc {
```

Here the advantage of dependency documentation of the `import` statements are shown. In opposite to C/++ programming where the dependencies are documented with included headers, the used modules are immediately obviously. In C/++ maybe hidden dependencies may be existing because an header can include other headers, and the association between header and implementation files is weak. In Java it is strong, clarified and obviously.

You see that external dependencies exists to one of standard modules, to test utils and to a special class for String preparation, which is used for test output.

The module class implements the `BlinkingLed_ifc` which is a module interface, see Java2Vhdl_Approaches.html#IfcModule, the implementation is shown in the chapter 5.3.5 interface implementation of the module

5.3.2. The references and sub modules of the module

The class definition continues with the

Java: module class references

```
  private static class Ref {

    /**Common module for save creation of a reset signal. */
```

```
    final Reset_ifc reset;

    final BlinkingLedCfg_ifc cfg;

    /**Specific module for clock pre-division. */
    final ClockDivider clkDiv;

    Ref(Reset_ifc reset, BlinkingLedCfg_ifc cfg, ClockDivider clkDiv) {
      this.reset = reset;
      this.cfg = cfg;
      this.clkDiv = clkDiv;
    }
  }

  private Ref ref;
```

The meaning and the writing style of the references is also explained in Java2Vhdl_Approaches.html#aggr.

Also there is explained how the references are set. It is the same example.

A module can also have sub modules to build a deeper tree of modules. It is adequate to the top level. (TODO 2022-05 not tested, should work)

5.3.3. Inner static classes in a module which builds a TYPE RECORD and PROCESS in VHDL

The following code snippet shows one PROCESS as a whole.

Java: module PROCESS class and instances

```
  @Fpga.VHDL_PROCESS public static final class Q{

    @Fpga.STDVECTOR(16) public final int ctLow;
    @Fpga.STDVECTOR(8) final int ct;
    final boolean led;
    int time;
    @Fpga.BITVECTOR(4) final State state;

    Q() {
      this.ctLow = 0;
      this.ct = 0;
      this.led = false;
      this.time = 0;
      this.state = State.nonInit;
    }

    @Fpga.VHDL_PROCESS Q(int time, Q z, Ref ref, Modules modules) {
         Fpga.checkTime(time, ref.clkDiv.q.time, 1);   // for the ce signal,
constrain...
      if(modules.ct_clkDiv.q.ce) {
          Fpga.checkTime(time, z.time, 20);          // check whether all own
process ...
         Fpga.checkTime(time, ref.cfg.time(), 20);// check all signals from the
ref...
        this.time = time;                          // all variables are declared
as ...
         if(ref.reset.res(time, 20)) {           // interface access to assigned
her...
          this.ct = ref.cfg.time_BlinkingLed();
          this.ctLow = 0x0000;
          this.state = z.state;
        }                                       // underflow detection to 111....
```

```
as...
        else if(Fpga.getBits(z.ctLow, 15, 13)==0b111) {    // check only 3 bits
an...
          this.ctLow = 0x61a7;  // 24999;                  // Period 25 ms, hint
c...
           //                      // TODO should convert automatically also a
given ...
           if(z.ct == 0x00) {                       // here a full 0 test with 8
bit ...
                this.ct = ref.cfg.time_BlinkingLed();// interface access to the
reload...
          } else {
            this.ct = z.ct -1;                    // high counter normally count
do...
          }
          this.state = State.fast;
        }
        else {
           this.ctLow = z.ctLow -1;                 // count down automatically
prope...
          this.ct = z.ct;                          // high counter copy the state
(n...
          this.state = z.state;
        }
           this.led = z.ct < ref.cfg.onDuration_BlinkingLed(); //set FF after
compari...
      }
      else {                                      // clock enable ce == false
          this.ct = z.ct;                              // copy the state (not
generated ...
        this.ctLow = z.ctLow;
        this.led = z.led;
        this.state = z.state;
        this.time = z.time;
      }
    }
  }

  public Q q = new Q();
  private Q q_d;
```

This is the core functionality for VHDL and hence explained elaborately:

- The inner class should be marked with the `@Fpga.VHDL_PROCESS` annotation to detect as such from the Java2Vhdl translator.
- Variables in the inner class are generated in a type record with the class name:

```
TYPE BlinkingLedCt_Q_REC IS RECORD
  ctLow : STD_LOGIC_VECTOR(15 DOWNTO 0);
  ct : STD_LOGIC_VECTOR(7 DOWNTO 0);
  led : BIT;
END RECORD BlinkingLedCt_Q_REC;
```

- Variables started with `time` are ignored for VHDL conversion, there are for timing test on Java level.
- `SIGNAL` definitions in VHDL with the `TYPE RECORD` (instances for this `RECORD`) are not created because of the module PROCESS class definitions. Instead, they are defined only if the module class is used, and then with the module(s) name(s). It is possible to have a module more as one time, then also more SIGNAL … RECORD variable are defined. Here for the usage in the `fpgatop/BlinkingLed_Fpga.java`:

  Java: top level part of Modules definition:

```
  public class Modules {
    ....
    public final BlinkingLedCt ct = new BlinkingLedCt(....
```

Vhdl: SIGNAL definition for this PROCESS of a module:

```
SIGNAL ce_Q : ClockDivider_Q_REC;
SIGNAL ct_Q : BlinkingLedCt_Q_REC;
SIGNAL res_Q : Reset_Q_REC;
```

- You can see in the mid of this three signals the module name `ct` from the module definition in `Modules` combined with the name of the Process `Q`. If you have more PROCESS classes in a module, of course you have more TYPE RECORD definitions and also more appropriate SIGNAL definitions.
- The constructor `Q() {…}` is only for Java test. It is (should be) related to the reset behavior of the FPGA: Signals are reset to 0-level. A specific reset behavior is not provided. It is also not full clarified in VHDL, depends on FPGA types. The 0-initialization is anyway applicable. For specific reset behavior the following process should use a reset signal functionality.
- The constructor `@Fpga.VHDL_PROCESS Q(Q z, Ref ref) { …` now describes the process for VHDL translation and also for simulation. It should/could have the following arguments with fixed naming conventions:
- `int time` reference to a time value used for timing test in Java, not used for VHDL
- `z` of the same type: It is the previous state for calculation, comes from the current state in step.
- `ref` reference to the `Ref` class of this module, to access referenced modules.
- `mdl` reference to the whole module class (set with `this` on call, see next chapter).
- `in` reference of a `Input` class of the module for a simple wiring without references.
- `out` reference to an `Output` class of this module, more exact to the `_d` instance (prepared values). This allows the simple wiring and `public` access in Java.
- The PROCESS inner class is `private`, should be used only in this module and not accessed from outside. The elements are ***package private*** (without designation in Java). It can be accessed anyway only in this module because of the `private` nature of the class.
- The content of this constructor is immediately translated to the VHDL PROCESS. To see it for this example look on the translated code:

  Vhdl: PROCESS due to the BlinkingLedCt.Q constructor:

```
ct_Q_PRC: PROCESS ( clk )
BEGIN IF(clk'event AND clK='1') THEN

  IF ce_Q.ce='1' THEN
      IF (res_Q.res)='1' THEN
          ct_Q.ct <= TO_STDLOGICVECTOR(BlinkingLed_Fpga_time_BlinkingLed);
      ELSE
        IF ct_Q.ctLow(15 DOWNTO 13) =  "111"  THEN
            ct_Q.ctLow <=  x"61a7";
            IF ct_Q.ct = x"00"  THEN
                ct_Q.ct <= TO_STDLOGICVECTOR(BlinkingLed_Fpga_time_BlinkingLed);
        ELSE
                ct_Q.ct <=  ct_Q.ct - 1 ;
            END IF;
        ELSE
            ct_Q.ctLow <=  ct_Q.ctLow - 1 ;
        END IF;
      END IF;
```

```
       IF ct_Q.ct < BlinkingLed_Fpga_onDuration_BlinkingLed  THEN ct_Q.led  <=
'1'...
  ELSE
  END IF;
END IF; END PROCESS;
```

- Variable with `time…` are ignored for VHDL, they are for timing checks in Java.
- Variable which are set with the same `z.` variable are ignored for VHDL because this is the standard behavior for VHDL: Not assigned variables preserve there values. But that assigments are necessary for Java.
- Because all variables should be `final` a complete unique assignment of all variables is necessary for Java. This helps preventing errors on forgotten not clarified functionality:
- Because you should write `this.var = z.var;` in Java it is clarified in the Java source that this is an unchanged value. The place and route for the FPGA can use either the own Q output of the FF for the logic, or can work with the CE (clock enable) input for the FF groups.
- You cannot forget variables, because Java checks setting of all `final` variables.
- Tip for writing sources with final variables: You can firstly remove all `final` keywords, set it one after another. Then the error messages for missing final are obviously step by step. You can set an assignment `this.var = z.var;` in all branches, then set the variable to `final`, then you get an error message on this positions where the variable is defined twice. Remove it and think about correct assignments due to the requested logic. At least you have all variables `final` and no errors.
- If you use already assigned `this.` variables on the right side for an assignment, this would be the value of the D-input of an FF. VHDL suggests using an internal variable for that. In the moment (2022-05) this is not regarded, but can be implemented in the Java2Vhdl translator.

  How statements and expressions are translated, see Java2Vhdl_TranslatorInternals.html.
- At last in the Java example with `private Q q = new Q();` and `private Q q_d;` two references for instances are defined.
- The `q` reference must have the same name as the PROCESS class, only start with a lower case character. Here it is only one character. This reference is used to access to the RECORD instances in VHDL both on `public` immediately definition of this reference (possible) and also for interface accesses.
- In this example the reference `q` is `private`. This means that you can be sure that there is no direct access from outside, both for the Java source level and for the VHDL record type.
- The `_d` instance is not used for translation and should only accessed in the `step(time)` and `update()` operations. It holds the prepared D-input values of the FlipFlops. It should be anytime `private`
- Note that Java knows for class instances only references (other than in C/++). All is a reference. The instances are organized in the heap.

5.3.4. step(…) and update() operations

The step and update in a module should call the process execution. It is not used for the VHDL translation, but for the test.

Java: module class step and update

```
  @Override
  public void step(int time) {
     if(this.ref.clkDiv.q.ce) {               // speed up simulation, only on ce
the ...
      this.q_d = new Q(time, this.q, this.ref, this.modules);
```

```
      this.vhdlink_ram_d = new Vhdlink_ram(time, this.modules.ram, this.modules,
t...
      this.ramUse_d = new RamUse(this.ramUse, this);
    }
  }

  @Override
  public void update() {
    this.q = this.q_d;
    this.vhdlink_ram = this.vhdlink_ram_d;
    this.ramUse = this.ramUse_d;
  }
```

See also Java2Vhdl_Approaches.html#stepupd

You see here how does it work:

- `step(time)` creates new instances of all process variables with new combinatoric values and stores it in the `private …_d` instance. Therefore, it is blocked for third-party access. Other modules should use only the Q-state of the FlipFlops to preserve the exact timing. All `step(time)` routines of all modules are executed firstly before `update()`
- `update()` stores all prepared D-values in the real used instances for all immediately and interface accesses.

  It means, any step in Java creates a new instance. The instance is allocated in the heap. If you have 120000000 steps for the BlinkingLed example, the Java simulation creates 120000000 instance of any PROCESS class type for any used module. The simulation runs on a Notebook with normal modern equipment approximately 5 seconds, no more. It needs approximately 0.3 GByte RAM (measured with the Task Manager on Windows-10). It means Java can proper deal with this request.

  If sub modules are present, of course this should be also called here.

5.3.5. interface implementation of the module

The next code snippet shows two accesses:

Java: module class interface implementation and simple access operation

```
  /**Implementation of a given interface which is also fulfilled by this module.
   * @return value for a led which should be blinking.
   */
  @Override public boolean ledBlinking() { return this.q.led; }

  /**This is an example for an access operation only for this module,
   * without abstraction of using an interface.
   * Hence it is only defined for this module, not with a universal interface,
    * it should be used especially for necessary test outputs wich have only
meanin...
   * not for input interfaces for other modules.
   * @return the counter.
   */
  @Fpga.GetterVhdl public int ct() { return this.q.ct; }
  //@Fpga.BITVECTOR(8) //TODO evaluate it
```

The first operation with the annotation `@Override` overrides the declaration in the `BlinkingLed_ifc`, see chapter [mdlclass]. It is used on the top level of the FPGA for the outputs in the `update()` operation of the top level in the first line, see code snippet below.

The second operation with the annotation `@Fpga.GetterVhdl` is an access operation without an interface. It means it cannot be used as univeral access, only for this module. Hence it is not

proper for an input reference of another module, because then another implementation and therefore access via an interface is necessary. But this operation is usable both for specific test outputs on the FPGA and to build output values only for the module specific implementation. It is used on the top level of the FPGA for the outputs in the `update()` operation of the top level in the last line:

Java: top level class using interface implementation and simple access operation for output

5.3.6. interface agents or access in a module

The so named ***interface agents*** are anonymous class definitions as interface implementation to access data in this module. They are usable for referencing.

Java: module class interface agents/access

```
public @Fpga.IfcAccess Bit_ifc getLedFast = new Bit_ifc ( ) {
  @Override public boolean getBit() {
    return Fpga.getBits(BlinkingLedCt.this.q.ct, 2,0) == 0b000;
  }
};
```

See the example in the top level, chapter 5.1.4 interface agents in the top level In this case the interface access is used for the output, see the last code snippet in the chapter above: 'top level class using interface implementation and simple access operation for output'.

The implementation accesses the `ct` variable in the `q` instance of the module class. The writing style `BlinkingLedCt.this.` means in Java the access to the (named) environment class. For debugging this reference is shown as `this$0` whereas `this$1` etc. are higher level environment classes. In Java this can be omitted, but then it is lesser obviously. For the Java2Vhdl translator it is (yet 2022-05) necessary to write.

5.3.7. test output

The module source.java is continued with

TestSignalRecorderHead

Java: module class TestSignalRecorder class definition

```
public class TestSignals extends TestSignalRecorder {
  ....
}
```

This class is not used for the VHDL generation, but used for test. It is explained in the next main chapter.

5.4. Instantiate of entities from other VHDL files (PORT MAP)

Till now this description suggest using only one VHDL file. There are some modules, each module is a java class, but all modules are flattened in the only one generated VHDL file.

But VHDL knows by itself also a hierarchy of modules.

5.4.1. How to use other VHDL files

To include another module (as submodule) respectively instantiate another entity in the own VHDL file (wording of some VHDL descriptions) two steps are necessary:

Definition of the entity header

Written after `ARCHITECTURE` and before `BEGIN`, in the definition part:

```
COMPONENT ExmInclVhdl
PORT (
```

```
  Clock: IN STD_LOGIC ;
  ClockEn: IN STD_LOGIC ;
  Reset: IN STD_LOGIC ;
  WE: IN STD_LOGIC ;
  Address: IN STD_LOGIC_VECTOR(6 DOWNTO 0) ;
  Data: IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  Q: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
 );
END COMPONENT;
```

This definition should be the same as the `PORT` definition in the used VHDL file, which should be present on routine process in the adequate FPGA design tool. A system of header files as in C/++ which contains the declarations and data definitions for both, the implementation (own compilation unit) and usage (other compilation unit) seems to be not present in VHDL.

Instantiation of the other VHDL module as sub module

Written after `BEGIN`, in the implementaion part:

```
ct_Ram_vhdlMdl: ExmInclVhdl
Port MAP(
  Address =>   ct_Q.ctLow(6 DOWNTO 0) ,
  Clock =>  TO_STDULOGIC(clk) ,
  ClockEn =>  TO_STDULOGIC(ce_Q.ce) ,
  Data =>   ct_Q.ctLow(8 DOWNTO 1) ,
  Reset =>  TO_STDULOGIC(res_Q.res) ,
  WE =>   ce_Q.ct(0) ,
  Q => ct_Ram.ramData
 );
```

You can instantiate the same entity (VHDL module) more as one time. First a identifier of the instance, after colon the type identifier, following from the assignments of signals of the PORT of the instantiated module to internal signals of this module. All input signals should be associated, possible as shown in the example with type conversion and bit selection.

The outputs should be assigned to only one type matching variable. It is not necessary to assign (use) all outputs.

The access to the output of the module is done with access to this assigned variables.

Note, that this are not '***assignments***' but signal connections. For the FPGA content it is the same to access the connected variable as the element in the instantiated VHDL module. This is other than calling a funtion in C where this are often move machince code operations.

5.4.2. User stories for modularity with VHDL files

The Java2Vhdl translator allows modularity in Java and generates only one VHDL file as input for place and route. This is sufficient. But some reasons may be given to include more VHDL files.

VHDL file describes specifics in the FPGA such as RAM, controller part

It is familiar to select specific features of a given FPGA, for example a RAM (as in the example above), or other special functionality. This is often tool guided, and produces after selection of the feature a VHDL file. The content of this VHDL file is not intended for further editing, but it is the input for the place and route. Now such VHDL files should be included in the Java2Vhdl created ones.

Using given solutions (legacy, from other colleagues or departments)

Using the Java2Vhdl approach is a specific edition. It presumes experience with the Java language, and also being open to this new idea. It cannot presumed, that all colleagues and other departments does the same. The common commitment is VHDL, traditional and proven. A project can be elaborately, so that the work of many people should match together.

Hence using other work based on VHDL is the first approach. Any department produces a VHDL input for modules, and the root VHDL for the FPGA uses it. How to produce this VHDL input, using Java2Vhdl, any other tool or manual written, is not important.

5.4.3. Module (class) in Java for given VHDL files

The first question is, it should be able to simulate in Java, whereas the simulation may not map the full functionality of the given Vhdl file, only the interface (black box thinking) should be fulfilled.

Following the example with the RAM, the class as representer for the given VHDL file starts as following:

```
//file: /exmpl_vishiaJ2Vhdl_BlinkingLed/java/org/vishia/fpga/exmplBlinkingLed/modu...
/**This is an example class for an included immediately VHDL.
 * In this case it presents a RAM Block in a XO2 Lattice FPGA.
 * The Java class presents the exact same interfaces, but a Java-specific implemen...
 * @author Hartmut Schorrig
 *
 */
@Fpga.VHDL_MODULE ( vhdlEntity = "RAM_SpiRamSel" ) public class RAM_SpiRamSel  imp...

  public static class Input {
    public @Fpga.STD_LOGIC boolean Clock;
    public @Fpga.STD_LOGIC boolean ClockEn;
    public @Fpga.STD_LOGIC boolean Reset;
    public @Fpga.STD_LOGIC boolean WE;
    public @Fpga.STDVECTOR(7) int Address;
    public @Fpga.STDVECTOR(8) int Data;

  }

  public static class Output {
    public @Fpga.STDVECTOR(8) int Q;
  }
```

6. The example Blinking LED, view to Java sources in respect to test on Java level

If you want to separate test files (with a lot of complicated test cases) from the sources, you can also place the test Java files in

```
src
 +-test                                        use the test sub folder
   +-java
      +-testJava_YourComponent
         +-package/path/your/component         This should be your own package pat...
            +-test/*.java                      only for test on Java level.
```

But for this simple example the test class is part of the same Java source tree:

```
src
 +-main
   +-java
      +-srcJava_FpgaExmplBlinkingLed
         +-org/vishia/fpga/exmplBlinkingLed
                              +-test/*,java    only for test on Java level
```

The test classes offers test cases/pattern and checks.

The source code of the modules offers Test output signal generation functionality which are universal usable for the tests.

6.1. The main test source

6.1.1. Class definition and instances to test and used for test

Look firstly to that `org/vishia/fpga/exmplBlinkingLed/test/Test_BlinkingLed.java` which contains the `main` start routine:

Java: main test routine class definition

```
package org.vishia.fpga.exmplBlinkingLed.test;

import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

import org.vishia.fpga.exmplBlinkingLed.fpgatop.BlinkingLed_Fpga;
import org.vishia.fpga.exmplBlinkingLed.fpgatop.BlinkingLed_IoAcc;
import org.vishia.fpga.testutil.CheckOper;
import org.vishia.fpga.testutil.TestSignalRecorder;
import org.vishia.fpga.testutil.TestSignalRecorderSet;
import org.vishia.util.TestOrg;

public class Test_BlinkingLed {

  BlinkingLed_Fpga fpga = new BlinkingLed_Fpga();
```

You see on the depedencies (`import` statements) that the test does not need knowledge about the modules. This is because the test signals are built independently in the modules, the access is via the `TestSignalRecorder` interface.

On top of the test class the `BlinkingLed_Fpga fpga = new BlinkingLed_Fpga();` FPGA top level is instantiated one time. It is also possible to instantiate more as one FPGA to check interaction. It is also possible to instantiate here some simulation replacements for environment hardware, all what's necessary.

If this is a module test class, then the module(s) should be instantiated here with their test environment (test bed).

6.1.2. Instantiate a horizontal output recorder

The horizontal output recorder is an instance of ../../docuSrcJava_vishiaFpga/org/vishia/fpga/testutil/TestSignalRecorderSet.html.

Java: output recorder instance and definiton in start of a test routine

```
  TestSignalRecorderSet outH;


  void test_All ( TestOrg testParent) throws IOException {
    this.outH = new TestSignalRecorderSet();
    this.outH.registerRecorder(this.fpga.modules.ct.new TestSignals("ct"));
    this.outH.registerRecorder(this.new TestSignals("io"));
    this.outH.clean();
```

The instance should be created new in any test routine for that test outputs. After them the desired TestSignalRecorder instances of the modules should be added. One module can have more as one instances of `TestSignalRecorder`. They are used to compile the desired output information. For different test cases you can get just different output information.

The detailed content of the output information is determined in the modules itself, see the chapter 6.2.1 Determination of information to record for output horizontal.

6.1.3. Organization of a checked test

The next line shows only the creation of a test check support class, see org.vishia.util.TestOrg.

Java: Instance of TestOrg in the test routine

```
    TestOrg test = new TestOrg("testTxSpe", 6, testParent);
```

This class helps to organize some tests with a concise output for automatic test case evaluation.

6.1.4. Initialize stimuli (signals) for the simulation

This are the initials settings of all input pins. It can be set either immediately or via interface operations.

Java: Initialization stimulis / signals

```
     BlinkingLed_IoAcc.setLowactive_reset_Inpin( this.fpga, true);            //
res...
```

For this example only the reset pin is used. For the test it is hold to inactive high during the whole operation. The functionality of the reset pin itself can be tested with another programmed test, or is already tested using the `reset` module in other contexts. Hence, it is not important here.

6.1.5. Run the simulation for this test case

In the simulation loop all FPGA simulations and also conditions of inputs to the FPGA which are not constant (depends on outputs, own stimuli signals) should be calculated. The basic step time is the clock.

Java: statements to run the test

```
    int time = 0;
    while(++time < 120000000) {
      this.fpga.step(++time);
      this.fpga.update();
       if(this.fpga.modules.ce.q.ce) {              // speed up simulation, only
on ...
        this.outH.addSignals(time);
      }
    }
```

To simulate the FPGA(s) the appropriate `step(time)` one after another and after them all `update()` should be called. After `update()` the preparation of inputs to the test bed can be gotten from FPGA outputs of this step time. The test bed calculations can be done then in the next loop as first before the `fpga.step()`. Here nothing else is to do.

Also after all updates the `outH.addSignals(time)` is called to record all signals in all modules, organized in the `outH` of type `TestSignalGeneratorSet`, see sub chapters above.

Here it is shown that the `addSignals(…)` is conditionally called only `if …ce`. This means that the output signals are not resolved down to the individual clock level. The `ce` is the central clock enable for the most module functions, and only outputs regarded to this `ce` are in focus. This may be important for longer runinng tests, such as here for more as 100 million steps. Usual, if the signals on any clock edge are interesting, the simulation time is lesser and focused on module functionality. This is not a principle approach, it is sensible.

6.1.6. Output recorded signals

The output of the recorded signals is only a simple writing of all StringBuilder registered in the `TestSignalRecorder` in the order of registering. The output written to a text file can be edited for presentation also afterwards.

Java: output signals of horizontal recording usual for manually elaboration

```
    this.outH.output(System.out);
    Writer fout = new FileWriter("build/test_BlinkingLed_Signals.txt");
    this.outH.output(fout);
    fout.close();
```

It is also possible to produce output signals in a file with vertical recording (one line is one time stamp, signals in the line). This is not shown here yet. Especially this format can be converted to a signal graphic similar as the output of FPGA simulation tools, with the possibility of zoom etc.

6.1.7. Automatically evaluation of test results

In the next code snippet two output lines of the horizontal signal recording are checked. The special function `checkOutput(…)` as specific implementation for the check tests the pattern in the line: The signals should be a minimal and a maximal number written one after another. The evaluation starts after the first change, not from beginning, because elsewhere the minimal number of characters may be violated.

Java: evaluation of test results for automatically test

```
      CheckOper.CharMinMax[] checkLedA = { new CheckOper.CharMinMax('_', 200,
9999),...
     //Note: a shorter low phase of LedB occurs on overflow of the high bit
counter.
      CheckOper.CharMinMax[] checkLedB = { new CheckOper.CharMinMax('_', 26,
200 ), ...
       String error = CheckOper.checkOutput(this.outH.getLine("io.ledA"), 20,
checkLe...
     test.expect(error == null, 5, "LedA off 200.. chars, on 200..300 chars" +
(err...
    error = CheckOper.checkOutput(this.outH.getLine("io.ledB"), 20, checkLedB);
     test.expect(error == null, 5, "LedB off 180..200 chars, on 26 chars" +
(error ...

    test.finish();
```

This test is programmed done. It produces with the following `test.finish()` a concise state "***Test ok or ERROR***" which can be simple automatically evaluated. It means this and more tests runs, and the result should be "***ok***" for all tests.

With this approach it is possible to test whether the functionality is proper (for the test cases) after some changes in the sources. May be specific changes were done, and the results were checked manually. But now the question arises: Has the change side effects? Is everything else still running?

To answer this, such tests, which can be elaborately, are important.

Of course, the test cases, the stimuli and the evaluation can also be faulty or incomplete. In conclusion, the effort for the test cases is often higher than the effort for the functionality itself. It depends on the type of use (long-term or intermittent) whether this expense is appropriate.

6.1.8. The main routine for test

The main routine is called immediately from command line level. It creates the own class, and starts some tests.

The top level `TestOrg` assembles all children `TestOrg` for more tests and a summarized evaluation.

Java: TestSignalGenerator implementation complete for conditional output

```
  public static void main(String[] args) {
    Test_BlinkingLed thiz = new Test_BlinkingLed();
    TestOrg test = new TestOrg("Test_BlinkingLed", 3, args);
```

```
    try {
      thiz.test_All(test);
    }
    catch(Exception exc) {
      System.out.println(exc.getMessage());
      exc.printStackTrace();
      test.exception(exc);
    }
    test.finish();
  }
```

Generally a `try … catch` should be present to catch non expected exceptions. With the stack trace the reason may be able to find also in non debugging environments. For example any uninitialized stuff may cause a null pointer exception which is not necessarily a flaw in the logic, only a small programming mistake. In general, Java's try-catch capability is well proven.

6.1.9. Test output preparation for the main level

This is the contribution of test signal output from the main test or the whole fpga to view and also for automatically check the results. This top level output test results is not implemented in the top level FPGA java file (here `BlinkingLed_Fpga.java`), instead in the test routine, to unburden the top level java file. The access to the Pin signals is anytime possible in a public way, in opposite to the situation in the modules, see next chapter.

The code snippet shows the complete `TestSignalRecorder`

Java: TestSignalRecorder implementation complete for conditional output

```
  class TestSignals extends TestSignalRecorder {

    StringBuilder sbLedA = new StringBuilder(500);
    StringBuilder sbLedB = new StringBuilder(500);

    private char cLedA, cLedB;

            /**This   instance   should   be   added   on   end   using   {@link
TestSignalRecorderSet#reg...
        * because it decides adding an information only depending of other
SignalReco...
     * @param sModule name, first part of line identifier
     */
    public TestSignals(String sModule) {
      super(sModule);
    }

    /**cleans all StringBuilder line and registered it. */
    @Override public void registerLines ( ) {
      super.clean();
      super.registerLine(this.sbLedA, "ledA");
      super.registerLine(this.sbLedB, "ledB");
    }

      @Override public int addSignals ( int time, int lenCurr, boolean bAdd)
throws ...
      BlinkingLed_Fpga fpga = Test_BlinkingLed.this.fpga;
       if(bAdd) {                                    //only calculate state if
anothe...
        if(fpga.modules.ioPins.output.led1) {
          if(fpga.modules.ioPins.output.led1) {
          }
        }
        this.cLedA = fpga.modules.ioPins.output.led1 ? 'A': '_';
```

```
        this.cLedB = fpga.modules.ioPins.output.led2 ? 'B': '_';
        this.sbLedA.append(this.cLedA);
        this.sbLedB.append(this.cLedB);
        return this.sbLedA.length();
      } else {
        return 0;  // no own contribution to length, regard add, sub ordinate.
      }
    }


    /**This operation is here overridden to add the character of the led state
ins...
     * It will be called in {@link TestSignalRecorderSet#addSignals(int)} after
in...
     * Which character is added, this is determined by addSignals above in this
cl...
    @Override protected void endSignals ( int pos) {
      while(this.sbLedA.length() < pos) { this.sbLedA.append(this.cLedA); }
      while(this.sbLedB.length() < pos) { this.sbLedB.append(this.cLedB); }
    }

  }
```

- The Recorder should define StringBuilder for each test signal for store and output, here two lines.
- The `char cLed…` are only locally, it stores the values for complete the lines in `endSignals(…)`.
- The constructor has the name of the module as parameter. The hint in the comment is for the order, see chapter 6.1.2 Instantiate a horizontal output recorder.
- The `clean()` operation should be overridden in the shown kind, should register the locally `StringBuilder` calling `registerLine(…)`.
- The `addSignals(…)` operation is subordinate here, it writes the state of the Led output pins, but only if another module, here it is especially `BlinkingLedCt` has written something. This is sufficient, because the Led signal is slow and long. The other module produces enough time stamps to see what happens. But such decisions depends heavily on the test case. Hence the output should tuned to the test case. On the other hand the test cases are often similar in output requirements.
- The `endSignals(…)` is here overridden to produce the repeated character for the LED. The standard implementation writes a space to separate hexa values, which is often the requested use case.

6.2. Test support in modules

The decision which signals are to be output for the display of the test results can only be made in a module itself, since only the module knows its own signals. This is often seen in a different way, namely that the test case itself should determine which signals should be displayed. However, this assumes that the signals are known and not changed. It also violates the private/public encapsulation of content in modules.

But of course the decision which signals to display may depend on the particular test case. For such possibilities more as one `TestSignalRecorder` inner class can be implemented. The implementation can be done due to a specialized test case, but should be designed in a more universal way. The question is which signals are interesting, for a detailed look at the module - or to get an overview. The structure of the module, not the specific opinion of the tester, should be determinative.

6.2.1. Determination of information to record for output horizontal

For this example a specific detail is programmed in the next code snippet: It is the elaborately view to the time spread where the low-bit counter overflows and triggers the high-bit counter. All other occurrences are irrelevant, they are clarified. This is the interesting point in the module, and may be also the interesting point for a global test view.

Java: TestSignalGenerator addSignals(…) with condition building

```
      @Override public int addSignals ( int time, int lenCurr, boolean bAdd )
throws...
      BlinkingLedCt thism = BlinkingLedCt.this;
      int zCurr = this.sbCt.length(); // current length for this time
      int zAdd = 0;                   // >0 then position of new length for this
time
       if(thism.ref.clkDiv.q.ce) {       // because the own states switches only
wi...
        if(thism.q.ctLow == 1) {        // on this condition
          this.wrCt = 5;               // switch on, write 5 steps info
        }
          if(--this.wrCt >0) {             // if one of the 5 infos shouls be
written:
             StringFunctions_C.appendHex(this.sbCtLow, thism.q.ctLow,4).append('
'); ...
                          StringFunctions_C.appendHex(this.sbCt,   thism.q.ct,2);
...
          if(checkLen(this.sbtime, zCurr)) {      // add the time information if
h...
            StringFunctions_C.appendIntPict(this.sbtime, time, "33'331.111.11");
...
          }
              zAdd = this.sbCtLow.length();   //length of buffers for new time
determin...
        }
         else if(this.wrCt ==0) {          // end of the 5 steps, append .... as
sep...
          this.sbCtLow.append("..... ");
              zAdd = this.sbCtLow.length();   //length of buffers for new time
determin...
        }
      }// if ce
                  return   zAdd;                     //   will   be   used   in
TestSignalRecorderSet.addSignals(zAdd)...
    }//addSignals
```

Here the output is triggered if the counter reaches the value 1, back counting before zero-crossing. Details of this are also used to present the common approach in Java2Vhdl_Approaches.html#testOutp.

6.2.2. Store and restore the state of modules as well as the whole simulation state

For the example the following `Store` class is able to find in the common offered `Reset` module:

Java: Store class in a module

```
  /**Stores the state for special tests.
   * You can use this implementation as template for your modules.
   */
  public static class Store extends StateStoreFpga < Reset > {
    final Q q;

    /**Creates a Store instance, which refers the data from the {@link Reset#q}
in...
```

```
        * it is the PROCESS data, able to call after a defined simulation procedure,
     * to resume later exact from this state.
     * @param time The time stamp of the simulation
     * @param src The reference to the module.
     */
    public Store(int time, Reset src) {
      super(time, src);
      this.q = src.q;
    }

    /**Restore the state to the same module, which is used on creation.
        * It is presumed that the {@link Reset#q} instance was not changed meanwhile.
      * However, this is guaranteed if the Application Pattern Style Guide is follo...
     * and also because all members in {@link Reset.Q} are <code>final</code>.
     */
    @Override public int restore() {
      super.dst.q = this.q;
      return super.time;
    }
  }
```

If you look to the description of the base class:

../../docuSrcJava_vishiaFpga/org/vishia/fpga/testutil/StateStoreFpga.html

you find also the pattern for application. This is not done in the simple Example.

The storage of the whole state of the simulation (all modules in the FPGA and also the environment, test bed state) is helpfully if you want to simulate variants starting from a dedicated state. You save the effort to reach the start state again from beginning.

7. Requests for Change (RFC) for the Java2Vhdl tool

This is just an incomplete list, only a simple collection.

It does not include fixes which are errors. It describes generally desired features.

RFC1 It should be checked that `this….value` is never used for VHDL generation in an expression (right side) inside a process.

Why: Access to this accesses the new (not yet set) values. This is possible in Java but not in VHDL. Access to given values in VHDL is anytime mapped to access to the `z..` reference in a `PROCESS` constructor.

Prio: Because it is also possible to look manually to this fact, the prio is less.

Effort: middle (estimated)

RFC2 More conversions, if a BIT_VECTOR is detected used for numeric operations, it should be automatically converted to a STD_LOGIC_VECTOR before usage.

Prio: If not available, the user should taken care about the types. Introducing conversion functions in Java for VHDL is not recommended! Hence if it is desired, should be done.

Effort: not high, think about, do and test.

RFC3 @Fpga.PROCESS and also other should have a string designation. With this designation and an additional config file the translator should regard this parts to Vhdl translation or not.

Why: It should be possible to have more parts in Java, for test, for notice for enhancements, or also for variants, than necessary in the particular VHDL result. This is a little bit similar as conditional compilation in C/C++.

Prio: It is interesting, higher priority.

Effort: middle

RFC4 Usage of `INTEGER` variables

Why: May be interesting for some designs

Prio: Because integer operations are also possible with `STD_LOGIC_VECTOR` types it is lesser prio. Only if a stakeholder desires it. The possibility to define ranges is not related to common practice in other programming languages. The number of bits used is very interesting for FPGA logic. But this is clarified also by `STD_LOGIC_VECTOR`. But maybe a complete functionality should include it.

Effort: middle