# AspectVHDL Stage 1: The Prototype of an Aspect-Oriented Hardware Description Language

Matthias Meier
Technische Universität Dortmund
matthias2.meier@tu-dortmund.de

Stefan Hanenberg
University of Duisburg-Essen
stefan.hanenberg@icb.uni-due.de

Olaf Spinczyk
Technische Universität Dortmund
olaf.spinczyk@tu-dortmund.de

## Abstract

Hardware description languages are a promising field for the application of aspect technology. In a case study with the MB-Lite soft core CPU, which is an open, cycle accurate re-implementation of Xilinx' Microblaze processor, we show that crosscutting concerns in hardware structures actually exist. After discussing the semantic differences between programming languages and hardware description languages, we introduce our first version of AspectVHDL, an aspect-oriented extension of the popular hardware description language VHDL. The evaluation of an aspect-oriented variant of the MB-Lite CPU gives first evidence that using AspectVHDL for the implementation of crosscutting hardware concerns does not induce any costs in terms of chip space.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Design, Languages

***Keywords*** AspectVHDL, Aspect-Oriented Programming, Hardware Description Language

## 1. Introduction

This is a follow up on an earlier ACP4IS/MISS paper that described our motivation and first ideas regarding an aspect-oriented extension of VHDL [3]. Based on experience with empirical studies on Aspect-Oriented Programming [4], our goal is now to develop AspectVHDL in a process that is driven by studies with users. We regard this as crucial as the educational background of hardware developers differs significantly from the background of the typical users of aspect technology. Besides this, the development of such process is interesting research in itself.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*MISS'12,* March 27, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1217-2/12/03…$10.00

In order to "drive" the language development process by empirical studies, AspectVHDL has to be designed incrementally in several *stages*. This paper describes *stage 1*. It is a very simple language extension, because the number of early design decisions has been kept low *intentionally*. We only added well-known AOP concepts, namely *before/after/around advice* and *introductions*, to VHDL. Based on a small case study that we conducted ourselves, we will show in this paper that even this simple language extension is already beneficial and that an implementation will not cause overhead in terms of wasted chip space. Therefore, it is well suited for initial empirical studies. At the end of the paper we will also give a brief overview on ideas for the next two stages. *Stage 2* is planned to address *component and aspect instantiation*, which are special in hardware descriptions, while *stage 3* will introduce further kinds of VHDL-specific join points into the language.

## 2. Motivating Example

The aforementioned case study with the MB-Lite CPU now also serves as a motivating example. The MB-Lite VHDL source code consists mainly of the four components that represent pipeline stages and three packages that contain type declarations and helper functions.

One of the typical use cases for aspects in software is the implementation of optional features. In our other projects with the AspectC++ language we frequently model "minimal extensions" with "extension aspects" as they often crosscut the system's core abstractions [6]. We observed that optional extensions of the MB-Lite processor also tend to crosscut the code base. Figure 1 illustrates the code scattering of three optional hardware accelerator features, namely hardware support for multiplication, shifting, and floating point operations.[1] The code fragments that deal with these extensions can be found in three of five components and in all of the three packages of the MB-Lite processor.

A more detailed look at the VHDL source code (in total 5294 lines) shows that the three optional extensions can be configured with a special VHDL language feature called

[1] An FPU is normally not part of the MB-Lite. We have integrated this feature during an earlier project.

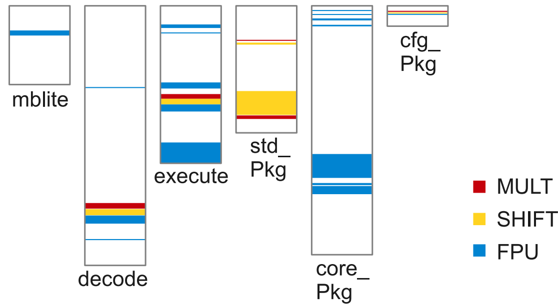


**Figure 1.** Code scattering of optional MB-Lite extensions

*Generics*. A generic is a configuration parameter, which can be used by "if ... generate" statements. This is similar to what most system software developers know as conditional compilation with "#ifdef" in C – a mechanism that is often criticized for causing misconfiguration bugs and maintenance problems if used in larger project settings [7]. We thus regard a modular and easily pluggable implementation of the extensions with aspects as desirable and as a good example to study stage 1 of AspectVHDL.

## 3. Aspects in VHDL

Hardware structures that are described in VHDL follow a strictly hierarchical organization: The *top-level design entity* can contain and connect other *entities*, which in turn can also contain and connect further sub-entities. Each entity can have multiple implementation variants called *architectures*. In the MB-Lite project there is only one architecture per entity. In software terms, we can regard these entity/architecture pairs as the interface and the implementation of components. *Packages* mainly contain reusable definitions of *types*, *functions*, and *procedures* to be used by the entities. For example, types are needed to describe the interface connections of entities or allowed values of internal component states (*signals*). Functions and procedures contain sequences of VHDL statements. They encapsulate reusable logic decisions, calculations, or signal assignments.

It is worth to note here that if a procedure is "called" from within an architecture description, this leads to the instantiation of the hardware structure that implements the procedure. There is no such thing as a function call stack or binary code reuse in VHDL. Nevertheless, on the source code level, there are procedures, which look familiar to software developers. Another interesting property of VHDL is that statement sequences are not necessarily executed one after the other. In the concurrent part of an architecture description all statements are executed in parallel. This is because each statement produces hardware structures. There is no processor that is interpreting instructions – we are building processors here. If a design shall perform actions in a sequential order, this has to be described explicitly as a *process*. Each architecture can define multiple processes.

### 3.1 Join Points

The definition of an aspect-oriented language extension has to start with the join-point model. It determines the events or structures that can be affected by aspect code. In AspectVHDL stage 1 we support the following four kinds of join points:

**Procedures/functions:** All statements in procedures or functions are executed sequentially, regardless whether they are used from within a process or from within the concurrent part of an architecture. This makes them ideal candidates for classic *before/after/around advice*, which changes the dynamic control flow. In contrast to functions, procedures don't have a type and result value. However, they can have *in*, *out*, and *inout* parameters.

**Types:** By modifying types it is possible to introduce structural extensions into various components, because a type is usually used at various points in the code. In stage 1, AspectVHDL supports introductions into *record* types as well as into *enumeration* types.

**Architectures:** As the architecture describes the structure and behavior of the design entities, it has to be possible to introduce additional elements, such as concurrent statement, processes, signals, etc. In AspectVHDL these extensions can be grouped in architecture fragments called *slices* (inspired by AspectC++).

**Process triggers:** A process of an architecture is not started by a call. It has a list of signals, the *sensitivity list*, which can trigger its execution. A process trigger join point is a sensitivity list. Aspects have to be able to extend it.

Even though we can imagine more join-point types (see section 5), this set is sufficient for basic structural and behavioral extensions.

### 3.2 Pointcut Expressions

AOP languages typically contain special syntactic elements to describe pointcut, i.e. sets of join points to be affected by aspects. The pointcut expressions, which are used for this purpose directly reflect the join-point model. The following listing gives some examples of pointcut expressions that we used in our case study:

```
1 within(core_Pkg) and type(ctrl_execution)
2 within(arch of execute) and process(execute_comb)
3 within(* of core) and call(enable(*))
4 architecture (* of *)
```

The pointcut functions *type*, *process*, *call*, and *architecture* are used to identify join points of a particular kind and name. The *process* function is used to identify a process *execute_comb* (line 2). However, at the moment it can only be used to extend the processes' sensitivity list. The *call* pointcut function identifies a procedure or function. In VHDL it makes no sense to distinguish between call and execution – functions and procedures are instantiated for each call anyway. Therefore, we decided that a single *call* pointcut func-

tion is sufficient. As in other AOP languages wildcards can be used for argument or result types (functions only). The *within* pointcut function can be used to select a join point unambiguously. It can be used in combination with types, processes, and functions/procedures as shown in lines 1 to 3. The argument either describes a package (line 1) or an architecture (lines 2 and 3). As architecture names are only unique per design entity, the syntax "*<architecture-name>* of *<entity-name>*" is used. In order to support more complex pointcut expressions we also added algebraic operators *and*, *or*, and *not*. The last pointcut function *args* will be explained in the next section.

### 3.3 Advice Code

AspectVHDL stage 1 supports two types of advice: The *before/after/around advice* can be used to redirect the control flow in the case of a function or procedure call. Introductions can be used to insert a slice into a type, architecture, or process trigger. The following example shows around advice:

```
advice around(...) : within(arch of core)
  and call(enable(*))
  and args(ena_i, mem_i.ena_i, stall) is
      ena_i <= mem_i.ena_i and (not stall);
end advice;
```

This around advice will intercept any call of *enable()* in the architecture *arch* of the entity *core*. Inside the around advice the built-in procedure *proceed* can be used to invoke the original function or procedure. In the example we don't use *proceed* but only assign a signal instead. The identifiers that are used in the advice code body are bound to context information, namely function arguments, with the *args* pointcut function. This is the same mechanism as in AspectJ and AspectC++. The types of the identifiers have to be declared in the argument list of the *around* advice, which is omitted in the listing.

With introductions it is possible, for instance, to add new statements to the declarative part of an architecture, to insert entire processes into the statement part of an architecture, or to extend a record by new element declarations. A slice declaration describes the fragment that is to be introduced:

```
slice ctrl_slice is record
  fpu_op   :  fpu_operation;
end record ctrl_slice;
...
advice slice : within(core_Pkg)
  and type(ctrl_execution) is ctrl_slice;
```

This code fragment shows the extension of a record. A record is a compound type as a struct in C. In the example we extend the record *ctrl_execution* of the package *core_Pkg* by the element *fpu_op* of type *fpu_operation*.

### 3.4 Aspects

As in other AOP languages aspects are used to group *advice* and *slices*. In our case study each CPU extension is implemented by one aspect. Within an aspect it is also possible to use ordinary VHDL statements to declare functions and procedures, types, constants, or signals as well as to instantiate sub-components. The following example shows the skeleton of our FPU extension:

```
aspect FPU is
  type fpu_states is (idle, running, ready);

  advice around() : within(arch of execute)
    and call(execution(*)) is
        ...
  end advice;
end aspect FPU;
```

As aspects can potentially affect all VHDL translation units of the project, there has to be a mechanism to locate aspect definitions automatically and efficiently. We therefore store aspect code only in files with the extension .avhd.

### 3.5 Syntax of AspectVHDL

To summarize our language design and to provide a more precise definition of AspectVHDL stage 1, Figure 2 shows the grammar in Backus-Naur Form. The extension is seamlessly integrated into VHDL. We took over the wordy style of the language and hope that hardware developers will feel familiar. The syntax has to be read as an extension to the grammar presented in IEEE Standard 1076-1993 [8].

## 4. Case Study: MB-Lite Extensions

In order to show that AspectVHDL stage 1 is already a useful in a real-world VHDL project, we refactored the source code of the MB-Lite processor[2], which is a cycle-accurate re-implementation of Xilinx' MicroBlaze processor. The aim was to remove the three aforementioned configurable extensions, to re-integrate them by means of AspectVHDL, and to compare the two MB-Lite variants.

Confronted with the real VHDL code, we found that the implementation lacks the necessary join points for our aspects. For example, the decode stage and the execute stage of the pipeline are implemented in a *very* long process with almost no procedure or function calls. This "spaghetti code" makes it impossible to use aspects for an extension of the pipeline stages. For now we modularized the code manually by moving code fragments from the long process descriptions into procedures. In the future we have to rely on the extensions that are planned for stage 3 in combination with guidelines for aspect-aware VHDL code.

Based on the well-modularized variant of the MB-Lite code, we removed the three extensions and added them again as three well-separated aspects. This new source code completely avoids the code scattering problem that was shown in Figure 1. The "aspect weaving" was performed manually as our weaver implementation is not yet fully finished. For advice on procedure calls it was mainly necessary to add a few helper procedures. Introductions were woven in a straightforward manner. It was very helpful that in VHDL the order

[2] Freely available from `http://opencores.org/project,mblite`

```
aspect ::=
  aspect identifier is
    aspect_declaration
  end [ aspect ] [ aspect_name ];

aspect_declaration ::=
  { aspect_declarative_item }

aspect_declarative_item ::=
    advice
  | slice
  | subprogram_declaration
  | subprogram_body
  | aspect_type_declaration
  | constant_declaration
  | signal_declaration
  | component_declaration

advice ::=
    subprogram_advice
  | introduction_advice

introduction_advice ::=
  advice slice : pc_expr is slice_name

subprogram_advice ::=
  advice advice_type : pc_expr is
    subprogram_statement_part
  end [ advice ];

advice_type ::=
    before(formal_parameter_list)
  | after(formal_parameter_list)
  | around(formal_parameter_list)

slice ::=
  slice identifier is aspect_slice_def;

pointcut_expr ::=
    pointcut_expr and pointcut_expr
  | pointcut_expr or pointcut_expr
  | not pointcut_expr
  | (pointcut_expr)
  | builtin_pointcut_function

aspect_type_declaration ::=
  type identifier is aspect_type_def;

builtin_pointcut_function ::=
    call(subprogram_pat)
  | architecture(name_pat of name_pat)
  | type(name_pat)
  | process(name_pat)
  | within(name_pat [ of name_pat ])
  | args(parameter_pat)

aspect_slice_def ::=
    aspect_architecture_definition
  | aspect_type_definition
  | aspect_process_definition

aspect_architecture_definition ::=
  architecture is
    architecture_declarative_part
  begin
    architecture_statement_part
  end [ architecture ]

aspect_type_def ::=
    enumeration_type_definition
  | record_type_definition

aspect_process_definition ::=
  process(sensitivity_list)
```

**Figure 2.** The AspectVHDL extension to the VHDL grammar

| | VHDL | mod. VHDL | AspectVHDL |
|---|---|---|---|
| max. Freq. (MHz) | 17.366 | 17.366 | 17.396 |
| Slices | 8,712 | 8,530 | 8,479 |
| Look-Up Tables | 13,671 | 13,302 | 13,277 |
| Flip-Flops | 2,883 | 2,884 | 2,849 |

**Table 1.** Synthesis Results

of declarations or statements is rarely relevant. For example, statements can be simply inserted at the end of the concurrent part of the architecture, because everything runs in parallel anyway.

Table 1 shows a comparison of the required resources for the unmodified VHDL project, the modularized implementation, and the "woven" AspectVHDL project. The results were obtained with the Xilinx ISE 10.1.03 using default settings for a Xilinx Spartan-3E XC3S1200E FPGA (speed grade -4). All three variants are fully functional. The maximal possible frequency is quite similar for all three projects with a small advantage for the AspectVHDL project. The differences between these frequencies are hard to explain because of the closed synthesis tool from Xilinx. However, there is evidence that the use of aspects will not necessarily have a negative effect on the maximal supported frequency. The same observation holds for FPGA resources, such as slices, look-up tables, and flip-flops.

## 5. Outlook

In this section we introduce our early plans and ideas for the next stages of AspectVHDL.

***Stage 2: Instantiation Model Extensions*** The pointcut expression language of AspectVHDL stage 1 only supports matching of architectures, but not to distinguish their *instances*. As a consequence, aspects affect all instances of an architecture in the same way. In our case study context this means that if we integrate the MB-Lite processor into a multiprocessor system, all processors would have the same extensions. This is clearly a flaw. A similar problem exists in AspectJ or AspectC++ programs, which also don't support advice *per object*, but it is less dramatic, because the code of a class, which is affected by aspects, is not copied for each object.

For AspectVHDL stage2 we therefore have to think about pointcut expressions that can select *instances* of hardware components. This could be done by the use of instantiation labels and the instantiation path, which unambiguously identify each instantiated component. The following example shows a possible syntax:

```
advice slice : instance(mblite1.execute1)
  is exec_slice;
```

Here the pointcut function *instance* selects the MB-Lite with the label *mblite1* and the sub-component *execute1*. For an implementation of this feature in an aspect weaver it would be necessary to duplicate the architecture description.

***Stage 3: Join Point Model Extensions*** For stage 3 of AspectVHDL we want to integrate more fine-grained and hardware-specific join point types. For example, join points that could be useful to modify the transitions of finite state machines. Finite state machines are frequently used in hardware designs. They are typically implemented by a case statement as shown in the following listing:

```
type fpu_states is (idle, running, ready);
signal state : fpu_states := idle;
case state is
  when idle =>
    if (x = '1') then
      state <= running;
    end if;
  when running =>
    ...
  when ready =>
    ...
end case;
```

Here the finite state machine has three states *idle*, *running* and *ready*. The transition between the two states *idle* and *running* is performed by a signal assignment ("*signal* <= *value*"). If we wanted an aspect to affect a state transition, e.g. jump to *ready* instead of *running*, a fine-grained join point type would be needed to select the signal assignment. An AspectJ-style "set join point" would be a solution, but we also think about higher-level state machine abstractions.

A second idea for stage 3 is to add the input and output signals of components to the set of join point types. The reason is that due to the strict hierarchy in VHDL designs it is extremely tedious to route signals between arbitrary components or between a component and the outside world. It also leads to a lot of redundant code. From the AOP perspective, this is a homogeneously crosscutting concern and the new join point type would allow us to implement this kind of signal routing in a very easy and modular way.

## 6. Related Work

Among others, Dharbe and Medeiros [2] applied Aspect-Oriented Programming in combination with SystemC. SystemC is a high-level hardware description language, which is mainly used for simulation. Tools for hardware synthesis from SystemC code are still not widely used.

In [5], the authors applied the concept of Feature-Oriented Programming to Verilog. Their work has similar objectives, but we believe that Aspect-Oriented Programming is the more powerful approach and, thus, better suited to explore new ways to modularize hardware descriptions.

The *Aspect Described Hardware Programming Language* (ADH) [1] is a domain-specific language that can be translated by a compiler to VHDL code. This is powerful approach, but we think that an aspect-oriented extension of VHDL has more potential for wide-spread use than a completely new language.

In [9], an aspect-oriented approach for the verification of hardware systems is presented by means of the "e" hardware verification language. With respect to AOP features "e" is quite limited. For example, each advice can affect only a single join point. There is no "quantification".

## 7. Conclusions

Even though a hardware description language looks like an ordinary programming language *syntactically*, there are many important semantic differences. Examples are the parallel execution of statement sequences in the concurrent part of an architecture or the lack of a procedure call stack, which means that all procedures are always expanded inline. A non-technical difference seems to be the educational background of the "programmers": Procedures are well-defined in the VHDL-standard and supported by the synthesis tools we tested, but the MB-Lite developer was no exception in *not* making use of this modularization feature.

For our aspect-oriented language extension of VHDL we therefore cannot stop with stage 1, i.e. a language which only copies well-known concepts from the AOP software domain. It is important to take the human factor into account during the language design process. This motivates empirical studies, which we are currently planning. It is also crucial to come up with design guidelines that lead to better modularized code and more potential join points at the same time. In order to convince VHDL developers it will also be necessary to design hardware-specific AOP abstraction such as the transparent routing of signals mentioned in Section 5.

## Acknowledgments

This work was partly supported by the German Research Council (DFG) under grant no. SP 968/4-1 and within the Collaborative Research Center SFB 876, project A4.

## References

[1] A. Bainbridge-Smith and S.-H. Park. ADH: An aspect described hardware programming language. In *Proc. of FPT*, 2005.

[2] D. Déharbe and S. Medeiros. Aspect-oriented design in SystemC: Implementation and applications. In *Proc. of SBCCI*, 2006.

[3] M. Engel and O. Spinczyk. Aspects in hardware - what do they look like? In *Proc. of ACP4IS*, 2008.

[4] S. Hanenberg, S. Kleinschmager, and M. Josupeit-Walter. Does aspect-oriented programming increase the development speed for crosscutting code? An empirical study. In *Proc. of ESEM*, 2009.

[5] Y. Jun, T. Qingping, L. Tun, and C. Guorong. FeatureVerilog: Extending verilog to support feature-oriented programming. In *Proc. of IPDPSW*, 2011.

[6] D. Lohmann, W. Hofer, W. Schröder-Preikschat, and O. Spinczyk. Aspect-aware operating system development. In *Proc. of AOSD*, 2011.

[7] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proc. of EuroSys*, 2011.

[8] The Design Automation Standards Committee of the IEEE. *IEEE Standard 1076-1993: VHDL*. 1993.

[9] M. Vax. Conservative aspect-oriented programming with the e language. In *Proc. of AOSD*, 2007.