University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Computer Architecture for Embedded Systems group

Master thesis

A basis for the next VHDL revision

B.J.M. de Jong

August 22, 2014

Abstract

Over the years, the number of transistors that is available to hardware designers has exponentially increased. Because of this it is very important to have an effective hardware description language. One of the most important hardware description languages is the VHSIC hardware description language (VHDL). To cope with the increased number of transistors one could ask the question: "What language constructs can be added to the VHSIC hardware description language to keep it effective and relevant for the future?". To answer this question, a basic understanding of how compilers work is needed. The compiler starts by lexing files and then parsing the output of the lexer into an abstract syntax tree. Once the abstract syntax tree is made it can be traversed. Also a good understanding of how simulation- and synthesis-tools work is important. Simulators work with two types of events: timed events and delta events. The timed events cause signal changes and each timed event is followed by one or more delta events to bring the system back into a stable state. Synthesis tools are less predictable than simulation tools, since it is not always clear what exact gate level description the synthesis tool will come up with.
The language enhancements that have been added to the language are: An independent compilation order of the input files, having no separation between the declaration area and the body area of architectures, subprograms, entities and blocks, being able to assign signals immediately after declaring them, being able to overload the assignment operator so less type transformation functions are needed, having namespaces that encapsulate all language elements and a new and fast attribute system.
To be able to use these enhancements, a compiler is created that can compile the language with enhancement, back to the VHDL 2008 standard. The compiler uses five passes. During the first pass the VHDL files are linked and parsed. In the second pass all the declaration elements are collected. During the third pass these elements are linked together. In the fourth pass the expressions are reparsed since they are context sensitive. Finally the output of the compiler is generated during the fifth pass.
Next the compiler is tested with a use-case design of Astron. With some small changes, the design can be compiled with the new compiler, and can be tested with the test-bench that was also provided by Astron. After this, parts of the code have been rewritten with the new language constructs and the output was tested successfully with the same test-bench.
A multi-pass compiler gives a lot of freedom in terms of what can be added to the language. We can conclude that the new features improve the language, but that they are hard to implement on the existing compilers. In general the conclusion can be made that due to the design of the VHDL language it is hard to add new features to it.

Foreword

In September 2013 I started my master thesis at the CAES group. Though the initial idea for the master assignment was to create a complete new hardware description language, it soon became clear that this was far from feasible in the given time frame. Instead, an improvement of VHDL was chosen. The compiler I made for this thesis was a software project that was more than ten times larger than I have ever done before. The compiler consists of over a hundred classes, and it took quite some cups of coffee to achieve this. Also the help of my main committee members, Bert Molenkamp and André Kokkeler, has helped to give direction to achieve the final result. My thanks also goes out to Philip Hölzenspies for sharing his broad knowledge on type systems and Harm-Jan Pepping of Astron for the large amount of patience he showed while getting the test-bench working. Finally I would like to mention that though VHDL is an acronym we will use it as a noun throughout this thesis, while it is mostly used as such. The correct spelling would be of course 'the VHDL'.

Contents

Chapter 1

Introduction

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing 1950

1.1 Brief history of hardware design

In 1981 the United states department of defence faced the problem of the rapid semiconductor downscaling, which made the reproduction of old designs quite difficult. This was mostly due to the poor behavioural description of the designs, which where made with several languages, which behaved differently on different simulators [1]. To solve this problem the VHSIC hardware description language (VHDL)[2] was commissioned which had to be technology independent and replace the existing languages. At that time, programmable logic had been around for several years. The first field programmable gate array (FPGA) was released in 1985 however, which makes it quite unlikely that the designers of VHDL will have had these kind of devices in mind when designing the language. Ironically VHDL is nowadays mostly used for FPGA designs. When VHDL was officially released in 1987, a single chip could hold approximately 300.000 transistors. In 2012 Nvidia released the Keppler GK110 architecture, which contains more than 7 billion transistors [3] (which is approximately 23000 times as much). Though it must be mentioned that this GPU architecture has quite some repetitive logic, one can deduce that the size of the designs are much larger than 25 years ago, and therefore the descriptions are much more complex. Another important language that emerged is Verilog [4]. The newest versions of Verilog are now called System Verilog [5]. This thesis will focus on VHDL however.

1.2 Problem statement

Now that it is clear that the complexity of the work of designing a single chip has significantly increased in the past 25 years, one can conclude that it would be desirable to have a hardware description language (HDL) that is more effective than the currently available hardware description languages. 'Effective' alludes to the amount of code needed to describe hardware, the time needed for

detecting bugs, the rate at which bugs are introduced and the maintainability of existing code. With this in mind the following question is posed:

What language constructs can be added to the VHSIC hardware description language to keep it effective and relevant for the future?

To answer this question several language constructs are introduced that could be added to the language. To test the effects these additions have on the design speed, the language compiler and the language itself, a compiler is created that can compile a design that makes use of the proposed additions, to a behaviourally and constructionally equivalent design which is VHDL 2008 compliant. This second design could in turn be compiled by a VHDL 2008 compliant compiler like depicted in figure 1.1.

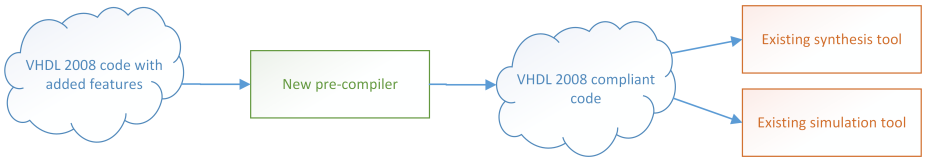

Figure 1.1: Tool flow with the new compiler.

1.3 Structure

In chapter 2: 'Compilers' starts with a short introduction on how compliers work because this can have a large impact on the design of a language, and will thus also have effect on the additions that can or cannot be made to VHDL. Chapter 3: 'Synthesis versus Simulation' shows how VHDL simulators work and gives some important differences between simulation and synthesis tooling. Chapter 4: 'Language Enhancements' shows the language enhancements that are added to the language. Chapter 5: 'The Compiler Design' shows the design of the compiler that should compile the language with enhancements back to the VHDL 2008 standard. In chapter 6: 'A Use Case' the compiler is put to the test, and finally chapter 7: 'Conclusion & Future work' gives the conclusions that can be drawn from this work.

Chapter 2

Compilers

A parser for things
Is a function from strings
To lists of pairs
Of things and strings.

Graham Hutton 2007

This chapter will shortly explain how most compilers work. This is done because the structure of a programming language is much influenced by the way compilers work.

The task of a compiler is to translate a file or a set of files to another file or set of files. One could for example make a compiler that translates a C-program to machine code or a compiler that translates French texts to English texts. When the compiler starts with an input file, it reads this file as a stream of characters. These characters are processed by a compiler part called the Lexer (also sometimes referred to as Tokenizer). The Lexer groups the characters it receives into structures called tokens and outputs these tokens as a new stream. Some characters are often dropped, like whitespace characters. This is also the case with VHDL, because they do not hold any significance any more (they are only used for spacing between the tokens). The resulting token stream is fed to a compiler part called the Parser. The Parser takes the tokens and tries to structure them into a tree structure called an abstract syntax tree (AST). This structuring is done based on a set of rules that describe the structure of the language. Listing 2.1 gives an example of a textual representation of some of these rules. This textual representation is a simplified form of the extended Backus–Naur form (EBNF) [6] which is an extension of the normal Backus–Naur form (BNF). The main difference is that EBNF is more expressive, and is therefore used more often by language designers. The presented rules are a simplified version of the actual syntax rules of the official VHDL specification.

```
signal_assignment = target '<=' delayed_value;
delayed_value = value ('AFTER' delay)?
value = Constant;
delay = value Unit;
target = Identifier;

Constant = Digit Digit*;
Unit = 'hr' | 'min' | 'sec' | 'ms' | 'us' | 'ns' | 'ps' | 'fs';
Identifier = Character (Character | Digit)*;
```

Listing 2.1: Simplified EBNF.

The '*'-sign implies that the preceding rule is repeated zero or more times and the '?'-sign signals that the rule is optional. Though not in this example, the '+'-sign denotes that the preceding rule is repeated one or more times. The '|'-sign denotes a choice between rules and brackets are used for grouping syntax rules. Text between quotes are tokens that must match a token of the input stream. Rules marked with a capital also mark tokens, but instead of having a literal representation, like the tokens between quotes, these tokens are identified by some rules that define the structure of the token. The constant rule for example will match to all tokens that have an integer representation.

Figure 2.1 gives a graphical impression of what the Lexer and the Parser do with a small piece of VHDL code. The tokens can now be found at the leaves of the AST, and the syntax rules of the EBNF are at the branches.

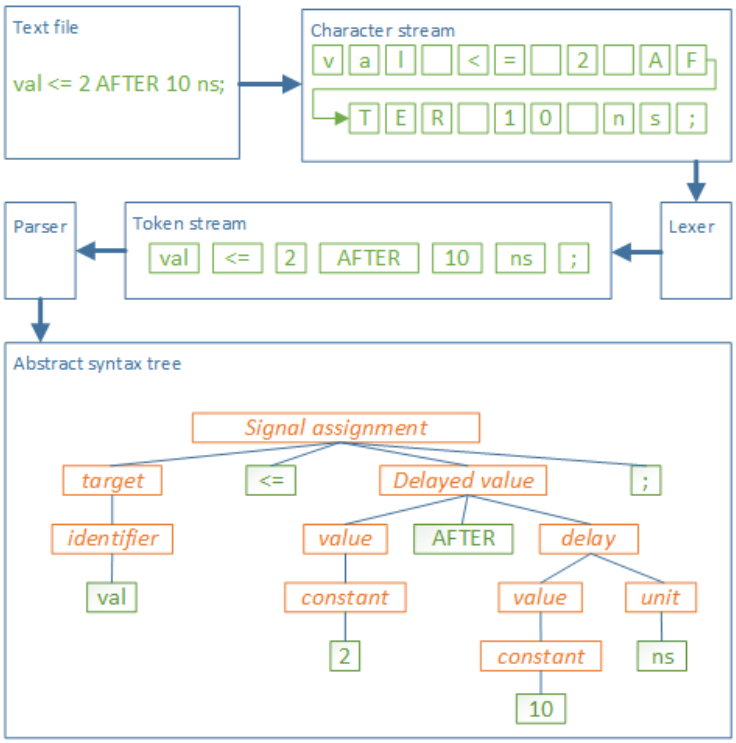

Figure 2.1: Graphical representation of the front-end of a compiler.

Once the AST is constructed the tree can be 'walked'. This means that a process iterates trough the tree and take certain actions when certain nodes are

reached. What these actions are depends on what the compiler must achieve. Each time a process *walks* through the entire tree is called a 'compiler pass'. The first pass can be done while the parser is constructing the AST.

2.1 The parser

Generating an AST from a token stream is not a straight forward task. Several methods of doing this are developed, but the focus in this thesis will be on the recursive descend method.

When the parser receives its first token it will try to match this token with a start rule of the BNF. In the example the start rule is the *signal_assignment* rule. When entering the *signal_assignment* rule the parser will first have to match the *target* rule. The parser now enters the *target* rule and detects it has to match an *Identifier*. The parser will match the 'val' token with the *Identifier* rule. After this, the parser is at the end of the *target* rule and will continue with the *signal_assignment* rule. According to the BNF the next token should be '<=', which it is. Now the parser will enter the *delayed_value* rule. The first rule of the *delayed_value* rule is processed in a similar fashion as the *target* rule. Now the parser has to decide whether the ('AFTER' delay) part is used. This can be done quite easily because this part can only be used if the next token is 'AFTER'. In the example this is the case, so the ('AFTER' delay) part is evaluated. After this the *delayed_value* and *signal_assignment* rules are finished.

2.1.1 LL(1) Grammars

The parsing of the presented example grammar is quite easy compared to the actual VHDL grammar. This has several reasons. The first reason is that the example grammar is an LL(1) grammar. This means that when the parser can read its input tokens from left to right, and when the compiler must make a decision, it can do this based on the next incoming token (like with the ('AFTER' delay) part). The VHDL grammar is not LL(1). An example of this can be seen in the three simplified VHDL rules of Listing 2.2.

```
concurrent_statement = procedure_call | signal_assignment;
procedure_call = Identifier '(' association_list ')';
signal_assignment = target '<=' delayed_value;
target = Identifier;
```

Listing 2.2: Non LL(1) grammar.

When the parser enters the concurrent_statement rule and has received an identifier token it cannot decide whether the procedure_call rule, or the signal_assignment rule should be chosen. Instead the parser will have to look one token ahead to see whether that token is '(' or '<='. There are several ways to deal with this problem, but they are outside the scope of this text. It is important to remark however that the number of tokens that have to be read and the number of rules the parser has to choose from, can be significant for the VHDL grammar, which has a negative effect on the performance of the compiler.

2.1.2 Context sensitivity

Another issue of the VHDL grammar is its context sensitivity. In listing 2.3 for example it is not possible to determine what 'foo(x)' means. It could be a call to a function called foo with argument x, it could also mean that foo is an array and so foo(x) means element 'x' of foo, but it could also mean that x is casted to a type called foo. The actual meaning of this statement is dependent on how foo and x are declared, which is done somewhere in the context of the statement, hence context sensitive.

```
y := foo(x);
```

Listing 2.3: A context sensitive statement.

2.1.3 Left recursion

A final problem of the VHDL EBNF is that it has left recursive syntax rules. This means that there exists a path through the syntax rules where the same rule is visited twice without a token being consumed in between. This is an insurmountable issue for most parsers, because they can get caught in this loop when trying to determine the right path through the EBNF. Often these rules can be rewritten to make them non left recursive. Listing 2.4 gives a very simple example of a left recursive EBNF and the same rule rewritten to make it non left recursive.

```
Left recursive:     a = a? | B;
Non left recursive: a = B+;
```

Listing 2.4: Left recursion

Chapter 3

Synthesis versus Simulation

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Edsger W. Dijkstra 1972

In this chapter some of the differences between synthesis tools and simulation tools are shown. It is important to understand these differences when transformations are made from the "new version of VHDL" to the VHDL 2008 version.

3.1 The VHDL Simulator

The difficulty of making a good VHDL simulator is coping with the large amount of parallelism. Parallelism can lead to race conditions and deadlocks if not handled properly. It is therefore important to understand which pieces of code can be executed sequentially and which pieces can be executed in parallel.
Concurrent statements can all be executed in parallel. These are essentially all the statements that can be found directly in architecture bodies. Sequential statements cannot be executed in parallel and these are essentially all the statements that can be found in sub-programs and processes. All concurrent statements can be transformed into processes without sensitivity lists, but with a 'wait on'-statement and putting all the signals of the sensitivity list in that statement. Processes with sensitivity lists can also be converted to a process without a sensitivity list by also adding a 'wait on' clause at the end of the process. Concurrent signal assignments (assignments that use the "<=" assignment operator) can be converted by creating a process with a 'wait on' clause at the end, which contains all the signals at the right hand side of the assignment operator. All concurrent statements can be converted in similar ways.

3.1.1 Events

Simulators are event based. This means that time does not progress continuously, but with jumps. To understand this way of simulating, two types of events need to be distinguished; timed events and delta events. Timed events are moments at which some signal is scheduled to change. They can be ordered based on the moment in time they occur. In the time gab between the timed

events the system is considered stable.
Delta events follow each other in an infinite small time frame. They are the result of a timed event, and will –in most cases– bring the system back to a stable state. Delta events will continue to occur until the system is stabilized. It is possible to write code for a hardware model that will never become stable after some timed event. As a result there will be an infinite number of delta events. Simulators therefore detect when the number of delta events reaches a certain threshold. When this threshold is reached the simulator normally breaks off the simulation with a warning or error message. Figure 3.1 gives a graphical impression of what the architecture of a VHDL simulator could look like.

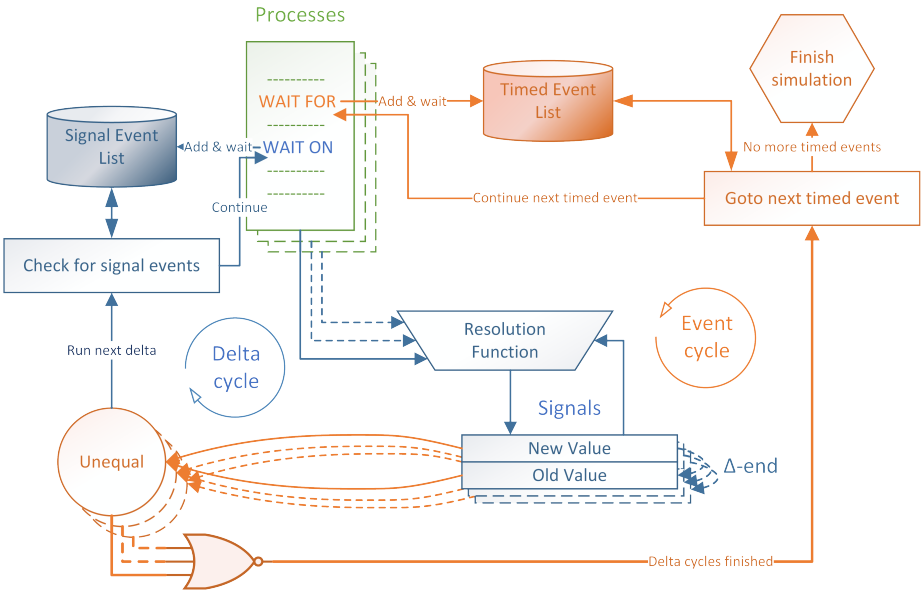

Figure 3.1: Diagram of a HDL simulator.

After the first delta cycle is finished the simulator will check if there are signals that have a different value than before the cycle started. If so, a new cycle will be started with the new value of the signal. In this cycle all 'wait on' clauses that include the changed signal will continue to run until a new wait statement is encountered. At some point there might be transactions (setting the value of a signal) on the signals, but the values remain the same. At this point the delta loop is finished and the next timed event is started. When there are no more timed events the simulator will finish the simulation. Often test benches have a clock signal that will keep adding timed events and these simulations will never be finished by the simulator.
The order in which the processes are executed is not important and can be nondeterministic. The only race condition that can occur is when multiple processes try to change the value of the same signal within a delta cycle. In VHDL this issue can be solved in two ways; using resolved signals (for example signal type `std_logic`) or using unresolved signals (for example signal type `std_ulogic`).
The simulator is able to detect when a signal is written twice within the same

delta cycle. When this happens for an unresolved signal the simulator will simply give an error. If the signal is of a resolved type however, a resolution function is used to determine the actual value of the signal. In this case, the first process that writes to the signal can do this in the same way this normally happens. Subsequent processes use a special function that is part of the resolved type called a resolution function, in combination with the value they want to write and the value the previous process has written. The resolution function normally makes use of a resolution table. When the resolution function is called, it looks-up one of its input values on the horizontal axis of the table, and its other input value on the vertical axis. The location where these two axes cross contains the resulting value of the function. This also shows why resolution tables –though not required by the language– should always be symmetrical. Table B.1 shows the resolution table of 'std_logic'. The coloured entries show that the result is the same if one would first write a '1' and then a '0' or vice versa.

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

Table 3.1: Resolution function table of std_logic.

3.2 The VHDL Synthesis Tool

Most synthesis tools convert a register transfer level (RTL) description to a gate level description. An RTL description describes how data flows between registers, and how this data is functionally manipulated in between these registers. When the synthesis tool creates a gate level description from the RTL description, it preserves the registers and the functional behaviour, but gives an exact description of which gates are used, and how they are connected to get the functional behaviour that was described in the RTL description. Both descriptions can be VHDL code, and if they are, both descriptions can be simulated. The gate level description simulation, also called the post-synthesis simulation, will be more accurate, since the delays of the actual hardware cells are now known. Of course not all VHDL constructs can be supported by synthesis tools like delays for example, but a lot is supported nowadays. There is a VHDL sub-standard for synthesis which defines a minimum set of constructs a synthesis tool must support to be IEEE compliant, however this standard is currently marked as withdrawn because it was considered unnecessary and was not used by many synthesis tool vendors[7]. The standard does not specify what the resulting hardware should look like, but the behaviour of each construct is specified. The fact that the resulting hardware can differ from tool to tool, can cause language transformations to have different results while the behaviour of

the design remains the same. This is especially true for VHDL, because behaviours can often be described in several ways. It is therefore not possible to guarantee that language transformations made by a compiler, will have any negative or positive effects on the resulting hardware.
When it comes to hierarchy, most synthesis tools prefer to completely flatten the design, meaning that all hierarchy is lost. This can be advantageous when optimizing the design, because logic that used to be in different hierarchies can now be combined or evaluated together. It is often also possible to give the synthesis tool some regions that should be evaluated separately. The goal of this is to still have some kind of encapsulation of certain parts of the design after synthesis, which can be used when mapping the design on a chip. Cache memory can for example be grouped together. Since a cache memory uses little energy compared to its surface area the power lines can be relatively narrow when designing an application specific integrated circuit (ASIC). The computational part needs much more energy and thus needs wider power lines. Having this post synthesis hierarchy makes routing the design much easier while still having an optimized design. There is much more involved in designing the resulting ASIC or FPGA, but this is outside the scope of this report.

Chapter 4

Language Enhancements

If someone claims to have the perfect programming language, he is either a fool or a salesman or both.
Bjarne Stroustrup

In this chapter discusses the constructions that are added to VHDL which should make the language easier to use. Though there are many things that can be thought of, this thesis will focus on structuring the code which should be particularly beneficial for large projects.

4.1 Independent Compilation Order

The current version of VHDL is sensitive to the order in which files are compiled. Unlike most other languages that have this dependency VHDL is not capable of specifying this order in the language itself. Instead this must be done with a compiler setting that specifies the compilation order. This can be particularly problematic when using large designs or when the design must be compiled by different compilers (for example a simulator and a synthesis tool). Some tools try to solve this issue by automatically generating the compilation order, however this does not always yield the right result.
There are essentially three ways of solving this issue. The first one is to add language constructs that give the user the possibility of textually specifying the compilation order. A second option is to let the language specify the dependencies of each design unit (which are entities architectures packages and so on) and compile these before the design unit itself is compiled. This is essentially what the C-language does. The automatic compilation order generation of VHDL compilers work in a similar way. Issues can however arise with this approach when there is a circular dependency between design units.
The third option is using a compiler with multiple passes. In a first pass identifiers and declarations can be collected. Then, during the second pass, these identifiers and declarations can be linked to each other. After this pass, signal declarations have a link to the declaration of their type for example. After this, in a third pass, the expressions can be evaluated.
The latter option is not practical for all languages since the design of a language can be such that there cannot be a finite number of passes, however for VHDL

this option should be possible. It is also the only option that effectively relieves the programmer from the burden of specifying the compilation order (or even think about it for that matter). It does however come at the cost of extra compilation passes. Nevertheless this option is chosen for the compiler since it is the only solution that really solves the compilation order issue.

4.2 Declaration Area's

Constructs like architectures, entities, procedures and blocks are divided into two parts; the declaration area and the body area. In the declaration area (like the name suggests) several things can be declared, like types, variables, signals, sub-programs and so on. In the body area the declared elements can be used. So, variables can be assigned to and sub-programs can be called for example. The advantage of having these two separate regions is, that when the body area starts, the compiler knows exactly what elements can be used in that area.
If one takes a closer look at what actually happens when a signal is declared inside a declaration area, it becomes clear that actually several things happen. First of all the compiler is told that there exists a signal, and that the signal has a name which can be used to refer to it. Secondly the compiler is told what properties the signal has by assigning a type to the signal. Finally, the signal can optionally have an initial value (which should be of the same type as the type of the signal). For variables this approach is the same.
In the body area the declarations can be used. The motivation for this approach becomes clear when a closer look is taken at the VHDL simulator. Since each process can be considered to be a small program, the simulator will need to reserve some memory space for the process. When generating machine code for the process, the memory allocation operations must be performed before the machine code for the actual body can be generated. If the compiler is single pass, this can only work if all declarations appear before any operations. The separation of declaration and body area ensures this.
The previous section showed the reason for using a multi-pass compiler. Using such a system remits the motivation for having a separate declaration and body region. In a multi-pass compiler the memory allocation machine code can be generated once the types of the (signal) declarations are linked to the type declarations, as this is the moment at which the memory needed for that declaration is known. In a subsequent pass, the machine code for the expressions can be generated.
To stay backward compatible, separated regions will still be supported. The original declaration statements and the 'BEGIN'-keyword (which separates the two regions) are therefore made optional. Adding all the declarations that can be made in the declaration part declarable in the body part is also made possible. The result will be that declarations can be done in both the declaration and body area, and that the declaration area can be completely omitted by the user. Listing 4.1 shows the original EBNF and the new EBNF of the architecture rule. The rules for other elements of the language that have a separate declaration and body region are modified in a similar fashion.

```
        Before:
architecture_body
: ARCHITECTURE Identifier
    OF entity_name IS
      block_declarative_item*
  BEGIN
      concurrent_statement*
  END ARCHITECTURE? Identifier? ';'
;
```

```
        After:
architecture_body
: ARCHITECTURE Identifier
    OF entity_name IS
    (block_declarative_item* BEGIN)?
    (block_declarative_item |
        concurrent_statement)*
  END ARCHITECTURE? Identifier? ';'
;
```

Listing 4.1: No declaration area in the architecture.

Having no separation between the declaration and body parts has some advantages. Signals that are only used for intermediate values and thus do not have a meaningful name can be declared near the place they are actually used. Signals (or variables) that do have a significant meaning or function can still be declared at the beginning of a module, but these declarations are no longer cluttered by the less significant ones. Another advantage of not having two separate regions is that it opens the possibility of directly assigning a signal or variable. This is discussed in the next section.

4.3 Direct Assignment

Now that there no longer are separate declaration and body regions, opportunities arise for combining elements of both regions. One of the things that can be combined is the signal declaration and the assignment of this signal. This means that on the same line a signal is declared, also a value can be assigned to it. This can be combined with the initial value assignment of the signal. Listing 4.2 gives an example of how this new construct could be used.

```
SIGNAL x : std_logic := '0' <= not y;
```

Listing 4.2: Direct signal assignment.

This new construct not only shortens the code, but it also encourages the user to use more intermediate variables. This will shorten the average expression length which makes the code easier to debug and because the expressions become less complex, the speed at which bugs are introduced will be reduced.
A disadvantage of using more intermediate signals is that they can negatively influence the simulation speed if the compiler does not do some optimizations. This speed reduction is caused by more delta cycles being started, when more signals are subsequently assigned to. Listing 4.3 shows two pieces of code; one with intermediate signals and one without. Figure 4.1 shows the two delta cycles of both implementations.

```
-- Without intermediate signals
SIGNAL a, d : std_logic := '1';
SIGNAL u, v, w: std_logic := '0';
d <= ((a xor u) xor v) xor w;
```

```
-- With intermediate signals
SIGNAL a, b, c, d : std_logic := '1';
SIGNAL u, v, w: std_logic := '0';
b <= a xor u;
c <= b xor v;
d <= c xor w;
```

Listing 4.3: Signal assignments with and without intermediate signals.

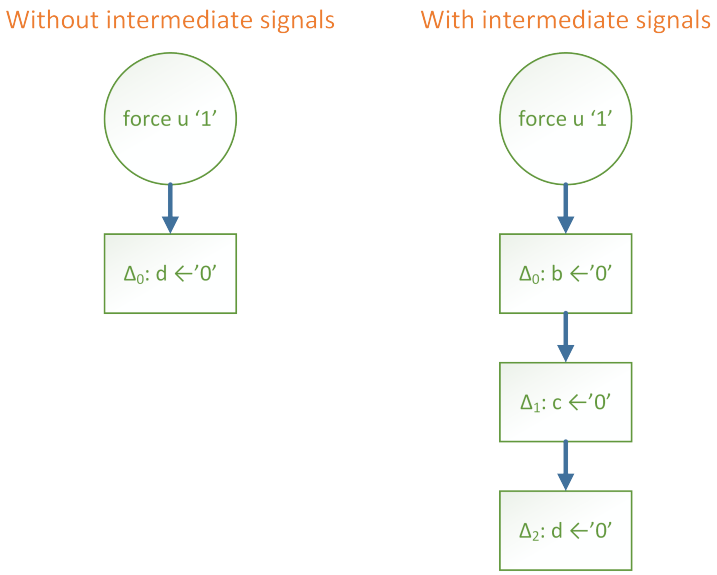

Figure 4.1: Delta cycles of the assignments with and without intermediate signals.

A smart compiler will be able to combine the subsequent signal assignments if it can detect that the signals are not used anywhere else. The resulting performance should then be the same as the listing without the intermediate signals. A performance improvement is also possible. If in the example signal 'w' changes instead of signal 'u' then only one 'xor'-expression has to be evaluated instead of all the 'xor'-expressions. A smart compiler might however also be able to do this with the assignment without intermediate signals.

4.4 Assignment Operator Overloading

VHDL is said to be strongly typed. There is however no complete consensus in the field on the definition of "strongly typed" [8]. Knowing the type of each variable, expression and sub-expression at all times, would probably be a sufficient definition for now. Because of the strong type system, it is not always trivial to convert one type into another. If an `integer` constant would have to be converted to an `unsigned` for example, a conversion function would be needed. It would be much more natural if this could be done without the conversion function. This is however not in compliance with the strong type system.
An approach that can be taken is to look in a different way at an assignment. If the assignment would be seen as an operation, then it could also be seen as an operation that converts one type into another. To do this, functionality needs to be added to the assignment operator. A language construct called overloading can be used for this. Overloading is normally done with functions, but when the assignment operator is looked at as if it is a function then it could also be overloaded.
Function overloading means that two or more functions with the same name can be created. When the function is used, the compiler decides which function is used based on the types of the arguments and the return type. Listing 4.4 gives

an example of this. Unlike most other languages, functions in VHDL can be overloaded based on there return type. There is a good reason why these other languages do not allow this, however this is outside the scope of this thesis. By letting the assignment operator be overloaded in several ways, it could be used to assign different types to each other in a predefined way.

```
FUNCTION foo(bar : integer) RETURN boolean; --this one is used
FUNCTION foo(bar : string)  RETURN boolean; --this one isn't used
SIGNAL x, y : integer;

x <= foo(y);
```

Listing 4.4: Function overloading.

VHDL essentially has two types of functions; procedures and functions. A procedure is essentially a function without a return type. Though it may seem logical to see the assignment as a function, a procedure will be more practical in this case. Procedures do not have a return value, but they can have output arguments. An assignment operator in procedure form would be a procedure with one input argument and one output argument like shown in listing 4.5.

```
PROCEDURE "<=" (input: IN integer; output: OUT signed);
```

Listing 4.5: Procedure for overloading an assignment operator.

The procedure is chosen over a function because properties of the output value can now be read. When referring to listing 4.5, the vector length of the `output` signal can be gotten for example. This would not have been possible if a function would have been used for this.
A good example is when one would use a assignment overload of the `to_signed` function of the `numeric_std` library. This function can convert an integer to a signed variable, but this function needs two arguments; the integer value and the length of the signed variable. It needs this length because the function cannot get the length of the return type because it cannot access the return type. Using a procedure for the overload solves this problem since the return type is now an argument. Listing 4.6 gives a complete example of the implementation of an overloaded assignment operator and how it can be used.

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   USE ieee.numeric_std.all;
4   PACKAGE numeric_std_additions IS
5       PROCEDURE "<=" (input: IN integer; output: OUT signed);
6   END PACKAGE;
7
8   PACKAGE BODY numeric_std_additions IS
9       PROCEDURE "<=" (input: IN integer; output: OUT signed) IS
10      BEGIN
11          output := to_signed(input, input'LENGTH);
12      END PROCEDURE;
13  END PACKAGE BODY;
14
15  LIBRARY ieee;
16  USE ieee.std_logic_1164.all;
17  USE ieee.numeric_std_additions.all;
18  ENTITY foo IS
19      PORT (a: IN integer; b: OUT signed);
20  END ENTITY;
21
22  ARCHITECTURE arch OF foo IS
23  BEGIN
24      b <= a;
25  END ARCHITECTURE;
```

Listing 4.6: assignment operator overloading.

At line 5 the overload procedure is declared. Like every procedure declaration the declaration starts with the `PROCEDURE`-keyword. This is followed by the signal assignment operator between quotation marks to indicate that this procedure is an assignment overload. Next there is the input argument followed by the output argument. On lines 9 to 12 the body of the overload is implemented. The `to_signed` function is used at line 11 to convert the integer into a signed type.
At line 24 the assignment overload is used. Since 'a' is of type integer, and 'b' is of type signed, 'a' can normally not be assigned to 'b'. The compiler will detect this and finds that there is an overload for the assignment that can convert an integer into a signed type. The output of the compiler will therefore replace the assignment of 'a' to 'b' with a call to the assignment overload procedure where 'a' will be the first argument, and 'b' will be the second argument.

4.5 Namespaces

Most non-natural languages have one or more encapsulation methods. So does VHDL. There is a clear hierarchical structure with the libraries on top followed by packages, entities, architectures and so on. These in turn can encapsulate sub-programs for example. Packages are the only second level hierarchy where the contents of the hierarchy level can be accessed by all other hierarchies. A sub-program that is declared inside a package can for example be called from an entity. Packages also are the only second level hierarchy that do not contain any functionality Themselves; all functionality is contained within sub hierarchies of the package. Taking these points together the conclusion can be made that a package is a container for language elements that can be accessed from anywhere in the language.
When looking at the formulation of libraries it would be practically the same as the formulation of packages. The only real difference is that the name of

the library is not determined by the language itself, but by the tool that uses the language files. As a result designers often create only one package within a library with (almost) the same name as the library.
The declaration of a package consists of two parts; The package itself and the package body. The package body contains all the functional elements, and the package essentially defines which of these elements can be accessed and how they are accessed. The result is that the separation of the package and the package body is essentially an elaborate way to solve scope visibility.
A single package cannot be split up unto different files. The same holds true for a package body. As a result these package files and especially the package body files become quite large (over 1000 lines).
To summarize, there are essentially four issues. The function of packages and libraries are very similar, The names of libraries cannot be described by the language itself, but must be described by the tool that uses the code, making elements in packages is solved in an unnecessary complicated way and package files become too large. To solve all these issues a namespace construct is proposed.
The namespace is like a package, however it can be split up over multiple files. Also namespaces can contain everything that is normally contained within a package, a package body and everything that is normally put directly inside a library (like an entity). Also for sub-programs the sub-program header is implicitly made by the compiler if no other sub-program headers are declared for that sub-program.

The declaration of the namespaces will be much like a package declaration. Listing 4.7 shows the BNF of the new language construct. The namespace can be preceded by context items which are library clauses, use clauses and context clauses. Also the entities, architectures configurations and context declarations inside a namespace can be preceded by context items. Namespace constructs with the same name can be declared in multiple files. The compiler will virtually combine these namespaces into a single one. Context items that are used on a namespace in one file will only have effect on that file and will thus not have effect on the other files where the same namespace is declared.

```
namespace_body
    : NAMESPACE namespace_name IS
        (
            context_item* (
                  entity_declaration
                | configuration_declaration
                | architecture_body
                | context_declaration
            )
            | namespace_declarative_item
        )*
      END NAMESPACE? namespace_name? ';'
    ;

namespace_name
    : Identifier ('.' Identifier)*
    ;
```

Listing 4.7: BNF of the namespace construct.

The `namespace_declarative_item` contains everything that one would normally find in a package. Since there are no longer separate package and pack-

age body regions, another easier way of declaring what is visible from outside the namespace and what is not is needed. For this two access modifiers are introduced, namely 'public' and 'private'. In contrast to most languages, elements in a namespace are public by default when no access modifier is used, since public access is demanded most of the time. When a `namespace_declarative_item` is declared as private, the elements can only be accessed by other `namespace_declarative_items` in the same namespace, and they cannot be accessed by entities, architectures, configurations and contexts, even if these are declared in the same namespace. Listing 4.8 shows the BNF of the `namespace_declarative_item` rule.

```
namespace_declarative_item
    : (PUBLIC | PRIVATE)? (
          alias_declaration
        | attribute_declaration
        | attribute_specification
        | component_declaration
        | constant_declaration
        | disconnection_specification
        | file_declaration
        | group_declaration
        | group_template_declaration
        | package_body
        | package_instantiation_declaration
        | signal_declaration
        | subprogram_body
        | subprogram_declaration
        | subprogram_instantiation_declaration
        | subtype_declaration
        | type_declaration
        | variable_declaration
    )
    ;
```

Listing 4.8: BNF of the namespace_declarative_item.

4.6 Fast Attributes

VHDL has two types of attributes; predefined attributes and user-defined attributes. The predefined attributes are properties of the language element for which they are defined. For array types for example, the length attribute is defined which gives the number of elements in the array.
The user-defined attributes are used to add extra properties, also sometimes called meta data, to elements in the language. This data can later be read and used. User-defined attributes can also be used by tool vendors to get or add tool specific properties to the language. For the Altera Quartus II software one can for example tell the language compiler to limit the fan-out of a certain register, tell which language version to use, or tell which top level entity ports should be mapped to which pins on the physical chip.
To apply and use a user-defined attribute there are some steps involved. First the attribute has to be declared. During the declaration, the name and the type of the attribute is defined. Secondly the attribute has to be created for one or more language elements. By doing this, the attribute that was just declared is referenced, the compiler is told which language element the attribute is created for, the compiler is told which type the language element has and what the value of the attribute is. Finally the attribute data from the language element can be

gotten. Listing 4.9 shows an example of creating and using an attribute for a signal.

```
1  ARCHITECTURE implementation OF attributeExample IS
2      -- Declare the attribute.
3      ATTRIBUTE exampleAttribute : string;
4
5      -- Declare a signal we will create the attribute for.
6      SIGNAL sig : integer;
7
8      -- Create the attribute for the signal.
9      ATTRIBUTE exampleAttribute OF sig: SIGNAL IS "Hello World!";
10 BEGIN
11     -- Use the attribute
12     ASSERT FALSE REPORT sig'exampleAttribute;
13 END ARCHITECTURE;
```

Listing 4.9: Declaring, defining and using an attribute.

When this code is run, the user will get the message "Hello World!". This is because the assertion at line 12 will fail. Therefore the simulator will display the `exampleAttribute` attribute of the `sig` signal. At line 9 this `exampleAttribute` attribute is created for the `sig` signal with the string value "Hello World!", so this message will be displayed.

At the line 9 it can be seen that some of the information that is given to the compiler is redundant. Since it is known that `sig` is a signal, and no other elements with the same name exist, there should be no real need to tell the compiler explicitly that `sig` is a signal in this case.
Several file types have been developed to pass this information to the tools that use VHDL while attributes could be used just as well. This is probably due to the fact that adding attributes is to much 'work' for the designer. Therefore propose a simpler way to create attributes for language elements is proposed. Listing 4.10 shows the creation of the same attribute as the previous example, but now in the proposed way.

```
1  ARCHITECTURE implementation OF attributeExample IS
2      -- Declare the attribute.
3      ATTRIBUTE exampleAttribute : string;
4
5      -- Declare a signal with an attribute.
6      {exampleAttribute : "Hello World!"}
7      SIGNAL sig : integer;
8  BEGIN
9      -- Use the attribute
10     ASSERT FALSE REPORT sig'exampleAttribute;
11 END ARCHITECTURE;
```

Listing 4.10: Declaring, defining and using an attribute with the new language construct.

The same information is contained within the attribute creation, but now the code is shorter. This is possible because the positional information of the statement is used. This means that the compiler knows that the attribute is declared for the `sig` element because this is the first language element after the attribute declaration; The position of the attribute creation compared to the

signal declaration defines what language element the attribute is created for. Now that the compiler knows the attribute is declared for `sig`, it also knows that `sig` is of element type `SIGNAL`, since this is part of the declaration of `sig` at line 7. Listing 4.11 shows the BNF for the new syntax rule.

```
relative_attribute_specification
    : '{' Identifier (OF entity_designator)? ':' expression  '}'
    ;
```

Listing 4.11: new attribute BNF.

With this improvement it is more likely that the attributes are used more often and can take over the tasks of the external meta data files like SDC[9] and SDF[10].

Chapter 5

The Compiler Design

divide et impera
Philip II of Macedon

This chapter will discuss the compiler that converts the VHDL with the added features to the VHDL 2008 standard. The effects that the added features have on the design of the compiler are also shown. The compiler does not support everything a normal VHDL compiler supports. Generics are for example not supported. Also ranges of array types are not exactly known by the compiler. This is partly because the range could be set by a generic. Most elements of the language are supported however.
Some of the images in this chapter might be too small for some readers. Therefore larger versions of these images are included in appendix A.

5.1 A Five Pass System

As mentioned in the previous chapter, compilation of designs would become easier if there is no specific order in which the files have to be compiled. To make this possible the compilation process will have to be split up into several passes. The goal is to do as much as possible within one compilation pass, to keep the number of passes as low as possible. There is however a minimum number of passes. This minimum is determined by the dependencies within the language.

As mentioned before, there is a clear dependency between declarations and statements. A signal assignment statement for example, assigns a value to a signal that was declared earlier by a signal declaration. There also exists a dependency between the declarations them selves. The signal declaration assigns a type to the signal it declares. This type is declared somewhere else, maybe even in an other file. The following sections will show how these dependencies are dealt with. The five pass compiler is explained with the aid of the VHDL code of listing 5.1. The AST and the class diagrams of in the upcoming sections are simplified versions of the actual AST and class diagrams; only the elements which are relevant to the example are shown.

```
1  architecture arch of ent is
2      signal x : std_logic;
3      signal y : std_logic_vector(7 downto 0);
4  begin
5      y <= x(0);
6  end;
```

Listing 5.1: Example architecture.

5.1.1 Pass 1: Lexing and Parsing

The first step that needs to be done is the lexing and parsing. For this a tool called ANTLR[11] is used. This tool can generate a lexer and a parser from an EBNF. In contrast to most VHDL compilers, our parser does not have any knowledge of the context of the lines it is parsing. When it comes to expressions, VHDL is a context sensitive language. Expressions can be found within variable and signal assignment statements. To cope with this issue expressions are parsed without the context during this pass, and are re-parsed later in a different pass when the context is known. Parsing the expressions without the context is possible, however some elements will be identified incorrectly. At this point the parser will just choose one of the possibilities, which will likely be wrong. This will be fixed once the context of the expression is known, which is during pass 4. Figure 5.1 shows the resulting AST of listing 5.1.

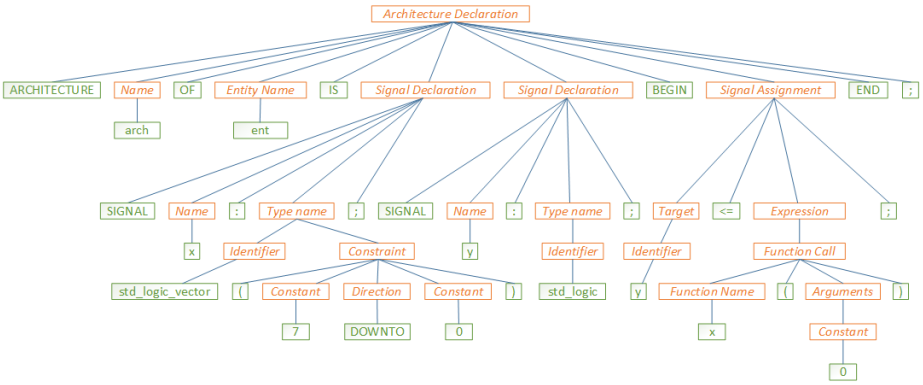

Figure 5.1: Data structure after pass 1.

5.1.2 Pass 2: Element Collecting

Ones the AST is generated it can be traversed. During this pass all the scopes and all declarations are collected. By doing this, the first context information is gathered. It is now known for example what the scopes of all the declarations are. For each element that is collected, a separate object with a reference to the location in the AST where the VHDL object was declared is created. Figure 5.2 shows these objects graphically. The links to the AST's are depicted with yellow arrows and the links between objects with blue arrows. The parts that where already there from the previous pass are behind the shaded surface. The nodes and leaves of the AST's contain the locations of the text stream where they originate from. This information can be used when showing an error message to the user with the location of the error.

The only links between the objects that are made at this point, are the links that can be made based on the hierarchical structure of the language. The compiler knows for example, that the signal declarations are part of the architecture, since this can be directly deduced from the structure of the AST. A link to the entity of the architecture can for example not be made at this point since the entity could be declared in another file and may not be compiled at this point.

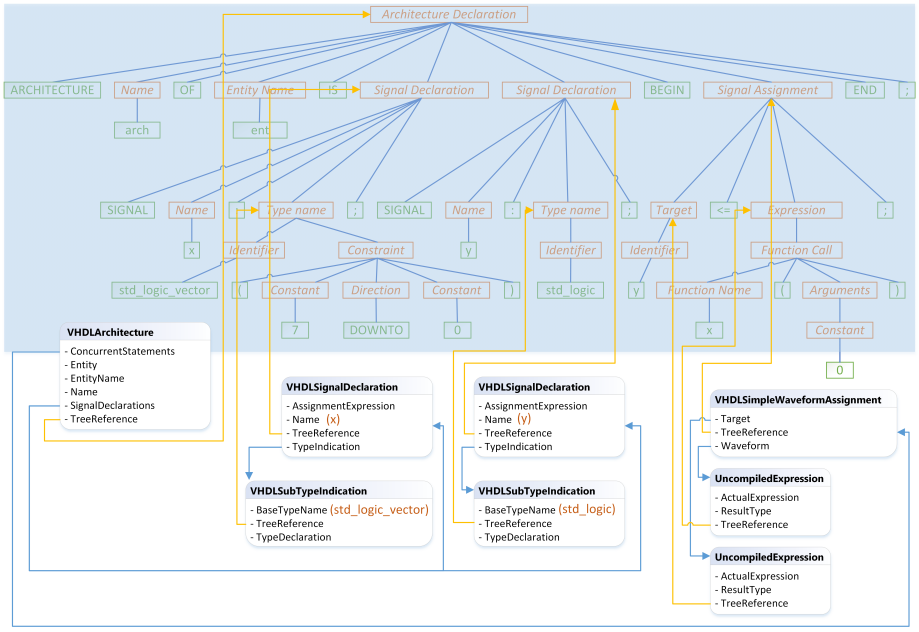

Figure 5.2: Data structure after pass 2.

5.1.3 Pass 3: Element Linking

At this point all elements of the code are collected and they can now be linking them together. First the architectures are linked to their entities, and the package bodies to the packages. This is done first because when the entity for example has a use-directive (for example `USE ieee.std_logic_1164.all;`) this use directive is also valid for the architectures of the entity. The same holds true for packages and package bodies. After this, the types of the signal declarations with the type declarations can be linked. This is done based on the name of the type and a list of all the types that are in scope. Figure 5.3 shows the example with arrows for the entities and the types. Since the type declarations and the entity declaration are not part of the example code, the arrows cannot be drawn to these objects in the figure, but the reader should be able to imagine what this construction would look like. If the entity or type declarations would not exist, the compiler will give an error message.

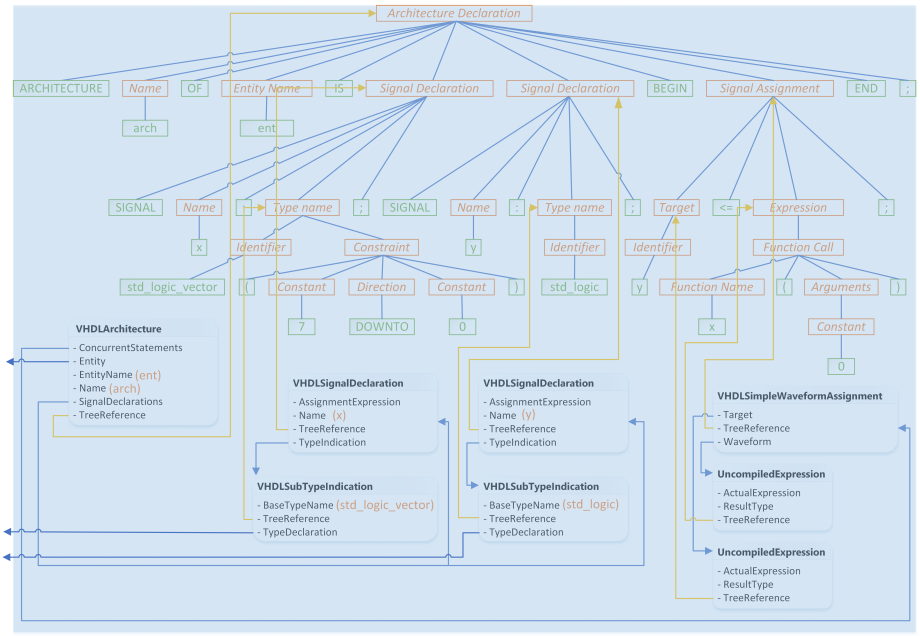

Figure 5.3: Data structure after pass 3.

5.1.4 Pass 4: Expression Parsing

At this point the expressions can be re-parsed because now all the context information has been collected. Figure 5.4 shows the AST of the re-parsed expressions. The parser now knows that `x` is a `std_logic_vector` and not a function. After this objects can be created for the expression AST's like during pass 2. At the same time type checking is started. If types do not match or if there are ambiguous expressions the compiler is able to find them and report an error.

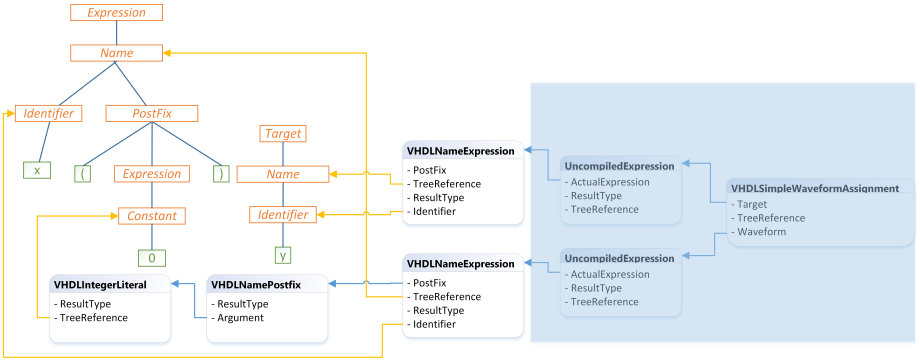

Figure 5.4: Data structure of the expression after pass 4.

5.1.5 Pass 5: Generating Output

The final step in the compilation process is generating the output. This is done by traversing the object tree. Each object generates its output via a

special output writer. This writer takes care of the correct outlining of the code. Listing 5.2 shows the output of the compiler. Each entity, architecture, package, package body and each configuration is put in its own file. Therefore the number of files after the compilation is larger than the number of input files. The files have logical naming, which makes it easy to find elements in the output of the compiler.

```
1  ARCHITECTURE arch OF ent IS
2      SIGNAL x : std_logic;
3      SIGNAL y : std_logic_vector(7 DOWNTO 0);
4  BEGIN
5      y <= x(0);
6  END ARCHITECTURE;
```

Listing 5.2: Example architecture after compilation.

The output of the compiler is almost exactly the same as the input. This is what one could have expected since VHDL 2008 code was compiled to VHDL 2008 code. The next step is adding the new language structures to the compiler so the compiler can actually do something useful.

5.2 Type checking

In section 5.1.4 was mentioned that after each expression is parsed it is type checked. Type checking is the process of making sure that all types that go into a function or operation are what these operations and functions expect. If one would for example give a function that expects a bit, not a bit, but an integer, the compiler should give an error saying that the types are incompatible. To be able to do this the compiler needs to find the result type of each sub expression. Listing 5.3 shows a simple expression, and figure 5.5 shows the resulting expression tree after parsing. The types of the expression can be determined in two ways; top down and bottom up. The compiler –like most compilers– uses the bottom up approach.

```
SIGNAL x, y, z, u: integer := 4;
u <= x + y - z;
```

Listing 5.3: A simple expression.

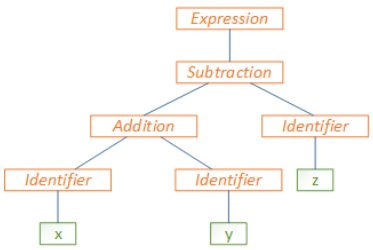

Figure 5.5: Expression tree of listing 5.3.

The bottom up approach starts by determining the types at the bottom of the tree. In this case these are tokens `x` and `y`. These tokens are combined in the

addition branch. VHDL has an addition operator for several types like signed numbers, unsigned numbers, natural numbers, but in this case the compiler will choose for an addition that adds two integers, since its input types are integers. The result of the addition is also of type integer. Now the type of the subtraction is evaluated. The subtraction is also defined for several types, but the result of the addition is an integer and `z` is an integer, so a subtraction of two integers is used, resulting in an integer. This means that the final result of the expression is also an integer. Now that the expression is type checked the compiler can check if the type result of the expression matches the type of the target signal `u`. This is the case and so the type checking succeeds. If the compiler would not have found an addition or a subtraction that matched the input types it would have given an error.
Though a bottom up approach is chosen, VHDL compilers should use a top down approach, since all other compilers use a top down approach. By using a bottom up approach type qualifiers will be needed at different places to let the code compile. Therefore some expressions that regular VHDL compilers can evaluate cannot be evaluated by our compiler and vice versa.

5.3 The new language constructs

Now that the architecture of an easy to adapt compiler that can parse and check VHDL is created, the compiler can be augmented with the new features. The first addition is the independent compilation order. This feature can be got for free because of the flexible multi-pass set-up of the compiler.
The second addition is being able to do declarations in the body regions. First the parser has to be changed so it accepts this new structure like showed in listing 4.1. It is also important to preserve the order of the declarations. When the compiler reaches pass 2, it can register the declarations in the body region the same way as the declarations in the declaration area. The rest of the compiler does not have to be changed; when the compiler starts writing its output it will put all declarations in the declaration region and the rest in the body region.
Next there is the direct signal assignment. The BNF of the direct signal assignment is depicted in listing 5.4. When this rule is visited during pass 2, both a signal declaration object and a signal assignment statement object is created. During the following passes both objects are treated like they would if they were declared separately. So again with little effort the new feature is implemented by making use of the flexible compiler design.

```
direct_concurrent_simple_signal_assignment
    : SIGNAL target ':' subtype_indication signal_kind?
    (
        (':=' expression '<=' GUARDED? delay_mechanism? waveform ';')
        |
        ('<=' GUARDED? delay_mechanism? waveform (':=' expression)? ';')
    )
    ;
```

Listing 5.4: A simple direct signal assignment.

The assignment operator overloading feature makes use of the type checking system. The first step is done during pass 4. Once the expression of a statement

is type checked, the compiler will check if the type of the left side of the assignment matches the type on the right side of the assignment. If this is not the case, the compiler will look for an assignment overload procedure that converts the right side type to the left side type. If this is not the case an error will be given, but if such a procedure does exist, the statement is flagged so pass 5 will know that the assignment operator is overloaded. When pass 5 reaches an assignment statement that is flagged by pass 4 it will generate output that contains the assignment overload procedure. The exact output is dependent on the type of the assignment. VDHL 2008 has seven types of signal assignment statements, and three types of variable assignment statements. The assignment overloading mechanism can be used for all variable assignments and for two of the signal assignment statements. The ones that cannot be correctly converted are the release and force assignments, since force and release statements cannot be included in procedure arguments. Also the simple waveform assignment cannot be converted, since it contains delay mechanisms.
The signal assignments that can be converted sometimes need to be packed in a process. Listing 5.5 shows a conditional signal assignment, and listing 5.6 shows the output of the compiler for this assignment. Other assignments are done in a similar way.

```
1   ARCHITECTURE arch OF testEnt IS
2       signal y: integer;
3       constant u: integer := 3;
4       constant v: integer := 4;
5       constant w: integer := 5;
6
7       signal x: unsigned(7 DOWNTO 0) <=
8           u when y = 1 ELSE
9           v when y = 10 ELSE
10          w;
11  END ARCHITECTURE;
```

Listing 5.5: overloaded concurrent conditional signal assignment.

```
1   ARCHITECTURE \work.arch\ OF \work.testEnt\ IS
2       SIGNAL y :  integer;
3       CONSTANT u :  integer := 3;
4       CONSTANT v :  integer := 4;
5       CONSTANT w :  integer := 5;
6       SIGNAL x :  unsigned(7 DOWNTO 0);
7   BEGIN
8       PROCESS (ALL) IS
9       BEGIN
10          IF (y = 1) THEN
11              \<=\(x, u);
12          ELSIF (y = 10) THEN
13              \<=\(x, v);
14          ELSE
15              \<=\(x, w);
16          END IF;
17      END PROCESS;
18
19  END ARCHITECTURE;
```

Listing 5.6: Compiler output.

The signal assignment at lines 7 to 10 of listing 5.5 are converted to the process statement at lines 8 to 17 of listing 5.6. Each condition of the original code is converted to an if-else branch in the process. At each branch (at lines

11, 13 and 15) the assignment overload procedure is used to convert the integer types to the unsigned type.

The next feature is the namespace construct. The output files of the compiler are put in folders which have the same name as the library of the elements in the files. Also each namespace is put in its own folder, so it is treated like a library. If the namespace contains package elements, these are put in a package with the same name as the library. Each entity- and architecture-name is prefixed with the namespace name and separated with a dot like \NamespaceName.EntityName\. Also all the references to the entities and architectures are prefixed. In pass 2 all the namespaces are registered and put the elements that are declared inside it. The namespace class shares most of its behaviour with the package class. Therefore only a limited amount of code needs to be implemented create the namespace class. When the compiler generates the output files for the elements inside the namespace during pass 5, it does this dependent on whether the elements are in the namespace.
Finally there is the new attribute system. When the compiler encounters an attribute that is declared in the new way, it will put this attribute in a temporary list. Then, for each following element in the same scope, the target properties of the attribute is compared by the compiler, with the properties of the element. If they match, the attribute is removed from the temporary list and is registered as an attribute of the element. If the end of the scope in which the attribute is declared is reached, and the attribute is still not removed from the list, the compiler checks if the attribute can be bound to the scope. If not, the compiler will give an error to indicate it cannot bind the attribute to anything. When the compiler reaches pass 5, the attributes are generated like they where created using the original attribute system.

Chapter 6

A Use Case

The three most dangerous things in the world are a programmer with a soldering iron, a hardware type with a program patch and a user with an idea.

Rick Cook in: The Wizardry Consulted 1995

To see if the added features will have the wanted effects, the compiler needs to be tested, and its results must be compared to the original code. To do this a design of a beam-former is used that was received from Astron[12]. The design consists of about 340 design files and some test-bench files.

6.1 The beam-former

A beam-former is a device that can be used in combination with a phaced array antenna. Figure 6.1 gives a schematic representation of such a antenna array.

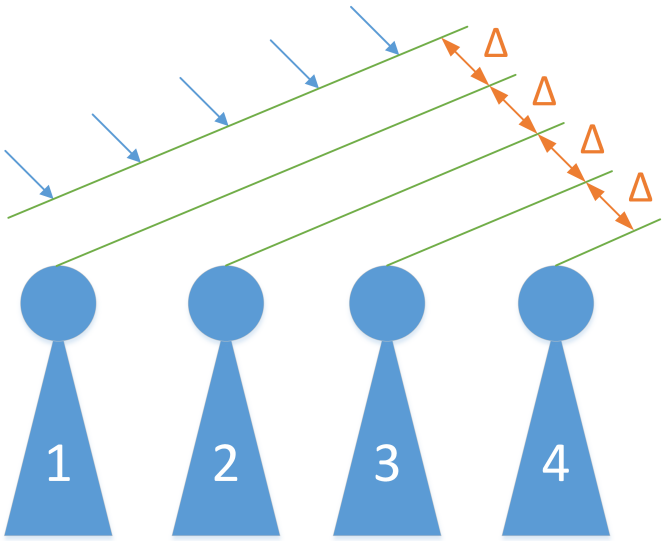

Figure 6.1: Schematic representation of a phaced-array antenna.

In this example the antennas are spaced with an equal distance. The green lines in the figure represent a wave front that is reaching the antenna array. As can be seen, the wave front reaches antenna 1 first, then antenna 2, then antenna 3 and finally antenna 4. Because the antennas are equally spaced, the delay between the signal reaching antennas 1 and 2, equals the delay between antennas 2 and 3, and antennas 3 and 4. The actual value of the delay is dependant on the propagation speed of the wave, which can be considered a constant, and the angle of the wave front relative to the orientation of the antenna array. The delay will be the largest when the wave comes in perpendicular to the antenna array, in which case the delay would be the distance between the antennas divided by the propagation speed of the wave. The other extreme is when the wave comes in parallel to the antenna array, in which case the delay will be zero. All other delays will be somewhere between these two values. These delays and angles can be interpolated, giving every angle a corresponding delay.
Because of this relation between the angle and the delay, it is possible to 'listen' in a particular direction; by delaying the incoming signals of the arrays, and adding them up, all the signals that are not coming from the chosen angle are attenuated.

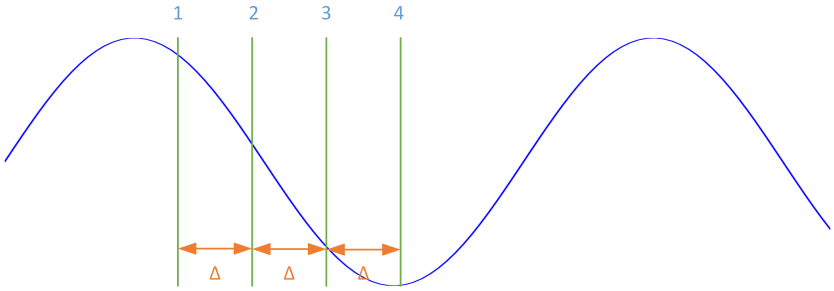

Figure 6.2: Moments in time at which the wave hits the antennas.

Because delaying signals is costly in terms of hardware, a 'trick' can be used. When looking at figure 6.2, one can see a wave and the moments at which the wave hits each antenna of figure 6.1. If the signal can be considered as a sine wave, the delay corresponds to a phase shift. So by phase shifting the incoming wave at each antenna, so that the lines of figure 6.2 will be on top of each other, the same is reached as delaying the signals.
The phase shift can be done by multiplying the incoming samples of the antennas with a range of smartly chosen constants. This is what the beam-former does. The use-case contains these multipliers, but also some statistics components and interface components.

6.2 Compiling the original design

Since our compiler does not support all features of VHDL, some files have been adapted to make them suitable for our compiler. A list of the modified files can be found in appendix B. This list also gives the reason for the modifications of the files. The following sections will shortly explain the modifications and why they were made.

Overloads on return type

Our compiler does not allow function overloading on the return type. Therefore all the functions in the use-case that are overloaded on there return type are renamed so they all have unique names.

Identifier reuse

At some places in the use-case, the same name is used for both a function and for signals. The compiler currently does not support this. It probably would not take much effort to implement this, but for the time being the signals have been renamed.

Type inversion

Because of the bottom up type checking approach, discussed in section 5, there are some places in the code where the compiler has to be told what type some literals have. This also means that there are probably some places in the code where types have been explicitly identified while our compiler does not need this, but all the other compilers do.
There are also some places where our compiler cannot determine the return type of an attribute. Here the type also has been specified. With these changes the use case can be compiled.

6.3 The test-bench

Checking if the output of our compiler is still functionally equivalent to the original code has to be done next. This is done using the Astron test-bench system. The test-bench is driven by the python language. A Python script generates the test-vectors and the control signals and sends these through files to the Modelsim environment. When the VHDL model has processed the input vectors it puts the results back in a file which is read by the Python script. In parallel, the python script does the same computations as the VHDL model, and once the output from the test-bench is read it compares the results.
When the output of our compiler is taken as the VHDL model, it can be seen that the results are the same as the original VHDL model that was received from Astron. Next parts of the code are rewritten, but now with the new language constructs.

6.4 The new features

Appendix D shows a rewritten version of one of the VHDL files of the beam-former. At line 6 the declaration of a namespace can be seen. The namespace contains an entity (line 8) and an architecture (line 28). The architecture does not contain the `begin` keyword. The architecture starts with an assert statement and is followed by two signal declarations. The second signal declaration is directly assigned a value.
The output of the compiler for the architecture of this file can be found in appendix E. The architecture does not contain any use-statements since these are

at the entity declaration. All the signal declarations and a constant declaration have been moved to the declaration part. The declarations that where directly assigned to are now split up into a separate declaration and a separate statement. The name of the architecture is prefixed with the namespace name and all references to this architecture will also use the prefixed name. Several other files also have been adapted. These files also contain the signal overloading. The new attribute system could not be tested with this use case since attributes are not used in the design. When the new files are compiled and tested with the test-bench, the results equal the other simulations.

Chapter 7

Conclusion & Future work

At the start of this thesis the question "What language constructs can be added to the VHSIC hardware description language to keep it effective and relevant for the future?" was posed. Six improvements have been shown; an independent compilation order, declarations inside body areas, direct signal assignments, assignment operator overloading, namespaces and a new attribute system. To be able to have an independent compilation order a five pass compiler has been introduced. Having multiple passes has shown to give flexibility in terms of features that can be added to the language. Also the declaration outside the body areas and the direct signal assignment features make use of this multi-pass system. Whether the code is easier to read because of these features is somewhat subjective, but in our experience it is. Being able to directly assign some signals makes the code a little shorter, but its biggest advantage is being able to see the type of the assigned signal or variable at the place it is assigned to. The namespaces have shown to work well and have some nice encapsulation properties.
Though the conclusion can be made that the language becomes more convenient to use with the added features, a side note must also be made. Because most of the added features rely on the five pass system, the additions are less likely to be adopted by the existing tool vendors since this would mean that their existing compilers would completely have to be rewritten. The use of a pre-compiler like the one that is presented in this thesis will take too much effort to completely and correctly implement, and therefore these features are not very likely to be adopted. Furthermore not many other features could be added, since the existing parts of the language make the freedom of new features quite limited. The conclusion can be made that the relevance of VHDL mostly comes from the relevance it had in the past, and the lack of competition of other languages. Not from the features it currently has or will have in the future. Therefore the future of VHDL is mostly dependent on what other languages will do.

7.1 Future work

The features that are presented in this thesis focus mostly on the structuring of the code. With the number of available transistors the need for higher abstraction levels also has grown. An example for this need is the fact that Astron

uses python scripts to check the functional behaviour of the VHDL code. It is still important however to be able to make these abstractions more concrete in small steps, which is not always trivial.
Since the possibility to add new features to VHDL is limited, it is probably a good idea to create a new language. Though many attempts for this have been made, none of them have truly succeeded. This has several reasons. First of all it is important not to make the same mistakes the designers of the current hardware description languages made. Secondly it is important that the current set of features that designers have is not limited. And finally it is important to be able to slowly integrate the language into the existing environments. This could be done by first making the language compilable to either VHDL or Verilog so that the language can be synthesized by all existing synthesis tools. One important pitfall many of the failed HDL's have made is taking a programming language as a starting point, and add features for hardware simulations. Though this gives extensive library support, many of the programming features do not translate to hardware or result in very poor hardware. To prevent confusion on what is synthesizeble and what is not, the new language should be very clear in this. All in all, such a language will be quite difficult to create and therefore VHDL will be around for some time to come.

Nomenclature

ANTLR ANother Tool for Language Recognition

ASIC Application Specific Integrated Circuit

AST Abstract Syntax Tree

BNF Backus-Naur From

EBNF Extended Backus-Naur From

FPGA Field Programmable Gate Array

HDL Hardware Description Language

RTL Register Transfer Level

SDC Synopsys Design Constraint

SDF Standard Delay Format

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

Appendices

Appendix A

Figures of chapter 5

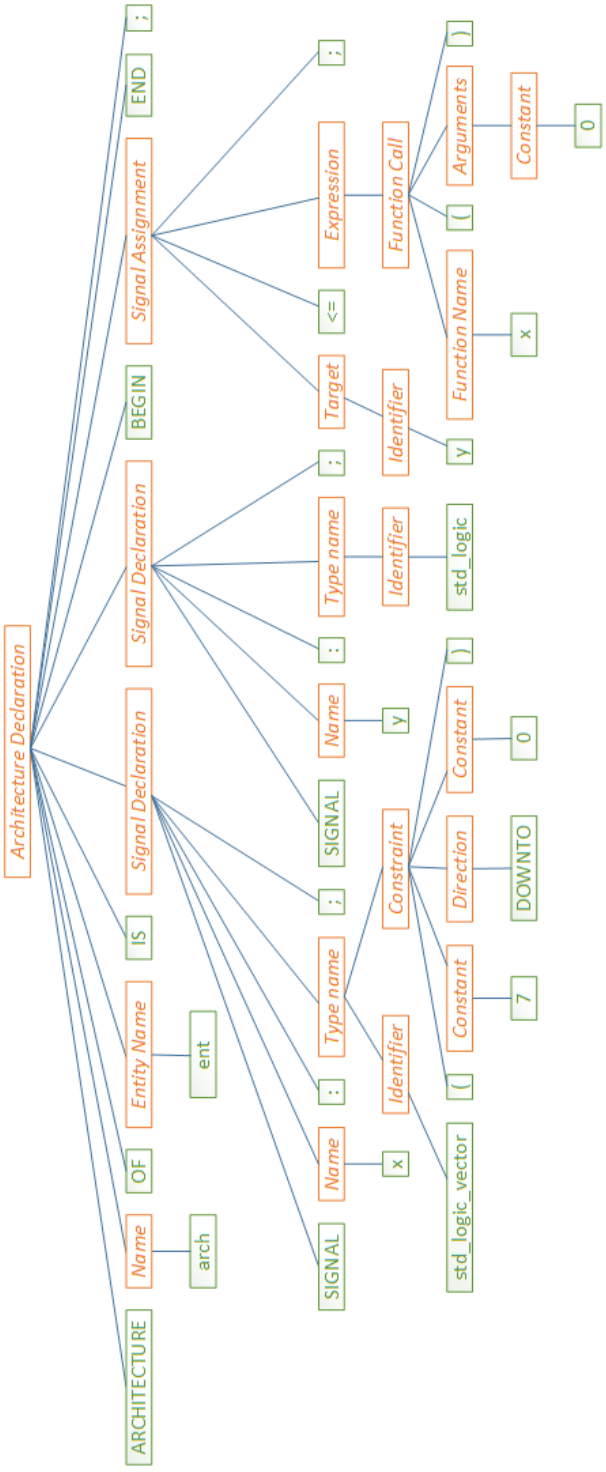

Figure A.1: Data structure after pass 1.

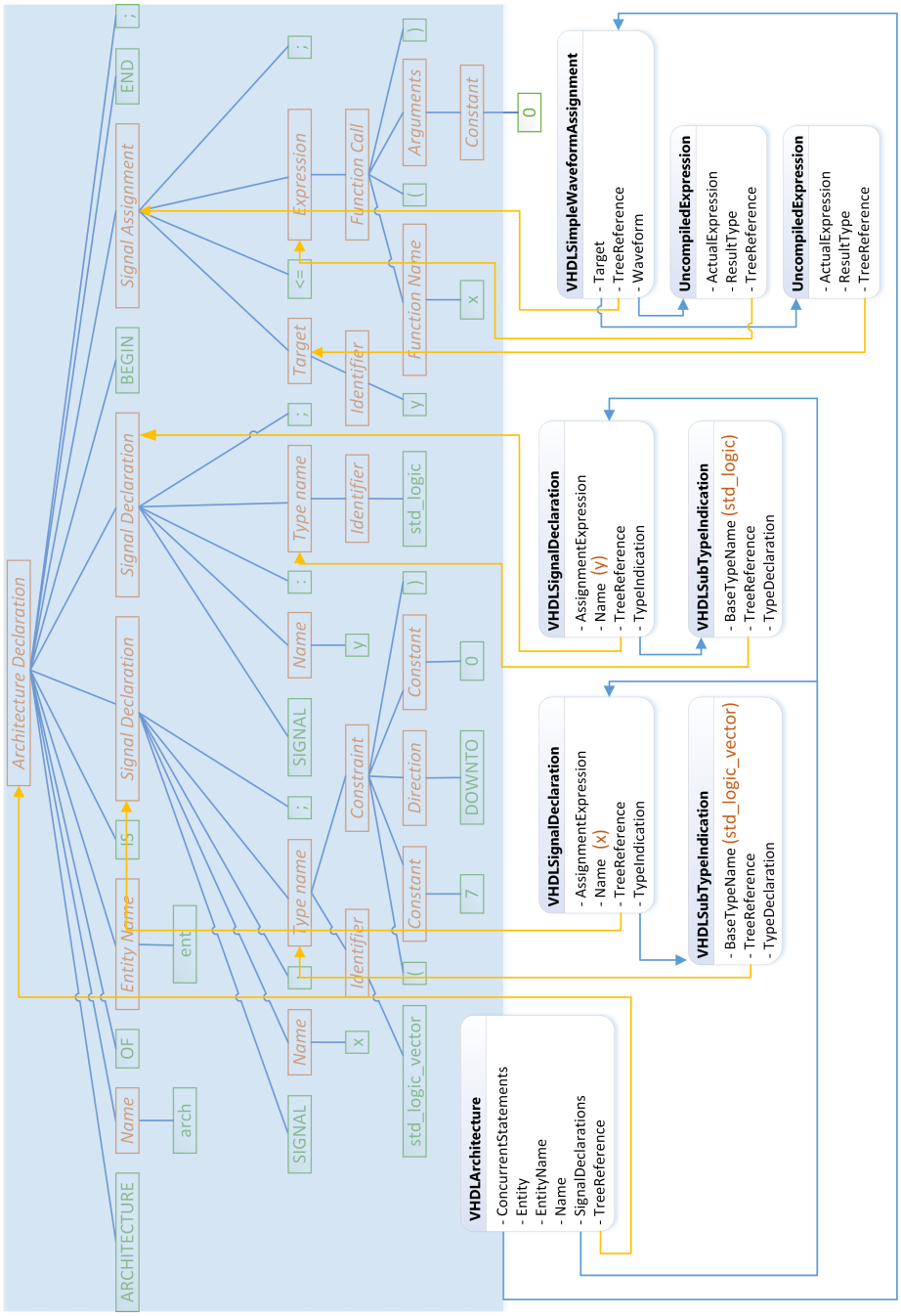

Figure A.2: Data structure after pass 2.

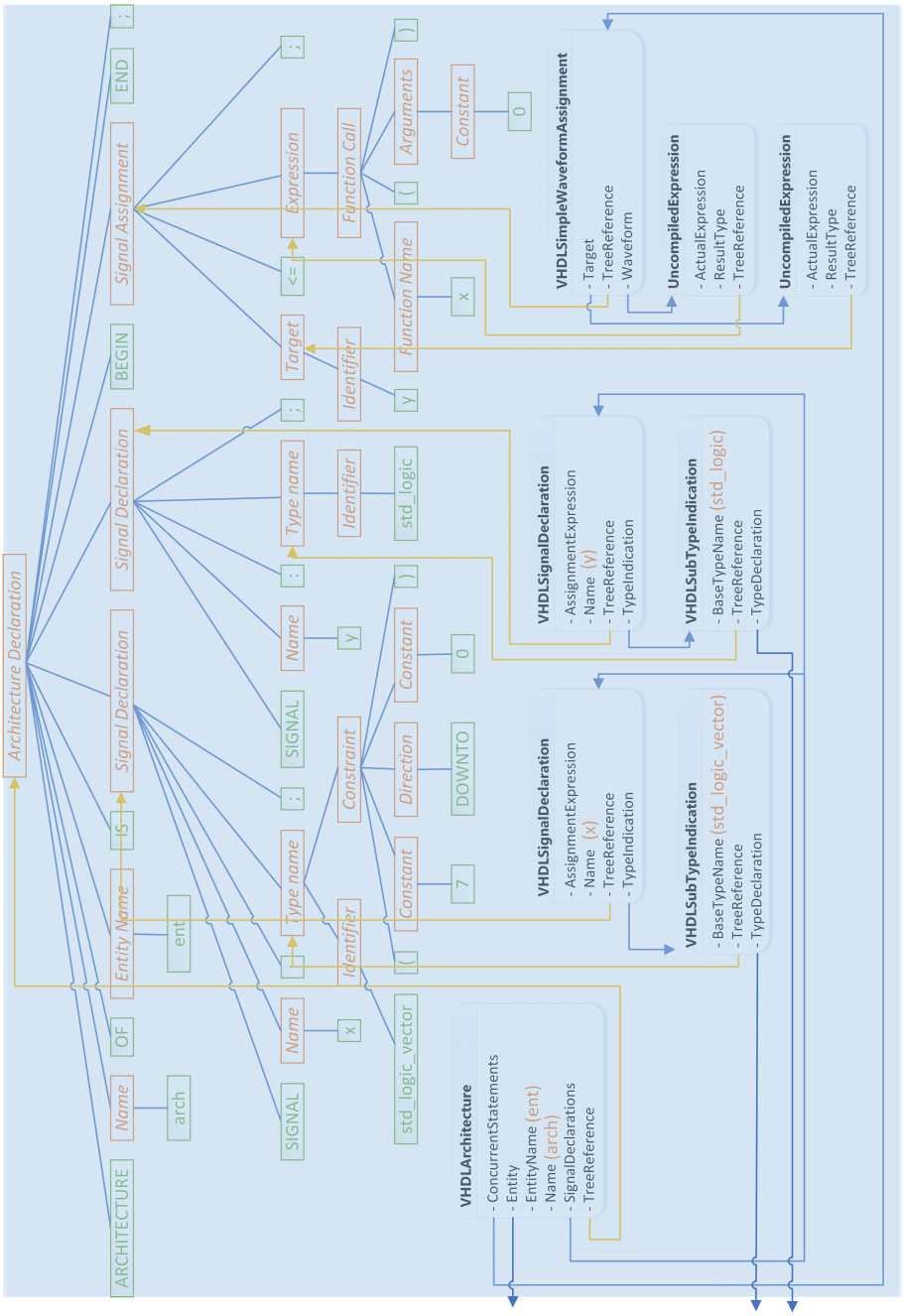

Figure A.3: Data structure after pass 3.

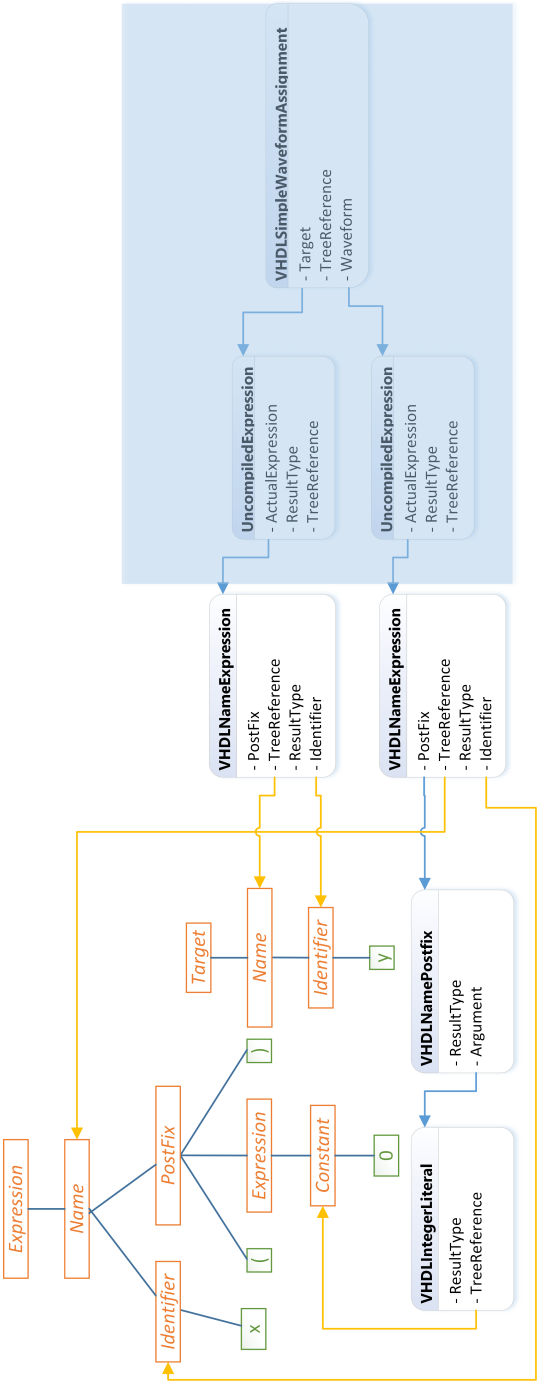

Figure A.4: Data structure of the expression after pass 4.

Appendix B

Modified files

Name	Return Overload	Id reuse	Type Inversion
common_async.vhd			X
” _clip.vhd			X
” _clock_active_detector.vhd			X
” _complex_mult.vhd			X
” _complex_mult_a_str_stratix4.vhd			X
” _debounce.vhd			X
” _evt.vhd			X
” _fifo_sc_a_stratix4.vhd			X
” _init.vhd			X
” _iobuf_in.vhd	X		
” _mult_add2_a_stratix4.vhd			X
” _pkg.vhd		X	
” _reg_r_w.vhd			X
” _reorder_symbol.vhd	X		
” _resize.vhd			X
” _stable_delayed.vhd			X
” _str_pkg.vhd	X		
dp_distribute.vhd	X		
dp_fifo_dc_mixed_widths.vhd			X
dp_mux.vhd	X		
dp_packet_enc.vhd			X
dp_stream_pkg.vhd		X	
ram_cr_cw.vhd			X
ram_crw_crw.vhd			X
ram_crwk_crw.vhd			X

Table B.1: Resolution function table of std_logic.

Appendix C

st_acc (original)

```
1  ------------------------------------------------------------------------------
2  --
3  -- Copyright (C) 2010
4  -- ASTRON (Netherlands Institute for Radio Astronomy) <http://www.astron.nl/>
5  -- P.O.Box 2, 7990 AA Dwingeloo, The Netherlands
6  --
7  -- This program is free software: you can redistribute it and/or modify
8  -- it under the terms of the GNU General Public License as published by
9  -- the Free Software Foundation, either version 3 of the License, or
10 -- (at your option) any later version.
11 --
12 -- This program is distributed in the hope that it will be useful,
13 -- but WITHOUT ANY WARRANTY; without even the implied warranty of
14 -- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15 -- GNU General Public License for more details.
16 --
17 -- You should have received a copy of the GNU General Public License
18 -- along with this program.  If not, see <http://www.gnu.org/licenses/>.
19 --
20 ------------------------------------------------------------------------------
21
22 LIBRARY IEEE, common_lib;
23 USE IEEE.std_logic_1164.ALL;
24 USE IEEE.numeric_std.ALL;
25 USE common_lib.common_pkg.ALL;
26
27
28 -- Purpose:
29 --   Accumulate input data to an accumulator that is stored externally. In this
30 --   way blocks of input samples (e.g. subband products) can be accumulated to
31 --   a set of external accumulators. At the in_load the accumulator input value
32 --   is ignored so that the accumulation restarts with the in_dat.
33 --
34 -- Description:
35 --   if in_load = '1' then
36 --     out_acc = in_dat + 0         -- restart accumulation
37 --   else
38 --     out_acc = in_dat + in_acc    -- accumulate
39 --
40 -- Remarks:
41 -- . in_val propagates to out_val after the pipeline latency but does not
42 --   affect the sum
43
44 ENTITY st_acc IS
45   GENERIC (
46     g_dat_w            : NATURAL;
47     g_acc_w            : NATURAL;  -- g_acc_w >= g_dat_w
48     g_hold_load        : BOOLEAN := TRUE;
49     g_pipeline_input   : NATURAL;  -- 0 no input registers, else register input
       after in_load
50     g_pipeline_output  : NATURAL   -- pipeline for the adder
51   );
```

```
52    PORT (
53      clk         : IN  STD_LOGIC;
54      clken       : IN  STD_LOGIC := '1';
55      in_load     : IN  STD_LOGIC;
56      in_dat      : IN  STD_LOGIC_VECTOR(g_dat_w-1 DOWNTO 0);
57      in_acc      : IN  STD_LOGIC_VECTOR(g_acc_w-1 DOWNTO 0);
58      in_val      : IN  STD_LOGIC := '1';
59      out_acc     : OUT STD_LOGIC_VECTOR(g_acc_w-1 DOWNTO 0);
60      out_val     : OUT STD_LOGIC
61    );
62  END st_acc;
63
64
65  ARCHITECTURE rtl OF st_acc IS
66
67    CONSTANT c_pipeline  : NATURAL := g_pipeline_input + g_pipeline_output;
68
69    -- Input signals
70    SIGNAL hld_load        : STD_LOGIC := '0';
71    SIGNAL nxt_hld_load    : STD_LOGIC;
72    SIGNAL acc_clr         : STD_LOGIC;
73
74    SIGNAL reg_dat         : STD_LOGIC_VECTOR(g_acc_w-1 DOWNTO 0) := (OTHERS=>'0');
75    SIGNAL nxt_reg_dat     : STD_LOGIC_VECTOR(g_acc_w-1 DOWNTO 0);
76    SIGNAL reg_acc         : STD_LOGIC_VECTOR(g_acc_w-1 DOWNTO 0) := (OTHERS=>'0');
77    SIGNAL nxt_reg_acc     : STD_LOGIC_VECTOR(g_acc_w-1 DOWNTO 0);
78
79    -- Pipeline control signals, map to slv to be able to use common_pipeline
80    SIGNAL in_val_slv      : STD_LOGIC_VECTOR(0 DOWNTO 0);
81    SIGNAL out_val_slv     : STD_LOGIC_VECTOR(0 DOWNTO 0);
82
83  BEGIN
84
85    ASSERT NOT(g_acc_w < g_dat_w)
86      REPORT "st_acc: output accumulator width must be >= input data width"
87      SEVERITY FAILURE;
88
89    ------------------------------------------------------------------------------
90    -- Input load control
91    ------------------------------------------------------------------------------
92
93    p_clk : PROCESS(clk)
94    BEGIN
95      IF rising_edge(clk) THEN
96        IF clken='1' THEN
97          hld_load <= nxt_hld_load;
98        END IF;
99      END IF;
100   END PROCESS;
101
102   nxt_hld_load <= in_load WHEN in_val='1' ELSE hld_load;
103
104   -- Hold in_load to save power by avoiding unneccessary out_acc toggling when
      in_val goes low
105   -- . For g_pipeline_input>0 this is fine
106   -- . For g_pipeline_input=0 this may cause difficulty in achieving timing closure
      for synthesis
107   use_in_load : IF g_hold_load = FALSE GENERATE
108     acc_clr <= in_load;  -- the in_load may already be extended during in_val
```

```
109     END GENERATE;
110     use_hld_load : IF g_hold_load = TRUE GENERATE
111       acc_clr <= in_load OR (hld_load AND NOT in_val);
112     END GENERATE;
113
114     -- Do not use g_pipeline_input of u_adder, to allow registered acc clear if
        g_pipeline_input=1
115     nxt_reg_dat <= RESIZE_SVEC(in_dat, g_acc_w);
116     nxt_reg_acc <= in_acc WHEN acc_clr='0' ELSE (OTHERS=>'0');
117
118     no_input_reg : IF g_pipeline_input=0 GENERATE
119       reg_dat <= nxt_reg_dat;
120       reg_acc <= nxt_reg_acc;
121     END GENERATE;
122     gen_input_reg : IF g_pipeline_input>0 GENERATE
123       p_reg : PROCESS(clk)
124       BEGIN
125         IF rising_edge(clk) THEN
126           IF clken='1' THEN
127             reg_dat <= nxt_reg_dat;
128             reg_acc <= nxt_reg_acc;
129           END IF;
130         END IF;
131       END PROCESS;
132     END GENERATE;
133
134
135     -------------------------------------------------------------------------------
136     -- Adder for the external accumulator
137     -------------------------------------------------------------------------------
138
139     u_adder : ENTITY common_lib.common_add_sub
140     GENERIC MAP (
141       g_direction       => "ADD",
142       g_representation  => "SIGNED",  -- not relevant because g_out_dat_w = g_in_dat_w
143       g_pipeline_input  => 0,
144       g_pipeline_output => g_pipeline_output,
145       g_in_dat_w        => g_acc_w,
146       g_out_dat_w       => g_acc_w
147     )
148     PORT MAP (
149       clk     => clk,
150       clken   => clken,
151       in_a    => reg_dat,
152       in_b    => reg_acc,
153       result  => out_acc
154     );
155
156
157     -------------------------------------------------------------------------------
158     -- Parallel output control pipeline
159     -------------------------------------------------------------------------------
160
161     in_val_slv(0) <= in_val;
162     out_val       <= out_val_slv(0);
163
164     u_out_val : ENTITY common_lib.common_pipeline
165     GENERIC MAP (
166       g_representation => "UNSIGNED",
```

```
167       g_pipeline       => c_pipeline,
168       g_reset_value    => 0,
169       g_in_dat_w       => 1,
170       g_out_dat_w      => 1
171     )
172     PORT MAP (
173       clk     => clk,
174       clken   => clken,
175       in_dat  => slv(in_val),
176       out_dat => out_val_slv
177     );
178
179   END rtl;
180
```

Appendix D

st_acc rewritten

```
1  LIBRARY IEEE, common_lib;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4  USE common_lib.common_pkg.ALL;
5
6  NAMESPACE st_lib IS
7
8      ENTITY st_acc IS
9          GENERIC (
10             g_dat_w            : NATURAL;
11             g_acc_w            : NATURAL;  -- g_acc_w >= g_dat_w
12             g_hold_load        : BOOLEAN := TRUE;
13             g_pipeline_input   : NATURAL;  -- 0 no input registers, else register
               input after in_load
14             g_pipeline_output  : NATURAL   -- pipeline for the adder
15         );
16         PORT (
17             clk         : IN  std_logic;
18             clken       : IN  std_logic := '1';
19             in_load     : IN  std_logic;
20             in_dat      : IN  std_logic_vector(g_dat_w - 1 DOWNTO 0);
21             in_acc      : IN  std_logic_vector(g_acc_w - 1 DOWNTO 0);
22             in_val      : IN  std_logic := '1';
23             out_acc     : OUT std_logic_vector(g_acc_w - 1 DOWNTO 0);
24             out_val     : OUT std_logic
25         );
26     END st_acc;
27
```

```
28      ARCHITECTURE rtl OF st_acc IS
29
30          ASSERT NOT(g_acc_w < g_dat_w)
31              REPORT "st_acc: output accumulator width must be >= input data width"
32              SEVERITY FAILURE;
33
34          -------------------------------------------------------------------------------
35          -- Input load control
36          -------------------------------------------------------------------------------
37          SIGNAL hld_load : std_logic := '0';
38          SIGNAL nxt_hld_load : std_logic <=
39              in_load WHEN in_val = '1' ELSE
40              hld_load;
41
42          p_clk: PROCESS(clk)
43              IF rising_edge(clk) THEN
44                  IF clken = '1' THEN
45                      hld_load <= nxt_hld_load;
46                  END IF;
47              END IF;
48          END PROCESS;
49
50          -- Hold in_load to save power by avoiding unnecessary out_acc toggling when
            in_val goes low
51          -- . For g_pipeline_input>0 this is fine
52          -- . For g_pipeline_input=0 this may cause difficulty in achieving timing
            closure for synthesis
53          SIGNAL acc_clr : std_logic;
54          use_in_load: IF g_hold_load = FALSE GENERATE
55              acc_clr <= in_load;  -- the in_load may already be extended during in_val
56          END GENERATE;
57          use_hld_load: IF g_hold_load = TRUE GENERATE
58              acc_clr <= in_load OR (hld_load AND NOT in_val);
59          END GENERATE;
60
61          -- Do not use g_pipeline_input of u_adder, to allow registered acc clear if
            g_pipeline_input=1
62          SIGNAL nxt_reg_dat : std_logic_vector(g_acc_w - 1 DOWNTO 0) <= RESIZE_SVEC(
            in_dat, g_acc_w);
63          SIGNAL nxt_reg_acc : std_logic_vector(g_acc_w - 1 DOWNTO 0) <=
64              in_acc WHEN acc_clr = '0' ELSE
65              (OTHERS => '0');
66
67          SIGNAL reg_dat : std_logic_vector(g_acc_w - 1 DOWNTO 0) := (OTHERS => '0');
68          SIGNAL reg_acc : std_logic_vector(g_acc_w - 1 DOWNTO 0) := (OTHERS => '0');
69          no_input_reg: IF g_pipeline_input = 0 GENERATE
70              reg_dat <= nxt_reg_dat;
71              reg_acc <= nxt_reg_acc;
72          END GENERATE;
73
74
75
```

```
76          gen_input_reg: IF g_pipeline_input > 0 GENERATE
77              p_reg: PROCESS(clk)
78                  IF rising_edge(clk) THEN
79                      IF clken = '1' THEN
80                          reg_dat <= nxt_reg_dat;
81                          reg_acc <= nxt_reg_acc;
82                      END IF;
83                  END IF;
84              END PROCESS;
85          END GENERATE;
86
87          -------------------------------------------------------------------------------
88          -- Adder for the external accumulator
89          -------------------------------------------------------------------------------
90          u_adder : ENTITY common_lib.common_add_sub
91              GENERIC MAP (
92                  g_direction       => "ADD",
93                  g_representation  => "SIGNED", -- not relevant
94                  g_pipeline_input  => 0,
95                  g_pipeline_output => g_pipeline_output,
96                  g_in_dat_w        => g_acc_w,
97                  g_out_dat_w       => g_acc_w
98              )
99              PORT MAP (
100                 clk     => clk,
101                 clken   => clken,
102                 in_a    => reg_dat,
103                 in_b    => reg_acc,
104                 result  => out_acc
105             );
106
107         -------------------------------------------------------------------------------
108         -- Parallel output control pipeline
109         -------------------------------------------------------------------------------
110         SIGNAL in_val_slv   : std_logic_vector(0 DOWNTO 0);
111         SIGNAL out_val_slv  : std_logic_vector(0 DOWNTO 0);
112         CONSTANT c_pipeline : natural := g_pipeline_input + g_pipeline_output;
113
114         in_val_slv(0) <= in_val;
115         out_val       <= out_val_slv(0);
116
117         u_out_val : ENTITY common_lib.common_pipeline
118         GENERIC MAP (
119             g_representation => "UNSIGNED",
120             g_pipeline       => c_pipeline,
121             g_reset_value    => 0,
122             g_in_dat_w       => 1,
123             g_out_dat_w      => 1
124         )
125         PORT MAP (
126             clk     => clk,
127             clken   => clken,
128             in_dat  => slv(in_val),
129             out_dat => out_val_slv
130         );
131
132     END rtl;
133 END NAMESPACE;
```

Appendix E

st_acc rewritten compiler output

```
1
2  ARCHITECTURE \st_lib.rtl\ OF \st_lib.st_acc\ IS
3      SIGNAL hld_load :  std_logic := '0';
4      SIGNAL acc_clr :  std_logic;
5      SIGNAL reg_dat :  std_logic_vector (g_acc_w - 1 DOWNTO 0) := (OTHERS => '0');
6      SIGNAL reg_acc :  std_logic_vector (g_acc_w - 1 DOWNTO 0) := (OTHERS => '0');
7      SIGNAL in_val_slv :  std_logic_vector (0 DOWNTO 0);
8      SIGNAL out_val_slv :  std_logic_vector (0 DOWNTO 0);
9      CONSTANT c_pipeline :  natural := g_pipeline_input + g_pipeline_output;
10     SIGNAL nxt_hld_load :  std_logic;
11     SIGNAL nxt_reg_dat :  std_logic_vector (g_acc_w - 1 DOWNTO 0);
12     SIGNAL nxt_reg_acc :  std_logic_vector (g_acc_w - 1 DOWNTO 0);
13 BEGIN
14     ASSERT
15         not ((g_acc_w < g_dat_w))
16     REPORT
17         "st_acc: output accumulator width must be >= input data width"
18     SEVERITY
19         FAILURE;
20     nxt_hld_load <= in_load WHEN (in_val = '1') ELSE
21         hld_load;
22
23     p_clk: PROCESS(clk) IS
24     BEGIN
25         IF rising_edge(clk) THEN
26             IF (clken = '1') THEN
27                 hld_load <= nxt_hld_load;
28             END IF;
29         END IF;
30     END PROCESS;
31
32     use_in_load: IF (g_hold_load = FALSE) GENERATE
33         BEGIN
34             acc_clr <= in_load;
35     END GENERATE;
```

```
36      use_hld_load: IF (g_hold_load = TRUE) GENERATE
37          BEGIN
38              acc_clr <= in_load or (hld_load and not in_val);
39      END GENERATE;
40      nxt_reg_dat <= RESIZE_SVEC(in_dat, g_acc_w);
41      nxt_reg_acc <= in_acc WHEN (acc_clr = '0') ELSE
42          (OTHERS => '0');
43
44      no_input_reg: IF (g_pipeline_input = 0) GENERATE
45          BEGIN
46              reg_dat <= nxt_reg_dat;
47              reg_acc <= nxt_reg_acc;
48      END GENERATE;
49      gen_input_reg: IF (g_pipeline_input > 0) GENERATE
50          BEGIN
51              p_reg: PROCESS(clk) IS
52              BEGIN
53                  IF rising_edge(clk) THEN
54                      IF (clken = '1') THEN
55                          reg_dat <= nxt_reg_dat;
56                          reg_acc <= nxt_reg_acc;
57                      END IF;
58                  END IF;
59              END PROCESS;
60
61      END GENERATE;
62      u_adder: ENTITY common_lib.common_add_sub
63          GENERIC MAP(
64              g_direction => "ADD",
65              g_representation => "SIGNED",
66              g_pipeline_input => 0,
67              g_pipeline_output => g_pipeline_output,
68              g_in_dat_w => g_acc_w,
69              g_out_dat_w => g_acc_w)
70          PORT MAP(
71              clk => clk,
72              clken => clken,
73              in_a => reg_dat,
74              in_b => reg_acc,
75              result => out_acc)
76      ;
77      in_val_slv(0) <= in_val;
78      out_val <= out_val_slv(0);
79      u_out_val: ENTITY common_lib.common_pipeline
80          GENERIC MAP(
81              g_representation => "UNSIGNED",
82              g_pipeline => c_pipeline,
83              g_reset_value => 0,
84              g_in_dat_w => 1,
85              g_out_dat_w => 1)
86          PORT MAP(
87              clk => clk,
88              clken => clken,
89              in_dat => slv(in_val),
90              out_dat => out_val_slv)
91      ;
92  END ARCHITECTURE;
```

Bibliography

[1] Doulos *A Brief History of VHDL* URL `http://www.doulos.com/knowhow/vhdl_designers_guide/a_brief_history_of_vhdl/`

[2] IEEE Computer Society *IEEE Standard VHDL Language Reference Manual*. New York, Revision of IEEE Std 1076-2002, 2009.

[3] NVIDIA *NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110* 2012-11-02

[4] IEEE Computer Society *IEEE Standard for Verilog® Hardware Description Language*. New York, Revision of IEEE Std 1364-2001, 2006.

[5] IEEE Computer Society *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*. New York, IEEE Std 1800-2009, 2009

[6] International organization for standardization *Syntactic metalanguage - Extended BNF* 1996-12-15 ISO/IEC 14977

[7] Karen Bartleson *The Ten Commandments for Effective Standards* 2010 ISBN13: 9781617300004

[8] Wikipedia *Strong and weak typing* `http://en.wikipedia.org/w/index.php?title=Strong_and_weak_typing&oldid=616527316` Received on 22 July 2014

[9] Synopsys, Inc. Intrinsix Corp. *SDC Parser User's Manual*

[10] IEEE Computer Society and the International Electrotechnical Commission *Standard Delay Format (SDF) for the electronic design process* IEC 61523-3:2004/IEEE 1497-2001

[11] Terence Parr *The Definitive ANTLR 4 Reference*. ISBN-13: 978-1-93435-699-9

[12] *ASTRON Netherlands Institute for Radio Astronomy* http://www.astron.nl/