

# Evaluation of Sequential VHDL and C for System Description and Specification

Matthias Bauer, Wolfgang Ecker / Michael Gasteier, Manfred Glesner  
Siemens AG, ZFE T SE 5 / TH Darmstadt, Institut für Datentechnik, FG Mikroelektronische Systeme  
D-81730 München, Otto-Hahn-Ring 6 / D-64283 Darmstadt, Karlstraße 15

## Abstract

VHDL and C play an important role in nowadays system design. In order to examine the applicability of these two languages for system development, we mainly focus on performance aspects but also compare the language capabilities of VHDL and C. Language related benchmarks as well as DSP applications were used for examination.

We detected an about 5 times higher execution performance of C programs compared to similar VHDL models. This difference partially results from bit operations on integer and pointers to all explicitly and implicitly allocated pieces of memory, which are available in C but not in VHDL. But we also found a performance weakness of VHDL-tools primarily due to the lack of data-flow analysis based code optimization, automatic index pointer transformation, and automatic subroutine in-lining.

## 1 Introduction

VHDL and VHDL tools have already been analysed under a lot of different aspects. Simulation performance [Röh95] or synthesis subsets [SeEV95] were investigated to achieve information about the best tools under some certain aspects. An analysis of the overhead inherent to VHDL can be found in [LeMR94]. Features of VHDL have been analysed or compared with other hardware description languages [AyWS86], [Magi92] or system level specification methods [NaGa93].

A comparison between VHDL and C seems on the first point of view useless due to their different application domains. But C for example is used in some (VHDL) simulators to describe gate level library models or in the COSSAP [Kunk91] simulation system as behavioral language. Our intent is quite different. In [GIGM96] we have proposed a design flow for hardware/software co-design starting with a system description consisting of a single VHDL architecture containing multiple processes and a set of C programs linked to the VHDL world via named pipes. This approach requires co-simulation mechanisms and code transformations when repartitioning the system. An alternative approach could use one common description language, for example either VHDL or C for the initial executable specification of the system to reduce efficiency loss due to different languages. Thus a comparison of these to languages is required in order to evaluate their capabilities for system specification.

General language aspects, the type mechanism, and sequential statements are evaluated in a first step. A comparative performance analysis of both languages which is very important for computation intensive applications like DSP follows. Hardware specific features of VHDL like structure, multiple drivers, or timing may be reasons for selecting VHDL as specification language in early design steps. They are however not considered in this paper due to the fact that they are not available in C.

## 2 Language Comparison

For objection if VHDL and/or C are suited to be used as a single language for system description in hardware software co-design, it is useful to compare these languages to identify the common ground and main differences. Results of this comparison are also important for the explanation of different results in performance evaluation.

### 2.1 General Aspects

One important characteristic of both languages is besides the wide spread use that they are standardized: VHDL by the IEEE-1076 standard, C by the ANSI standard X3.159-1989. However, VHDL was developed by a consortium financed by the DoD with the intent to make a standard HDL whereas C was developed by a group of individuals.

VHDL and C were designed for two completely different purposes: While VHDL is a language for describing hardware, C was originally developed for programming of operating systems [KeRi88]. C allows very short and concise programming. Its control flow is purely sequential, which is sufficient for the initial application

domain<sup>1</sup>. Since hardware implies concepts like parallel execution and hierarchical structure, VHDL, on the other hand, has to support a much wider range of concepts including additional aspects like timing and structure. This HW-related aspects however are not considered in this paper as already mentioned.

An additional intention in development of VHDL was to replace the often tens of thousands pages of natural language documentation by a more precise, executable description. Therefore, some short but sometimes cryptic constructs provided in C are not part of VHDL.

The programming language concepts for VHDL including hierarchy, types and sequential statements were derived from ADA [Barn91].

## 2.2 Language Aspects

Structuring and encapsulation mechanisms in VHDL are much more powerful than the ones of C. On the top, VHDL specifies a set of logical libraries while C doesn't. Each VHDL library contains source file independent design units, especially packages and package bodies. Functions in C are always declared at the top level and therefore known by all other functions within a file. They can be collected in C libraries which are very similar to VHDL packages but strongly differ with VHDL libraries.

VHDL enables procedure and function hiding by declaration of subprograms within entities, architectures, packages and other subprograms similar to e.g. Pascal [JeWi85]. This mechanism is extended further by the possibility to select specific items from a library or a package, whereas in C libraries are accessed as a whole. One more difference stems from the way in which library functions are included. While VHDL installs a kind of link via the **use clause**, C includes header files and links the appropriate library files afterwards, resulting in an overhead of code to be compiled. Furthermore, the necessity of compiling higher level modules after modification of lower level modules can be avoided in VHDL by separating entities and architectures as well as package declarations and package bodies into different files.

## 2.3 Types and Type Mechanism

Both languages support type checking mechanisms, though in C they are much weaker than in VHDL. C is able to interpret the contents of memory locations according to all available types, thus allowing general type conversion, whereas VHDL restricts implicit and predefined type conversion to closely related types. The supported basic types are very similar: VHDL and C provide multiple scalar types (character, integer, real, enumerations etc.), composite types, array types and pointers (access types in VHDL). Due to the possibility to extend the set of supported types by creation of user defined types, most existing differences between the two languages regarding data types can be resolved. The predefined VHDL type **Boolean** for example is not provided in C, but can easily be defined by the user. Also the VHDL type **severity level** does not exist in C.

Operations on floating point and complex objects are supported contemporarily in both languages. However, C libraries and VHDL packages exist to close this gap. Correct results of exhaustive floating point number based computation can be assured in VHDL less than in C due to the small precision required for floating point objects. C on the other hand supports two precision classes of real objects which both provide a much higher degree of precision than those in VHDL.

As mentioned above, VHDL and C support integer types but VHDL shows some weakness as already described in [Eck95]. C provides more hardware oriented integer arithmetic operations. In case of overflow the results were cur, whereas VHDL breaks execution due to range violation.

Additionally to VHDL C supports bit interpretation of integer values including bit-wise logical as well as shift operations. Moreover, C supports a full 32-bit two's complement arithmetic whereas VHDL requires a symmetric integer range and thus not the number  $-2^{*}32$ . Due to its hardware relation, VHDL has predefined bit and bit chain types with predefined logical, shift, and slice operators. A package containing integer operations on bit chains is currently under ballot.

To ease the declaration of user defined types, VHDL supports the concept of subtypes which allow introduction of

---

1. Parallel programming in C can of course be done by usage of system calls, this is, however, not a feature of the language itself, but the underlying operating system.

new range restrictions in a comfortable way. Furthermore, all operations declared for a data type are also available for all subtypes derived from this data type.

Declaration of new type specific operators is also possible (overloading), whereas in C one has to define new function names to execute the required operations on user defined data types.

Information about objects and types, e.g. array boundaries etc. can be obtained in VHDL using attributes. These allow the declaration of procedures and functions containing unconstrained arrays as parameters. To achieve the same functionality in C, one has either to specify a specific function for each of the possible array sizes, pass additionally range information to the subroutine or declare a record containing the array and range information.

## 2.4 Objects

All data values manipulated by a VHDL or C program are stored in objects. While C supports two different object classes, constants and variables, VHDL supports a third hardware oriented object class, the so-called signals. Constants have same semantics in both languages whereas variables haven't. Variables may be initialized in both languages, but VHDL also defines an implicit initial value. Variables of class `auto` can be local to a subroutine and will loose their value if the subroutine is left. C also provides a class `static` which keeps the old value when re-entering a subroutine. Variables defined outside of a subroutine are globally available and keep their values until they were re-assigned in both languages but variables loose their value. In difference they keep their value if they are from storage class `static`. The storage class `register` exists in C but not in VHDL. However this class specifies an optimization directive only without any operational semantic.

VHDL provides some comfortable operations for data access which are not included in C. These are based on the concept of ranges and slices. A range is specified by two expressions which determine the boundaries of the range and an additional direction. Accessing an array using a range as index results in a slice, a set of subsequent array elements whose indices lie within the range boundaries. Slices can be used in assignments, allowing comfortable array operations. Aggregates, composite values used for initialization of arrays, may also refer to ranges. C, on the other hand, is much more powerful in addressing dynamic memory. Pointers are also available in VHDL (access types), they are, however, restricted to point only to objects, which were created using the alligator `new`, whereas C provides pointer arithmetic and pointers to any arbitrary memory locations.

## 2.5 Statements

According to the main intent of VHDL to describe hardware, a process which is the interface between the concurrent and sequential domain executes an implicit infinite loop. The statement `wait` without any sensitivity must be inserted to terminate the statement execution of a process. On the other hand, a C program terminates after the execution of the last sequential statement. Infinite execution must be described explicitly by a loop statement.

While in C the only way for grouping statements is a function, VHDL provides both procedures and functions. The VHDL function however is pure i.e. it accepts input values only and does not allow for side effects as might occur in C. Parameter handling is also slightly different. Both pass their parameter by value but C passes its parameter only to the subroutine whereas VHDL passes its parameters also back from the subroutine if they are of mode `out` or `inout`. Modifying a value passed as parameter in C is not directly possible, one has to pass a pointer to this value instead. It should be noted here that in case of composite types VHDL allows for call by reference also.

Grouping of statements can be achieved in C additionally in a very performant way by using parametrizable macros. They are in-line expanded before compilation and thus need no execution of code for subroutine call.

The statements for describing the control flow within a process, function or procedure are very similar in VHDL and C:

- Conditional execution is achieved by the `if`-statement. VHDL provides an `elsif` statement which resolves an existing shift reduce conflict during compilation by assigning the current else branch to the innermost then branch. C reaches the same behaviour by implicit compilation rules.  
VHDL requires a boolean expression as condition whereas C allows any scalar type including pointers and interprets a non zero value as true.
- A selection of one out of multiple choices can be executed by the `case` (VHDL) and `switch` (C) statement. VHDL

allows ranges and enumerations for the different branches, whereas C may execute multiple branches by omitting the **break**-statement.

Choices in C are restricted to scalar objects and values whereas VHDL allows for composite types and objects also.

- For iterative control VHDL provides **loop**, **for**, **while**, **exit**, **next**, **exit when** and **next when** statements. The corresponding behaviour can be achieved in C using **for**, **while**, **do-while**, **break**, and **continue** statements. VHDL is able to leave an arbitrary number of nested levels when terminating a loop. This can be modelled in C by usage of the **goto** statement which however does not exist in VHDL.

The loop index in VHDL is implicitly declared, but may be a by one incremented or decremented scalar type only. It is also a constant object inside the loop body. C additionally allows multiple loop indices, pointers as loop indices, and the modification of the loop index inside the loop body.

- The assertion statement of VHDL is contained in a C standard library.

## 2.6 File-IO

Both, VHDL and C support binary as well as ASCII reading and writing of files. Binary files written in VHDL are however not compatible between different VHDL tools and between VHDL tools and C programs. Thus data can be exchanged between VHDL and C via ASCII files<sup>1</sup> only.

VHDL has no explicit operation for opening and closing of files as C does. Opening files during simulation can be done in VHDL only by declaring a file inside a procedure and calling this procedure.

ASCII file access is in VHDL line<sup>2</sup> based and in C character based. VHDL thus first reads a line and then extracts required data from that line. Vice versa, data is converted into a line and this line is then written to a file. In difference, C allows for reading character wise data from a file. Read and write operation is buffered and relies on parametrizable macros. Higher level operations for reading and formatted writing of strings, characters and numbers are also part of C.

Moreover, C programs can control file access by using operating system calls. Applications are the specification of buffer size or flushing of buffer contents.

## 3 Performance Aspects

### 3.1 Test Cases

Our main intention is to compare the languages VHDL and C and no tools. The table in the appendix contains for this reason no information about tool vendors. It should be noted, however, that different tools with different simulation approaches (interpreter, generated C code, native code) from different vendors have been used for our examination. Both, VHDL and C tools have been evaluated in optimized and not optimized mode. Additionally the omission of VHDL range check was considered to compare exactly VHDL and C behaviour. This gained in an about 10 to 20% higher VHDL simulation performance. Detailed information about test cases is given in Section 6 "Benchmarks".

### 3.2 Analysis and Compilation

VHDL analyser and C compiler showed nearly the same performance. Sometimes the C compiler and sometimes the VHDL analyser was in maximum the factor 2 slower. The VHDL overhead in symbol handling resulting from hierarchical name spaces seems to be compensatable by module wise compilation into an intermediate and storing as well as accessing the intermediate.

An exception was the analysis of pure sequential statements. In optimized mode the C compiler required here a 500 times higher compilation time. We assume that data-flow analysis is performed in this case by the C compiler due to

- 
1. It is also possible to generate VHDL or C code containing tables, compiling this tables and linking this tables into a VHDL or C program. This method is however not suitable for a huge amount of data.
  2. A line is in VHDL a pointer to a string. The dynamic handling of this string is hidden from the user inside the read and write operations of VHDL. The string contains all characters until the next new line symbol.

the extremely high execution speed achieved for the data-flow benchmarks in this case.

### 3.3 Execution

The VHDL interpreter was about the factor 100 slower than executable programs and is for this reason not applicable for system evaluation based on larger behavioral simulation runs.

Other VHDL simulation runs needed in average the factor 5/10 (native code/compiled code) more time for the execution of control dominated benchmarks (loops, if, case, subroutine calls). In detail:

- Handling of recursive functions was the only case where VHDL tools execute the code faster.
- If and case branches execute in C two times faster than in VHDL.
- Probably the detection of loop invariant operations is the reason for the 50 times faster execution of the C program in case of the loop statement benchmark.
- The factor three higher C performance accessing arrays via loop indices results from an index/pointer transformation performed by the C compiler<sup>1</sup>.
- The measurable performance of the C program executing a sequence of 1000 statements was achieved by data flow considering code optimization.
- Accessing data via binary files depends on the used language and tool and performs between 6 to 37 times faster than for ascii files. In general, C showed a factor 2 to 15 faster file access than VHDL for binary write access.

We analysed also a sort algorithm and four real DSP applications. The execution of the VHDL models needed in these cases 2 - 20 times the time of C. The greatest difference occurred in an optimized model based on integer values only. A part of this overhead results from a missing bit interpretation and manipulation of VHDL integer types and the possibility to assign the address of an arbitrary array element to a C pointer. A complete behavioral simulation of a MPEG video part was the factor 10 slower than the equivalent C program.

The native code simulator was in our tests in most cases slower than the simulator which used generated C code. This may stem from the optimization capabilities of the C-compiler.

### 4 Conclusion

As expected, C programs can be executed faster than comparable VHDL programs. The measured difference of a factor 5 to 10, however, is much larger than assumed and has in our opinion mainly no language based reasons. It shows potential but also need for performance optimization of VHDL simulators in the field of behavioral descriptions. It may be also the reason for the use of C instead of VHDL for system level models under critical performance requirements.

A good interaction between VHDL and C is a must from the HW/SW co-design point of view. Even if VHDL and C show comparable capabilities in the domain of behavioral respectively sequential descriptions one language will never replace the other in their domain: C lacks the missing possibility to express concurrent behaviour and the software design process is too much C dominated.

Current possibilities for VHDL/C interfacing other than data exchange via ASCII files, however, are mainly simulator oriented, tool depended, and do not respect the application of C as algorithmic description language: Either the binding of C subroutines to VHDL subroutines is not possible at all or shows awful performance. The definition of the VHDL attribute FOREIGN is only a first small step. The definition of a VHDL/C interface semantic is a must for near future standardization activities.

Also no standardized practice for VHDL/C co-execution exists. All current approaches are based on TCP/IP and are simulator dependent due to the necessary VHDL/C interface. To overcome this problem, we presented in [GIGM96] a tool independent approach based on UNIX pipes. Performance optimization of VHDL based text-IO or clearly defined binary-IO could help to improve this portable method.

Our future work will concentrate on the comparison of Ada95 and VHDL with the goal to verify the assumption that the performance difference between C and VHDL has not mainly language reasons.

---

1. The C program showed exactly the same performance as another C program which explicitly accesses the array by pointer.

## 5 Bibliography

- [AES86] J. Ayler, R. Waxman, C. Scarrat: *VHDL Feature Description and Analysis*. IEEE Design & Test of Computers, 4, 1986, pp. 17-27.
- [Bar91] J. Barnes. *Programming in Ada plus Language Reference Manual*. Addison-Wesley Publishing Company, Workham, UK, 1991.
- [Eck95] W. Ecker. *VHDL Integer Package - A Call for a New VHDL Companion Standard*. In Proc. of the VIUF Spring 1995 Conference, 1995.
- [GaGl96] M. Gasteier, M. Glesner: Co-simulation of Mixed HW/SW Systems (orig. Cosimulation gemischter HW/SW-Systeme). In M. Glesner, editor, *Hardwarebeschreibungssprachen und Modellierungsparadigmen: 2. GI/ITG/GME Workshop*, pp 60-69, Feb. 1996, Shaker Verlag.
- [JW85] K. Jensen, N. Wirth: *Pascal User Manual and Report*. Springer Verlag, New York, 1985.
- [KR88] B.W. Kernighan, D.M. Ritchie. *The C Programming Language*. Second Edition. Prentice-Hall International, New Jersey, 1988.
- [Kunk91] J. Kunkel. *COSSAP: A Stream Driven Simulator*. In Proc. of Int. Workshop on Microelectronics in Communication, 1991.
- [LeMR94] O. Levia, S. Maginot, J. Rouillard. *Lessons in Language Design: Cost/Benefit Analysis of VHDL Features*. In Proc. of the DAC'94, 1994.
- [Magi92] S. Maginot. *Evaluation Criteria of HDLs: VHDL compared to VERILOG, UDL/I & M*. In Proc. of the EURODAC/VHDL'92, 1992.
- [NaGa93] S. Narayan, D. Gajski: *Features Supporting System Level Specifications in HDLs*. In Proc. of the EURODAC/VHDL'93, pp. 540-545, Hamburg, September 1993.
- [Röh95] E. Röhm. *Latest Simulator Benchmarks*. In Proc. of the EURODAC/VHDL'95, pp. 406-411, Brighton, September 1995.
- [SeEV95] M. Selz, W. Ecker, E. Villar: *VHDL Synthesis Description portability. The need for Level-x Synthesis Subsets*. In Proc. of the VHDL-FORUM 1995 Spring Meeting, 1995.

## 6 Benchmarks

Equivalent C and VHDL models have been used for benchmarking whereas pieces of code which could not directly be mapped from one language onto another have been slightly modified. Thus all pointer access to array elements as it was possible in C was modeled by explicit index access and index arithmetic in VHDL. Also predefined operations existing in one language only have been implemented as subroutine for use in the model of the other language. Tables containing detailed data from performance evaluation are built up as follows: The first column in the tables specifies the name of the benchmark. Column 2 to 6 represent the data of the VHDL models executed with an interpreter (2), a compiled code simulator (3), a compiled code simulator with optimization including the removal of all range checks (4), a native code simulator (5), and a native code simulator with optimization including the removal of all range checks (6). The last two columns represent the C model. Column (7) with not optimized and column (8) with optimized compile mode.

### 6.1 Benchmark Description

Language/Mode Benchmark	Description
recursive Functions	Power function, which is implemented by recursive call of multiplications. These multiplications are implemented recursively also by additions. This additions are finally implemented recursively also.
case Statements	Execution of all x branches of y levels of case statements
if Statements	Execution of all x branches of y levels of if statements
loop Statements	Execution of x levels of loops
array <sup>1-dim</sup> access	Access to a one dimensional array of integer
array <sup>2-dim</sup> access	Access to a two dimensional array of integer

Language/Mode Benchmark	Description
records	Access to records
Scalar Access (4*250)*10**5	10000 times execution of 1000 sequential statements assigning and reading to scalar variables and performing +,-/, and * integer operations
Array Access (4*250)*10**5	10000 times execution of 1000 sequential statements assigning and reading to array elements and performing +,-/, and * integer operations
Rightshift	Execution of right shift operations on positive as well as negative operations (no multiplication by 2**n)
AsciiRead	Reading 2*10**6 integer values from an ascii file
AsciiWrite	Writing 2*10**6 integer values to an ascii file
BinaryRead	Reading 2*10**6 integer values from a binary file
BinaryWrite	Writing 2*10**6 integer values to a binary file
Sort.	Bubble Sort
Idct	Optimized and integer based inverse discrete cosine transformation due to MPEG2 C reference model (Cheng wang Algorithm)
Idctf	Inverse discrete cosine transformation based on floating point numbers due to MPEG2 C reference model
Idctref	Not optimized inverse discrete cosine transformation due to MPEG2 C reference model
Getvlc	Inverse discrete cosine transformation due to MPEG2 C reference model

## 6.2 Compilation Time

Language Benchmark	Interpreter	C-Code	C-Code Optimized	Native Code	Native Code Optimized	C	C Optimized
recursive Functions	0.8	1.0	1.1	1.5	1.4	0.9	0.9
case Statements	0.4	0.6	0.6	0.8	0.7	1.1	1.3
if Statements	0.5	0.8	0.8	0.9	0.9	1.3	2.0
loop Statements	0.3	0.4	0.4	0.6	0.6	0.9	1.0
array access 1 dim	0.3	0.4	0.4	0.6	0.6	1.0	1.3
array access 2-dim	0.3	0.4	0.4	0.6	0.6	1.0	1.3
records	1.2	1.4	1.4	108.7	109.2	1.9	10.7
Scalar Access (4*250)*10**5	2.9	4.9	4.7	3.6	3.2	7.4	20.2
Array Access (4*250)*10**5	4.3	9.5	10.1	6.6	6.1	7.4	110.1

Language Benchmark	Interpreter	C-Code	C-Code Optimized	Native Code	Native Code Optimized	C	C Optimized
Rightshift	0.3	0.4	0.4	0.6	0.5	0.9	0.8
Sort	0.4	0.5	0.5	0.7	0.7	1.1	1.8
Idct	0.9	1.6	1.6	1.3	1.1	2.2	4.3
Idctf	0.9	1.7	1.6	1.2	1.3	2.3	4.0
Idctref	0.6	1.1	1.1	0.9	0.9	1.5	3.1
Getvlc	7.3	9.1	9.1	7.6	--	8.6	23.0

### 6.3 Execution Time

Language Benchmark	Interpreter	C-Code	C-Code Optimized	Native Code	Native Code Optimized	C	C Optimized
recursive Functions	126.2	13.7	13.9	3.8	2.9	15.9	15.6
case Statements	318.5	11.0	7.9	7.6	5.7	9.8	3.0
if Statements	151.0	13.4	8.0	10.1	5.8	13.0	3.2
loop Statements	1843.0	54.2	50.6	36.1	28.4	61.4	0.5
1 dim array access	717.0	24.8	16.6	11.6	10.5	21.7	6.8
2 dim array access	1105.0	48.8	22.4	33.3	31.8	28.7	6.5
records	7.9	6.0	6.0	3.3	3.6	3.1	3.0
Scalar Access (4*250)*10**5	1555.0	188.8	67.6	92.6	76.7	61.1	0.0
Array Access (4*250)*10**5	1580.0	185.0	bug	105.1	90.6	67.8	72.0
Rightshift	1581.4	176.8	127.6	124.8	118.9	7.0	0.6
AsciiRead	--	--	184.0	--	171.7	--	32.8
AsciiWrite	--	--	180.7	--	82.6	--	41.0
BinaryRead	--	--	29.4	--	4.6	--	1.9
BinaryWrite	--	--	23.3	--	6.0	--	60.4
Sort	730.0	21.8	8.0	9.7	5.9	12.4	3.2
Idct	bug	278.0	107.6	169.5	157.7	24.8	7.8
Idctf	bug	354.2	71.7	91.8	59.9	51.2	31.5
Idctref	12250.7	1675.8	120.1	270.1	252.3	227.7	67.6
Getvlc	761.5	44.3	29.4	3619.8	bug	5.8	2.7