VHDL Integer Package – A Call for a New VHDL Companion Standard

W. Ecker
Siemens AG, Corporate Research & Development
81730 Munich, Otto-Hahn-Ring 6
e-mail: Wolfgang.Ecker@sfe.siemens.de

Abstract

This paper proposes a VHDL integer package as a candidate for a new VHDL companion standard.

Introduction

VHDL can be used as specification, description, and design language in most steps of the design process. The advantages in doing so are various: A single language approach reduces the effort for designer training as well as tool costs. Moreover, pieces of code or even complete packages containing type and subroutine declarations can be reused from earlier design steps in later design steps. The standard aspect of VHDL also allows for exchange of code between tools of different VHDL vendors.

Nevertheless, other description languages1 than VHDL are used in early design stages. Graphical entry tools, e.g. for StateChart-based approaches, are used for conceptual work. The programming language C, naming a Software oriented language, is applied for functional descriptions with high performance requirements.

Due to the fact, that the description capabilities of C and sequential VHDL are very similar, VHDL could be used instead of C for descriptions. A comparison of VHDL simulation and C execution time of real-word models showed however, that VHDL simulation is 5-10 times slower than the execution of equivalent C programs2. This lack of VHDL simulation performance is in most cases unacceptable.

A detailed analysis showed, that the following classes of C-operations were mainly responsible for a faster execution of the C models:

1. pointer arithmetic and pointer based access to arrays,
2. global variables and
3. shift and bit manipulation operations applied on integer objects.

The first point, the use of pointers, remains a pure C domain due to a completely different type concept in VHDL based on strong type checking. It was observed, that a lot of performance gain can

---

1Languages are meant in this context as formal description medium, which may have a graphical representation as well as a pure textual representation.

2The analysis considered a set of C-compilers as well as VHDL simulators using different optimization efforts.
be achieved by index-pointer transformations for array access. Most of the C compilers but none of the VHDL compilers provided this feature.

Global variables should be included in VHDL in the near future (see [3]). This may help to reduce the overhead resulting from passing all global objects as parameters to subroutines\(^3\).

Shift and bit manipulation operations have major impact on the performance on data flow and arithmetic dominated models, which appear for example in the domain of digital signal processing. A standardized VHDL integer package helps to close the performance gap in the domain of integer operations. The approach can be compared to the VHDL mathematical package (see [4]) which provides operations for objects of type \texttt{real}.

The rest of the paper is organized as follows: First, the need of a standardized integer package is motivated from the discussion of different implementation alternatives for shift and bit manipulation operations. Afterwards, the requirements, the specification and implementation alternatives of an integer package are presented. The paper concludes with a set of open questions and an overview of the current state of the work.

1 Motivation

To allow for a comparison of functional models written in C and VHDL, shift and bit manipulation operations, which are C build in, need be implemented in VHDL.

1.1 Shift and Bit Manipulation Operations

An obvious approach for the implementation is the use of vectors of bit for all integer values. Most VHDL simulators support arithmetic operations on vectors of bit by supplying packages. Shift and logical operations could be implemented easily, if they are not already part of the packages mentioned before. VHDL models using these implementations, however, are about 100 times slower than a comparable C program.

Thus, integer C objects must be implemented in VHDL as integer objects for performance reasons. A package containing shift and bit manipulation operations is required to support the built in C-operations \(|\), \(\&\), \(\_\), \(\_\_\) and \(<<\) in VHDL. The implementation, however, is more complex than expected. The right-shift operation, for example, can not be replaced by a simple division operation. Positive and negative values as well as values \(n\), with \(n = -(2^i), i = 1...31\) must be considered separately. A possible implementation shown below illustrates this fact.

\[
\text{function RightShift}(i,n : \text{Integer}) \text{ return Integer is}
\begin{align*}
\text{constant s : INTEGER := } 2^n; \\
\text{begin} \\
\quad \text{if } (i >= 0) \text{ then} \\
\quad \quad \text{return } (i / s); \\
\quad \text{elsif } (i \text{ mod } s = 0) \text{ then} \\
\quad \quad \text{return } (i / s); \\
\quad \text{else} \\
\quad \quad \text{return } ((i / s) - 1); \\
\quad \text{end if} ; \\
\text{end RightShift;} \\
\end{align*}
\]

Using integer objects and VHDL implementations for missing operations results in a 5-10 times lower performance of VHDL models compared with C models.

\(^3\)It must be pointed out for clarification, that subroutines in packages have been considered in this case only.
Shift and bit manipulation operations could also be implemented using the VHDL'93 foreign language interface. Most VHDL simulators support this feature already for their VHDL'87 version. Performance measurements of models using the foreign language interface showed no performance improvement compared to a pure VHDL solution. One reason is the overhead of importing foreign models. More important is, that the foreign implementation still requires a subroutine call in contrast to C, which allows to implement the operations as a single assembler operation.

This difference can be eliminated only, if the VHDL simulator performs these operations by a single assembler operation, too. A standardized VHDL integer package containing shift and bit manipulation operations is the first step along these lines.

1.2 +,−,* and / operations, without overflow error

In C, an overflow in the result of + and − operations only causes on most machines⁴ the cutting of the most significant bit. VHDL, in contrast, gives out an error message and terminates simulation. Adding arithmetic subroutines with the behavior described above would have two advantages:

1. The operation could be executed faster, and more important
2. the operation would have a behavior, which is more closely related to hardware⁵.

Thus, it would be useful to include C-like arithmetic operators, which do not implicitly cause an error message, in a VHDL arithmetic package.

1.3 Bit-Level Interpretation of Integer Objects

Another problem arose with the implementation of bit manipulation operations: The missing of the value $-2^{31} = 0\times80000000$. This value needs not be supported by VHDL simulators⁶ due to the fact, that the LRM requires a minimum range for integer objects of $-2^{31} + 1$ to $2^{31} - 1$ only⁷.

Additionally, a bit-level interpretation of integer values is required to allow for bit manipulation operations on integer values.

2 Requirements

Summarizing the points in the motivation, a VHDL integer package should support:

1. Bit-level interpretation of integer,
2. full 32 bit support,
3. logical operators on integer objects,
4. shift left and shift right operators on integer objects and
5. arithmetic operators without overflow error.

⁴Note: Neither the "C." nor the "C++"-standard describes how overflow must be handled. A VHDL standard can require such a behavior.

⁵Imagine an up-counter, which would not start with the value 0, when an overflow occurs but which would instead set a piece of hardware in an error state!

⁶Most, but not all, investigated simulators supported the value $-2^{31}$ for integer types.

⁷It should be mentioned, that the lack of the value caused a lot of problems, when implementing a performant model of a 32-bit CPU. Moreover, the missing of this value caused several problems by the implementation of the IEEE1076.3 synthesis packages.
3 Specification

A proposal for the VHDL integer package, called Int32Pack, is listed in the appendix. It contains the declarations of the constant Bit32, which specifies the number of bits of an integer value, the declaration of the type Int32, for the integer type with full 32-bit support and the declaration of the subtype Int5, which is required for the specification of subroutine parameters. Following subroutines or operators, respectively, are declared in the package, too:

1. The functions al and sr for C-like shift operations,
2. all VHDL logical operators not, and, nand, or, nor, xor and nxor\(^8\),
3. the functions add, sub, mul and div for C-like arithmetic operations, and finally
4. all VHDL shift and rotate operators\(^9\): all, srl, sla, sra, ror and rol.

The VHDL shift and rotate operations are included in the package also due to the fact, that they have a slightly different semantic as C-like shift operations. The package additional contains comments enumerating the implicitly defined VHDL operators for the type Int32.

4 Implementation

Three different implementation alternatives for the package body are discussed in this section: A pure VHDL based implementation, an implementation based on the foreign language interface and a fully integrated simulator solution.

4.1 VHDL-Implementation

The VHDL implementation is used to declare the behavior of the subroutines exactly. Performance aspects play a subordinate role. Parts\(^10\) of the implementation are shown in the appendix also.

The major idea of the implementation is shown in the subsequent figure.

---

\(^8\)This operation is currently declared as function according to VHDL'87. A VHDL'93 version of the package must declare this as operation.

\(^9\)This operations are valid for VHDL'93 only. The current implementation is based on subroutine calls, which correspond to VHDL'87.

\(^10\)Please note, that a full listing of the implementation would result in additional 10 to 15 pages.
All integer-parameters of an operation or a subroutine are converted in a corresponding 32-bit representation. The functions required for conversion also specify the bit-level interpretation of integer values. The converted values are passed as parameters to 32-bit implementations of the operations and subroutines. Finally, the 32-bit result is converted back to integer and set as return value of the integer based subroutines. Some local subroutines, like one bit-shift operations, are also part of the package.

4.2 Foreign-Language-Interface Implementation

A more performant implementation of the subroutines can be achieved by using the foreign language interface. Moreover, this implementation can be improved, if the tool-vendor make use of special undocumented features of the interface. The performance of this kind of implementation is still limited by the subroutine call, which must be performed as mentioned above.

4.3 Simulator Built-In Implementation

The highest possible speed can only be achieved by full integration of the operations in both, VHDL-analyzer and VHDL simulator. The analyzer must detect on the one hand, that the subroutines can be directly mapped on a single C/assembler operation and the simulator, on the other hand, must execute this operation directly.

A standardized package could help to focus the work and provide the users another possibility to improve their portability.

5 Open Questions

Several open questions remain. In the following subsection general questions and questions concerning additional functionality are enumerated.

5.1 General

The coding style for the VHDL code in the appendix writes reserved words in capital letters, all functions and operator names in small letters, all single character identifiers in small letters and all other identifiers with a first capital letter followed by any letters or digits. Moreover, the VHDL code was processed by a tool, which formats comments emphasis, reserved words bold and all other code in roman. Coding style as well as naming of the types, parameters and subroutines is open for discussion.

Anticipating the following section, an other general point, is the use of one or more packages. Reasons for having more packages are a separate 64-bit implementation and both, VHDL’87 as well as VHDL’93 compatibility. Overflow handling of adding operations must be discussed in detail, too.

A final general point is the relation of the integer packages to the synthesis packages currently under development. One possibility is the use of the synthesis packages for the definition of the semantic of the subroutines and operators. This would make the implementation easier but creates a strong dependency between both standards. Another possibility is the use of the synthesis packages for the validation of the implementations.
5.2 Additional Features

Several additional features seem to be useful to be integrated in the integer package. The most important point is, forced by actual 64-bit CPUs, a second full 64-bit integer type.

Another point of interest are additional bit manipulation functions, which allow to set, preset, clear or test one or a sequence of bits of an integer value. Their subroutine declaration could be:

```plaintext
function preset( a: Int32; n: Nat5 ) return Int32;
function clear ( a: Int32; n: Nat5 ) return Int32;
function set ( a: Int32; n: Nat5; v : Boolean ) return Int32;
function test ( a: Int32; n: Nat5; ) return Boolean;

function preset( a: Int32; l,r: Nat5 ) return Int32;
function clear ( a: Int32; l,r: Nat5 ) return Int32;
function set ( a: Int32; l,r: Nat5; v : Int32 ) return Int32;
function test ( a: Int32; l,r: Nat5; v : Int32 ) return Boolean;
```

The parameter n or the parameters l,r specify the bit-index or the left and right bit-index in the parameter a, which are modified.

Considering abstract implementations of busses, a wired-or or wired-and resolved integer type can help to model abstract busses in a performant way.

Conclusion and Outlook

This paper presents a proposal for an integer package, which is a good candidate for a VHDL companion standard. The author plans to contact all VHDL tool-vendors as soon as feedback from the reviewers arrived.

Concurrently with the submission of the paper, the author plans to contact IEEE concerning standardization.

Depending on the outcome of the standardization process within IEEE the author also plans to submit the package for standardization within IEC.

References

package Int32Pack is

  constant Bits32 : Natural := 32;

  type Int32 is range - ( 2**((Bits32-1)) ) to 2**((Bits32-1)) - 1;
  subtype Nat5 is Int32 range 0 to 31;

  function sl ( a : Int32; n : Nat5 ) return Int32; -- C-operator "<<"
  function sr ( a : Int32; n : Nat5 ) return Int32; -- C-operator ">>"

  function "not" ( a : Int32 ) return Int32; -- C-operator "~" 
  function "and" ( a, b : Int32 ) return Int32; -- C-operator "&" 
  function "or" ( a, b : Int32 ) return Int32; -- C-operator "|" 
  function "xor" ( a, b : Int32 ) return Int32; -- C-operator "^"

  function add ( a, b : Int32 ) return Int32; -- C-operator "+"
  function sub ( a, b : Int32 ) return Int32; -- C-operator "-"
  function mul ( a, b : Int32 ) return Int32; -- C-operator "*"
  function div ( a, b : Int32 ) return Int32; -- C-operator "/"

  function sll ( a, n : Int32 ) return Int32;
  function srl ( a, n : Int32 ) return Int32;
  function sla ( a, n : Int32 ) return Int32;
  function sra ( a, n : Int32 ) return Int32;
  function rrl ( a, n : Int32 ) return Int32;
  function ror ( a, n : Int32 ) return Int32;

  -- VHDL'93 shift and rotate operators, need to be defined for VHDL'87
  -- as functions

2.11
Local Type and Constant Declarations

-- Definition of an equivalent Bit_vector type, for the specification
-- of the semantic of the operations and types of the package.

constant Bits32Right : NATURAL := 0;
constant Bits32Left : NATURAL := Bits32 - 1;
subtype Signed32 is Bit_vector(Bits32Left downto Bits32Right);
constant Signed32Max : Signed32 := ( Bits32Left => '0', others => '1' );
constant Signed32Min : Signed32 := ( Bits32Left => '1', others => '0' );
constant Signed32Null : Signed32 := ( others => '0' );

Conversion Functions

function Convert ( p : in Signed32 ) return Int32 is
variable l : Signed32;
variable Carry : Bit;
variable Sum : Bit;
variable Sign : Int32 range -1 to 1;
variable r : Int32;
begins
l := p;
if ( l(Bits32Left) = '1' ) then
   -- check sign
   Sign := -1;
   l(Bits32Left) := '0';
   Carry := '1';
   for i in Bits32Right to Bits32Left - 1 loop
      l(i) := not l(i);
      Sum := l(i) xor Carry;
      Carry := l(i) and Carry;
      l(i) := Sum;
   end loop;
end if;
return r;
end Convert;
else
  Sign := 1;
end if;

r := 0;
for i in 30 downto 0 loop
  r := r * 2 + Bit'pos( l(i) );
end loop;

if ( r = 0 and Sign = -1 ) then
  r := Int32'left;
else
  r := r * Sign;
end if;

return r :
end Convert :

function Convert ( p : in Int32 ) return Signed32 is
  variable r : Signed32 :
  variable Sign : Bit :
  variable l : Int32 :
  variable Carry : Bit :
  variable Sum : Bit :

begin
  if p = Int32'low then
    return Signed32Min:
  end if :

if p < 0
  Sign := '1';
else
  Sign := '0';
end if :

for i in Bits32Right to Bits32Left - 1 loop
  r(i) := Bit'VAL( l mod 2 );
l := l / 2 :
end loop :

if Sign = '1' then
  Carry := '1';
  for i in Bits32Right to Bits32Left loop -- calculate 2th complement
    r(i) := not r(i) :
    Sum := r(i) xor Carry :
    Carry := r(i) and Carry :
    r(i) := Sum :
  end loop :
end if :

return r :
end Convert :

2.13
Support Functions for Shift Right and Shift Left

```
167  -- support functions
169  function BasicS1 ( a : Signed32 ) return Signed32 is
170      variable r : Signed32;
171  begin
172      for i in Bits32Left downto Bits32Right + 1 loop
173         r ( i ) := a ( i - 1 );
174      end loop;
175      r ( Bits32Right ) := '0';
176  return r;
177  end BasicS1;
178
179  function BasicSr ( a : Signed32 ) return Signed32 is
180      variable r : Signed32;
181  begin
182      for i in Bits32Right to Bits32Left - 1 loop
183         r ( i ) := a ( i + 1 );
184      end loop;
185      r ( Bits32Left ) := a ( Bits32Left );
186  return r;
187  end BasicSr;
```

Shift Right and Shift Left Function

```
545  function sl ( a : Int32; n : Nat5 ) return Int32 is
546      variable r : SIGNED32;
547      variable p : SIGNED32;
548  begin
549      p := Convert ( a );
550      r := sl ( p, n );
551      return ( Convert ( r ) );
552  end sl;
553
554  function sr ( a : Int32; n : Nat5 ) return Int32 is
555      variable r : SIGNED32;
556      variable p : SIGNED32;
557  begin
558      p := Convert ( a );
559      r := sr ( p, n );
560      return ( Convert ( r ) );
561  end sr;
```