european space agency
european space research
and technology centre

ASIC/001
Issue 1
September 1994

VHDL Modelling Guidelines

Approved by

R. Creasey
Onboard Data Division

R. Coirault
Radio Frequency Systems Division

Prepared by P. Sinander

Onboard Data Division (WD)
Keplerlaan 1 - Noordwijk - The Netherlands
Mail address: Postbus 299 - 2200 AG Noordwijk - The Netherlands
Tel: +31-1719-83667 - Telex: 39098 - Cables: Spaceurop, Noordwijk - Fax: +31-1719-84295

Table of contents

1 INTRODUCTION

1.1 Purpose and scope

This document defines requirements on VHDL models and testbenches, and is intended to be used as an applicable document for ESA developments involving VHDL modelling. It is mainly focused on digital models; specific requirements for analog modelling have not been covered.

The requirements concern simulation and documentation aspects of VHDL models delivered to ESA; specific rules and guidelines for logic synthesis from VHDL have not been included. Nevertheless, the requirements of this document are compatible with the use of logic synthesis. The requirements are not applicable for the case when a design database is transferred in VHDL format.

The purpose of these requirements is to ensure a high quality of the developed VHDL models, so they can be efficiently used and maintained with a low effort throughout the full life-cycle of the modelled hardware.

The requirements are based on the VHDL-93 standard, to minimise future maintenance efforts for updating models. However, in an initial stage the models shall be backward compatible with VHDL-87 as far as possible, since some tools will not be updated immediately.

The requirements have been structured in a general part applicable to all VHDL models, and additional requirements applicable to different kinds of models. In addition, VHDL code examples and a list of common problems encountered have been included in order to provide some guidance to the VHDL developer. If not stated which kind of model is to be developed, the default kind is a model for Component simulation.

Requirements expressed in a Statement of Work or similar document have precedence over this document.

1.2 Applicable Documents

AD1 IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993
AD2 IEEE Standard Multivalue Logic System for VHDL Model Interoperability (std_logic_1164), IEEE Std 1164-1993

1.3 Reference Documents

RD1 IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987
RD2 IEEE Standards Interpretations: IEEE Standard VHDL Language Reference Manual, IEEE Std 1076/INT-1991

2 REQUIREMENTS FOR ALL KINDS OF MODELS

2.1 General

The models shall be written in VHDL-93 as defined in AD1. All code shall be written with the intent to be simulator independent (as far as possible, using all available information); the use of non-standard constructs or supersets is not allowed. Note that the code is not necessarily correct just because it compiles and executes on one simulator without errors; many tools do not detect all possible errors (see further appendix B). In case of ambiguities the interpretations in RD2 shall have precedence.

Unusual language constructs should be avoided, since this will reduce the clarity and have a potential to stimulate bugs in other VHDL tools.

All models shall be compliant with VHDL-93 as defined in AD1. To allow backward compatibility with VHDL-87 in an initial stage, the VHDL code shall as far as possible also be compliant with RD1. The usage of the new features of VHDL-93 shall be agreed with ESA before being introduced.

All documentation, identifiers, comments, messages, file names etc. shall use or be based on the English language.

The code shall be consistent in writing style and naming conventions. The VHDL reserved words shall appear in uniform casing; they shall either all appear in lower-case, or all appear in upper-case. It is recommended to write identifiers using mixed casing. Consistent casing shall be used in all the code.

The code shall be concise and use the most straightforward and intuitive constructs. Using more code than necessary leads to poorer readability and lower simulation speed. Wherever possible, unused parts of the code shall be removed. Temporary assignments shall not be used unless necessary.

The code shall emphasize good readability. It shall contain maximum one statement per line, and have maximum 80 characters per line. The code shall be properly indented, for example using 3 *space* characters; the indentation shall be the same in all the code. The *TAB* character shall not be used, being environment dependent. Related constructs should be grouped together, and these groups should be separated e.g. using blank lines or lines made of dashes where this increases the readability. Identifiers, comments etc. should be aligned vertically where this improves the readability.

Automatically generated VHDL models, for example from schematics or from State Machine diagrams, may be accepted subject to explicit ESA approval. To obtain such approval, the contractor shall provide complete written information about any possible non-compliancies w.r.t. the requirements before commencing the development, after which ESA may grant the usage. If granted, all additional design documentation (e.g. the schematics or the State Machine diagrams) should be delivered in addition to the VHDL code itself. Automatically translated models, e.g. from Verilog, are generally not acceptable, not fulfilling the requirements of this document.

2.2 Names

Meaningful non-cryptic identifier names shall be used, based on the English language. The same identifier name as for the actual hardware and as in the data sheet or similar documentation shall be used. For signals and variables that are active low, this shall be clearly indicated by their name, for example by suffixing *_N* as in *Reset_N*. In case a name would not be legal VHDL, it should be close to the original name and a comment should be included for clarification. The VHDL-93 extended identifiers (any string enclosed by two \ characters) may only be used in case ESA approval has been obtained before commencing the development.

A name should indicate the purpose of the object and not its type. Example: an eight-bit loadable counter used for addressing should be called *AddressCounter* (its purpose) rather than *CountLoad8* (its type).

The naming convention (e.g. how active low and internal signals are indicated, if registers are indicated with a special suffix etc.) used should be documented in each file, close to the signal declarations or in the file header.

It is recommended to write identifiers using mixed casing, with consistent casing in all the code. It is recommended to use less than 15 characters in the normal case, though the number of characters used for an identifier shall never exceed 28 due to an NFS limitation for file names.

The VHDL name of the predefined identifiers, including the identifiers in the *Std* and *IEEE* design libraries shall never be used for other identifiers. Note for example the formfeed character *FF* and the *Time* unit *Min*.

2.3 Comments

The purpose of comments is to allow the function of a model, package or testbench to be understood by a designer not involved in the development of the VHDL code.

All models shall be fully documented with explanatory comments in English. The comments shall be placed close to the part of the code they describe; a description only in the file header without comments in the executable part is not acceptable. All comments shall be indented and aligned for good readability. The comments shall be descriptive and not just direct translations or repetitions of the VHDL code.

Each file shall include a header, as a minimum containing the following information:
- Name of the design unit(s) in the file;
- File name;
- Purpose of the code, description of hardware modelled;
- Limitations to the model and known errors, if any, including any assumptions made;
- Design library where the code is intended to be compiled in;
- List of all analysis dependencies, if any;

- Author(s) including full address;
- Simulator(s), simulator version(s) and platform(s) used;
- Change list, containing version numbers, author(s), the dates and a description of all changes performed (as a minimum this list shall be updated for each delivery, in case a change has taken place).

Each subprogram declaration, subprogram, process, block etc. shall be immediately preceded by a description of its function, including any limitations and assumptions. For subprograms, the parameters and result shall also be described.

For port and generic clauses in entity and component declarations, there shall be one signal declaration per line, directly followed by a comment describing the signal. Describing the signals in a group of comments separate from the declarations themselves are not recommended, being likely to become inconsistent in case of modification.

Where functionality is represented by data, as for example microcode or a PLA fuse-map program, the functionality shall be fully described. This applies regardless of the data representation (e.g. hard-coded constants or read from an ASCII file).

2.4 Types

The leftmost bit of an array shall be the most significant, regardless of the bit ordering. <u>Example:</u> In *Bit_Vector(0* ***to*** *15)*, bit 0 is the Most Significant Bit (MSB), whereas in *Bit_Vector(15* ***downto*** *0)*, bit 0 is the Lest Significant Bit (LSB).

It is recommended to write the code so it is possible to change the type of a signal or variable without changing the simulation behaviour. This implies:
- Avoid relying on default initialisation of a variable or a signal unless a reset policy ensures that the model is initialised in an explicit way (typical for synthesizable constructs);
- Avoid relying on the number of type values in a type declaration;
- Avoid dependencies on the order in the type declaration.

Real literals shall only be written in decimal format. Based literals shall only be specified in base 2, 8, 10 or 16, and should not have an exponent. The use of underscore characters in literals should be restricted to binary, octal and hexadecimal literals. Hexadecimal literals shall be written using uppercase characters, for example *16#9ABC#*.

2.5 Files

For portability reasons the only allowed file type is *Std.Textio.Text*. However, it should be noted that there are still certain variances, such as (see further AD1 section 14.3):
- Line delimiters might not be readable, and therefore characters with a lower rank than the *space* character should be avoided;
- *Underline* character(s) and/or an exponent may be absent or present when writing values of the *Integer*, *Real* and *Time* types;
- The casing of the identifier when writing values of the *Boolean* type may vary.

Consequently, in case values of the *Boolean*, *Integer*, *Real* or *Time* types are written using *Std.TextIO*, possible impact on the portability should be analyzed. The same applies when characters with lower rank than the *space* character is read from a file.

The predefined file *Std.TextIO.Input* should be avoided, since its implementation on different simulators varies. In particular, it shall never be used in testbenches for automated verification, since this could preclude the verification to be performed using a script. Also note that assertions may be output to the *Std.TextIo.Output* file by some simulators, but not by all.

When data is to be read from a text file, e.g. for initialising a memory, the format of the file shall be fully and clearly specified in the VHDL code implementing the reading function. An example should also be included.

It is recommended to limit the number of characters per line in a file to be read to 80 characters. In any case, it shall never exceed 255 characters.

2.6 Signals and ports

The same name shall be used for a signal throughout all levels of the model, wherever possible. In cases where exactly the same name cannot be used, e.g. when two identical sub-components have been instantiated, names derived from the same base name should be used.

The index ordering (i.e. using **to** or **downto**) of the model top-level entity port clause signals shall be identical to the one used in the data sheet or similar documentation. It is recommended to use the same index ordering in the whole model, but in case the index order is reversed within the model, this shall be clearly marked every time the index order is different w.r.t. the corresponding signal at the highest level of the hierarchy.

The **buffer** mode shall never appear in the port clause of the model's top-level entity declaration.

The port clause signal declarations shall appear in a logical order. It is recommended to order the signals in the port clause after their mode; first input signals, followed by bi-directional signals and last output signals. Nevertheless the signals could be grouped together according to their function, and within each such group according to their mode. Port clauses shall be commented as specified in section 2.3.

Port maps for component instantiations shall use named association, unless all signals in the component instantiation have the same (or derived) name as in the component declaration. The same applies to generic maps where increasing the readability.

Duplicating a signal by assignment to another signal only to rename the signal, to allow another port mode to be used or to perform a type conversion should only be used where necessary or where clearly increasing the readability.

2.7 Assertions

Assertions shall be used to report model errors, timing violations and when signals have illegal or unknown values affecting the model behaviour. The following policy for assigning severity levels is recommended:

- *Failure*: Errors in the model itself (e.g. if a statement believed to be non-executable is actually executed);
- *Error*: Timing violations and invalid data affecting the state of the model, including illegal combinations of mode signals and of control signals (e.g. unknown data on a mode input or too short reset time);
- *Warning*: Timing violations and invalid data <u>not</u> affecting the state, but which could affect the simulation behavior of the model (e.g. if data to be sent out from an interface is invalid);
- *Note*: Essential information that is not classified in the other severity levels, such as reporting from which text file data is read, which testbench is executed, if an event is detected on an input signal whose function has not been implemented (e.g. activation of production test) etc.

A model should not issue assertions for insignificant events, for example at start, during reset or if an event has no impact on the simulation behavior. Neither should unnecessary messages be generated, e.g. as reporting whether Worst Case or Best Case timing has been selected.

The assertion report shall give a clear description of the reason for the assertion, and shall include the hierarchical path to the instance or package, as well as identifying the signal(s) where applicable. It is sufficient to report the hierarchical path relative to the top-level entity of the model before VHDL-93 has been fully introduced (then the new predefined attribute *'Instance_Name* should be used).

Testbenches could use *Std.TextIO.Output* instead of assertions where advantageous.

2.8 Subprograms, processes, entities, architectures, component declarations

All processes shall be associated with a descriptive label. The same applies for other concurrent statements where this will increase the readability.

A process with only one **wait** statement (e.g. typical for synthesizable processes) should use a process sensitivity list instead of the **wait** statement, since this increases the readability.

Wherever possible, all language constructs such as subprograms, package declarations and bodies, entities, architectures and **loop** statements shall be qualified, i.e. the identifier associated with the construct shall also appear at its end.

Procedures that modify signals or variables not passed as parameters in the procedure call should be avoided. Nevertheless, in some cases such as testbenches, this technique could actually increase the readability of the code. If used, it shall be clearly commented which signals and variables can be modified by the procedure call.

The top-level entity should have the same name as the device or hardware modelled. Declarations other than generic and port clauses should be avoided in an entity declaration.

The identifier, port clause and generic clause of a component declaration shall be identical (i.e. use the same identifiers and the same ordering) to the declarations in the corresponding entity declaration.

2.9 Configurations

There shall be no configuration specifications within the architectures of the model itself, since it would then not be possible to use another configuration without modifying the source code.

A testbench should preferably use explicit configuration specifications within its architecture.

2.10 Packages

Where possible, packages approved by the IEEE should be used rather than redeveloping similar functionality, in order to reduce development cost as well as the number of errors in the packages and to allow speed optimised versions to be provided with the VHDL simulators. At the time of writing, only the *IEEE.Std_Logic_1164* package has been approved; in case a package to be approved by the IEEE is used before approval it shall be placed in the same design library as the model itself.

Packages specific to a particular CAD tool should only be used after ESA approval before commencing the development. In particular, any source code distribution restrictions should be assessed.

The number of packages used by a model shall not be excessive. There shall be no empty or almost empty packages, unless where this clearly increases code readability. It is recommended to place VHDL code concerning different functionality areas in different packages, e.g. all timing parameters in one package, all subprograms related to timing in another etc. However, there should not be a separate package for each entity where constants etc. used by that entity are defined.

The declarations in a package body shall appear in the same order as the corresponding declarations in its package declaration.

The package declaration shall contain full documentation about the declared types, constants, subprograms etc.

Each package containing one or more subprograms - except packages approved by the IEEE - shall be separately and extensively verified as specified in section 2.13, using a testbench allowing automated verification as described in section 3.4.1.

2.11 Design libraries

The model design units shall be placed in a design library other than *Work*. This will normally be a separate design library for each model, though families of devices, such as 54-series logic or a collection of different memories, are preferably grouped together in one design library.

This design library shall be named after the device, respectively the family, with the suffix *Lib* appended. The top-level entity to be used for simulation shall have the same name as the device. Example: a device *XYZ* should be placed in the library *XYZ_Lib*, and should be used as *XYZ_Lib.XYZ*. It is recommended to consult ESA regarding the name choice, to avoid the same name being used in different developments.

This design library shall contain all design units used by the model itself (including packages), except for the packages in design library *Std*, the packages in the *IEEE* design library (at the time of writing only *Std_Logic_1164*), and common packages used by many different models. Whether a particular package is to be considered as common is subject to ESA approval before commencing the development.

The testbench(es) used for the development and verification of the device shall be placed in a design library different from the device design library, such as *XYZ_TB_Lib* or *Work*. This design library should contain all hierarchical sub-components and packages used, except the model to be tested (already being in a separate library) and standard and common packages in the same way as above.

The *IEEE* design library shall not contain any other packages than those approved by the IEEE. Neither shall these packages be modified or extended. At the time of writing only the *IEEE.Std_Logic_1164* package has been approved. Some CAD companies may place own defined packages in the *IEEE* design library. In case such a package is used, it shall be moved to the design library where it is used.

2.12 Constructs to be avoided

The VHDL code shall be fully deterministic when executed regardless of the simulator used. This means for example:

- There shall be no communication between different parts of the model through files;
- Resolution functions shall always be commutative and associative;
- Shared variables (VHDL-93) shall only be used after justification and ESA approval before commencing the development. It shall then be proven by analysis that the usage is fully deterministic, which shall be documented;
- Care should be taken when using floating point values, especially conversion to and from floating point values, comparisons between floating point values and events on floating point values. Note that using pseudo-random test patterns is not portable if the pseudo-random generator is using the *Real* type;
- The *Std.TextIO* portability limitations shall be avoided, see further section 2.5.

Refer to appendix C of AD1 for more information.

CAD tool specific types shall not be used. Features specific to an operating system, e.g. the */dev/null* file on Unix systems, should be avoided. Neither shall absolute paths be used for filenames.

Objects with an implicitly declared index, for example a line returned from the *Std.TextIO.Read* procedure for a string, shall never be used with absolute indexing. Instead the predefined attributes for indexing, such as *'Left*, shall be used. As a consequence absolute indexing shall be used when declaring an object to be referenced using an absolute index.

The dependence on implementation defined limitations, for example 32-bit limitations on the *Integer* and *Time* types, shall be minimized. In particular, the model should not encounter implementation defined limitations on *Time* as long as the simulated time does not exceed the limitation.

Subprograms and components should not be renamed by encapsulating them with subprograms or components with other names unless where clearly increasing the readability.

Signals, variables, constants, subprograms or components shall not be hidden by declaring another object with the same name (overloading is not considered as hiding, and is encouraged where beneficial).

The predefined operators, subprograms, attributes etc. shall never be redefined. This shall also apply to the packages in the *IEEE* design library. Neither shall similar declarations using the same names be created.

Since the model shall be placed in another design library than *Work*, there shall be no references to *Work* within the code for the model and its packages.

The constructs below are considered as obsolescent. Being not strictly necessary to use for modelling, they should therefore not be used, unless otherwise agreed with ESA before commencing the development:
- Guarded expressions, signals and assignments, including the reserved words **bus**, **disconnect**, **guarded**, **register**;
- The **linkage** mode for interface declarations;
- The *Allowed Replacement Characters* defined in section 13.10 of AD1;
- The *Std.TextIO.EndLine* function, *L'Length = 0* could be used instead (*EndLine* was excluded from VHDL-87 being illegal VHDL);
- File types other than *Std.TextIO.Text*.

2.13 Verification

The purpose of the verification shall be to verify that the developed model is correct, with few or no errors being found. It shall not be a means to locate errors in the VHDL code in order to patch them.

The verification shall be performed by somebody not involved in the creation of that model or package, to avoid that a misunderstanding of the functionality is masked by the same misunderstanding in the verification.

In case another simulation model is available, the VHDL model shall also be verified versus this other model (regardless whether the other model is a VHDL model).

The verification shall solely be performed using VHDL testbenches as specified in section 3.4, no simulator specific features or commands shall be used.

The verification shall encompass the full functionality, including all assertions and error messages. As a minimum requirement every executable line of the model shall be executed, which shall be proven and documented. The following guidelines shall apply:

- Only sequential and concurrent statements, excluding component instantiations and block statements, shall be counted as executable (empty lines, comments, declarations, specifications etc. shall not be counted);
- Statements that cannot be removed but can be shown to be non-executable should be excluded. An example is the **others** choice in a State Machine decoding <u>only</u> covering non-existing states. Wherever possible such statements shall be associated with an assertion of severity level *Failure* reporting model failure;
- Only statements executed by a testbench verifying the complete model or a package may be counted as executed; the coverage obtained when verifying a sub-component of the model shall be disregarded;
- Only statements executed for the purpose of verifying the model versus the functional requirements may be counted as executed. Statements included to implement testability can nevertheless be counted, in case it can be shown that they are actually used. <u>Example:</u> If a Built-In Self Test function happens to execute certain statements in the model they should not be counted as executed, except for those included only for the purpose of implementing this Built-In Self Test.

In addition, subprograms placed in packages shall be verified for all possible boundary conditions and singularities. This shall include unknown and not initialized values, as well as ascending, descending and invalid ranges, and null arrays. Each such package (except IEEE approved packages) shall be fully verified by its corresponding separate testbench.

The results shall be presented in a verification compliance matrix for each VHDL model and package, clearly describing each test and its extent, when, how and by whom it was performed and the result. Each separate test shall be presented together with the date of verification and a signature. In addition, the results shall be summarised for each model, clearly identifying any discrepancies from the specifications, including agreed differences.

2.14 Format of deliverable items

All models and packages shall be delivered with their respective testbench(es) in electronic format, using the two organisations specified below (both shall be delivered):

- Using separate files for each design unit or design unit pair (entity with corresponding architecture, or package declaration with corresponding package body). All design units shall be delivered, except from design library *Std*. Note that this includes the *IEEE.Std_Logic_1164* package and other common packages, if any;

- Using one file for the model design library, and one file for the testbench design library. Each file shall contain all design units of the respective design library, as specified in section 2.11, in the following order:
  - Top-level entity;
  - All packages;
  - Remaining entities and architectures combined in pairs;
  - The top-level architecture.

  The headers of each design unit (pair) shall be included in the file. For a model, the header of the file could be derived from the header of the top-level entity.

The files shall contain the design units in a correct compilation order. Each file shall have the same name as the contained entity, package, configuration respectively design library. If a file contains a separate architecture or package body, this should be indicated in the filename by appending the architecture name respectively the word *body*. VHDL files shall have a *.vhd* or a *.vhdl* suffix. The uniqueness of a filename shall not depend on case sensitivity. <u>Examples:</u> An entity *XYZ* together with its architecture *Behavioral* are together placed in the file *xyz.vhd*, or in the separate files *xyz.vhd* and *xyz.behavioral.vhd*. The design library *XYZ_Lib* is placed in the file *xyz_lib.vhd*.

There shall be a script file for each design library which when executed compiles all the separate files (design units or design unit pairs) of the design library. For models where automated verification is required, a script file performing the full verification shall also be delivered. The scripts shall be executable under a standard *Unix sh* or *csh* shell.

Any files associated with the code shall be delivered, such as files read by *Std.TextIO*.

In case automated verification is to be performed by writing an ASCII file to be compared with a reference file, each such reference file shall be delivered. Each reference file shall have the same name as the file written by the testbench, with the recommended suffix *.chk* appended.

All files shall use the ASCII character representation (*Unix* ASCII).

Unless otherwise specified, the default delivery media for the files shall be QIC-150 tape cartridges suitable for archive storage (high quality, not reused etc.), and it shall be possible to restore the files on a Sun Sparc workstation using the *tar* command. The contents (VHDL model, developer, issue, date) and the procedure to retrieve all information shall be clearly indicated on the tape cartridge itself.

3 ADDITIONAL REQUIREMENTS

3.1 Models for Component simulation

The main purpose of a model for component simulation is to be used for verification of a component under development, before proceeding with the manufacture. This implies that the model should exactly reflect the structure and functions of the underlying hardware; accuracy being more important than simulation speed. The model shall have correct timing characteristics, at least using estimated (e.g. pre-layout) values for timing parameters.

The model need not be synthesizable, unless so specified. The model can be on the gate level or on the Register Transfer level. Phenomena such as EMC, transmission line effects etc. need not be modelled.

However, for some developments it is also specified that a model for board-level simulation shall be developed. The same entity declarations shall then be used for both models (i.e. the model for Component simulation will be represented by one architecture, and the model for Board-level simulation by another architecture).

An accurate block diagram showing the relationship between different VHDL modules, their input and output signals etc. shall be created. It is suggested not to mix structural and behavioral descriptions within the same architecture.

3.1.1 Names

The model structure and naming convention shall be the same as for all other design descriptions, including the Architectural Design Document, the Detailed Design Document and the data sheet. It is recommended to use an architecture name reflecting the level of the description, such as *GateLevel* or *RTL* for the architecture associated with the top-level entity.

3.1.2 Types

The VHDL predefined types such as *Bit*, *Bit_Vector*, *Boolean* and *Integer*, together with the types defined in the *IEEE.Std_Logic_1164* package are preferred. For Finite State Machines, the states could be represented by constants of type *Bit_Vector* or *Std_ULogic_Vector*, or by enumerated types. Complex data types should be avoided unless were beneficial.

3.1.3 Model interface

The preferred types for the model interface are *Std_Logic* and *Std_Logic_Vector* from the *IEEE.Std_Logic_1164* package for digital signals. The *Bit* and *Bit_Vector* types may also be used, but no other types are allowed. In the case of analog signals, the *Real* type is suggested.

Global signals shall not be used; all signals of the component shall be specified in the top-level entity port clause, also including signals whose functions have not been modelled, such as signals activating specific test modes etc. Power pins and unconnected pins need not be included. The model interface should only include signals actually present on the component.

It is recommended that the top-level entity declaration is not preceded by any other library and use clauses than necessary for defining the interface signals (*IEEE.Std_Logic_1164*) and the timing (e.g. the *Vital_Timing* package). No user-defined subtypes should be used in the port clause.

3.2 Models for Board-level simulation

The main purpose of a model for Board-level simulation is to be used for the verification of a board using the component, normally together with several other components. This can be seen as the simulation version of breadboarding. This implies that the model must have acceptable simulation speed, but only need to model the functionality possibly affecting the board and the other models. The model should be on the Register Transfer level or higher, a gate level netlist is not acceptable. The model need necessarily not reflect the actual internal structure of the component.

The model behaviour shall include the full functionality, though specific test modes only used during manufacturing test need not be implemented (activation should be reported as specified in section 2.7). The interface signals shall have the correct digital waveform behavior as can be observed at the interfaces of the components. Timing shall be modelled for the interface, including checking violations on inputs and assigning output delays.

The model shall be coded for efficient simulation w.r.t. simulation time. This implies that the number of processes, signals and signal assignments shall be minimized, due to their negative impact on the simulation speed. There should not be more design entities than there are blocks in the architectural block diagram. Where possible variables should be used instead of signals. Resolved signals should be avoided where not functional. By using types on higher abstraction levels - e.g. *Integer* instead of *Bit_Vector* - models with higher simulation speed will be obtained in most cases. It should be avoided to execute statement when not necessary.

The memory usage shall be optimised when necessary, e.g. when modelling memory devices, since otherwise simulation could be impossible due to the memory requirements of the simulator. One technique could be to divide the memory area into a number of blocks, which would be allocated only when used.

It is suggested to model the timing and handling of unknowns in the top-level architecture.

The model should avoid reading files, since this complicates model distribution and usage.

Each model shall be delivered with a full configuration declaration for the top-level entity, explicitly binding all entities and architectures of the model.

An accurate block diagram showing the relationship between different VHDL modules, their input and output signals etc. shall be created.

A User's Manual shall be written, allowing a Board-level designer not involved in the development activity to efficiently use the developed VHDL models to perform Board-level simulation at a later stage, without needing the VHDL source code.

3.2.1 Names

The model naming convention shall be the same as for all other design descriptions, and especially the data sheet. Unless otherwise agreed with ESA, the architecture associated with the top-level entity should be named *BoardLevel*.

3.2.2 Model interface

The types used for the model interface shall be *Std_Logic* and *Std_Logic_Vector* from the *IEEE.Std_Logic_1164* package, no other types are allowed for digital signals. In the case of analog signals, the *Real* type is suggested.

Pull-up and pull-down on inputs and outputs shall be correctly modelled; the *IEEE.Std_Logic_1164* values 'L' and 'H' on an input shall result in the same simulation response as the values '0' and '1', respectively. The *IEEE.Std_Logic_1164* values 'L', 'H' and 'W' shall only appear on outputs having weak drivers for those states.

Global signals shall not be used; all signals of the component shall be specified in the top-level entity port clause, also including signals whose functions have not been modelled, such as signals activating specific test modes etc. Power pins and unconnected pins need not be included. The model interface shall only include signals actually present on the component.

The top-level entity declaration should not be preceded by any other library and use clauses than necessary for defining the interface signals and the timing. No user-defined subtypes shall be used in the port clause.

3.2.3 Handling of unknown values

Unless otherwise specified, handling of unknown values (X-handling) may be limited to only reporting the offending values using assertions. If propagation of unknown values is implemented, it should only apply to data <u>not</u> affecting the state of the model; there should be no propagation on control signals or mode signals affecting the model state. In all cases assertions shall be issued for unknown values that would affect the simulation behaviour (on all inputs); insignificant occurrences should not be reported.

The handling of unknown values should be documented in the header of the top-level entity as well as in the User's Manual.

The *IEEE.Std_Logic_1164* values 'U', 'Z', 'W' and '-' on an input shall result in the same simulation response as the value 'X', though propagation of the un-initialised value 'U' should be considered for combinational functionality and low complexity devices. Models that have not been initialized, as well as parts thereof, should produce the *IEEE.Std_Logic_1164* value 'U' when accessed. The '-' value shall never appear on an output.

3.2.4 Timing

All inputs shall be checked w.r.t. period, pulse width, setup time and hold time as applicable, and all significant violations reported using assertions. Violations that would not affect the simulation behaviour should not be reported. All outputs shall be assigned output delays, including tristate modelling. The timing shall be correctly modelled w.r.t. the internal or external signals generating the change of the signal.

All timing parameters shall have the simulation condition selectable between Worst Case, Typical Case or Best Case timing, controlled by a generic parameter *SimCondition* of type *SimConditionType* defined in the package *ESA.Simulation* (see appendix E), with the default simulation condition being Worst Case. The simulation conditions for CMOS processes are defined as follows:

- Worst Case: The timing at the lowest voltage (e.g. 4.5 Volt), highest temperature (e.g. 125 °C) and slowest process characteristics;
- Typical Case: The timing at the nominal voltage (e.g. 5.0 Volt), temperature (e.g. 25 °C) and process characteristics;
- Best Case: The timing at the highest voltage (e.g. 5.5 Volt), lowest temperature (e.g. -55 °C) and fastest process characteristics.

The values of the timing parameters shall be specified in a separate package as deferred constants, allowing the values to be changed by only recompiling the package body. This package shall be named after the component name with the suffix *Timing* appended, as in *XYZ_Timing*. The data sheet timing parameter names shall be clearly indicated for each timing parameter.

The timing parameters shall be updated with accurate values after final layout and manufacture. The values shall be taken from the component data sheet. If all values are not available, the designer or manufacturer should be contacted for advice. In case no information can be obtained, suitable values should be established in consultation with ESA. The timing parameters shall be specified including an appropriate loading, which should be specified in the timing package, in the header for the top-level entity and in the User's Manual.

As a baseline, timing parameters should be given in an integer number of *ns* in order to avoid simulation time limitations, with values rounded in a pessimistic way.

The model shall allow timing check disabling, controlled by a generic parameter *TimingChecksOn* of type *Boolean* declared in the top-level entity declaration. When *TimingChecksOn* has the value *False* no timing checks shall be performed. The default value shall be *False*. The implementation shall ensure minimum simulation time penalty when timing checks are disabled.

When the packages implementing the *Vital Model Development Specification* have been approved by the IEEE, it is recommended to use them for checking and reporting setup and hold times etc. In this case it is allowed that the severity level for timing violations are all *Error* (as implemented in the Vital subprograms). The types defined therein may also be used. In case a package is used before IEEE approval, it should be placed in the same library as the model itself.

Timing parameters should use names compliant with the *Vital Model Development Specification*, which could allow back-annotation on the board-level to be performed using the Standard Delay File (SDF) format in the future, or alternatively the same names as in the data sheet should be used. Vital-compliant naming for some types of timing parameters has been listed below:

- tpd_<OutPort> — Propagation delay applicable to all delay paths for the output <OutPort>;
- tpd_<InPort>_<OutPort> — Propagation delay only applicable to the specified Input-to-Output delay path;
- tsetup_<InPort> — Setup time for the input <InPort> w.r.t. any clock;
- tsetup_<InPort>_<ClkPort> — Setup time for the input <InPort> w.r.t. the clock signal <ClkPort>;
- thold_<InPort> — Hold time for the input <InPort> w.r.t. any clock;
- thold_<InPort>_<ClkPort> — Hold time for the input <InPort> w.r.t. the clock signal <ClkPort>;
- tperiod_min_<ClkPort> — Minimum allowable period time for <ClkPort>;
- tperiod_max_<ClkPort> — Maximum allowable period time for <ClkPort>;
- tpw_hi_min_<InPort> — Minimum pulse width for a high value at the input <InPort>;
- tpw_hi_max_<InPort> — Maximum pulse width for a high value at the input <InPort>;
- tpw_lo_min_<InPort> — Minimum pulse width for a low value at the input <InPort>;
- tpw_lo_max_<InPort> — Maximum pulse width for a low value at the input <InPort>.

It is recommended to only report timing violations, and not to generate unknown values. In case generation of unknown values is implemented, a generic parameter *XGenerationOn* of type *Boolean* should be declared in the generic clause of the top-level entity. When *XGenerationOn* has the value *False* timing violations should not lead to unknown values being generated. The default value should be *False*.

It is not required to check timing violations for changes between similar logic levels (e.g. ’0’ and ’L’, ’1’ and ’H’); to differentiate delays for falling and rising signals or to assign separate delay values for each element of a *Std_Logic_Vector*. Neither is it required to proportionally model loading, temperature, voltage or radiation impact on the timing parameters.

Optional: In case more detailed timing modelling is desired, such as differentiating delays for rising and falling edges, assigning separate delays for each element of a vector or providing wire-load delays for the inputs, it is recommended to be compliant with the requirements for a Vital level 0 model. The same applies in case it is desired that the timing parameters appear in the generic clause of the top-level entity to allow easy modification of the timing on a per-instance basis.

3.2.5 Verification

The verification shall be performed using a testbench allowing automated verification as described in section 3.4.1. The verification shall include assigning all nine values of the *Std_Logic* type to each input (including **inout** ports), and to produce timing violations on each input.

3.3 Models for System-level simulation

The main purpose of a model for system-level simulation is to provide the functionality of a board, a subsystem, an algorithm or a protocol, with a simulation speed allowing trade-offs to be performed. No similarity with any hardware is necessary, as long as the desired functionality is achieved. The behaviour may be approximated w.r.t. details such as timing aspects, exactly which clock cycle an event occurs, the exact numerical value of a result etc.

The model shall be coded for efficient simulation, not to slow down simulations. This implies that the number of entities, processes, signals and signal assignments shall be minimized, due to their negative impact on the simulation speed. Where possible, variables should be used instead of signals. Resolved signals should only be used when advantageous. By using types on higher abstraction levels - e.g. *Integer* instead of *Bit_Vector* - models with higher simulation speed will be obtained in most cases. It should be avoided to execute statement when not necessary.

The memory usage shall be optimised when necessary, e.g. when modelling memory devices, since otherwise simulation could be impossible due to the memory requirements of the simulator. One technique could be to divide the memory area into a number of blocks, which would be allocated only when used.

3.3.1 Model interface

The model interface should use the types most suitable for the intended usage of the model, be that *IEEE.Std_Logic_1164* types (e.g. if a electronic board is modelled) or more abstract types (e.g. if a protocol is modelled).

3.3.2 Verification

Unless otherwise specified, the verification should be performed using a testbench allowing automated verification as described in section 3.4.1.

3.4 Testbenches

The purpose of a testbench is to verify the functionality of a developed model or package. A testbench shall be a distinct design unit separated from the model or package to be verified, placed in a design library separate from the model itself.

If the testbench incorporates models of components surrounding the model to be tested, they need only to incorporate functions and interfaces required to properly operate with the model under test; it is not necessary to develop complete VHDL models of them. If external stimuli or configuration data is required, it shall be implemented by reading an ASCII file using the *Std.TextIO* package in order to ensure portability.

<u>Every</u> testbench shall stop by itself when the test has been completed, in order to allow the verification to be done using a script, independent of the simulator used.

The root entity shall neither have port nor generic clauses, being potentially not portable.

If several testbenches are used for the verification of a package or a model, no re-compilation shall be necessary in order to perform the complete verification. Neither shall it be necessary to copy any files (or create soft links) used by the testbenches or the model.

If several testbenches are used it is recommended to place the component declaration(s), some signal declarations etc. in a package instead of including them in each testbench.

3.4.1 Automated verification

All testbenches for models for Board-level simulation, for models for System-level simulation and for packages containing subprograms should allow automated verification to be performed. Automated verification allows a reduction of the future maintenance effort, such as verification of the model operation on a different simulator, platform or operating system. Since it enables fast and reliable verification of a model when modifications have been introduced, it is recommended for all types of models.

The verification of error messages and timing parameters can be difficult due to assertions, and may therefore be performed without using automated verification.

The recommended approach is to write testbenches that are self-checking, reporting success or failure for each sub-test. Alternatively, and subject to ESA approval before commencing the development, a testbench could write all values of the signals generated by the model together with time stamps to a text file, which could be verified separately for example by using the Unix *diff* utility. In case a specific program is needed for the file comparison, it shall be delivered in compiled form together with the fully documented source code in the C language. Care should be taken with non-portable issues of *Std.TextIO*, see section 2.5.

APPENDIX A: ABBREVIATIONS

ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
CAD	Computer Aided Design
EIA	Electronic Industries Association
EMC	Electro Magnetic Compatibility
e.g.	exempli gratia (Latin: for example)
etc.	et cetera
ESA	European Space Agency
i.e.	id est (Latin: that is; in other words)
IEEE	Institute of Electrical and Electronics Engineers
LSB	Least Significant Bit
MSB	Most Significant Bit
NFS	Network File System
PLA	Programmable Logic Array
QIC	Quarter Inch Cartridge
RTL	Register Transfer Logic
SDF	Standard Delay File
std	standard
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VITAL	VHDL Initiative Towards ASIC Libraries
w.r.t.	with respect to

APPENDIX B: COMMON ERRORS ENCOUNTERED

This appendix contains examples of some common errors concerning the VHDL syntax and semantics found in some of the VHDL models delivered to ESA. Before delivering any VHDL code, it should be verified w.r.t. these error types in order to avoid the model being immediately rejected for example due to compilation errors.

B.1 Inconsistent subprogram declarations

The subprogram declaration in a package declaration must be identical to the subprogram declaration in the package body, e.g. whenever any of the mode indicators **in**, **out**, **inout** or **buffer** appear in one declaration they must appear exactly at the same position in the other declaration. This also applies for *default expressions* in the declarations.

B.2 Brackets around case statement expression

The following code is illegal if *BV* is a *Bit_Vector*, due to the brackets around the expression:

```
case (BV) is
   when others => null;
end case;
```

B.3 Time limitation encountered at time 0

It shall be ensured that the model does not execute constructs requiring higher resolution than 32 bits on the *Time* type at time 0 or for short simulation times.

B.4 Subtype assigned value outside subtype range

A variable or a signal of a subtype shall not be assigned a value outside the range of that subtype. The following code fragment is illegal in case *UX01* is the subtype defined in the *IEEE.Std_Logic_1164* package, which can only take the values 'U', 'X', '0' and '1':

```
signal OneSignal: UX01 := 'Z';
```

B.5 No range check on values of type Integer and Time

Some simulators do not implement range check on the *Integer* and *Time* types; instead of reporting a run-time error the value is wrapped around (e.g. *Time'Low - 1* gives *Time'High*).

APPENDIX C: COMPATIBILITY BETWEEN VHDL-87 AND VHDL-93

Note: this appendix contains only a limited set of compatibility issues.

In case it has been agreed to start a VHDL model development using the VHDL-87 standard instead of the VHDL-93 standard, the code shall be written so as to require minimal modifications when updating to VHDL-93. As an example, the following identifiers shall not be used, being reserved words in VHDL-93:

> **group**, **impure**, **inertial**, **literal**, **postponed**, **pure**, **reject**, **rol**, **ror**, **shared**, **sla**, **sll**, **sra**, **srl**, **unaffected**, **xnor**

The predefined attributes *'Behavior* and *'Structure* shall not be used, being removed from the VHDL-93 standard.

The constructs for handling files, including the *Std.TextIO* package, are different for VHDL-93 compared to VHDL-87. Therefore shall the code involving file handling be written considering a future update; these constructs should be concentrated to as few places in the code as possible, and clearly commented.

APPENDIX D: VHDL CODE EXAMPLES

This appendix is included as a guidance for VHDL model developers. In case of discrepancies, the requirements have precedence over the examples. The code is provided as is, no functionality is guaranteed.

D.1 VHDL constructs

This section contains code fragments of various VHDL constructs. It is not exhaustive, but contains a sufficient set of constructs to create most types of models. The code fragments have not necessarily been fully commented.

D.1.1 Entity declaration

```
entity ABC is
   generic(SimCondition: SimConditionType := WorstCase);

   port(
      Clk:     in  Bit;                        -- Clock signal
      Reset_N: in  Bit;                        -- Asynchronous Reset
      In1:     in  Bit;                        -- Input 1
      In2:     in  Bit_Vector(1 downto 0);     -- Input 2

      Out1:    out Bit_Vector(7 downto 0));    -- Output, bit 0 is LSB
end ABC;
```

D.1.2 Architecture body

```
architecture RTL of ABC is

   -- Declarations, such as type declarations, constant declarations,
   -- subprograms, component declarations, signal declarations etc.

begin  -- Architecture RTL of ABC

   -- Concurrent statements, e.g. processes, signal assignments and
   -- component instantiations

end RTL;
```

D.1.3 Configuration declaration

This configuration configures the components used to design the TMEncoder design, which is a board with ten components.

```
configuration TMEncoderConfig of TMEncoder is

   for Structural
      for VCA0, VCA1, VCA2, VCA7: VCA
         use configuration VCA_Lib.VCA_Config;
      end for;

      for SRAM0, SRAM1, SRAM2, SRAM7: SRAM
         use entity Mem_Lib.MA9264(BoardLevel);
      end for;

      for VCM1: VCM
         use configuration VCM_Lib.VCM_Config;
      end for;

      for MA1916_1: MA1916
         use configuration MA1916_Lib.MA1916_Config;
      end for;
   end for;

end TMEncoderConfig;
```

D.1.4 Package declaration

```
package TCSuiteDef is

   -- Declarations of (deferred) constants, types, files, subprograms,
   -- components etc. For example:

   subtype   Byte   is Bit_Vector(0 to 7);   -- Bit 0 is MSB
   subtype   Word16 is Bit_Vector(0 to 15);  -- Bit 0 is MSB

   type      ByteArray     is array(Integer range <>) of Byte;
   type      TailErrorType is (All5s,        -- Normal Tail (55555...)
                               SingleFill,   -- Single error + Fill bit
                               Double,       -- Double error
                               DoubleFill);  -- Double error + Fill bit

   constant  CrcPoly: Bit_Vector := X"1021"; -- x16 + x12 + x5 + 1
   constant  InitCrc: Bit_Vector := X"FFFF"; -- Init. to all ones

   --------------------------------------------------------------------
   -- The AddCrc function calculates the CCSDS CRC (syndrome x16 +
   -- x12 + x5 + 1, register initiated to all ones before each data)
   -- over an array of bytes, and appends the calculated CRC.
   -- Data is an unconstrained array of bytes, and the result is of
   -- the same type, with the length increased by 2 (for the CRC).
   --------------------------------------------------------------------
   function AddCrc(Data: ByteArray) return ByteArray;

   -- Description of subprogram function and parameters
   procedure ADFrame(       NR:             Byte;
                            Segment: inout ByteArray;
                     signal TCOut:   out    Bit);

end TCSuiteDef;
```

D.1.5 Package body

```
package body TCSuiteDef is

   -- Declarations of subprograms, deferred constants etc., in the same
   -- order as they appeared in the package declaration.
   -- Also declaration of objects not visible outside the package body.

end TCSuiteDef;
```

D.1.6 Component declaration

```
   component ABC
      generic(SimCondition: SimConditionType := WorstCase);

      port(
         Clk:     in  Bit;                        -- Clock signal
         Reset_N: in  Bit;                        -- Asynchronous Reset
         In1:     in  Bit;                        -- Input 1
         In2:     in  Bit_Vector(1 downto 0);     -- Input 2

         Out1:    out Bit_Vector(7 downto 0));    -- Output
   end component;
```

D.1.7 Component instantiation

In case all signals outside and inside the component have the same name, positional association could be used instead of named association.

```
   U1: ABC
      generic map(SimCondition => BestCase)

      port map(
         Clk     => Clk,
         Reset_N => Reset_N,
         In1     => DataIn1,
         In2     => BaudRate,

         Out1    => DataBusA);
```

D.1.8 Procedure declaration and body

```
   -- Description of subprogram function and parameters
   procedure ADFrame(       NR:             Byte;
                            Segment: inout  ByteArray;
                     signal TCOut:   out    Bit) is
   begin

      -- Sequential statements

   end ADFrame;
```

D.1.9 Function declaration and body

```
---------------------------------------------------------------------
-- The AddCrc function calculates the CCSDS CRC (syndrome x16 +
-- x12 + x5 + 1, register initiated to all ones before each data)
-- over an array of bytes, and appends the calculated CRC.
-- Data is an unconstrained array of bytes, and the result is of
-- the same type, with the length increased by 2 (for the CRC).
---------------------------------------------------------------------
function AddCrc(Data: ByteArray) return ByteArray is
   variable Crc:      Word16 := InitCRC;
   variable Xor1:     Bit;
   variable Result:   ByteArray(0 to (Data'Length + 1));
begin
   -- Calculate the CRC over all the data
   EachByte: for i in Data'Range loop
      EachBit: for BitNo in Byte'Range loop
         Xor1 := Crc(0) xor Data(i)(BitNo);
         Crc  := Crc(1 TO 15) & '0';        -- Shift left 1 bit
         if Xor1 = '1' then
            Crc := Crc xor CRCPoly;
         end if;
      end loop EachBit;
   end loop EachByte;

   -- Add the CRC after the data
   Result(0 to Result'High - 2) := Data;
   Result(Result'High - 1)      := Crc(0 TO  7);
   Result(Result'High)          := Crc(8 TO 15);

   return Result;
end AddCrc;
```

D.1.10 Signal assignment

```
Reset_N <= '0',
           '1' after    79 ns,
           '0' after 10491 ns,
           '1' after 10627 ns;

D       <= DOut       after Tpd_D when DEnable = '1' else
           "ZZZZZZZZ" after Tpd_D;
```

D.1.11 Process statement

```
---------------------------------------------------------------------
-- Process header
---------------------------------------------------------------------
SyncRxIn: process(Clk, Reset_N)             -- Rx synchronizer
begin
   if Reset_N = '0' then                     -- Asynchronous reset
      RxInSync <= '1';
   elsif Clk'Event and Clk = '1' then        -- Rising Clk edge
      RxInSync <= RxIn;
   end if;
end process SyncRxIn;
```

D.1.12 If statement

```
if RxInSync = '1' and RxReg(0) = '0' then -- Wait for start bit
   BaudCount := 0;
elsif (BaudRate = Baud1200 and BaudCount >= Count1200) or
      (BaudRate = Baud9600 and BaudCount >= Count9600) then
   BaudCount := 0;
else
   BaudCount := BaudCount + 1;
end if;
```

D.1.13 Case statement

```
case TailError is
   when All5s =>                              -- Normal tail sequence
      Result := EndCB;
   when SingleFill =>                         -- Set filler bit
      Result := Data;
      Result(7)(7) := '1';                    -- Set filler bit
   when Double | DoubleFill =>                -- Double error
      Result := InjectError(Data);
      if TailError = DoubleFill then
         Result(7)(7) := '1';                 -- Set filler bit
      end if;

   when others =>                             -- No action
      null;
end case;
```

D.1.14 Loop statement

```
EachByte: for i in Data'Range loop

   -- Statements to be executed in the loop

end loop EachByte;
```

D.1.15 Assertion statement

Note that when VHDL-93 has been fully introduced, the new predefined attribute *'Instance_Name* should be used to report the full instantiation path.

```
assert (TestMode = '0')
   report InstancePath & ": Non-implemented test mode activated"
   severity Note;
```

D.2 Complete examples

D.2.1 RS-232 VHDL receiver

This example is representative for one module of a larger component (called *XYZ*). The model is synthesizable with a complexity of about 400 equivalent gates. It is however efficiently and concisely coded to be acceptable as a model for Board-level simulation though timing checks and output delays have not been modelled here.

```
--============================================================================--
-- Design units : RS232_Receive(RTL) (Entity and architecture)
--
-- File name    : rs232_receive.vhd
--
-- Purpose      : The module receives a serial RS-232 bit stream. The
--                bit stream should contain 1 start bit ('0'), 8 data
--                bits and finally 2 stop bits ('1'). The baud rate
--                is selectable to 1200, 2400, 4800 or 9600. The last
--                received data is output in 8-bit parallel format.
--
-- Note         : This model can be synthesized by Synopsys VHDL
--                Compiler and Mentor AutoLogic VHDL.
--
-- Limitations  : The baud rates have been approximated in order to
--                allow a simpler implementation. A Clk frequency of
--                10 MHz is assumed.
--
-- Errors:      : None known
--
-- Library      : XYZ_Lib
--
-- Dependencies : None
--
-- Author       : Peter Sinander
--                ESTEC Onboard Data Division (WD)
--                P.O. Box 299
--                2200 AG Noordwijk
--                The Netherlands
--
-- Simulator    : Synopsys v. 3.0c, on Sun Sparcstation 10, SunOS 4.1.3
--------------------------------------------------------------------------------
-- Revision list
-- Version Author Date       Changes
--
-- 1.0     PSI     4 Mar 94  New version
-- 2.0     PSI    10 May 94  Baudrate index changed to descending;
--                           Constants introduced for baud count;
--                           Header and comments modified.
--------------------------------------------------------------------------------

-- Naming convention: Active low signals are indicated by "_N",
-- synchronized signals are indicated by "Sync".

entity RS232_Receive is
   port(
      Clk:      in  Bit;                       -- Clock, nominally 10 MHz
      Reset_N:  in  Bit;                       -- Asynchronous Reset
      RxIn:     in  Bit;                       -- Serial data in
      BaudRate: in  Bit_Vector(1 downto 0);    -- Bit rate selection

      RxOut:    out Bit_Vector(7 downto 0));   -- Last received data,
end RS232_Receive;                             -- Bit 0 is LSB
```

```
--========================== ARCHITECTURE ============================--

architecture RTL of RS232_Receive is

   -- These constant would normally be placed in a package defining all
   -- constants and subprograms used by the design, but in this
   -- example they have been placed here.
   constant Baud1200:  Bit_Vector := "00";         -- Baud selections
   constant Baud2400:  Bit_Vector := "01";
   constant Baud4800:  Bit_Vector := "10";
   constant Baud9600:  Bit_Vector := "11";

   constant Count1200: Integer := 4096;            -- End count values
   constant Count2400: Integer := 2048;
   constant Count4800: Integer := 1024;
   constant Count9600: Integer :=  512;

   constant InitRxReg: Bit_Vector := "1111111110"; -- Init. pattern

   signal RxInSync:    Bit;                        -- Synchronised Rx

begin  -- Architecture RTL of RS232_Receive

   ---------------------------------------------------------------------
   -- Input serial data is synchronized with Clk to protect against
   -- meta-stability. This process could be merged with the Rs232
   -- process in order to increase the simulation performance (it was
   -- written separately here for the cause of clarity).
   ---------------------------------------------------------------------
   SyncRxIn: process (Clk, Reset_N)
   begin
      if Reset_N = '0' then                        -- Asynchron. reset
         RxInSync <= '1';
      elsif Clk'Event and Clk = '1' then           -- Rising Clk edge
         RxInSync <= RxIn;
      end if;
   end process SyncRxIn;


   ---------------------------------------------------------------------
   -- The Rs232 process contains a counter which toggles the Sample
   -- signal two times per bit period. The rising edge of Sample (which
   -- occurs in the middle of the input bit) is synchronously detected
   -- by comparing it to DelaySample (the Sample signal delayed one Clk
   -- cycle); at this time the data bit is clocked into the shift
   -- register.
   --
   -- The State machine controlling the shift register has been merged
   -- with the shift register itself. When the last bit - RxReg(0) - is
   -- 0 the retrieval cycle has completed and the process is waiting
   -- for the next start bit. When a start bit is detected, the counter
   -- starts incrementing, at each sample time shifting in one data bit
   -- (a start bit shorter than a half bit period will have no impact).
   -- When the start bit, which is '0', reaches RxReg(0) the data is
   -- copied to the output, and the process will wait for the next start
   -- bit.
   ---------------------------------------------------------------------
   Rs232: process (Clk, Reset_N)
      variable BaudCount:   Integer range 0 to 8191; -- 13 bit counter
      variable Sample:      Bit;                     -- For bit sample
      variable DelaySample: Bit;                     -- To detect edge
      variable RxReg:       Bit_Vector(9 downto 0);  -- 10 bit shift
                                                     -- register
```

```
   begin
      if Reset_N = '0' then                           -- Asynchron. reset,
         BaudCount   := 0;                            -- initialize all
         Sample      := '0';                          -- values
         DelaySample := '0';
         RxReg       := InitRxReg;
         RxOut       <= X"00";

      elsif Clk'Event and Clk = '1' then              -- Rising Clk edge
         -- Wait for RxInSync to be 0, i.e. the start bit in the serial
         -- input stream.
         if RxInSync = '1' and RxReg(0) = '0' then
            -- Waiting for the start bit; initialise values
            BaudCount := 0;
            Sample    := '0';
            RxReg     := InitRxReg;
         elsif (BaudRate = Baud1200 and BaudCount >= Count1200) or
               (BaudRate = Baud2400 and BaudCount >= Count2400) or
               (BaudRate = Baud4800 and BaudCount >= Count4800) or
               (BaudRate = Baud9600 and BaudCount >= Count9600) then
            -- The counter has reached half a bit period (assuming that
            -- Clk runs at 10 MHz); reset counter and toggle the Sample
            -- signal (the exact bit rates are 1220, 2441, 4882 & 9765)
            BaudCount := 0;
            Sample    := not Sample;

         else -- RxInSync = '0' or RxReg(0) = '1'
            BaudCount := BaudCount + 1;
         end if;

         if Sample = '1' and DelaySample = '0' then
            -- Rising Sample edge; shift in one data bit
            RxReg := RxInSync & RxReg(9 downto 1);

            if RxReg(0) = '0' and RxReg(8) = '1' and RxReg(9) = '1' then
               -- Last bit acquired, copy data to output if stop
               -- bits are  both '1'
               RxOut <= RxReg(8 downto 1);
            end if;
         end if;

         -- Sample delayed one Clk
         DelaySample := Sample;
      end if;
  end process Rs232;

end RTL; --=========== End of RS232_Receive(RTL) =====================--
```

D.2.2 VHDL model for Board-level simulation

This is an example showing the principle of a VHDL model for board-level simulation. All design units have been included, except the package defining the sub-programs for timing checks:

- ExampleDefinition: Defines constants, functions and conversion functions;
- ExampleTiming: Defines the timing parameters as deferred constants;
- ExampleCore: The functional core, written for high simulation efficiency (most of the code in one process), and with an interesting implementation of the reset functionality;
- Example: The top-level entity/architecture, with signal strength stripping and the timing implementation.

The margins have been extended in order to allow 80 characters per line.

```
--============================================================================--
-- Design units : ExampleDefinition (Package declaration and body)
--
-- File name    : exampledefinition.vhd
--
-- Purpose      : Package defining constants and functions for the Example.
--                Defines constants and types for the functions as implemented
--                by the Example.
--                Defines conversion functions/procedures.
--
-- Limitations  : None
--
-- Errors:      : None known
--
-- Library      : Example_Lib
--
-- Dependencies : IEEE.Std_Logic_1164
--
-- Author       : Peter Sinander
--                ESTEC Onboard Data Division (WD)
--                P.O. Box 299
--                2200 AG Noordwijk
--                The Netherlands
--
-- Simulator    : Synopsys v. 3.0c, on Sun Sparcstation 10, SunOS 4.1.3
--------------------------------------------------------------------------------
-- Revision list
-- Version Author Date       Changes
--
-- 1.0     PSI     1 Sep 94  New version
--------------------------------------------------------------------------------

library IEEE;
use IEEE.Std_Logic_1164.all;


package ExampleDefinition is
   -----------------------------------------------------------------------------
   -- Definition of common Std_ULogic vector sizes
   -- Note: Bit 0 is the MSB
   -----------------------------------------------------------------------------
   subtype Std_Byte         is Std_ULogic_Vector(0 to 7);
   subtype Std_Word16       is Std_ULogic_Vector(0 to 15);
   subtype Std_Word32       is Std_ULogic_Vector(0 to 31);
```

```
---------------------------------------------------------------------------
-- Definition of the fixed part of the preamble
---------------------------------------------------------------------------
constant FixPreamble:    Std_Word32 := To_StdULogicVector(X"89_AB_CD_EF");

---------------------------------------------------------------------------
-- Constant fixed field of the preamble
---------------------------------------------------------------------------
constant FixedField:     Std_ULogic_Vector := "00";

---------------------------------------------------------------------------
-- Length of preamble
-- Position of the Line Count field after the preamble
---------------------------------------------------------------------------
constant PreambleLen:    Integer := FixPreamble'Length + 8;
constant LineCountEnd:   Integer := PreambleLen + 8;

---------------------------------------------------------------------------
-- Number Clk cycles for the Built In Self Test, BIST, and time after reset
-- when no BIST is running
---------------------------------------------------------------------------
constant BistClks:       Integer := 16384;
constant NoBistClks:     Integer :=     1;

---------------------------------------------------------------------------
-- Number of entries in the memory
-- Definition of Mem type
---------------------------------------------------------------------------
constant MemSize:        Integer := 255;
type     MemType is      array(0 to MemSize-1) of Integer range 0 to 255;

---------------------------------------------------------------------------
-- Calculation of Valid and FSM lengths
-- Valid is 1200, 2400, 4800 or 9600 depending on Mode
-- FSM is same as Valid, plus a gap of 400 system clocks between data bursts
-- when LowSpeed is 1
---------------------------------------------------------------------------
function CalcValidLength(Mode:     Std_ULogic_Vector(0 to 1))
                         return    Integer;

function CalcFSMLength  (Mode:     Std_ULogic_Vector(0 to 1);
                         LowSpeed: Std_ULogic)
                         return    Integer;

---------------------------------------------------------------------------
-- Converts Natural to Std_ULogic_Vector of length Len
-- Leftmost bit is most significant
---------------------------------------------------------------------------
function To_StdULogicVector(I:     Natural;
                            Len:   Positive)
                            return Std_ULogic_Vector;

---------------------------------------------------------------------------
-- Converts unsigned Std_ULogic_Vector to Natural
-- Leftmost bit is most significant
-- No warning for unknowns (U, X, W, Z, -), they are converted to 0
-- Verifies whether vector is too long (> 31 bits)
---------------------------------------------------------------------------
function To_Integer(V:     Std_ULogic_Vector)
                    return Natural;

---------------------------------------------------------------------------
-- Wrap-around addition between two Std_ULogic_Vectors of the same length
-- Leftmost bit is most significant
-- Verifies whether both vectors have the same length
---------------------------------------------------------------------------
function "+"(R, L:  Std_ULogic_Vector)
             return Std_ULogic_Vector;
end ExampleDefinition;
```

```
package body ExampleDefinition is

   -----------------------------------------------------------------------------
   -- Calculation of Valid length
   -- Valid is 1200, 2400, 4800 or 9600 depending on Mode
   -----------------------------------------------------------------------------
   function CalcValidLength(Mode:  Std_ULogic_Vector(0 to 1))
                            return Integer is
   begin
      if Mode = "00" then
         return 1200;                                  -- Mode 0
      elsif Mode = "01" then
         return 2400;                                  -- Mode 1
      elsif Mode = "10" then
         return 4800;                                  -- Mode 2
      else
         -- Default value for unknowns as well as for 11
         return 9600;                                  -- Default mode
      end if;
   end CalcValidLength;


   -----------------------------------------------------------------------------
   -- Calculation of FSM length
   -- FSM is 1200, 2400, 4800 or 9600 depending on Mode, plus a gap of 400
   -- system clocks between data bursts when LowSpeed is 1
   -----------------------------------------------------------------------------
   function CalcFSMLength(Mode:     Std_ULogic_Vector(0 to 1);
                          LowSpeed: Std_ULogic)
                            return    Integer is
   begin
      if LowSpeed = '0' then
         -- Highest speed, no gap between data bursts, same as Valid length
         if Mode = "00" then
            return 1200;                               -- Mode 0
         elsif Mode = "01" then
            return 2400;                               -- Mode 1
         elsif Mode = "10" then
            return 4800;                               -- Mode 2
         else
            -- Default value for unknown Mode as well as for 11
            return 9600;                               -- Default mode
         end if;
      else
         -- Insert gap of 400 system clocks between data bursts
         if Mode = "00" then
            return 1600;                               -- Mode 0 + 400
         elsif Mode = "01" then
            return 2800;                               -- Mode 1 + 400
         elsif Mode = "10" then
            return 5200;                               -- Mode 2 + 400
         else
            -- Default value for unknown Mode as well as for 11
            return 10000;                              -- Default mode + 400
         end if;
      end if;
   end CalcFSMLength;


   -----------------------------------------------------------------------------
   -- Converts Natural to Std_ULogic_Vector of length Len
   -- Leftmost bit is most significant
   -----------------------------------------------------------------------------
   function To_StdULogicVector(I:     Natural;
                               Len:   Positive)
                               return Std_ULogic_Vector is
      variable Tmp:    Integer;
      variable Result: Std_ULogic_Vector(0 to Len - 1);
```

```
   begin
      Tmp := I;

      for j in Result'Reverse_Range loop
         if (Tmp mod 2) = 1 then
            Result(j) := '1';
         else
            Result(j) := '0';
         end if;
         Tmp := Tmp / 2;
      end loop;

      return Result;
   end To_StdULogicVector;


   ------------------------------------------------------------------------------
   -- Converts unsigned Std_ULogic_Vector to Natural
   -- Leftmost bit is most significant
   -- No warning for unknowns (U, X, W, Z, -), they are converted to 0
   -- Verifies whether vector is too long (> 31 bits)
   ------------------------------------------------------------------------------
   function To_Integer(V:     Std_ULogic_Vector)
                      return Natural is
      variable Result: Integer := 0;
   begin
      assert V'Length <= 31
         report "Can not convert more than 31 bit Std_ULogic_Vectors"
         severity Failure;
      for i in V'Range loop
         Result := Result * 2;
         if (V(i) = '1') or (V(i) = 'H') then
            Result := Result + 1;
         end if;
      end loop;

      return Result;
   end To_Integer;


   ------------------------------------------------------------------------------
   -- Wrap-around addition between two Std_ULogic_Vectors of the same length
   -- Leftmost bit is most significant
   -- Verifies whether both vectors have the same length
   ------------------------------------------------------------------------------
   function "+"(R, L:  Std_ULogic_Vector)
               return Std_ULogic_Vector is
      variable Carry: Std_ULogic := '0';
      variable RTmp, LTmp, Result: Std_ULogic_Vector((R'Length - 1) downto 0);
   begin
      assert R'Length = L'Length
         report "Vectors to be added are not of same length"
         severity Failure;

      RTmp := R;                          -- To get the range (MSB downto 0)
      LTmp := L;                          --             -- " --
      for i in 0 to RTmp'Length - 1 loop
         -- Calculate sum using carry from previous step, then carry out
         Result(i) := RTmp(i) xor LTmp(i) xor Carry;
         Carry :=    (RTmp(i) and LTmp(i)) or (RTmp(i) and Carry) or
                     (LTmp(i) and Carry);
      end loop;
      return Result;
   end "+";

end ExampleDefinition;
```

```
--============================================================================
-- Design units : ExampleTiming (Package declaration and body)
--
-- File name    : exampletiming.vhd
--
-- Purpose      : In this package, all timing parameters for the Example are
--                defined as deferred constants; their value can be modified
--                by recompiling only the package body and no other files.
--
-- Note         : The timing figures have been taken from the data sheet.
--                The timing figures are based on 50 pF load on the outputs.
--
-- Limitations  : Best case and typical figures have been estimated.
--                Note that simulation with timing checks CANNOT replace
--                a worst case timing analysis.
--
-- Errors       : None known
--
-- Naming       : Names of timing parameters are compliant with SDF (Standard
-- convention     Delay Format).
--
-- Library      : Example_Lib
--
-- Dependencies : ESA.Simulation
--
-- Author       : Sandi Habinc, Peter Sinander
--                ESTEC Onboard Data Division (WD)
--                P. O. Box 299
--                2200 AG Noordwijk
--                The Netherlands
--
-- Simulator    : Synopsys v. 3.0c, on Sun Sparcstation 10, SunOS 4.1.3
--------------------------------------------------------------------------------
-- Revision list
-- Version Author Date       Changes
--
-- 1.0     PSI     1 Sep 94  New version
--------------------------------------------------------------------------------

library ESA;
use ESA.Simulation.all;


package ExampleTiming is

   -----------------------------------------------------------------------------
   -- Deferred constants for the timing parameters, all values are defined in
   -- the package body.
   --
   -- Test, Mode, LowSpeed, Code : not allowed to change while Reset_N is
   -- de-asserted (checked in model).
   --
   -- Reset_N, CS_N de-asserted after write: timing requirement expressed in
   -- number of clock cycles (checked in model).
   -----------------------------------------------------------------------------

   -- System signal timing parameters                  Data sheet reference
   constant tperiod_Clk:      TimeArray;            -- TCp
   constant tpw_hi_min_Clk:   TimeArray;            -- TCLo
   constant tpw_lo_min_Clk:   TimeArray;            -- TCHi
```

```
   -- Mem interface timing parameters
   constant tsetup_A_CS_N:    TimeArray;             -- T5
   constant thold_A_CS_N:     TimeArray;             -- T6
   constant tsetup_RW_N_CS_N: TimeArray;             -- T1
   constant thold_RW_N_CS_N:  TimeArray;             -- T2
   constant tpw_lo_min_CS_N:  TimeArray;             -- T3
   constant tsetup_D_CS_N:    TimeArray;             -- T5
   constant thold_D_CS_N:     TimeArray;             -- T6
   constant tpd_CS_N_D:       TimeArray;             -- T7
   constant tpd_CS_N_D_Z:     TimeArray;             -- T9
   constant tpd_A_D:          TimeArray;             -- T8

   -- Serial input interface timing parameters
   constant tsetup_Clk_Ready: TimeArray;             -- T10
   constant thold_Clk_Ready:  TimeArray;             -- T11
   constant tsetup_Clk_SIn:   TimeArray;             -- T12
   constant thold_Clk_SIn:    TimeArray;             -- T13

   -- Output interface timing parameters
   constant tpd_Clk_SOut:     TimeArray;             -- T4
   constant tpd_Clk_Valid:    TimeArray;             -- T4
end ExampleTiming;


package body ExampleTiming is
   ----------------------------------------------------------------------------
   -- Deferred constants for the timing parameters, all values are defined in
   -- the package body
   --
   -- Test, Mode, LowSpeed, Code : not allowed to change while Reset_N is
   -- de-asserted (checked in model).
   --
   -- Definition of default timing parameter values with 50 pF load
   -- The timing figures have been taken from the data sheet
   ----------------------------------------------------------------------------

   -- System signal timing parameters       WC     Typ    BC         Ref.
   constant tperiod_Clk:      TimeArray := (80 ns, 66 ns, 50 ns); -- TCp
   constant tpw_hi_min_Clk:   TimeArray := (40 ns, 33 ns, 25 ns); -- TCLO
   constant tpw_lo_min_Clk:   TimeArray := (40 ns, 33 ns, 25 ns); -- TCHI

   -- Mem interface timing parameters       WC     Typ    BC         Ref.
   constant tsetup_A_CS_N:    TimeArray := (10 ns,  7 ns,  5 ns); -- T5
   constant thold_A_CS_N:     TimeArray := (10 ns,  7 ns,  4 ns); -- T6
   constant tsetup_RW_N_CS_N: TimeArray := ( 0 ns,  0 ns,  0 ns); -- T1
   constant thold_RW_N_CS_N:  TimeArray := ( 3 ns,  5 ns,  6 ns); -- T2
   constant tpw_lo_min_CS_N:  TimeArray := (50 ns, 40 ns, 30 ns); -- T3
   constant tsetup_D_CS_N:    TimeArray := (10 ns,  7 ns,  5 ns); -- T5
   constant thold_D_CS_N:     TimeArray := (10 ns,  7 ns,  4 ns); -- T6
   constant tpd_CS_N_D:       TimeArray := (45 ns, 35 ns, 25 ns); -- T7
   constant tpd_CS_N_D_Z:     TimeArray := (35 ns, 35 ns, 35 ns); -- T9
   constant tpd_A_D:          TimeArray := (60 ns, 53 ns, 45 ns); -- T8

   -- Serial input interface timing         WC     Typ    BC         Ref.
   constant tsetup_Clk_Ready: TimeArray := ( 5 ns,  4 ns,  3 ns); -- T10
   constant thold_Clk_Ready:  TimeArray := (10 ns,  8 ns,  5 ns); -- T11
   constant tsetup_Clk_SIn:   TimeArray := ( 5 ns,  4 ns,  3 ns); -- T12
   constant thold_Clk_SIn:    TimeArray := (10 ns,  8 ns,  5 ns); -- T13

   -- Output interface timing parameters    WC     Typ    BC         Ref.
   constant tpd_Clk_SOut:     TimeArray := (30 ns, 22 ns, 15 ns); -- T5
   constant tpd_Clk_Valid:    TimeArray := (30 ns, 22 ns, 15 ns); -- T5
end ExampleTiming;
```

```
--============================================================================--
-- Design units : ExampleCore(FunctionalCore) (Entity and architecture)
--
-- File name    : examplecore.vhd
--
-- Purpose      : This is the functional core of an example VHDL model called
--                Example. The core implements all the functionality, except
--                the multiplexing of the data bus D which is performed in the
--                top-level architecture.
--
-- Note         : All timing, checking and conversion of logical values are
--                performed in the top-level architecture.
--                X-propagation is implemented for the SIn and Code inputs, but
--                not for data written to the parallel interface.
--
--                The functionality does not represent an existing component.
--
--                The model is intended for efficient simulation at board level
--                and is not synthesizable.
--
--                Since no real function is modelled, the comments have
--                sometimes been reduced.
--
-- Limitations  : BIST internal function not modelled, only the resulting delay
--                after reset. Manufacturing test not modelled.
--
-- Errors:      : None known (model not verified)
--
-- Naming       : Active low signals are indicated by _N.
-- convention     All external signals have been named as in the data sheet.
--
-- Library      : Example_Lib
--
-- Dependencies : IEEE.Std_Logic_1164,
--                Example_Lib.ExampleDefinition
--
-- Author       : Peter Sinander
--                ESTEC On-board Data Division (WD)
--                P. O. Box 299
--                2200 AG Noordwijk
--                The Netherlands
--
-- Simulator    : Synopsys v. 3.0c, on Sun Sparcstation 10, SunOS 4.1.3
--------------------------------------------------------------------------------
-- Revision list
-- Version Author Date       Changes
--
-- 1.0     PSI     1 Sep 94  New version
--------------------------------------------------------------------------------


library IEEE;
use IEEE.Std_Logic_1164.all;

library Example_Lib;
use Example_Lib.ExampleDefinition.all;


entity ExampleCore is
   port (
      -- System signals
      Test0:     in    Std_ULogic;                     -- 0 to activate BIST
      Clk:       in    Std_ULogic;                     -- System clock
      Reset_N:   in    Std_ULogic;                     -- System async reset

      -- Mode pins for selecting the operation + static fields
      Mode:      in    Std_ULogic_Vector(0 to 1);      -- Selects mode
      LowSpeed:  in    Std_ULogic;                     -- Lower speed when 1
      Code:      in    Std_ULogic_Vector(0 to 5);      -- Code input 6 bits
```

```
      -- Parallel interface
      A:          in    Std_Byte;                     -- Address bus
      CS_N:       in    Std_ULogic;                   -- Chip select, act. low
      RW_N:       in    Std_ULogic;                   -- Read/write, read = 1
      D:          in    Std_Logic_Vector(0 to 7);     -- Data bus in
      DOut:       out   Integer range 0 to 255;       -- Data bus output
      DEnable:    out   Boolean;                      -- Data bus enable

      -- Serial input interface
      Ready:      in    Std_ULogic;                   -- Data input ready
      SIn:        in    Std_ULogic;                   -- Serial input data

      -- Resulting serial output and valid strobe
      SOut:       out   Std_ULogic;                   -- Serial data output
      Valid:      out   Std_ULogic);                  -- 1 when output valid
end ExampleCore;

--============================== ARCHITECTURE ================================--

architecture FunctionalCore of ExampleCore is
   signal ValidLen:    Integer range 0 to  9600;       -- Valid FSM states
   signal EndOfFSM:    Integer range 0 to 10000;       -- Where the FSM ends
   signal Preamble:    Std_ULogic_Vector(0 to PreambleLen-1); -- Concat preamble
   signal MainReset:   Boolean := True;                -- Reset or BIST

   signal DWrite:      Integer range 0 to 255;         -- Memory data to write
   signal AWrite:      Integer range 0 to 255;         -- Address to write data
   signal WStrobe:     Std_ULogic;                     -- Async. write strobe

begin --=========== Architecture FunctionalCore of ExampleCore ===============--

   ----------------------------------------------------------------------------
   -- Calculation of valid and FSM lengths
   ----------------------------------------------------------------------------
   ValidLen  <= CalcValidLength(Mode);
   EndOfFSM  <= CalcFSMLength(Mode, LowSpeed) - 1;

   ---------------------------------------------------------------------------
   -- Generation of preamble part that seldom changes
   ---------------------------------------------------------------------------
   Preamble <= FixPreamble & FixedField & Code;

   ----------------------------------------------------------------------------
   -- Implementation of all functionality driven by Clk, i.e. ...
   -- (Here a full description should normally be placed)
   -- Note that the Reset signal is synchronized, and is therefore not included
   -- in the sensitivity list.
   -- Inclusion of events on the A address signal in order to synchronize
   -- data and address from the asynchronous memory interface.
   ----------------------------------------------------------------------------
   ClkRegion: process(Clk, A)
      variable Reset1_N:    Std_ULogic := '1';         -- Synchronized reset
      variable Reset2_N:    Std_ULogic := '1';         -- Synchronized reset
      variable BistCount:   Integer range -1 to BistClks := -1; -- No init = -1

      variable FSMCount:    Integer range 0 to 10000;  -- Which bit of FSM
      variable LineCount:   Std_Byte;                  -- Line Counter
      variable DataOut:     Std_ULogic;                -- Serial data output

      variable DelayedSIn:  Std_ULogic;                -- Registered Sin bit

      variable MemData:     Integer range 0 to 255;    -- Data read from Mem
      variable Mem:         MemType;                   -- 256*8 bit memory
      variable A_Integer:   Integer range 0 to 255;    -- A in integer format
      variable AWrite1:     Integer range 0 to 255;    -- Delayed write address
      variable DWrite1:     Integer range 0 to 255;    -- Delayed Mem write data
      variable AWrite2:     Integer range 0 to 255;    -- Delayed write address
      variable DWrite2:     Integer range 0 to 255;    -- Delayed Mem write data
```

```
begin
  if Falling_Edge(Clk) then                           -- Falling Clk edge

     ------------------------------------------------------------------
     -- Code dealing with the Reset initialization
     ------------------------------------------------------------------
     -- Delay 2 Clk of Reset_N due to synchronization
     Reset2_N := Reset1_N;
     Reset1_N := Reset_N;

     if Reset2_N = '0' then                           -- Reset the Example
        -- Select delay for BIST or for no BIST
        if Test0 = '1' then
           BistCount := NoBistClks;                   -- BIST disabled
        else
           BistCount := BistClks;
        end if;

        FSMCount     := 0;
        LineCount    := "00000000";

        DelayedSIn   := '0';
        AWrite1      := 0;
        DWrite1      := 0;
        AWrite2      := 0;
        DWrite2      := 0;
        Mem          := (others => 0);                -- Initialize memory
        DOut         <= Mem(A_Integer);

        -- Output values at reset
        SOut         <= '0';
        Valid        <= '0';

     ------------------------------------------------------------------
     -- Normal operation after reset and BIST (if enabled)
     ------------------------------------------------------------------
     elsif (BistCount = 0) then
        ---------------------------------------------------------------
        -- The serial data output, containing of the Preamble, the line
        -- count and the serial input data SIn
        if FSMCount < LineCountEnd then
           -- Optimized if-structure to execute only when necessary
           if FSMCount < PreambleLen then             -- Sync. Mark +
              DataOut := Preamble(FSMCount);          -- Preamble bytes
           else                                       -- Line Counter byte
              DataOut := LineCount(FSMCount mod 8);
           end if;

        elsif FSMCount < ValidLen then                -- Output data from SIn
           DataOut := DelayedSIn;
        else                                          -- Reed-Solomon codes
           DataOut := '0';
        end if;


        ---------------------------------------------------------------
        -- Generation of SOut
        -- Generation of Valid; '1' while the input data is being output
        -- if the data input is ready (i.e. Ready = '1')
        SOut <= DataOut;
        if FSMCount < FixPreamble'Length then         -- Output invalid
           Valid <= '0';
        else
           if FSMCount = FixPreamble'Length then
              Valid <= Ready;
           elsif FSMCount = ValidLen then
              Valid <= '0';
           end if;
        end if;
```

```
        --------------------------------------------------------------------
        -- Writing of data into the Mem; delayed 2.5 Clk cycles for
        -- synchronization reasons (first delay on rising Clk edge)
        -- Change DOut in case the corresponding Mem data was changed
        Mem(AWrite2) := DWrite2;
        AWrite2      := AWrite1;
        DWrite2      := DWrite1;
        DOut         <= Mem(A_Integer);


        --------------------------------------------------------------------
        -- Delay of SIn with 1 Clock cycle (it was registered in order
        -- to reduce the setup time)
        DelayedSIn := SIn;


        --------------------------------------------------------------------
        -- Implementation of FSM counter (for FSM) and Line Counter
        if FSMCount < EndOfFSM then
           -- Increment bit counter
           FSMCount  := FSMCount + 1;
        else
           -- End of FSM reached: reset FSM counter & increment Line Count
           FSMCount  := 0;
           LineCount := LineCount + "00000001";
        end if;


     -----------------------------------------------------------------------
     -- Model Bist delay. In case Reset has never been asserted,
     -- BistCount = -1, and no action will take place
     -----------------------------------------------------------------------
     elsif BistCount > 0 then
        BistCount := BistCount - 1;

        -- Release MainReset when the BIST has completed
        -- Prepare Reset1_N & Reset2_N for the next reset
        if BistCount = 0 then
           MainReset <= False;
           Reset1_N  := '1';
           Reset2_N  := '1';
        end if;
     end if;


  --------------------------------------------------------------------------
  -- First latching of parallel interface address & data on Rising Clk edge
  --------------------------------------------------------------------------
  elsif Rising_Edge(Clk) then
     AWrite1 := AWrite;
     DWrite1 := DWrite;
  end if;

  --------------------------------------------------------------------------
  -- Output parallel data on internal bus whenever the address changes
  -- Only convert A to integer when it changes (used elsewhere in process)
  --------------------------------------------------------------------------
  if A'Event then
     A_Integer := To_Integer(A);
     DOut      <= Mem(A_Integer);
  end if;
end process ClkRegion;
```

```
-----------------------------------------------------------------------------
-- Latching of address & data for the parallel interface
-- Generation of external data bus enable
-----------------------------------------------------------------------------

-- Data and address to be written is latched on the rising edge of WStrobe
WStrobe <= CS_N or RW_N;

WriteMem: process(WStrobe, MainReset)
begin
   if MainReset then
      AWrite <= 0;
      DWrite <= 0;
   elsif WStrobe'Event and WStrobe = '1' then
      AWrite <= To_Integer(A);
      DWrite <= To_Integer(To_StdULogicVector(D));
   end if;
end process WriteMem;

-- Enabled for read cycles when not Reset
DEnable <= (Reset_N = '1') and (RW_N = '1') and (CS_N = '0');

end FunctionalCore; --==== End of ExampleCore(FunctionalCore) =================--
```

```
--============================================================================
-- Design units : Example(BoardLevel) (Entity and architecture)
--
-- File name    : example.vhd
--
-- Purpose      : This is an example VHDL model called Example. For a real
--                model the functionality should be described here, together
--                with a reference to the applicable data sheet.
--
-- Note         : Selection of Worst Case, Typical or Best Case timing
--                is performed by changing the SimCondition generic.
--
--                X-propagation is implemented for the Code and SIn inputs, but
--                not for data written to the parallel interface.
--
--                Timing violations will not lead to unknown being generated.
--
--                The model is intended for efficient simulation at board level
--                and is not synthesizable.
--
-- Limitations  : BIST internal function not modelled, only the resulting delay
--                after reset. Manufacturing test not modelled.
--
--                Do not use timing modelling to replace worst case timing
--                analysis; the timing modelling is not always accurate.
--
-- Errors:      : Timing and X checks have not been implemented for all inputs.
--
-- Naming       : Active low signals are indicated by _N.
-- convention     All external signals have been named as in the data sheet.
--                Internal, strength converted signals are named after their
--                new strength, for example _X01. Internal signals without
--                output delay are indicated by _NoTime.
--
-- Library      : Example_Lib
--
-- Dependencies : IEEE.Std_Logic_1164
--                ESA.Simulation
--                Example_Lib.ExampleCore
--                Example_Lib.ExampleDefinition
--                Example_Lib.ExampleTiming
--                Example_Lib.TimingChecks (Note: code not included)
--
-- Author       : Sandi Habinc, Peter Sinander
--                ESTEC Onboard Data Division (WD)
--                P. O. Box 299
--                2200 AG Noordwijk
--                The Netherlands
--
-- Simulator    : Synopsys v. 3.0c, on Sun Sparcstation 10, SunOS 4.1.3
--------------------------------------------------------------------------------
-- Revision list
-- Version Author Date       Changes
--
-- 1.0     PSI     1 Sep 94  New version
--------------------------------------------------------------------------------

library IEEE;
use IEEE.Std_Logic_1164.all;                          -- For signal types

library ESA;
use ESA.Simulation.all;                               -- For simulation mode


entity Example is
   generic(
      SimCondition:   SimConditionType := WorstCase;   -- Simulation condition
      InstancePath:   String           := "Example:";  -- For Assertions
      TimingChecksOn: Boolean          := False);      -- Timing disabling
```

```
   port (
      -- System signals (4)
      Test:       in    Std_Logic_Vector(0 to 1);       -- Test inputs
      Clk:        in    Std_Logic;                      -- System clock
      Reset_N:    in    Std_Logic;                      -- System async reset

      -- Mode pins for selecting the operation + static fields (9)
      Mode:       in    Std_Logic_Vector(0 to 1);       -- Selects mode
      LowSpeed:   in    Std_Logic;                      -- Lower speed when 1
      Code:       in    Std_Logic_Vector(0 to 5);       -- Code input 6 bits

      -- Parallel interface (18)
      A:          in    Std_Logic_Vector(0 to 7);       -- Address bus
      CS_N:       in    Std_Logic;                      -- Chip select, act. low
      RW_N:       in    Std_Logic;                      -- Read/write, read = 1
      D:          inout Std_Logic_Vector(0 to 7);       -- Data bus

      -- Serial input interface (2)
      Ready:      in    Std_Logic;                      -- Data input ready
      SIn:        in    Std_Logic;                      -- Serial input data

      -- Resulting serial output and valid strobe (2)
      SOut:       out   Std_ULogic;                     -- Serial data output
      Valid:      out   Std_ULogic);                    -- 1 when output valid
end Example;

--================================ ARCHITECTURE =================================--

library Example_Lib;
use Example_Lib.ExampleDefinition.all;                  -- For functions
use Example_Lib.ExampleTiming.all;                      -- For timing parameters
use Example_Lib.TimingChecks.all;                       -- Code not included


architecture BoardLevel of Example is

   -----------------------------------------------------------------------------
   -- Component declaration
   -----------------------------------------------------------------------------
   component ExampleCore
      port (
         -- System signals
         Test0:      in    Std_ULogic;                  -- 0 to activate BIST
         Clk:        in    Std_ULogic;                  -- System clock
         Reset_N:    in    Std_ULogic;                  -- System async reset

         -- Mode pins for selecting the operation + static fields
         Mode:       in    Std_ULogic_Vector(0 to 1);   -- Selects mode
         LowSpeed:   in    Std_ULogic;                  -- Lower speed when 1
         Code:       in    Std_ULogic_Vector(0 to 5);   -- Code input 6 bits

         -- Parallel interface
         A:          in    Std_Byte;                    -- Address bus
         CS_N:       in    Std_ULogic;                  -- Chip select, act. low
         RW_N:       in    Std_ULogic;                  -- Read/write, read = 1
         D:          in    Std_Logic_Vector(0 to 7);    -- Data bus in
         DOut:       out   Integer range 0 to 255;      -- Data bus output
         DEnable:    out   Boolean;                     -- Data bus enable

         -- Serial input interface
         Ready:      in    Std_ULogic;                  -- Data input ready
         SIn:        in    Std_ULogic;                  -- Serial input data

         -- Resulting serial output and valid strobe
         SOut:       out   Std_ULogic;                  -- Serial data output
         Valid:      out   Std_ULogic);                 -- 1 when output valid
   end component;
```

```
---------------------------------------------------------------------------
-- Local signal declarations, for input strength conversion, output signals
-- without delay and signals for the data bus control
---------------------------------------------------------------------------
signal Test0_X01:       Std_ULogic;                   -- 0 to activate BIST
signal Clk_X01:         Std_ULogic;                   -- System clock
signal Reset_N_X01:     Std_ULogic;                   -- System async reset
signal Mode_X01:        Std_ULogic_Vector(0 to 1);    -- Selects mode
signal LowSpeed_X01:    Std_ULogic;                   -- Lower speed when 1
signal Code_X01:        Std_ULogic_Vector(0 to 5);    -- Code input 6 bits

signal A_X01:           Std_Byte;                     -- Address bus
signal CS_N_X01:        Std_ULogic;                   -- Chip select, act. low
signal RW_N_X01:        Std_ULogic;                   -- Read/write, read = 1
signal D_X01:           Std_Logic_Vector(0 to 7);     -- Input data
signal DOut:            Integer range 0 to 255;       -- Data bus output
signal DOutDelayed:     Integer range 0 to 255;       -- D delayed wrt address
signal DEnable:         Boolean;                      -- Data bus enable
signal DEnDelayed:      Boolean;                      -- Enable delayed wrt CS

signal Ready_X01:       Std_ULogic;                   -- Data input ready
signal SIn_X01:         Std_ULogic;                   -- Serial input data
signal SOut_NoTime:     Std_ULogic;                   -- Serial data output
signal Valid_NoTime:    Std_ULogic;                   -- 1 when output valid

-- Used for enabling the input timing checks and for storing timing check status
signal AfterReset:      Boolean;                      -- True after reset
signal ClkInfo:         Time;                         -- Status for Clk period
signal CS_NInfo:        Time;                         -- Status for Clk period
begin --=============== Architecture BoardLevel of Example ===================-

---------------------------------------------------------------------------
-- Strength stripping to X01 using the Std_Logic_1164 provided routines
---------------------------------------------------------------------------
Test0_X01    <= To_X01(Test(0));
Clk_X01      <= To_X01(Clk);
Reset_N_X01  <= To_X01(Reset_N);
Mode_X01     <= To_StdULogicVector(To_X01(Mode));
LowSpeed_X01 <= To_X01(LowSpeed);
Code_X01     <= To_StdULogicVector(To_X01(Code));
A_X01        <= To_StdULogicVector(To_X01(A));
CS_N_X01     <= To_X01(CS_N);
RW_N_X01     <= To_X01(RW_N);
D_X01        <= To_X01(D);
Ready_X01    <= To_X01(Ready);
SIn_X01      <= To_X01(SIn);


---------------------------------------------------------------------------
-- Check for unknown values on the static inputs, and that they only change
-- during reset). Check for unknown values on Reset_N.
-- Activating production test and changing the code inputs do not change the
-- state of the model, and have therefore severity level Note resp. Warning.
---------------------------------------------------------------------------
CheckStaticInputs: process(Reset_N_X01, Mode_X01, LowSpeed_X01, Code_X01)
begin
   if (Now /= 0 ns) and (Reset_N_X01 = '1') then
      -- No assertions at start-up or when Reset is asserted
      assert not Is_X(Test)
         report InstancePath & " 'X' on Test inputs"       severity Error;
      assert (Test(1) = '0')
         report InstancePath & " Prod. test not modelled"  severity Note;
      assert not Is_X(Mode_X01)
         report InstancePath & " 'X' on Mode input"        severity Error;
      assert LowSpeed_X01 /= 'X'
         report InstancePath & " 'X' on LowSpeed input"    severity Error;
      assert not Is_X(Code_X01)
            report InstancePath & " 'X' on Code inputs"    severity Warning;
```

```
      -- Check if the static pins changed after Reset
      assert not Test'Event
         report InstancePath & " Test changed after reset" severity Error;
      assert not Mode_X01'Event
         report InstancePath & " Mode changed after reset" severity Error;
      assert not LowSpeed_X01'Event
         report InstancePath & " LowSpeed changed after reset" severity Error;
      assert not Code_X01'Event
         report InstancePath & " Code changed after reset" severity Warning;

   elsif (Now /= 0 ns) and Reset_N'Event then -- Check for X on Reset_N
      assert Reset_N_X01 /= 'X'
         report InstancePath & " 'X' on Reset_N input"     severity Error;

   end if;
end process CheckStaticInputs;


---------------------------------------------------------------------------
-- Timing checks on inputs (setup, hold, period, pulse width).
---------------------------------------------------------------------------

-- Enabling of the checkers when reset is de-asserted (1 ns delay in order
-- to avoid messages at start-up
AfterReset   <= TimingChecksOn and (Reset_N_X01 = '1') after 1 ns;

-- Clk period, high and low times (TCp, TCLo, TCHi)
PeriodCheck(   TestPort      => Clk_X01,
               TestPortName  => "Clk",
               PeriodMin     => tperiod_Clk   (SimCondition),
               Pw_Hi_Min     => tpw_hi_min_Clk(SimCondition),
               Pw_Lo_Min     => tpw_lo_min_Clk(SimCondition),
               Info          => ClkInfo,
               CheckEnabled  => TimingChecksOn,
               HeaderMsg     => InstancePath,
               SeverityLevel => Error);

-- CS_N asserted time (T3), PeriodMin and Pw_Hi_Min defined by default values
PeriodCheck(   TestPort      => CS_N_X01,
               TestPortName  => "CS_N",
               Pw_Lo_Min     => tpw_lo_min_CS_N(SimCondition),
               Info          => CS_NInfo,
               CheckEnabled  => TimingChecksOn,
               HeaderMsg     => InstancePath,
               SeverityLevel => Error);

-- Ready setup & hold wrt Clk (T10, T11); does not affect state => Warning
SetupHoldCheck(TestPort      => Ready_X01,
               TestPortName  => "Ready",
               RefPort       => Clk_X01,
               RefPortName   => "Clk",
               RefEdge       => '0',
               TSetup        => tsetup_Clk_Ready(SimCondition),
               THold         => thold_Clk_Ready (SimCondition),
               CheckEnabled  => AfterReset,
               HeaderMsg     => InstancePath,
               SeverityLevel => Warning);

-- SIn setup & hold wrt Clk (T12, T13); does not affect state => Warning
SetupHoldCheck(TestPort      => SIn_X01,
               TestPortName  => "SIn",
               RefPort       => Clk_X01,
               RefPortName   => "Clk",
               RefEdge       => '0',
               TSetup        => tsetup_Clk_SIn(SimCondition),
               THold         => thold_Clk_SIn (SimCondition),
               CheckEnabled  => AfterReset,
               HeaderMsg     => InstancePath,
               SeverityLevel => Warning);
```

```
-----------------------------------------------------------------------------
-- Assignment of output delays.
-----------------------------------------------------------------------------

SOut        <= SOut_NoTime      after tpd_Clk_SOut(SimCondition);
Valid       <= Valid_NoTime     after tpd_Clk_Valid(SimCondition);

-- Generation of the tristate or drive of the external Data bus.
-- DOut delayed wrt the address
-- DEnable delayed, with different timing for tristating
DOutDelayed <= transport DOut    after tpd_A_D(SimCondition);
DEnDelayed  <= transport DEnable after tpd_CS_N_D(SimCondition)
                  when DEnable else
                  DEnable after tpd_CS_N_D_Z(SimCondition);
D           <= To_StdLogicVector(To_StdULogicVector(DOutDelayed, 8))
                  when DEnDelayed else
               "ZZZZZZZZ";


-----------------------------------------------------------------------------
-- Instantiation of the ExampleCore modelling the functionality
-----------------------------------------------------------------------------

ExampleCore1: ExampleCore
   port map (
      Test0    => Test0_X01,
      Clk      => Clk_X01,
      Reset_N  => Reset_N_X01,
      Mode     => Mode_X01,
      LowSpeed => LowSpeed_X01,
      Code     => Code_X01,
      A        => A_X01,
      CS_N     => CS_N_X01,
      RW_N     => RW_N_X01,
      D        => D_X01,
      DOut     => DOut,
      DEnable  => DEnable,
      Ready    => Ready_X01,
      SIn      => SIn_X01,
      SOut     => SOut_NoTime,
      Valid    => Valid_NoTime);

end BoardLevel; --======== End of Example(BoardLevel) =======================-
```

APPENDIX E: SELECTION OF SIMULATION CONDITION

In order to achieve a common interface for all VHDL models intended for Board-level simulation, the package below has been created, ensuring a similar interface for VHDL models for Board-level simulation created under ESA contracts. Work is ongoing to find a more widespread method, and it is therefore recommended to consult ESA regarding the timing interface before starting the modelling.

```
--======================================================================--
-- Design unit  : Simulation (Package declaration)
--
-- File name    : simulation.vhd
--
-- Purpose      : In this package the enumerated type SimConditionType,
--                to be used to select Worst, Typical or Best Case
--                values for timing parameters in VHDL models for
--                board-level simulation.
--
--                The simulation condition will normally be selected
--                by a generic parameter in the top-level entity
--
-- Note         : A type TimeArray has been defined, which can be used
--                for defining the timing parameters.
--
-- Errors:      : None known
--
-- Library      : ESA
--
-- Dependencies : None
--
-- Author       : Sandi Habinc, Peter Sinander
--                ESTEC Onboard Data Division (WD)
--                P.O. Box 299
--                2200 AG Noordwijk
--                The Netherlands
--
-- Simulator    : Synopsys v. 3.0c, on Sun Sparcstation 10, SunOS 4.1.3
------------------------------------------------------------------------
-- Revision list
-- Version Author Date       Changes
--
-- 1.0     PSI     1 Sep 94  New version
------------------------------------------------------------------------


package Simulation is

   -- Definition of the SimConditionType type
   type SimConditionType is (WorstCase, TypCase, BestCase);


   -- Definition of Time array type which can be used for the timing
   -- parameters
   type TimeArray is array(SimConditionType) of Time;

end Simulation;  --======= End of package Simulation =================--
```

Page intentionally left blank