



FP6-IST-507219

PROSYD

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Repository of PSL Sugar for common protocols/interfaces

(Deliverable 1.5/1)

Due date of deliverable: December 31, 2006

Actual Delivery date: December 31, 2006

Start date of project: 01.01.2004

Duration: 3 years

Organisation name of lead contractor for this deliverable: ST France

Revision: 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2000-2006)		
Dissemination Level		
PU	Public	<input checked="" type="checkbox"/>
PP	Restricted to other programme participants (including the Commission Services)	<input type="checkbox"/>
RE	Restricted to a group specified by the consortium (including the Commission Services)	<input type="checkbox"/>
CO	Confidential, only for members of the consortium (including the Commission Services)	<input type="checkbox"/>

Notices

For information, contact lyes.benalycherif@st.com

This document is intended to fulfil the obligations of the PROSYD project concerning deliverable 1.5/1, described in contract number 507219.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2004-2006. All rights reserved.

Table of Revisions

Version	Date	Description and Reason	By	Affected Sections
0.1	18.12.2006	Full document draft	Lyes Benalycherif	All
0.2	27.12.2006	Corrections according review comments	Lyes Benalycherif	All
1.0	28.12.2006	Final approval by project management	Cindy Eisner	Version number

Authors

Lyes Benalycherif
Neil Dunlop
Andrea Fedeli
Anthony McIsaac
Klaus Winkelmann

Executive Summary

Reuse is widely spread in the hardware design flows and concerns all the involved materials including properties. The goal of this task is to develop sets of PSL properties that can be reused for differing designs. These properties cover essential areas of hardware designs, protocols, FIFOs, stacks and configuration registers.

Purpose

The purpose of this document is to present the content and use model of the developed sets of PSL properties.

Intended Audience

This document is intended for designers or verification engineers who need to develop and verify hardware designs using PSL properties.

Background

The concerned sets of properties are written in standard PSL language [1].

Contents

Table of Revisions.....	iii
Authors	iii
Executive Summary	iii
Purpose	iii
Intended Audience.....	iii
Background.....	iii
List of Figures	iv
Glossary	v
1 Introduction.....	1
2 Bus protocols.....	2
Functional rule-based approach	2
Property set content	2
Use model	5
Transaction based approach.....	6
Property set content	7
Use model	9
3 FIFO	10
Property set content	10
Use model	11
4 Stack	14
Property set content	15
Use model	18
5 Configuration registers	19
Property set content	19
Use model	20
6 References.....	23
Annex A: AHB Lite protocol.....	24
Master property subset.....	24
Slave property subset.....	40
Annex B: FIFO.....	52
Annex C: Stack	59
Annex D: Configuration registers	64

List of Figures

Figure 4-1 Example Operation of a Stack	14
---	----

Glossary

AHB: The Advanced High-performance Bus, a part of the Amba hierarchy of buses specification.

Amba: The Advanced Microcontroller Bus Architecture, ©ARM Limited 1999.

APB: The Advanced Peripheral Bus, a part of the Amba hierarchy of buses specification.

Assertion: 1. A property that the DUV has to fulfil. 2. In some contexts (e.g. in dynamic verification), synonym of Property.

Assumption: A property that the DUV verification takes for granted; assumptions are properties used to model input behaviours.

Behaviour: A succession of states of a design.

Block: A component in a design. A block may contain instances of other blocks.

Design: A model of a piece of hardware, described in some hardware description language (HDL).

DUV: Design Under Verification. That is the subject of the verification process (it has to be a model for the set of properties used as assertions).

Environment: The part of a system surrounding the DUV. Often represented in an abstract form (i.e. describing only what is strictly needed to let the verification converge towards a close result, passing or failing, leaving all the other information left unspecified) it is described using the modelling layer of PSL.

FIFO: First In First Out. A component used in hardware designs for buffering and flow control which consists of a set of read and write pointers, storage and control logic.

Formal Verification: Verification based on the usage of a formal technique; in PROSYD context this is mainly used a synonym for model checking.

GDL (Generalized Description Language)

The modelling language of the IBM verification tools, RuleBase PE and FoCs. Its primary purpose is to describe the environment for formal verification. However, it is also used in conjunction with PSL to aid specification of a design. Also it is a PSL flavour.

HDL (Hardware Description Language)

One of several specialized high-level languages used by semiconductor designers to describe the features and functionality of chips and systems prior to handoff to the IC layout process. HDL descriptions are used in both the design implementation and verification flows. Currently, the two standard HDLs in use worldwide are Verilog HDL and VHDL. Several proprietary HDLs also exist, mainly for describing logic targeted for vendor-specific programmable logic devices.

ITL (InTerval Language)

A formal property language used in OneSpin's property checker. It corresponds to a subset of PSL without the unbounded * operator, see references [5], [6].

Model Checking: The automatic or almost automatic verification of a property of a model of a hardware component or in some cases of software.

Property: A collection of logical and temporal relationships between expressions involving design signals, that represents a set of behaviours.

Property Checking: 1) Syn. of Model Checking. 2) (less usual) syn. of ABV.

Protocol: A set of rules governing communication between participants in a system.

PSL (Property Specification Language)

The language for specification of designs upon which PROSYD is based. It comprises 4 different layers: Boolean, Temporal, Verification, and Modelling, used to express, respectively, propositions on events at a given time, temporal relationships among propositions related to possibly different times, properties to be proven or to be assumed for a given verification task, and environment behaviours. The Verification and Temporal layers are PSL constructs whereas the Modelling and Boolean layers are based on the syntax of the underlying HDL. Thus PSL is said to come in various flavours among which VHDL, Verilog and GDL.

RAM: Random Access Memory. Type of memory of electronic devices that need frequent updating used in computer to hold the program code and data during computation.

Specification: The process of defining the expected behaviour of a hardware design.

Verification: The process of falsifying or verifying the functional and performance requirements of a design, be it chip, board or system. Many different kinds of verification tools are in use today, including simulation, formal verification, emulation and rapid prototyping.

Verilog

One of two standardized hardware description languages used to specify the structure and behaviour of electronic systems in textual format. Also it is a PSL flavour.

VHDL (Very High Speed Integrated Circuit Hardware Description Language)

One of two standardized hardware description languages used to specify the structure and behaviour of electronic systems in textual format. Also it is a PSL flavour.

Vprop: PSL construct - verification unit type containing nothing but assert statement.

Vmode: PSL construct - verification unit type containing any modelling code and any PSL statement but assert statement.

Vunit: PSL construct - most general type of verification unit containing any PSL statement including assert statement and modelling code.

1 Introduction

The increasing System On Chip (SOC) complexity and time to market pressure has imposed design reuse at the heart of the hardware flows. As shown by the methodology document on reuse-aware property driven specification, D1.1/2 [2] the PSL properties play an important role in structuring the reuse process and rendering it effective. As part of an IP distribution, the set of PSL properties can be reused in SOC containing the IP. Furthermore, a specific design feature can be characterized by a set of standard signals and PSL properties involving these signals. These elements can serve as basis for the specification, the implementation and the verification of any IP comprising the feature in question. Among the features which concern most of hardware designs figure:

- The interfaces / protocols. The interactions between hardware components require a shared interface and follow protocol rules. Examples of hardware bus protocols at SOC level are the ST proprietary bus and the Amba AHB protocol [3].
- The data buffering components. They consist often in FIFOs but also in stacks. For a FIFO, the data are released according to their entering order in the component. Conversely, for a stack, the last data to be added to the component is the first one to be removed.
- The configuration registers. They constitute within a design a programmable area dedicated to storing information related to the design operating modes.

We developed for each of these features, a set of PSL properties associated to a standard set of signals with the purpose to be formally verified.

Section 2 is about bus protocols following two approaches, the transaction-based one promoted by OneSpin Solutions and the functional rule-based approach used by ST. Both were applied to AHB. This section presents for each approach, the set of resulting properties. Section 3 is dedicated to the FIFO property set. Then follows the stack one in section 4 and the configuration registers property set in section 5. Each section addresses first the property set content and then the use model. The document ends with four annexes each containing the PSL code of the concerned property sets.

2 Bus protocols

This section deals with sets of predefined properties covering bus protocols. Two development approaches can be followed resulting in two differing property set organizations, the functional rule-based approach used by ST and the transaction-based one initiated by OneSpin Solutions. The Amba AHB protocol was treated according to both approaches and the developed sets of predefined properties are reported hereafter.

Functional rule-based approach

The protocol specification defines both the structure and the semantics of the protocol signaling information and its underlying transactions. It also describes the protocol rules in the form of behavioral relationships between the signals required to vehicle the transactions. The functional rule-based approach consists to proceed in a systematic way and identify in the protocol specification document the statements or subsets of statements covering these relationships. Each identified rule is then captured in the form a PSL property. The set of properties constitute an alternative formulation of the unformal protocol specification with a clear mapping between the PSL properties and the rules described in the specification document. Prior to PROSYD, this approach was followed by ST to develop the PSL property specification of the ST proprietary bus protocol. The approach was then applied within the framework of PROSYD to the Amba AHB protocol considering its Lite version [4] and focusing on the core part of the protocol. The arbitration related aspects are out of this scope. The Lite version is intended for AHB systems where only one master is used. This excludes from the full AHB specification the request/grant handshake signals to the arbiter and the split/retry responses from the slave.

Property set content

The AHB Lite protocol is articulated according to master and slave roles hence two property subsets. Four groups of properties can be distinguished for the master subset:

- legal transitions for the transfer type
- signal stability requirements during wait states
- well-formedness requirements of the burst transactions
- structural requirements (involving the transfer size, the data bus width and the addresses)

and three groups for the slave:

- response requirements in reaction to the master (in presence of busy or idle transfers)
- response requirements at the initiative of the slave (in case of non okay responses)
- boundedness requirement for consecutive wait states

For the sake of clarity, the AHB signals are denoted in upper case in the following.

Legal transitions for the transfer type

- When HBURST is single and HTRANS is nonseq, next value for HTRANS can only be idle or nonseq
- When HTRANS is idle, it can only take idle or nonseq value
- After a busy transfer, HTRANS can only take busy or seq value
- After the first transfer of a burst, HTRANS can only take busy or seq value

Signal stability requirements during wait states

- The master must keep HWDATA constant until HREADY is sampled high.
- When HTRANS is nonseq or seq, the master must keep HTRANS constant until HREADY high unless HRESP is not okay
- When HTRANS is nonseq or seq, the master must keep HTRANS constant until HREADY high unless HRESP is not okay
- When HTRANS is not idle, the master must keep HADDR constant until HREADY high unless HRESP is not okay
- When HTRANS is non or seq, the master must keep HWRITE constant until HREADY high unless HRESP is not okay
- When HTRANS is non or seq, the master must keep HBURST constant until HREADY high unless HRESP is not okay
- When HTRANS is non or seq, the master must keep HSIZE constant until HREADY high unless HRESP is not okay
- When HTRANS is non or seq, the master must keep HPROT constant until HREADY high unless HRESP is not okay

Well-formedness requirements of the burst transactions

- The number of beats is as expected for single transfers unless response not okay
- The number of beats is as expected for 2 beat burst transfers unless response not okay
- The number of beats is as expected for 4 beat burst transfers unless response not okay

- The number of beats is as expected for 8 beat burst transfers unless response not okay
- The number of beats is as expected for 16 beats bursts transfers unless response not okay
- HWRITE must be kept constant by the master along a burst unless HRESP is not okay
- HSIZE must be kept constant by the master along a burst unless HRESP is not okay
- HBURST must be kept constant by the master along a burst unless HRESP is not okay
- HPROT must be kept constant by the master along a burst unless HRESP is not okay
- HADDR must be kept constant by the master when HTRANS is busy, not at end of burst unless HRESP is not okay
- The number of consecutive cycles when HTRANS is busy is bounded
- Bursts must not cross 1KB address boundary
- Beats addresses are correctly incremented in incrementing bursts
- Beats addresses are correctly incremented in wrapping bursts

Structural requirements

- All transfers must be aligned to the address boundary equal to the size of transfer
- The master must never attempt a transfer of which size is wider than the data bus size

Response requirements in reaction to the master

- Slave must always provide a zero wait state okay response to idle transfers
- Slave must always provide a zero wait state okay response to busy transfers

Response requirements at the initiative of the slave

- A non okay response requires two cycles

Boundedness requirement for consecutive wait states

- Every slave must have a predetermined maximum number of wait states

Use model

These properties were initially coded in PSL Verilog flavour. The first checking of this code involved a beta version of Rulebase for what concerns the Verilog flavour support. In order to permit further exploitations of these properties in more flexible conditions, it was decided at that time to convert the code in question to GDL flavour. The properties in PSL GDL flavour can be found in Annex A under the form of two self-sufficient subsets a master one and a slave one. These subsets can be reused to check the protocol compliance of any hardware design with an AHB Lite interface. All properties are safety ones. Each subset first contains the properties that are required to hold under the assumption that the environment of the concerned design has a legal behaviour in terms of protocol. Therefore the subset contains in addition all the other side properties turned into assumptions.

How to relate the properties to the design?

Any particular design model will have its own names for the signals involved in an AHB Lite interface either master or slave. The way to relate these signals to the appropriate predefined properties is as follows.

Each subset is expressed in terms of variables such as `master_htrans` and macros such as `data_bus_size`. Users of the subset do not touch the `vmodes`, `vprops` and `vunits` containing the properties and environment assumption expressed in terms of these variables and macros. However, there is a definition `vmode` (“`ahb_master_definitions`” for the master subset and “`ahb_slave_definitions`” for the slave subset) consisting of definitions of these variables and macros, and the user can edit the right-hand side of these definitions, by entering the appropriate signal name and parameter value, for example

```
define master_htrans := HTRANS;
```

The `vmode` containing these definitions is inherited by all the `vunits` containing the properties.

The unedited definitions `vmode` supplied with the library gives default values to each variable, for example

```
define master_htrans := ones(2);
```

Thus the property subset does in fact consist of legal PSL, and the `vunits` can be loaded, and indeed run, by a verification tool. However, the results are not of any significance until the editing has been done.

The user has to edit the definition `vmode` of the considered subset i.e. either “`ahb_master_definitions`” or “`ahb_slave_definitions`”; the whole subset is then ready for use.

How to use the properties?

Only one AHB interface of a given design can be considered at a time to be formally verified.

The slave and master subsets share the same structure in a symmetrical way which gives for the master (resp. slave):

- the `vmode` “`ahb_constants`” containing all the AHB encoding values
- the `vmode` “`ahb_master_additional_variables`” (resp. “`ahb_slave_additional_variables`”) containing the variables that correspond

to expressions frequently used in the properties and directly based on AHB interface signals

- the definition vmode “ahb_master_definitions” (resp. “ahb_slave_definitions”)
- the slave (resp. master) assumption vmodes each of which being a slave (resp. master) property turned into an assumption
- the vmode “ahb_slave_legal_behaviour” (resp. “ahb_master_legal_behaviour”) inheriting all the slave (resp. master) assumption vmodes
- as many couple of vprop and vunit as master (resp. slave) properties with
 - the vprop containing the property itself
 - the vunit inheriting the vprop, the vmodes “ahb_slave_legal_behaviour” (resp. “ahb_master_legal_behaviour”), “ahb_constants”, “ahb_master_additional_variables” (resp. “ahb_slave_additional_variables”)

All the default definitions in the definition vmodes are required to be edited in order to bind the predefined property subset to the hardware design of which the AHB Lite interface is under check. This editing has to be done as follows.

- Replace the default definitions of the data bus width, upper bound number of consecutive wait states, upper bound number of consecutive busy states macros with appropriate values corresponding to the considered design
- Replace the default definitions of all the AHB interface signal variables with actual signals names of the considered design

These configurations steps fully cover the AHB Lite interface under check but do not take into account the environment assumptions that are related to interface signals of the considered design, not part of the AHB Lite interface in question. As they are specific to each considered design, these additional assumptions have to be explicitly formulated by the user before running the AHB property checks.

This AHB property set was integrated to the ST flow which already comprised the ST bus protocol property set. It contributes to a homogeneous protocol compliance static checking solution for any IP HDL implementation with an interface of either protocol.

Transaction based approach

Rather than on rules and requirements, the starting point in this approach is given by the various transactions defined in the protocol, and the goal is to capture the full specification of the protocol in properties by formalizing the conditions and effects of each transaction.

This work was performed at OneSpin, starting with properties in the ITL language. A full property set was developed, applied to one module, and reused for a different module. This experience is reported below.

The initial goal of delivering reusable PSL code was abandoned, for two reasons: First, OneSpin’s business strategy puts less emphasis on PSL support than initially expected, and therefore first experiments on re-use were based on ITL (OneSpin’s own property language)[5][6] rather than PSL. Second, it turned out that re-use of properties requires additional adaptation effort, as explained in the following section – and this effort was beyond the available project resources.

Property set content

A set of properties covering all aspects of the AHB protocol was developed, using OneSpin’s ITL language. We describe here the general approach taken, and the problems encountered in this experiment.

The AHB protocol defines

- certain operations such as single transfer, incr4, incr8, retry,
- certain generic signals such as HREADY, HTRANS, HGRANT, HWDATA ...
- certain states allowing each operation
- the behaviour of these signals for each operation

The core idea of the transaction-based approach is that each such operation of the protocol is translated to one ITL property, according to the following steps:

- Some general assumptions are made regarding the protocol compliance of the module’s environment, such as other modules on the same AHB bus, e.g.
 - there is no reset during a transaction
 - The HREADY signal defines the sequence of phases according to the AHB specification
- Additional knowledge about the reachable state space is specified and proven as Invariant, and then used to help the individual proofs.
- Each property is of the type
 ALWAYS ((StartingState and TriggerCondition)
 ->
 ((TransactionBehaviour1 and NextState1) or
 (TransactionBehaviour2 and NextState2) or
 ...
 (TransactionBehaviour_n and NextState_n)).
- Here the TransactionBehaviours correspond to various transfer types identified by the HTRANS signal.

This is illustrated by the following example, which describes, what can happen when the module is in the “START_SINGLE” state: this can be either an idle transaction, or any of SINGLE, INCR4, INCR8, INCR16 or INCR as specified in the AHB protocol.

```
theorem start_single_state2next_okay;
for timepoints:
  ta = t + 0,
  td = ta + 1..MAX_WAIT_CYCLE_NUMBER;
```

```

freeze:
  new_data = HWDATA @ ta+1;
assume:
  START_SINGLE_STATE(ta);
  INPUT_CONSTRAINT;
  INVARIANT;
  HREADY_BEHAVIOUR(ta, td);
  during [t_first, t_last] : HRESET = '1';
  at ta : HGRANT= '1';
  during [ta+1, td] : HRESP = OKAY;
prove:
  either
    LAST_BEAT_BEHAVIOUR(ta, td, new_data);
    IDLE_BEHAVIOUR(ta, td);
    IDLE_STATE(td);
  or
    LAST_BEAT_BEHAVIOUR(ta, td, new_data);
    START_BURST_BEHAVIOUR(ta, td, SINGLE);
    START_SINGLE_STATE(td);
  or
    LAST_BEAT_BEHAVIOUR(ta, td, new_data);
    START_BURST_BEHAVIOUR(ta, td, INCR4);
    START_INCR4_STATE(td);
  or
    LAST_BEAT_BEHAVIOUR(ta, td, new_data);
    START_BURST_BEHAVIOUR(ta, td, INCR8);
    START_INCR8_STATE(td);
  or
    LAST_BEAT_BEHAVIOUR(ta, td, new_data);
    START_BURST_BEHAVIOUR(ta, td, INCR16);
    START_INCR16_STATE(td);
  or
    LAST_BEAT_BEHAVIOUR(ta, td, new_data);
    START_BURST_BEHAVIOUR(ta, td, INCR);
    START_INCR_STATE(td);
  end either;
end theorem;

```

All symbols in capital letters refer to macros which refine the high-level concept (operation, state). As an example, consider the behaviour of the HREADY signal:

```

HREADY_BEHAVIOUR(t_address : timepoint;
                 t_data      : timepoint) : assertion :=
at      t_address      : HREADY = '1';
during [t_address+1, t_data-1] : HREADY = '0';
at      t_data         : HREADY = '1';
end HREADY_BEHAVIOUR;

```

While this example only refers to a signal of the AHB specification, ultimately the behaviour needs to be mapped to a given implementation, such as in the following example:

```

IDLE_STATE(t0 : timepoint) : assertion :=
at t0: rBurst4Count_x = 0;
at t0: rlose_bus_true = 0;
at t0:   r_masterBusReq   = 0;
at t0:   r_masterTrans   = AHB_HTRANS_IDLE;
at t0:   r_masterBurst   = AHB_HBURST_INCR;
at t0:   r_masterWrite   = 0;
at t0:   r_masterWrByteVal = 0;
at t0:   rAddrPhase     = 0;
at t0:   rDataPhase     = 0;
at t0:   rRetryPending  = 0;
at t0:   rBusDmaErrResp = 0;
at t0:   rTransferDone  = 0;
at t0:   rDoneOnRetry   = 0;

```

```
at t0:      rBusDmaDone    = 0;
end IDLE_STATE;
```

Use model

This section explains what is required to reuse the transaction-based AHB properties. As the above example shows, the bodies of some macros are implementation-dependent.

In addition, as the `start_single_state2next_okay` property shows, INVARIANTs and INPUT_CONSTRAINTs need to be taken into account in order to prove a property. To summarize, re-use of this property set requires customizing it to a given implementation by:

- mapping the generic signal names
- mapping the generic states
- specifying input constraints
- finding design invariants.

OneSpin has used the concept outlined here to verify two different implementations of an AHB interface. In each case, the effort for the customizing was between 6 and 8 person weeks (work done outside PROSYD). This was considered too much for a routine application. Further research is required to automate some of the customizing process sketched above.

Comparison between approaches

The transaction-based approach and the rule-based one represent two different ways to structure the protocol rules. The latter follows the definition items of the protocol functional specification whereas the former unfolds the protocol starting from its underlying transactions. Both approaches rely on a systematic way to achieve 100% coverage of the protocol. The OneSpin Solutions tools suite provides a concrete framework to meet this objective.

In terms of dependency on the considered design, the rule-based approach only involves interface signals whereas the transaction-based one may require further internal consideration of the design. This results in more reuse flexibility for the former approach.

3 FIFO

Property set content

The set of properties consists of two groups: one containing the essential properties characterizing any FIFO, and one containing additional properties that may or may not be requirements for a particular FIFO.

Essential properties

There are four essential properties of FIFOs:

- It must be possible for any number of writes to be made to the FIFO.
- At no time can there have been more reads than writes;
- If there have been more writes than reads, it must be possible for there to be further reads.
- For each positive integer n , the data value at the n -th read is equal to the data value at the n -th write.

There is a separate PSL vunit for each of these properties. The second and third properties ensure that there is a one-to-one correspondence between reads and writes; and then the fourth property ensures that the data values at any read is equal to the data value at the corresponding write. Note that we do not include an explicit property that there are no writes to the FIFO when it is full – indeed, the size of the FIFO does not need to be known. As long as the four properties above are satisfied, the FIFO will be operating correctly. If a write occurs when the FIFO is full, then one of these properties will be violated; for example, the data may be lost, or it may overwrite previous data, so that the fourth property is violated.

Additional properties

The additional properties fall into three categories

- Protocol requirements of the specific environment of the FIFO (e.g. the FIFO may not grant a write unless the environment is requesting to write). Such requirements are specific to the particular blocks the FIFO has interfaces with, and will vary from case to case.
- Performance (e.g. latency between a write and the data becoming available for reading). In general, a FIFO can function correctly without providing any guarantee of minimum latencies; but such guarantees may be requirements in particular cases.

- Status reporting (e.g. indicating whether the FIFO is full). A FIFO may or may not have outputs that indicate something about its status, but if it does have such outputs, their values should be correct.

Use model

The property set code in PSL GDL flavour can be found in Annex B.

How to relate the properties to the design?

Any particular design model will have its own names for the interface signals that are relevant to the properties. The way to relate these signals to the library properties is as follows.

The properties are expressed in terms of variables such as `data_in` and of macros such as `data_width`. Users of the library do not touch the vunits containing the properties expressed in terms of these variables. However, there is a `vmode` consisting of definitions of these variables and macros, and the user can edit the right-hand side of these definitions, by entering the appropriate signal name and parameter value, for example

```
define data_in := write_data;
```

The `vmode` containing these definitions is inherited by all the vunits containing the properties.

The unedited definitions `vmode` supplied with the library gives default values to each variable, for example

```
define data_in := 0;
```

Thus the library does in fact consist of legal PSL, and the vunits can be loaded, and indeed run, by a verification tool. However, the results are not of any significance until the editing has been done.

Some variables represent not just signals, but events or states. For example, there is a variable `write_event` representing the event of a write to the FIFO; and there are variables `fifo_ready_for_write`, representing the fact that the FIFO is in a position to accept a write, and `environment_ready_to_write`, representing the fact that the environment is in a position to make a write. In these cases, the definitions of the variables may be expressions rather than just signals. The possible choices for such expressions cover a variety of ways that data can enter and leave the FIFO:

- There may be different clocks for reads and writes
- It may be either the FIFO or the environment that takes the initiative for reads or for writes; and the side that is not taking the initiative may either indicate willingness to participate as soon as it is ready, or only in response to the initiative, or it may be obliged to participate when the other side takes the initiative.

In all cases, a write event occurs when the variables `fifo_ready_for_write` and `environment_ready_to_write` are both 1 at a tick of the write clock, and similarly for reads.

How to use the property set?

The properties are contained in a number of vunits, which use variables that can be defined in terms of signal names and parameter values. The user should edit the two vmodes “fifo_definitions” and “fifo_optional_definitions”; the whole set of vmodes and vunits is then ready for use.

The default definitions in the vmode “fifo_definitions” should be edited as follows.

- If the FIFO size could be greater than 100, replace the default value of 100 for `fifo_upper_bound` by some number that you are confident is greater than the actual maximum size of the FIFO. This upper bound does not appear directly in the properties; it is used to keep the range of certain other variables finite, and it does not have to be the actual FIFO size.
- Replace the default definitions of the data width macro, and the `data_in` and `data_out` variables, by the correct ones in terms of the actual design signals.
- Replace the definitions of `write_clock` and `read_clock` by expressions in the actual clock signals; for example, define `write_clock := rose(clk_bus)` if writes are clocked on the rising edge of `clk_bus`. If writes or reads are asynchronous, leave the definition of `write_clock` or `read_clock` as 1.
- Provide definitions of `fifo_ready_for_write` and `environment_ready_to_write`, as follows.
 - If the environment makes a write by asserting some signal `w`, and the FIFO has no signal to control whether the write is made, define `environment_ready_to_write := w` and `fifo_ready_for_write := 1`;
 - If the environment has a signal `w` as above, but the FIFO has a signal `g` such that the write can only be made if `g` is asserted, then:
 - If the environment is allowed to wait for `g` to be asserted before asserting `w`, define `environment_ready_to_write := (g -> w)`; otherwise define `environment_ready_to_write := w`.
 - If the FIFO is allowed to wait for `w` to be asserted before asserting `g`, define `fifo_ready_for_write := (w -> g)`; otherwise define `fifo_ready_for_write := g`.
- Provide definitions of `fifo_ready_for_read` and `environment_ready_to_read` in a similar way.

The default definitions in the vmode “fifo_optional_definitions” should be edited as follows. If the relevant properties do not apply to the particular FIFO, the default definition can be left unchanged.

- If there is a requirement about the latency between the FIFO not being full and the FIFO accepting writes, or if there is a status output indicating that the FIFO is full, the exact FIFO size needs to be known. Define `exact_fifo_size` appropriately. It should be the maximum number of entries that can have been written but not yet read.
- If there are protocol requirements on the signals for grants of reads and writes, define the variables `fifo_grant_write`, `fifo_grant_read`, `environment_request_write` and `environment_request_read` to be equal to the appropriate signals.
- If there are requirements on the latencies between when the FIFO is not empty and when data becomes ready for reading, or between when the FIFO is not full and when the FIFO becomes ready for writing, define the `read_latency` and `write_latency` variables appropriately. The read latency should be expressed as a number of read clock cycles, the write latency as a number of write clock cycles.

- If there are signals indicating that the FIFO is full, or empty, define the variables `fifo_full` or `fifo_empty` to be equal to these signals. By default, the `fifo_full` signal is clocked on ticks of the write clock, and the `fifo_empty` signal on ticks of the read clock. Change these definitions if necessary.

There are now four vunits, in PSL GDL flavour, containing properties that should be checked for every FIFO. These are

- `fifo_write_liveness`: the assertion fails if it's possible for the FIFO to get in a state where it refuses to accept any further writes, even though the environment is willing to read from it.
- `fifo_read_liveness`: the assertion fails if there have been more writes to the FIFO than reads, but the FIFO refuses to accept any further reads.
- `fifo_no_read_when_empty`: the assertion fails if there have been the same number of reads as writes, but the FIFO is willing to accept a further read.
- `fifo_data_correspondence`: the assertion fails if the data associated with some read is different from the data associated with the corresponding write (where the correspondence is given by the order of the writes and reads).

There are also several vunits containing additional properties that should only be checked if they apply to the particular case:

- `fifo_write_grant_only_on_request`: the assertion fails if the grant signal for writes is asserted by the FIFO when the environment is not requesting to write.
- `fifo_read_grant_only_on_request`: the assertion fails if the grant signal for reads is asserted by the FIFO when the environment is not requesting to read.
- `fifo_not_read_and_write`: the assertion fails if the FIFO is asserting its grant signal for both reads and writes simultaneously.
- `fifo_read_latency`: the assertion fails if the interval between a time when the FIFO is not empty and the next time at which data can be read is greater than the specified latency.
- `fifo_write_latency`: the assertion fails if the interval between a time when the FIFO is not full and the next time at which data can be written is greater than the specified latency.
- `fifo_full`: the assertion fails if either the FIFO is full and the `fifo_full` signal is not asserted, or the FIFO is not full and the `fifo_full` signal is asserted.
- `fifo_empty`: the assertion fails if either the FIFO is empty and the `fifo_empty` signal is not asserted, or the FIFO is not empty and the `fifo_empty` signal is asserted.

These configurations steps fully cover the FIFO under check but do not take into account the environment assumptions that are related to interface signals of the considered design, not directly involved in the FIFO properties. As they are specific to each considered design, these additional assumptions have to be explicitly formulated by the user before running the FIFO property checks.

4 Stack

This chapter gives reusable properties for verifying a stack design.

A stack is a contiguous block of memory that contains data and an associated stack pointer to indicate the top of the stack (with the bottom of the stack at location 0). Data can be *pushed* onto the top of the stack or *popped* off, as illustrated in the figure below.

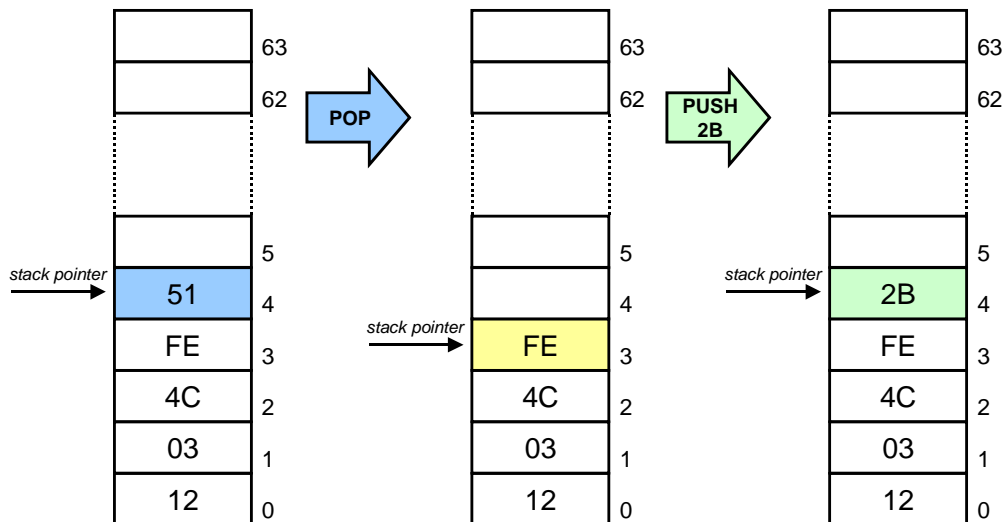


Figure 4-1 Example Operation of a Stack

The properties presented here have been verified on several different implementations of a stack, each one containing a RAM given by a model suitable for property checking. Many of the properties can be applied to each implementation as they only differ in a border case, naming what happens when there is a simultaneous push and pop request.

Border cases often contain errors and so should always be verified. Other border cases to consider are what happens when the stack is empty or full, can an overflow or underflow bug occur? A simple mistake to make when creating a stack is to simply assert the empty flag when the stack pointer is zero, but the stack pointer can also be zero when the stack contains just one filled cell.

Property set content

The basic properties for a stack are these:

- Reset Proof
- No-Operation Proof (stack unchanged)
- Read (Pop) (data taken off the top of the stack)
- Write (Push) (data written onto the top of the stack)

Corner case and multiple-timestep properties are then described in the following order:

- Stack Full
- Stack Empty
- Simultaneous Read + Write Proofs
- Write then Read-back Proofs

Each of these is briefly explained below, and the corresponding PSL code in vhdl flavour is given in annex C.

Reset Proof

Following a reset, a stack should be empty with the stack pointer at zero. (see property: stack-rst)

Remarks:

- This is a synchronous reset, hence the reset takes effect at time $t+1$. However, if the standard clocking scheme is applied, the same property would hold for an asynchronous reset. In this case the property could be strengthened to prove the reset state at time t and $t+1$.
- The memory and data output are not checked as it is implemented with a RAM that is not initialized following a reset. However, the design could be modified such that the reset will disable the chip-select for one cycle (which will set the data output to a default value).

No-Operation Proof

When a stack has no push or pop requests, the stack pointer and the stack memory should be unchanged. (see property: stack-nop)

A macro is used for the commitment so that it can be re-used for other properties. This macro is itself described in terms of other macros, as shown below.

Read (Pop)

When a stack is popped, the memory location at the stack pointer is read and the stack pointer is then decremented. A pop is only allowed if the stack is not empty. (see property: stack-r)

Remarks

- A macro is used for the commitment so that it can be re-used for other properties.
- An invariant macro (`s_ptr_range`) is used to avoid an invalid debug sequence in which the stack pointer starts above its maximum. This is proven separately in vunits `stack_invariants_Reset` and `stack_invariants_Induct`.

Write (Push)

When a stack is pushed, its stack pointer is incremented and data is written to the memory location at the address of the new stack pointer (i.e. one higher). A push is only allowed if the stack is not full. (see property: `stack-w`)

Remarks:

- A macro is used for the commitment so that it can be re-used for other properties.
- Two invariant macros (`s_ptr_range` and `sp_zero_if_empty`) are used to avoid invalid debug sequences. These are proven separately in vunits `stack_invariants_Reset` and `stack_invariants_Induct`.
- This property is not quite complete, it does not verify that all other memory locations remain unchanged. This is proven with the separate property `stack-serve_ram`.

Efficient Memory Conservation

A further property is required to prove the write operation. What's lacking from the `stack-w` property is a check that aside from the one written cell, the rest of the memory is unchanged. The `stack-serve_ram` property states that a memory location can only change if it is written to.

Remarks:

- For simultaneous push and pop the pushed data may bypass the RAM. Therefore the RAM is allowed to keep its value after push.
- Data are correctly written to the RAM if necessary, as proven with the push and simultaneous push and pop properties.
- This property is generic because the maximum stack address in the for statement is expressed by a generic parameter.

Stack Full

When the stack is full a write cannot be performed but a read or NOP can occur. Due to the use of macros for the read and NOP properties, this property is concise. (see property: `stack-full`).

Remarks:

- The push request is don't-care when the stack is full.
- This property also checks that empty and full can never both be asserted.
- Two arguments to the `stable_after_nop` macro are constant, they are empty and full respectively.

Stack Empty

When the stack is empty a read cannot be performed but a write or NOP can occur. Due to the use of macros for the write and NOP properties, this property is concise. (see property: stack-empty)

Remarks:

- The pop request is don't-care when the stack is empty.
- This property also checks that empty and full can never both be asserted.
- Two arguments to the commitment macros are the constants empty and full.

Simultaneous Read + Write Proofs

A border case to consider is what happens when there is a simultaneous push and pop. This is a distinguishing factor and can affect the design of the stack itself, resulting in different architectures as discussed below.

Alternative Stack Designs

1 push_overrides

This is a simple priority scheme, with the push serviced (unless the stack is full). (see property: stack-rw_push_overrides)

- Priority is on push unless the stack is full.

2 full_bypass

This stack can completely bypass the memory and route the push data to service the pop request in the following cycle. It requires an input register that always stores the incoming data and provide the one-cycle delay that a pop request would expect. Although simultaneous, the push is effectively serviced just prior to the pop. This stack ignores the full and empty flags (as the memory is not used). (see property: stack-rw_full_bypass)

- For this property, the stack memory is bypassed and the input data is output on the next cycle i.e. the push data services the pop.

As the RAM is unchanged, the full/empty signals don't have to affect the outcome.

3 safe_bypass

This stack is the same as full_bypass, but is sensitive to the full/empty flags. Separate properties are required for each design, as given below. (see property: stack-rw_safe_bypass)

- When the safe_bypass stack has a simultaneous push and pop the stack is normally bypassed, i.e. memory unchanged and the input data is output on the next cycle. However, if the full or empty flags are asserted then there is no bypass. This behaviour is potentially safer because the interfaces to the stack may be relying on such a response.
- The commitment for this property has a case analysis on the full/empty flags. The stack is only bypassed if the stack is neither full nor empty.

Write then Read-back Proofs

An interesting check is that if you push data onto the stack in one cycle and then read-back from the same location in a later cycle, then the same data will be output

(as long as there was no write or read in between). The later cycle can either be the very next cycle or could occur after a number of NOPs in between.

These checks play the role of sanity checks. The properties above prove the stack completely. (see property: stack-w_rb)

The above property can be extended by allowing one or more NOPs in between. This is checked with a * operator in PSL. (see property: stack-w_rb_by3)

Use model

How to relate the properties to the design?

Any particular design model will have its own names for the signals involved in a stack. The way to relate these signals to the appropriate predefined properties is as follows.

Users of the subset do not touch the vunits containing the properties and environment assumption expressed in terms of these variables. However, there is a definition vmode (stack_definitions) consisting of definitions of the variables:

- interface signals push, pop, d_in, rst, empty, full
- Memory array memory
- Stack pointer s_ptr

The user can edit the right-hand side of these definitions, by entering the appropriate signal name and parameter value, for example

```
define push := xyz_push;
```

The vmode containing these definitions is inherited by all the vunits containing the properties.

How to use the properties?

All the default definitions in the definition vmode are required to be edited as described above.

Furthermore, note that PSL does not define on which set of traces the property is to be checked, and the following choice was made here: the checker should consider an arbitrary starting state, i.e. not necessarily start from reset. Otherwise the properties as written here would not detect any failure at later cycles, because no explicit “always” is stated. When using a checker starting from reset, the properties should be modified by wrapping them with an “always” operator.

5 Configuration registers

Most HDL designs present a block that handles a set of configuration registers which affect the operation of the design. The registers are usually dedicated a specific interface to be accessed in read or mode write. They can also be updated by other external input signals. The handling and management of these registers obey to common rules captured by the set of properties described hereafter.

Property set content

The properties are of two types either related to the register functionality or to the protocol requirements of the main interface through which the registers can be read and written. A section by section property description follows.

Functional properties

- Reads return most recently written value.

For each register, it is expected that reading it will always return its most recently written value. If no writes occurred since reset, its reset value is expected to be returned. Any external input signals that affect the value of a register will need to be taken into account in the most recent written byte description for the register.

- Reads from write-only registers always return 0.

Any write-only register will never yield its data value through register reads; any attempt to do so will always return 0.

- Writes affect output signals.

For those registers that are writeable and affect output signals, the appropriate property can be used to cover these cases. This can include write-only registers. Whether the new value of the output signal appears in the same cycle as the register write being completed, or in the following cycle, this will need to be reflected appropriately in the property by using the suffix implication or the next suffix implication PSL operator.

- Changes in output signals solely due to writes.

This is the reverse of the previous property and ensures that there are no other possible events that cause the output signals to change. Whether there is one cycle delay between the register write and the change in the output signal or no delay, this will need the use in the property of the prev PSL built-in function or not.

- Output signals derived from current register data and inputs.
If some input signals can also affect output signals (for example interrupts), then this property will need to be used and adapted for precise description of the effects.

Protocol requirements of the register interface

Such requirements are specific to the main interface through which the registers are accessed.

The functional properties involve information that vary from a design to another and that are specifically provided by the functional specification document of the considered design:

- main interface for register access
- list of external inputs that can update the registers (concerned registers and involved values)
- list of the used registers, along with their addresses, whether they are read-only, read/write or write-only, their bit-width and their value on reset
- list of output signals affected by writes to registers together with the way they are influenced

Use model

The property set is based on the following interface framework defined by:

- two handshake signals, input signal req and output signal r_req
- two data busses, input data and output r_data
- an input address bus addr
- a status signal, output signal error

The majority of register access interfaces are structured according to this framework. Such is the case for the ST bus type 1 target interface and the Amba APB slave interface. If a register access interface is not compliant with this framework, the property set needs to be adapted accordingly but the required effort is expected to be weak as the distance between the structures should be small.

The property set code in PSL GDL flavour can be found in Annex D.

How to relate the properties to the design?

Any particular design model will have its own names for the interface signals. The way to relate these signals to the predefined properties is as follows.

The properties set is expressed in terms of variables such as register_req and macros such as data_bus_size. These variables and macros are defined in two definition vmodes “register_interface_definitions” and “register_additional_definitions”. The former is dedicated to the interface signals and the design parameters whereas the

latter captures the register names and addresses as well as the validity and effectiveness conditions of register use (write, read, valid operation, valid address and byte selection conditions).

The user can edit the right-hand side of these definitions, by entering the appropriate signal name and parameter value, for example

```
define register_req := REQ;
```

The vmodes containing these definitions is inherited by all the vunits containing the properties.

The unedited definition vmodes supplied with the library gives default values to each variable, for example

```
define register_req := 1;
```

Thus the property subset does in fact consist of legal PSL, and the vunits can be loaded, and indeed run, by a verification tool. However, the results are not of any significance until this editing has been done.

The user has not only to edit the right-hand side of the given definitions but also to reproduce a certain number of these definitions. For example, the definition of the macro `reg_i_addr` (capturing the register name and its address) has to be replicated as many times as there are used registers listed in the functional specification document of the considered design.

How to use the properties?

The property set comprises:

- the vmodes “register_interface_definitions” and “register_additional_definitions”
- the vmodes covering protocol assumptions
- the vunits dedicated to the protocol requirements and the functional properties

The two definition vmodes are required to be edited in order to bind the predefined property set to the considered hardware design. This editing has to be done as follows.

- Replace the default definitions of the macros `address_upper_bit`, `address_lower_bit`, `address_width`, `data_bus_size`, `data_upper_bit` with appropriate values corresponding to the considered design.
- Replace the default definitions of the interface signal variables with actual signals names of the considered design.
- Replace the default definitions of the variables `write`, `read`, `valid_operation` with appropriate boolean conditions involving actual design interface signals.
- Replace the default definitions of the macro `reg_i_addr` and the variable `byte_j`. Then reproduce them according to the list of used registers and the number of bytes allowed by the data bus width.
- Replace the default definition of the variable `valid_address` condition with the appropriate boolean condition involving the macros `reg_i_addr`.

The supplied vmodes and vunits related to the protocol requirements do not need to be edited by the user whereas all the other vunits are subject to being edited.

Any non protocol vunit refers to a given register and requires to be replicated for each register when it is applicable for the register. The replication must also be made w.r.t. each byte or bit of the data bus when it is applicable for the vunit and the register. The vunits that relate register to output signal or input signal have as well to be replicated for each concerned output or input signal.

The vunits with modelling code keeping track of the most recent written byte need to be edited if the concerned register is affected by external input signals. Any such signals have to be taken into account in this modelling code.

The non protocol vunits are supplied in two ways uncommented and commented. The uncommented vunits are valid PSL that can be run without editing but not in a significant way. They may require editing w.r.t. the external input signals as mentioned in the previous paragraph.

The commented vunits are based on specific information for the considered design. Example of this information is the number and the names of the input or output signals in the case of the properties which relate these signals to the registers. When they are applicable for the considered design, these vunits need to be uncommented and edited. The code of these vunits has to be updated with the actual names of the concerned signals. The assertion statements themselves need to be updated to precisely reflect the relationships between the concerned signals and the registers. Such is the case for the vunits “writes_to_reg_i_byte_j_bit_k_affect_outsig1” and “outsig3_from_reg_i_and_insig”.

These configurations steps cover the whole logic perimeter of the configuration registers under check but do not take into account the environment assumptions that are related to interface signals of the considered design, not directly part of the perimeter in question. As they are specific to each considered design, these additional assumptions have to be explicitly formulated by the user before running the configuration register property checks.

6 References

- [1] IEEE Standard for Property Specification Language (PSL). IEEE Std 1850™-2005.

- [2] M. Farkash, A. Fedeli, L. Gluhovsky, A. Maggiore, A. McIsaac and V. Preis, Reuse-Aware Property Specification, December 2004. Prosyd D1.1/2.

- [3] Amba™ Specification (Rev 2.0), IHI001A, ARM Limited 1999.

- [4] AHB-Lite Overview, DVI0044A, ARM Limited 2001.

- [5] J. Bormann, C. Spalinger, Formal Verification for Non-Formalists (in German), IT+TI, 1/2001, Oldenbourg.

- [6] J. Bormann, C. Blank, K. Winkelmann, Technical and Managerial Data About Property Checking With Complete Functional Coverage, EuroDesignCon 2005.

Annex A: AHB Lite protocol

Master property subset

```
vmode ahb_constants {

#define         idle           00B
#define         busy           01B
#define         nonseq        10B
#define         seq           11B

#define         single        000B
#define         incr          001B
#define         wrap4        010B
#define         incr4        011B
#define         wrap8        100B
#define         incr8        101B
#define         wrap16       110B
#define         incr16       111B

#define         bits8         000B
#define         bits16        001B
#define         bits32        010B
#define         bits64        011B
#define         bits128       100B
#define         bits256       101B
#define         bits512       110B
#define         bits1024      111B

#define         okay          00B
#define         error         01B
#define         retry         10B
#define         split         11B

}

vmode ahb_master_definitions {

#define data_bus_size 32
#define wait_states_bound 15
#define busy_bound 15
-- Replace these numbers by appropriate values

#define upper_data_bit (data_bus_size - 1)

define master_hclk := 1;
define master_htrans(1..0) := ones(2);
define master_hready := 1;
```

```

define master_hresp(1..0) := ones(2);
define master_hburst(2..0) := ones(3);
define master_hsize(2..0) := ones(3);
define master_hprot(3..0) := ones(4);
define master_hwdata(upper_data_bit..0) := ones(data_bus_size);
-- Replace ones by actual signal names

}

vmode ahb_master_additional_variables {

var transfer_size : {8, 16, 32, 64, 128, 256, 512, 1024};
assign transfer_size := case
master_htrans(2..0)=bits8 : 8;
master_htrans(2..0)=bits16 : 16;
master_htrans(2..0)=bits32 : 32;
master_htrans(2..0)=bits64 : 64;
master_htrans(2..0)=bits128 : 128;
master_htrans(2..0)=bits256 : 256;
master_htrans(2..0)=bits512 : 512;
master_htrans(2..0)=bits1024: 1024;
esac;

define effective_burst :=
(burst!=single) & !((burst = incr16) & (transfer_size = 1024)) &
!((burst = wrap16) & (transfer_size = 1024));

define effective_incr_burst :=
(((master_hburst(2..0)=incr) | (master_hburst(2..0)=incr4) |
(master_hburst(2..0)=incr8) | (master_hburst(2..0)=incr16))) &
!((burst = incr16) & (master_htrans(1..0)_size = 1024));

define effective_wrap_burst :=
((master_hburst(2..0)=wrap4) | (master_hburst(2..0)=wrap8) |
(master_hburst(2..0)=incr16)) &
!((burst = incr16) & (master_htrans(1..0)_size = 1024));

var adr_incr : 1..128;
assign adr_incr := case
(transfer_size=8) : 1;
(transfer_size=16) : 2;
(transfer_size=32) : 4;
(transfer_size=64) : 8;
(transfer_size=128) : 16;
(transfer_size=256) : 32;
(transfer_size=512) : 64;
(transfer_size=1024) : 128;
esac;

var wrap_upper_addr_bit : 1..9;

assign wrap_upper_addr_bit := case
master_hburst(2..0) = wrap4 & transfer_size = 8 : 1;
master_hburst(2..0) = wrap4 & transfer_size = 16 : 2;
master_hburst(2..0) = wrap4 & transfer_size = 32 : 3;
master_hburst(2..0) = wrap4 & transfer_size = 64 : 4;
master_hburst(2..0) = wrap4 & transfer_size = 128 : 5;
master_hburst(2..0) = wrap4 & transfer_size = 256 : 6;
master_hburst(2..0) = wrap4 & transfer_size = 512 : 7;

```

```

master_hburst(2..0) = wrap4 & transfer_size = 1024 : 8;
master_hburst(2..0) = wrap8 & transfer_size = 8 : 2;
master_hburst(2..0) = wrap8 & transfer_size = 16 : 3;
master_hburst(2..0) = wrap8 & transfer_size = 32 : 4;
master_hburst(2..0) = wrap8 & transfer_size = 64 : 5;
master_hburst(2..0) = wrap8 & transfer_size = 128: 6;
master_hburst(2..0) = wrap8 & transfer_size = 256: 7;
master_hburst(2..0) = wrap8 & transfer_size = 512 : 8;
master_hburst(2..0) = wrap8 & transfer_size = 1024 : 9;
master_hburst(2..0) = wrap16 & transfer_size = 8 : 3;
master_hburst(2..0) = wrap16 & transfer_size = 16 :4;
master_hburst(2..0) = wrap16 & transfer_size = 32 : 5;
master_hburst(2..0) = wrap16 & transfer_size = 64 : 6;
master_hburst(2..0) = wrap16 & transfer_size = 128: 7;
master_hburst(2..0) = wrap16 & transfer_size = 256: 8;
master_hburst(2..0) = wrap16 & transfer_size = 512 : 9;
esac;

sequence waiting = {(master_htrans(1..0)=busy) & !master_hready)
[*];((master_htrans(1..0)=seq) & !master_hready)[*]) |
((master_htrans(1..0)=busy) & master_hready)[*]};

sequence body = { waiting ; ((master_htrans(1..0)=seq) &
master_hready)};

sequence header = {(master_htrans(1..0)=nonseq) & master_hready &
(master_hburst(2..0)=single)};

sequence header4 = {(master_htrans(1..0)=nonseq) & master_hready &
((master_hburst(2..0)=incr4) |
(master_hburst(2..0)=wrap4))};

sequence header8 = {(master_htrans(1..0)=nonseq) & master_hready &
((master_hburst(2..0)=incr8) |
(master_hburst(2..0)=wrap8))};

sequence header16 = {(master_htrans(1..0)=nonseq) & master_hready &
((master_hburst(2..0)=incr16) |
(master_hburst(2..0)=wrap16))};

sequence next_transaction = {(master_htrans(1..0) =idle)
|(master_htrans(1..0) = nonseq)};

}

vmode assumption_zero_wait_state_okay_response_to_idle_transfer {

-- slave must always provide a zero wait state okay response to idle
-- transfers.

ASSUME
always((master_hready & (master_htrans(1..0)=idle))
-> next(master_hready & (master_hresp(1..0)=okay)));

}

vmode assumption_zero_wait_state_okay_response_to_busy_transfer {

```

```

-- slave must always provide a zero wait state okay response to busy
-- transfers.

ASSUME
always((master_hready & (master_htrans(1..0)=busy))
-> next(master_hready & (master_hresp(1..0)=okay)));

}

vmode assumption_
first_cycle_of_not_okay_response_followed_by_second_cycle {

-- A non okay response requires two cycles

ASSUME
always(!master_hready & (master_hresp(1..0) !=okay))
-> next(master_hready &
(master_hresp(1..0)=prev(master_hresp(1..0))));

}

vmode assumption_second_cycle_of_not_okay_response_after_first_cycle
{

-- A non okay response requires two cycles

ASSUME
always((master_hready & (master_hresp(1..0)!=okay))
-> (!prev(master_hready) & (prev(master_hresp(1..0))!=okay)));

}

vmode assumption_number_of_consecutive_wait_states_is_bounded {

-- every slave must have a predetermined maximum
-- number of wait states

ASSUME always (fell(master_hready) ->
next_a[0..wait_states_bound](!master_hready));

}

vmode ahb_slave_legal_behaviour {

inherit assumption_zero_wait_state_okay_response_to_idle_transfer;
inherit assumption_zero_wait_state_okay_response_to_idle_transfer
inherit assumption_
first_cycle_of_not_okay_response_followed_by_second_cycle;
inherit
assumption_second_cycle_of_not_okay_response_after_first_cycle;
inherit assumption_number_of_consecutive_wait_states_is_bounded;

}

vprop property_transfer_type_after_nonseq_single {

-- When hburst is single and htrans is nonseq, next value for HTRANS
-- can only be idle or nonseq

```

```

ASSERT
always(((master_htrans(1..0)=nonseq) & (master_hburst(2..0)=single))
-> next((master_htrans(1..0)=nonseq) | (master_htrans(1..0)=idle)));
}

vprop property_transfer_type_after_idle {

-- When htrans is idle, it can only take idle or nonseq value

ASSERT
always((master_htrans(1..0) =idle)
-> next((master_htrans(1..0)=idle) | (master_htrans(1..0)=nonseq)));
}

vprop property_transfer_type_after_busy {

-- After a busy transfer, htrans can only take busy or seq value

ASSERT
always(((master_htrans(1..0)=busy) & master_hready)
-> next((master_htrans(1..0)!=nonseq) & (master_htrans(1..0)!=idle)));
}

vprop property_transfer_type_after_nonseq_burst {

-- After first transfer of a burst, HTRANS can only take busy
-- or seq value

ASSERT
always(((master_htrans(1..0)=nonseq) & (master_hburst(2..0)!=single) &
master_hready)
-> next((master_htrans(1..0)=seq) | (master_htrans(1..0)=busy)));
}

vprop property_hwdata_stable_until_hready_high {

-- The master must keep HWDATA constant until HREADY is sampled high.

ASSERT
forall d(upper_data_bit..0) in boolean :
always ((!master_hready &
(master_hwdata(upper_data_bit..0)=d(upper_data_bit..0))&
master_hwrite)
-> next((master_hwdata(upper_data_bit..0)=d(upper_data_bit..0)) until
master_hready));
}

vprop property_htrans_stable_until_hready_high {

-- When htrans is nonseq or seq, the master must keep htrans
-- constant until hready high unless hresp is not okay

ASSERT

```

```

forall d(1..0) in {seq, nonseq} :
always(!master_hready & (master_htrans(1..0)=d(1..0)))
-> next((master_htrans(1..0)=d(1..0)) until master_hready)) abort
(master_hresp(1..0) !=okay);

}

vprop property_haddr_stable_until_hready_high {

-- When htrans is not idle, the master must keep haddr constant
-- until hready high unless hresp is not okay

ASSERT
forall d(31..0) in boolean :
always(!master_hready & (master_haddr(31..0)= d(31..0)) &
(master_htrans(1..0) in {seq, nonseq, busy}))
-> next((master_haddr(31..0)=d(31..0)) until master_hready)) abort
(master_hresp(1..0) !=okay);

}

vprop property_hwrite_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hwrite
-- constant until hready high unless hresp is not okay

ASSERT
forall d in boolean :
always(!master_hready & (master_hwrite=d) & (master_htrans(1..0) in
{seq, nonseq}))
-> next((master_hwrite=d) until master_hready)) abort
(master_hresp(1..0) !=okay);

}

vprop property_hburst_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hburst
-- constant until hready high unless hresp is not okay

ASSERT
forall d(2..0) in {single,incr,wrap4,incr4,wrap8,incr8,wrap16,incr16}
:
always(!master_hready & (master_hburst(2..0)=d) &
(master_htrans(1..0) in {seq, nonseq}))
-> next((master_hburst(2..0)=d(2..0)) until master_hready)) abort
(master_hresp(1..0) !=okay);

}

vprop property_hsize_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hsize
-- constant until hready high unless hresp is not okay

ASSERT
forall d in {8, 16, 32, 64, 128, 256, 512, 1024} :
always(!master_hready & (transfer_size=d) & (master_htrans(1..0) in
{seq, nonseq}))

```

```

-> next((transfer_size=d) until master_hready)) abort
(master_hresp(1..0) !=okay);
}

vprop property_hprot_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hprot
-- constant until hready high unless hresp is not okay

ASSERT
forall d(3..0) in boolean :
always(!master_hready & (master_hprot(3..0)=d(3..0)) &
(master_htrans(1..0) in {seq, nonseq}))
-> next((master_hprot(3..0)=d(3..0)) until master_hready)) abort
(master_hresp(1..0) !=okay);
}

vprop property_number_of_beats_as_expected_for_single_transfer {

-- Number of beat as expected unless response not okay

ASSERT
always ({header} | => {waiting[*]; next_transaction}) abort
(master_hresp(1..0) != okay);
}

vprop property_number_of_beats_as_expected_for_4_beats_burst {

-- Number of beat as expected unless response not okay

ASSERT
always ({header4} | => {body[*3]; next_transaction}) abort
(master_hresp(1..0) != okay);
}

vprop property_number_of_beats_as_expected_for_8_beats_burst {

-- Number of beat as expected unless response not okay

ASSERT
always ({header8} | => {body[*7]; next_transaction}) abort
(master_hresp(1..0) != okay);
}

vprop property_number_of_beats_as_expected_for_16_beats_burst {

-- Number of beat as expected unless response not okay

ASSERT
always ({header16} | => {body[*15]; next_transaction}) abort
(master_hresp(1..0) != okay);
}

```

```

vprop property_hwrite_stable_along_burst {

-- Hwrite must be kept constant by the master along a burst
-- unless hresp is not okay

ASSERT
forall d in boolean :
always (((master_htrans(1..0)=nonseq) & master_hready &
(master_hburst(2..0)!=single) & (master_hwrite=d))
->next((master_hwrite=d) until ((master_htrans(1..0)=idle) |
(master_htrans(1..0)= nonseq))))
abort (master_hresp(1..0) !=okay);

}

vprop property_hsize_stable_along_burst {

-- Hsize must be kept constant by the master along a burst
-- unless hresp not okay

ASSERT
forall d in {8, 16, 32, 64, 128, 256, 512, 1024}:
always (((master_htrans(1..0)=nonseq) & master_hready &
(master_hburst(2..0)!=single) & (transfer_size=d))
->next((transfer_size=d) until ((master_htrans(1..0)=idle) |
(master_htrans(1..0)= nonseq))))
abort (master_hresp(1..0) !=okay);

}

vprop property_hburst_stable_along_burst {

-- Hburst must be kept constant by the master along a burst
-- unless hresp not okay

ASSERT
forall d in single,incr,wrap4,incr4,wrap8,incr8,wrap16,incr16}:
always (((master_htrans(1..0)=nonseq) & master_hready &
(master_hburst(2..0)!=single) & (master_hburst(2..0)=d(2..0))
->next((master_hburst(2..0)=d(2..0)) until
((master_htrans(1..0)=idle) | (master_htrans(1..0)= nonseq))))
abort (master_hresp(1..0) !=okay);

}

vprop property_hprot_stable_along_burst {

-- Hprot must be kept stable by the master along a burst
-- unless hresp not okay

ASSERT
forall d(3..0) in boolean :
always (((master_htrans(1..0)=nonseq) & master_hready &
(master_hburst(2..0)!=single) & (master_hprot(3..0)=d(3..0)))
->next((master_hprot(3..0)=d(3..0)) until ((master_htrans(1..0)=idle)
| (master_htrans(1..0)= nonseq))))
abort (master_hresp(1..0) !=okay);

}

```

```

vprop property_haddr_stable_when_master_busy {

-- Haddr must be kept stable when htrans busy not at end of burst

ASSERT
forall d(31..0) in boolean :
always( ((master_htrans(1..0)=busy) & (master_haddr(31..0)=d(31..0)))
-> next((master_haddr(31..0)=d(31..0)) until
(master_htrans(1..0)=seq))) abort (master_hresp(1..0) !=okay);

}

vprop property_number_consecutive_busy_is_bounded {

-- The number of consecutive busy transfers is bounded

ASSERT always (rose((master_htrans(1..0)=busy)) ->
next_a[0..busy_bound]((master_htrans(1..0)=busy)));

}

vprop property_1KB_address_boundary_for_bursts {

-- Bursts must not cross 1KB address boundary

ASSERT
forall d(31..0) in boolean :
always (((master_htrans(1..0)=nonseq) & master_hready &
effective_burst & (master_haddr(31..10)=d(31..0)))
->next((master_haddr(31..10)=d(31..0)) until ((master_htrans(1..0) =
idle) | (master_htrans(1..0) = nonseq))))
abort (master_hresp(1..0) !=okay);

}

vprop property_incrementing_burst_addresses {

-- Beats addresses are correctly incremented in incrementing bursts

ASSERT
always ((master_hready & ((master_htrans(1..0)=seq) |
(master_htrans(1..0)=nonseq)) & effective_incr_burst)
-> (bvtoi(master_haddr(9..0)) <= (1024 - adr_incr)));

ASSERT
always ((master_hready & ((master_htrans(1..0)=seq) |
(master_htrans(1..0)=nonseq)) & effective_incr_burst)
-> next ((master_htrans(1..0)=seq) | (master_htrans(1..0)=busy)) -> (
master_haddr(9..0)=(prev(master_haddr(9..0))+ adr_incr)))
);

}

vprop property_wrapping_burst_addresses {

-- Beats addresses are correctly incremented in wrapping bursts

```

```

%for j in 1..7 do
ASSERT
always((master_hready & ((master_htrans(1..0)=seq) |
(master_htrans(1..0)=nonseq)) &
effective_wrap_burst &
(wrap_upper_addr_bit = %{j})))
->next(((master_htrans(1..0)=seq) | (master_htrans(1..0)=busy)) ->
((master_haddr(9..%{j+1})=prev(master_haddr(9..%{j+1}))) &
(master_haddr(%{j}..0)=prev(master_haddr(%{j}..0))+adr_incr))));
%end

ASSERT
always((master_hready & ((master_htrans(1..0)=seq) |
(master_htrans(1..0)=nonseq)) &
Effective_wrap_burst &
(wrap_upper_addr_bit = 8))
->next(((master_htrans(1..0)=seq) | (master_htrans(1..0)=busy)) ->
((master_haddr(9)=prev(master_haddr(9))) &
(master_haddr(8..0)=prev(master_haddr(8..0))+adr_incr))));

ASSERT
always((master_hready & ((master_htrans(1..0)=seq) |
(master_htrans(1..0)=nonseq)) &
effective_wrap_burst &
(wrap_upper_addr_bit = 9))
->next(((master_htrans(1..0)=seq) | (master_htrans(1..0)=busy)) ->
((master_haddr(9..0)=prev(master_haddr(9..0))+adr_incr))));

}

vprop property_address_alignment {

--all transfers must be aligned to the address boundary equal to the
-- size of transfer

ASSERT
always ((transfer_size=16) ->
(!master_haddr(0)));

ASSERT
always ((transfer_size=32) -> (master_haddr(1..0)=zeroes(2)));

ASSERT
always ((transfer_size=64) -> (master_haddr(2..0)=zeroes(3)));

ASSERT
always ((transfer_size=128) -> (master_haddr(3..0)=zeroes(4)));

ASSERT
always ((transfer_size=256) -> (master_haddr(4..0)=zeroes(5)));

ASSERT
always ((transfer_size=512) -> (master_haddr(5..0)=zeroes(6)));

ASSERT
always ((transfer_size=1024) -> (master_haddr(6..0)=zeroes(7)));

}

```

```

vprop property_hsize_not_exceeding_data_bus_size {

-- the master must never attempt a transfer of which size
-- is wider than the data bus size

ASSERT
always (transfer_size <= data_bus_size);

}

vunit transfer_type_after_nonseq_single {

-- When hburst is single and htrans is nonseq, next value for htrans
-- can only idle or nonseq

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_transfer_type_after_nonseq_single;

}

vunit transfer_type_after_idle {

-- When htrans is idle, it can only take idle or nonseq value

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_transfer_type_after_idle;

}

vunit transfer_type_after_busy {

-- After a busy transfer, htrans can only take busy or seq value

ASSERT
inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_transfer_type_after_busy;

}

vunit transfer_type_after_nonseq_burst {

-- After first transfer of a burst, htrans can only
-- take busy or seq value

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;

```

```

inherit property_transfer_type_after_nonseq_burst;

}

vunit hwd_data_stable_until_hready_high {

-- The master must keep HWDATA constant until HREADY is sampled high.

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hwd_data_stable_until_hready_high;

}

vunit htrans_stable_until_hready_high {

-- When htrans is nonseq or seq, the master must keep htrans constant
-- until hready high unless hresp is not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hwd_data_stable_until_hready_high;

}

vunit haddr_stable_until_hready_high {

-- When htrans is not idle, the master must keep haddr constant until
-- hready high unless hresp is not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_haddr_stable_until_hready_high;

}

vunit hwrite_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hwrite
-- constant until hready high unless hresp is not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hwrite_stable_until_hready_high;

}

vunit hburst_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hburst constant
until

```

```

-- hready high unless hresp is not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hwrite_stable_until_hready_high;

}

vunit hsize_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hsize
-- constant until hready high unless hresp is not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hsize_stable_until_hready_high;

}

vunit hprot_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hprot
-- constant until hready high unless hresp is not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hprot_stable_until_hready_high;

}

vunit number_of_beats_as_expected_for_single_transfer {

-- number of beat as expected unless response not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_number_of_beats_as_expected_for_single_transfer;

}

vunit number_of_beats_as_expected_for_4_beats_burst {

-- number of beat as expected unless response not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_number_of_beats_as_expected_for_4_beats_transfer;

}

```

```

vunit number_of_beats_as_expected_for_8_beats_burst {

-- number of beat as expected unless response not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_number_of_beats_as_expected_for_8_beats_transfer;

}

vunit number_of_beats_as_expected_for_16_beats_burst {

-- number of beat as expected unless response not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_number_of_beats_as_expected_for_16_beats_transfer;

}

vunit hwrite_stable_along_burst {

-- Hwrite must be kept constant by the master along a burst unless
-- hresp is not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hwrite_stable_along_burst;

}

vunit hsize_stable_along_burst {

-- Hsize must be kept constant by the master along a burst
-- unless hresp not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hsize_stable_along_burst;

}

vunit hburst_stable_along_burst {

-- Hburst must be kept constant by the master along a burst
-- unless hresp not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;

```

```

inherit property_hburst_stable_along_burst;
}

vunit hprot_stable_along_burst {

-- Hprot must be kept stable by the master along a burst
-- unless hresp not okay

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hprot_stable_along_burst;

}

vunit haddr_stable_when_master_busy {

-- Haddr must be kept stable when htrans busy not at end of burst

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_haddr_stable_when_master_busy;

}

vunit number_consecutive_busy_is_bounded {

-- the number of consecutive busy transfers is bounded

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_haddr_stable_when_master_busy;
inherit property_number_consecutive_busy_is_bounded;

}

vunit 1KB_address_boundary_for_bursts {

-- burst must not cross a 1KB address boundary

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_1KB_address_boundary_for_bursts ;
}

vunit incrementing_burst_addresses {

-- Beats addresses are correctly incremented in incrementing bursts

inherit ahb_constant;
inherit ahb_master_definitions;

```

```

inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_incrementing_burst_addresses ;

}

vunit wrapping_burst_addresses {

-- Beats addresses are correctly incremented in wrapping bursts

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_wrapping_burst_addresses ;

}

vunit address_alignment {

--all transfers must be aligned to the address boundary equal to the
-- size of transfer

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_address_alignment ;

}

vunit hsize_not_exceeding_data_bus_size {

-- the master must never attempt a transfer of which size
-- is wider than the data bus size

inherit ahb_constant;
inherit ahb_master_definitions;
inherit ahb_master_additional_variables;
inherit ahb_slave_legal_behaviour;
inherit property_hsize_not_exceeding_data_bus_size;

}

```

Slave property subset

```
vmode ahb_constant {

#define         idle             00B
#define         busy             01B
#define         nonseq          10B
#define         seq             11B

#define         single          000B
#define         incr            001B
#define         wrap4           010B
#define         incr4           011B
#define         wrap8           100B
#define         incr8           101B
#define         wrap16          110B
#define         incr16          111B

#define         bits8           000B
#define         bits16          001B
#define         bits32          010B
#define         bits64          011B
#define         bits128         100B
#define         bits256         101B
#define         bits512         110B
#define         bits1024        111B

#define         okay            00B
#define         error           01B
#define         retry           10B
#define         split           11B

}

vmode ahb_slave_definitions {

#define data_bus_size 32
#define wait_states_bound 15
#define busy_bound 15
-- Replace these numbers by appropriate values

#define upper_data_bit (data_bus_size - 1)

define slave_hclk := 1;
define slave_htrans(1..0) := ones(2);
define slave_hready := 1;
define slave_hresp(1..0) := ones(2);
define slave_hburst(2..0) := ones(3);
define slave_hsize(2..0) := ones(3);
define slave_hprot(3..0) := ones(4);
define slave_hwdata(upper_data_bit..0) := ones(data_bus_size);
-- Replace ones by actual slave signal names

}
```

```

vmode ahb_slave_additional_variables {

var transfer_size : {8, 16, 32, 64, 128, 256, 512, 1024};
assign transfer_size := case
slave_htrans(2..0)=bits8 : 8;
slave_htrans(2..0)=bits16 : 16;
slave_htrans(2..0)=bits32 : 32;
slave_htrans(2..0)=bits64 : 64;
slave_htrans(2..0)=bits128 : 128;
slave_htrans(2..0)=bits256 : 256;
slave_htrans(2..0)=bits512 : 512;
slave_htrans(2..0)=bits1024 : 1024;
esac;

define effective_burst := (burst!=single) & !((burst = incr16) &
(transfer_size = 1024)) &
!((burst = wrap16) & (transfer_size = 1024));

define effective_incr_burst :=
(((slave_hburst(2..0)=incr) | (slave_hburst(2..0)=incr4) |
(slave_hburst(2..0)=incr8) | (slave_hburst(2..0)=incr16))) &
!((burst = incr16) & (slave_htrans(1..0)_size = 1024));

define effective_wrap_burst :=
((slave_hburst(2..0)=wrap4) | (slave_hburst(2..0)=wrap8) |
(slave_hburst(2..0)=incr16)) &
!((burst = incr16) & (slave_htrans(1..0)_size = 1024));

var adr_incr : 1..128;
assign adr_incr := case
(transfer_size=8) : 1;
(transfer_size=16) : 2;
(transfer_size=32) : 4;
(transfer_size=64) : 8;
(transfer_size=128) : 16;
(transfer_size=256) : 32;
(transfer_size=512) : 64;
(transfer_size=1024) : 128;
esac;

var wrap_upper_addr_bit : 1..9;

assign wrap_upper_addr_bit := case
slave_hburst(2..0) = wrap4 & transfer_size = 8 : 1;
slave_hburst(2..0) = wrap4 & transfer_size = 16 : 2;
slave_hburst(2..0) = wrap4 & transfer_size = 32 : 3;
slave_hburst(2..0) = wrap4 & transfer_size = 64 : 4;
slave_hburst(2..0) = wrap4 & transfer_size = 128 : 5;
slave_hburst(2..0) = wrap4 & transfer_size = 256 : 6;
slave_hburst(2..0) = wrap4 & transfer_size = 512 : 7;
slave_hburst(2..0) = wrap4 & transfer_size = 1024 : 8;
slave_hburst(2..0) = wrap8 & transfer_size = 8 : 2;
slave_hburst(2..0) = wrap8 & transfer_size = 16 : 3;
slave_hburst(2..0) = wrap8 & transfer_size = 32 : 4;
slave_hburst(2..0) = wrap8 & transfer_size = 64 : 5;
slave_hburst(2..0) = wrap8 & transfer_size = 128 : 6;
slave_hburst(2..0) = wrap8 & transfer_size = 256 : 7;
slave_hburst(2..0) = wrap8 & transfer_size = 512 : 8;
slave_hburst(2..0) = wrap8 & transfer_size = 1024 : 9;

```

```

slave_hburst(2..0) = wrap16 & transfer_size = 8 : 3;
slave_hburst(2..0) = wrap16 & transfer_size = 16 : 4;
slave_hburst(2..0) = wrap16 & transfer_size = 32 : 5;
slave_hburst(2..0) = wrap16 & transfer_size = 64 : 6;
slave_hburst(2..0) = wrap16 & transfer_size = 128 : 7;
slave_hburst(2..0) = wrap16 & transfer_size = 256 : 8;
slave_hburst(2..0) = wrap16 & transfer_size = 512 : 9;
esac;

sequence waiting = {
((slave_htrans(1..0)=busy) & !slave_hready)
[*];((slave_htrans(1..0)=seq) & !slave_hready)[*] |
((slave_htrans(1..0)=busy) & slave_hready)[*]};

sequence body = { waiting ; ((slave_htrans(1..0)=seq) &
slave_hready)};

sequence header = {(slave_htrans(1..0)=nonseq) & slave_hready &
(slave_hburst(2..0)=single)};

sequence header4 = {(slave_htrans(1..0)=nonseq) & slave_hready &
((slave_hburst(2..0)=incr4) | (slave_hburst(2..0)=wrap4))};

sequence header8 = {(slave_htrans(1..0)=nonseq) & slave_hready &
((slave_hburst(2..0)=incr8) | (slave_hburst(2..0)=wrap8))};

sequence header16 = {(slave_htrans(1..0)=nonseq) & slave_hready &
((slave_hburst(2..0)=incr16) | (slave_hburst(2..0)=wrap16))};

sequence next_transaction = {(slave_htrans(1..0) =idle)
|(slave_htrans(1..0) = nonseq)};
}

vmode assumption_transfer_type_after_nonseq_single {
-- When hburst is single and htrans is nonseq, next value for HTRANS
-- can only be idle or nonseq

ASSUME
always(((slave_htrans(1..0)=nonseq) & (slave_hburst(2..0)=single))
-> next((slave_htrans(1..0)=nonseq) | (slave_htrans(1..0)=idle)));
}

vmode assumption_transfer_type_after_idle {
-- When htrans is idle, it can only take idle or nonseq value

ASSUME
always((slave_htrans(1..0) =idle)
-> next((slave_htrans(1..0)=idle) | (slave_htrans(1..0)=nonseq)));
}

vmode assumption_transfer_type_after_busy {
-- After a busy transfer, htrans can only take busy or seq value

ASSUME

```

```

always(((slave_htrans(1..0)=busy) & slave_hready)
-> next((slave_htrans(1..0)!=nonseq) & (slave_htrans(1..0)!=idle)));

}

vmode assumption_transfer_type_after_nonseq_burst {

-- After first transfer of a burst, HTRANS can only take busy
-- or seq value

ASSUME
always(((slave_htrans(1..0)=nonseq) & (slave_hburst(2..0)!=single) &
slave_hready)
-> next((slave_htrans(1..0)=seq) | (slave_htrans(1..0)=busy)));

}

vmode assumption_hwdata_stable_until_hready_high {

-- The master must keep HWDATA constant until HREADY is sampled high.

ASSUME
forall d(upper_data_bit..0) in boolean :
always ((!slave_hready &
(slave_hwdata(upper_data_bit..0)=d(upper_data_bit..0))& slave_hwrite)
-> next((slave_hwdata(upper_data_bit..0)=d(upper_data_bit..0)) until
slave_hready));

}

vmode assumption_htrans_stable_until_hready_high {

-- When htrans is nonseq or seq, the master must keep htrans constant
-- until hready high unless hresp is not okay

ASSUME
forall d(1..0) in {seq, nonseq} :
always ((!slave_hready & (slave_htrans(1..0)=d(1..0)))
-> next((slave_htrans(1..0)=d(1..0)) until slave_hready)) abort
(slave_hresp(1..0) !=okay);

}

vmode assumption_haddr_stable_until_hready_high {

-- When htrans is not idle, the master must keep haddr constant until
-- hready high unless hresp is not okay

ASSUME
forall d(31..0) in boolean :
always ((!slave_hready & (slave_haddr(31..0)= d(31..0)) &
(slave_htrans(1..0) in {seq, nonseq, busy}))
-> next((slave_haddr(31..0)=d(31..0)) until slave_hready)) abort
(slave_hresp(1..0) !=okay);

}

vmode assumption_hwrite_stable_until_hready_high {

```

```

-- When htrans is seq or nonseq, the master must keep hwrite constant
-- until hready high unless hresp is not okay

ASSUME
forall d in boolean :
always((!slave_hready & (slave_hwrite=d) & (slave_htrans(1..0) in
{seq, nonseq}))
-> next((slave_hwrite=d) until slave_hready)) abort
(slave_hresp(1..0) !=okay);

}

vmode assumption_hburst_stable_until_hready_high {
-- When htrans is seq or nonseq, the master must keep hburst constant
-- until hready high unless hresp is not okay

ASSUME
forall d(2..0) in {single,incr,wrap4,incr4,wrap8,incr8,wrap16,incr16}
:
always((!slave_hready & (slave_hburst(2..0)=d) & (slave_htrans(1..0)
in {seq, nonseq}))
-> next((slave_hburst(2..0)=d(2..0)) until slave_hready)) abort
(slave_hresp(1..0) !=okay);

}

vmode assumption_hsize_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hsize constant
-- until hready high unless hresp is not okay

ASSUME
forall d in {8, 16, 32, 64, 128, 256, 512, 1024} :
always((!slave_hready & (transfer_size=d) & (slave_htrans(1..0) in
{seq, nonseq}))
-> next((transfer_size=d) until slave_hready)) abort
(slave_hresp(1..0) !=okay);

}

vmode assumption_hprot_stable_until_hready_high {

-- When htrans is seq or nonseq, the master must keep hprot constant
-- until hready high unless hresp is not okay

ASSUME
forall d(3..0) in boolean :
always((!slave_hready & (slave_hprot(3..0)=d(3..0)) &
(slave_htrans(1..0) in {seq, nonseq}))
-> next((slave_hprot(3..0)=d(3..0)) until slave_hready)) abort
(slave_hresp(1..0) !=okay);

}

vmode assumption_number_of_beats_as_expected_for_single_transfer {

-- number of beat as expected unless response not okay

```

```

ASSUME
always ({header} |> {waiting[*]; next_transaction}) abort
(slave_hresp(1..0) != okay);

}

vmode assumption_number_of_beats_as_expected_for_4_beats_burst {

-- number of beat as expected unless response not okay

ASSUME
always ({header4} |> {body[*3]; next_transaction}) abort
(slave_hresp(1..0) != okay);

}

vmode assumption_number_of_beats_as_expected_for_8_beats_burst {

-- number of beat as expected unless response not okay

ASSUME
always ({header8} |> {body[*7]; next_transaction}) abort
(slave_hresp(1..0) != okay);

}

vmode assumption_number_of_beats_as_expected_for_16_beats_burst {

-- number of beat as expected unless response not okay

ASSUME
always ({header16} |> {body[*15]; next_transaction}) abort
(slave_hresp(1..0) != okay);

}

vmode assumption_hwrite_stable_along_burst {

-- Hwrite must be kept constant by the master along a burst unless
-- hresp is not okay

ASSUME
forall d in boolean :
always (((slave_htrans(1..0)=nonseq) & slave_hready &
(slave_hburst(2..0)!=single) & (slave_hwrite=d))
->next((slave_hwrite=d) until ((slave_htrans(1..0)=idle) |
(slave_htrans(1..0)= nonseq))))
abort (slave_hresp(1..0) !=okay);

}

vmode assumption_hsize_stable_along_burst {

-- Hsize must be kept constant by the master along a burst
-- unless hresp not okay

ASSUME
forall d in {8, 16, 32, 64, 128, 256, 512, 1024}:
always (((slave_htrans(1..0)=nonseq) & slave_hready &
(slave_hburst(2..0)!=single) & (transfer_size=d))

```

```

->next((transfer_size=d) until ((slave_htrans(1..0)=idle) |
(slave_htrans(1..0)= nonseq))))
abort (slave_hresp(1..0) !=okay);

}

vmode assumption_hburst_stable_along_burst {

-- Hburst must be kept constant by the master along a burst
-- unless hresp not okay

ASSUME
forall d in single,incr,wrap4,incr4,wrap8,incr8,wrap16,incr16}:
always (((slave_htrans(1..0)=nonseq) & slave_hready &
(slave_hburst(2..0)!=single) & (slave_hburst(2..0)=d(2..0))
->next((slave_hburst(2..0)=d(2..0)) until ((slave_htrans(1..0)=idle)
| (slave_htrans(1..0)= nonseq))))
abort (slave_hresp(1..0) !=okay);

}

vmode assumption_hprot_stable_along_burst {

-- Hprot must be kept stable by the master along a burst
-- unless hresp not okay

ASSUME
forall d(3..0) in boolean :
always (((slave_htrans(1..0)=nonseq) & slave_hready &
(slave_hburst(2..0)!=single) & (slave_hprot(3..0)=d(3..0)))
->next((slave_hprot(3..0)=d(3..0)) until ((slave_htrans(1..0)=idle) |
(slave_htrans(1..0)= nonseq))))
abort (slave_hresp(1..0) !=okay);

}

vmode assumption_haddr_stable_when_slave_busy {

-- Haddr must be kept stable when htrans busy not at end of burst

ASSUME
forall d(31..0) in boolean :
always( ((slave_htrans(1..0)=busy) & (slave_haddr(31..0)=d(31..0)))
-> next((slave_haddr(31..0)=d(31..0)) until (slave_htrans(1..0)=seq)))
abort (slave_hresp(1..0) !=okay);

}

vmode assumption_number_consecutive_busy_is_bounded {

-- the number of consecutive cycles when HTRANS is busy, is bounded

ASSUME always (rose((slave_htrans(1..0)=busy)) ->
next_a[0..busy_bound]((slave_htrans(1..0)=busy)));

}

vmode assumption_1KB_address_boundary_for_bursts {

```

```

-- burst must not cross a 1KB address boundary

ASSUME
forall d(31..0) in boolean :
always (((slave_htrans(1..0)=nonseq) & slave_hready & effective_burst
& (slave_haddr(31..10)=d(31..0)))
->next((slave_haddr(31..10)=d(31..0)) until ((slave_htrans(1..0) =
idle) | (slave_htrans(1..0) = nonseq))))
abort (slave_hresp(1..0) !=okay);

}

vmode assumption_incrementing_burst_addresses {

-- Beats addresses are correctly incremented in incrementing bursts

ASSUME
always ((slave_hready & ((slave_htrans(1..0)=seq) |
(slave_htrans(1..0)=nonseq)) & effective_incr_burst)
-> (bvtoi(slave_haddr(9..0)) <= (1024 - adr_incr)));

ASSUME
always ((slave_hready & ((slave_htrans(1..0)=seq) |
(slave_htrans(1..0)=nonseq)) & effective_incr_burst)
-> next (((slave_htrans(1..0)=seq) | (slave_htrans(1..0)=busy)) -> (
slave_haddr(9..0)=(prev(slave_haddr(9..0))+ adr_incr))
));

}

vmode assumption_wrapping_burst_addresses {

-- Beats addresses are correctly incremented in wrapping bursts

%for j in 1..7 do
ASSUME
always((slave_hready & ((slave_htrans(1..0)=seq) |
(slave_htrans(1..0)=nonseq)) &
effective_wrap_burst &
(wrap_upper_addr_bit = %{j})))
->next(((slave_htrans(1..0)=seq) | (slave_htrans(1..0)=busy)) ->
((slave_haddr(9..%{j+1})=prev(slave_haddr(9..%{j+1}))) &
(slave_haddr(%{j}..0)=prev(slave_haddr(%{j}..0))+adr_incr)))));
%end

ASSUME
always((slave_hready & ((slave_htrans(1..0)=seq) |
(slave_htrans(1..0)=nonseq)) &
Effective_wrap_burst &
(wrap_upper_addr_bit = 8))
->next(((slave_htrans(1..0)=seq) | (slave_htrans(1..0)=busy)) ->
((slave_haddr(9)=prev(slave_haddr(9))) &
(slave_haddr(8..0)=prev(slave_haddr(8..0))+adr_incr))));

ASSUME
always((slave_hready & ((slave_htrans(1..0)=seq) |
(slave_htrans(1..0)=nonseq)) &
effective_wrap_burst &
(wrap_upper_addr_bit = 9))

```

```

->next(((slave_htrans(1..0)=seq) | (slave_htrans(1..0)=busy)) ->
((slave_haddr(9..0)=prev(slave_haddr(9..0))+adr_incr)));
}

```

```

vmode assumption_address_alignment {

```

```

--all transfers must be aligned to the address boundary equal to the
-- size of transfer

```

```

ASSUME

```

```

always ((slave_htrans(1..0)_size=16) ->
(!slave_haddr(0)));

```

```

ASSUME

```

```

always ((transfer_size=32) -> (slave_haddr(1..0)=zeroes(2)));

```

```

ASSUME

```

```

always ((transfer_size=64) -> (slave_haddr(2..0)=zeroes(3)));

```

```

ASSUME

```

```

always ((transfer_size=128) -> (slave_haddr(3..0)=zeroes(4)));

```

```

ASSUME

```

```

always ((transfer_size=256) -> (slave_haddr(4..0)=zeroes(5)));

```

```

ASSUME

```

```

always ((transfer_size=512) -> (slave_haddr(5..0)=zeroes(6)));

```

```

ASSUME

```

```

always ((transfer_size=1024) -> (slave_haddr(6..0)=zeroes(7)));
}

```

```

vmode assumption_hsize_not_exceeding_data_bus_size {

```

```

-- the slave must never attempt a transfer of which size
-- is wider than the data bus size

```

```

ASSUME

```

```

always (transfer_size <= data_bus_size);
}

```

```

vmode ahb_slave_legal_behaviour {

```

```

inherit assumption_transfer_type_after_nonseq_single;
inherit assumption_transfer_type_after_idle;
inherit assumption_transfer_type_after_busy;
inherit assumption_transfer_type_after_nonseq_burst;
inherit assumption_hwdata_stable_until_hready_high;
inherit assumption_htrans_stable_until_hready_high;
inherit assumption_haddr_stable_until_hready_high;
inherit assumption_hwrite_stable_until_hready_high;
inherit assumption_hburst_stable_until_hready_high;
inherit assumption_hsize_stable_until_hready_high;
inherit assumption_hprot_stable_until_hready_high;

```

```

inherit assumption_number_of_beats_as_expected_for_single_transfer;
inherit assumption_number_of_beats_as_expected_for_4_beats_burst;
inherit assumption_number_of_beats_as_expected_for_8_beats_burst;
inherit assumption_number_of_beats_as_expected_for_16_beats_burst;
inherit assumption_hwrite_stable_along_burst;
inherit assumption_hsize_stable_along_burst;
inherit assumption_hburst_stable_along_burst;
inherit assumption_hprot_stable_along_burst;
inherit assumption_haddr_stable_when_master_busy;
inherit assumption_number_consecutive_busy_is_bounded;
inherit assumption_1KB_address_boundary_for_bursts ;
inherit assumption_incrementing_burst_addresses ;
inherit assumption_wrapping_burst_addresses ;
inherit assumption_address_alignment;
inherit assumption_hsize_not_exceeding_data_bus_size;

}

vprop property_zero_wait_state_okay_response_to_idle_transfer {

-- slave must always provide a zero wait state okay response to
-- idle transfers.

ASSERT
always((slave_hready & (slave_htrans(1..0)=idle))
--> next(slave_hready & (slave_hresp(1..0)=okay)));

}

vprop property_zero_wait_state_okay_response_to_busy_transfer {

-- slave must always provide a zero wait state okay response
-- to busy transfers.

ASSERT
always((slave_hready & (slave_htrans(1..0)=busy))
-> next(slave_hready & (slave_hresp(1..0)=okay)));

}

vprop property_
first_cycle_of_not_okay_response_followed_by_second_cycle {

-- A non okay response requires two cycles

ASSERT
always((!slave_hready & (slave_hresp(1..0) !=okay))
-> next(slave_hready &
(slave_hresp(1..0)=prev(slave_hresp(1..0)))));

vprop property_second_cycle_of_not_okay_response_after_first_cycle
{

-- A non okay response requires two cycles

ASSERT
always((slave_hready & (slave_hresp(1..0)!=okay))
-> (!prev(slave_hready) & (prev(slave_hresp(1..0))!=okay)));

}

```

```

vprop property_number_of_consecutive_wait_states_is_bounded {

-- every slave must have a predetermined maximum
-- number of wait states

ASSERT
always (fell(slave_hready) ->
next_a[0..wait_states_bound](!slave_hready));

}

vunit zero_wait_state_okay_response_to_idle_transfer {

-- slave must always provide a zero wait state okay response
-- to idle transfers.

inherit ahb_constant;
inherit ahb_slave_definitions;
inherit ahb_slave_additional_variables;
inherit ahb_master_legal_behaviour;

inherit property_zero_wait_state_okay_response_to_idle_transfer;

}

vunit zero_wait_state_okay_response_to_busy_transfer {

-- slave must always provide a zero wait state okay response
-- to busy transfers.

inherit ahb_constant;
inherit ahb_slave_definitions;
inherit ahb_slave_additional_variables;
inherit ahb_master_legal_behaviour;

inherit property_zero_wait_state_okay_response_to_busy_transfer;

}

vunit first_cycle_of_not_okay_response_followed_by_second_cycle {

-- A non okay response requires two cycles

inherit ahb_constant;
inherit ahb_slave_definitions;
inherit ahb_slave_additional_variables;
inherit ahb_master_legal_behaviour;

inherit
property_first_cycle_of_not_okay_response_followed_by_second_cycle;

}

vunit second_cycle_of_not_okay_response_after_first_cycle {

-- A non okay response requires two cycles

inherit ahb_constant;
inherit ahb_slave_definitions;

```

```
inherit ahb_slave_additional_variables;
inherit ahb_master_legal_behaviour;

inherit property_
first_cycle_of_not_okay_response_followed_by_second_cycle;

}

vunit number_of_consecutive_wait_states_is_bounded {

-- every slave must have a predetermined maximum
-- number of wait states

inherit ahb_constant;
inherit ahb_slave_definitions;
inherit ahb_slave_additional_variables;
inherit ahb_master_legal_behaviour;

inherit property_number_of_consecutive_wait_states_is_bounded

}
```

Annex B: FIFO

```
vmode fifo_definitions {

#define fifo_upper_bound 100
-- could be bigger than actual FIFO size;
-- we just need some finite bound.
#define data_width 32;
-- Replace these numbers by appropriate values.

define data_in(data_width - 1 .. 0) := zeroes(data_width);
define data_out(data_width-1 .. 0) := zeroes(data_width);
-- Replace by actual signal names for read and write data.

define write_clock := 1;
define read_clock := 1;
-- Replace by the expressions on which writes and reads are clocked.
-- If, for example, writes are clocked on the rising edge of clk_in,
-- the expression should be rose(clk_in).

define fifo_ready_for_write := 1;
-- Replace 1 by the appropriate signal name, e.g. write_gnt.
-- If no grant from FIFO is required, leave as 1.
-- If grant is allowed to depend on request, replace 1 by e.g.
-- write_req -> write_gnt.

define environment_ready_to_write := 1;
-- Replace 1 by the appropriate signal name, e.g. write_req
-- If request is allowed to depend on grant, replace 1 by e.g.
-- write_gnt -> write_req.

define fifo_ready_for_read := read_gnt;
-- Replace 1 by the appropriate signal name, e.g. read_gnt.
-- If no grant from FIFO is required, leave as 1.
-- If grant is allowed to depend on request, replace by e.g
-- read_req -> read_gnt.

define environment_ready_to_read := 1;
-- Replace by the appropriate signal name, e.g. read_req.
-- If no request from environment is required, leave as 1.
-- If request is allowed to depend on grant, replace by e.g.
-- read_gnt -> read_req.

define write_event :=
    environment_ready_to_write & fifo_ready_for_write & write_clock;
define read_event :=
    environment_ready_to_read & fifo_ready_for_read & read_clock;

}
```

```

vmode fifo_optional_definitions {
-- definitions for properties that don't have to hold
-- for all fifos; they may represent specific protocol
-- requirements for a particular environment, etc.

#define exact_fifo_size 16
-- Replace by size of FIFO (number of data entries it can hold).

define fifo_grant_write := 1;
-- Replace by the name of the grant signal for writes.
-- fifo_grant_write may be different from fifo_ready_for_write; e.g.
-- if grant is allowed to depend on request, then fifo_ready_for_write
-- is (write_req -> write_gnt), but fifo_grant_write is just
-- write_gnt.

define fifo_grant_read := 1;
-- Replace by the name of the grant signal for reads
-- fifo_grant_read may be different from fifo_ready_for_read; e.g.
-- if grant is allowed to depend on request, then fifo_ready_for_read
-- is (read_req -> read_gnt), but fifo_grant_read is just read_gnt.

define environment_request_write := 1;
-- Replace by the name of the request signal for writes.
-- environment_request_write may be different from
-- environment_ready_to_write;
-- e.g. if request is allowed to depend on grant, then
-- environment_ready_to_write is
-- (write_gnt -> write_req), but environment_request_write is
-- just write_req.

define environment_request_read := 1;
-- Replace by the name of the request signal for reads
-- environment_request_read may be different from
-- environment_ready_to_read;
-- e.g. if request is allowed to depend on grant, then
-- environment_ready_to_read is
-- (read_gnt -> read_req), but environment_request_read is
-- just read_req.

#define read_latency 0;
-- Replace by number of cycles of read clock between when there's
-- data in the FIFO and when the data is required
-- to be available for reading.

#define write_latency 0;
-- Replace by number of cycles of write clock between any time
-- when the FIFO is not full and when the FIFO is required to be
-- available for writing.

define fifo_full := 0;
-- Replace by name of output indicating that FIFO is full.

define fifo_empty := 0;
-- Replace by name of output indicating that FIFO is empty.

define status_full_clock := write_clock;
-- Replace, if necessary, by expression on which fifo_full
-- is clocked

define status_empty_clock := read_clock;

```

```

-- Replace, if necessary, by expression on which fifo_empty
-- is clocked.

}

vmode fifo_count_fifo_entries {

inherit fifo_definitions;

var number_of_fifo_entries : 0..fifo_upper_bound - 1;

assign
  init(number_of_fifo_entries) := 0;
  next(number_of_fifo_entries) :=
    case
      (number_of_fifo_entries < fifo_upper_bound)
      & write_event & !read_event:
        number_of_fifo_entries + 1;
      (number_of_fifo_entries > 0) & read_event & !write_event :
        number_of_fifo_entries - 1;
      else : number_of_fifo_entries;
    esac;

}

vmode fifo_count_entries_ahead {

-- We keep track of the number of entries in the FIFO ahead of a
-- distinguished entry, that occurs at an arbitrary time when the step
-- function element_has_entered rises.
-- This vmode should be used with a property where entry_event
-- occurs at the same time as element_has_entered rises.

inherit fifo_definitions;

var element_has_entered : boolean;
assign
  init(element_has_entered) := 0;
  next(element_has_entered) :=
    case
      element_has_entered : 1;
      else : {0,1};
    esac;

var entries_ahead : 0..fifo_upper_bound;

assign
  init(entries_ahead) := 0;
  next(entries_ahead) :=
    case
      !element_has_entered & (entries_ahead > 0) & exit_event &
      !entry_event :
        entries_ahead - 1;
      !element_has_entered & (entries_ahead < fifo_upper_bound) &
      entry_event & !exit_event :
        entries_ahead + 1;
      element_has_entered & (entries_ahead > 0) & exit_event :
        entries_ahead - 1;
      else : entries_ahead;
    esac;

```

```

}

vunit fifo_write_liveness {

-- If the environment doesn't refuse ever to read from the fifo,
-- the FIFO must eventually be ready to accept writes.

inherit fifo_definitions;

ASSUME always eventually! environment_ready_to_read & read_clock;

ASSERT always eventually! fifo_ready_for_write & write_clock ;

}

vunit fifo_read_liveness {

-- If there are entries in the fifo, the FIFO must eventually
-- be ready to let them be read.

inherit fifo_definitions;
inherit fifo_count_fifo_entries;

ASSERT
  always
    (!(number_of_fifo_entries = 0) ->
      eventually! fifo_ready_for_read & read_clock);

}

vunit fifo_no_read_when_empty {

-- Can't read from FIFO when there are no entries in it.

inherit fifo_definitions;
inherit fifo_count_entries;

ASSERT
  always ((number_of_fifo_entries = 0) ->
    !(fifo_ready_for_read & read_clock));

}

vunit fifo_data_correspondence {

-- Focus on the data written the FIFO at one particular arbitrary
-- time. If there are n entries in the FIFO at that time,
-- then the nth subsequent read from the FIFO must have that data
-- value.

inherit fifo_count_entries_ahead;

define w := data_width;

ASSERT
  forall v(w-1..0) in boolean:
    {[*]; entry_event & (data_in(w-1..0) = v(w-1..0)) &
      rose(element_has_entered) ;
      (exit_event & (elements_ahead = 0))[->]}
    |->
    {data_out(w-1..0) = v(w-1..0)};

```

```

}

vunit fifo_write_grant_only_on_request {

-- Applies if FIFO is not allowed to ASSERT grant for write
-- unless the environment is requesting.

inherit fifo_definitions;
inherit fifo_optional_definitions;

ASSERT
  (always( fifo_grant_write ->
    environment_request_write))@write_clock;
}

vunit fifo_read_grant_only_on_request {

-- Applies if FIFO is not allowed to ASSERT grant for read
-- unless the environment is requesting.

inherit fifo_definitions;
inherit fifo_optional_definitions;

ASSERT
  (always(fifo_grant_read -> environment_request_read))@read_clock;
}

vunit fifo_not_read_and_write {

-- Applies if the FIFO is not allowed to accept reads
-- and writes simultaneously.

inherit fifo_definitions;
inherit fifo_optional_definitions;

ASSERT
  never(fifo_grant_read & read_clock & fifo_grant_write &
    write_clock);
}

vmode fifo_environment_not_read_and_write {

-- Applies if the environment is not allowed to make reads
-- and writes simultaneously.

inherit fifo_definitions;
inherit fifo_optional_definitions;

ASSUME
  never(environment_request_read & read_clock &
    environment_request_write & write_clock);
}

vunit fifo_read_latency {

-- Applies if the FIFO is required to make data available

```

```

-- for reading within a certain number of cycles of
-- the read clock, if it is not empty.

inherit fifo_definitions;
inherit fifo_optional_definitions;
inherit fifo_count_entries;

ASSERT
  always
    ((number_of_fifo_entries > 0) ->
     (next_e[0..read_latency](fifo_ready_for_read))
     @read_clock);

}

vunit fifo_write_latency {

-- Applies if the FIFO is required to become available
-- for writing within a certain number of cycles of
-- the write clock, if it is not full.

inherit fifo_definitions;
inherit fifo_optional_definitions;
inherit fifo_count_entries;

ASSERT
  always
    ((number_of_fifo_entries < exact_fifo_size) ->
     (next_e[0..write_latency](fifo_ready_for_write))
     @write_clock);

}

vunit fifo_full {

-- Applies if there is an output signifying that the FIFO is full.

ASSERT
  (always
   (fifo_full <->
    (number_of_fifo_entries = exact_fifo_size)))
   @ status_full_clock;

}

vunit fifo_empty {

-- Applies if there is an output signifying that the FIFO is empty.

ASSERT
  (always
   (fifo_empty <-> (number_of_fifo_entries = 0)))
   @ status_empty_clock;

}

```


Annex C: Stack

```
vmode stack_definitions {

define push      := my_push;
define pop       := my_pop;
define d_in     := my_d_in;
define rst       := my_reset;
define empty    := my_empty;
define full     := my_full;
define memory   := my_memory;
define s_ptr    := my_stackpointer;
-- Replace the right side of these definitions by appropriate signals
}

vunit stack_rst {

    inherit stack_definitions;

    ASSUME rst = '1';

    ASSERT {[*1]; s_ptr = B"0" and empty = '1' and full = '0'};
}

vunit stack_nop {

    inherit stack_macros;

    ASSUME pop = '0' and push = '0' and rst = '0';

    ASSERT {[*1]; stable_after_nop};
}

vunit stack_r {

    inherit stack_macros;

    ASSUME pop = '1' and empty = '0' and push = '0' and rst = '0' and
s_ptr_range;

    ASSERT {[*1]; update_after_pop};
}

vunit stack_w {
```

```

inherit stack_macros;

ASSUME push = '1' and full = '0' and pop = '0' and rst = '0' and
s_ptr_range and sp_zero_if_empty;

ASSERT {[*1]; update_after_push};
}

vunit stack_conserve_ram {
    inherit stack_definitions;

    variable STP_t: unsigned (a_msb downto 0);

    ASSUME {memory(s_ptr) /= d_in; memory(prev(s_ptr)) = prev(d_in)};

    ASSERT push = '1' and full = '0';
}

vunit stack_full {
    inherit stack_macros;

    ASSUME full = '1' and rst = '0' and sp_zero_if_empty;

    ASSERT {empty = '0'};
    ASSERT {pop = '1'} | => {update_after_pop};
    ASSERT {pop /= '1'} | => {stable_after_nop};
}

vunit stack_empty {
    inherit stack_macros;

    ASSUME empty = '1' and rst = '0' and sp_zero_if_empty;

    ASSERT {full = '0'};
    ASSERT {push = '1'} | => {update_after_push};
    ASSERT {push /= '1'} | => {stable_after_nop};
}

vunit stack_rw_push_overrides { -- simultaneous read/write (pop/push)

    inherit stack_macros;

    ASSUME push = '1' and pop = '1' and rst = '0'
        and s_ptr_range and sp_zero_if_empty;

    ASSERT {full = '0'} | => {update_after_push};
    ASSERT {full /= '0'} | => {update_after_pop};
}

vunit stack_rw_full_bypass {

    inherit stack_definitions;
    inherit stack_macros;

```

```

    ASSUME push = '1' and pop = '1' and rst = '0';

    ASSERT {[*1]; d_out = prev(d_in) and storage_unchanged};
}

vunit stack_rw_safe_bypass {

    inherit stack_macros;

    ASSUME push = '1' and pop = '1' and rst = '0' and sp_zero_if_empty;

    ASSERT {full = '0' and empty = '0'} | => {d_out = prev(d_in) and
storage_unchanged};
    ASSERT {empty = '1'} | => {update_after_push};
    ASSERT {full = '1'} | => {update_after_pop};

}

vunit stack_w_rb {

    inherit stack_definitions;

    ASSUME {push = '1' and full = '0' and pop = '0' and rst = '0';
pop = '1' and push = '0' and rst = '0'};

    ASSERT {[*2]; d_out = prev(d_in, 2)};
}

vunit stack_w_rb_by3 {

    inherit stack_definitions;
    verification_window = 5;
    -- Note: the "verification_window" construct is
    -- specific to OneSpin's tool and can be deleted
    -- when using other tools.

    variable pushed: unsigned(d_msb downto 0);
    ASSUME always stable(pushed);
    ASSUME pushed = d_in;

    ASSERT {{push = '1' and full = '0' and pop = '0';
push = '0' and pop = '0' [*1 to inf];
pop = '1' and push = '0'} && rst = '0'[*]} | => {d_out = pushed};

}

vunit stack_invariants_Reset {

    inherit stack_macros;

    assume rst = '1';

    assert {[*1]; s_ptr_range and sp_zero_if_empty};

}

vunit stack_invariants_Induct {

    inherit stack_macros;

```

```

assume s_ptr_range and sp_zero_if_empty;

assert {[*1]; s_ptr_range and sp_zero_if_empty};
}

vmode stack_macros {

inherit stack_definitions;

function s_ptr_range
  return boolean
is
begin
  return s_ptr <= a_max;
end;

function sp_zero_if_empty
  return boolean
is
begin
  if empty = '1' then
    return s_ptr = B"0";
  else
    return s_ptr >= 0;
  end if;
end;

function storage_unchanged
  return boolean
is
begin
  return stable(s_ptr) and stable(memory) and stable(empty) and
    stable(full);
end;

function stable_after_nop
  return boolean
is
begin
  return d_out = B"0" and storage_unchanged;
end;

function update_after_push
  return boolean
is
begin
  return d_out = B"0" and
    empty = '0' and
    memory(s_ptr) = prev(d_in) and
    ((prev(empty) = '1' and s_ptr = 0)
     or (prev(empty) /= '1' and s_ptr = prev(s_ptr) + 1)) and
    ((s_ptr = a_max and full = '1') or
     (s_ptr /= a_max and full = '0'));
end;

function update_after_pop
  return boolean
is
begin
  return full = '0' and

```

```
    d_out = memory(prev(s_ptr)) and
    stable(memory) and
    ((prev(s_ptr) = B"0" and empty = '1' and s_ptr = 0)
     or (prev(s_ptr) /= 0 and empty = '0' and
         s_ptr = prev(s_ptr - 1)));
end;

}
```

Annex D: Configuration registers

```
vmode register_interface_definitions {

#define addr_upper_bit          15
#define adr_lower_bit          2
#define adr_width              14
#define data_bus_size          32
#define data_upper_bit         31
-- Replace these numbers by appropriate values from design

define register_req := 1;
define register_addr(addr_upper_bit..addr_lower_bit) :=
ones(addr_width);
define register_data(data_upper_bit..0) := ones(32);
define register_r_req := 1;
define register_r_data(data_upper_bit..0) := ones(32);
define register_error := 1;
-- Replace ones by actual design signal names

}

vmode register_additional_definitions {

#define reg_i_addr 6000H

-- Replace reg_i with actual register name
-- Replace value with its address
-- Reproduce this macro definition for each register

define reg_i_read := register_req &
register_addr(upper_adr_bit..lower_adr_bit) = reg_i_addr) & read;
define reg_i_write := register_req &
register_addr(upper_adr_bit..lower_adr_bit) = reg_i_addr) & write;
-- Reproduce these two definitions for each used register

define valid_address := register_addr(upper_adr_bit..lower_adr_bit)
in {reg_i_addr
-- ,...
};
-- Complete register address set with all relevant address values

define byte_j := 1;
-- Replace 1 by boolean condition involving actual design signal
-- names
-- Reproduce this macro for each byte of data bus
```

```

define write := 1;
-- Replace 1 by boolean condition involving actual design input
-- signal names

define read := 1;
-- Replace 1 by boolean condition involving actual design input
-- signal names

    define valid_operation := 1
-- Replace 1 by boolean condition involving actual design input
-- signal

}

--
-- Protocol assumptions
--

vmode requests_not_retracted {

    --Assumption: requests are not retracted

    ASSIGN
        next(register_req) :=
            case
                register_req & !register_r_req : register_req;
                else : {0,1};
            esac;

}

vmode request_data_stable {

    --Assumption: request data do not change while waiting for response

    ASSIGN
        next(register_data(data_upper_bit ..0)) :=
            case
                register_req & !register_r_req :
                    register_data(data_upper_bit..0);
                else : nondets(data_bus_size);
            esac;

}

vmode request_addr_stable {

    --Assumption: request do not change while waiting for response

    ASSIGN
        next(register_addr(addr_upper_bit)) :=
            case
                register_req & !register_r_req :
                    register_addr(addr_upper_bit..addr_lower_bit);
                else : nondets(addr_width);
            esac;

}

```

```

--
-- Protocol requirements
--

vunit requests_get_responses {

-- Requests are eventually answered

    inherit register_interface_definitions;
    inherit requests_not_retracted;

    ASSERT
        always (register_req -> eventually! register_r_req);

}

vunit no_unsolicited_responses {

--No unsolicited responses

    inherit register_interface_definitions;
    inherit requests_not_retracted;

    ASSERT
        always (!register_req -> !register_r_req);

}

vunit response_after_request {

--Response comes at least 1 cycle after request

    inherit register_interface_definitions;
    inherit requests_not_retracted;

    DEFINE
        new_req := register_req & (!prev(register_req) |
prev(register_r_req));

    ASSERT
        always {new_req} |-> {!register_r_req};

}

vunit invalid_accesses_set_error {

--Invalid accesses always set error = 1

    inherit register_interface_definitions;
    inherit register_additional_definitions;
    inherit requests_not_retracted;
    inherit request_data_stable;

    ASSERT
        always {register_req & (!valid_operation | !valid_address) &
register_r_req} |-> {register_error};

}

--
-- Functional properties

```

```

--
vunit reg_i_read_byte_j {
    -- Reads always return the most recently written value to byte j

    inherit register_interface_definitions;
    inherit register_additional_definitions;
    inherit requests_not_retracted;
    inherit request_data_stable;

#define data_bit_high 7
#define data_bit_low 0
-- Replace these numbers by values corresponding to byte_j

    DEFINE
        valid_read := reg_i_read & byte_j & valid_operation;
        valid_write := reg_i_write & byte_j & valid_operation ;

    VAR
        most_recent_written_byte(7..0) : boolean;

    ASSIGN
        init(most_recent_written_byte(7..0)) := 0; -- reset value
        next(most_recent_written_byte(7..0)) :=
            case
                valid_write & register_r_req :
                    register_data(data_bit_high..data_bit_low);
--            external_input : ...;
                else : most_recent_written_byte(7..0);
            esac;

    ASSERT
        always {valid_read & register_r_req} |->
            {register_r_data(data_bit_high..data_bit_low) =
                most_recent_written_byte(7..0)};

}
-- uncomment the case branch if needed
-- Replace external_input pattern occurrence by name of relevant
-- actual input design signal
-- Complete modelling code related to most_recent_written_byte
-- Reproduce this property for each relevant combination
-- of register and byte

vunit reads_from_wo_reg_return_0 {
    -- Reads from wo_reg (a write-only register) always return 0

    inherit register_interface_definitions;
    inherit register_additional_definitions;
    inherit requests_not_retracted;
    inherit request_data_stable;

    ASSERT
        always {wo_reg_read} |->
            {register_r_data(upper_data_bit..0) = 0};

}

```

```

-- Reproduce this property for each write_only register

--vunit writes_to_reg_i_byte_j_bit_k_affect_outsig1 {
--
-- Writes affect output signals
--
-- inherit register_interface_definitions;
-- inherit register_additional_definitions;
-- inherit requests_not_retracted;
-- inherit request_data_stable;
--
--
-- DEFINE
--   valid_write := reg_i_write & byte_j & valid_operation;
--
-- ASSERT
--   forall b in boolean:
--     always {valid_write & (register_data(bit_k) = b)}
--       | => {outsig1 = b};
--
--}
-- Uncomment this vunit if needed
-- Replace outsig1 pattern occurrences by name of relevant actual
-- output design signal
-- Replace bit_k occurrences by actual design bit position
-- Change ASSERTion code if needed
-- Reproduce this property for each relevant combination
-- of register, byte, bit and output signal

--vunit outsig2_change_due_to_reg_i_byte_j_write {
--
-- Changes in output signals solely due to writes
--
-- inherit register_interface_definitions;
-- inherit register_additional_definitions;
-- inherit requests_not_retracted;
-- inherit request_data_stable;
--
--
-- DEFINE
--   valid_write := reg_i_write & byte_j & valid_operation;
--
-- ASSERT
--   always {outsig2 != prev(outsig2)} | -> {prev(valid_write)};
--
--}
-- Uncomment this vunit if needed
-- Replace outsig2 pattern occurrences by name of relevant actual
-- output design signal
-- Reproduce this property for each relevant combination
-- of register, byte and output signal

--vunit outsig3_from_reg_i_and_insig {
--
-- Output signals derived from current register data and inputs
--
-- inherit register_interface_definitions;
-- inherit register_additional_definitions;
-- inherit requests_not_retracted;

```

```

-- inherit request_data_stable;
--
--
-- DEFINE
--     valid_write := reg_i_write & byte_j & valid_operation;
--
-- VAR
--     most_recent_written_byte(7..0) : boolean;
--
-- ASSIGN
--     init(most_recent_written_byte(7..0)) := 0; -- reset value
--     next(most_recent_written_byte(7..0)) :=
--         case
--             valid_write & register_r_req :
--                 register_data(data_bit_high..data_bit_low);
--             external_input : ...;
--             else : most_recent_written_byte(7..0);
--         esac;
--
-- ASSERT
--     always(most_recent_written_byte(4) & insig) = outsig3);
--
--}
-- Uncomment this vunit if needed
-- Replace external_input pattern occurrence by name of relevant
-- actual input design signal
-- Complete modelling code related to most_recent_written_byte
-- Change ASSERTion code as appropriate
-- Reproduce this property for each relevant combination
-- of register, byte and input signal

```