Multilevel Hardware Generation and Verification

Master Thesis

Produced at
Chair of Electronic Design Automation,
Department of Electrical and Computer Engineering
Technische Universität Kaiserslautern, Germany

by
Harald Ovsthus

Kaiserslautern, 2020

Supervisors: Prof. Dr.-Ing. Wolfgang Kunz
Dipl.-Ing. Tobias Ludwig

Certification

I hereby declare that this submission is my own work and that I only used the resources specified.

Kaiserslautern, March 17, 2020

Contents

Chapter 1

Introduction

The end of Moore's law is a challenge that chipmakers and manufacturers have known of for at least a decade. It is a driving factor for the increase in *System on Chip* (SoC) complexity. When the number of transistors per cm^2 cannot be increased at the same rate, other alternatives are used to increase performance. Among such alternatives are multiple cores and hardware accelerators. As a result todays SoC's must consist of more and more subsystems of increasing size which subsequently lead to a increasingly vast design space. Over the years much development has been made in describing complex hardware systems at the *Electronic System Level* (ESL). There are some well established methods in using software to represent hardware, a notable one being SystemC with *Transaction Level Modelling* (TLM). The strength of TLM is flexiility and speed, the latter being a result of abstraction. Simulating a large design at the ESL takes orders of magnitude less time than at the *Register Transfer Level* (RTL). Many design decisions can be hedged at this early design stage, along with the detection of numerous bugs. These ESL descriptions already lead to a substantial decrease of the time to market of new products. However, despite these strengths it is the RTL that remains the point of reference for creating the golden design model. The reason for this is the semantic gap between the two abstraction layers ESL and RTL. There is simply not enough trust in the ESL description so functional correctness must be fully verified at the RTL. In spite of much progress in both simulation-based and formal verification techniques, verification at the RTL is the main bottleneck in the SoC design flow.

Some recent research proposes a methology to bridge the semantic gap between the two abstraction layers [1]. This research is further developed into a new method of hardware design. With this new method; *Property Driven Development* (PDD), a system can be fully verified at the ESL and a sound relationship between the ESL and RTL is generated in the form of interval properties [2]. This research is the foundation that this thesis is based on, and it will be elaborated further in chapter 2. Due to reasons that will be explained in chapter 2 section 2.2 one need to

follow a certain set of rules for PDD when describing the system at the ESL. The focus of this thesis is to find a way to represent the complicated bus architecture AMBA-AHB for PDD. Furthermore, to reduce future design efforts a generator is made to enable flexible reuse of this work. The most notable related work is obviously the case study that is the basis of this thesis. The case study is concerned with describing a Wishbone bus architecture at the ESL at two different levels of abstraction. Both ESL descriptions have interval properties generated, and each their RTL designed based on these properties. The case study shows that a higher level of abstraction leads to substantially faster simulation speed. Furthermore, the properties from the abstract design are then refined to hold on the less abstract design showing the flexibility of PDD. This feature is what this thesis is based on, representing complex multi level hardware using an abstract ESL. The case study can be found in [2] and its project files in [3]. The Wishbone bus implementation is a single master non-pipelined design whereas the AMBA-AHB is a multi-master pipelined architecture. The full implementation and its challenges is described in chapter 3.

Chapter 2

Theoretical background

In this chapter all the fundamental theory, methology and protocols used in this implementation are elaborated. This is meant to serve as a tool to aid in an intuitive understanding to further clarify the design and its underlying choices. As stated PDD is the research of which this thesis is based. Before delving in to the theory it is important to clarify what *Interval Property Checking* (IPC) is and how it can be used to create a sound relationship between the two abstraction layers with *Path Predicate Abstraction* (PPA). In section 2.1 IPC is described together with necessary additional information. What PPA is and how it is used to bridge the semantic gap is covered in section 2.2. Section 2.3 then explains PDD and the certain rules it imposes on the ESL descriptions. Finally the AMBA-AHB specification is reviewed in section 2.4.

2.1 Interval Property Checking

Formal verification is an approach of functional verification where design specification is formally formulated as a set of properties. Using mathematical algorithms it can be proven that the RTL description fulfills the property set. As opposed to simulation based RTL verification, formal verification can guarantee the absence of bugs in the design. Recent industrial languages such as *Interval Language* (ITL) provide an intuitive alternative to formulate such properties using a formal language such as *Linear Temporal Logic* (LTL). An interval property, or operation property is a pair of assumptions A_t0 and commitments C_t1. The assumptions describe the state and inputs of some RTL cluster over a time t_0 whereas the commitments describe the state and outputs of the same RTL cluster over time t_1. Using a property checker one attempts to prove that if the assumptions hold on the design, the commitments do as well. Here the time variables t_0 and t_1 represent a a time period relative to an arbitraty timepoint t. The property structure is illustrated in figure **??**.

```
1   property grant_master is
2     assume:
3       at t: some_state;
4       at t: request_signal;
5       at t+1: grant_signal;
6
7     prove:
8       at t+2: another_state;
9       at t+2: not(request_signal);
10      at t+2: address_signals;
11  end property;
```

Figure 2.1: Example operation property

Starting from all possible starting states the property checker searches for an input sequence and a path in the *Finite State Machine* (FSM) where the implication of the A_t0, C_t1 pair does not hold. The property checker used in this thesis is Onespin 360 DV, or Onespin for short [4].

2.1.1 Complete Interval Property Checking (C-IPC)

A set of properties only guarantee that the design works according to specification if the appropriate termination criterion is used. This criterion was first defined in [5]. For a property set to qualify as complete some conditions must be fulfilled. The idea is that, starting from reset there are property covering every aspect of the I/O behaviour and transitions through all important states of the design. The termination criterion hereby referred to as completeness is checked using the property checker Onespin. A completeness check is not run automatically on the design, rather a completeness description must be manually created. This description contains all inputs, determination requirements (outputs, state variables) and a property graph connecting the properties as a FSM. It is not generated automatically to help detect faults in the property set rather than having them transferred to the completeness description. The design behaviour can be guaranteed by a series of tests.

1. *Reset test*: It is proven that reset can be applied deterministically and that all outputs and state variables are determined after reset based on listed assumptions.
2. *Successor test*: Proves that the assumptions of all properties are either inputs or state variables determined by a preceeding property. All properties must be properly hooked together, meaning that all assumptions (except inputs) in a property at time t (left hook) is determined by all predecessing properties

at the same timepoint (right hook). This ensures there are no gaps between properties where signals are undefined.

3. *Determination test*: Proves that all state variables are determined at the right hook in every property, and that all relevant outputs are determined at all times. Relevant in this context is meant by the validity of certain data. In communication data may be invalid unless f.ex a flag is set, the completeness checker can be told to ignore this value otherwise by describing it as *if(flag) then determined(data)*.

4. *Case split test*: Proves that there is a property covering every transition between important states of the design. In other words there is a property covering every input scenario in every important state.

As long as only inputs are listed as inputs in the completeness description, and all outputs are listed as determination requirements, the collective hold of all four tests prove completeness. Out of sheer relevance to this thesis key aspects of the process on how to achieve completeness on a design is provided in the following section.

2.1.2 Gap Free Verification

Out of consistency some terms from the previous section are redefined. A state variable is redefined as a *Visible register*. It is a register that stores information used between properties. It is not necessarily used to determine the state of the system. This is where the term *Conceptual state* comes in. Earlier referred to as important states, conceptual states are abstractions of actual states in the designs FSM. It may directly map to a state in the FSM, parts of one, contain many or be a collection of visible registers and inputs of a design. The process of creating a property set that proves completeness is divided in four phases. The specifics of the four phases will not be recounted here, for that the reader is referred to [6]. Rather some implications and key information are re-described.

The process starts with the definition of the *Conceptual State Machine* (CSM) and a set of skeleton properties are made to represent this CSM in the property graph. All visible registers required to correctly represent the assumptions of the properties are added as local determination requirements and determined at right hook in all properties. By making sure all properties only assume inputs and these registers (at correct timepoint) the successor test will hold. Further, all outputs are listed as determination requirements together with the remaining visible registers needed to help determine them, they are subsequently determined in all properties. If this is done correctly the determination test will hold. Outputs are only determined based on inputs and visible registers. If the mapping of a visible register is incorrect or missing, this is sure to appear as an error in the completeness check. If all possible

transitions of the CSM is covered by properties, the case split test will hold. If not, these operations are identified with help of the debugger and added until the case split test holds. The reset test is just a special case of these three tests, and is unlikely to fail if all three hold. When all four tests hold the property set is complete.

Clusters

A design is ideally verified in a single cluster, to keep abstraction at the highest level. In some cases verifying a design in a single cluster is not feasible. The main reason for this is parallelism.

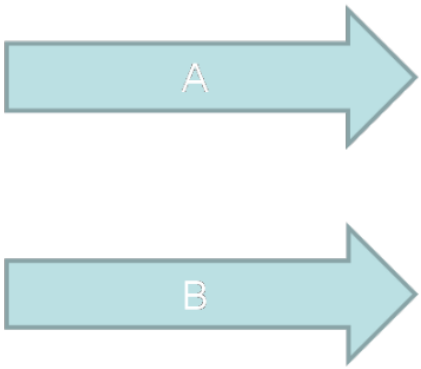


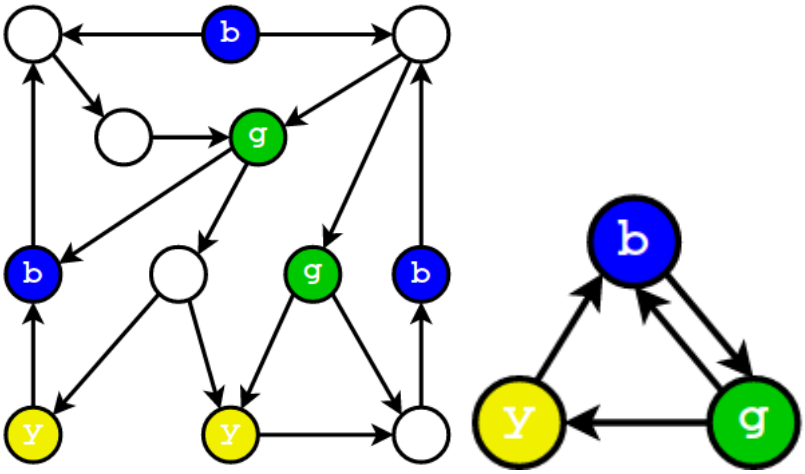
Figure 2.2: Parallel dataflow

Consider two dataflows occuring in parallel. To verify this completely, every scenario must be accounted for. A gets data and not B, B gets data and not A, both A and B or neither. This needs to be accounted for in every clock cycle. In a large design with many operation properties, verification becomes unfeasible even with two parallel data flows.

Where one draws the conceptual border of a cluster, often falls naturally on the interface between to modules. Sometimes however, it is necessary to do a cluster split where the border consists of arbitrary signals in the middle of a module. Any signal can be determined as an input or output to the completeness plan of a cluster[7]. When splitting a design into clusters it is important to verify all clusters completely and that any signal that is defined as an input to a cluster is either an input to the design, or an output from another cluster. When defining the signals that represent inputs and outputs between clusters it is recommended to use the same signal names in both clusters. An example of this is using the top level signal in the RTL design that connects cluster A to cluster B. This is to eliminate the possbility that an output from cluster A actually represent a different signal than its respective input to cluster B. By following these guidelines there are no gaps in verification between the clusters.

2.2 Path Predicate Abstraction

This section reviews the principles of PPA and how it is used to describe RTL at a higher level of abstraction using operation properties. This does not serve as a standalone description of the full theory, for that the reader is referred to [1]. To give an intuitive understanding of the abstraction mechanism PPA is discussed with regard to FSM's. Consider an arbitrary RTL design and its accompanying FSM. This FSM has its own input and output alphabet and describes a possibly complex sequence of operations. PPA is used to simplify this sequence of operations by identifying only the important states, a process illustrated with operational graph coloring.



(a) Operationally colored graph (b) Resulting PPA

With operational coloring one is able to choose quite freely which states to group together in a conceptual state, and which to leave out as "unimportant" states as long as two rules are followed.

1. All cyclic paths must be broken by at least one colored node, there must be no cyclic path in only uncolored nodes.
2. If there is a path between two colored nodes $g \rightarrow y$, there must exist a path to a node of color y from any node of color g.

Figure **??** a) shows the operationally colored FSM of the RTL and b) shows the resulting PPA. It can be easily verified that the color sequence of any path in a) is represented in b). Through abstraction the FSM has been greatly simplified, although some information is lost in the process. It is not possible to extract the original FSM based on the abstraction. An accompanying abstraction of the input and output alphabet of the original must be derived. A communication spanning several nodes in the FSM can now be represented using a single compound in one transition. What before was SDRAM controllers *burst_read* spanning or 8 more

cycles can now be represented as a single operation with $sdram \rightarrow read(compound)$. All transitions between the nodes of the PPA together with the abstract input and output alphabet are formulated in operation properties. This set of properties can now be refined to hold on the original design, and a completeness analysis can be run. As covered in section 2.1.1 this verifies the complete I/O behaviour of the design, and the set of abstract properties can be said to be a sound abstraction of the RTL.

2.3 Property Driven Development

In this section the idea of PDD is described, with the intention of giving an overview of how two cycle accurate communicating RTL modules can be abstracted to an untimed ESL description. For a complete overview of the design flow, case studies and proofs the reader is referred to [2]. Instead this section focuses on the abstraction of communication and what restrictions this puts on the implementation language; SystemC. In the previous section properties were manually described bottom-up based on an existing RTL description. With PDD this process is reversed (top-down) and properties are automatically generated from an ESL description written in this restriced subset of SystemC, referred to as SystemC-PPA. The RTL is then designed based on these properties, in a process similar to test driven development in software. A software tool *DeSCAM* [3] analyses the SystemC-PPA and a complete set of properties representing design behaviour can be printed.

A system model consist of multiple units transferring information between eachother by some means of communication. Communication at the hardware level is divided in two main categories, asynchronous and synchronous. In asynchronous communication transmission is enabled through event signalling, where the receiving end must always be ready to receive such an event. Synchronous communication is enabled through use of a common clock, it is understood through explicit timing that the receiver is ready. There is also the case of unilateral synchronization, where only one communication partner sends such an event. It must then be guaranteed through timing that the other partner is ready. The abstract system model consists of asynchronous PPA's which communicate with eachother using events. In SystemC-PPA this communication is done over channels using port interfaces.

2.3.1 Port interfaces

In SystemC-PPA there is three port interfaces available, two of which model the asynchronous and unilateral synchronization mentioned above. The last provide a means of modelling a volatile memory.

1. *Blocking*: The blocking port models asynchronous communication through a four phase handshake. To modules communicate using $blocking \rightarrow read()$ and $blocking \rightarrow write()$ where the module calling the function is blocked until the other party signals a synchronization signal. This is implemented in SystemC using *event* and $wait(event)$. This behaviour is represented in the properties by use of $notify$ and *sync* signals for both parties. Each blocking write or read implies an important state and two properties, one for transfer and one for wait. The wait property proves the system is halted; no state, visible registers or outputs are modified while *sync* is de-asserted. These event notifiers must be implemented in RTL to satisfy both properties, which carries some overhead. To guarantee that no transmission is lost, each notify/sync must be de-asserted in turn between each transfer. There is also a non-blocking alternative available within the interface, which does not guarantee transmission without explicit timing constraints. As a result system is not halted and a wait property is not generated when using the non-blocking alternative. A non-blocking write will however, in any case hand away control by calling a $wait(arbitrarytime)$ after every write. The read on the other only calls such a wait if there is no writer blocked on the port.

2. *MasterSlave*: To model the unilateral synchronization a master/slave interface models the case where a master can initiate a transfer at any point without regard. The transfer is guaranteed here because the slave will always be ready to respond to the request. The master will issue a synchronization signal and its SystemC-PPA can be modelled rather freely. On the slave side certain rules apply which $DeSCAM$ will check for. All slave ports must be written in every run and no port can be written twice before all other have been and no slave module can use a blocking interface.

3. *Shared*: The shared port implements no event synchronization and calls no $wait()$ function. It is meant for modelling unordered input data like sensor values. It can be useful to provide additional information in combination with one of the other interfaces.

2.4 The AMBA-AHB

This section is a shorthand guide to the AMBA AHB and its specification to aid in understanding the aspects of the protocol and architecture used in this implementation. For the full AMBA AHB specification refer to [8]. The AMBA AHB is a pipelined high performance bus architecture supporting multiple masters and slaves. Important aspects of the AMBA AHB are highlighted:

Bus master(s)	Can when granted, initiate a transfer, a read or write in the form of a burst or as a single transfer. Multiple master can not transfer concurrently.
Bus slave(s)	Responds to the granted master by reading its control signals in one phase, and responding in the next.
Arbiter	Chooses which master gets a bus grant by using a chosen arbitration scheme. Relevant schemes are fixed priority and round robin. The arbiter controls the address/control and write data mux
Decoder	Decodes the address, selects appropriate slave and controls the response data mux.
Default slave	Response given when no valid slave is selected, it is usually integrated in the decoder.
Default Master	The arbiter grants a selected default master the bus when idle.

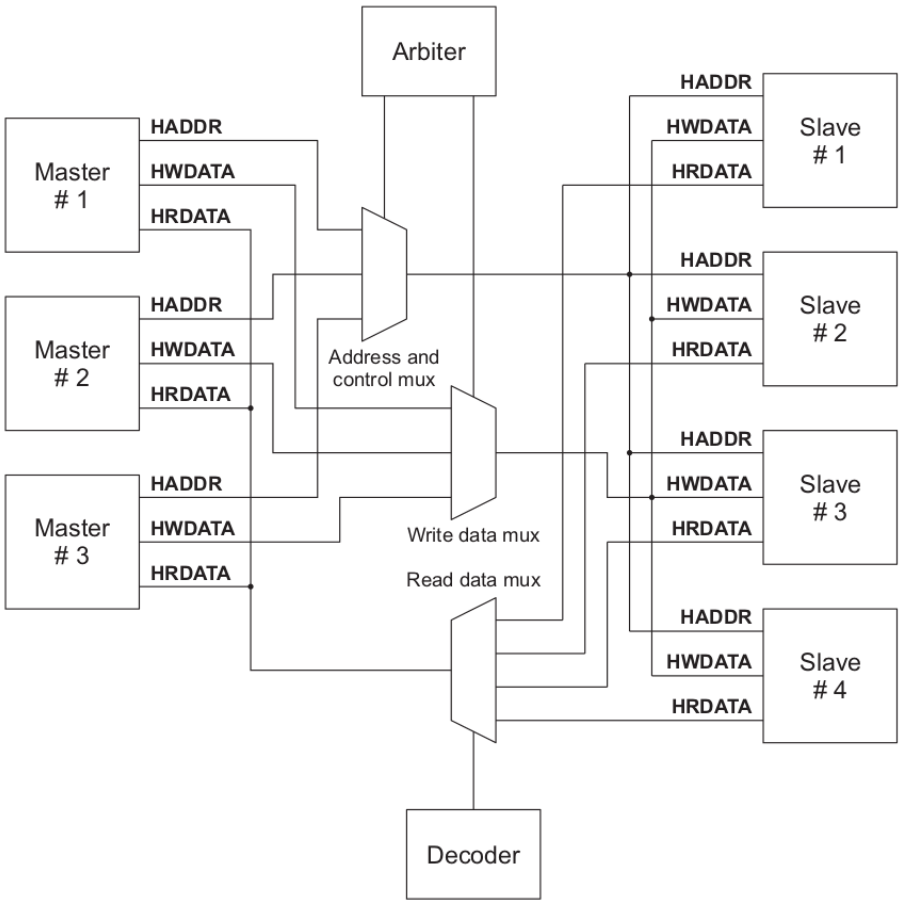
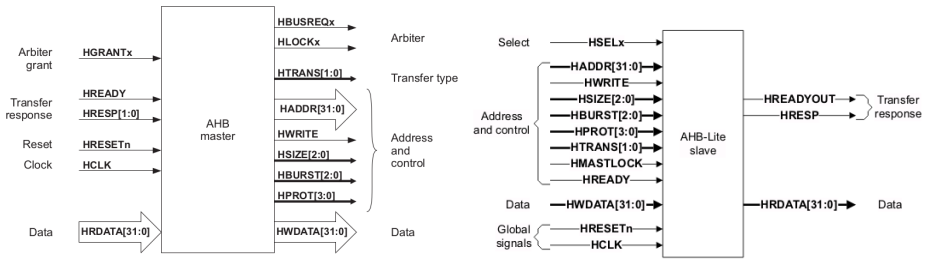


Figure 2.4: An overview of AHB interconnect, reprinted from [8]. The mux outputs are referred to as address or data buses.



(a) Master signals (b) Slave signals

Figure 2.5: Master and slave signals, reprint from left:[8] right:[9]. AHBLite slaves are compatible with AHB systems[10]. **HRESP[1:1]** is therefore hardwired to 0.

As figure 2.4 illustrates, the selected address/control and data signals are broadcast to all receivers simultaneously. On the slave side every signal is ignored unless the **HSELx** signal is set by the decoder. Slave **x** then reacts to the **HTRANS[1:0]** and **HREADY** control signal. Since the master is the initiator it will only react to appropriate signals when expected.

2.4.1 Transfers

Overview of transfer

A master requests the bus by asserting its **HBUSREQx** signal. The master waits until **HREADY** and its **HGRANTx** is set and provides appropriate address and control signals in the next cycle. The transfer is now in the address phase, all set values must be kept valid until **HREADY** is set. The transfer is now in the data phase. If the transfer is a write, the master must provide valid data and keep it valid until **HREADY** is set. Otherwise it is a read, and the slave does not need to provide valid data until it sets **HREADY**. Master samples the data when **HREADY** is set.

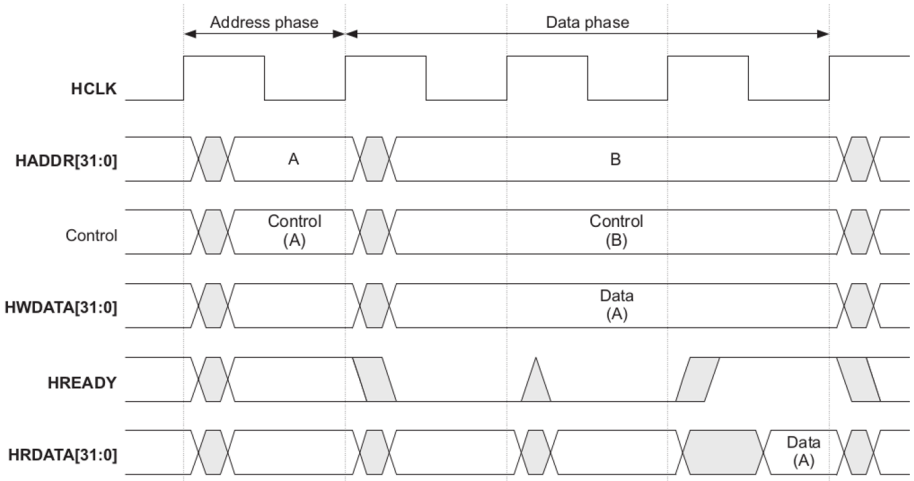


Figure 2.6: A transfer with wait states, modified reprint from [8]. In this document the letters represent different masters

In figure 2.6 master A initiated a transfer, and the slave extends the data phase to give itself time to handle the request. When the slave is finished it responds using **HREADYOUT**, **HRESP** and if it is a read request, **HRDATA**, see **??**. As seen from the figure, extending the data phase for A has a side effect of extending the address phase for B. B is the address bus owner, but any selected slave knows not to react to its control signals. **HREADYOUT** is routed back to the slaves input as **HREADY** through the address mux, and must be set for the slave to sample address and control. For this reason only **HREADY** is referred to for the remainder of this document.

2.4.2 Control signals

The following signals determine which actions the slaves perfom:

HTRANS[1:0]	**Type**	**Description**
00	*idle*	No transfer required, used when a master is granted the bus but does not wish to initiate a transfer. Selected slave must always provide a zero wait state *okay* response and ignore all other signals.
01	*busy*	Used by master to insert idle cycles in the middle of a burst sequence.
10	*nonseq*	Indicates the start of a transfer, address and control is unrelated to the previous transfer. Single transfers are treated as burst of one, the transfer type is therefore nonsequential
11	*seq*	The remaining transfers in a burst are sequential.

Table 2.1: Transfer type

HWRITE indicates direction of transfer, a write request is performed when set.

HSIZE[2:0]	**Size**	**Description**
000	8 bits	*Byte*
001	16 bits	*Halfword*
010	32 bits	*Word*
remainder	>32 bits	*not implemented.*

Table 2.2: Data width

2.4.3 Slave responses

After a transfer has been started only the active slave has the ability to end it. The active uses **HREADY** in combination with **HRESP** to signal the status of the transfer. The slave can either extend and complete the transfer, or provide a two cycle error response.

If the address provided on the address bus is outside the range of any existing slave the default slave response must be provided. If the encoding on **HTRANS[1:0]** is *idle* or *busy* default slave must provide a zero cycle okay response. Otherwise the default slave must provide the two cycle error response.

HRESP	HREADY	Description
0	0	Wait state
0	1	Transfer complete/Okay response
1	0	First cycle of error response
1	1	Second cycle of error response

Table 2.3: Slave responses

Chapter 3

Implementation

This chapter details the implementation of the AMBA-AHB at the ESL together with its proof that it is a sound abstraction of the RTL. As with the Wishbone bus, the protocol can not be directly implemented without the use of so called agents. They allow for flexibility by allowing any master/slave using a blocking port to be connected to the bus without explicitly having to implement the protocol themselves. The AMBA-AHB is a complicated protocol/architecture which would require extensive design effort and testing to implement from scratch. For this reason it was decided to use a trusted existing open source implementation of the interconnect, including arbiter and decoder [11]. The existing architecture and the agents are described in section 3.1. The biggest challenge in representing the AHB with SystemC-PPA is when multiple masters are introduced to the design. The masters are restricted by arbitration, but are otherwise free to act independently from eachother. This leads to a great deal of parallelism and an exponential increase in possible outputs/states for each added master, if the system is viewed as a whole. Just to paint the picture, consider the FSM in figure 3.2. The transition $IdlePhase \rightarrow RequestPhase$ has 4 possibilities for two masters. However, if one master transitions then the possibility of transition $IdlePhase \rightarrow RequestPhase$ for the other master must be accounted for in each clock cycle throughout the whole FSM. The full ESL description and what this challenge entails is covered in section 3.2. The refinement of the formal properties generated for the design is described in section 3.2.2. System simulations at both levels are described in section 3.3 with the generator introduced in section 3.4.

3.1 Hardware overview

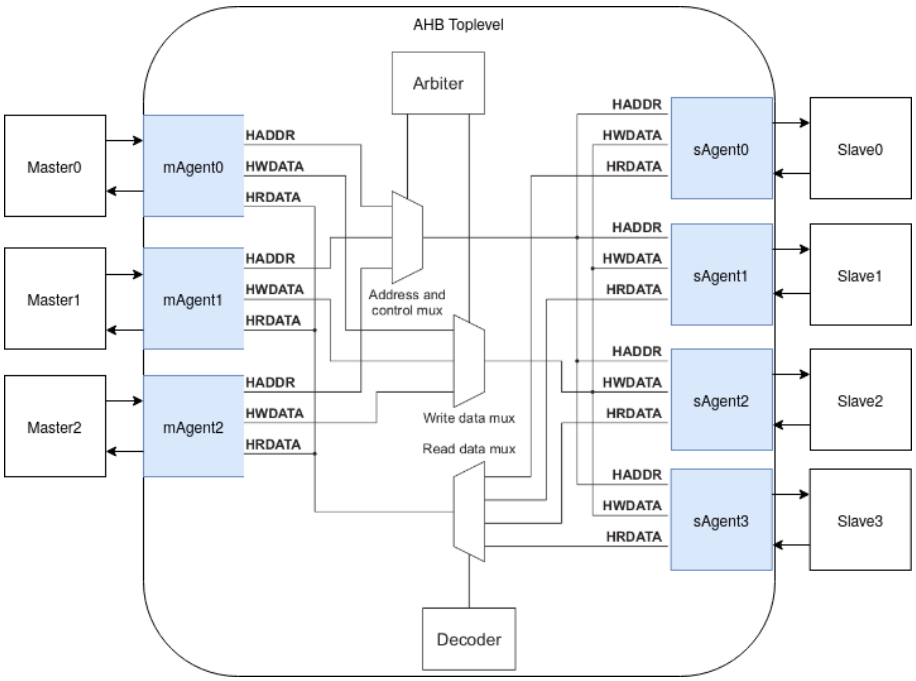


Figure 3.1: Hardware toplevel with 3 masters and 3 slaves connected

3.1.1 Existing hardware

The existing hardware is a stripped and slighlty modified version of the ahb system generator [11]. Only the modules concerning the arbiter, decoder and interconnect are taken from this design. This architecture is a trusted implementation of the AHB protocol, and is often referred to when students inquire online about the AHB implementation. It is designed to be generated for a selected number of master and slaves so it is perfect for the purpose of this thesis. It provides the option to select between six modes of arbitration; Fixed, Round Robin and random. The remaining three are the modified versions of these. With the modified version a grant is not taken away when first given, if the request remains asserted. This implementation will be using Fixed modified, to simplify a possible implementation of burst transfers. The maximum number of masters and slaves on this arhcitecture is 15 each. This hardware is what connects the agents together in figure 3.1 and can be referred to as the AHB- or bus matrix. A bottom up abstraction was done on this design, which lead to the discovery of an erroneous slave respone from the default slave. The design provided a constant error signal on **HRESP** regardless of encoding on **HTRANS[1:0]**. While it still provided the two cycle error response properly, it did not provide the zero cycle okay response as described in section 2.4.3. This may go

undetected in simulation if no master is explicitly dependent on the zero cycle okay response being correct. In this case it would greatly complicate verification, for this reason it was corrected.

3.1.2 Master agent

The master agent receives a payload representing a subset of the AHB master interface from the master. Parts of this subset is defined with different types than the original interface to enhance readability and abstraction on the ESL. Table **??** highlights these changes.

Signal	**Type**	**Content**
HADDR	unsigned int	32-bit
HWDATA	unsigned int	32-bit
HWRITE	enum	AHB_READ, AHB_WRITE
HSIZE	enum	MT_B (byte), MT_H (halfword), MT_W (word)

Table 3.1: Master out payload

The reason behind using a subset is that the remainder of the signals are either hardwired or provided by the agent. The response data payload consists of **HRDATA** and **HRESP**. **HRESP** is included in the payload at a cost of latency for consecutive writes from the same master. Instead of providing new address and data the agent needs to report success to the master. Reporting success back to the master is crucial since the master could possibly attempt to write to an unknown location. Repeated attemps or dependencies may lead to system crash or deadlock.

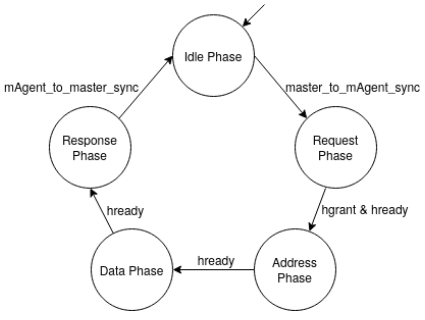


Figure 3.2: Master agent FSM

As seen from figure 3.2 extra states are added to the system to enable the communication between the agent and the master. The agent will always be ready for a payload from the master, and when received it advances to the *Request Phase*. At this point the agent has already translated and written the entire payload to the bus. In the *Request Phase* **HBUSREQx** remains asserted until both **HGRANTx** and **HREADY** is set. When both signals are set the agent proceeds to the *Address Phase* where it encodes *nonsec* on its **HTRANS[1:0]** until **HREADY** is set. Otherwise **HTRANS[1:0]** will always be encoded with *idle*. The agent proceeds to the *Data Phase* where it waits until the

HREADY is again set before sampling **HRDATA** and **HRESP**. It proceeds to the *Response Phase* where it writes the payload to its master and returns to *Idle Phase*.

The entire payload is written to the bus already in the Request Phase to enable a higher level of abstraction in the bus matrix. This way it is not necessary to keep track of which master is granted at all timepoints, greatly increasing abstraction. It can safely be done as described so long as the encoding of **HTRANS[1:0]** is used in a correct manner. The only overhead is that it reduces throughput, but this is already reduced as a consequence of the blocking port combined with the need to report status back to the master. In a system with bursts implemented, the encoding of **HTRANS[1:0]** will also include *seq*, but this feature is explored in chapter (ref something). Because of the decision to report status back to master in every transfer, the response data is sampled regardless of the transfer direction. This reduces ESL and verification complexity significantly, at the cost of one extra clock cycle latency for reads.

3.1.3 Slave agent

Similarily to the master agent, only a subset of the AHB slave interface is written from the slave agent to the slave. This is the same payload as master agents.

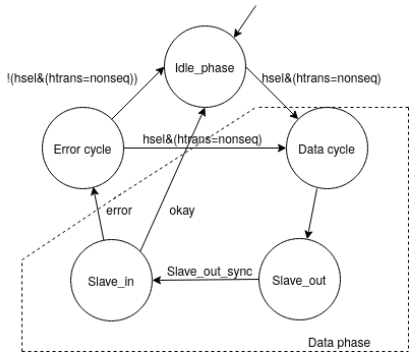


Figure 3.3: Master agent FSM

As figure 3.3 shows the slave agent FSM is more complex than that of the master agent. To make the slave agent comply with the AHB protocol it needs to both obey the rules of the transfer phases and provide the proper response while communicating with its slave. A slave agent is always ready to receive a payload from its master, namely the AHB matrix. Starting from the *idle_phase* the agent stays idle until it is both selected and the encoding of *nonseq* is detected on its inputs. When true it samples address and control signals from the AHB matrix and proceeds to the *data cycle* where it samples the data. After sampling the data, the payload is written to the slave out port named *sAgent_to_slave*. At this point the agent waits for a handshake, which should normally occur instantly but that is not a requirement. After the handshake is received it proceeds to wait for the response data and status from the slave, as with the output there is not a strict requirement on the wait time but it is recommended to keep the wait states under 16 cycles in total. The slave agent deasserts **HREADY** throughout its entire conceptual data phase, and zeroes **HRDATA** one clock cycle after asserting **HREADY**.

The slave agent may at any given timepoint be operated by another master. This is why the slave deasserts its **HREADY** throughout its entire conceptual data phase. It is standard for a slave to introduce wait states. As with a DRAM module, there are delays associated with activating a row (insert data). As with DRAM this delay is minimized in sequental address transfers using bursts. In contrast to the diagram in figure 2.6, the conceptual data phase of the slave agent does not include the assertion of **HREADY**. This is a slight misnomer since the data phase does infact include the assertion as seen from the properties. It is however not included in the FSM to not diffuse the differences between the states. The *idle_phase* and *error cycle* only differentiate in the value of **HRESP**. Although the assertion of **HREADY** technically is a part of the data phase, it is however beneficial to highlight that these are seperate states, as they are represented as such in the RTL description. The zeroing of **HRDATA** is a design choice to increase abstraction.

3.2 System level representation

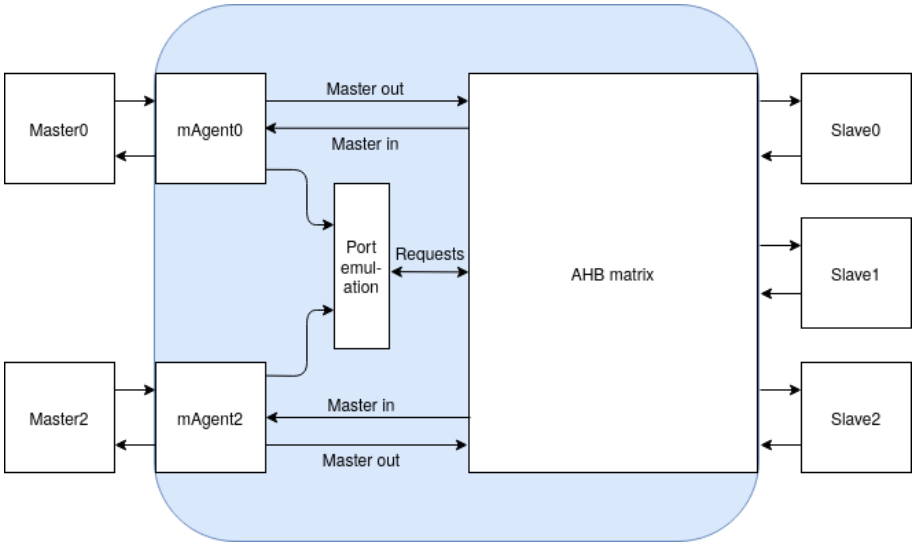


Figure 3.4: ESL toplevel with 2 masters and 3 slaves connected

Describing the AHB system with SystemC-PPA in a single cluster is no longer feasible when more than one master is connected to the bus matrix. The master agents must be contained within their own clusters, or PPAs so that it does not need to be accounted for every possible combination. Since only one slave may be operated at a time, including them in the bus matrix PPA reduces the overhead. The interface between the master agent and the bus matrix have signals with mostly identical values for all masters, the exception is **HGRANTx**. This signal is purely combinational in the implementation, meaning that it changes its value within the same clock cycle as its initiator. Using any combination of existing ports there was not found any manner to represent this interface for multiple masters. The decision was made to create the properties bottom-up, to determine if the interface could be represented using existing ports and if not, where the gap is. Gap free verification was used for this purpose, with the style of generated properties in mind.

3.2.1 Bottom-up abstraction

The first challenge of carrying out the GFV process is determining the CSM of the AHB. It has to be represented in such a way that it is feasible to describe at the ESL. Looking at other formal verification efforts made on the AHB [**?**], it can be seen that the real FSM of an AHB is incredibly complex. Furthermore, it is described how the state of the bus is dependent not just on state and inputs but on previous states as well. It is necessary to know the state of the bus, namely if there is a transfer being carried out or not. It is not as simple as paying attention to bus ownership

since there is no way of knowing if there is a transfer in progress when the default master owns the data bus, without looking at past values on **HTRANS[1:0]**. One would furthermore need to constrain the design to establish how far into the past to look. One could go with the suggested maximum delay of slaves for 16 clock cycles. A much simpler solution is to define the master agents states as outputs to their clusters, and define this as inputs to the main cluster. By only concerning with which masters are requesting and where the data goes, the CSM can be made quite simple. If one was concerned with which master owned the address and which master owned the data bus at all times it would lead to state explosion, making an ESL description unfeasible even for a low number of masters.

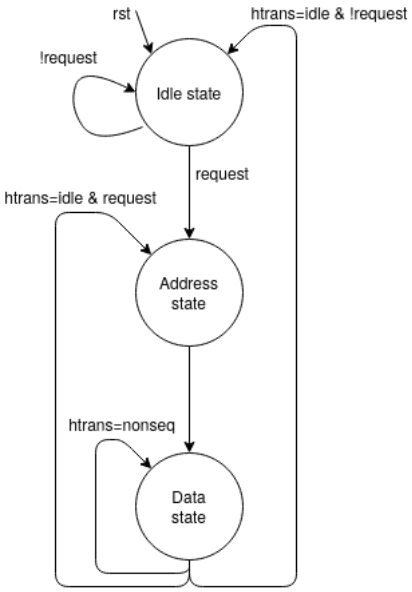


Figure 3.5: Abstract CSM

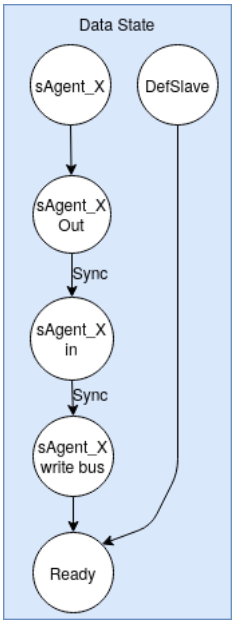


Figure 3.6: Detailed data state CSM

By using a fixed priority arbitration scheme it is simple to determine which master is granted the bus. By dividing the transfer into three main states with the data state divided into five further, there is no need to impose any constraints on wait states. The bus is ready (**HREADY** is set) at the end of each state and any request is ignored unless the bus is ready. The states can be defined as follows:

- Idle state: No transfer in progress.
- Address state: One master is in the address phase of the transfer, no masters are in the data phase. There is no trigger for the next state as the bus/slaves will always be ready here.
- Data state: One master is in the data phase of the transfer. This state is broken down into five substates per slave, with default slave as exception.

It is worth noting that in the idle and address state the bus/slaves are always ready.

The payload is transferred between the states using visible register which are named *AS_regs* for the address state and *DS_regs* for the data state. The Data state is divided into several important states to describe the transaction between the slave and its agent while simultaneously describing all necessary outputs. The default slave has only two states and they are self explanatory because of the two cycle error response. The remaining slave agents are explained by the following:

- First state: There is one clock cycle delay required to allow the slave agent time to sample the possible data provided in the data phase.
- Second state: Slave agent writes payload to slave using blocking write.
- Third state: Slave agent reads back from slave using blocking read.
- Fourth state: Slave writes payload back to bus. This requires two cycles due to the possibility of error, error cycle was removed from master agent due to inconsistencies in simulation. For this reason information on error is only contained within the visible registers handling response.
- Fifth state: The bus is ready, payload is sampled, requests are handled and next state is determined based on the value of **HTRANS[1:0]**, which has been added to *AS_regs* alongside the payload.

It is easy to see that the differentiation between read and write within the AHB matrix would lead to the double amount of data states, with the gain being one clock cycle saved in the case of a read. The entire design is represented by this CSM using only properties with length $t = 1$ so determining the output is straightforward with the exception of **HGRANTx**. As this is an output that is not stored in a register its value updates in the same clock cycle as its instigator. This is not a problem to model in the properties themselves, but this representation does not allow for any sound abstraction between the RTL and ESL. Determining the value of **HGRANT** in the next clock cycle is not feasible in this implementation. One would have to account for every **HBUSREQx** as an assumption at $t + 1$ in every property. An alternative could be to enable this output through a register but this would lead to unpredictable behaviour. This output is simply determined as the output of a function *mx_grant* and is at the ESL represented using an enum.

The design choice to zero **HRDATA** and modify the default slave response entails that **HRDATA** and **HRESP** will be zero unless it is in the fourth or fifth concepceptual data state. After adding all determination requirements and proving completeness the properties show that the interface between the master agents and AHB matrix cannot be represented using a single existing SystemC-PPA port. Referring back to section 2.3.1 there is a choice between three ports. For clarity the implications of using each port is commented.

- *Shared*: Explicit synchronization is needed to ensure that the value of **HREADY**

and **HGRANTx** is neither overwritten nor obsolete. A solution was experimented with to highlight the increased abstraction. It was not successful without introducing illegal statemets with respect to SystemC-PPA, even when disregarding the combinational response of **HGRANTx**.

- *MasterSlave*: With MasterSlave some implicit synchronization is enabled. However, in this system one would be left with one of two choices. The bus matrix is the slave, or the master agents are the slaves. In either case the values of **HGRANTX** and **HREADY** would need to be representing correct values in every cycle. One side must always be ready for communication, so neither alternative is worth concidering.
- *Blocking*: The blocking port can be used to represent all signals functionally, but with blocking ports alone there are some issues. On the master agent side **HBUSREQx**, **HGRANTx** and **HREADY** can be represented both functionally and in the properties by the notify and wait used for synchronization. On the bus matrix side the functionality could be represented, but not the properties due to the states implied by each port. They all require their own state, and in turn a separate clock cycle to assert and deassert these signals.

The remaining option is to emulate a single compound port using a combination of shared and blocking ports.

Combinatory port emulation

When examining the FSM in figure 3.2 it is seen that the traversal of the master agent state machine is always blocked by an input signal. The *Request-*, *Address-* and *Data_phase* are blocked by the AHB matrix's response signals **HREADY** and **HGRANTx**. Although the value of **HGRANTx** is impossible to determine effectively in the next clock cycle, its functionality with respect to the state machine can still be properly represented using a blocking port. The handshake from the master agents side represent the request, whereas the handshake from the AHB matrix side represent **HGRANTx** and **HREADY**. Due to the pipeline nature of the bus, a request can occur at any main state of the bus as seen from figure 3.6. This creates the requirement of a separate output representing **HREADY** alone, to allow for state machine traversal after bus has been granted. Only one port read can be represented in each clock cycle for blocking ports. Due to each master agent requiring their own port for each of the cases, it is necessary to contain these ports in a seperate module, and treat it as a new type of port interface. This port determines internally which master agent gets granted/unblocked with a simple fixed priority arbitration scheme. The final issue is having the port correctly represent updated requests. For this reason the port waits for a synchronization signal (handshake) from the AHB matrix. When this handshake is received the port peeks on all its request inputs and forwards this information to the AHB matrix

through a *Shared* interface, while simultaneously unblocking the highest priority requesting agent and every **HREADY** port that may be blocked. Special care has been taken to ensure that this happens in an atomic and synchronized manner.

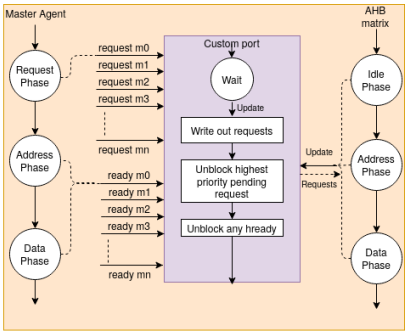


Figure 3.7: Illustration of port connection

The port in figure 3.7 is illustrated using the actual port directions in the ESL description, raher than the direction they symbolize. The inputs representing **HREADY** must be atomically unblocked which only happens when there is a pending handshake. To check for this the *peek* function must be called, which is only available for reader ports. On the AHB matrix side the synchronization call to update the requests must always hand over control to the port by use of a wait function, which is only unconditionally called by use of a write port. When control is handed back to the AHB matrix the updated requests can be fetched from a shared port.

Master agent

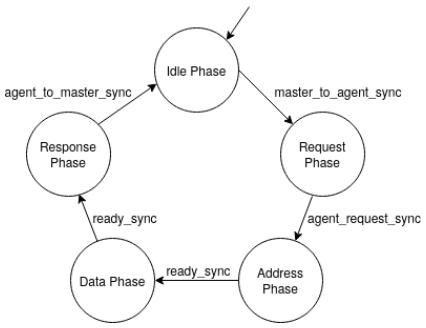


Figure 3.8: master agent FSM

The master agent is at the ESL in a quite simple manner using blocking ports to determine important states. All states from figure 3.2 are deemed important states. They are all needed to communicate with the individual states of the bus matrix, or their masters. Transitions between the states are dependant on the synchronization signals alone, and all remaining interface signals are passed through shared ports. The generated properties are refined with minimal effort and proven to be a sound abstraction of the master agent RTL.

Bus matrix

The bus matrix is a slighly more complex implementation, as can be seen from figure 3.6. All shared outputs are written, or determined in all of the states to enable the generated properties to satisfy the completeness criterion. The remaining signals to

the interface are implicitly determined at all times by the blocking port. To aid in understanding section 3.2.2 an overview of important signals is provided.

Signal	Type	Description
update_requests	blocking_out	Uses non blocking write to signal the emulated port to update requests.
requests_in	shared_in	A compound of all pending requests provided by the emulated port
AS_regs	visible reg	The payload from the master but including **HTRANS[1:0]**. Stores the values of granted agent in its address phase.
DS_regs	Visible reg	The payload from the master, stored for any master in its data phase.
resp	Visible reg	stores response from the slave.

Table 3.2: Important signals

3.2.2 Refining properties

3.3 Simulation

3.3.1 Starvation

3.4 Generator

Chapter 4

Summary and outlook

List of Tables

List of Figures

Bibliography

[1] Joakim Urdahl, Dominik Stoffel, and Wolfgang Kunz. Path predicate abstraction for sound system-level models of RT-level circuit designs. *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 33(2):291–304, Feb. 2014.

[2] T. Ludwig, J. Urdahl, D. Stoffel, and W. Kunz. Properties first – correct-by-construction rtl design in system-level design flows. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2019. ISSN 1937-4151. doi: 10.1109/TCAD.2019.2921319.

[3] Tobias Ludwig. *Design from SystemC Abstract Models*. Techniche Universität Kaiserslautern.

[4] Onespin Solutions GmbH. OneSpin 360 DV-Verify. https://www.onespin.com/products/360-dv-verify/.

[5] Jörg Bormann and Holger Busch. Verfahren zur Bestimmung der Güte einer Menge von Eigenschaften (Method for determining the quality of a set of properties). European Patent Application, Publication Number EP1764715, 09 2005.

[6] Gapfreeverification guide (itl). Onespin documentation, 2020.

[7] Gapfreeverification guide (itl). Onespin documentation, 2020.

[8] Amba specification. Available at: https://developer.arm.com/docs/ihi0011/a/amba-specification-rev-20, 1999. Accessed: 20/02/2020.

[9] Amba 3 ahblite protocol. Available at: http://www.eecs.umich.edu/courses/eecs373/readings/ARM$_I HI0033A_A MBA_A HB-Lite_S PEC.pdf$, 2006. *Accessed* : 21/02/2020.

[10] Amba design kit. Available at: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dd 2007. Accessed: 20/02/2020.

[11] Ahb system generator. https://opencores.org/projects/ahb$_s ystem_g enerator. Accessed$: $2019 - 11 - 12$.