Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Synthesis of AMBA AHB from Formal Specification: A Case Study

Yashdeep Godhal, Krishnendu Chatterjee, Thomas A. Henzinger

IST Austria (Institute of Science and Technology Austria)

Abstract. The standard hardware design flow involves: (a) design of an integrated circuit using a hardware description language; (b) extensive functional and formal verification; and (c) logical synthesis. However, the above mentioned processes consume significant effort and time. An alternative approach is to use a formal specification language as a high-level hardware description language and synthesize hardware from formal specifications. Our work is a case study of the synthesis of the widely and industrially used AMBA AHB protocol from formal specifications. Bloem et. al. presented the first formal specifications for the AMBA AHB Arbiter and synthesized the AHB Arbiter circuit. However, in the first formal specification some important assumptions were missing. Our contributions are as follows: (1) We present detailed formal specifications for the AHB Arbiter incorporating the missing details, and obtain significant improvements in the synthesis results (both with respect to the number of gates in the synthesized circuit and with respect to the time taken to synthesize the circuit); and (2) we present formal specifications to generate compact circuits for the remaining two main components of AMBA AHB, namely, AHB Master and AHB Slave. Thus with systematic description we are able to automatically and completely synthesize an important and widely used industrial protocol.

1 Introduction

Hardware design flow. In traditional hardware design procedure, the first step is the description of a designed circuit in hardware description language. The first step is followed by extensive verification and logical synthesis. The outcome of logical synthesis is a netlist i.e., gate level implementation of the circuit. Among the above steps, the verification step is the most time consuming process and requires a lot of effort. An alternative approach for the design flow is to automatically synthesize the circuit from a formal specification for the circuit.

Synthesis from formal specification. Historically, automatic synthesis of digital designs from temporal logic specifications has been considered as one of the most challenging problems in circuit design. The problem was first presented by Church [4] and several methods have been proposed as solutions such as in [3] and in [13]. The problem was considered again in [12] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL). The method proposed in [12] for a given LTL specification φ starts by constructing a non-deterministic Büchi automaton which is converted into a deterministic Rabin automaton. This translation may require a doubly exponential complexity in the size of φ. The high complexity established in [12] caused synthesis to be deemed hopelessly intractable and discouraged practitioners from attempting to use it for system development. Yet, if the specification of the design is restricted to simpler automata or partial fragments of LTL, it has been shown that the synthesis problem can be solved in polynomial time. Major progress has been achieved in [11], which shows that designs can be automatically synthesized from LTL formulas belonging to the class of generalized reactivity of rank 1 (GR(1)), in time N^3, where N is the size of the state space of the design. The class GR(1) covers a large class of properties that arise in specifications of circuits. The approach of [11]

was implemented by Bloem. et al [1] in a tool called Anzu [6]. Anzu produces not only a BDD representing a set of possible implementations, but also an actual circuit.

AMBA AHB Protocol. In this work we study the automatic synthesis of an important and widely used industrial protocol, namely, AMBA AHB protocol. ARM's *Advanced Microcontroller Bus Architecture* (AMBA) [10] specification defines an on chip communications standard for designing high-performance embedded microcontrollers. AMBA is today the de-facto standard for embedded processors because it is well documented and can be used without royalties. It is widely used in network interconnect chips, RAM controllers, Flash memory controllers, DMA controllers, level-2 cache controllers and SoCs(System on Chip) including application processors used in portable mobile devices like smartphones. A few industrial examples of its use are the IXP42X Product Line of Intel Network Processors, the Infineon gateway controller ADM5120. The most important bus defined within the AMBA specification is the *Advanced High-performance Bus (AHB)*. The AHB acts as the high-performance system backbone bus. AHB supports the efficient connection of processors, on-chip memories, DMA controllers and off-chip external memory interfaces. Thus AMBA AHB is a widely used industrial bus protocol. AMBA AHB system consists of the following main components: (a) AHB Arbiter; (b) AHB Master and (c) AHB Slave. In this work we have synthesized the above three components of the AMBA AHB protocol.

Our contributions. Bloem et al. presented the first formal specifications for AMBA AHB Arbiter and synthesized the AHB Arbiter circuit [2,1]. However, in the first formal specification some important details were missing. The first is that the HTRANS signal, which plays an important role in AHB transfers, was not considered, and the second is related to the de-assertion of the HLOCK signal. Though the result was correct, the synthesized circuit was not optimized with respect to the specifications because some important assumptions were missing. We show that by completing the specification and considering all assumptions, significant improvement is achieved in the synthesis results. The contributions of this work are as follows:

1. We present formal specifications for a complete AHB Arbiter, and obtain a significant (order of magnitude) improvement in the synthesis results (both in the size of synthesized circuit and the time taken for synthesis). We use the same tool (Anzu) as used in [2,1]. For example, in the case of ten arbiters, the synthesis time required by our new specification is around thirty times faster, and the synthesized circuit is around fifteen times smaller as compared to [2,1]. The synthesis results of [2, 1] could handle upto ten masters, whereas our synthesis results can handle sixteen masters (the maximum number of masters specified by the AMBA AHB standard).
2. We present, for the first time, the formal specifications for AMBA AHB Master and AMBA AHB Slave (which are the two remaining main components of the protocol). We are able to synthesize very compact circuits from formal specifications. Thus we are able to completely synthesize an important and widely used industrial protocol from its formal specifications.

From the lessons learnt in the process of rewriting specifications, we discuss a few principles that were useful for writing the specifications for efficient synthesis.

Input, output and capabilities shown. Our input is the specification of the AMBA AHB protocol in GR(1), which is a subset of LTL. The output is synthesized circuit. Our results show how the GR(1) subset of LTL can be used to efficiently synthesize completely an industrially used protocol, if the complete specifications are written systematically. Our synthesis results are obtained using the Anzu tool [6], which implements a symbolic synthesis algorithm for GR(1), and our contributions are mainly the specifications of the AMBA AHB protocol in GR(1).

Organization. Our paper is organized as follows. In Section 2 we present the preliminaries, the basic definitions, and the classical theoretical results for synthesis. In Section 3 we describe the AMBA AHB protocol and its components. In Sections 4, 5 and 6 we present the specifications and the synthesis results for AMBA AHB Arbiter, Master and Slave, respectively. We end with a discussion of our results and lessons learned in Section 7.

2 Preliminaries

In this section we present preliminaries related to property specification language and synthesis. The definitions of this section are standard definitions of specification and synthesis, and hence the section is similar to the definitions in [2,1].

2.1 Property Specification Language

Property Specification Language (PSL) is a specification language to express temporal logic spec-

ifications (a detailed description of PSL is available in [5]). In this paper we present specifications following the familiar LTL style notations. In particular, we use G, F, and X to denote `always`, `eventually`, and `next`, respectively. The `until_` operator requires the first operand to hold either forever or up to and including the time that the second operand holds. Thus the temporal logic formula (Φ `before` Ψ) is equivalent to ($\neg\Psi$ `until_` Φ). Along with the traditional operators, we use one additional operator (`until_`$[i]$) that is not in PSL: (Φ `until_`$[i]$ Ψ) means that Φ holds either forever or up to and including the i^{th} time that Ψ holds.

2.2 Synthesis of GR(1) Properties

We first recall the basic results related to synthesis of GR(1) properties from [11]. We consider the question of *realizability* of PSL specifications (cf [12]). Consider two sets of Boolean variables, namely, X and Y, where X is the set of input variables controlled by the environment and Y is the set of system variables. The *realizability* questions is to determine whether there exists an *open controller* that satisfies the specification, where an open controller is an automaton which, at every step, reads values of the X variables and outputs values for the Y variables.

We focus on a subset of PSL with efficient algorithm for the realizability (synthesis) question. The specifications we consider are of the form: $\phi = \phi^e \rightarrow \phi^s$. We require that ϕ^α for $\alpha \in \{e, s\}$ can be rewritten as a conjunction of the following parts.

1. ϕ_i^α - a Boolean formula which characterizes the initial states of the implementation.
2. ϕ_t^α - a formula of the form $\wedge_i$(`always` B_i), where each B_i is a Boolean combination of variables from $X \cup Y$; and expressions of the form (`next` v) where $v \in X$ if $\alpha = e$, and $v \in X \cup Y$ otherwise.
3. ϕ_g^α - has the form $\wedge_{i \in I}$ (`always eventually` B_i), where each B_i is a Boolean formula.

We augment the set of variables with *deterministic monitors* to express formulas of other forms, such as `always`$(p \rightarrow (q$ `until_` $r))$, where p, q, and r are Boolean variables. A deterministic monitor is a variable whose behavior is deterministic according to the choice of the inputs and the outputs. The monitors follow the truth value of the expression nested inside the `always` operator. We rewrite these types of formulas to the form (`always eventually` b), where b is a Boolean formula using the variables of the monitor. We note that even with the restrictions, all possible (finite state) designs can be expressed as a set of properties [1].

The realizability problem for PSL formulas is reduced to the decision problem of determining the winner of a two-player game on a graph, where the players are the system and the environment, respectively. The objective of the system is to satisfy the specification irrespective of the behavior of the environment. A *game structure* is a multigraph whose nodes are all the truth assignments to X and Y. The set of edges are as follows: a node v_1 is connected (by an edge) to all the nodes v_2 such that the truth assignments to X and Y satisfy $\phi_t^e \wedge \phi_t^s$, where v_1 supplies the assignments to the current values and v_2 to the next values. We then group all the edges that agree on the assignment of X in v_2 to one multi-edge. A play in the game graph is as follows: (a) in the start step, the environment chooses an assignment to X and the system chooses a state in $\phi_i^e \wedge \phi_i^s$ that agrees with this assignment; and (b) in every following step of the play, the environment chooses a multi-edge and the system chooses one of the nodes connected to this multi-edge. The system wins if the interaction produces an infinite play that satisfies $\phi_g^e \rightarrow \phi_g^s$.

We *solve* the game to decide whether the game is winning for the environment or the system. By determinacy of the games, either the environment or the system has a winning strategy. If the environment has a winning strategy, then the specification is *unrealizable*. If the system has a winning strategy, then the game solving algorithm *synthesizes* a winning strategy. The winning strategy, represented as a BDD, is a nondeterministic representation of an implementation that realizes the specification. We summarize the result of synthesis of PSL specifications in the following theorem.

Theorem 1 *[11] Let X and Y be sets of variables controlled by the environment and the system, respectively. Consider a PSL formula ϕ of the form $\phi^e \rightarrow \phi^s$ with m conjuncts for ϕ^e and n conjuncts for ϕ^s. There is a symbolic algorithm to determine whether ϕ is realizable in time proportional to $O(m \cdot n \cdot 2^{d+|X|+|Y|})$, where d is the number of variables added by the monitors for ϕ.*

2.3 Generating circuits from BDDs

We recall the basic results from [2] related to generating circuits from BDDs. The winning strategy is a BDD over the variables X, Y, X' and Y', where X are input variables, Y are output variables and the primed versions represent the next state variables. The corresponding circuit contains $|X| + |Y|$ flipflops to store the values of

the inputs and outputs in the last clock tick. Let $I = X \cup Y \cup X^{'}$ and $O = Y^{'}$. In every step, the circuit reads the next input values $X^{'}$ and determines the next output values using combinational logic with inputs I and outputs O. The strategy does not prescribe a unique combinational output for every combinational input. In most cases, multiple outputs are possible, in states that are not reachable (assuming that the system follows the strategy), no outputs may be allowed.

We write $o \in O$ for a combinational output and $i \in I$ for a combinational input. We denote by $O \backslash o$ the set of combinational outputs excluding output o, and we denote the strategy as S. For all combinational outputs $o \in O$ we construct a function f in terms of the combinational inputs I that is compatible with the given strategy BDD. The algorithm proceeds through the combinational outputs $o \in O$ one by one: First, build $S^{'}$ to get a BDD that restricts only o in terms of I. Then the *positive and negative cofactors* (p, n) of $S^{'}$ are built with respect to o, that is, the algorithms finds the sets of inputs for which o can be 1 (0, respectively). For the inputs that occur in the positive and in the negative cofactors, both values are allowed. The combinational inputs that are neither in the positive nor in the negative cofactor are outside of the winning region. The combinational inputs outside the winning region represent situations that cannot occur given the environment satisfies the assumptions. Thus, f has to be 1 in $p \wedge \neg n$ and 0 in $\neg p \wedge n$, which give us the set of care states. We minimize the positive cofactors with the care set to obtain the function f. Finally, we substitute variable o in S by f (the substitution is necessary as combinational outputs may be related) and then proceed with the next variable.

The resulting circuit for the specification is obtained by writing the BDDs for the functions using CUDD's DumpBlif command [14]. The resulting circuit is then optimized using the tool ABC [15] and mapped to a library of standard cells. The tool ABC is also used to estimate the number of gates required for the circuit.

3 AMBA AHB Protocol

In this section we present the main components of the AMBA AHB protocol. ARM's *Advanced Microcontroller Bus Architecture* (AMBA) [10] specification defines an on-chip communications standard for designing high-performance embedded microcontrollers. The most important bus defined within the AMBA specification is *Advanced High-performance Bus* (AHB). The AHB acts as the high-performance system backbone bus. AHB supports the efficient connection of processors, on-chip memories, DMA controllers and off-chip external memory interfaces. The AMBA AHB design consists of the following components:

3.1 AHB Master

A bus master initiates read and write operations by providing address and control information. Only one bus master is allowed to use the bus actively at any given time. Hence, before initiating any transfer, it sends a request to the arbiter for accessing the bus. Once the arbiter grants master the access (to the bus), the master initiates read/write operation. Master 0 is the *default master* and is selected whenever there are no requests for the bus.

3.2 AHB Arbiter

A bus arbiter ensures that at one time only one bus master is allowed to initiate data transfers. Every bus master has a REQUEST/GRANT interface to the bus arbiter. The arbiter uses a prioritization scheme to decide which bus master is the highest priority master requesting access to the bus. Each master generates $\mathrm{HLOCK_x}$ signal which indicates that the bus master requires exclusive access to the bus. The arbitration protocol is not specified and can be defined differently for each application.

3.3 AHB Decoder

The decoder in an AMBA system is used to perform a centralized address decoding function. It provides one select signal to each slave in the system. If the input address to the decoder belongs to the address range specified for a particular AHB slave, then the select signal from the decoder to that slave would be high. Hence the function of an AHB decoder is similar to a de-multiplexer.

3.4 AHB Slave

A bus slave responds to transfers initiated by bus masters within the system. The slave uses a select signal $\mathrm{HSEL_x}$ from the decoder to determine when it should respond to a bus transfer. All other signals required for the transfer, such as the address and control information, are generated by the bus master. The bus slave signals back to the active master of the success, failure or waiting status of the data transfer.

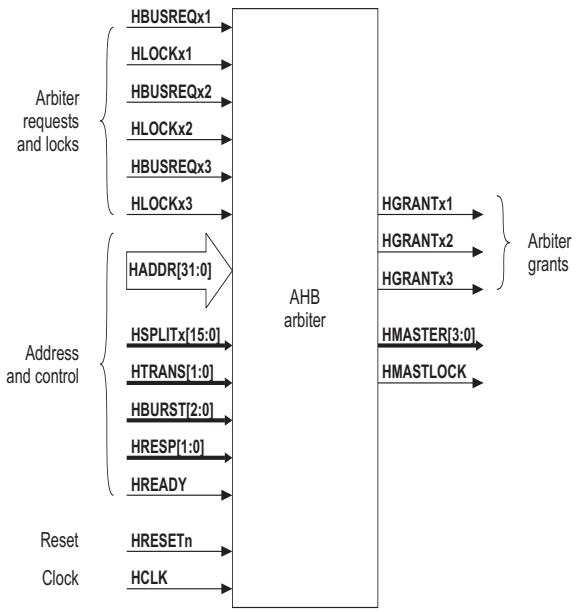

Fig. 1: AHB Arbiter [10]

The AHB is a pipelined bus. It means that different masters can be in different stages of communication. A bus access can be a *single* transfer or a *burst*, which consists of a specified or unspecified number of transfers. Access to the bus is controlled by the arbiter. All devices that are connected to the bus are Moore machines, that is, the reaction of a device to an action at time t can only be seen by the other devices at time $t + 1$. For a system with single slave, the select signal shall always be high, if a valid address is put on bus.

4 AMBA AHB Arbiter Synthesis

In this section we present our results related to synthesis of the AHB arbiter. We first present the arbiter signals, followed by formal specifications and synthesis results.

4.1 AHB Arbiter Signals

Figure 1 displays AHB arbiter signals. The description of these signals are as follows (the notation S[n:0] denotes an (n+1)-bit signal):

1. HBUSREQ_i - This signal from bus master i to the bus arbiter indicates that the bus master requests access to the bus.
2. HLOCK_i - This signal is output from a bus master i and input to the bus arbiter. If asserted, this signal indicates that the bus master requires locked access to the bus. No other master should be granted access to the bus until this signal is lowered.
3. HREADY - This signal is driven by the bus slave and it is input to the bus arbiter. This signal is lowered to extend a transfer. When asserted, it indicates that the slave is ready to accept the transfer.
4. HGRANT_i - This signal is output of the bus arbiter and input to a bus master i. When asserted, it means that the bus master i has been granted access to the bus. It also indicates that if HREADY is high, then HMASTER= i will hold in the next clock tick.
5. HMASTLOCK - This output from the arbiter indicates that the current bus master is performing a locked sequence of transfers.
6. HMASTER[3:0] - These signals from the arbiter indicate which bus master is currently performing a transfer.

The following signals are multiplexed using HMASTER as the control signal. For example, although every master has an address bus, only the address provided by the currently active master is visible on HADDR.

1. HADDR[31:0] - These signals indicate the address where read or write transaction takes place.
2. HBURST[1:0] - These signals inform about nature of transfer. It can be a single transfer (SINGLE), burst of four transfers (INCR4), unspecified length burst (INCR).
3. HTRANS[1:0] - These signals indicate the type of the current transfer, which can be NONSEQ, SEQ or IDLE.

4.2 Formal Specifications

The first set of formal specifications for AMBA AHB arbiter was given in [1]. We have (i) rewritten some of the specifications for efficient synthesis with meaning kept intact, and (ii) more importantly, included important components of the specifications that were missed in [1].

Important changes. The two important changes in our specification are as follows:

1. The HTRANS[1:0] signal, that plays an important role in AHB transfers, was not used in earlier specifications. With use of the HTRANS signal, we are able to make the formal specifications more compact.
2. The other important change from the specifications of [1] is related to de-assertion of HLOCK signal. According to ARM [7], the AHB Master should de-assert the HLOCK signal when the

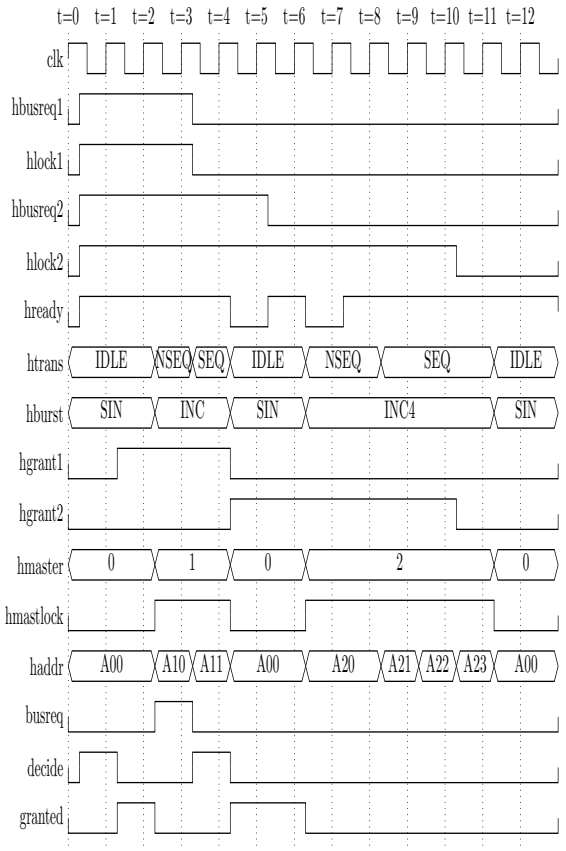

Fig. 2: Signals for the AHB Arbiter and timing diagram

address phase of the last transfer in the locked sequence has started. This is included in our formal specification.

Along with the signals described above, we have used two auxiliary signals DECIDE and BUSREQ, that were introduced in [2]. The signal DECIDE indicates the time slot in which the arbiter decides who the next master will be and whether its access will be locked. The decision is based on HBUSREQ_i and HLOCK_i. The signal BUSREQ points to the HBUSREQ_i signal of the master that currently owns the bus. Two auxiliary variables START and LOCKED, that were introduced in [1], are not used in our specification. Due to inclusion of the HTRANS signal and change of the nature in de-assertion of HLOCK signal, the START and LOCKED variables have become redundant. We have introduced a new auxiliary variable GRANTED which is driven by the arbiter. The signal GRANTED is used for deciding start of new access. When both GRANTED and HREADY signals are high simultaneously, new access shall start in next cycle. Thus a decision can be executed at the earliest two time steps after the HBUSREQ_i and HLOCK_i signals are read.

Table 1: Sample PSL Specifications for AHB Arbiter.

A4	`always` (¬HREADY → (HTRANS = 0 ↔ `next` HTRANS = 0)) `always` (¬HREADY → (HBURST = 0 ↔ `next` HBURST = 0))
A5	`always` ((HTRANS = IDLE) → (`next` (HTRANS ≠ SEQ)))
A6	`always` (((HTRANS = NONSEQ) ∧ (HBURST = INCR4) ∧ HREADY) → (`next` (HTRANS = SEQ)))
G2	`always` ((HMASTLOCK ∧ (HBURST = INCR) ∧ HREADY ∧ (HTRANS = NONSEQ)) → `next` ((HTRANS = SEQ) `until_` ¬BUSREQ))
G7	`always` ((HREADY ∧($\vee_{\text{i=0}}^{\text{n-1}}$ (HLOCK_i ∧ HGRANT_i)) → `next` (HMASTLOCK)))
G8	$\forall i$: `always` ((¬HREADY∨ ¬GRANTED) → (HMASTER= i↔ `next` HMASTER= i)) `always` ((¬HREADY∨ ¬GRANTED) → (HMASTLOCK↔ `next` HMASTLOCK))

We follow the convention used in [1]- guarantees are properties that the arbiter must fulfill, and assumptions are properties that the arbiters environment must fulfill. Our specification for the arbiter, that adheres to the AMBA AHB standard consists of *9 assumptions* and *12 guarantees*, whereas the specification from paper [1] had *4 assumptions* and *11 guarantees*.

Figure 2 shows the timing diagram for AHB arbiter signals and illustrates the behavior of the auxiliary signals with respect to other signals. Table 1 contains some sample assumptions and guarantees for the arbiter in PSL (for complete PSL specifications of the arbiter, refer to Table 7 in the appendix). The bold faced **A** and **G** signify new/re-written property whereas non-bold faced indicate existing property from [2]. The assumptions(A) and guarantees(G) for the arbiter are described below.

Assumptions. We present the assumptions below, and with each assumption we mention how the assumption is obtained directly from the AMBA AHB standard.

A1 During a locked burst transfer of unspecified length, leaving HBUSREQ_i high locks the bus. (The assumption A1 is a formal specification obtained from [8]). (G1 is used to convert HBUSREQ_i to BUSREQ in Table 7).

A2 Leaving HREADY low locks the bus, the standard forbids it. (The assumption A2 is the formal specification obtained from the property described in page 59 of [10]).

A3 Signals $HLOCK_i$ and $HBUSREQ_i$ are asserted by AHB Master at the same time (i.e., these signals go from low to high at the same time). (The assumption A3 is the formal specification that is obtained from the property described in page 62 of [10]).

A4 When HREADY signal is low, control signals i.e., HBURST and HTRANS should hold their values. (The assumption A4 is the formal specification obtained from page 42 of [10]).

A5 If no transfer is taking place, the HTRANS signal cannot become SEQ in the next cycle. (The assumption A5 is the formal specification obtained from page 43 of [10]).

A6 In burst sequence (i.e., HBURST = INCR4), if HREADY is high, the NONSEQ transfer shall always be followed by SEQ transfer. (The assumption A6 is obtained from page 42 of [10]).

A7 The first transfer of any AHB sequence is NONSEQ in nature. (The assumption A7 is obtained from page 42 of [10]).

A8 When no AHB Masters is making a request for bus, no transfer will take place. (The assumption A8 is obtained from [9]).

A9 All input signals are low initially. This assumption is valid because at power up, all hardware signals are at reset values.

Guarantees. The guarantees are as follows.

G1 Variable BUSREQ points to $HBUSREQ_i$ of the master that is currently granted access to the bus.

G2 When a locked unspecified length burst starts, a new access does not start until the current master (i) releases the bus by lowering HBUSREQi. (This guarantee is obtained from page 62 of [10]).

G3 When a length-four locked burst starts, no other accesses start until the end of the burst. We can transfer data only when HREADY is high, so the current burst ends at the fourth occurrence of HREADY. (This guarantee is obtained from page 62 of [10]).

G4 If there is at least one bus request present and signal DECIDE is high, then GRANTED gets asserted in next cycle. (This guarantee is based on information from page 63 of [10]).

G5 If HREADY and GRANTED signals are simultaneously high, then GRANTED gets deasserted in next cycle. If GRANTED signal is high and HREADY is low, then GRANTED signal holds its value in next cycle. (This guarantee is obtained from page 64 of [10]).

G6 The HMASTER signal follows the grants: When HREADY is high, HMASTER is set to the master that is currently granted. It implies that no two grants may be high simultaneously and the arbiter cannot change HMASTER without giving a grant. (This guarantee is obtained from page 64 of [10]).

G7 Whenever signal HREADY, $HLOCK_i$ and $HGRANT_i$ are simultaneously high, HMASTLOCK gets asserted in the following cycle. (This guarantee is obtained from page 62 of [10]).

G8 When any of GRANTED or HREADY signals is low, the HMASTER and HMASTLOCK signals do not change.

G9 Whenever DECIDE is low, $HGRANT_i$ signal do not change.

G10 We do not grant the bus without a request, except to Master 0. If there are no requests, the bus is granted to Master 0. (This guarantee is obtained from page 63 of [10]).

G11 We have a fair bus i.e., every master that has made a request shall be serviced eventually.

G12 The signals DECIDE and HGRANT0 are high at first clock tick and all others are low.

Description of changes. Assumptions A1, A2, A3, A9 and Guarantees G1, G2, G3, G6, G8, G9, G10, G11, G12 mentioned above are taken directly from [1]. Remaining guarantees in [1] were related to auxiliary signals which have become redundant in our case with inclusion of HTRANS signal. Out of the above, G2, G3 and G8 have been re-written with the original meaning kept intact. The rewriting was achieved as follows:

1. The property G2 is mentioned in [1] as follows: `always` ((HMASTLOCK $\wedge$ (HBURST = INCR) $\wedge$ HREADY $\wedge$ (START)) $\rightarrow$ `next` ((¬ START) `until_` ¬BUSREQ)).
   In [1], in the formal specification of the properties G2 and G3, the authors have used an auxiliary signal "START" which is no longer required in our specification due to inclusion of HTRANS signal. The property has been modified as follows in our specification:
   `always` ((HMASTLOCK $\wedge$ (HBURST = INCR) $\wedge$ HREADY $\wedge$ (HTRANS = NONSEQ)) $\rightarrow$ `next` ((HTRANS = SEQ) `until_` ¬BUSREQ)).
   A similar modification is done for the property G3.
2. The property G8 is re-written with the inclusion of GRANTED signal rather than DECIDE. In [1] the property is specified as follows:

Num of Masters	Synthesis time (sec) from Figure 8 in [2]	Synthesis time (sec) for specifications [2] in our experiments	Minimum synthesis time (sec) of the last two columns	Synthesis time(sec) for our specifications
2	2	2	2	1
3	20	22	20	8
4	100	103	100	18
5	200	203	200	56
6	800	677	677	189
7	2400	2696	2400	326
8	12000	7931	7931	181
9	2000	2533	2000	1017
10	19000	18789	18789	948
11				1138
12				1394
13				2339
14				3697
15				4339
16				4284

Table 2: Synthesis time comparison

Num of Masters	Gate count from Fig 9 in [2]	Gate count for specifications [2] in our experiments	Minimum gate count of the last two columns	Gate count for our specifications
2	1000	982	982	244
3	3500	2626	2626	545
4	8500	6801	6801	641
5	11000	9033	9033	1208
6	18000	12448	12448	1269
7	15000	19777	15000	2177
8	36000	NM	36000	2000
9	NA	50012	50012	2524
10	50000	45912	45912	3208
11				3842
12				4433
13				4765
14				4324
15				5392
16				7600

Table 3: Gate count comparison

`always` ((¬DECIDE) → (LOCKED↔ `next` LOCKED)).
For our specification the above property has been re-written as:
`always` ((¬HREADY∨ ¬GRANTED) → (HMASTLOCK↔ `next` HMASTLOCK)).

Thus besides adding missed details, all assumptions and guarantees from [1] are also taken care of in our specifications.

Summary. In essence, we have added missing details about the AHB arbiter (inclusion of HTRANS signal in specifications as per the standard [10] and information about HLOCK signal de-assertion from standard [7]) and we have re-written some properties. As shown above all the assumptions and guarantee are either directly taken from [1] or obtained directly from the AMBA AHB standard. In the following subsection we show that the systematic re-writing of the specification, and addition of the important missing details lead to significant improvement in the synthesis results.

4.3 Synthesis Results

Anzu [6] is used to synthesize circuits from specifications. Table 2 shows comparison of time taken by Anzu to synthesize AHB arbiter for different specifications. Column 1 shows number of masters for which arbiter was synthesized. Column 2 shows data taken from Figure 8 of [2] and Column 3 shows time taken in synthesizing specification from [2] on our machine (2GB RAM). In column 4, we have taken the minimum of column 2 and column 3 to have the best possible estimate of synthesis time for arbiter specifications in [2]. Column 5 shows the time in seconds for the arbiter synthesized using our formal specifications.

The results (Table 2) show that using the earlier specifications from [2], the synthesis procedure fails for more than 10 masters. **Yash: With our specification we can synthesize arbiter serving maximum number of masters given by the standard, 16, in around an hour**. Moreover, our specifications show significant (order of magnitude) improvement over the earlier specification: for example, for arbiter serving 10 masters the synthesis of earlier specifications takes nearly 5 hours, **Yash: whereas our specification is synthesized in less than 16 minutes (thus the new results are around *twenty* times faster)**.

In Table 3, NA corresponds to not available (no corresponding point in Figure 9 in [2]) and NM refers to not mappable by ABC (The tool gives an error that it can not synthesize such a big circuit).

Anzu [6] generates a file in .blif format. This file is mapped by using ABC [15] to standard library. ABC tool is also useful for counting number of gates required to realize the circuit. Table 3 shows comparison of number of gates mapped by ABC for realizing different specifications for

arbiter. Column 1 shows the number of masters for which the arbiter is synthesized. Column 2 shows data taken from Figure 8 in [2] and column 3 shows number of gates mapped by ABC tool on our machine (2GB RAM) for existing specification in [2]. In column 4 , we have taken the minimum of second and third columns to have a best estimate of number of gates for existing specifications. In column 5, gate count for our circuit synthesized from our specification. Table 3 shows that arbiter synthesized using specifications from [2] serving 10 masters has nearly forty-six thousand gates, whereas, the AHB arbiter synthesized with our specifications serving 10 masters has only **Yash: around three thousand gates (nearly *fifteen* times more compact), and even arbiter serving 16 masters needs less than eight thousand gates**.

Comparison with manual implementation. Manual implementation for arbiter serving 10 masters comprises of around 1000 gates (see Figure 9 of [2]). **Yash: The synthesis results for our specifications generated arbiter with 3208 gates for arbiter serving 10 masters**.

Thus by simply re-writing some specifications systematically, and adding missing details (without any change in the synthesis algorithm or the tool) we are not only able to improve the time taken for synthesis, but also significantly improve the gate count of the synthesized circuit by an order of magnitude. Graphical comparisons for arbiters serving different number of masters are shown in Figure 3 and Figure 4. Figure 3 shows comparison for synthesis time whereas Figure 4 depicts comparison for gate count.

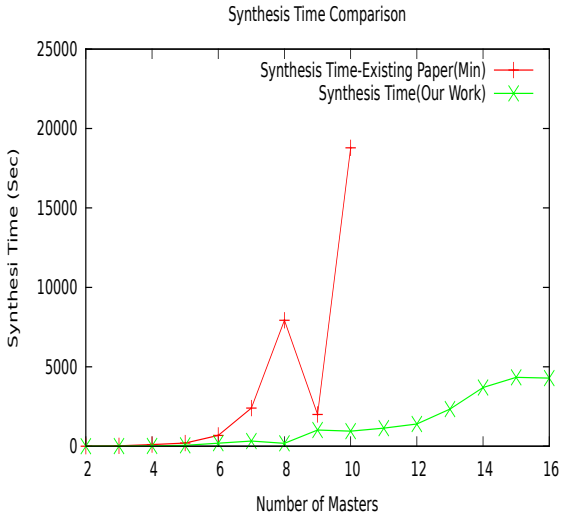

Fig. 3: Synthesis Time Comparison.

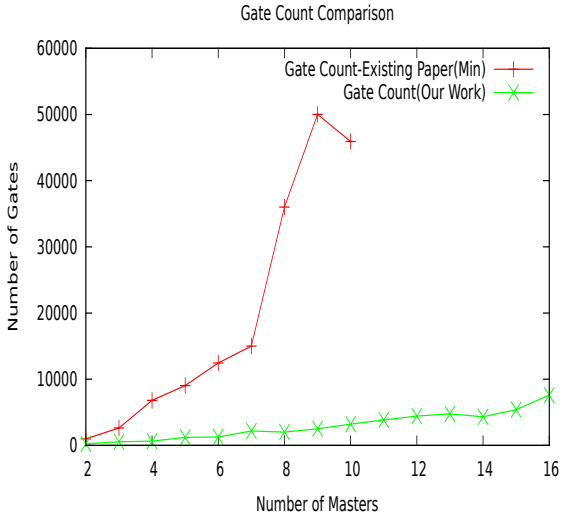

Fig. 4: Gate Comparison.

5 AMBA AHB Master Synthesis

In this section we present the synthesis results for AHB Master: we first present the signals, then the specification, and finally the synthesis results. This module is synthesized for the first time directly from its formal specifications.

5.1 AHB Master Signals

We first introduce the signals for AHB Master that have not been introduced so far.

1. HWRITE - This signal from the bus master indicates the nature of transfer. When HWRITE is low, it indicates read transfer. If high, it indicates write transfer.
2. HADDR[31:0] - These signals from the master provide information about address location where write or read transfer shall take place.
3. HWDATA[31:0] - These signals from the master provide information about data to be written in case of write transaction.
4. HRDATA[31:0] - These signals from the bus slave to the bus master provide information about data read in case of read transaction.
5. HSIZE[2:0] - This signal from the bus master to the bus slave provides information about the bus width. It can be a definite value that corresponds to one of byte(8-bit), half-word(16-bit), word(32-bit) up to 1024 bits. In this work, data bus width is fixed to 32-bit.
6. HRESP[1:0] - This signal from the bus slave to the bus master provides transfer response.

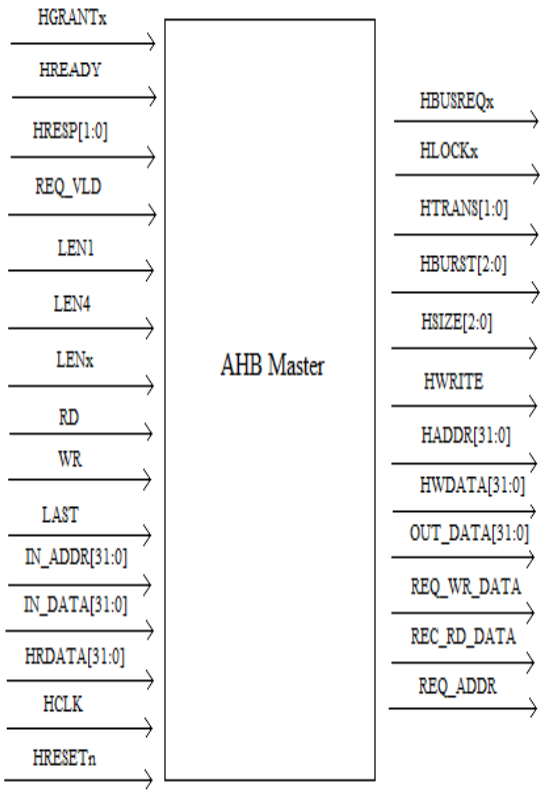

Fig. 5: AHB Master

We introduce a few auxiliary signals. They are as follows:

1. REQ_VLD - This signal is input to the bus master. It is used by the bus master for deciding HBUSREQ. HBUSREQ signal is asserted whenever REQ_VLD is asserted.
2. WR - This signal is input to the bus master. It indicates that write transaction shall take place. HWRITE shall be set HIGH if WR is high.
3. RD - This signal is input to the bus master. If high, it indicates that read transaction shall take place and hence HWRITE shall be set LOW.
4. LEN1 - This signal is input to the bus master. It indicates that single transfer shall take place.
5. LEN4 - This signal is input to the bus master. It informs that the transfer should be a burst sequence of four transfers.
6. LENX - This signal is input to the bus master. It informs that the transfer should be a burst sequence of unspecified length.
7. IN_ADDR[31:0] - These signals are input to the master providing information about address. These signals are used to decide HADDR.
8. IN_DATA[31:0] - These signals are input to the master providing information about write data. These signals are used to decide HWDATA.
9. LAST - This signal is input to the bus master. It indicates the last transfer in a sequence of transfers.
10. OUT_DATA[31:0] - These signals from the master provide information about read data.
11. REQ_ADDR - This signal from the master indicates request for address. If this signal is high, then in the next clock cycle, master shall receive IN_ADDR.
12. REQ_WR_DATA - This signal from the master is request for data. If this signal is high, then in the next clock cycle, master shall receive IN_DATA.
13. REC_RD_DATA - This signal from the master acts as valid signal for read data. If it is high, HRDATA shall be copied to OUT_DATA.

Figure 5 shows the signals of the AHB Master and Figure 6 shows a timing diagram for those signals. The timing diagram shows the behavior of auxiliary signals with respect to the input and the output signals.

5.2 Formal Specifications

In the formal specification of AMBA AHB Master, we have 10 assumptions and 15 guarantees. A sample of PSL specifications of assumptions and guarantees are in Table 4 and Table 5, respectively, and the complete specifications are given in Table 8 and Table 9 in the appendix. The assumptions and guarantees are as follows.

Assumptions. The assumptions are as follows.

A1 Length of transfer will be specified with REQ_VLD signal i.e., whenever REQ_VLD is high, one of LEN1, LEN4 and LENX signal shall be high.

A2 Nature of transfer will be specified with REQ_VLD signal i.e., whenever REQ_VLD signal is high, at least one of RD or WR signal shall be high.

A3 If REQ_VLD signal is low, then RD, WR, LEN1, LEN4 and LENX shall hold their values.

A4 There cannot be conflict between signals indicating nature of transfer, thus RD and WR signal cannot be high simultaneously.

A5 There cannot be conflict between signals indicating length of transfer thus LEN1, LEN4 and LENX signals cannot be high simultaneously.

A6 Input HRESP signal shall be OKAY throughout.

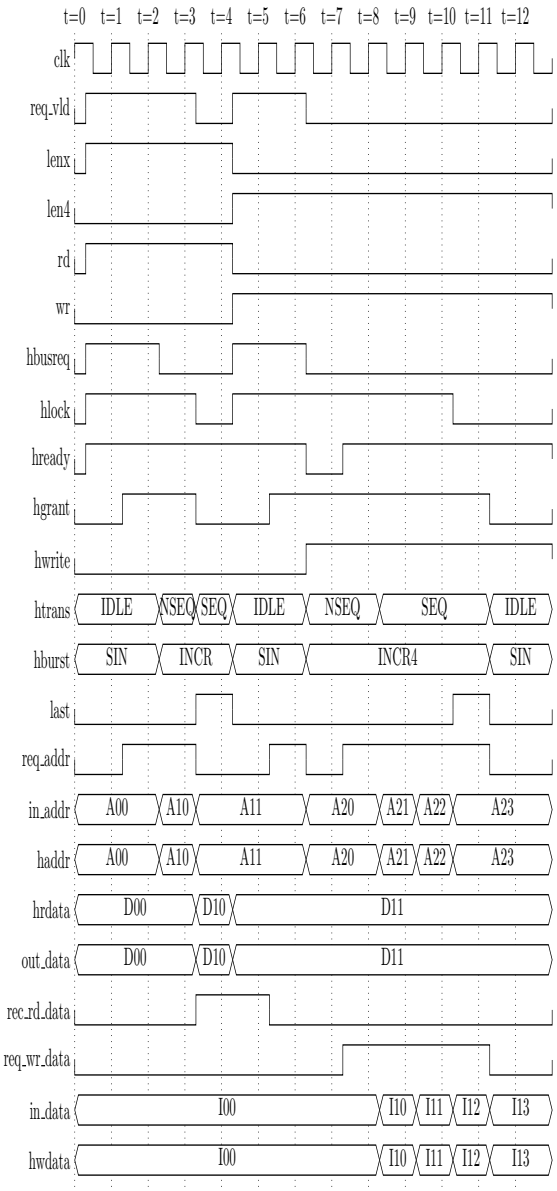

Fig. 6: Signals for the AHB Master

A7 The bus is a fair one, hence every HBUSREQ shall eventually be answered.
A8 During a locked unspecified length burst, leaving HBUSREQ high locks the bus.
A9 Eventually HREADY will be high.
A10 Eventually REQ_VLD and HGRANT signals will be low.

Guarantees. The guarantees are as follows.

G1 Data bus is 32-bit wide. Thus HSIZE shall be fixed to WORD throughout.
G2 HBUSREQ signal gets asserted and deasserted with REQ_VLD.

Table 4: Sample PSL specifications for AHB Master - Assumptions

A4	`always` (WR $\rightarrow \neg$ RD) `always` (RD $\rightarrow \neg$ WR)
A5	`always` (LENX $\rightarrow$ ($\neg$LEN1 $\wedge$ $\neg$LEN4)) `always` (LEN1 $\rightarrow$ ($\neg$LENX $\wedge$ $\neg$LEN4)) `always` (LEN4 $\rightarrow$ ($\neg$LENX $\wedge$ $\neg$LEN1))
A8	`always` ((HLOCK $\wedge$ (HBURST = INCR)) $\rightarrow$ `next eventually` $\neg$REQ_VLD)

G3 Bus master requests only for locked transfer.
G4 If the ongoing transfer is last transfer of an AHB sequence, then HLOCK shall be lowered.
G5 Length four burst (HBURST = INCR4) shall end at fourth occurrence of HREADY.
G6 HBURST shall be set according to length of the transfer indicated by LEN1, LEN4 and LENX.
G7 First transfer of an AHB sequence is always NONSEQ in nature. All following transfers in sequence shall be SEQ in nature.
G8 Nature of transfer shall be set according to WR and RD signals.
G9 If HREADY is low, then all control signals shall hold their values.
G10 When HREADY and HGRANT are simultaneously high, REQ_ADDR signal shall be high. It ensures that in next cycle, master can put address on address bus.
G11 When both REQ_ADDR and WR signals are high, REQ_WR_DATA signal shall also be high. It ensures that data shall be put on data bus one cycle after address is put on address bus.
G12 When a read transfer is taking place and HREADY is high, REC_RD_DATA signal shall also be high.
G13 When REQ_ADDR is high, the input signals IN_ADDR will be copied to address bus in the next cycle.
G14 When REQ_WR_DATA is high, the input signals IN_DATA will be copied to data bus in the next cycle.
G15 When read transaction is in progress and HREADY is high, OUT_DATA will copy the value of HRDATAin the next cycle.

All the assumptions and guarantees are again directly obtained from the AMBA AHB standard.

5.3 Synthesis Results

Yash: The synthesis time for AHB Master is 5 seconds. The generated circuit is mapped

Table 5: Sample PSL Specifications for AHB Master - Guarantees

G3	`always` ((¬HBUSREQ ∧ `next` HBUSREQ ∧ ¬HLOCK) → `next` HLOCK)
G10	`always` ((HREADY ∧ HGRANT) → REQ_ADDR)
G12	`always` ((HREADY ∧ ((HTRANS = NONSEQ) ∨ (HTRANS = SEQ)) ∧¬HWRITE) → REC_RD_DATA)

using ABC tool. It has 146 gates with area 189 square units (units specific to the standard cell library). Thus the synthesized circuit is very small. Thus we are not only able to synthesize the AHB Master from its formal specifications, but the synthesized circuit is also compact. The synthesis results are comparable to manual implementation of AHB Master.

6 AMBA AHB Slave Synthesis

In this section we present the synthesis results for AHB Slave. The synthesis for this module is also performed for the first time directly from its formal specifications.

6.1 AHB Slave Signals

The signals that are useful for AHB slave are already described in previous sections. We have introduced an interface between a slave and a memory so that read and write transactions can be implemented. We are considering memory with two status signals EMPTY and FULL.

Two auxiliary signals have also been added named START and LAST. The START signal indicates start of an AHB transfer or sequence whereas LAST signal is used to indicate last transfer of an AHB sequence.

The signals used in this interface are shown in Figure 7. Figure 8 shows the timing diagram from AHB slave signals. The timing diagram shows the behavior of auxiliary signals with respect to the input and the output signals. The signals used in interface between slave and memory is given below:

1. FULL - This signal is input to the bus slave indicating memory is full. When the memory is full, i.e.. FULL is high, no more data can be written into it without first being read.

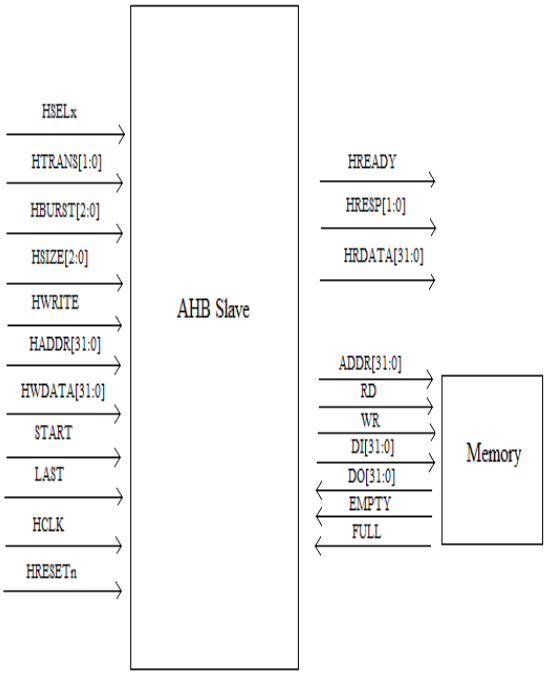

Fig. 7: AHBSlave

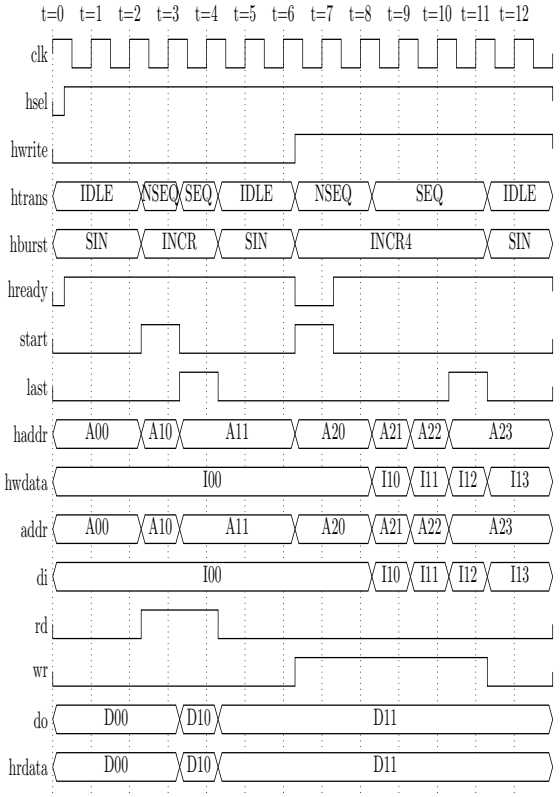

Fig. 8: Signals for the AHB Slave

Table 6: Sample PSL Specifications for AHB Slave

A1	`always` (¬HSEL → ((HTRANS = IDLE)))
A2	`always` ((HTRANS = IDLE) → ((HBURST = SINGLE) ∧ ¬HWRITE ∧ ¬START ∧ ¬LAST))
A4	`always` (¬LAST ∧ (HTRANS = NONSEQ) ∧ HREADY → `next` (HTRANS = SEQ))
A5	`always` ((HLOCK ∧ (HBURST = INCR4) ∧ HREADY ∧ (HTRANS = NONSEQ)) → `next`((HTRANS = SEQ) `until_`[3] HREADY))
G1	`always` (¬HSEL → HREADY)
G5	`always` ((HSEL ∧ FULL ∧ WR) → (HRESP = ERROR)) `always` ((HSEL ∧ EMPTY ∧ RD) → (HRESP = ERROR))
G6	`always` ((HSEL ∧ ((HTRANS = NONSEQ) ∨ (HTRANS = SEQ)) ∧ HWRITE) → WR) `always` ((HSEL ∧ ((HTRANS = NONSEQ) ∨ (HTRANS = SEQ)) ∧¬HWRITE) → RD)

2. EMPTY - This signal is input to the bus slave indicating memory is empty.When memory is empty, i.e., EMPTY is high, no more data can be read from it without first being written.
3. ADDR[31:0] - These signals are output from the slave providing address information.
4. DI[31:0] - These signals are output from the slave and input to the memory providing information about data that should be written into memory.
5. DO[31:0] - These signals are output from the memory and input to the slave providing information about data that has been read from memory.
6. RD - This signal is input to the memory from the slave. It indicates that the read operation is being executed.
7. WR - This signal is input to the memory from the slave. It indicates that the write operation is being executed.

6.2 Formal Specifications

In the formal specification of AMBA AHB Slave, we have 7 assumptions and 9 guarantees. Table 6 gives some sample PSL specifications of the assumptions and guarantees and Table 10 in the appendix gives all PSL specifications. The assumptions and guarantees are as follows.

Assumptions. The assumptions are as follows.

A1 When the slave is not selected by the decoder, all control signals shall be low.

A2 When HTRANS is IDLE, all control signals shall be low.

A3 First transfer of any sequence is NONSEQ in nature.

A4 Non-first transfer of an AHB sequence will always be SEQ in nature.

A5 Burst sequence of length four shall end at fourth occurrence of HREADY.

A6 If this is last transaction of a sequence and next cycle is not start of another sequence, HTRANS shall be IDLE in next cycle.

A7 If HREADY is low, then all control signals, address and data buses shall hold their values.

Guarantees. The guarantees are as follows.

G1 When the slave is not selected by the decoder, HREADY signal shall be high.

G2 When the slave is not selected by the decoder, HRESP shall be OKAY.

G3 When no AHB transaction is taking place, HRESP shall be OKAY.

G4 RD and WR signal cannot be high simultaneously.

G5 If memory is full and write transfer is attempted, then the slave shall send an ERROR response. Similarly, if the memory is empty and a read transfer is attempted, then the slave shall send an ERROR response.

G6 When slave is involved in a transfer, HWRITE is used to decide values of WR and RD.

G7 When slave is involved in any transfer, signal HADDR is used to decide ADDR.

G8 When slave is involved in write transfer, signal HWDATA is used to decide DI.

G9 When slave is involved in read transfer, signal DO is used to decide HRDATA.

All the assumptions and guarantees are obtained directly from the AMBA AHB standard.

6.3 Synthesis Results

Yash: The synthesis time for the AHB Slave is 13 seconds. The circuit generated, when mapped using ABC, has 276 gates with area 545 square units (units specific to the standard cell library) . Thus we are not only able to synthesize the AHB Slave from its formal specifications, but the synthesized circuit is also compact. The synthesis results are comparable to manual implementation of AHB Slave.

7 Discussion

In this section we discuss about our experimentation to validate the synthesis results, and discuss

some principles followed while writing our specifications.

Validation. Along with the synthesis of the circuits we also experimented for validation. Anzu generates the equivalent circuit description in Verilog from our input formal specifications. We obtained the Verilog description of the AHB arbiter as output from Anzu, given our input specifications, we then integrated the arbiter with manually written AHB Master and Slave, and simulated them with the tool Icarus verilog [16] for different stimulus and interconnection. Then we simulated with the Icarus verilog tool the synthesized AHB Master along with manually written AHB Arbiter and AHB Slave, and the synthesized AHB Slave with manually written AHB Arbiter and AHB Master. Finally we simulated all the synthesized Verilog (of AHB Arbiter, Master and Slave) with the Icarus verilog tool. In all cases the circuits functioned flawlessly.

Some principles. In the process of re-writing the formal specifications for efficient synthesis, we learnt a few lessons about writing formal specifications for synthesis. We discuss them with examples below.

1. In the process of writing specifications, we first simplified the design (whenever possible), then wrote realizable specifications for the simple design that can be synthesized efficiently, and finally added necessary complexities for the complete specification. For example, while writing AHB Master specifications, we first fixed all data and address signals width to one bit. We wrote specifications for the simpler design and efficiently synthesized for the simple design. This was followed by increasing the data and the address signal widths to 32-bit and adding necessary changes to the AHB Master specifications to make it complete and synthesizable. We illustrate with an example: Consider guarantee G14 for the AHB master that states "When REQ_WR_DATA is high, the input signals IN_DATA shall be copied to data bus in the next cycle". We first considered the above property as follows:
   `always` (REQ_WR_DATA $\rightarrow$ ((`next` (IN_DATA = HWDATA))),
   assuming both IN_DATA and HWDATA as one bit signals. After all properties were written with this simplification (address and data bus width fixed to one bit), we synthesized the design. We then proceeded to increase the bit-width of data and address bus. Such simplifications may also be useful to rule out many unrealizable specifications for the simpler design before considering the complete design.
2. While writing specifications, we proceeded with the execution/procedural order of events. This process was very helpful to prune the set of unrealizable specifications and obtain a realizable specification. For example, while writing AHB Arbiter specifications, we proceeded with writing properties related to requesting access, granting access followed by properties related to AHB transactions. Thus we started with properties related to bus request assertions and de-assertions, e.g., guarantee G1 for the AHB arbiter: During a locked burst transfer of unspecified length, leaving HBUSREQ_i high locks the bus. We then added properties related to transactions, e.g. guarantee G3: When a length-four locked burst starts, no other accesses start until the end of the burst.
3. The use of auxiliary signals was helpful in scenarios that could not be emulated using only input output signals. For example, it is not straightforward to emulate the interactions between the AHB slave and the internal memory. Hence, we have introduced many auxiliary signals such as FULL, EMPTY, RD, WR, and then wrote specifications with the auxiliary signals to capture the required interactions.
4. The liveness (eventual) specifications were the most time-consuming and difficult ones for synthesis. Hence special attention was devoted to write those specifications.

In general, most data intensive applications are not reactive designs of degree one, and the above approach may not be ideal for those applications, but we believe that the above principles should be helpful for many control specific applications.

Optional features. The AMBA AHB standard is not a completely closed protocol and hence there are some optional features. Few optional features of AHB standard were not considered in the formal specifications (also not considered in [1] as they are not the key component of the protocol, but optional ones). For example, one optional feature is that a slave is allowed to split a burst access and request that it be continued later. For AHB master, protection controls are also allowed as an optional feature. These optional features can be included in the formal specifications in a straightforward way.

Acknowledgment. We thank Barbara Jobstmann for explaining the changes made in the specifications from [1] to [2]. We are grateful to Saqib bin Sohail for many insightful comments about our specifications that helped us improve our paper.

References

1. Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *DATE*, pages 1188–1193, 2007.
2. Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.
3. J. Richard Buchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.
4. Alonzo Church. Logic, arithmetic, and automata. In *Proc. Internat. Congr. Math (Stockholm)*, pages 23–35, 1963.
5. Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
6. Barbara Jobstmann, Stefan Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. In *CAV*, pages 258–262, 2007.
7. ARM Ltd. Arm information center. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/588.html`.
8. ARM Ltd. Arm information center. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka3455.html`.
9. ARM Ltd. Arm information center. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka3462.html`.
10. ARM Ltd. Amba specification (rev. 2), 1999. `http://www.arm.com/products/solutions/AMBA_Spec.html`.
11. Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.
12. Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.
13. Michael Oser Rabin. *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, Boston, MA, USA, 1972.
14. F. Somenzi. Cudd: Cu decision diagram package. University of Colorado at Boulder, `ftp://vlsi.colorado.edu/pub/`.
15. Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, release 61225. `http://www.eecs.berkeley.edu/~alanmi/abc/`.
16. Stephen Williams. Icarus verilog tool. `ftp://ftp.icarus.com/pub/eda/verilog/v0.9`.

Table 7: PSL Specifications for AHB Arbiter.

A1	always ((HMASTLOCK ∧ HBURST = INCR) → (next eventually ¬BUSREQ))
A2	always eventually HREADY
A3	$\forall i$: always ((¬HBUSREQ_i ∧ ¬HLOCK_i ∧ (next HLOCK_i)) → (next HBUSREQ_i))
A4	always (¬HREADY → (HTRANS = 0 ↔ next HTRANS = 0)) always (¬HREADY → (HBURST = 0 ↔ next HBURST = 0))
A5	always ((HTRANS = IDLE) → (next (HTRANS ≠ SEQ)))
A6	always (((HTRANS = NONSEQ) ∧ (HBURST = INCR4) ∧ HREADY) → (next (HTRANS = SEQ)))
A7	always ((GRANTED ∧ HREADY) → (next (HTRANS = NONSEQ)))
A8	always ($(\wedge_{i=0}^{n-1} \neg \text{HBUSREQ}_i)$ → (HTRANS = IDLE))
A9	$\forall i$: (¬HBUSREQ_i ∧¬HLOCK_i ∧¬HREADY ∧ (HTRANS = IDLE) ∧ (HBURST = SINGLE))
G1	$\forall i$: always ((HMASTER = i) → (BUSREQ ↔ HBUSREQ_i))
G2	always ((HMASTLOCK ∧ (HBURST = INCR) ∧ HREADY ∧ (HTRANS = NONSEQ)) → next ((HTRANS = SEQ) until_ ¬BUSREQ))
G3	always ((HMASTLOCK ∧ (HBURST = INCR4) ∧ HREADY ∧ (HTRANS = NONSEQ)) → next ((HTRANS = SEQ) until_[3] HREADY))
G4	always ((DECIDE ∧$(\vee_{i=0}^{n-1}$ $\text{HBUSREQ}_i)$) → (next GRANTED))
G5	always ((GRANTED ∧¬HREADY) → (next GRANTED)) always ((GRANTED ∧ HREADY) → (next ¬GRANTED))
G6	$\forall i$: always (HREADY → (HGRANT_i ↔ next (HMASTER = i)))
G7	always ((HREADY ∧$(\vee_{i=0}^{n-1}$ (HLOCK_i ∧ HGRANT_i)) → next (HMASTLOCK)))
G8	$\forall i$: always ((¬HREADY∨ ¬GRANTED) → (HMASTER= i↔ next HMASTER= i)) always ((¬HREADY∨ ¬GRANTED) → (HMASTLOCK↔ next HMASTLOCK))
G9	$\forall i$: always (¬DECIDE → (HGRANT_i ↔ next HGRANT_i))
G10	$\forall i \neq 0$: always (¬HGRANT_i → (HBUSREQ_i before HGRANT_i)) always (DECIDE ∧$\forall i$: ¬HBUSREQ_i → next HGRANT0)
G11	$\forall i$: always (HBUSREQ_i → eventually (¬HBUSREQ_i ∨ (HMASTER = i)))
G12	DECIDE ∧ HGRANT0 ∧ (HMASTER = 0) ∧ ¬GRANTED ∧¬HMASTLOCK ∧$\forall i \neq 0$: ¬HGRANT_i

Table 8: Specifications for AHB Master - Assumptions

A1	always (REQ_VLD → (LENX ∨ LEN1 ∨ LEN4))
A2	always (REQ_VLD → (WR ∨ RD))
A3	always ((next ¬REQ_VLD) → (¬LEN1 ↔ (next ¬LEN1))) always ((next ¬REQ_VLD) → (¬LENX ↔ (next ¬LENX))) always ((next ¬REQ_VLD) → (¬LEN4 ↔ (next ¬LEN4))) always ((next ¬REQ_VLD) → (¬WR ↔ (next ¬WR))) always ((next ¬REQ_VLD) → (¬RD ↔ (next ¬RD)))
A4	always (WR → ¬ RD) always (RD → ¬ WR)
A5	always (LENX → (¬LEN1 ∧ ¬LEN4)) always (LEN1 → (¬LENX ∧ ¬LEN4)) always (LEN4 → (¬LENX ∧ ¬LEN1))
A6	always (HRESP = OKAY)
A7	always (REQ_VLD → eventually HGRANT)
A8	always ((HLOCK ∧ (HBURST = INCR)) → next eventually ¬REQ_VLD)
A9	always (eventually HREADY)
A10	always (eventually (¬REQ_VLD ∧ ¬HGRANT))

Table 9: PSL Specifications for AHB Master - Guarantees

G1	always (HSIZE = WORD)
G2	always (REQ_VLD ↔ HBUSREQ)
G3	always ((¬HBUSREQ ∧ next HBUSREQ ∧ ¬HLOCK) → next HLOCK)
G4	always (LAST → ¬HLOCK)
G5	always ((HLOCK ∧ (HBURST = INCR4) ∧ HREADY ∧ (HTRANS = NONSEQ)) → next ((HTRANS = SEQ) until_[3] HREADY))
G6	always (HBUSREQ ∧ HGRANT ∧ (HTRANS = IDLE) ∧ HREADY ∧ LEN1 → next (HBURST = SINGLE)) always (HBUSREQ ∧ HGRANT ∧ (HTRANS = IDLE) ∧ HREADY ∧ LENX → next (HBURST = INCR)) always (HBUSREQ ∧ HGRANT ∧ (HTRANS = IDLE) ∧ HREADY ∧ LEN4 → next (HBURST = INCR4))
G7	always (HBUSREQ ∧ HGRANT ∧ (HTRANS = IDLE) ∧ HREADY → next (HTRANS = NONSEQ)) always (¬LAST ∧ (HTRANS = NONSEQ) ∧ HREADY → next (HTRANS = SEQ)) always ((HTRANS = IDLE) → (HBURST = SINGLE))
G8	always (HGRANT ∧ (HTRANS = NONSEQ) ∧ HREADY ∧ WR → HWRITE) always (HGRANT ∧ (HTRANS = NONSEQ) ∧ HREADY ∧ RD → ¬HWRITE)
G9	always (¬HREADY → ((HTRANS = j) ↔ next (HTRANS = j))) always (¬HREADY → ((HBURST = j) ↔ next (HBURST = j)))
G10	always ((HREADY ∧ HGRANT) → REQ_ADDR)
G11	always ((REQ_ADDR ∧ WR) → REQ_WR_DATA)
G12	always ((HREADY ∧ ((HTRANS = NONSEQ) ∨ (HTRANS = SEQ)) ∧¬HWRITE) → REC_RD_DATA)
G13	∀i : always (REQ_ADDR → ((next (IN_ADDR$_i$ = HADDR$_i$))))
G14	∀i : always (REQ_WR_DATA → ((next (IN_DATA$_i$ = HWDATA$_i$))))
G15	∀i : always (HREADY ∧¬HWRITE ∧ ((HTRANS = SEQ) ∨ (HTRANS = NONSEQ)) → ((next (HRDATA$_i$ = OUT_DATA$_i$))))

Table 10: PSL Specifications for AHB Slave

A1	always (¬HSEL → ((HTRANS = IDLE)))
A2	always ((HTRANS = IDLE) → ((HBURST = SINGLE) ∧ ¬HWRITE ∧ ¬START ∧ ¬LAST))
A3	always (START → (HTRANS = NONSEQ))
A4	always (¬LAST ∧ (HTRANS = NONSEQ) ∧ HREADY → next (HTRANS = SEQ))
A5	always ((HLOCK ∧ (HBURST = INCR4) ∧ HREADY ∧ (HTRANS = NONSEQ)) → next((HTRANS = SEQ) until_[3] HREADY))
A6	always ((LAST ∧ next ¬START) → next (HTRANS = IDLE))
A7	always (¬HREADY →((HTRANS = j) ↔ next (HTRANS = j))) always (¬HREADY →((HBURST = j) ↔ next (HBURST = j))) always (¬HREADY →((HADDR = j) ↔ next (HADDR = j))) always (¬HREADY →((HWDATA = j) ↔ next (HWDATA = j))) always (¬HREADY →((DO = j) ↔ next (DO = j)))
G1	always (¬HSEL → HREADY)
G2	always (¬HSEL → (HRESP = OKAY))
G3	always ((HTRANS = IDLE) → (HRESP = OKAY))
G4	always ((WR ∧ HSEL) → ¬RD) always ((RD ∧ HSEL) → ¬WR)
G5	always ((HSEL ∧ FULL ∧ WR) → (HRESP = ERROR)) always ((HSEL ∧ EMPTY ∧ RD) → (HRESP = ERROR))
G6	always ((HSEL ∧ ((HTRANS = NONSEQ) ∨ (HTRANS = SEQ)) ∧ HWRITE) → WR) always ((HSEL ∧ ((HTRANS = NONSEQ) ∨ (HTRANS = SEQ)) ∧¬HWRITE) → RD)
G7	always ((HSEL ∧ ((HTRANS = NONSEQ) ∨ (HTRANS = SEQ)) → ((HADDR = j) ↔ (ADDR = j)))
G8	always ((HSEL ∧ ((HTRANS = NONSEQ) ∨ (HTRANS = SEQ)) ∧ HWRITE) → ((HWDATA = j) ↔ (DI = j)))
G9	always ((HSEL ∧ ((HTRANS = NONSEQ) ∨ (HTRANS = SEQ)) ∧¬HWRITE) → ((DO = j) ↔ (HRDATA = j)))