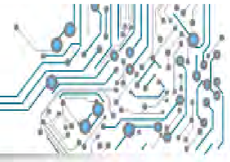
Department of Electrical and Computer Engineering



Διπλωματική Εργασία

"Έλεγχος και επαλήθευση πρωτοκόλλων bus και Systems-on-Chip"

Σοφία Γκουργκούνια

Επιβλέποντες : Ιωάννης Μούντανος, Νέστορας Ευμορφόπουλος

Βόλος, Ιούλιος 2013

“ Έλεγχος και επαλήθευση πρωτοκόλλων bus και Systems-on-Chip ”

Οι πυκνοτές silicon, τόσο για ASICs όσο για FPGAs, μπορούν πλέον να υποστηρίξουν πραγματικά Systems-on-Chip. Αυτό το επίπεδο σχεδιασμού απαιτεί συστήματα διαύλων για την διασύνδεση των διάφορων στοιχείων, συμπεριλαμβανομένου ενός ή παραπάνω μικροεπεξεργαστών, μνήμη, περοφερειακά και ειδική μνήμη.

Η αρχιτεκτονική επικοινωνίας για On-chip διαύλους (bus) είναι μεταξύ των κορυφαίων προκλήσεων στην CMOS SoC τεχνολογία εξαιτίας της ταχείας αύξησης των συχνοτήτων λειτουργίας και της αύξησης των μεγεθών των chip. Σε γενικές γραμμές, η απόδοση ενός SoC σχεδίου εξαρτάται σε πολύ μεγάλο βαθμό από την αποτελεσματικότητα της δομής του διαύλου. Η ισσοροπία μεταξύ υπολογισμού και επικοινωνίας σε οποιαδήποτε εφαρμογή ή εργασία είναι, φυσικά, γνωστή ως θεμελιώδης και καθοριστικός παράγοντας της τελικής απόδοσης. Συνήθως, οι IP πυρήνες, ως συστατικά των SoCs, σχεδιάζονται με πολλές διαφορετικές διεπαφές και πρωτόκολλα επικοινωνίας. Η ενσωμάτωση τέτοιων πυρήνων σε ένα SoC συχνά απαιτεί την εισαγωγή μιας “suboptimal glue logic”.

Πρότυπα των on-chip διαύλων αναπτύχθηκαν για να αποφευχθεί αυτό το πρόβλημα. Σ’ αυτή την διπλωματική εργασία εξετάζουμε τις αρχιτεκτονικές διαύλων AMBA, της ARM.

Το πρωτόκολλο AMBA είναι ένα ανοιχτό πρότυπο, μια on-chip προδιαγραφή διασύνδεσης για τη σύνδεση και τη διαχείρηση των λειτουργικών τμημάτων σε ένα System-on-Chip. Διευκολύνει τη “right-first-time” ανάπτυξη σχεδίων πολυ-επεξεργαστών με μεγάλο αριθμό ελεγκτών και περιφερειακών. Το AMBA προωθεί την επαναχρησιμοποίση σχεδίων με τον καθορισμό κοινών προτύπων διεπαφής για SoC ενότητες. Μέλη αυτής της οικογενείας πρωτοκόλλων είναι και τα AHB (Advanced High Performance Bus) και APB (Advance Peripheral Bus) που εξετάζουμε σ’αυτή την εργασία.

Συγκεκριμένα σχεδιάσαμε μια AHB2APB γέφυρα η οποία παρέχει μια διεπαφή μεταξύ, του υψηλής ταχύτητας διαύλου AHB και του χαμηλής ενέργειας APB. Η AHB2APB γέφυρα προσομοιώθηκε αρχικά με το GTKWave όπου παρουσιάζουμε τόσο μεταφορές ανάγνωσης όσο και εγγραφής και έπειτα με το Insive Formal Verifier.

Τέλος, χρησιμοποιήσαμε έναν AHB master ελέγξαμε κατά πόσο ικανοποιεί κάποιες από τις ιδιότητες που προδιαγράφει το πρωτόκολλο AHB.

CONTENTS

1. Introduction……………………………………………………………………………4
2. Advanced High Performance Bus……………………………………………………….4
   2.1 A typical AMBA AHB-based microcontroller……………………………………...5
   2.2 Bus Interconnection…………………………………………………………………5
   2.3 Overview of AMBA AHB operation………………………………………………..6
   2.4 Basic Transfer……………………………………………………….………….7
   2.5 Transfer Type……………………………………………………….…………..9
   2.6 Burst operation ………………………………………………………………….10
   2.7 Control Signals ………………………………………………………………….12
   2.8 Address decoding ………………………………………………………………14
   2.9 Slave Transfer Response………………………………………………………. .15
      2.9.1 Transfer Done……………………………………………………………. 15
      2.9.2 Transfer Response………………………………………………………... 15
      2.9.3 Two cycle response………………………………………………………. 16
      2.9.4 Error Response……………………………………………………………17
      2.9.5 Split and Retry……………………………………………………………17
   2.10 Data Buses…………………………………………………………………. 18
   2.11 Arbitration…………………………………………………………………. 19
      2.11.1 Requesting Bus Access…………………………………………………... 19
      2.11.2 Granting Bus Access ……………………………………………………...20
      2.11.3 Early Burst termination…………………………………………………... 22
      2.11.4 Locked Transfers…………………………………………………………. 22
      2.11.5 Default bus master ……………………….……………………………..23
      2.11.6 Split transfers ……………………………………………………………..23
      2.11.7 Reset ………………………………………………………………………27
      2.11.8 AHB AMBA components…………………………………………………28
      2.11.9 AHB bus matrix topology…………………………………………………31
3. Advanced Peripheral Bus……………………………………………………………….31
   3.1 APB Bridge………………………………………………………………………...32
   3.2 APB Slave…………………………………………………………………………32
   3.3 APB State Diagram ……………………………………………………………..34
   3.4 APB Write Transfer……………………………………………………………..36
   3.5 APB Read Transfer……………………………………………………………...37
   3.6 AHB vs APB………………………………………………………………………37
4. AHB2APB Bridge…………………………………………………………………...39
   4.1 AHB2APB bridge module signals………………………………………………40
   4.2 AHB2APB function and operation……………………………………………...41
5. Synthesis and Simulation ………………………………………………………………45

Abstract-**Advanced microcontroller bus architecture (AMBA) protocol family provides metric-driven verification of protocol compliance, enabling comprehensive testing of interface intellectual property (IP) blocks and System-on-Chip (SoC) designs. This bachelor thesis presents a work aimed to design the AMBA AHB2APB bridge modeled in VHDL hardware description language (HDL) and simulate the results for read and write operation of data and address using the INCISIVE Cadence tool**.

1. Introduction

Embedded system designers have a choice of using a share or point-to-point bus in their designs. Typically, an embedded design will have a general purpose processor, cache, SDRAM, DMA port, and Bridge port to a slower I/O bus, such as the Advanced Micro controller Bus Architecture (AMBA) Advanced Peripheral Bus (APB). In addition, there might be a port to a DSP processor, or hardware accelerator, common with the increased use of video in many applications. As chip-level device geometries become smaller and smaller, more and more functionality can be added without the concomitant increase in power and cost per die as seen in prior generations.
The Advanced Microcontroller Bus Architecture (AMBA) was introduced by ARM Ltd 1996 and is widely used as the on-chip bus in system on chip (SoC) designs. AMBA is a registered trademark of ARM Ltd. The first AMBA buses were Advanced System Bus (ASB) and Advanced Peripheral Bus (APB). In its 2nd version, AMBA 2, ARM added AMBA High-performance Bus (AHB) that is a single clock-edge protocol. In 2003, ARM introduced the 3rd generation, AMBA 3, including AXI to reach even higher performance interconnects and the Advanced Trace Bus (ATB) as part of the Core Sight on-chip debugs and trace solution. These protocols are today the de-facto standard for 32-bit embedded processors because they are well documented and can be used without royalties. In 2010 the AMBA 4 specifications were introduced starting with AMBA 4 AXI4, then in 2011 extending system wide coherency with AMBA 4 ACE. In 2013 the AMBA 5 CHI (Coherent Hub Interface) specification was introduced, with a re-designed high-speed transport layer and features designed to reduce congestion. The thesis has been organized as follows. The first section contains the description of AHB protocol. Second section describes the APB protocol. Third section describes AHB-to-APB bridge module. Fourth section shows how we used INCISIVE for synthesis and simulation.

2. Advanced High Performance Bus (AHB)

AHB is the second generation of AMBA bus which is intended to address the requirements of high-performance synthesizable designs. AMBA AHB is a new level of bus which sits above the APB and implements the features required for high-performance, high clock frequency systems including:

- burst transfers
- split transactions
- single cycle bus master handover
- single clock edge operation
- non-tristate implementation

• wider data bus configurations (64/128 bits)

2.1 A typical AMBA AHB-based microcontroller

An AMBA-based microcontroller typically consists of a high-performance system *backbone* bus, able to sustain the external memory bandwidth, on which the CPU and other *Direct Memory Access* (DMA) devices reside, plus a bridge to a narrower APB bus on which the lower bandwidth peripheral devices are located. Figure1 shows both AHB and APB in a typical AMBA system.

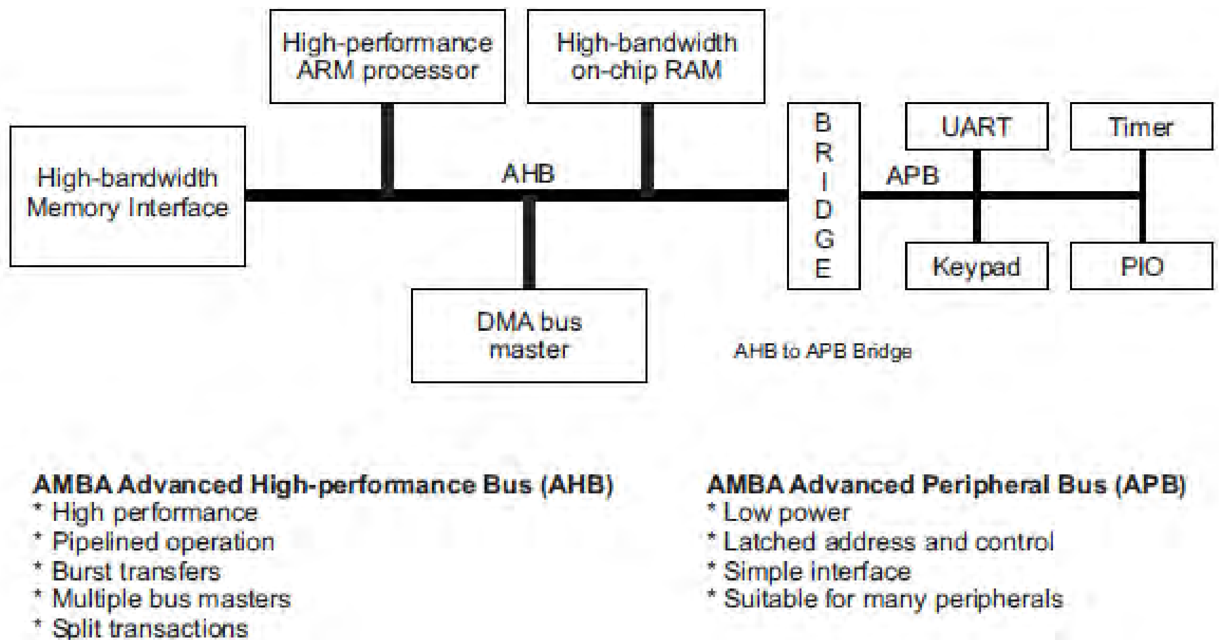


Figure 1

2.2 Bus Interconnection

The AMBA AHB bus protocol is designed to be used with a central multiplexor interconnection scheme. Using this scheme all bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all of the slaves. A central decoder is also required to control the read data and response signal multiplexor, which selects the appropriate signals from the slave that is involved in the transfer.

Figure 2 illustrates the structure required to implement an AMBA AHB design with three masters and four slaves.

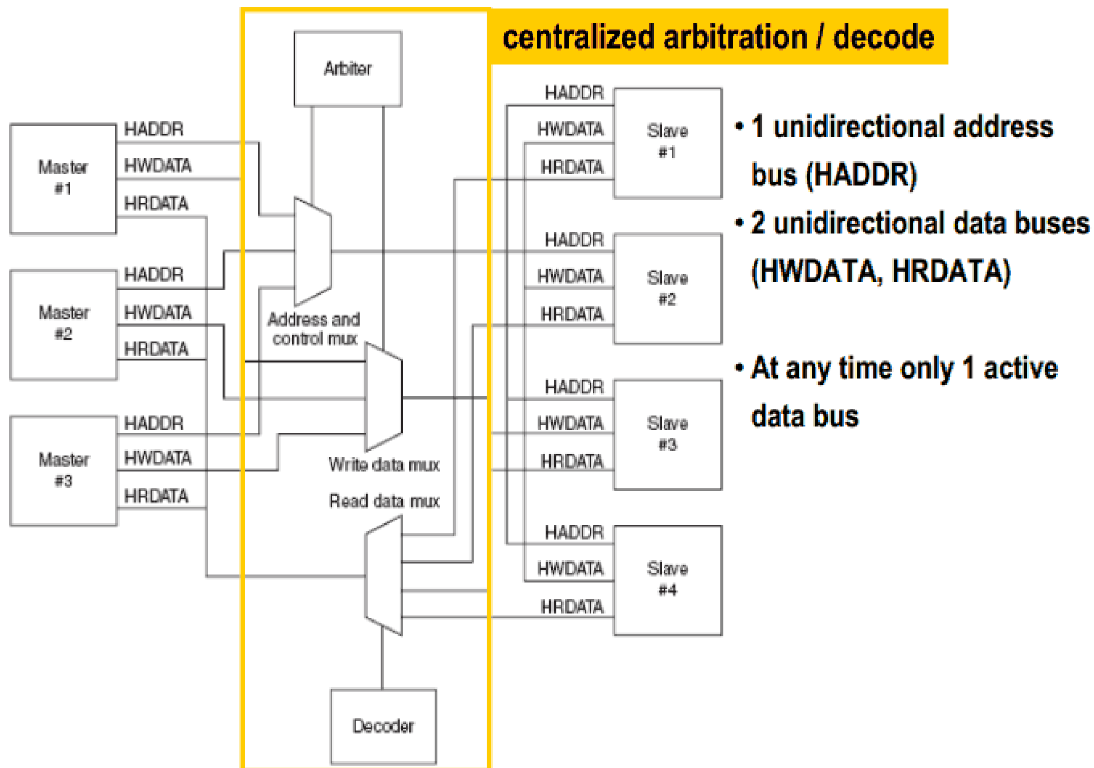


Figure 2

2.3 Overview of AMBA AHB operation

Before an AMBA AHB transfer can commence the bus master must be granted access to the bus. This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master will be granted use of the bus. A granted bus master starts an AMBA AHB transfer by driving the address and control signals. These signals provide information on the address, direction and width of the transfer, as well as an indication if the transfer forms part of a burst. Two different forms of burst transfers are allowed:

- incrementing bursts, which do not wrap at address boundaries
- wrapping bursts, which wrap at particular address boundaries.

A write data bus is used to move data from the master to a slave, while a read data bus is used to move data from a slave to the master. Every transfer consists of:

- an address and control cycle
- one or more cycles for the data.

The address cannot be extended and therefore all slaves must sample the address during this time. The data, however, can be extended using the **HREADY** signal. When LOW this signal causes wait states to be inserted into the transfer and allows extra time for the slave to provide or sample data.

During a transfer the slave shows the status using the response signals, **HRESP[1:0]**:

OKAY The OKAY response is used to indicate that the transfer is

progressing normally and when **HREADY** goes HIGH this shows the transfer has completed successfully.

ERROR	The ERROR response indicates that a transfer error has occurred and the transfer has been unsuccessful.
RETRY and SPLIT	Both the RETRY and SPLIT transfer responses indicate that the transfer cannot complete immediately, but the bus master should continue to attempt the transfer.

In normal operation a master is allowed to complete all the transfers in a particular burst before the arbiter grants another master access to the bus. However, in order to avoid excessive arbitration latencies it is possible for the arbiter to break up a burst and in such cases the master must re-arbitrate for the bus in order to complete the remaining transfers in the burst.

2.4 Basic Transfer

An AHB transfer consists of two distinct sections:

- The address phase, which lasts only a single cycle.
- The data phase, which may require several cycles. This is achieved using the **HREADY** signal.

Figure 3 shows the simplest transfer, one with no wait states

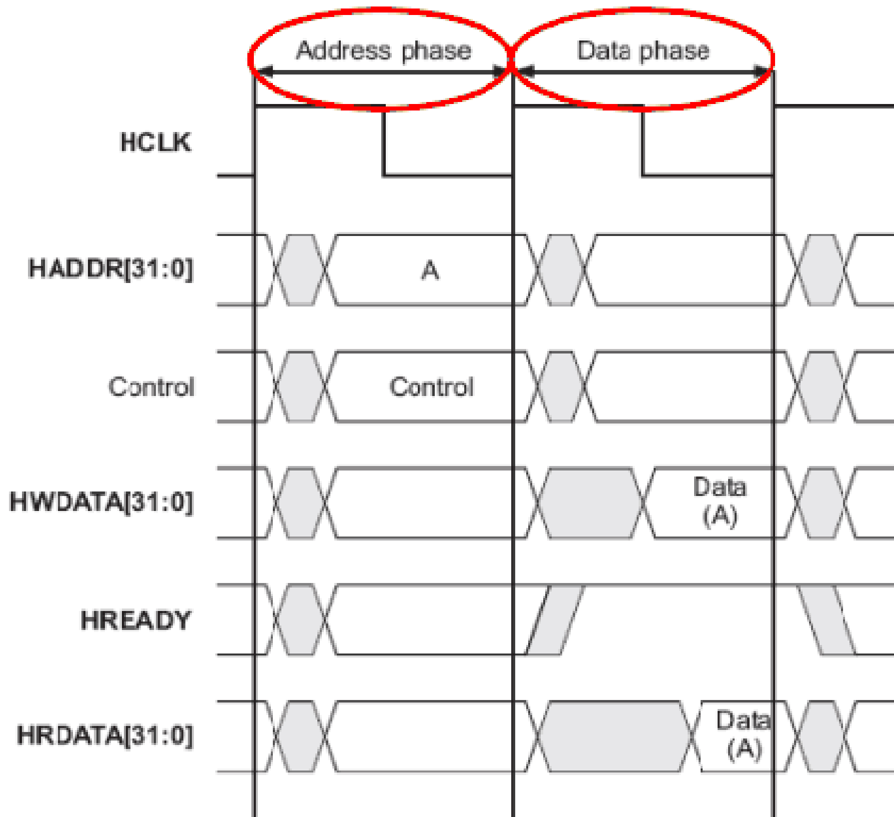


Figure 3

- The master drives the address and control signals onto the bus after the rising edge of **HCLK**.
- The slave then samples the address and control information on the next rising edge of the clock.
- After the slave has sampled the address and control it can start to drive the appropriate response and this is sampled by the bus master on the third rising edge of the clock.

This simple example demonstrates how the address and data phases of the transfer occur during different clock periods. In fact, the address phase of any transfer occurs during the data phase of the previous transfer. This overlapping of address and data is fundamental to the pipelined nature of the bus and allows for high performance operation, while still providing adequate time for a slave to provide the response to a transfer.

A slave may insert wait states into any transfer, as shown in Figure 4, which extends the transfer allowing additional time for completion.

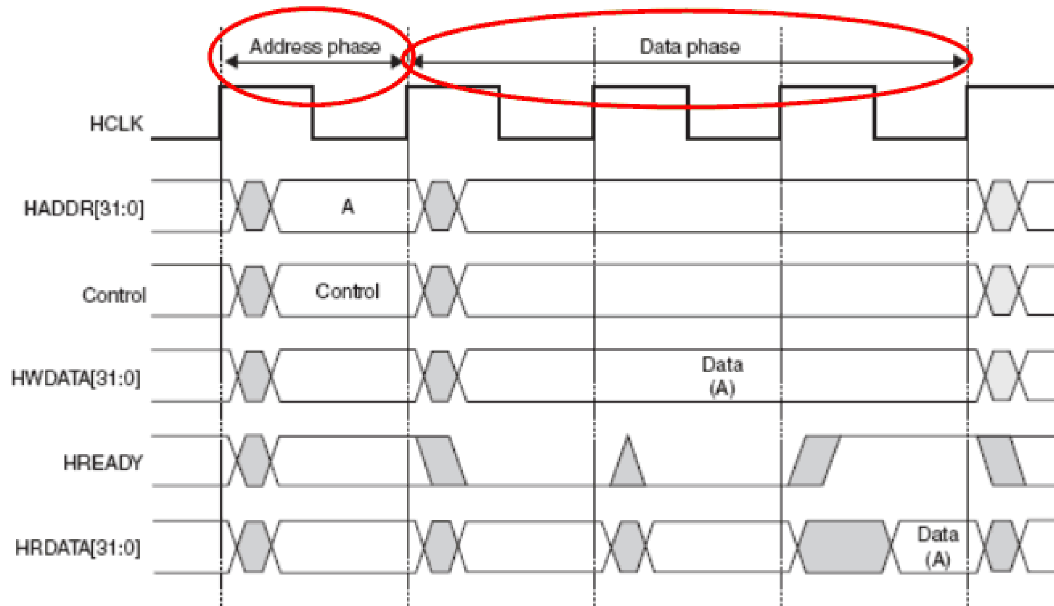


Figure 4

When a transfer is extended in this way it will have the side-effect of extending the address phase of the following transfer. This is illustrated in Figure 5 which shows three transfers to unrelated addresses, A, B & C. Transaction pipelining increases bus bandwidth.

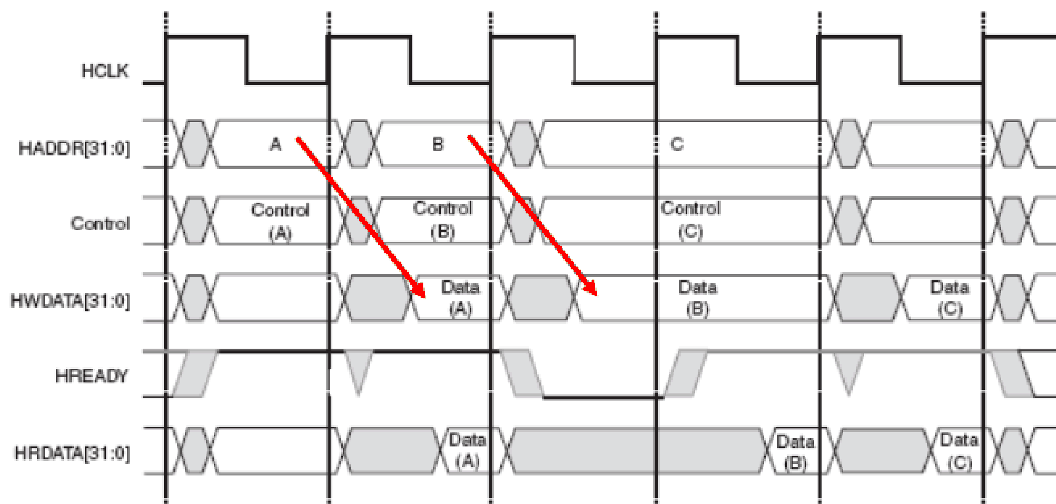


Figure 5

2.5 Transfer Type

Every transfer can be classified into one of four different types, as indicated by the **HTRANS[1:0]** signals as shown in Table 1

HTRANS[1:0]	Type	Description
00	IDLE	Indicates that no data transfer is required. The IDLE transfer type is used when a bus master is granted the bus, but does not wish to perform a data transfer. Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer should be ignored by the slave.
01	BUSY	The BUSY transfer type allows bus masters to insert IDLE cycles in the middle of bursts of transfers. This transfer type indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. When a master uses the BUSY transfer type the address and control signals must reflect the next transfer in the burst. The transfer should be ignored by the slave. Slaves must always provide a zero wait state OKAY response, in the same way that they respond to IDLE transfers.
10	NONSEQ	Indicates the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer. Single transfers on the bus are treated as bursts of one and therefore the transfer type is NONSEQUENTIAL.
11	SEQ	The remaining transfers in a burst are SEQUENTIAL and the address is related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). In the case of a wrapping burst the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8 or 16).

Table 1

2.6 Burst Operation

Both incrementing and wrapping bursts are supported in the protocol:

- Incrementing bursts access sequential locations and the address of each transfer in the burst is just an increment of the previous address.

- For wrapping bursts, if the start address of the transfer is not aligned to the total number of bytes in the burst (size x beats) then the address of the transfers in the burst will wrap when the boundary is reached. For example, a four-beat wrapping burst of word (4-byte) accesses will wrap at 16-byte boundaries. Therefore, if the start address of the transfer is 0x34, then it consists of four transfers to addresses 0x34, 0x38, 0x3C and 0x30.

Burst information is provided using **HBURST[2:0]** and the eight possible types are defined in Table 2.

Fixed length bursts

HBURST[2:0]	Type	Description
000	SINGLE	Single transfer
001	INCR	Incrementing burst of unspecified length
010	WRAP4	4-beat wrapping burst
011	INCR4	4-beat incrementing burst
100	WRAP8	8-beat wrapping burst
101	INCR8	8-beat incrementing burst
110	WRAP16	16-beat wrapping burst
111	INCR16	16-beat incrementing burst

Table 2

Bursts must not cross a 1kB address boundary. Therefore it is important that masters do not attempt to start a fixed-length incrementing burst which would cause this boundary to be crossed.

There are certain circumstances when a burst will not be allowed to complete (early burst termination) and therefore it is important that any slave design which makes use of the burst information can take the correct course of action if the burst is terminated early. The slave can determine when a burst has terminated early by monitoring the **HTRANS** signals and ensuring that after the start of the burst every transfer is labelled as SEQUENTIAL or BUSY. If a NONSEQUENTIAL or IDLE transfer occurs then this indicates that a new burst has started and therefore the previous one must have been terminated. If a bus master cannot complete a burst because it loses ownership of the bus then it must rebuild the burst appropriately when it next gains access to the bus. For example, if a master has only completed one beat of a four-beat burst then it must use an undefined-length burst to perform the remaining three transfers.

In Figure 6 we can see an example of a four-beat incrementing burst that shows how bursts cut down on arbitration, handshaking time and improve performance.

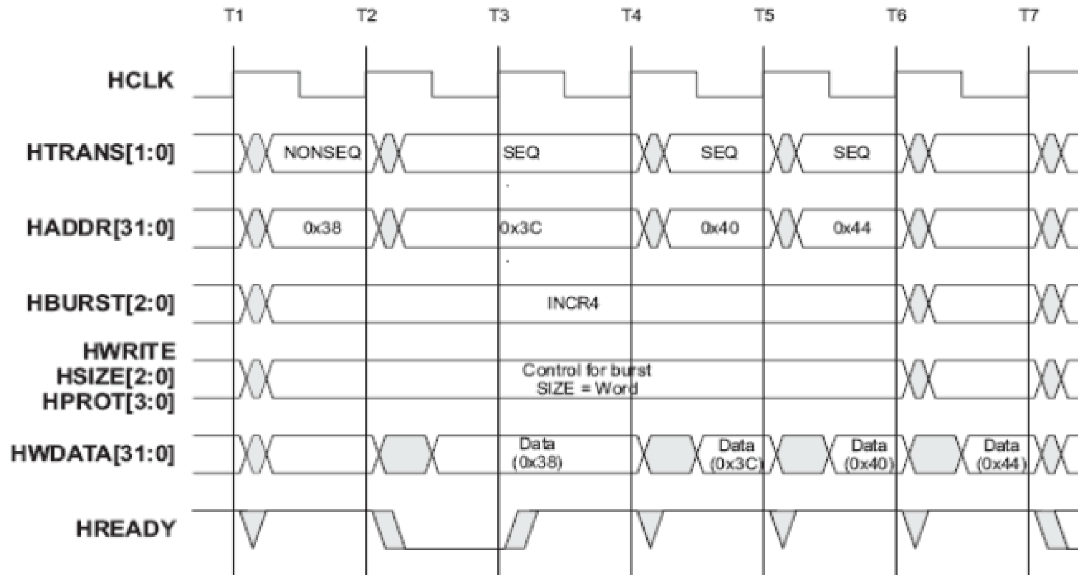


Figure 6

2.7 Control Signals

As well as the transfer type and burst type each transfer will have a number of control signals that provide additional information about the transfer. These control signals have exactly the same timing as the address bus. However, they must remain constant throughout a burst of transfers.

- Transfer direction

  When HWRITE is HIGH, this signal indicates a write transfer and the master will broadcast data on the write data bus, HWDATA[31:0]. When LOW a read transfer will be performed and the slave must generate the data on the read data bus HRDATA[31:0].

- Transfer size

  HSIZE[2:0] indicates the size of the transfer, as shown in Table 3.

HSIZE[2]	HSIZE[1]	HSIZE[0]	Size	Description
0	0	0	8 bits	Byte
0	0	1	16 bits	Halfword
0	1	0	32 bits	Word
0	1	1	64 bits	-
1	0	0	128 bits	4-word line
1	0	1	256 bits	8-word line
1	1	0	512 bits	-
1	1	1	1024 bits	-

Table 3

The size is used in conjunction with the HBURST[2:0] signals to determine the address boundary for wrapping bursts.

- Protection Control

The protection control signals, HPROT[3:0], provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection

HPROT[3] cacheable	HPROT[2] bufferable	HPROT[1] privileged	HPROT[0] data/opcode	Description
-	-	-	0	Opcode fetch
-	-	-	1	Data access
-	-	0	-	User access
-	-	1	-	Privileged access
-	0	-	-	Not bufferable
-	1	-	-	Bufferable
0	-	-	-	Not cacheable
1	-	-	-	Cacheable

Table 4

Not all bus masters will be capable of generating accurate protection information, therefore it is recommended that slaves do not use the HPROT signals unless strictly necessary.

2.8 Address decoding

A central address decoder is used to provide a select signal, HSELx, for each slave on the bus. The select signal is a combinatorial decode of the high-order address signals, and simple address decoding schemes are encouraged to avoid complex decode logic and to ensure high-speed operation.

A slave must only sample the address and control signals and HSELx when HREADY is HIGH, indicating that the current transfer is completing. Under certain circumstances it is possible that HSELx will be asserted when HREADY is LOW, but the selected slave will have changed by the time the current transfer completes.

The minimum address space that can be allocated to a single slave is 1kB. All bus masters are designed such that they will not perform incrementing transfers over a 1kB boundary, thus ensuring that a burst never crosses an address decode boundary.

In the case where a system design does not contain a completely filled memory map an additional default slave should be implemented to provide a response when any of the nonexistent address locations are accessed. If a NONSEQUENTIAL or SEQUENTIAL transfer is attempted to a nonexistent address location then the default slave should provide an ERROR response. IDLE or BUSY transfers to nonexistent locations should result in a zero wait state OKAY response. Typically the default slave functionality will be implemented as part of the central address decoder.

Figure 7 shows a typical address decoding system and the slave select signals

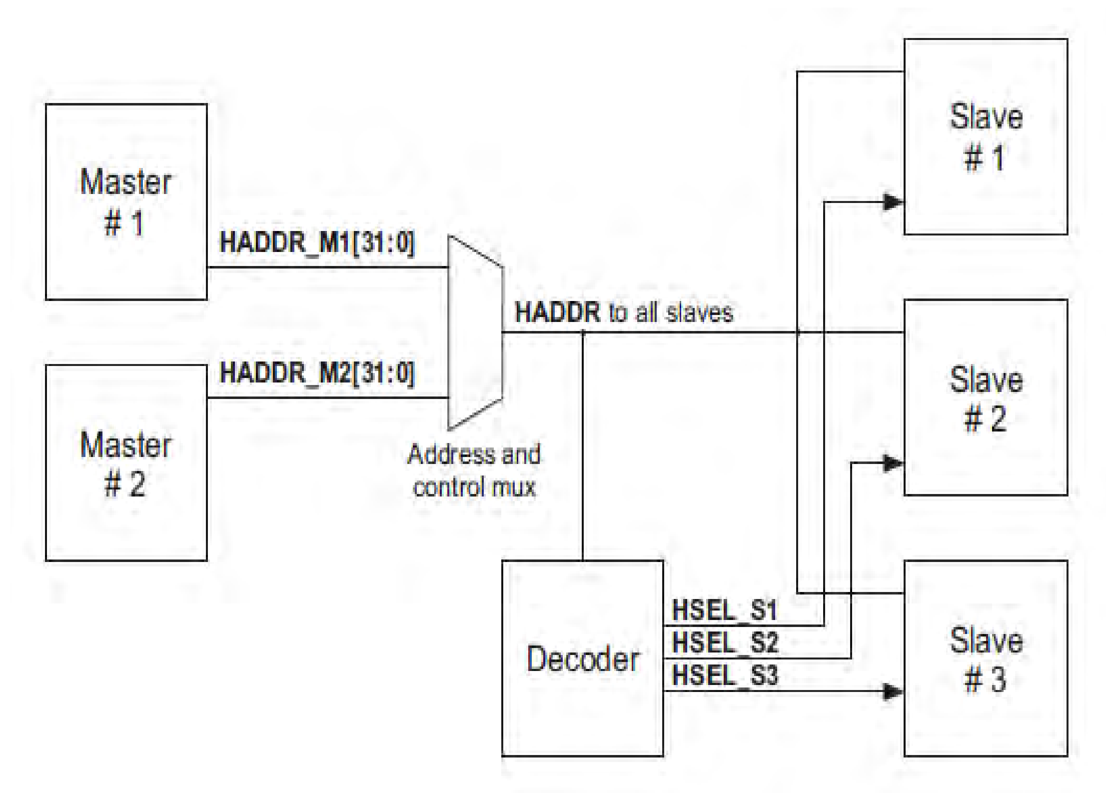


Figure 7

2.9 Slave Transfer Response

After a master has started a transfer, the slave then determines how the transfer should progress. No provision is made within the AHB specification for a bus master to cancel a transfer once it has commenced.

Whenever a slave is accessed it must provide a response which indicates the status of the transfer. The HREADY signal is used to extend the transfer and this works in combination with the response signals, HRESP[1:0], which provide the status of the transfer.

The slave can complete the transfer in a number of ways. It can:

- complete the transfer immediately
- insert one or more wait states to allow time to complete the transfer
- signal an error to indicate that the transfer has failed
- delay the completion of the transfer, but allow the master and slave to back off the bus, leaving it available for other transfers

2.9.1 Transfer Done

The HREADY signal is used to extend the data portion of an AHB transfer. When LOW the HREADY signal indicates the transfer is to be extended and when HIGH indicates that the transfer can complete.

2.9.2 Transfer Response

A typical slave will use the HREADY signal to insert the appropriate number of wait states into the transfer and then the transfer will complete with HREADY HIGH and an OKAY response, which indicates the successful completion of the transfer.

The ERROR response is used by a slave to indicate some form of error condition with the associated transfer. Typically this is used for a protection error, such as an attempt to write to a read-only memory location.

The SPLIT and RETRY response combinations allow slaves to delay the completion of a transfer, but free up the bus for use by other masters. These response combinations are usually only required by slaves that have a high access latency and can make use of these response codes to ensure that other masters are not prevented from accessing the bus for long periods of time.

The encoding of **HRESP[1:0]**, the transfer response signals, and a description of each response are shown in Table 5.

HRESP[1]	HRESP[0]	Response	Description
0	0	OKAY	When **HREADY** is HIGH this shows the transfer has completed successfully. The OKAY response is also used for any additional cycles that are inserted, with **HREADY** LOW, prior to giving one of the three other responses.
0	1	ERROR	This response shows an error has occurred. The error condition should be signalled to the bus master so that it is aware the transfer has been unsuccessful. A two-cycle response is required for an error condition.
1	0	RETRY	The RETRY response shows the transfer has not yet completed, so the bus master should retry the transfer. The master should continue to retry the transfer until it completes. A two-cycle RETRY response is required.
1	1	SPLIT	The transfer has not yet completed successfully. The bus master must retry the transfer when it is next granted access to the bus. The slave will request access to the bus on behalf of the master when the transfer can complete. A two-cycle SPLIT response is required.

Table 5

2.9.3 Two cycle Response

Only an OKAY response can be given in a single cycle. The ERROR, SPLIT and RETRY responses require at least two cycles. To complete with any of these responses then in the penultimate (one before last) cycle the slave drives HRESP[1:0] to indicate ERROR, RETRY or SPLIT while driving HREADY LOW to extend the transfer for an extra cycle. In the final cycle HREADY is driven HIGH to end the transfer, while HRESP[1:0] remains driven to indicate ERROR, RETRY or SPLIT.

If the slave needs more than two cycles to provide the ERROR, SPLIT or RETRY response then additional wait states may be inserted at the start of the transfer. During this time the HREADY signal will be LOW and the response must be set to OKAY.

The two-cycle response is required because of the pipelined nature of the bus. By the time a slave starts to issue either an ERROR, SPLIT or RETRY response then the address for the following transfer has already been broadcast onto the bus. The two cycle response allows sufficient time for the master to cancel this address and drive HTRANS[1:0] to IDLE before the start of the next transfer.

For the SPLIT and RETRY response the following transfer must be cancelled because it must not take place before the current transfer has completed. However, for the ERROR response, where the current transfer is not repeated, completion of the following transfer is optional.

Figure 8 shows an example of a RETRY operation.

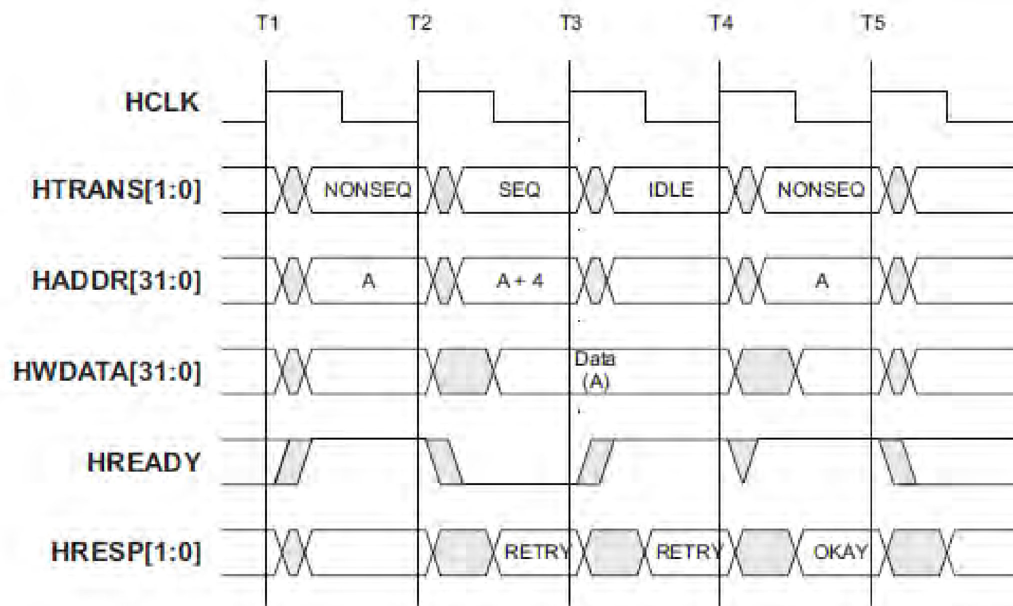


Figure 8

The following events are illustrated:

- The master starts with a transfer to address A.
- Before the response is received for this transfer the master moves the address on to A + 4.
- The slave at address A is unable to complete the transfer immediately and therefore it issues a RETRY response. This response indicates to the master that the transfer at address A is unable to complete and so the transfer at address A +4 is cancelled and replaced by an IDLE transfer.

2.9.4 Error Response

If a slave provides an ERROR response then the master may choose to cancel the remaining transfers in the burst. However, this is not a strict requirement and it is also acceptable for the master to continue the remaining transfers in the burst.

2.9.5 Split and Retry

The SPLIT and RETRY responses provide a mechanism for slaves to release the bus when they are unable to supply data for a transfer immediately. Both mechanisms allow the transfer to finish on the bus and therefore allow a higher-priority master to get access to the bus.

The difference between SPLIT and RETRY is the way the arbiter allocates the bus after a SPLIT or a RETRY has occurred:

- For RETRY the arbiter will continue to use the normal priority scheme and therefore only masters having a higher priority will gain access to the bus.

- For a SPLIT transfer the arbiter will adjust the priority scheme so that any other master requesting the bus will get access, even if it is a lower priority. In order for a SPLIT transfer to complete the arbiter must be informed when the slave has the data available.

The SPLIT transfer requires extra complexity in both the slave and the arbiter, but has the advantage that it completely frees the bus for use by other masters, whereas the RETRY case will only allow higher priority masters onto the bus.

A bus master should treat SPLIT and RETRY in the same manner. It should continue to request the bus and attempt the transfer until it has either completed successfully or been terminated with an ERROR response.

2.10 Data Buses

In order to allow implementation of an AHB system without the use of tristate drivers separate read and write data buses are required. The minimum data bus width is specified as 32 bits.

- HWDATA[31:0]

  The write data bus is driven by the bus master during write transfers. If the transfer is extended then the bus master must hold the data valid until the transfer completes, as indicated by HREADY HIGH.

  All transfers must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries (that is A[1:0] = 00), halfword transfers must be aligned to halfword address boundaries (that is A[0] = 0).

  For transfers that are narrower than the width of the bus, for example a 16-bit transfer on a 32-bit bus, then the bus master only has to drive the appropriate byte lanes. The slave is responsible for selecting the write data from the correct byte lanes.

- HRDATA[31:0]

  The read data bus is driven by the appropriate slave during read transfers. If the slave extends the read transfer by holding HREADY LOW then the slave only needs to provide valid data at the end of the final cycle of the transfer, as indicated by HREADY HIGH.

  For transfers that are narrower than the width of the bus the slave only needs to provide valid data on the active byte lanes. The bus master is responsible for selecting the data from the correct byte lanes.

A slave only has to provide valid data when a transfer completes with an OKAY response. SPLIT, RETRY and ERROR responses do not require valid read data.

2.11 Arbitration

The arbitration mechanism is used to ensure that only one master has access to the bus at any one time. The arbiter performs this function by observing a number of different requests to use the bus and deciding which is currently the highest priority master requesting the bus. The arbiter also receives requests from slaves that wish to complete SPLIT transfers.

Any slaves which are not capable of performing SPLIT transfers do not need to be aware of the arbitration process, except that they need to observe the fact that a burst of transfers may not complete if the ownership of the bus is changed.

A brief description of each of the arbitration signals is given below:

- HBUSREQx **:** The bus request signal is used by a bus master to request access to the bus. Each bus master has its own **HBUSREQx** signal to the arbiter and there can be up to 16 separate bus masters in any system.

- HLOCKx : The lock signal is asserted by a master at the same time as the bus request signal. This indicates to the arbiter that the master is performing a number of indivisible transfers and the arbiter must not grant any other bus master access to the bus once the first transfer of the locked transfers has commenced. HLOCKx must be asserted at least a cycle before the address to which it refers, in order to prevent the arbiter from changing the grant signals.

- HGRANTx : The grant signal is generated by the arbiter and indicates that the appropriate master is currently the highest priority master requesting the bus, taking into account locked transfers and SPLIT transfers.

  A master gains ownership of the address bus when HGRANTx is HIGH and HREADY is HIGH at the rising edge of HCLK.

- HMASTER : The arbiter indicates which master is currently granted the bus using the HMASTER[3:0] signals and this can be used to control the central address and control multiplexor. The master number is also required by SPLIT-capable slaves so that they can indicate to the arbiter which master is able to complete a SPLIT transaction.

- HMASTLOCK : The arbiter indicates that the current transfer is part of a locked sequence by asserting the HMASTLOCK signal, which has the same timing as the address and control signals.

- HSPLIT : The 16-bit Split Complete bus is used by a SPLIT-capable slave to indicate which bus master can complete a SPLIT transaction. This information is needed by the arbiter so that it can grant the master access to the bus to complete the transfer.

2.11.1 Requesting Bus Access

A bus master uses the HBUSREQx signal to request access to the bus and may request the bus during any cycle. The arbiter will sample the request on the rising of the clock and then use an internal priority algorithm to decide which master will be the next to gain access to the bus.

Normally the arbiter will only grant a different bus master when a burst is completing.
However, if required, the arbiter can terminate a burst early to allow a higher priority master access to the bus.

If the master requires locked accesses then it must also assert the HLOCKx signal to indicate to the arbiter that no other masters should be granted the bus.

When a master is granted the bus and is performing a fixed length burst it is not necessary to continue to request the bus in order to complete the burst. The arbiter observes the progress of the burst and uses the HBURST[2:0] signals to determine how many transfers are required by the master. If the master wishes to perform a second burst after the one that is currently in progress then it should re-assert the request signal during the burst.

If a master loses access to the bus in the middle of a burst then it must re-assert the HBUSREQx request line to regain access to the bus.

For undefined length bursts the master should continue to assert the request until it has started the last transfer. The arbiter cannot predict when to change the arbitration at the end of an undefined length burst.

It is possible that a master can be granted the bus when it is not requesting it. This may occur when no masters are requesting the bus and the arbiter grants access to a default master. Therefore, it is important that if a master does not require access to the bus it drives the transfer type HTRANS to indicate an IDLE transfer.

2.11.2 Granting Bus Access

The arbiter indicates which bus master is currently the highest priority requesting the bus by asserting the appropriate HGRANTx signal. When the current transfer completes, as indicated by HREADY HIGH, then the master will become granted and the arbiter will change the HMASTER[3:0] signals to indicate the bus master number.

Figure 9 shows the cost of arbitration in AHB. The ownership of the data bus is delayed from the ownership of the address bus. Whenever a transfer completes, as indicated by HREADY HIGH, then the master that owns the address bus will be able to use the data bus and will continue to own the data bus until the transfer completes.

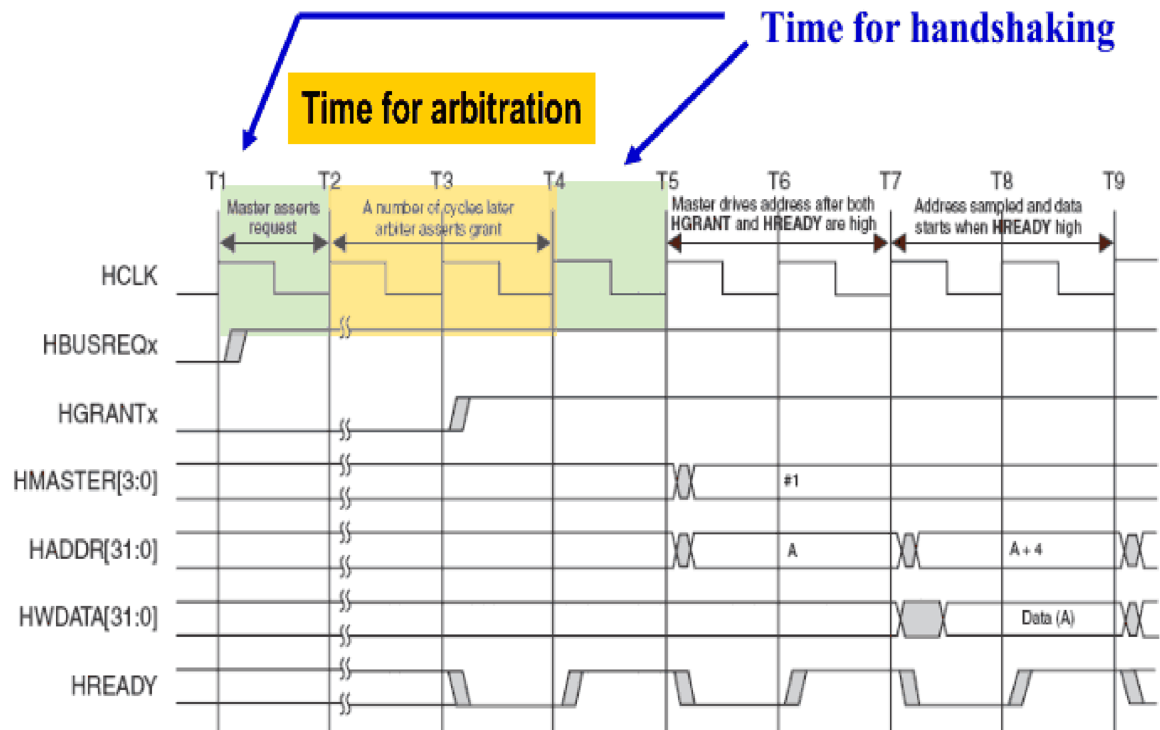


Figure 9

It's also worth mentioning that although arbitration protocol is specified, arbitration policy is not. Figure 10 shows how HGRANTx and HMASTER signals are used in a system. Because a central multiplexor is used, each master can drive out the address of the transfer it wishes to perform immediately and it does not need to wait until it is granted the bus. The HGRANTx signal is only used by the master to determine when it owns the bus and hence when it should consider that the address has been sampled by the appropriate slave.

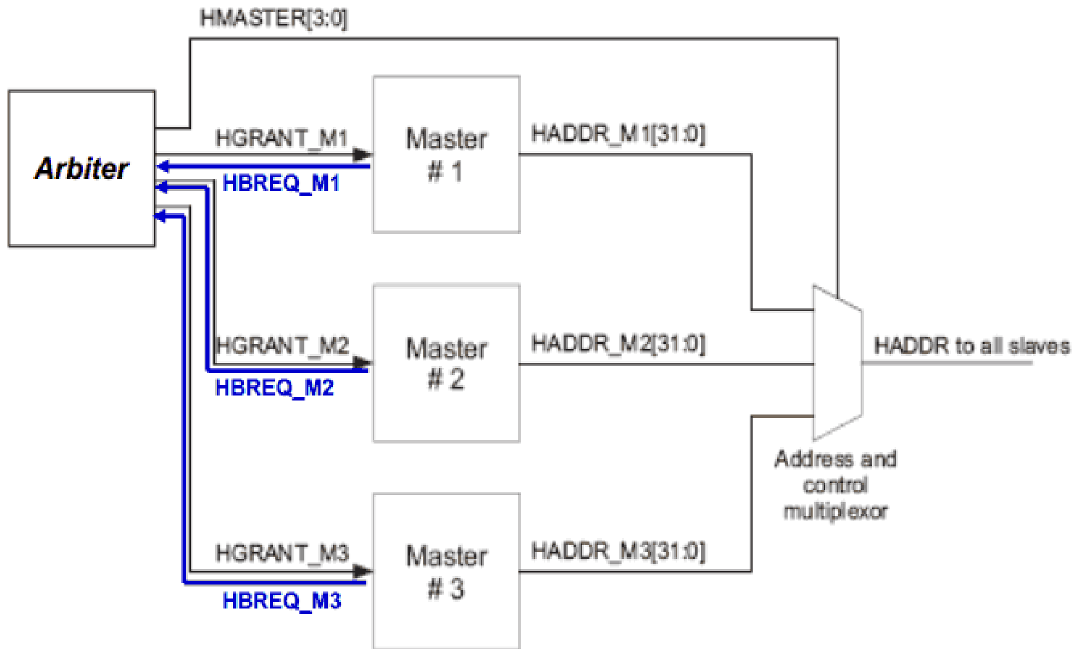


Figure 10

2.11.3 Early burst termination

Normally the arbiter will not hand over the bus to a new master until the end of a burst of transfers. However, if the arbiter determines that the burst must be terminated early in order to prevent excessive access time to the bus then it may transfer the grant to another bus master before a burst has completed.

If a master loses ownership of the bus in the middle of a burst it must re-arbitrate for the bus in order to complete the burst. The master must ensure that the HBURST and HTRANS signals are adapted to reflect the fact that it no longer has to perform a complete 4, 8 or 16-beat burst.

For example, if a master is only able to complete 3 transfers of an 8-beat burst, then when it regains the bus it must use a legal burst encoding to complete the remaining 5 transfers. Any legal combination can be used, so either a 5-beat undefined length burst or a 4-beat fixed length burst followed by a single-beat undefined length burst would be acceptable.

2.11.4 Locked Transfers

The arbiter must observe the HLOCKx signal from each master to determine when the master wishes to perform a locked sequence of transfers. The arbiter is then responsible for ensuring that no other bus masters are granted the bus until the locked sequence has completed.

After a sequence of locked transfers the arbiter will always keep the bus master granted for an additional transfer to ensure that the last transfer in the locked sequence has completed successfully and has not received either a SPLIT or RETRY response. Therefore it is recommended, but not mandatory, that the master inserts an IDLE transfer after any locked sequence to provide an opportunity for the arbitration to change before commencing another burst of transfers.

The arbiter is also responsible for asserting the HMASTLOCK signal, which has the same timing as the address and control signals. This signal indicates to any slave that the current transfer is locked and therefore must be processed before any other masters are granted the bus.

2.11.5 Default bus master

Every system must include a default bus master which is granted the bus if all other masters are unable to use the bus. When granted, the default bus master must only perform IDLE transfers.

If no masters are requesting the bus then the arbiter may either grant the default master or alternatively it may grant the master that would benefit the most from having low access latency to the bus.

Granting the default master access to the bus also provides a useful mechanism for ensuring that no new transfers are started on the bus and is a useful step to perform prior to entering a low-power mode of operation.

The default master must be granted if all other masters are waiting for SPLIT transfers to complete.

2.11.6 Split transfers

SPLIT transfers improve the overall utilization of the bus by separating (or splitting) the operation of the master providing the address to a slave from the operation of the slave responding with the appropriate data.

When a transfer occurs the slave can decide to issue a SPLIT response if it believes the transfer will take a large number of cycles to perform. This signals to the arbiter that the master which is attempting the transfer should not be granted access to the bus until the slave indicates it is ready to complete the transfer. Therefore the arbiter is responsible for observing the response signals and internally masking any requests from masters which have been SPLIT.

During the address phase of a transfer the arbiter generates a tag, or bus master number, on HMASTER[3:0] which identifies the master that is performing the transfer. Any slave issuing a SPLIT response must be capable of indicating that it can complete the transfer, and it does this by making a note of the master number on the HMASTER[3:0] signals.

Later, when the slave can complete the transfer, it asserts the appropriate bit, according to the master number, on the HSPLITx[15:0] signals from the slave to the arbiter. The arbiter then uses this information to unmask the request signal from the master and in due course the master will be granted access to the bus to retry the transfer. The arbiter samples the HSPLITx bus every cycle and therefore the slave only needs to assert the appropriate bit for a single cycle in order for the arbiter to recognize it.

In a system with multiple SPLIT-capable slaves the HSPLITx buses from each slave can be ORed together to provide a single resultant HSPLIT bus to the arbiter.

In the majority of systems the maximum capacity of 16 bus masters will not be used and therefore the arbiter only requires an HSPLIT bus which has the same number of bits as there

are bus masters. However, it is recommended that all SPLIT-capable slaves are designed to support up to 16 masters.

The basic stages of a SPLIT transaction are:

1. The master starts the transfer in an identical way to any other transfer and issues address and control information

2. If the slave is able to provide data immediately it may do so. If the slave decides that it may take a number of cycles to obtain the data it gives a SPLIT transfer response.

   During every transfer the arbiter broadcasts a number, or tag, showing which master is using the bus. The slave must record this number, to use it to restart the transfer at a later time.

3. The arbiter grants other masters use of the bus and the action of the SPLIT response allows bus master handover to occur. If all other masters have also received a SPLIT response then the default master is granted.

4. When the slave is ready to complete the transfer it asserts the appropriate bit of the HSPLITx bus to the arbiter to indicate which master should be regranted access to the bus.

5. The arbiter observes the HSPLITx signals on every cycle, and when any bit of HSPLITx is asserted the arbiter restores the priority of the appropriate master.

6. Eventually the arbiter will grant the master so it can re-attempt the transfer. This may not occur immediately if a higher priority master is using the bus.

7. When the transfer eventually takes place the slave finishes with an OKAY transfer response.

Multiple split transfers

The bus protocol only allows a single outstanding transaction per bus master. If any master module is able to deal with more than one outstanding transaction it requires an additional set of request and grant signals for each outstanding transaction that it can handle. At the protocol level a single module may appear as a number of different bus masters, each of which can only have one outstanding transaction.

It is, however, possible that a SPLIT-capable slave could receive more transfer requests than it is able to process concurrently. If this happens then it is acceptable for the slave to issue a SPLIT response without recording the appropriate address and control information for the transfer and it is only necessary for the slave to record the bus master number. The slave can then indicate that it can process another transfer by asserting the appropriate bits on the HSPLITx bus for all masters that the slave has previously SPLIT, but that the slave has not recorded the address and control information.

The arbiter is then able to regrant the masters access to the bus and they will retry the transfer, giving the address and control information required by the slave. This means

that a master may be granted the bus a number of times before it is finally allowed to complete the transfer it requires.

Preventing deadlock

Both the SPLIT and RETRY transfer responses must be used with care to prevent bus deadlock. A single transfer can never lock the AHB as every slave must be designed to finish a transfer within a predetermined number of cycles. However, it is possible for deadlock to occur if a number of different masters attempt to access a slave which issues SPLIT or RETRY responses in a manner which the slave is unable to deal with.

Split transfers

For slaves that can issue a SPLIT transfer response, bus deadlock is prevented by ensuring that the slave can withstand a request from every master in the system, up to a maximum of 16. The slave does not need to store the address and control information for every transfer, it simply needs to record the fact that a transfer request has been made and a SPLIT response issued. Eventually all masters will be at a low priority and the slave can then work through the requests in an orderly manner, indicating to the arbiter which request it is servicing, thus ensuring that all requests are eventually serviced.

When a slave has a number of outstanding requests it may choose to process them in any order, although the slave must be aware that a locked transfer will have to be completed before any other transfers can continue.

It is perfectly legal for the slave to use a SPLIT response without latching the address and control information. The slave only needs to record that a transfer attempt has been made by that particular master and then at a later point the slave can obtain the address and control information by indicating that it is ready to complete the transfer. The master will be granted the bus and will rebroadcast the transfer, allowing the slave to latch the address and control information and either respond with the data immediately, or issue another SPLIT response if a number of additional cycles are required.

Ideally the slave should never have more outstanding transfers than it can support, but the mechanism to support this is required to prevent bus deadlock.

Retry transfers

A slave which issues RETRY responses must only be accessed by one master at a time. This is not enforced by the protocol of the bus and should be ensured by the system architecture. In most cases slaves that issue RETRY responses will be peripherals which need to be accessed by just one master at a time, so this will be ensured by some higher level protocol.

Hardware protection against multiple masters accessing RETRY slaves is not a requirement of the protocol, but may be implemented as described in the following paragraph. The only bus-level requirement is that the slave must drive HREADY HIGH within a predetermined number of clock cycles.

If hardware protection is required then this may be implemented within the RETRY slave itself. When a slave issues a RETRY it can sample the master number. Between that point and the time when the transfer is finally completed the RETRY slave can check every transfer attempt that is made to ensure the master number is the same. If it ever detects that the master number is different then it can take an alternative course of action, such as:

- an ERROR response
- a signal to the arbiter
- a system level interrupt
- a complete system reset

Bus handover with split transfers

The protocol requires that a master performs an IDLE transfer immediately after receiving a SPLIT or RETRY response allowing the bus to be transferred to another master. Figure 11 shows the sequence of events that occur for a split transfer.

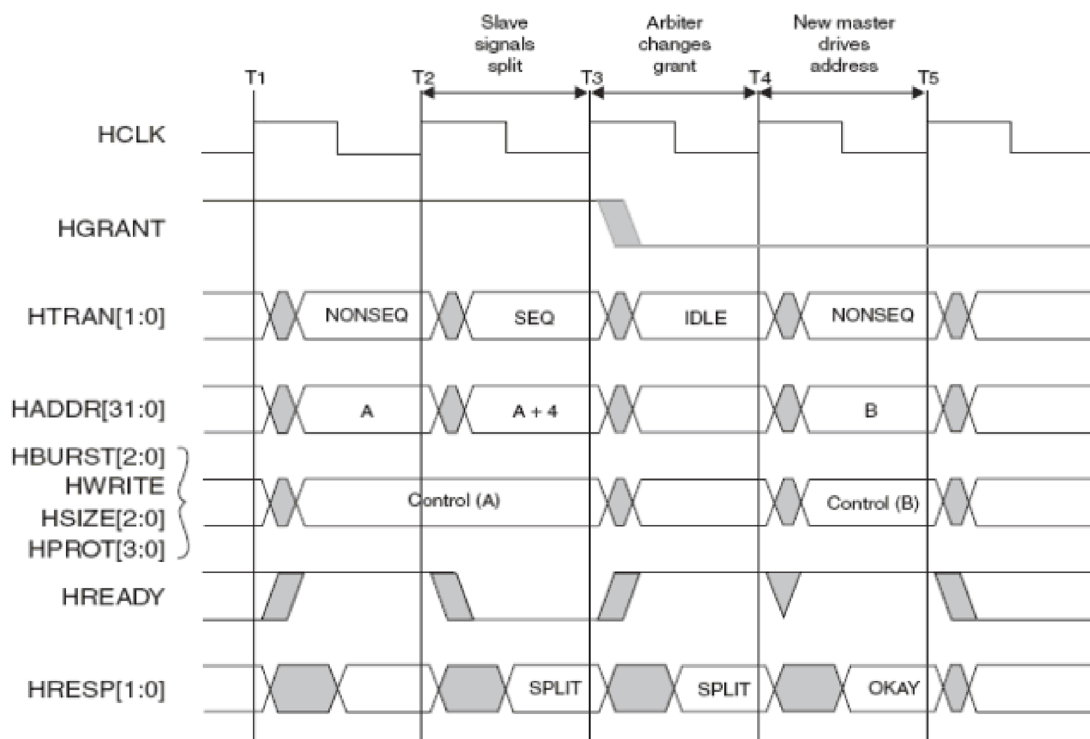


Figure 11

A split transfer improves bus utilization but may cause deadlocks if it's not carefully implemented.

2.11.7 Reset

The reset, HRESETn, is the only active LOW signal in the AMBA AHB specification and is the primary reset for all bus elements. The reset may be asserted asynchronously, but is deasserted synchronously after the rising edge of HCLK.

During reset all masters must ensure the address and control signals are at valid levels and that HTRANS[1:0] indicates IDLE.

2.11.8 AHB AMBA compnents

AHB bus slave

An AHB bus slave responds to transfers initiated by bus masters within the system. The slave uses a HSELx select signal 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, will be generated by the bus master**.**

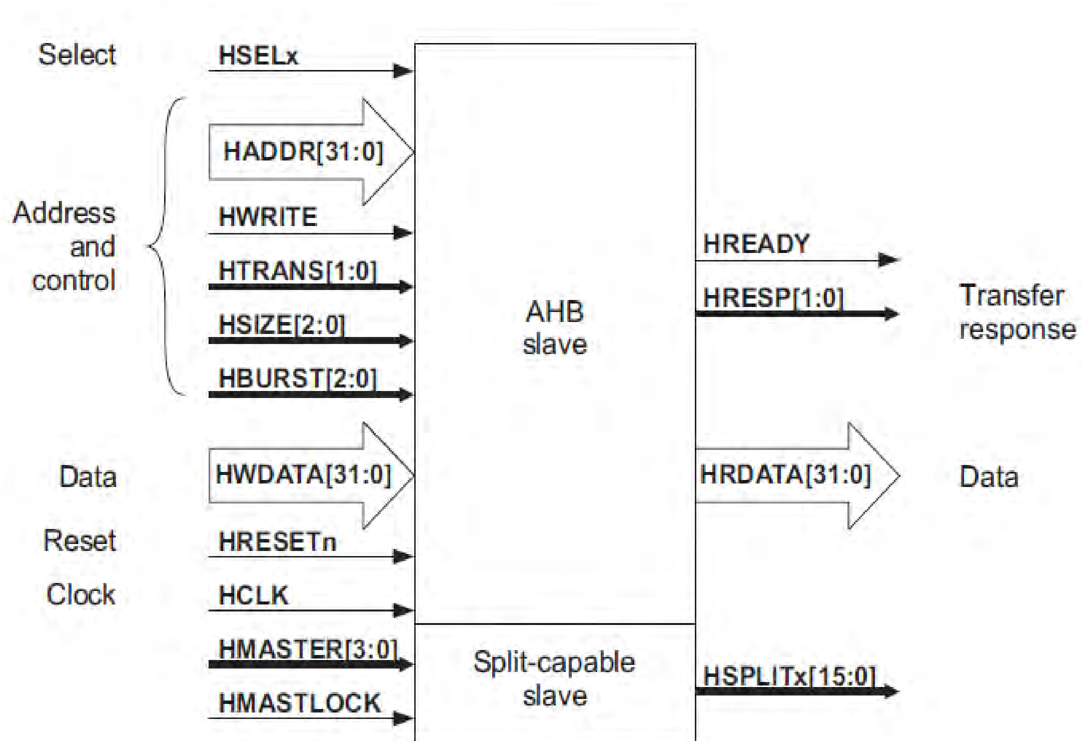


Figure 12 AHB bus slave interface diagram

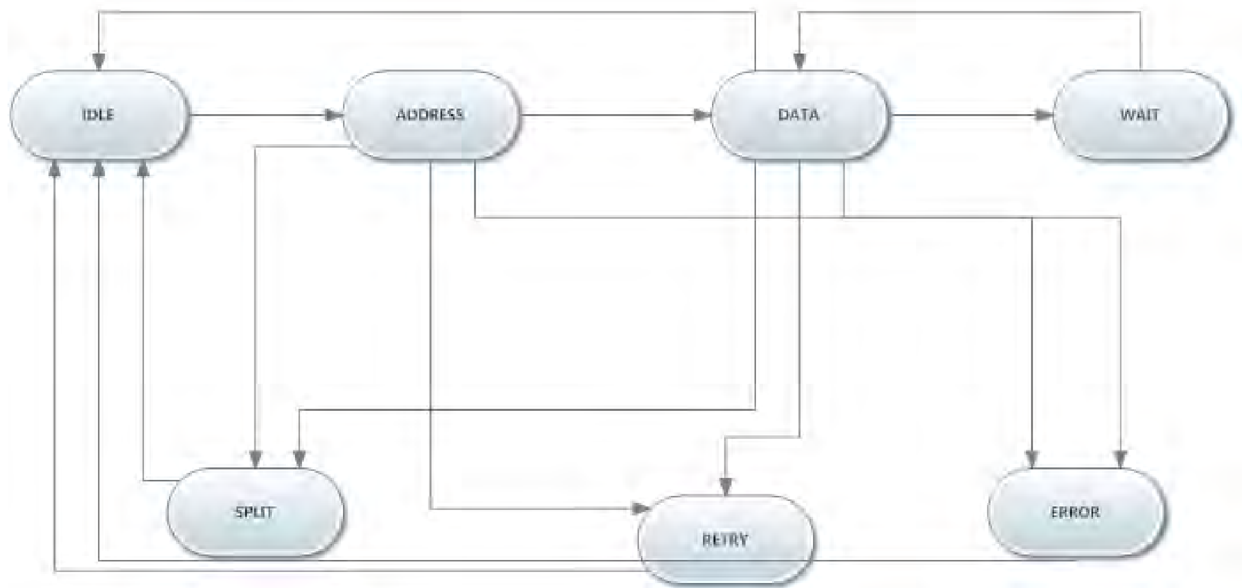


Figure 13 AHB Slave FSM

AHB bus master

An AHB bus master has the most complex bus interface in an AMBA system. Typically an AMBA system designer would use predesigned bus masters and therefore would not need to be concerned with the detail of the bus master interface.

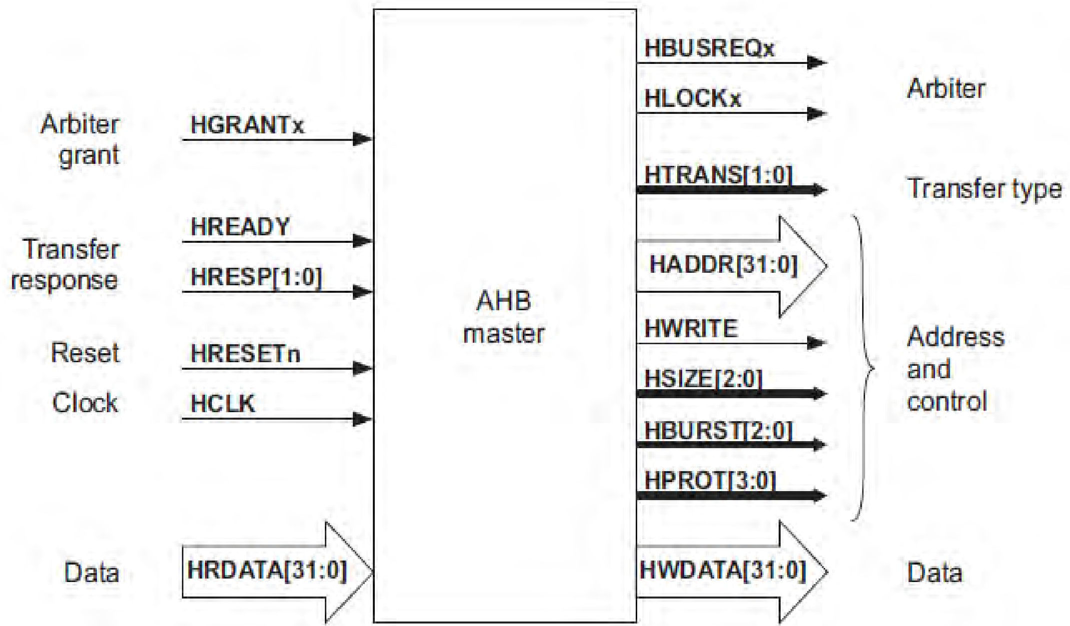


Figure 14 AHB bus master interface diagram

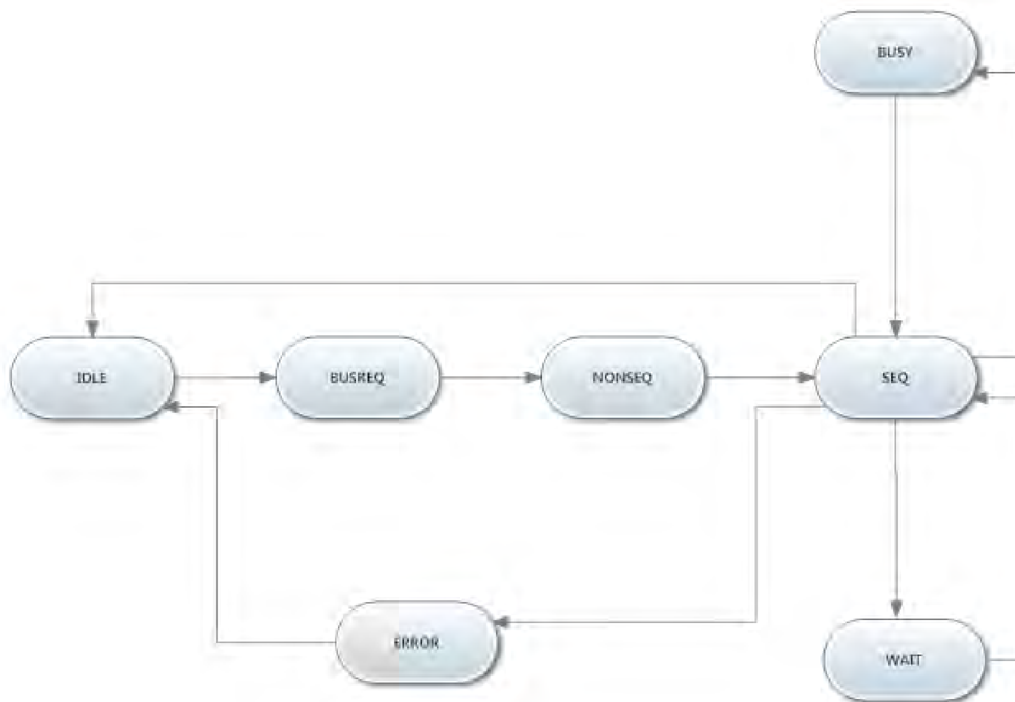


Figure 15 AHB Master FSM

AHB bus arbiter

The role of the arbiter in an AMBA system is to control which master has access to the bus. Every bus master has a REQUEST/GRANT interface to the arbiter and the arbiter uses a prioritization scheme to decide which bus master is currently the highest priority master requesting the bus.

Each master also generates an HLOCKx signal which is used to indicate that the master requires exclusive access to the bus.

The detail of the priority scheme is not specified and is defined for each application. It is acceptable for the arbiter to use other signals, either AMBA or non-AMBA, to influence the priority scheme that is in use.

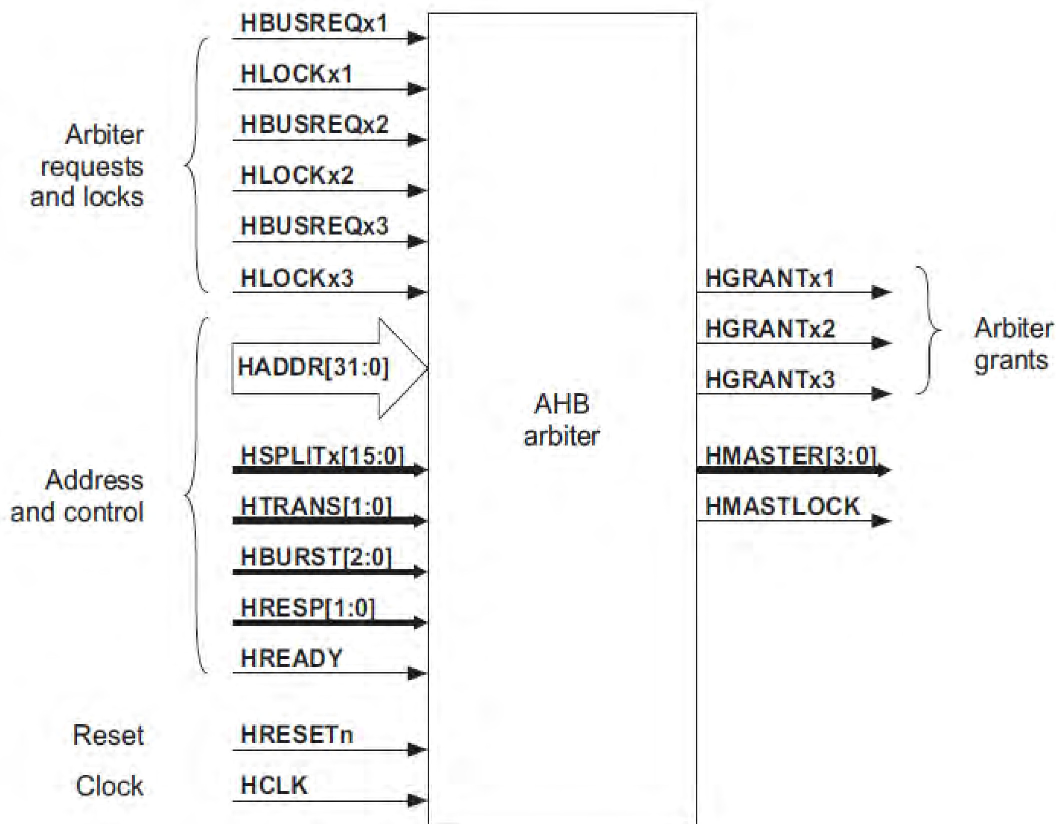


Figure 16 AHB arbiter interface diagram

AHB bus decoder

The decoder in an AMBA system is used to perform a centralized address decoding function, which improves the portability of peripherals, by making them independent of the system memory map.

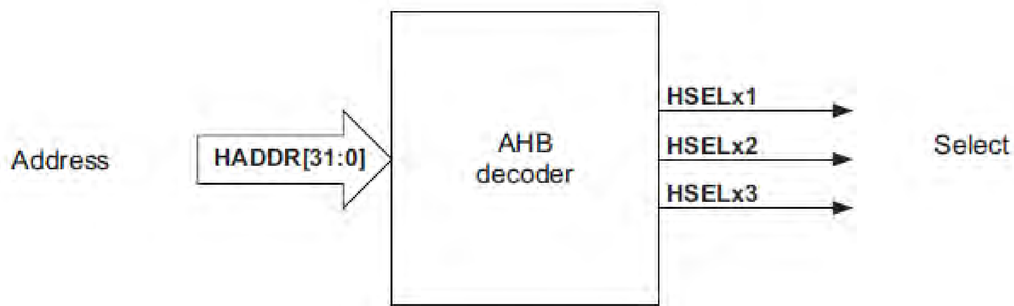


Figure 17 AHB decoder interface diagram

2.11.9 AHB bus matrix topology

In addition to shared bus and hierarchical bus, AHB can be implemented as a bus matrix.

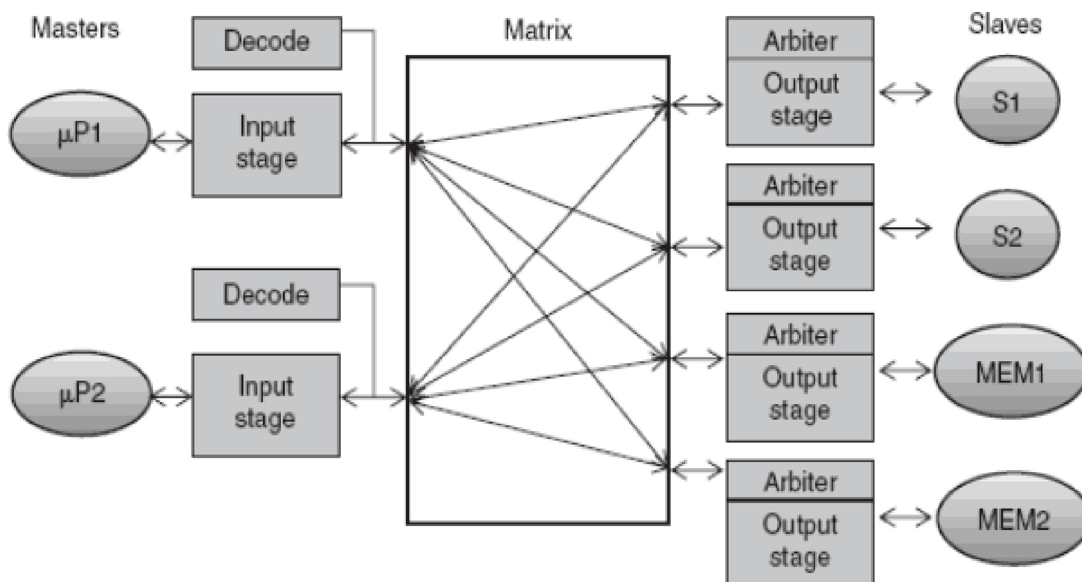


Figure 18 AHB bus matrix topology

3. Advanced Peripheral Bus (APB)

The Advanced Peripheral Bus (APB) is part of the Advanced Microcontroller Bus Architecture (AMBA) hierarchy of buses and is optimized for minimal power consumption and reduced interface complexity.

The AMBA APB should be used to interface to any peripherals which are low bandwidth and do not require the high performance of a pipelined bus interface.

The latest revision of the APB ensures that all signal transitions are only related to the rising edge of the clock. This improvement means the APB peripherals can be integrated easily into any design flow, with the following advantages:

- performance is improved at high-frequency operation
- performance is independent of the mark-space ratio of the clock
- static timing analysis is simplified by the use of a single clock edge
- no special considerations are required for automatic test insertion
- many Application-Specific Integrated Circuit (ASIC) libraries have a better selection of rising edge registers
- easy integration with cycle based simulators

These changes to the APB also make it simpler to interface it to the new Advanced High-performance Bus (AHB).

3.1 APB Bridge

The APB bridge is the only bus master on the AMBA APB. In addition, the APB bridge is also a slave on the higher-level system bus.

The bridge unit converts system bus transfers into APB transfers and performs the following functions:

- Latches the address and holds it valid throughout the transfer.
- Decodes the address and generates a peripheral select, PSELx. Only one select signal can be active during a transfer.
- Drives the data onto the APB for a write transfer.
- Drives the APB data onto the system bus for a read transfer.
- Generates a timing strobe, PENABLE, for the transfer.

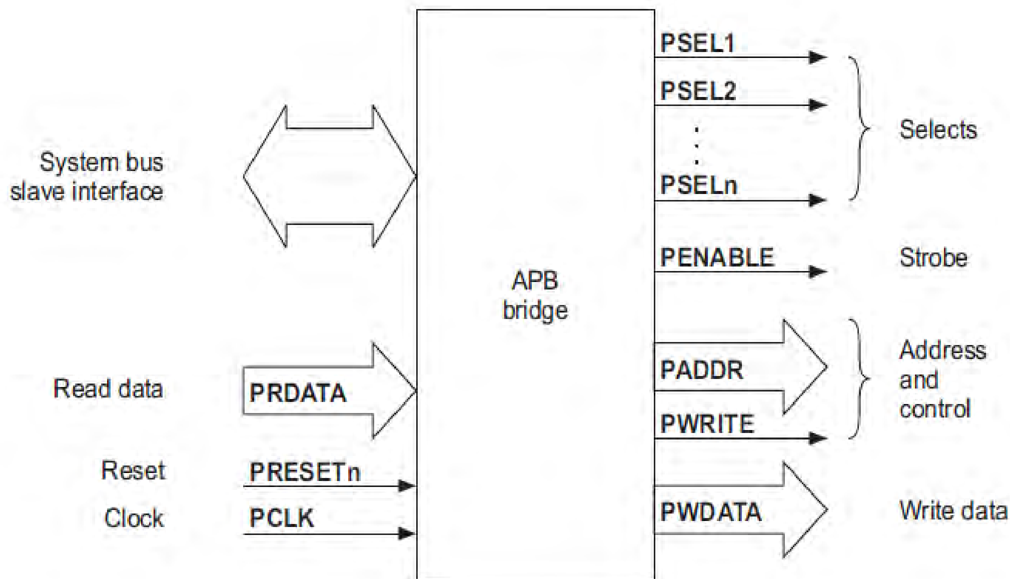


Figure 19 APB bridge interface diagram

3.2 APB Slave

APB slaves have a simple, yet flexible, interface. The exact implementation of the interface will be dependent on the design style employed and many different options are possible.

The APB slave interface is very flexible.

For a write transfer the data can be latched at the following points:

- on either rising edge of PCLK, when PSEL is HIGH
- on the rising edge of PENABLE, when PSEL is HIGH.

The select signal PSELx, the address PADDR and the write signal PWRITE can be combined to determine which register should be updated by the write operation.

For read transfers the data can be driven on to the data bus when PWRITE is LOW and both PSELx and PENABLE are HIGH. While PADDR is used to determine which register should be read.

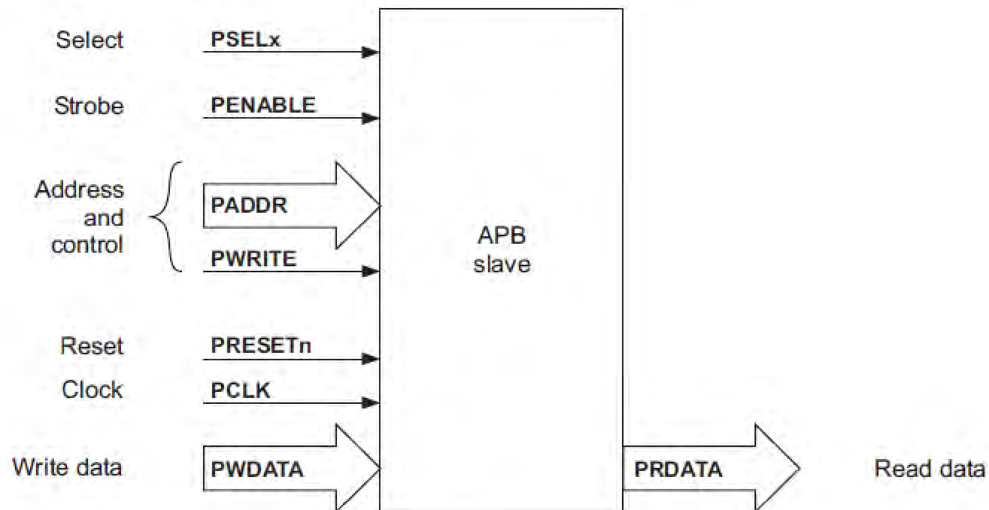


Figure 20 APB Slave interface diagram

3.3 APB State Diagram

The state diagram, shown in Figure 19, can be used to represent the activity of the peripheral bus.

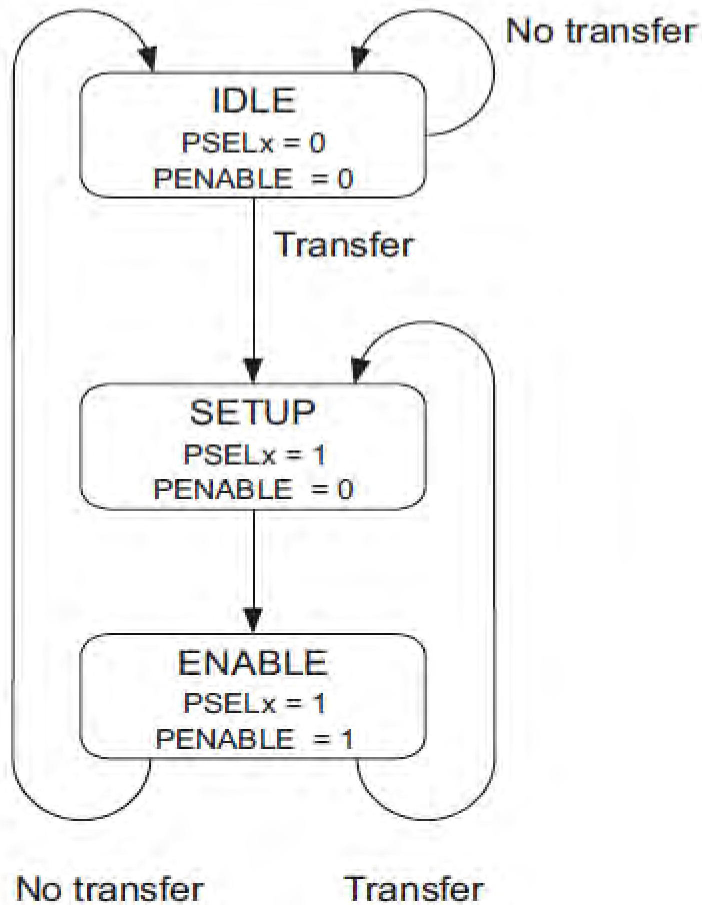


Figure 21 APB FSM

Operation of the state machine is through the three states described below:

IDLE	The default state for the peripheral bus.
SETUP	When a transfer is required the bus moves into the SETUP state, where the appropriate select signal, PSELx, is asserted. The bus only remains in the SETUP state for one clock cycle and will always move to the ENABLE state on the next rising edge of the clock.
ENABLE	In the ENABLE state the enable signal, PENABLE is asserted. The address, write and select signals all remain stable during the transition from the SETUP to ENABLE state.

The ENABLE state also only lasts for a single clock cycle and after this state the bus will return to the IDLE state if no further transfers are required. Alternatively, if another transfer is to follow then the bus will move directly to the SETUP state.

It is acceptable for the address, write and select signals to glitch during a transition from the ENABLE to SETUP states.

3.4 APB Write Transfer

The write transfer starts with the address, write data, write signal and select signal all changing after the rising edge of the clock. The first clock cycle of the transfer is called the SETUP cycle. After the following clock edge the enable signal PENABLE is asserted and this indicates that the ENABLE cycle is taking place. The address, data and control signals all remain valid throughout the ENABLE cycle. The transfer completes at the end of this cycle.

The enable signal, PENABLE, will be deasserted at the end of the transfer. The select signal will also go LOW, unless the transfer is to be immediately followed by another transfer to the same peripheral.

In order to reduce power consumption the address signal and the write signal will not change after a transfer until the next access occurs.

The protocol only requires a clean transition on the enable signal. It is possible that in the case of back to back transfers the select and write signals may glitch.

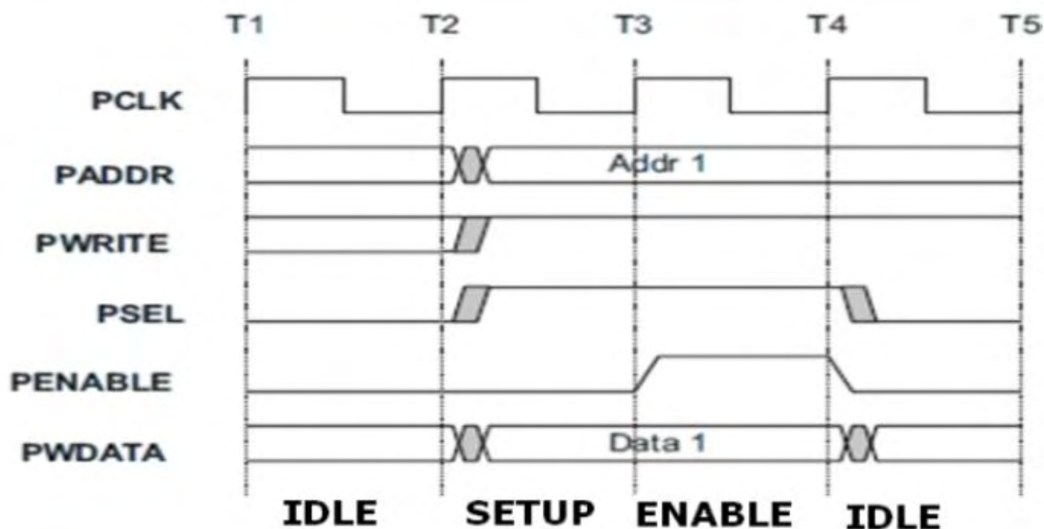


Figure 22 APB write transfer

3.5 APB Read transfer

The timing of the address, write, select and strobe signals are all the same as for the write transfer. In the case of a read, the slave must provide the data during the ENABLE cycle. The data is sampled on the rising edge of clock at the end of the ENABLE cycle.

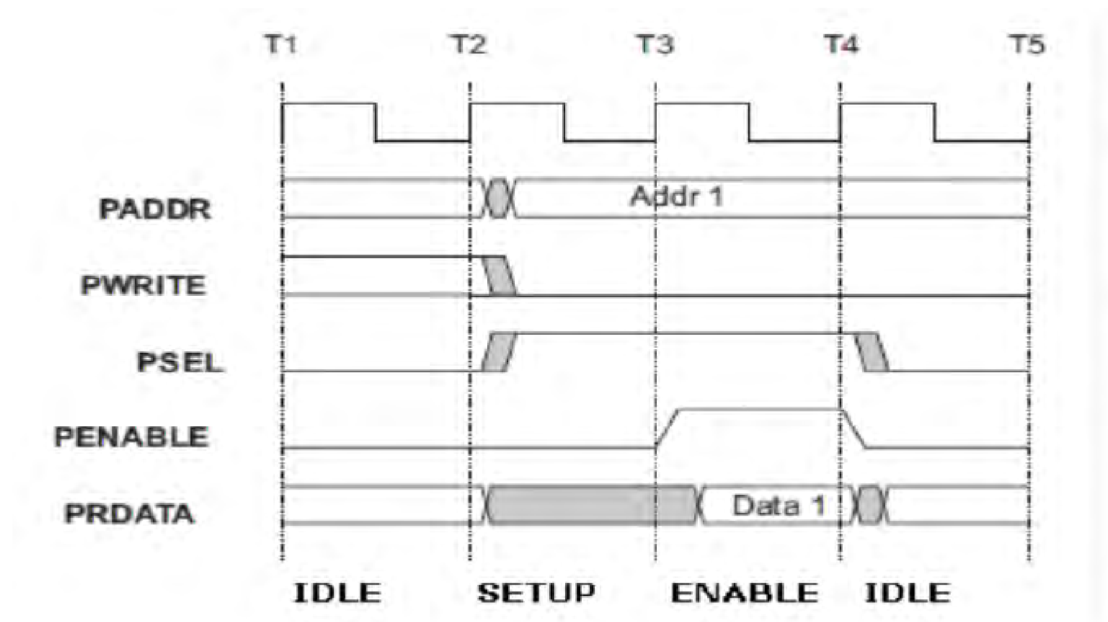


Figure 23 APB read transfer

3.6 AHB vs. APB

When talking of the difference between the two, the AHB uses a full duplex parallel communication whereas the APB uses massive memory-I/O accesses.

Both the AHB and the APB are on chip Bus standards. The Advanced High-performance Bus is capable of waits, errors and bursts. The ADH, which is pipelined, mainly connects to memories.

When comparing the usage, the APB is simpler than the AHB. Unlike the AHB, there is no pipelining in APB. The APB is mainly proposed for connecting to simple peripherals. Looking at the AHB and the APB, it can be seen that the APB comes with a low power peripheral.

It can also be seen that Advanced Peripheral Bus is sometimes optimized for reduced interface complexity and minimal power consumption for supporting peripheral functions. This Bus can also be used in union with either version of the system bus.

When looking at the features of AHB, it has a single edge clock protocol, several bus masters, split transactions, single-cycle bus master handover, burst transfers, large bus widths and non-tristate implementation.

In AHB, the transaction consists of an address phase and a data phase. In case of AHB, there is only one Bus master at a time.

When compared to Advanced High-performance Bus, the Advanced Peripheral Bus is only used for low bandwidth control accesses. Though the APB has an address phase and data phase as like that of the AHB, it comes with a list of low complexity signal.

	AMBA AHB	AMBA APB
Feature	High performance Pipelined operation Multiple bus masters Burst transfers Spilt transactions Single-clock edge operation	Low power Latched address and control Simple interface Single-clock edge operation
Components	AHB master AHB slave AHB arbiter AHB decoder	APB bridge APB slave

Figure 24 AHB vs APB

4 AHB2APB Bridge

The AHB to APB bridge is an AHB slave, providing an interface between the high-speed AHB and the low-power APB. Read and write transfers on the AHB are converted into equivalent transfers on the APB. Because the APB is not pipelined, wait states are added during transfers to and from the APB when the AHB is required to wait for the APB. Figure 25 shows the block diagram of the APB bridge module.

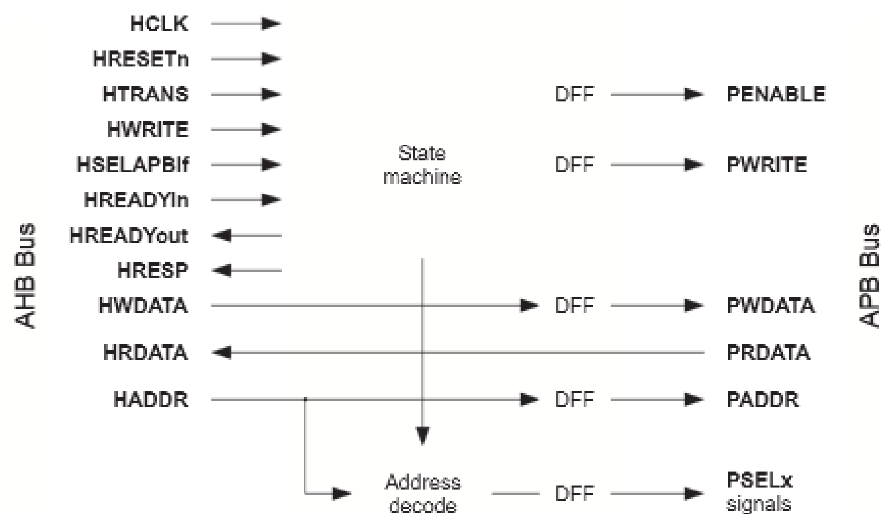


Figure 25 Block diagram of bridge module

The main sections of this module are:

- AHB slave bus interface
- APB transfer state machine, which is independent of the device memory map
- APB output signal generation.

To add new APB peripherals, or alter the system memory map, only the address decode sections have to be modified.

4.1 AHB2APB bridge module signals

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HADDR[31:0]	Address bus	Input	The 32-bit system address bus.
HTRANS[1:0]	Transfer type	Input	This indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE	Transfer direction	Input	When HIGH this signal indicates a write transfer, and when LOW, a read transfer.
HWDATA[31:0]	Write data bus	Input	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended. However, this can easily be extended to allow for higher bandwidth operation.
HSELAPBif	Slave select	Input	Each APB slave has its own slave select signal, and this signal indicates that the current transfer is intended for the selected slave. This signal is a combinatorial decode of the address bus.
HRDATA[31:0]	Read data bus	Output	The read data bus is used to transfer data from bus slaves to the bus master during read operations. A minimum data bus width of 32 bits is recommended. However, this can easily be extended to allow for higher bandwidth operation.
HREADYin **HREADYout**	Transfer done	Input/ output	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal can be driven LOW to extend a transfer.
HRESP[1:0]	Transfer response	Output	The transfer response provides additional information on the status of a transfer. This module always generates the OKAY response.
PRDATA[31:0]	Peripheral read data bus	Input	The peripheral read data bus is driven by the selected peripheral bus slave during read cycles (when **PWRITE** is LOW).
PWDATA[31:0]	Peripheral write data bus	Output	The peripheral write data bus is continuously driven by this module, changing during write cycles (when **PWRITE** is HIGH).
PENABLE	Peripheral enable	Output	This enable signal is used to time all accesses on the peripheral bus. **PENABLE** goes HIGH on the second clock rising edge of the transfer, and LOW on the third (last) rising clock edge of the transfer.
PSELx	Peripheral slave select	Output	There is one of these signals for each APB peripheral present in the system. The signal indicates that the slave device is selected, and that a data transfer is required. It has the same timing as the peripheral address bus. It becomes HIGH at the same time as **PADDR**, but is set LOW at the end of the transfer.
PADDR[31:0]	Peripheral address bus	Output	This is the APB address bus, which can be up to 32 bits wide and is used by individual peripherals for decoding register accesses to that peripheral. The address becomes valid after the first rising edge of the clock at the start of the transfer. If there is a following APB transfer, the address changes to the new value, otherwise it holds its current value until the start of the next APB transfer.
PWRITE	Peripheral transfer direction	Output	This signal indicates a write to a peripheral when HIGH, and a read from a peripheral when LOW. It has the same timing as the peripheral address bus.

4.2 AHB2APB function and operation

The APB bridge responds to transaction requests from the currently granted AHB master. The AHB transactions are then converted into APB transactions. The state machine, shown in Figure 26, controls:

- the AHB transactions with the HREADY out signal
- the generation of all APB output signals.

The individual PSELx signals are decoded from HADDR, using the state machine to enable the outputs while the APB transaction is being performed.

If an undefined location is accessed, operation of the system continues as normal, but no peripherals are selected.

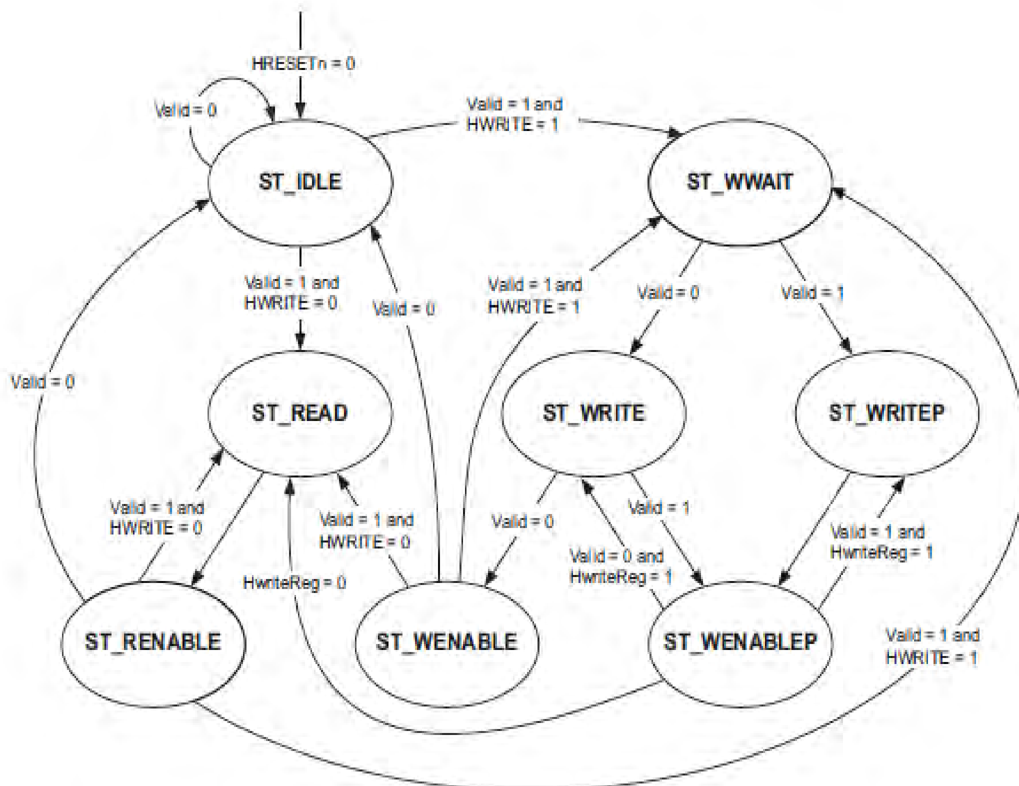


Figure 26 State machine for AHB to APB interface

The individual states of the state machine operation are described in the following sections:

ST_IDLE

During this state the APB buses and PWRITE are driven with the last values they had, and PSEL and PENABLE lines are driven LOW.

The ST_IDLE state is entered from:

- reset, when the system is initialized
- ST_RENABLE, ST_WENABLE, or ST_IDLE, when there are no peripheral transfers to perform.

The next state is:

- ST_READ, for a read transfer, when the AHB contains a valid APB read transfer
- ST_WWAIT, for a write transfer, when the AHB contains a valid APB write transfer.

ST_READ

During this state the address is decoded and driven onto PADDR, the relevant PSEL line is driven HIGH, and PWRITE is driven LOW. A wait state is always inserted to ensure that the data phase of the current AHB transfer does not complete until the APB read data has been driven onto HRDATA.

The ST_READ state is entered from ST_IDLE, ST_RENABLE, ST_WENABLE, or ST_WENABLEP during a valid read transfer.

The next state is always ST_RENABLE.

ST_WWAIT

This state is needed because of the pipelined structure of AHB transfers, to allow the AHB side of the write transfer to complete so that the write data becomes available on HWDATA. The APB write transfer is then started in the next clock cycle.

The ST_WWAIT state is entered from ST_IDLE, ST_RENABLE, or ST_WENABLE, during a valid write transfer.

The next state is always ST_WRITE.

ST_WRITE

During this state the address is decoded and driven onto PADDR, the relevant PSEL line is driven HIGH, and PWRITE is driven HIGH.

A wait state is not inserted, because a single write transfer can complete without affecting the AHB.

The ST_WRITE state is entered from:

- ST_WWAIT, when there are no more peripheral transfers to perform
- ST_WENABLEP, when the currently pending peripheral transfer is a write, and there are no more transfers to perform.

The next state is:

- ST_WENABLE, when there are no more peripheral transfers to perform
- ST_WENABLEP, when there is one more peripheral write transfer to perform.

ST_WRITEP

During this state the address is decoded and driven onto PADDR, the relevant PSEL line is driven HIGH, and PWRITE is driven HIGH. A wait state is always inserted, because there must only ever be one pending transfer between the currently performed APB transfer and the currently driven AHB transfer.

The ST_WRITEP state is entered from:

- ST_WWAIT, when there is a further peripheral transfer to perform.
- ST_WENABLEP, when the currently pending peripheral transfer is a write, and there is a further transfer to perform.

The next state is always ST_WENABLEP.

ST_RENABLE

During this state the PENABLE output is driven HIGH, enabling the current APB transfer. All other APB outputs remain the same as the previous cycle.

The ST_RENABLE state is always entered from ST_READ.

The next state is:

- ST_READ, when there is a further peripheral read transfer to perform
- ST_READ, when there is a further peripheral read transfer to perform
- ST_WWAIT, when there is a further peripheral write transfer to perform
- ST_IDLE, when there are no more peripheral transfers to perform.

ST_WENABLE

During this state the PENABLE output is driven HIGH, enabling the current APB transfer. All other APB outputs remain the same as the previous cycle.

The ST_WENABLE state is always entered from ST_WRITE.

The next state is:

- ST_READ, when there is a further peripheral read transfer to perform
- ST_WWAIT, when there is a further peripheral write transfer to perform
- ST_IDLE, when there are no more peripheral transfers to perform.

ST_WENABLEP

A wait state is inserted if the pending transfer is a read because, when a read follows a write, an extra wait state must be inserted to allow the write transfer to complete on the APB before the read is started.

The ST_WENABLEP state is entered from:

- ST_WRITE, when the currently driven AHB transfer is a peripheral transfer
- ST_WRITEP, when there is a pending peripheral transfer following the current write.

The next state is:

- ST_READ, when the pending transfer is a read
- ST_WRITE, when the pending transfer is a write, and there are no more transfers to perform
- ST_WRITEP, when the pending transfer is a write, and there is a further transfer to perform.

5. Synthesis and Simulation

For the creation of the AHB2APB bridge, source code from OpenCores, European Space Agency and ARM was used.

We used the AHB master and APB master from the open source LEON microprocessor.

- LEON is a 32-bit CPU microprocessor core, based on the SPARC-V8 RISC architecture and instruction set. It was originally designed by the European Space Research and Technology Centre (ESTEC), part of the European Space Agency (ESA) and after that by Gaisler Research. It is described in synthesizable VHDL, the core is configurable through VHDL generics and is used in SoC designs both in research and commercial settings.

OpenCores provided us with the AHB arbiter code in Verilog, which we translated in synthesizable VHDL and ARM provided us the ARM Package declaration.

The AHB2APB bridge we have designed consists of two masters, one arbiter, one bridge and a simple APB memory slave. The AHB slave component is used as the bridge which conveys the signals from AHB to APB bus.

The design was initially compiled and simulated with Ghdl and GTKWave ave respectively. The test bench that we used to test the design generates two write transactions and then issues a read transaction to each location . The simple memory slave component prints a message each time, that is written to or read from.

The Figures show waveforms of the write and read transactions from the GTKWave simulation.

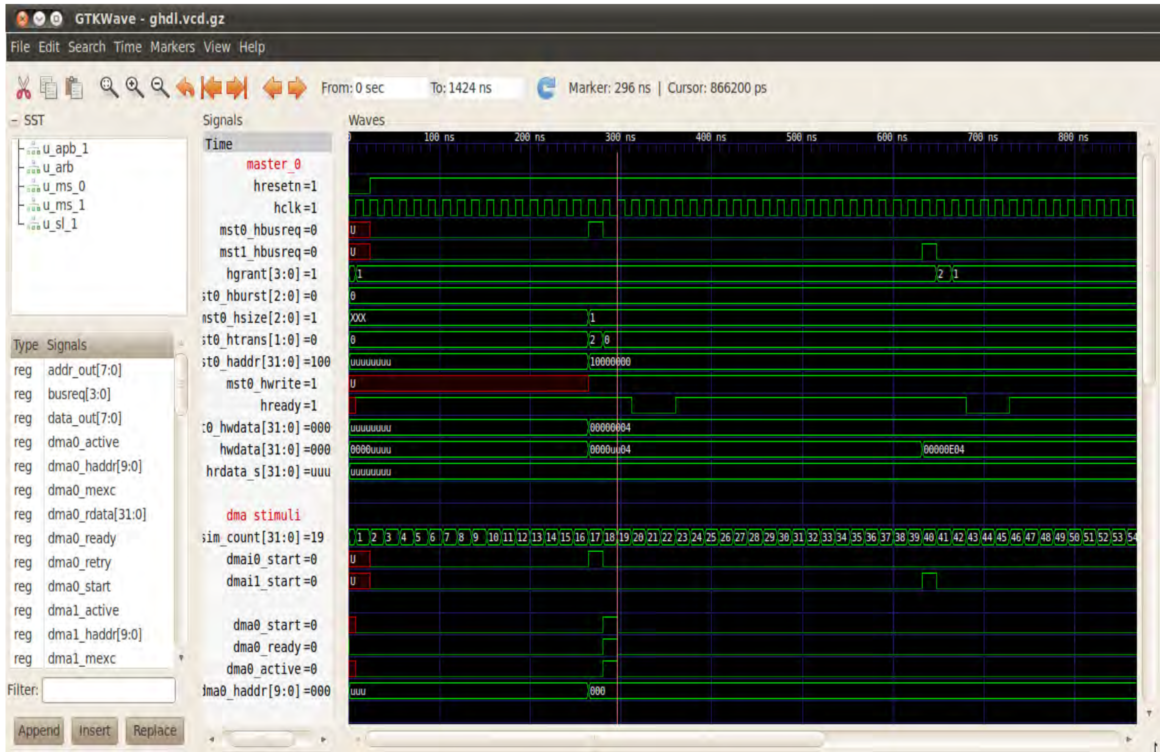


Figure 27 AHB2APB transaction

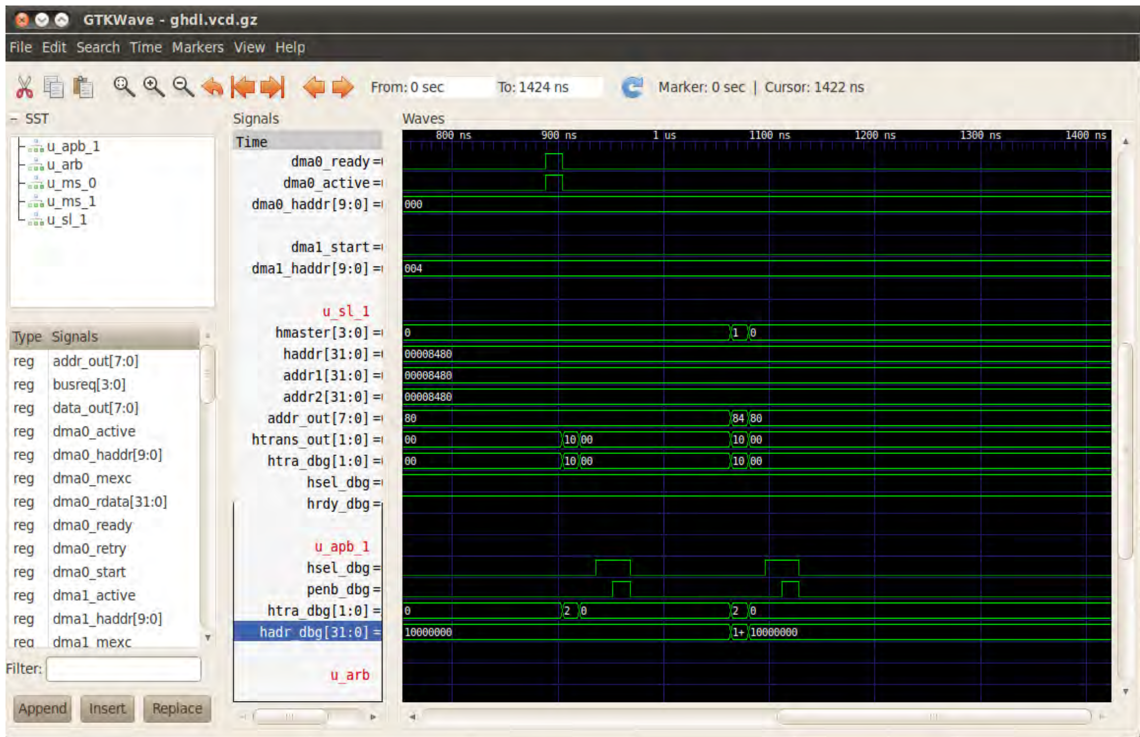


Figure 28 AHB2APB transaction

In order to also have a schematic view of the AHB2APB bridge and observe a transaction, we implemented the design on Incisive Formal Verifier. The following figures show some this implementation.

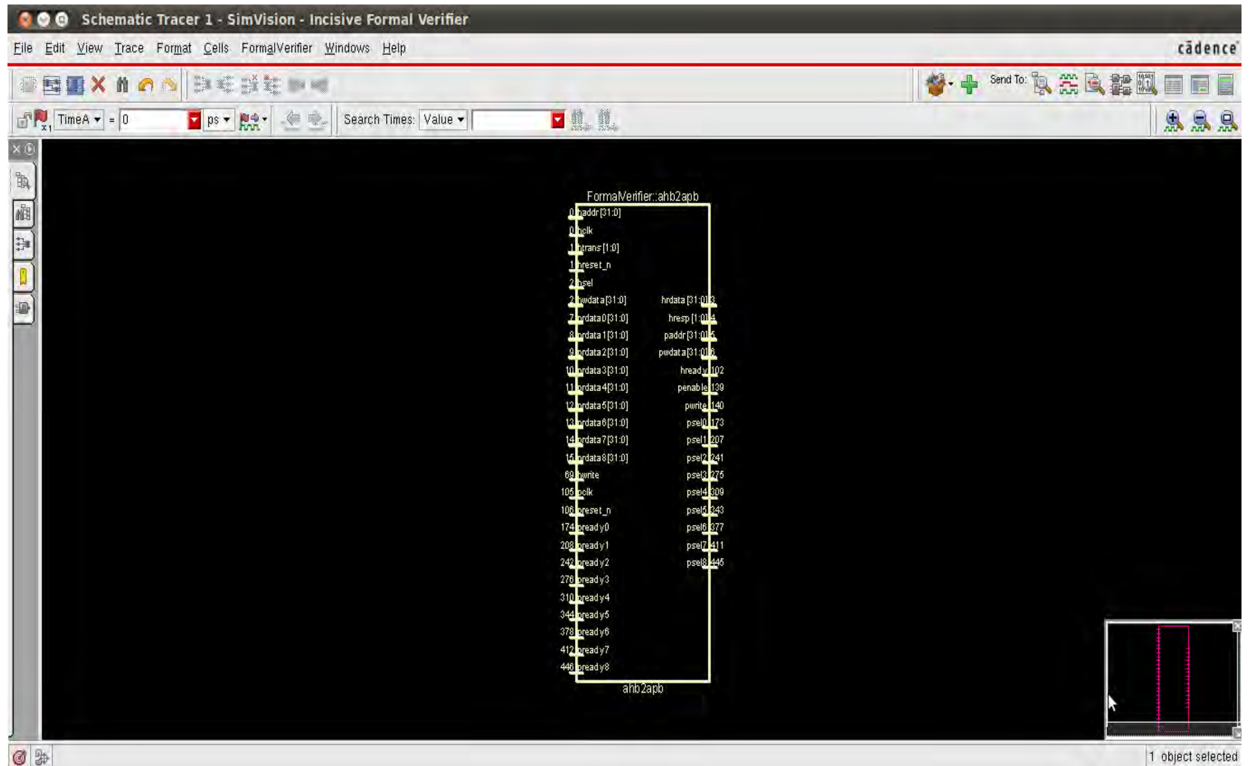


Figure 29 AHB2APB schematic

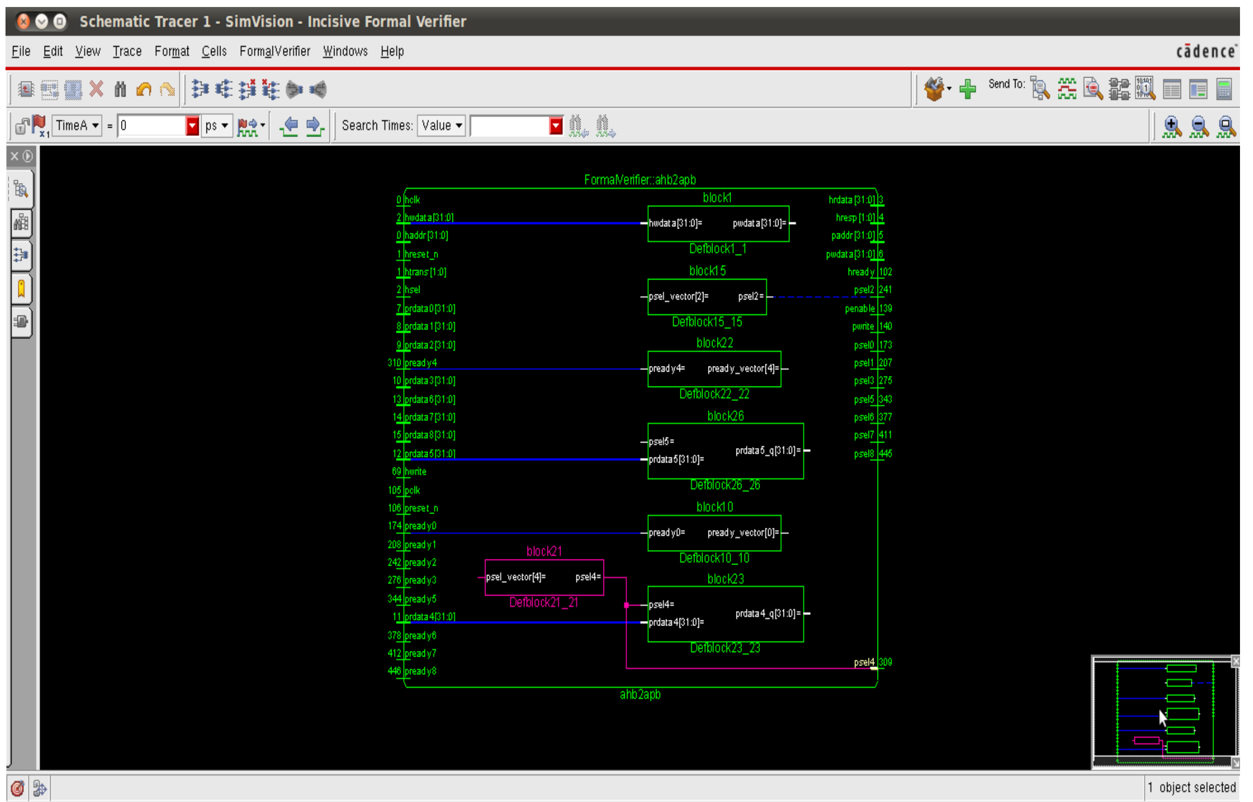


Figure 30 AHB2APB schematic

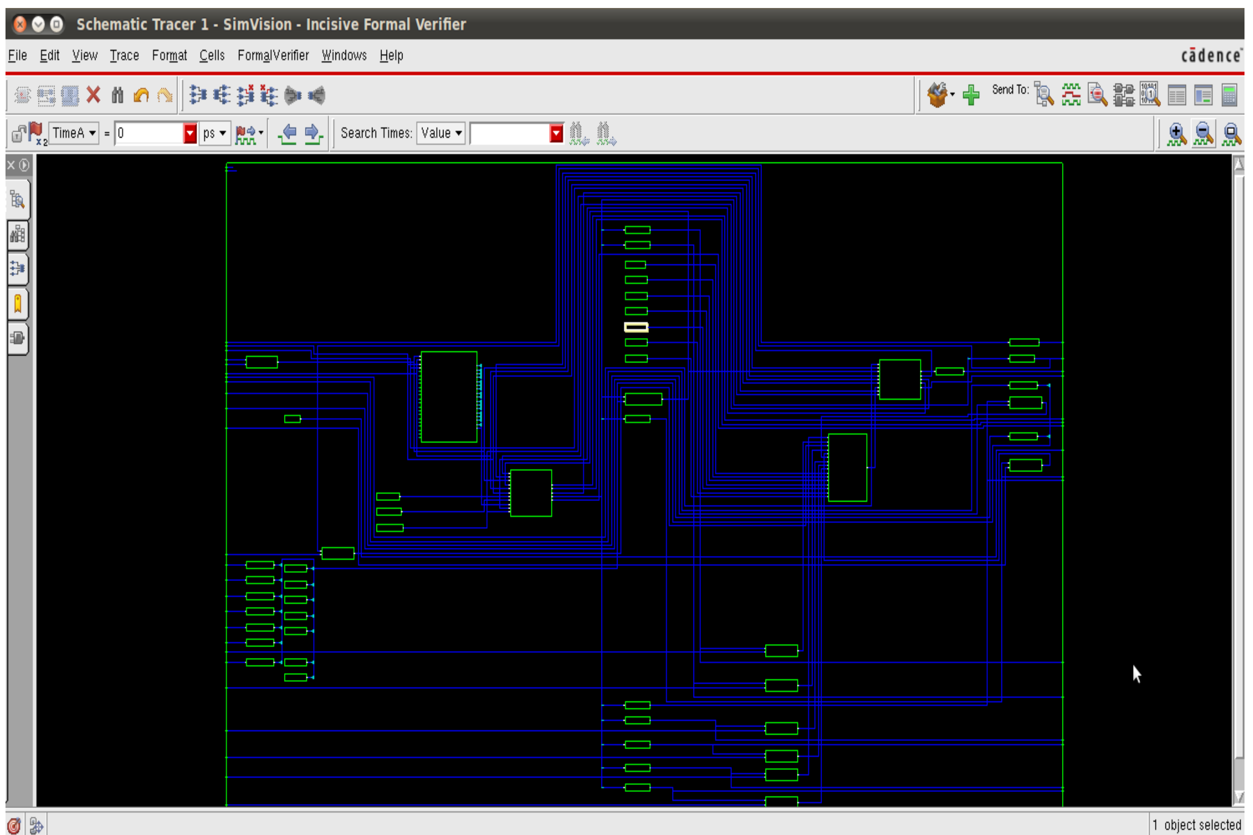


Figure 31 AHB2APB schematic

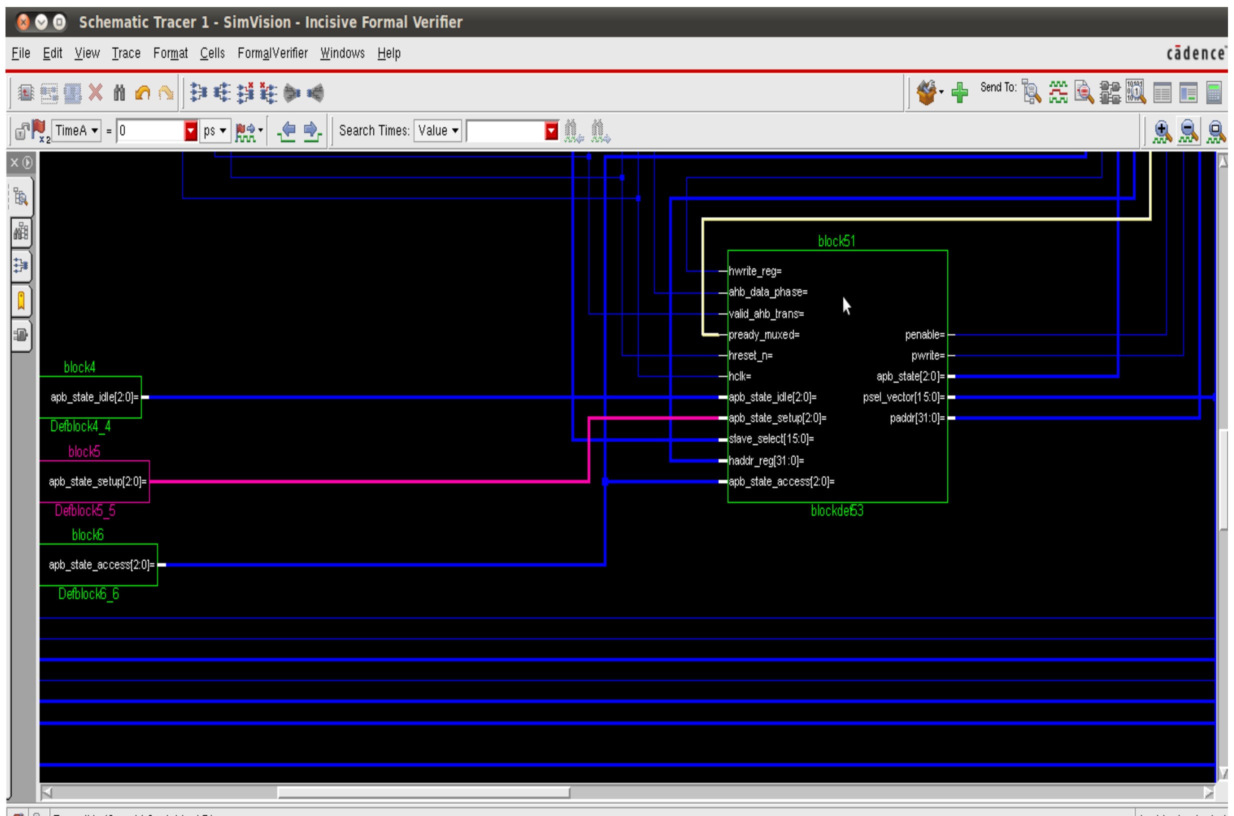


Figure 32 AHB2APB schematic

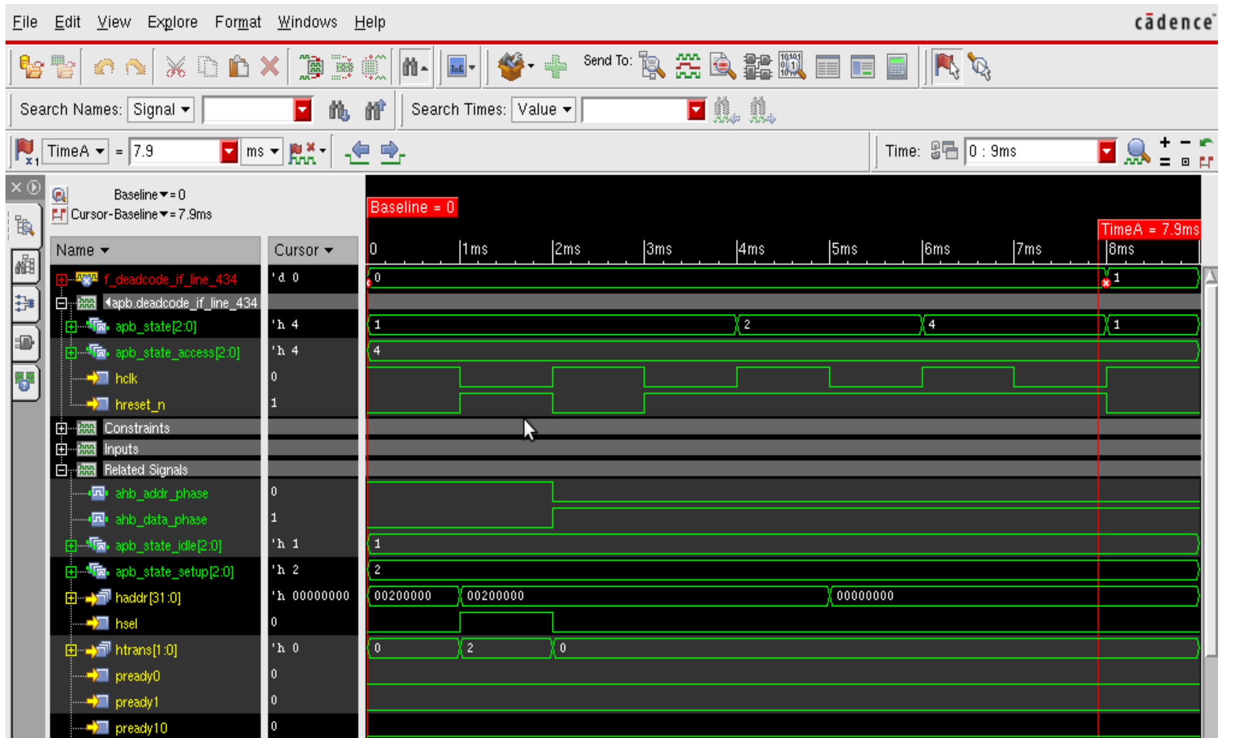


Figure 33 AHB2APB waveform

In order to showcase some of the Incisive Formal Verifier tool features we used the AHB master VHDL code provided by OpenCores as our Design under test.

The following images show the AHB master schematic view, waveform and FSM

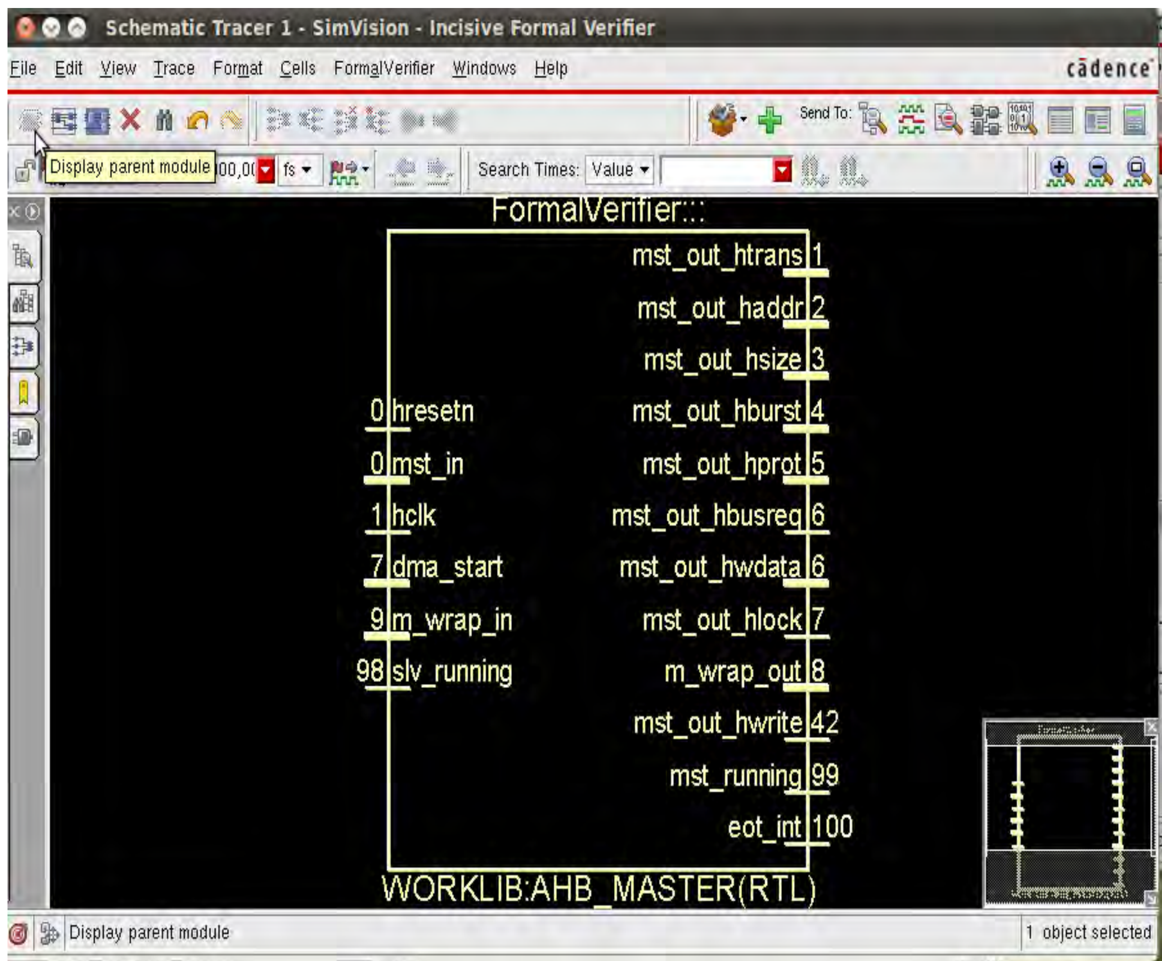


Figure 34 AHB Master schematic

Figure 35 AHB Master waveform

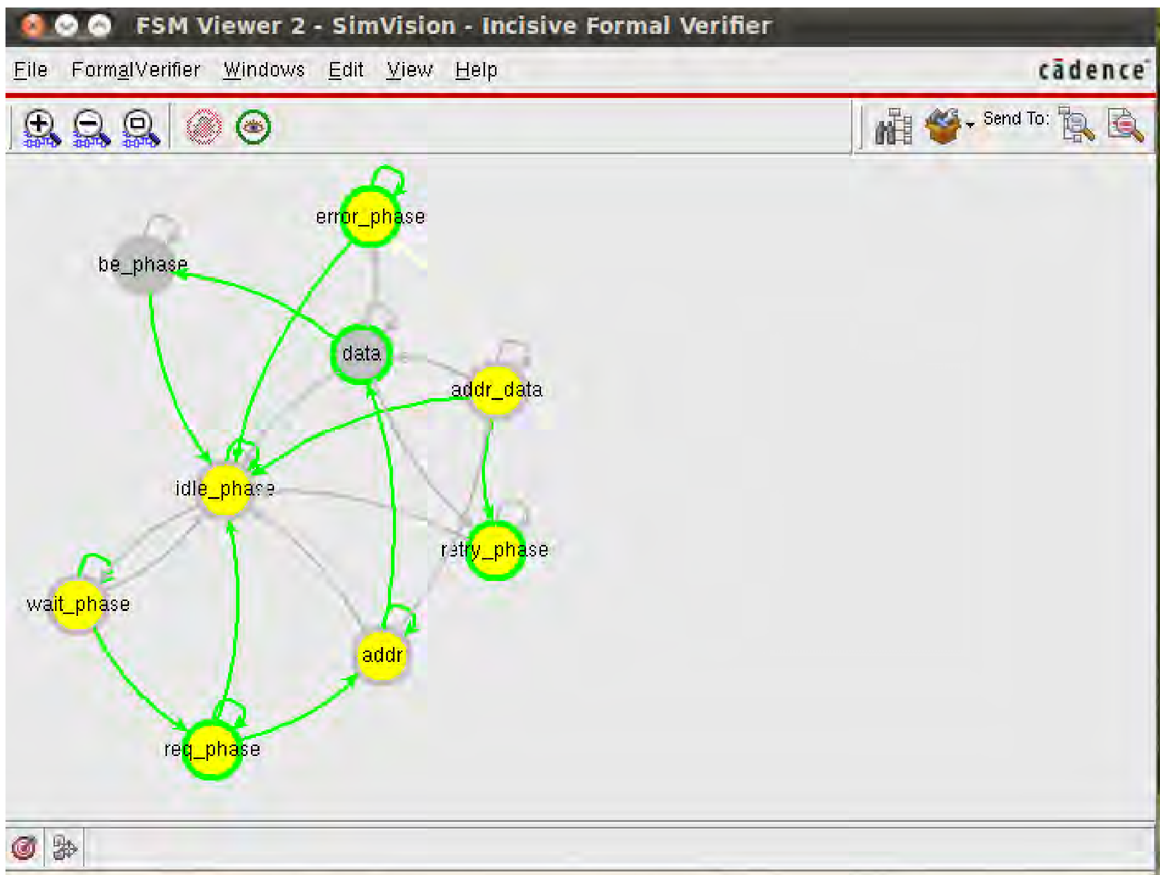


Figure 36 AHB2APB FSM

The last step for this thesis was to check AHB master for the following properties :

- If the transfer is not finished the transfer type should not change
- If a master requests access to the bus, it should eventually be granted

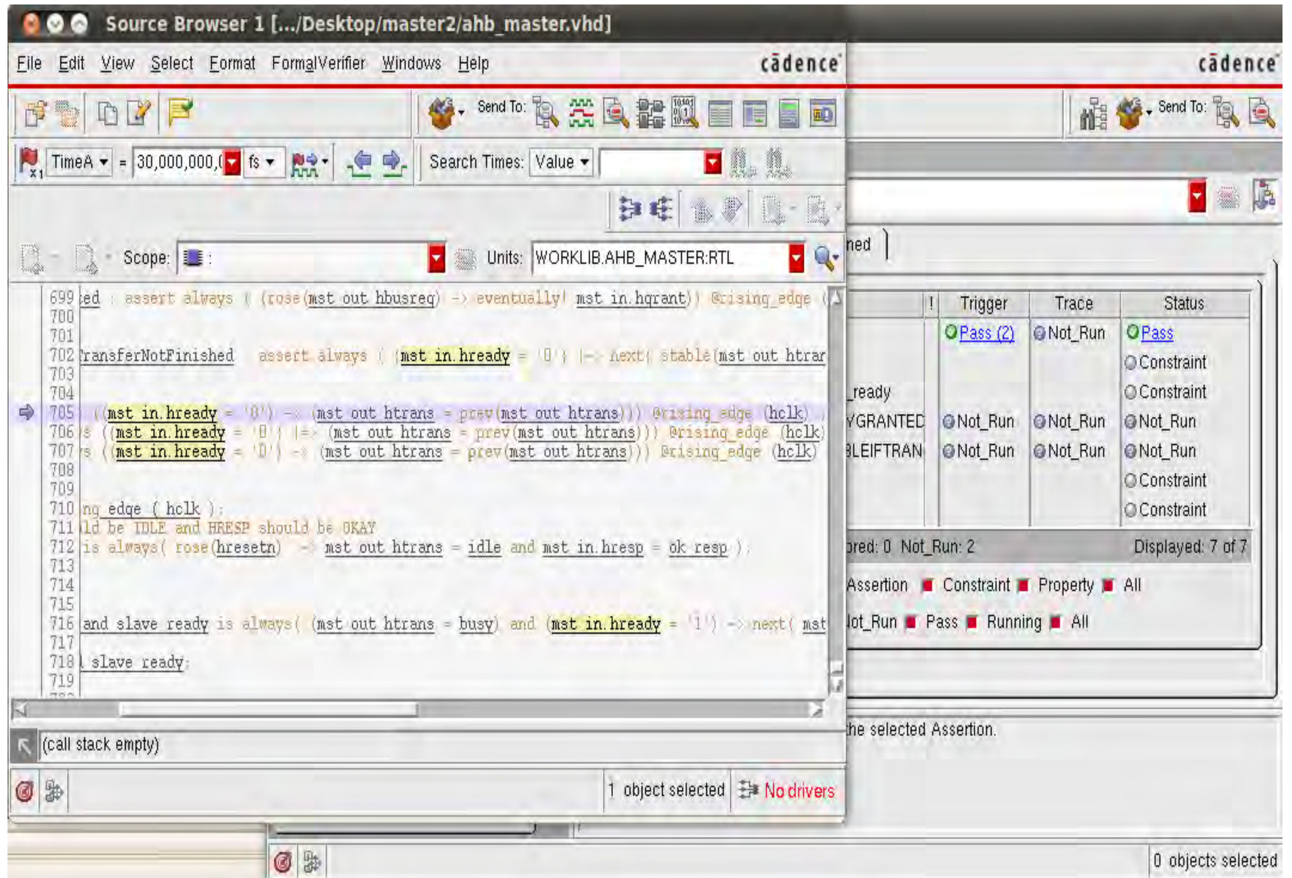


Figure 37 AHB master Property Pass

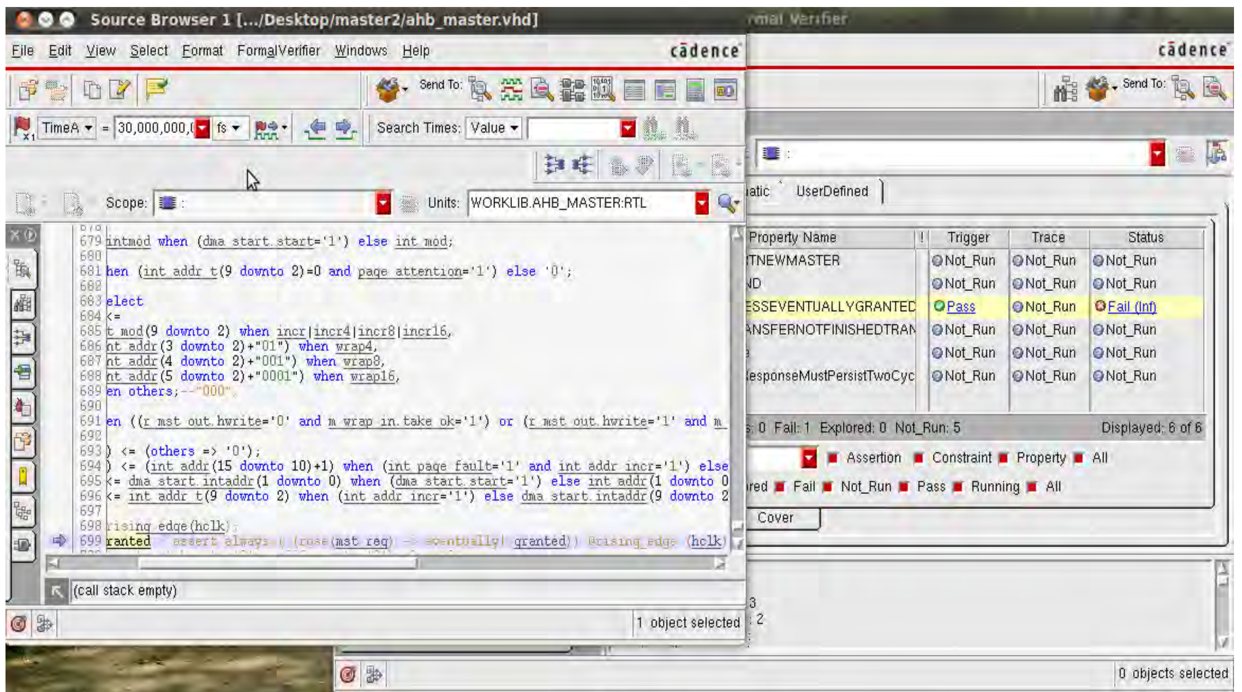


Figure 38 AHB master Liveness Property fails

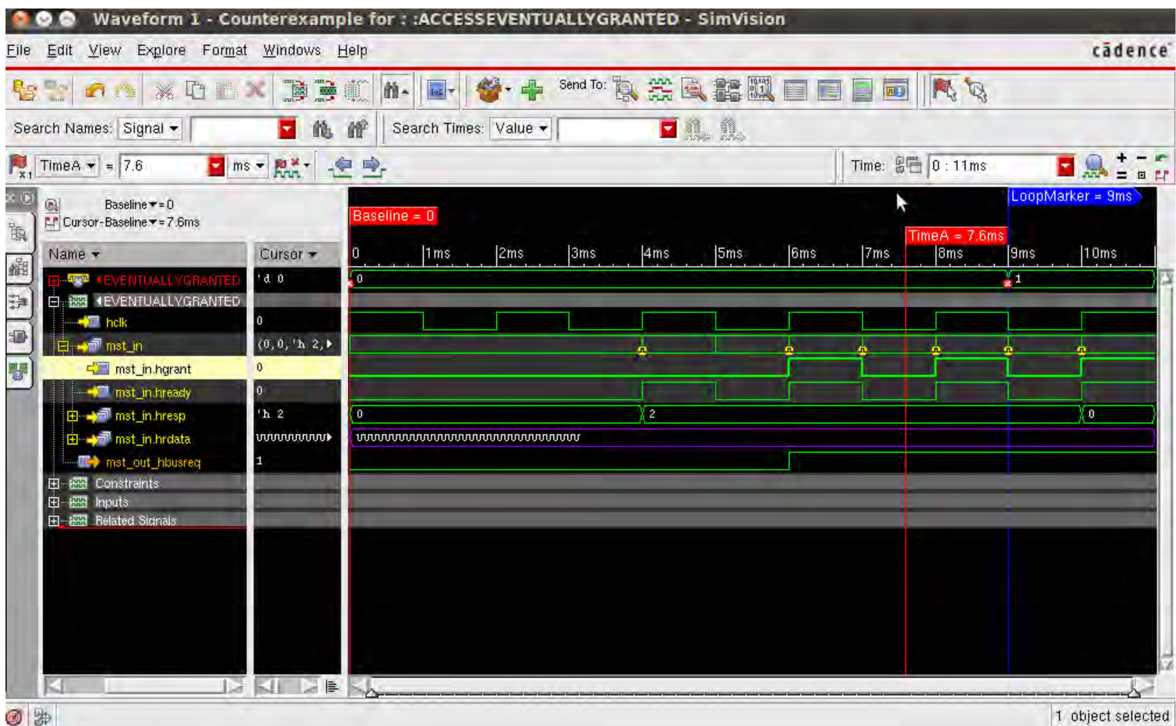


Figure 39 AHB master Liveness Property fails - counterexample

REFERENCES

[1] ARM, AMBA Specifications (Rev 2.0). Available at http://www.arm.com, 1999

[2] ARM, AMBA University Kit Technical Reference Manual. Available at http://www.arm.com, 2001

[3] Cadence, Assertion Checking in Simulation (Rev 10.2) . Available with Incisive Formal Verifier tool installation, 2011

[4] Cadence, Assertion Writing Guide (Rev 10.2) . Available with Incisive Formal Verifier tool installation, 2011

[5] Cadence, Formal Verifier Reference Manual (Rev 10.2) . Available with Incisive Formal Verifier tool installation, 2011

[6] Cadence, Formal Verifier User Guide (Rev 10.2) . Available with Incisive Formal Verifier tool installation, 2011

[7] Cadence, Simulating Your Design (Rev 10.2). Available with Incisive Formal Verifier tool installation, 2011