AMBA Advanced high-performance bus lite

Advanced Microcontroller Bus Architecture (AMBA) overview

What is Advanced Microcontroller Bus Architecture?

AMBA is a standardized design MCU architecture which arm developed to simplify and standardize the design of microprocessor, microcontrollers, peripherals at very different levels of abstraction. It is made to be widely reused in different SoC parts and ASICs utilizing reusability, compatibility, flexibility and support.

Bus interfaces, like AMBA, in general are categorized according to their **Bandwidth & Latency**, therefore amba was made to try and achieve the best possible results in both fields.

AMBA has gradually developed over time, with different components/features being added to it to ensure it being a state-of-the-art architecture which something as impactful as Arm's cortex M & several other influential and widely used paradigms to rely on.

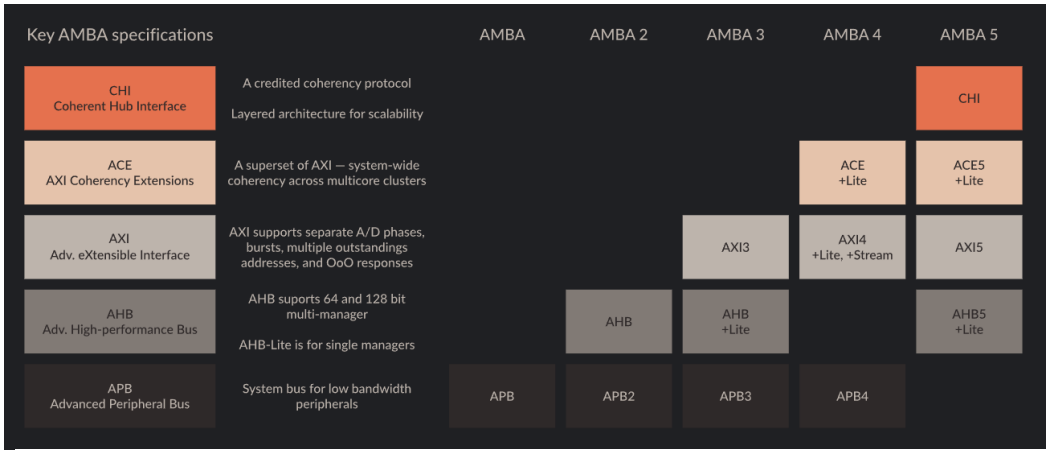


Figure 1: AMBA versions, components & specifications

In this document we're going to focus on AMBA AHB5, discuss its specifications & report on the design & verification

AMBA Advanced High-performance bus (AHB) lite overview

AMBA AHB is a bus interface suitable for **high-performance** synthesizable designs. It defines the interface between components,
such as Managers, interconnects,
and Subordinates.

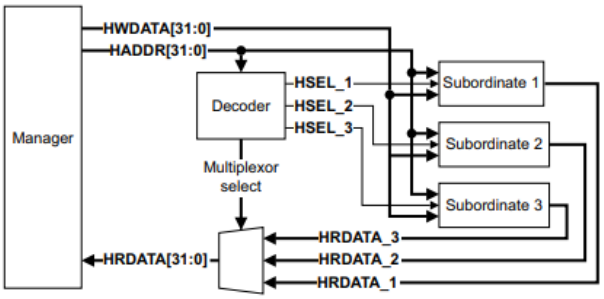


Figure 2
AMBA AHB lite

AHB lite bus constituents:

- Managers
- Subordinates
- Address decoders
- Multiplexors

AMBA AHB implements the features required for high-performance, high clock frequency systems including:

- Burst transfers
- Single clock-edge operation
- Non-tristate implementation
- Configurable data bus widths
- Configurable address bus widths

The most common **AHB Subordinates** are internal memory devices, external memory interfaces, and high-bandwidth peripherals. Although low-bandwidth peripherals can be included as AHB Subordinates, for system performance reasons, they typically reside on the AMBA Advanced Peripheral Bus (APB). Bridging between the higher performance AHB and APB is done using an AHB Subordinate, known as an APB bridge.

AMBA AHB lite Design components

1) Manager

A manager communicates with the subordinate through 2 types of feeds:

- **Control Signals**
- **Data Signals**

AHB also supports pipelined communication to increase throughput and maintain the high-performance paradigm.

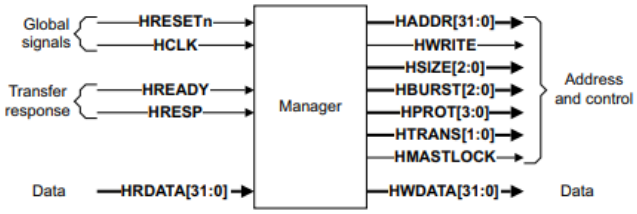


Figure 3
AMBA AHB Manager

2) Subordinate

A subordinate receives transfers from the manager and responds accordingly:

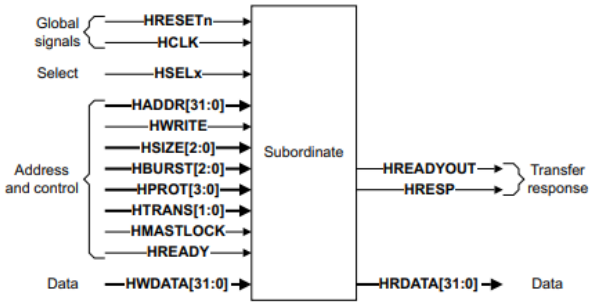


Figure 4
AMBA AHB Subordinate

The Subordinate signals back to the Manager:

- **The completion or extension of the bus transfer.**
- **The success or failure of the bus transfer.**

3) Interconnect

An interconnect component provides the connection between Managers and Subordinates in a system.

In case of Single Master-Multiple Subordinates (AHB lite), a decoder & a multiplexor are used in the interconnect's stead.

- **Decoder**

  The decoder receives the address provided by the Manager and in turn provides asserts the select line HSELx for the subordinate which the transaction is meant for.

  The decoder also sends the same signal to the multiplexor delayed by the number of stages of the pipelining, or the number of cycles needed for the subordinate to provide a response.

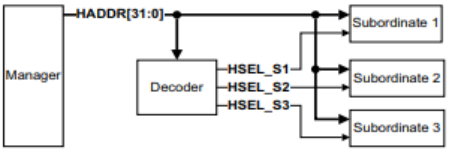


Figure 5
AMBA AHB lite decoder

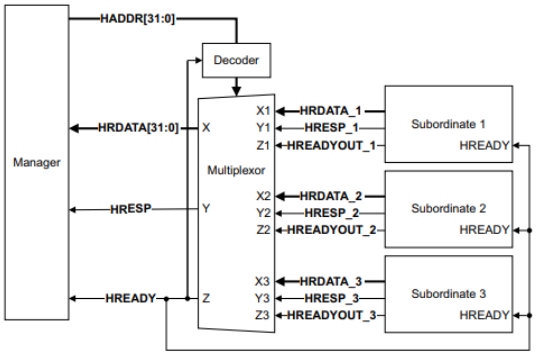


Figure 6
AMBA AHB lite Multiplexor

- **Multiplexor**

  The AHB protocol utilizes a read data multiplexor. After the Manager sends the address and control signals to all the Subordinates, with the decoder selecting the appropriate Subordinate during the data phase of the transfer. Any response data from the selected Subordinate, passes through the read data multiplexor to the Manager.

- **Interconnect with AHB interfaces**

  Generic interconnect products can offer AHB as an interface option, among others such as AMBA AXI or AMBA APB. Figure 4-3 shows how a generic interconnect might implement HTRANS, HREADY, and HSEL.

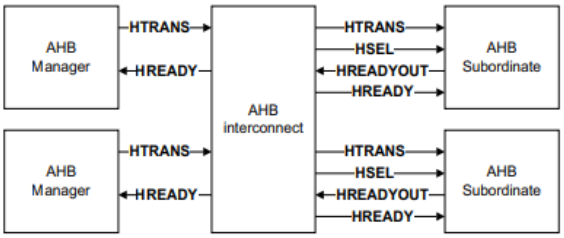
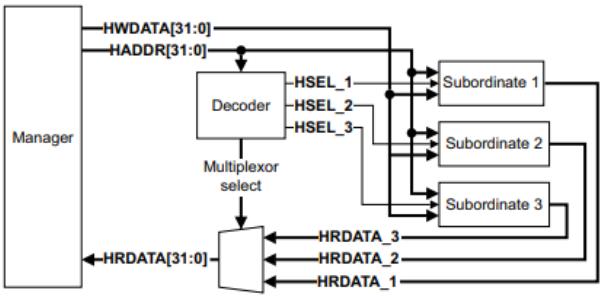


Figure 7
AMBA AHB Interconnect

The Manager side of the interconnect uses HTRANS to indicate valid transfers and has a single HREADY signal. HREADY is used to stall a transfer when the Subordinate has inserted wait states or when the Manager is waiting for arbitration from within the interconnect. The Subordinate side of the interconnect also includes an HSEL output and two HREADY signals. HREADYOUT from the Subordinate is passed to the Managers to insert wait states. The HREADY output from the interconnect can be used to stall a Subordinate if the data phase of the previous transfer is stalled.

An alternative implementation would be for HSEL to be tied HIGH on the Subordinates and the interconnect to override HTRANS to IDLE for unselected Subordinates.

Design of a re-configurable synthesizable pipelined AMBA AHB lite



AMBA Advanced high-performance bus lite

AMBA AHB-lite test-plan

TESTS	DESCRIPTION	NO. BUGS FOUND
1. reset_test	: Asserting reset for 15 clk cycles and checking the outputs	Passed
2. IDLE_test	: Randomizing stimulus and driving **HTRANS = IDLE**	Passed
3. WRITE_SINGLE_test	: Randomizing stimulus and READ from the AHB subordinates **HBURST = SINGLE**	Passed
4. READ_SINGLE_test	: Randomizing stimulus and WRITE to the AHB subordinates **HBURST = SINGLE**	Passed
5. WRITE_INCR_test	: Randomizing stimulus and driving a WRITE with **HBURST = INCR (randomized INCR length)**	Passed
6. READ_INCR_test	: Randomizing stimulus and driving a READ with **HBURST = INCR (randomized INCR length)**	Passed
7. WRITE_READ_INCR_test	: Randomizing stimulus and driving a WRITE then READ with **HBURST = INCR (randomized INCR length)**	Passed
8. WRITE_WRAP4_test	: Randomizing stimulus and driving a WRITE with **HBURST = WRAP4**	Passed
9. READ_WRAP4_test	: Randomizing stimulus and driving a READ with **HBURST = WRAP4**	Passed
10. WRITE_READ_WRAP4_test	: Randomizing stimulus and driving a WRITE then READ with **HBURST = WRAP4**	Passed
11. WRITE_INCR4_test	: Randomizing stimulus and driving a WRITE with **HBURST = INCR4**	Passed
12. READ_INCR4_test	: Randomizing stimulus and driving a READ with **HBURST = INCR4**	Passed
13. WRITE_READ_INCR4_test	: Randomizing stimulus and driving a WRITE then READ with **HBURST = INCR4**	Passed
14. WRITE_WRAP8_test	: Randomizing stimulus and driving a WRITE with **HBURST = WRAP8**	Passed
15. READ_WRAP8_test	: Randomizing stimulus and driving a READ with **HBURST = WRAP8**	Passed
16. WRITE_READ_WRAP8_test	: Randomizing stimulus and driving a WRITE then READ with **HBURST = WRAP8**	Passed
17. WRITE_INCR8_test	: Randomizing stimulus and driving a WRITE with **HBURST = INCR8**	Passed
18. READ_INCR8_test	: Randomizing stimulus and driving a READ with **HBURST = INCR8**	Passed
19. WRITE_READ_INCR8_test	: Randomizing stimulus and driving a WRITE then READ with **HBURST = INCR8**	Passed
20. WRITE_WRAP16_test	: Randomizing stimulus and driving a WRITE with **HBURST = WRAP16**	Passed

TESTS	DESCRIPTION	NO. BUGS FOUND
21. READ_WRAP16_test	: Randomizing stimulus and driving a READ with HBURST = WRAP16	Passed
22. WRITE_READ_WRAP16_test	: Randomizing stimulus and driving a WRITE then READ with HBURST = WRAP16	Passed
23. WRITE_INCR16_test	: Randomizing stimulus and driving a WRITE with HBURST = INCR16	Passed
24. READ_INCR16_test	: Randomizing stimulus and driving a READ with HBURST = INCR16	Passed
25. WRITE_READ_INCR16_test	: Randomizing stimulus and driving a WRITE then READ with HBURST = INCR16	Passed

Coverage Groups

Instance based cover groups

- **HWDATA_df_cg: Covering the toggling of each bit of HWDATA.**

The next cover groups are made using an array of instances and are very applicable/easy to implement and use in functional toggling functional coverage:

HWDATA is a re-configurable bus that can be configured as low as 32 bits of width and up to 1024 bits of width. How can we ensure that all its driving pins/wires are working correctly without writing extensive & exhausting code? Through this.

```
covergroup HWDATA_df_cg(input bit [DATA_WIDTH-1:0] position, ref bit [DATA_WIDTH-1:0] vector);
    df: coverpoint (vector & position) != 0;
    option.per_instance = 1;
endgroup : HWDATA_df_cg
```

HOW IT WORKS: Assume HWDATA is a 4-bit width bus

Sample1 at 1st clock edge: HWDATA/vector = 0000 for example,

i	position	vector	result
i = 0	0001	0000	0000
i = 1	0010	0000	0000
i = 0	0100	0000	0000
i = 0	1000	0000	0000

Now the bin of each instance has collected the coverage of ZERO for its respective bit of HWDATA.

Sample2 at 2nd clock edge: HWDATA/vector = 0101 for example,

i	position	vector	Initial result	result
i = 0	0001	0101	0000	0001
i = 1	0010	0101	0000	0000
i = 0	0100	0101	0000	0100
i = 0	1000	0101	0000	0000

Now the bin of each instance collects the coverage again, but this time, only 2 bits of HWDATA got toggled

Sample3 at 3rd clock edge: HWDATA/vector = 1010 for example,

i	position	vector	Initial result	old result	result
i = 0	0001	1010	0000	0001	0000
i = 1	0010	1010	0000	0000	0010
i = 0	0100	1010	0000	0100	0000
i = 0	1000	1010	0000	0000	1000

Now the bin of each instance collects the coverage again, but this time, the remaining bits of HWDATA got toggled.

therefore achieving 100% coverage for toggling for wide buses.

1. Making the cover group body:

```
covergroup HWDATA_df_cg(input bit [DATA_WIDTH-1:0] position, ref bit [DATA_WIDTH-1:0] vector);
   df: coverpoint (vector & position) != 0;
   option.per_instance = 1;
endgroup : HWDATA_df_cg
```

2. Making the array of instances, an instance for each bit of the bus:

```
   HWDATA_df_cg HWDATA_df_cg_bits  [DATA_WIDTH-1:0];
```

3. Construct each instance with the variable i being shifted to the left at each loop iteration (0001, 0010, 0100...etc.):

```
    foreach(HWDATA_df_cg_bits[i]) HWDATA_df_cg_bits[i] = new(1'b1<<i,HWDATA_cov);
```

4. Sample each time the write function is called in the coverage collector:

```
    foreach(HWDATA_df_cg_bits[i]) HWDATA_df_cg_bits[i].sample();
```

The previous cover group covered the **toggling of data frame values of the bits of HWDATA bus, this one** covers the **toggling data transition values of the bits of HWDATA** bus.

- **HWDATA_dt_cg: Covering the toggling of each bit of HWDATA.**

```
covergroup HWDATA_dt_cg(input bit [DATA_WIDTH-1:0] position, ref bit [DATA_WIDTH-1:0] vector);
   dt: coverpoint (vector & position) != 0 {
        bins tr[] = (0 => 1, 1 => 0);
    }
   option.per_instance = 1;
endgroup : HWDATA_dt_cg
```

The next cover groups do the same for different busses

- **HADDR_df_cg: Covering the toggling of each bit of HWDATA.**
- **HADDR_dt_cg: Covering the toggling of each bit of HWDATA.**
- **HSEL_df_cg: Covering the toggling of each bit of HWDATA.**
- **HSEL_dt_cg: Covering the toggling of each bit of HWDATA.**

```
covergroup HADDR_df_cg(input bit [ADDR_WIDTH-BITS_FOR_SUBORDINATES-1:0] position, ref bit [ADDR_WIDTH-BITS_FOR_SUBORDINATES-1:0]
   df: coverpoint (vector & position) != 0;
   option.per_instance = 1;
endgroup : HADDR_df_cg

covergroup HADDR_dt_cg(input bit [ADDR_WIDTH-BITS_FOR_SUBORDINATES-1:0] position, ref bit [ADDR_WIDTH-BITS_FOR_SUBORDINATES-1:0]
   dt: coverpoint (vector & position) != 0 {
        bins tr[] = (0 => 1, 1 => 0);
    }
   option.per_instance = 1;
endgroup : HADDR_dt_cg

covergroup HSEL_df_cg(input bit [BITS_FOR_SUBORDINATES-1:0] position, ref bit [BITS_FOR_SUBORDINATES-1:0] vector);
   df: coverpoint (vector & position) != 0;
   option.per_instance = 1;
endgroup : HSEL_df_cg

 covergroup HSEL_dt_cg(input bit [BITS_FOR_SUBORDINATES-1:0] position, ref bit [BITS_FOR_SUBORDINATES-1:0] vector);
   dt: coverpoint (vector & position) != 0 {
        bins tr[] = (0 => 1, 1 => 0);
    }
   option.per_instance = 1;
 endgroup : HSEL_dt_cg
```

Coverage Groups (cont.)

Instance based cover groups

- **HSIZE_dt_cg: Covering the transitions of the HSIZE input values.**

This one is a bit tricky, since **HSIZE can take values up to 7, which in turn forces HWDATA & HRDATA to WRITE & READ up to 1024 bits of data (depending on the configuration), so for example, HSIZE being 7 while the bus is configured to bit width of 512 would be illegal**, therefore, HSIZE's array of instances width of the covergroup is also controlled by the HWDATA_WIDTH configuration.

5. Making the cover group body:

```
covergroup HSIZE_dt_cg_bits(input bit [SIZE_WIDTH:0] position, ref bit [SIZE_WIDTH:0] vector);
  dt:coverpoint vector {
    bins tr[] = (vector => position);
  }
  option.per_instance = 1;
endgroup : HSIZE_dt_cg_bits
```

6. Making the cover group body:

```
foreach(HSIZE_dt_val_cg_bits[i]) HSIZE_dt_val_cg_bits[i]= new(i, HSIZE_cov);
```

7. Construct each instance with the value of i being the predeterminant to all the possible values of HSIZE

```
foreach(HSIZE_dt_val_cg_bits[i]) HSIZE_dt_val_cg_bits[i]= new(i, HSIZE_cov);
```

8. Sample each time the write function is called in the coverage collector:

```
foreach(HSIZE_dt_val_cg_bits[i]) HSIZE_dt_val_cg_bits[i].sample();
```

Traditional cover groups

- **RESET_covgrp: Covering the changes & transition of HRESETn.**
- **WRITE_covgrp: Covering the changes & transition of HWRITE.**
- **TRANS_covgrp: Covering the changes & transition of HTRANS.**
- **BURST_covgrp: Covering the changes & transition of HBURST.**
- **SIZE_covgrp: Covering the changes & transitions of HSIZE.**
- **SUBORDINATE_SELECT_covgrp: Covering the changes & transitions of HSEL.**
- **ADDR_covgrp: Covering the changes & transitions of HADDR.**
- **HWDATA_covgrp: Covering the changes & transitions of the HWDATA with all its different SIZEs.**

Creation of the cover groups

```
RESET_covgrp         = new;
WRITE_covgrp         = new;
TRANS_covgrp         = new;
BURST_covgrp         = new;
SIZE_covgrp          = new;
SUBORDINATE_SELECT_covgrp = new;
ADDR_covgrp          = new;
HWDATA_covgrp         = new;

foreach(HWDATA_df_cg_bits[i]) HWDATA_df_cg_bits[i] = new(1'b1<<i,HWDATA_cov);
foreach(HWDATA_dt_cg_bits[i]) HWDATA_dt_cg_bits[i] = new(1'b1<<i,HWDATA_cov);

foreach(HADDR_df_cg_bits[i]) HADDR_df_cg_bits[i] = new(1'b1<<i,HADDR_VALID_cov);
foreach(HADDR_dt_cg_bits[i]) HADDR_dt_cg_bits[i] = new(1'b1<<i,HADDR_VALID_cov);

foreach(HSEL_df_cg_bits[i]) HSEL_df_cg_bits[i] = new(1'b1<<i,HSEL_cov);
foreach(HSEL_dt_cg_bits[i]) HSEL_dt_cg_bits[i] = new(1'b1<<i,HSEL_cov);
```

Sampling of the cover groups

```
RESET_covgrp.sample();
WRITE_covgrp.sample();
TRANS_covgrp.sample();
BURST_covgrp.sample();
SIZE_covgrp.sample();
SUBORDINATE_SELECT_covgrp.sample();
ADDR_covgrp.sample();
HWDATA_covgrp.sample();

foreach(HWDATA_df_cg_bits[i]) HWDATA_df_cg_bits[i].sample();
foreach(HWDATA_dt_cg_bits[i]) HWDATA_dt_cg_bits[i].sample();

foreach(HADDR_df_cg_bits[i]) HADDR_df_cg_bits[i].sample();
foreach(HADDR_dt_cg_bits[i]) HADDR_dt_cg_bits[i].sample();

foreach(HSEL_df_cg_bits[i]) HSEL_df_cg_bits[i].sample();
foreach(HSEL_dt_cg_bits[i]) HSEL_dt_cg_bits[i].sample();
```

Assertions REPORT

Feature	Assertion
The **reset assertion duration** should at least be **15 clock cycles**.	***reset_duration_assert***
When reset is asserted, no subordinates should be selected, i.e. **(HADDR = 0)**	**reset_addr_assert**
When HTRANS == IDLE, **HREADY** response should always be **HIGH**	**Idle_ready_assert**
When HTRANS == IDLE, inputs (HSIZE, HBURST, HWRITE) **should all be 0**	Idle_inputs_assert
When a burst transfer is issued, it must be followed by an **IDLE transfer**	**Incr4_idle_assert** **Incr8_idle_assert** **Incr16_idle_assert** **wrap4_idle_assert** **wrap8_idle_assert** **wrap16_idle_assert**
When a burst transfer is issued, **HTRANS** must be == **NONSEQ**	**burst_trans_nonseq_assert**
When a burst transfer is issued, **HTRANS** must be **== SEQ AFTER 1 CYCLE** (except for INCR burst)	**burst_trans_seq_assert**

P.S. Check the coverage reports.

Commented [AY1]: @(posedge clk) $fell(HRESETn) |=> ##15 $rose(HRESETn);

Commented [AY2]: @(posedge clk) ~HRESETn |-> HADDR == 0;

Commented [AY3]: @(posedge clk) HTRANS == 0 |=> ##3 (HREADY);

Commented [AY4]: @(posedge clk) HTRANS == 0 |-> (HSIZE == 0) && (HBURST == 0) && (HWRITE == 0);

Commented [AY5]: @(posedge clk) (($rose(HBURST[0]) || $rose(HBURST[1]) || $rose(HBURST[2]) || $fell(HBURST[0]) || $fell(HBURST[1]) || $fell(HBURST[2])) && (HBURST == 3)) |-> ##4 (HTRANS==0);

Commented [AY6]: @(posedge clk) (($rose(HBURST[0]) || $rose(HBURST[1]) || $rose(HBURST[2]) || $fell(HBURST[0]) || $fell(HBURST[1]) || $fell(HBURST[2])) && (HBURST == 5)) |-> ##8 (HTRANS==0);

Commented [AY7]: @(posedge clk) (($rose(HBURST[0]) || $rose(HBURST[1]) || $rose(HBURST[2]) || $fell(HBURST[0]) || $fell(HBURST[1]) || $fell(HBURST[2])) && (HBURST == 7))|-> ##16 (HTRANS==0);

Commented [AY8]: @(posedge clk) (($rose(HBURST[0]) || $rose(HBURST[1]) || $rose(HBURST[2]) || $fell(HBURST[0]) || $fell(HBURST[1]) || $fell(HBURST[2])) && (HBURST == 2)) |-> ##4 (HTRANS==0);

Commented [AY9]: @(posedge clk) (($rose(HBURST[0]) || $rose(HBURST[1]) || $rose(HBURST[2]) || $fell(HBURST[0]) || $fell(HBURST[1]) || $fell(HBURST[2])) && (HBURST == 4)) |-> ##8 (HTRANS==0);

Commented [AY10]: @(posedge clk) (($rose(HBURST[0]) || $rose(HBURST[1]) || $rose(HBURST[2]) || $fell(HBURST[0]) || $fell(HBURST[1]) || $fell(HBURST[2])) && (HBURST == 6)) |-> ##16 (HTRANS==0);

Commented [AY11]: @(posedge clk) (($rose(HBURST[0]) || $rose(HBURST[1]) || $rose(HBURST[2]) || $fell(HBURST[0]) || $fell(HBURST[1]) || $fell(HBURST[2])) && (HBURST != 0)) |-> ##[0:1] (HTRANS == 2'b10);

Commented [AY12]: @(posedge clk) (HBURST != 0) && (HBURST != 1) && (HTRANS == 2'b10) |=> (HTRANS == 2'b11);