IL2450 - System Level Validation

Lab 2

Testing An AHB Bus Controller

V. 2.1 – April 7, 2009

Purpose of the lab

In this lab you will write a verification plan and different testbenches to test the functionality of a provided AHB controller. Designing an AHB master generating random stimuli will be the essential part of lab. From this lab you will get practice of both random constraints, used for the creation of the stimuli, and functional coverage, used to get coverage figures from the testbench.

Part 1 – AMBA AHB bus

This part of the lab contains an introduction to the AMBA bus. Be sure to go through it carefully as understanding how the bus works is very important to be able to complete correctly the lab.

The Advanced Microcontroller Bus Architecture was introduced in 1996 and is widely used as the on-chip bus for ARM processors. The first AMBA buses were Advanced System Bus (ASB) and Advanced Peripheral Bus (APB). In its 2nd version, ARM introduced AHB, a single clock-edge protocol. This protocol is today a de-facto standard for 32-bit embedded processors because it is well documented and can be used without royalties. In 2003, ARM introduced the 3rd generation of AMBA, including AXI high-performance interconnect. Many non-ARM-based systems use AMBA buses.

The AHB specifications can be found in the AMBA 2.0 specifications (see the literature section of the course website).

Before starting this lab, download the AMBA specifications document and study it. Chapter 1 will provide you with a good overview of AMBA-based systems. It is very important that you read carefully through this part of the document to understand what AMBA is and how it works. Chapter 2 will provide some information on the AMBA signals while chapter 3 is the chapter giving the AHB specifications. Be sure to read through it carefully and to understand the AHB protocol before you carry on with this lab.

Part 2 – Description of the DUV

The DUV you will test in this lab is a VHDL model of an AHB controller (AHB arbiter and decoder). The model is part of a library of opensource IPs called GRLIB and maintained by the Swedish company Gaisler Research. The library can be freely downloaded from the site of the company. The Gaisler library is documented in the GRLIB IP Library User's Manual and in the GRLIB IP Core User's Manual (see the literature section of the course webpage).

Read chapter 1 and sections 5.1 and 5.2 of the GRLIB IP Library User's Manual to get an overview of how the process of designing systems with GRLIB works. Read chapter 5 of the GRLIB IP core user's manual for an overview of GRLIB's AHB controller. The library contains a set of IPs which all have an AMBA interface to communicate through AHB/APB buses. Controllers for AHB and for APB buses are part of the library, along with a bridge to connect the APB bus with the AHB bus. The GRLIB IP library contains LEON3, a SPARC processor developed by Gaisler Research itself in collaboration with the European Space Agency. Other IPs in the library are 32-bit PC133 SDRAM controller, 32-bit PCI bridge with DMA, 10/100/1000 Mbit ethernet MAC, ATA controller, 16/32/64-bits DDR controller, USB-2.0 Debug link, TAP controller, CAN-2.0 core, 8/16/32-bit PROM and SRAM controller, SVGA frame buffer, generic UART, modular timer unit, interrupt controller, a 32-bit GPIO port. Memory and pad generators are available for different technologies such as Virage, Xilinx, UMC, Atmel, Altera, Lattice, and Actel.

Part 3 – Additional readings

This lab mainly focuses on random constraints and code coverage. Additional material on random constraints and code coverage can be found in chapters 13 and 18 of the IEEE 1800 SystemVerilog Language Reference Manual. The ICC Code Coverage User Guide from Cadence contains information on generating code coverage data using the NC simulators. You can also follow the ICC Tutorial (see the literature section of the course webpage).

Before starting with the tasks: setting up the environment

In this lab you will create a testbench for the bus controller of the system. You will create different testbenches in which different kind of platforms will be created, having different numbers of masters and slaves. As for lab 1, you will start by creating your lab environment. Download the files you need for this lab from the course website. Since the files needed for the labs are many, they come in the form of a tar.gz archive. Download the **lab2.tar.gz** archive into your home directory, then untar it by issuing the command:

```
[you@xenon ~]$ tar -xvzf lab2.tar.gz
```

You can now delete the **lab2.tar.gz** archive. A directory called **lab2** will be created and populated with all the files you need to build your system. The **lab2** directory will contain one directory called **lib**, which contains the VHDL files in the GRLIB, and a directory called **designs**. Under the directory **designs**, you will find a subdirectory called **lab2**. Move to that directory. This is your working directory and you will work from here during this lab.

Launch NCLaunch. **Select File → Set Work Library → work**. You have to select this library because the default work library for NCLaunch is a library called **worklib**, but the compilation script you are about to launch assumes that the work library is called work.

You will now run a compilation script that will compile all the entities in the GRLIB. The script is called **compile.ncsim** and was among the files you downloaded from the course website. To source it, type on the nclaunch console prompt:

```
nclaunch> input compile.ncsim
```

Do not worry about warning messages flying by. The compilation of the GRLIB takes some time but you need to do this only once.

Once the compilation of the GRLIB is finished, you must compile the files in your working directory. Start by selecting all the VHDL files that you can see in your NCLaunch file browser and compile them (remember to turn on support for VHDL 93). Then select all the Verilog/SystemVerilog files and compile them (remember to turn on support for SystemVerilog keywords). Again, do not worry about warning messages. The files will compile, but before being able to elaborate the compiled template testbench in **top.v**, you will need to write your own AHB master.

Task 1 – Write a verification plan for the AHB controller (10 pts)

Deliverables: **task1.txt**

As for the first lab, the first task consists in writing a *verification plan* for the AHB controller. You should write a document in which you mention the different test cases you would use to test the system (you can do it hierarchically). For each test case you should mention what is its purpose (what particular properties of the system is every test case going to test), how you would determine the correct response for each test case and how you would implement the tests (what would be the stimuli used as test vectors, how you would check the response). Specify if any of the test case implementations would satisfy more than one test case specification by testing more than one property of the system. Please use a standard text file and call it **task1.txt**.

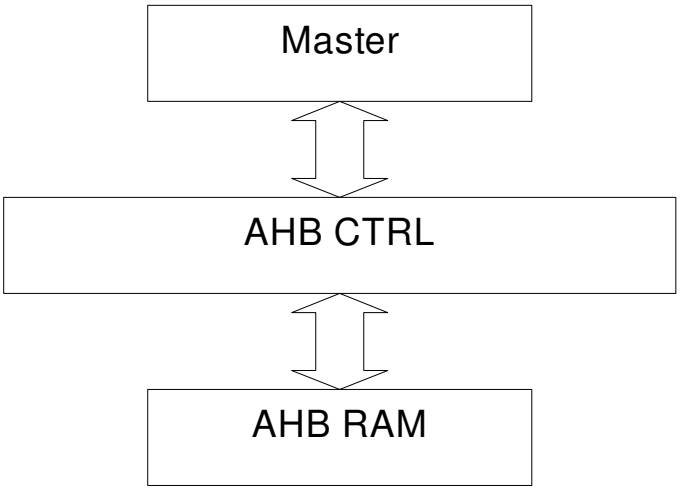
Task 2 – Single-master system (20 pts)

Deliverables: **tb_task2.v**

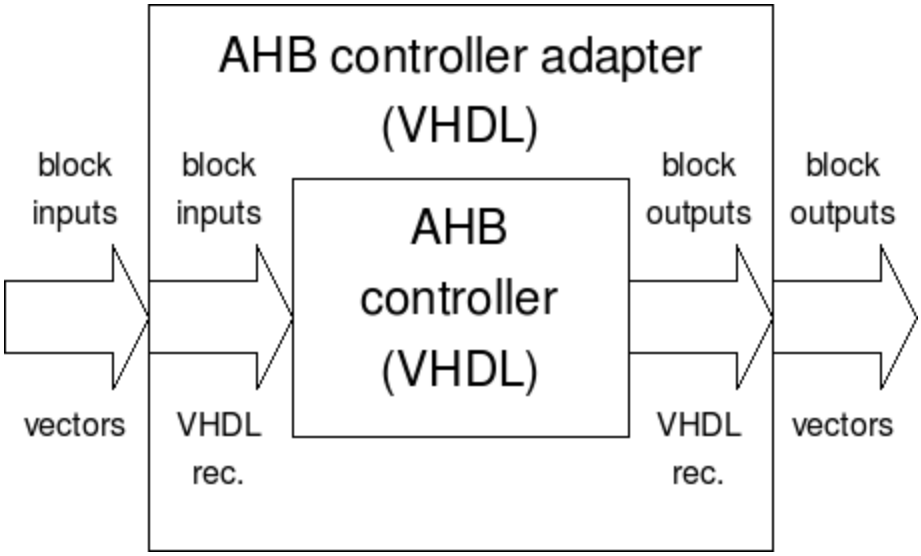
master_task2.v

iccr_task2

The objective of this task is to practice randomization and coverage points. You should write a testbench in which a simple system is instantiated. The system uses the DUV as the bus controller. To the DUV are connected one AHB RAM block and one master that you will have to write. The purpose of the master is to exercise the bus controller by issuing different AHB transactions (acting as a processor that accesses the RAM). The testbench you should create must be contained in a file called **tb_task2.v**. As for the first lab, the testbench must be in the form of a SystemVerilog testbench based on a module including and connecting the different blocks and on a program included by the module. Both the program and the module should be defined in a file called **tb_task2.v**. As part of the lab you should also create an AHB master in a file called **master_task2.v**. The structure of the testbench is shown in the following figure:



The AHB controller and the AHB RAM provided by Gaisler have structured output ports (VHDL records) and wrappers need to be used in order for them to be included in the Verilog testbench. The tools we are currently using, though allowing Verilog/VHDL mixed simulation, have limitations, and do not allow to directly interface VHDL records ports in one entity with structural Verilog ports in another module. The lab assistants have written wrapped RAMs and a wrapped AHB controller for you. The complete structure of the wrappers is shown in the following figure:



The most internal entity is the AHB controller itself provided by Gaisler research. The AHB controller adapter is a VHDL entity which converts the VHDL records inputs/outputs of the AHB controller into VHDL vectors. These outputs can be interfaced with a Verilog environment because the inputs and outputs of the blocks are vectors and not records.

A similar wrapper has been built also for the AHB RAM. Your testbench should contain instances of the wrapped entities, which are called *ahbctrl_w* and *ahbram_w*. The AHB controller and the AHB RAM are contained inside these blocks.

Because of the presence of the wrappers, the AMBA signals, originally in the form of VHDL records, are transformed into vectors, of different dimensions:

- *ahbmsti* (master input) becomes a vector of 84 bits;
- *ahbmsto* (master output) becomes a vector of 5936 bits; this signal combines the output of all the masters in the system. The maximum number of masters is 16, each master is identified by an index from 0 to 15. The output of each slave is a vector or 371 bits. The output of the master with index 0 is connected to the bits 5935 downto 5565, the output of the master with index 1 is connected to the bits 5564 downto 5194 and so on. The master with index 15 is connected to the bits 370 downto 0.
- ahbslvi (slave input) becomes a vector of 136 bits;
- ahbslvo (slave output) becomes a vector of 5504 bits; this signal combines the output of all the slaves in the system. The maximum number of slaves is 16, each slave is identified by an index from 0 to 15. The output of each slave is a vector of 344 bits. The output of the slave with index 0 is connected to the bits 5503 downto 5160, the output of the slave with index 1 is connected to the bits 5159 downto 4816 and so on. The slave with index 15 is connected to the bits 343 downto 0.

A template for the testbench showing the interconnections between the blocks has been written for you by the lab assistants and can be found in the file **top.v**. You will need to write your own AHB master. The master you will write must be connected to the vectorial signal *ahbmsti* and to the vectorial signal *ahbmsto*.

In the following table the correspondence between the signals in the vector and in the record versions of the signal *ahbmsti* are reported.

ahbmsti (vector)	***ahbmsti (record)***
ahbmsti(83 downto 68)	ahbmsti.hgrant
ahbmsti(67)	ahbmsti.hready
ahbmsti(66 downto 65)	ahbmsti.hresp
ahbmsti(64 downto 33)	ahbmsti.hrdata
ahbmsti(32)	ahbmsti.hcache
ahbmsti(31 downto 0)	ahbmsti.hirq

In the following table the correspondence between the signals in the vector and in the record version of the signal *ahbmsto* are reported. Note that each master with index i must drive only the signals corresponding to *ahbmsto(i)*.

ahbmsto (vector)	***ahbmsto(0) (record)***
ahbmsto((15-i) × 371 + 3 downto (15 - i) × 371 + 0)	ahbmsto(i).hindex
ahbmsto((15-i) × 371 + 35 downto (15 - i) × 371 + 4)	ahbmsto(i).hconfig(7)
ahbmsto((15-i) × 371 + 67 downto (15 - i) × 371 + 36)	ahbmsto(i).hconfig(6)
ahbmsto((15-i) × 371 + 99 downto (15 - i) × 371 + 68)	ahbmsto(i).hconfig(5)
ahbmsto((15-i) × 371 + 131 downto (15 - i) × 371 + 100)	ahbmsto(i).hconfig(4)
ahbmsto((15-i) × 371 + 163 downto (15 - i) × 371 + 132)	ahbmsto(i).hconfig(3)
ahbmsto((15-i) × 371 + 195 downto (15 - i) × 371 + 164)	ahbmsto(i).hconfig(2)
ahbmsto((15-i) × 371 + 227 downto (15 - i) × 371 + 196)	ahbmsto(i).hconfig(1)
ahbmsto((15-i) × 371 + 259 downto (15 - i) × 371 + 228)	ahbmsto(i).hconfig(0)
ahbmsto((15-i) × 371 + 291 downto (15 - i) × 371 + 260)	ahbmsto(i).hirq
ahbmsto((15-i) × 371 + 323 downto (15 - i) × 371 + 292)	ahbmsto(i).hwdata
ahbmsto((15-i) × 371 + 327 downto (15 - i) × 371 + 324)	ahbmsto(i).hprot
ahbmsto((15-i) × 371 + 330 downto (15 - i) × 371 + 328)	ahbmsto(i).hburst
ahbmsto((15-i) × 371 + 333 downto (15 - i) × 371 + 331)	ahbmsto(i).hsize
ahbmsto((15-i) × 371 + 334)	ahbmsto(i).hwrite
ahbmsto((15-i) × 371 + 366 downto (15 - i) × 371 + 335)	ahbmsto(i).haddr
ahbmsto((15-i) × 371 + 368 downto (15 - i) × 371 + 367)	ahbmsto(i).htrans

ahbmsto((15-i) × 371 + 369)	ahbmsto(i).hlock
ahbmsto((15-i) × 371 + 370)	ahbmsto(i).hbusreq

Similar correspondences exist also for the signals *ahbslvi* and *ahbslvo*. However, since you do not need to directly drive the lines composing those signals, the correspondence tables are not reported in this manual.

In this first task you need to instantiate a system with only one master and one slave. Both will have index zero. Therefore, as shown in the file **top.v**, your master must be connected only to the first line of the *ahbmsto* vector (5935 to 5565) and your slave must be connected only to the first line of the ahbslvo vector (5503 to 5160). Remember to set to zero all the signals that you do not drive in the *ahbmsto* and *ahbslvo*, just as it is done in the testbench in **top.v**.

The master must drive the AMBA signals and you will need to read the AHB specifications in order to understand how to do that. Remember to set the index of the master (zero), codified on 4 bits, on the signal *ahbmsto(0).hindex*.

The parameters used for the master and the slaves in the **top.v** testbench template set the behaviour of the blocks, you can find information on that in the GRLIB IP Core User's Manual. The first parameter of the list is important: it sets the index of the master or slave.

The AHB master you should write must have the following features (for reference see chapter 3 of the AMBA 2.0 specifications):

1. It generates bus request.
2. It waits until the transaction is completed before making a new transaction request.
3. It chooses randomly to read or write from or to the slave AHBRAM.
4. It can implement the first 4 types of AHB burst transactions (000 to 011), either Read or Write;
5. It generates the random values of AHB master output with following constraints:
    - *HADDR* from h'40000000 to h'400FFF80; (the upper limit has been chosen so that even the biggest transfers will not access locations outside of the allocated memory space)
    - *HBURST* from 000 to 011;
    - *HTRANS* from 00 to 11;
    - *HSIZE* from 000 to 111;
    - *HPROT* is equal to 0111;
    - *HADDR* is word-aligned.

The testbench must implement code coverage with the following coverage points:

- *AHBMI.HGRANT*
- *AHBMO.HADDR*
- *AHBSLVI.HWDATA*
- *AHBSLVO.HADDR*
- *AHBSLVO.HRDATA*

Note also that the AHB Controller and AHB RAM need four clock cycles to initialize. Therefore, your testbench should set the reset signal to 0 for four clock cycles and then set it to 1 at the

beginning of the simulation. The clock signal for all the blocks must also be generated by your testbench.

You should write a script for functional coverage using ICCR. Set the threshold value to 6. Call the script **iccr_task2**, this is another deliverable for this task.

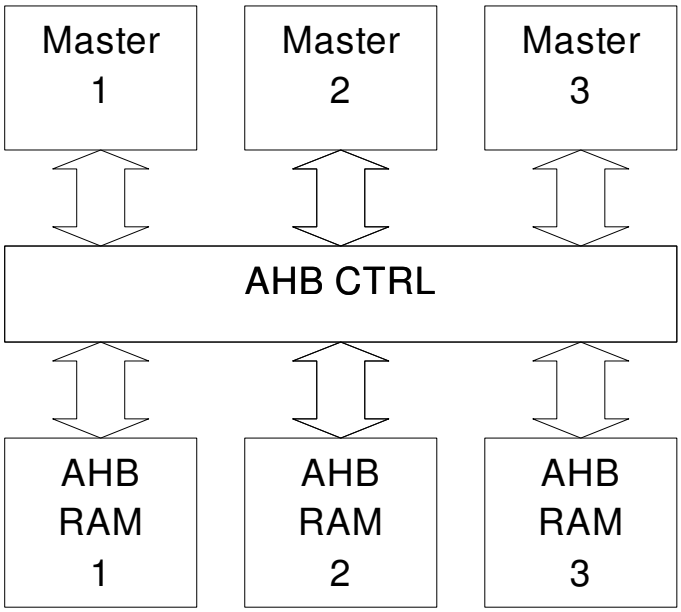
Task 3 – Multiple-master system (20 pts)

Deliverables: **tb_task3.v**

master_task3.v

iccr_task3

In the third task, you must define a SystemVerilog testbench with 3 masters and 3 slaves, as shown in the figure below:



The RAMs must still be 1 MB wide. They must be connected to the three different addresses:

- 40000000 (extends from 40000000 to 400FFFFF)
- 40100000 (extends from 40100000 to 401FFFFF)
- 40200000 (extends from 40200000 to 402FFFFF)

The AHB controller will take care of connecting the three RAMs to the right memory space as long as you set the first to index 0. the second to index 1 and the third to index 2. Remember that the index is determined by the first parameter when you instantiate the RAM. For the other parameters, use the default values that were used in **top.v**.

You also need to connect three AHB masters to the system. Set the index for the first to 0, for the second to 1 and for the third to 2.

The masters and the slaves must be connected to the right bit slices of the *ahbmsto* and *ahbslvo* signals:

- Master 1 (index 0) must be connected to *ahbmsto*(5935 downto 5565)
- Master 2 (index 1) must be connected to *ahbmsto*(5564 downto 5194)
- Master 3 (index 2) must be connected to *ahbmsto*(5193 downto 4823)
- Slave 1 (index 0) must be connected to *ahbslvo*(5503 downto 5160)
- Slave 2 (index 1) must be connected to *ahbslvo*(5159 downto 4816)
- Slave 3 (index 2) must be connected to *ahbslvo*(4815 downto 4472)
- All the masters must be connected to *ahbmsti*
- All the slaves must be connected to *ahbslvi*

The testbench must be in the form of a module including the submodules and the testbench program which drives the reset and the clock signal.

Some changes have to be made on the AHB master you created in task 2:

- *HADDR* should this time vary between 40000000 and 402FFF80, so that all of the 3 AHB RAMs are accessed.
- Also, the coverage points in your script should change. This time, you must use cross-coverage and check coverage of:
  - *AHBMO.HADDR* CROSS *AHBSLVI.HADDR*
  - *AHBMI.HRDATA* CROSS *AHBSLVI.HWDATA*
  - *AHBMO.HADDR* CROSS *AHBSLVI.HSEL*

Call the ICCR script **iccr_task3**. The testbench should be contained in a file called **tb_task3.v**. The AHB master should be defined in a file called **master_task3.v**.

Task 4 – Advanced random constraints (10 pts)

Deliverables: **master_task4.v**

iccr_task4

In this task you will work on the same multiple-master system that was introduced in the last task.

The objective of task 4 is to understand randomization further, using more advanced random constraints. You should add to the multiple-master system some advanced random constraints. The requirements are as below, the class used for randomization must inherit the properties from the class that has been used in the previous tasks, but a new element is added to randomly select a slave:

- The slave should be selected using “ranc”.
- The slave selection should be implemented as a distribution constraint.
- *HADDR* is constrained to select addresses within the memory range on which the selected slave is mapped.

Create a file called **master_task4.v** to define your master. You do not need to make any change to the testbench that has been defined in task 3.

Add to your ICCR scripts some commands to compare the report with the one generated for task 3. Save the script as **iccr_task4**.

Task 5 – Refined master (optional, 20 pts)

Deliverables: **master_task5.v**

In the former tasks you have designed an AHB master that can randomly write and read from and to the AHB RAM(s). Since the accessed addresses are chosen randomly, even the locations of the RAM that have never been written can be accessed by read transactions, and it is difficult to know if the operation was successful or not. In this task you should refine the code of the master defined for task 4 to make the master read only from the addresses to which it has already written. It is not necessary to take into account the locations to which the other masters have written, it is sufficient to take into account the locations written by the master itself. At the beginning of the simulation, therefore, the masters will only issue write transactions since every location in the RAM is

uninitialized. After the first RAM location has been initialized by a master, that same master will choose randomly whether to read from or to write to the RAM, and every read transaction will be performed only on those locations of the RAM to which the master has already written something. Call the new master **master_task5** and insert it into a file called **master_task5.v**.

Task 6 – Bins (optional, 10 pts)

Deliverables: **tb_task6.v**
master_task6.v
iccr_task6

In this task you will perform more advanced operations on bins. In the former tasks you did not specify the bins and the tool itself generated the default number of bins (64). When cross coverage is used, the number of bins becomes 64 × 64. You now need to redefine your bins:

- As specified in task 2, only some values of *HBURST* are legal. Define illegal bins for the *HBURST* signal.
- Define ignore bins for *HSIZE*, ignore cases in which the size of the transfer is bigger than one word (32 bits).
- Define three bins for *HADDR*: one for the range of addresses in the first RAM memory space, one for the range of addresses in the second RAM memory space and one for the range of addresses in the third RAM memory space.

Create a testbench in a file called **tb_task6.v** and a master in a file called **master_task6.v**. Define an ICCR script in a file called **iccr_task6**.

Task 7 – Random sequence (optional, 30 pts)

Deliverables: **tb_task7.v**

master_task7.v

Modify the testbench to implement random sequences. In this task you should randomize not only the output of the master, but also the sequence of its outputs. At any time, any of the masters has the possibility to write or read from or to any slave, and can be followed by any master write or read operation to any slave. For each transaction of the AHB bus, there are 18 possibilities, as there are three masters, each master can access any of the three slaves, and each transaction can be a read or a write transaction. However, since the arbiter has a round-robin policy and every master tries to access the bus as soon as it becomes idle, the masters will always be granted access to the bus in the sequence master 1, master 2, master 3, master 1, master 2, master 3... You should modify the testbench you wrote for task 5 so that, each time a new bus transaction must be issued, the master, the slave and the type of transaction (read or write) are chosen randomly. Different weights must be given to the three masters, the three slaves and the action, such as in the following example (you can experiment freely the weights to give to the different masters):

```
randsequence(main)
main: master slave action;
master: master1 :=5 | master2 :=3 | master3 :=2;
slave: slave1:=4 | slave2 :=4| slave3 :=2;
action: write :=7 | read:=3;
endsequence
```

Each master should react to changes in the randsequence and check whether it is its turn to use the

bus or not. If it is its turn, then a transaction can be issued. If not, the master can stay idle. Define your new master in a file called **master_task7.v** and the testbench in a file called **tb_task7.v**.