Hardware Verification – Main Coursework
Autumn Term 2022

Summary of Coursework Requirements
Version 1.0 1st December 2022

I was asked to consolidate the main coursework requirements into this one document. In general, it doesn't contain anything new, although I have added a couple of hints (marked in red).

I've also included the Teams messages that I posted about some issues with the original RTL.

Overall objective

The overall objective is to use the various verification techniques presented in the lectures to verify the GPIO and VGA blocks for which the RTL was supplied.

Some RTL modifications need to be made to satisfy the specification as detailed in 'AHB Peripherals Specification.pdf'

In summary you need to:

- Write a Verification plan (a list of what needs testing and how it will be tested)
- Write a brief ReadMe with any information that you think will be useful to me when assessing your work, e.g.
  - Location of files (and what they are for if not obvious)
  - Any important points about your verification strategy and any difficulties that you encountered
  - Brief summary of your verification results
- Modify the GPIO RTL to add parity generation/checking
- Instantiate a redundant VGA peripheral block plus add RTL for a comparator
- Provide a way of injecting faults to test the parity logic and the comparator
- Create unit-level constrained random testbenches written in SystemVerilog for the GPIO and VGA peripherals
  - Use SV constructs included in lectures in your testbench code
    - Interfaces
    - Clocking blocks
    - Class – e.g. for randomised variables
- Write functional coverage and demonstrate functional and code coverage has been achieved (when running the unit-level testbenches)
  - For the VGA peripheral, as you are only required to verify operation of the text display, you aren't expected to achieve full code coverage for that block
- Provide functional coverage and code coverage reports (as text or HTML)
- Write appropriate SystemVerilog assertions (SVA)
  - To cover designer intent (just some examples, I don't expect you to cover the full functionality of the VGA)
  - To describe behaviour of the interface(s)
- Attempt to prove some properties using Formal verification
- Develop checkers for the GPIO and VGA peripheral behaviour

        - For the GPIO, this should be straightforward (check that the GPIO outputs the correct value and reads back correctly what is on its input, and that parity generation and checking works correctly)
        - For the VGA, this is more complicated and I've added more details below
- Download the Cortex-M0 DesignStart model from Arm
- Demonstrate integration and verification at the Cortex-M0 SOC level
    - Write directed tests in Assembler for integration testing (you can use the example code supplied - src/cm0dsasm_AHB_Periph.s - as the basis of this)

Checking the VGA operation – further details

For the purposes of this coursework, you should at least verify that text characters are displayed in the correct part of the display, in other words in the "Visible area" as defined in the tables given in the 'AHB Peripherals Specification' and as show in the diagrams in the 'AHB Peripherals' Presentation slides. You can do this either in your unit-level simulation or your top-level simulation (when running code on the Cortex-M0) – or in both.

Text display operation

The VGA peripheral contains logic that drives the RGB[7:0], HSYNC and VSYNC signals at the correct times in order to display the characters at the correct place on the display. Each character is converted into a matrix of pixels and each row of that matrix is driven line by line to make up the complete character. If you look at the file font_rom.v, you will see how the matrix of pixels is derived for each ASCII character. For example, the ASCII code for the digit 1 is (in hex) 0x31 and the pixel matrix for this is as follows (there are blank lines above and below the character):

```
//code x31
11'h310: data = 8'b00000000; //
11'h311: data = 8'b00000000; //
11'h312: data = 8'b00011000; //
11'h313: data = 8'b00111000; //
11'h314: data = 8'b01111000; //   **
11'h315: data = 8'b00011000; //  ***
11'h316: data = 8'b00011000; // ****
11'h317: data = 8'b00011000; //   **
11'h318: data = 8'b00011000; //   **
11'h319: data = 8'b00011000; //   **
11'h31a: data = 8'b00011000; //   **
11'h31b: data = 8'b01111110; //   **
11'h31c: data = 8'b00000000; //   **
11'h31d: data = 8'b00000000; // ******
11'h31e: data = 8'b00000000; //
11'h31f: data = 8'b00000000; //
```

And so if you write the character '1' to the VGA peripheral via the AHB bus, this matrix of pixels should appear in the visible display area of the VGA monitor. See if you can develop a checker that verifies this. You can do this for just one or two characters, I don't expect you to verify the complete ASCII character set can be displayed! By using the 'Number of Pixels' and 'Number of Lines' information above and using the HSYNC and VSYNC signals as reference points, by counting clocks you should be able to work out when the characters will be displayed and then for a few lines record which pixels are being driven (RGB[7:0] is non-zero). You don't need to verify the colour information.

(For simplicity this VGA Peripheral is hard-wired to display all characters in the text display in green.)

Note that because the VGA uses RAMs for storage and these are not intialised in hardware, to prevent X's being propagated from the RAMs in your simulation, you need to write sufficient text and image data to fill the first lines of the screen's visible display area. This is what I posted in Teams related to this:

The other initialisation issue that you may run into relates to the two RAMs used in the instantiations of dual_port_ram_sync.v. I don't recommend trying to add hardware logic to initialise these RAMs. A better approach is to use software to write sufficient data to the text console and image buffer so that you are not trying to read uninitialised values from the RAMs - and thus suffer from X-propagation. You only need to verify that one line of text is displayed but to do this you need to run the simulation long enough to display 16 rows of pixels. One row of text is 30 characters. But you also need to ensure that the image buffer that is displayed for those 16 rows of pixels is also initialised - and you can do that by writing 4 rows worth of image data (each byte of image data is for a 4x4 matrix of pixels). From the example code in src/cmd0dasm_AHB_Periph.s, you should be able to work out how to do that - you need to write every 4x4 block from address 0x50000004 up to 0x50000640 (if my arithmetic is correct).

Possible bug in the VGA RTL

A couple of you have reported odd behaviour of the VGA controller, with incorrect pixels being displayed at the start of the line. I'm not sure what is causing this but I don't propose to fix the RTL - and you don't need to spend time trying to debug this. As long as you verify that the text string is displayed in the visible part of the screen, that will be ok. One role of a verification engineer is to identify bugs and report as much information about them to the RTL designer to help them go about fixing them. So you can just report any odd behaviour in your ReadMe that you submit with your coursework.

Updates to RTL to solve some reset issues in the original code (plus fixes to the source code files)

This was my Teams message about this:

I've now posted the updated RTL that should solve at least some of the reset issues. I've put this in a new directory called AHB_Peripherals_Files_Updated in the Main Coursework directory. That way you can just pick out the files that have changed and not disturb any of the RTL that you might have already modified. The files that I changed include:

AHBLITE_SYS.v : assigned reset_n to be inverted version of RESET; added reset to the clk_div logic
AHBVGASYS.v : added HRESETn to port list of VGAInterface block
vgasync.v : added resetn to port list and connected inverted version of resetn to the three counters
src/cmd0dasm_LED.s : corrected address of GPIO
src/cmd0dasm_AHB_Periph.s : removed superfluous code for timer module

Marking schedule

For the hardware verification coursework, the marks will be distributed across the deliverables listed here, with 20% of the marks reserved for assessing how well your verification covers the functionality.

- ReadMe and Verification Plan - 10
- Modified RTL code for the GPIO and the dual lock-step configuration of the VGA Peripheral - 10
- SystemVerilog files of your testbench(es) - 20
- SystemVerilog assertions and evidence of trying to prove some of these with formal verification - 15
- Evidence of running the unit-level testbench simulation - 5
- Assembler (or C) code for your top-level integration testing - 5
- If possible, a log file or screenshot showing the result of running your top-level test - 5
- Functional coverage points plus code coverage and functional coverage reports - 10
- Overall completeness of verification, including some form of checking - 20