**ECE 551 PROJECT REPORT**

**UNIVERSITY OF WISCONSIN-MADISON**

**DIGITAL STORAGE OSCILLOSCOPE**

**NAVIN SENGUTTAVAN**

( )

**PRASHANTH BALASUBRAMANIAN**

**(9069398601)**

**DEEPTI BHASKAR**

**(907119587)**

**INTRODUCTION**

The aim of the project was to design a 3-channel Digital Storage Oscilloscope. Verilog HDL was used for the design. The DSO supports two modes : triggered and autoroll. It also supports a number of configurable gains and trigger levels. The DSO is designed to interact with a host-PC through the UART and USB interfaces. Top of the shelf SRAMs were used as memory components to capture the signals from the ADCs which were driven by the 3 analog channels. The digital core of the DSO interacts with the analog front-end modules through a SPI interface. The design was then taken to synthesis with the TSMC 45nm library. Post synthesis simulations were done with certain delay constraints.

**HARD LESSONS LEARNT**

Most of the issues we faced were during debugging and validation. During our design phase we did not face too many issues. The two major modules that we had problems were the “command module” and the “capture module”. The command module was complex and had a lot of minute details. In addition a number of control signals were present that essentially were vital for other modules. Another issue during the design of the command module was designation of wait states. Our major concern was during the read and write EEPROM (SPI Transaction) and DUMP. Due to multiple number of case statements we had to carefully align the code in order to make sure that there were no errors with respect to the sequential flow (successive statements). Setting the *decimator* register to 0 posed several issues while simulation. We had to make changes that would work for this special case as well. System Verilog made our waveforms easier to observe eg. (“next/present” state) and in the state machine (due to typedef enum). The capture module was at the beginning working fine until we checked for “*autoroll*”. We had not considered this case in the design phase and had some combinational logic introduced in the last minute. Also due to timing issues there was an infinite loop in this module that we rectified later on. So we learnt how important it is to consider potential timing issues and consider those issues right at the beginning to avoid such problems. Setting and Resetting of the *capture\_done* bit in the *trig\_cfg* register from different modules had to be thought over to avoid data race in the regiser.

During validation we wasted a lot of time looking into our code and adding waveforms when the actual problem was due to “extra files” in our project window. We had two files, one with a .v and the other with a .sv extension. This led to major confusion during simulation. Another similar issue that we faced was when the ram\_shell.v was present in our project window. During simulation the “rdata” value in all three channels was filled with don’t cares. When we removed that file this issue was solved. Another common mistake in our modules was while instantiation of other modules. As we did not map ports by name there were errors during our simulation. Mapping ports by name made our work simpler and efficient. We could also access the registers and other signals with ease.

Self-checking test benches was another useful concept that we incorporated in our project. Due to self-checking test benches we were able to debug much faster and we knew where to look into exactly when there were errors. Some of the trivial issues we faced were warnings with respect to incorrect port sizes and port mismatches while connecting modules.

During synthesis we had to eliminate some flip flops and use assign statements to improve our area. Some of the positives were exhaustive test benches that helped improve our code coverage, commenting and indentation, port mapping by name, etc.

Another significant lesson we learnt was the importance of reading the project spec in excruciating detail. We faced a couple of issues while encoding the command to be processed by the command module. This minor misunderstanding led to a rather long verification and validation phase.

On the whole since we followed Cumming’s guidelines our project turned out be successful both during pre and post synthesis.

**PARTITIONING OF THE MODULES**

Digital Core

**Capture Module** -

Captures the samples from the A2Ds into the RAM based on inputs from Trigger module and generates trace\_end

**Command Processing unit** – Processes the commands received from the host over UART. Controls the SPI master on the DSO.

**Trigger Module** – Generates triggered signal based on the configuration registers. Used in the capture module

Initially we had planned to split our modules as

1. RAM interface
2. Command module
3. Capture and Trigger logic

While designing we found this to have drawbacks, so we partitioned it as

1. RAM interface and Capture logic
2. Command module
3. Trigger logic

The capture module and the RAM interface are intertwined so it made more sense to club them. The “enable” signal and “write enable” signal comes from the capture logic which is essential for the RAM to be able to Read/Write. This made the code easier to write as most of the signals were part of the same module.

Combining the Trigger and Capture logic turned out to be complex due to the largely separate functionality of each module so we kept them separately.

The command module was the core crux of the project as it had all the control signals and was the bridge between the various modules and the host. Keeping it as a separate module turned out to be the most logical approach. It also improved the code readability. As the command module interacted with the host it was best not to club it with other modules.