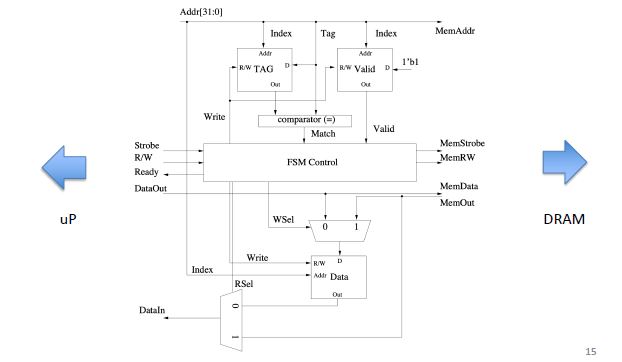
**Computer Architecture Lab 3**

Written by: Matthew Whitlock (11744571) and Nate Morton (11784201)

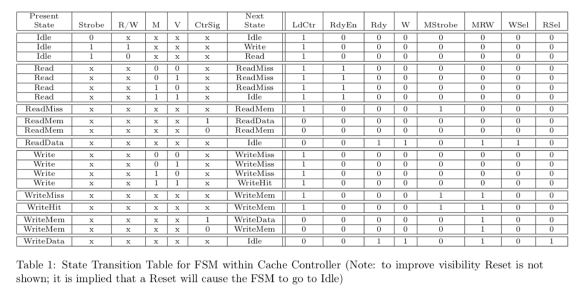
**Introduction**

In this lab, our goal was to write an instruction set architecture, using Verilog Hardware Descriptive Language (HDL) modeling language and implement it on the Elvis III board, but this time we were to implement a cache for the data memory. The purpose of this lab was for us to understand how the cache is design and is made to work together with the other components (mainly memory) to make up a pipelined ARM Instruction Set Architecture. For this lab the following instructions needed to be encoded into our program: ADC, ~~ADD~~, ~~AND~~, ASR, ~~B~~, BIC, CMN, CMP, EOR, ~~LDR~~, LSL, LSR, MOV, MVN, ~~ORR~~, ROR, SBC, ~~STR~~, ~~SUB~~, TEQ, and TST (the ones crossed out were already encoded for us before hand, also we were able to use a good portion of our code for the instructions from lab 3). In order to make sure our code was correct we simulated it on ModelSim making sure each of our instructions worked with the given input files we gave it. Once we had written all our instruction, and confirmed they worked in simulation, we then started working and making debugging changes in the code to where we could get it to work on the Elvis boards. We then counted the cycles for each test on the board and compared them with our simulated tests that had perfect memory. Since this lab was very similar to the previous lab, with the main difference being the pipelined architecture vs the single-cycle, it ended up not being too difficult and was less of a hassle to get working.

**Baseline Design**



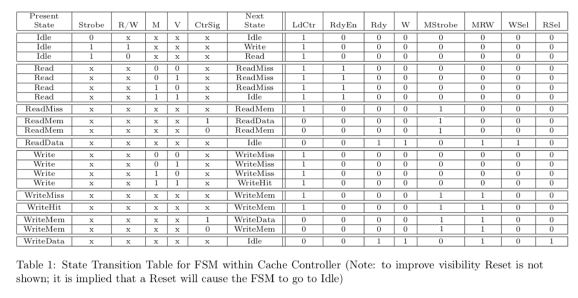
The design in the figure above is that of a simple Cache, However, it still serves it’s purpose in having us understand how caches work. The cache above is a write through allocation cache so if we make a write to memory then not only do we write to memory but we also write to cache.



When it comes to the cache the cache controller has a key element which is the Finite State Machine (FSM). The purpose of the FSM is to make sure the cache is actually working and if a possible error occurs then it is to alert the main memory so that it can either synchronize or replace a part of the cache. The FSM’s capabilities can be summed up as Read and Write. When When the FSM tries to either Read or Write, this starts with a Strobe. When testing memory we use a counter or a wait state, which begin counting once it enters memory. The counter counts down until it hits zero and once it reaches zero it enters it’s final state.

**Design**

The design we have chosen is only slightly modified from the initial design, with most of the modifications being small changes to the way the cache interacts with the hardware and how it works with the instructions. More notable changes were the change to the finite state machine so that memstrobe is on during ReadMem and WriteMem. Chnaged the ReadyRDY output from turning on when 1 cycle is done to an active low indicator of readiness for p ready.



**Testing strategy**

As a first step to testing, we ran the test-code from our ARMv8 simulator in ModelSim. This gave us a baseline of if simple instructions functioned as expected in the general-use case. However, it is nigh impossible to test every single possible input, so we also used the benchmark applications as tests. Even though the test application from our previous simulator test file succeeded, we ran into issues during the benchmark execution.

After ensuring that the CPU worked in ModelSim, we then used Vivado to export the CPU to an FPGA to test for correctness in the implementation, but we ran into some issues and were unable to figure out the problem and make corrections in time.

**Evaluation**

Regretfully though we put a considerable amount of time in this lab, we ran into a problem where we were unable to simulate our code due to some buggy errors that we didn’t know how to fix. So we were unable to complete the evaluation.