LAB 3 WRITE UP

Professor Heiner Litz - CSE125 Group 3 - Jake Liao - Mark Zakharov

Introduction

An issue presented in memory systems is processing and storing potentially asynchronous and bursty data without losing anything through poorly timed overwrites. The solution to this is a FIFO buffer, a series of registers which control data traffic flow and sequentially output data travelling through them. This is advantageous to have in a system as it captures and protects all data sent into it and has a controlled method of sending each separate set of data in order. FIFO stands for First In- First Out and that is the basis of the system implemented in this lab.

Method

In this implementation of the FIFO design, a single stage consists of combinatory logic, input multiplexer, and a register of parameterizable width. These stages are linked together using the verilog generate block on the top module 'fifo'. The top level module also controls the full and empty signal, stores an integer count that keeps track of the next stage to input value, and controls individual 'shift_in' and 'shift_out' signals for each stage.

The initial implementation of this lacked a counter and fed each stage's multiplexor a shift_in, shift_out, prev_full, current_full, and next_full signal. The assumption was made that the three full signals would inform each multiplexor on whether it should be outputting the currently held value, the new data, or the previous stages data. This functionality was removed as the full signals did not properly communicate with the multiplexor and its inputs and resulted in each stage shifting in the new data.

The top level module contains an always @(posedge clk) block which controls the full, empty, and shift_in and shift_out signals for individual stages. The count is used to identify the specific stage that would receive the shift_in or shift_out signal. Only one stage could receive shift_in at a given clock cycle, while all occupied stages would receive the shift_out signal. These buses containing shift_in and shift_out are cleared when unused to reduce the chances of an old signal causing unintended behavior. By manually controlling each stage's shift_in and shift_out signals through a top level if/else statement, each stage no longer relies on those around it and full/previous full signals but only on the count.

Resources and Timing

After synthesizing and performing implementation. Using 100ns as clock period, Fmax is calculated and plotted against varying depth. Figure 1. shows consistent decrease in Fmax as the number of stages increased(Depth) while the data size(Width) stays at 32 bits.

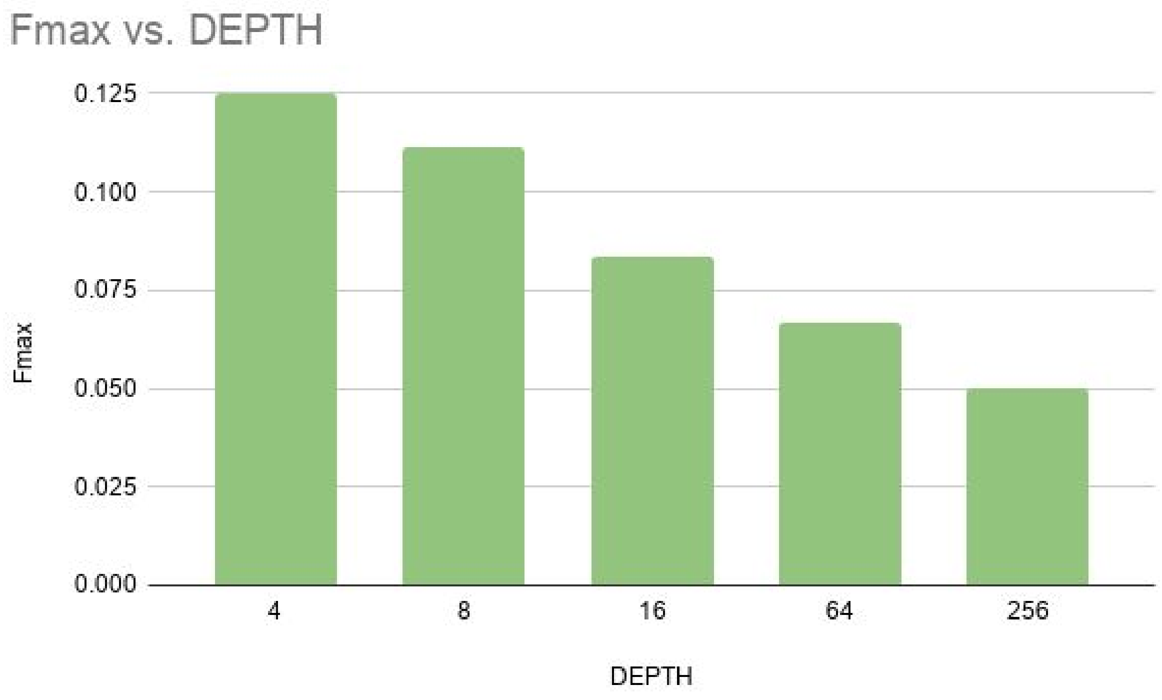

Figure 1. Fmax in relation to varying depth of FIFO

Figure 2. shows the amount of LUTs and FFs used with varying depth while the width stays at 32 bits. In this lab a significant amount of registers were used both in the individual stages and elements where timing was required, as seen in Figure 2. the amount of LUTs and FFs increases linearly with depth. An attempt was made to implement 64 bit width with varying depth to observe the behavior, however, it was made clear that this design required a significant amount of resources such that a 64 bit width by 8 depth could not be synthesized.

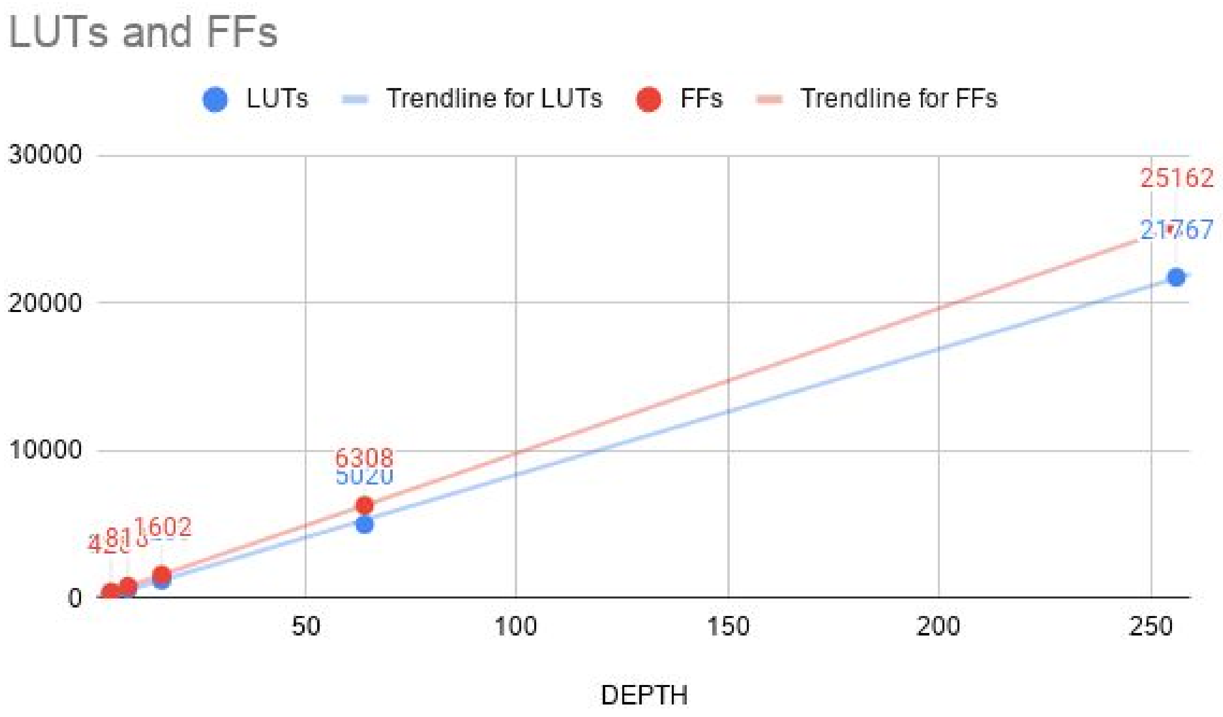

Figure 2. Resources used in relation to varying depth of FIFO

Testing

A directed testbench was created initially to confirm the basic functionality of the top level module was correct, such as working full/empty signals and the data_out correctly reflected the head stage's held value. Simple testbenches are useful as they are predictable and informative of general conditions being met.

The randomized testbench was created to more accurately reflect an erratic and unpredictable input for the FIFO. Randomness is essential in detecting corner case issues as well as prolonged activity and large sets of input and output data. By having a parameterizable loop variable to generate a set amount of random input values, large test cases are simple to create and analyze. How this was handled was a TEST_NUMBER variable was created to represent the amount of new values the FIFO would receive, and the $random command was used to fill up a vector of random values. This data was first fed into a .txt file and then $readmemb transferred the data in the text file into the input vector. A while loop with a count going to TEST_NUMBER was created, incrementing each time a random shift_in signal was created. Conversely, there was a random chance to receive a shift_out signal as well, which set a flag to copy out_data into another text file the following clock cycle. Preventative conditions were created for unwanted cases such as shifting into a full FIFO, or shifting out of an empty FIFO. In order to ensure there was a full signal, the first 7 counts forced a shift_in and prevented shift_out, but besides that the whole process is completely randomized. After all values are shifted in, a while loop runs until empty goes high to ensure the output of all shifted in values. Since a FIFO works on a first-in, first-out basis the input and output text files are expected to have the same content. Creating an automatic comparison of these is an effective way to mass test modules, as a few manually created test cases being visually examined by a human may leave some issues unnoticed. This is why the data in the text files was copied into arrays in Matlab and the difference between them was found and plotted.

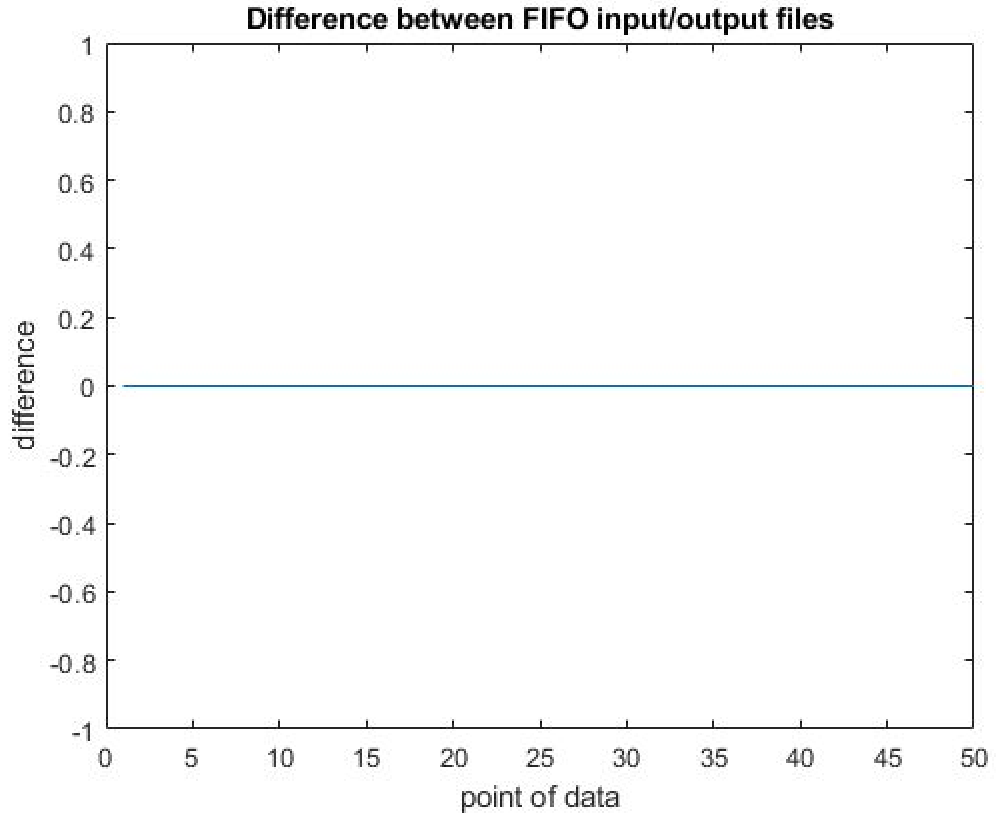

Figure 3: Plot of differences of I/O files used in randomized test

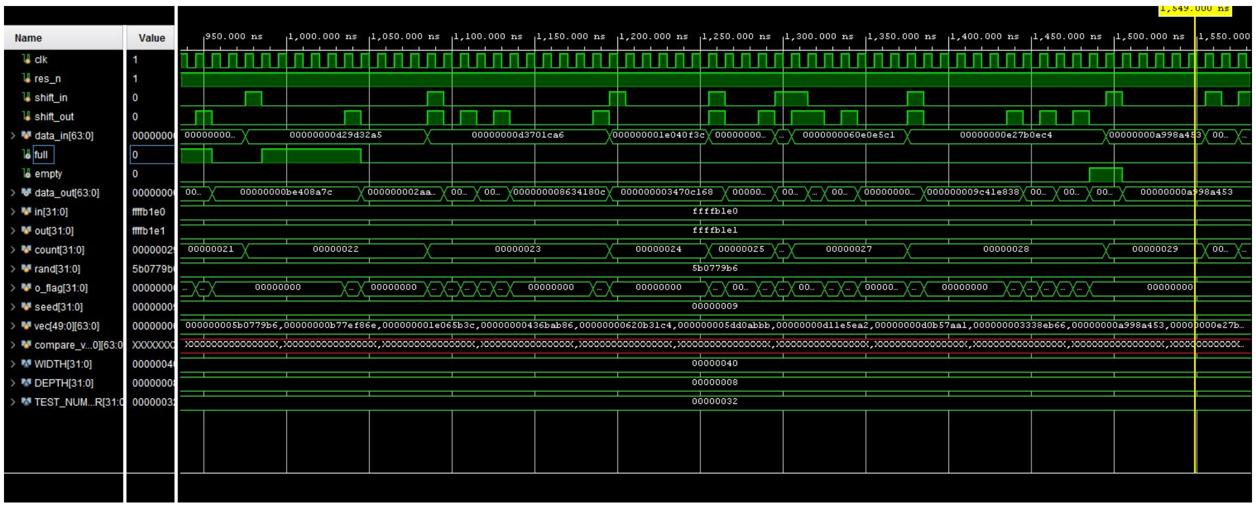

Figure 4: Random test waveform, shows active full and empty signals

Discussion

The amount of resources was used for the always @(posedge clk) block in the top level module. Since it was decided that full and prev_full would not be used to control the internal shifting operations, the amount of for loops, if statements, and buses in the top level consumed a significant amount of resources. Initially the design would not synthesize due to having for loops with complicated initial value and end condition. This was fixed by simplifying the complexity of the nested if statements by reducing the redundant for loops and combining if statements into case statements. After reducing the conditional statements, the final design still uses so many IO pins that implementation for the Pynq z1 failed with exceeding I/O pins. Another board with higher I/O count was used to run implementation benchmarks.