Lab 3- LIFO buffer

Cody Joy

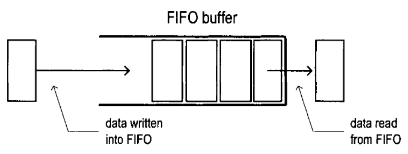
1 Introduction

The purpose of this lab is to create a last in-first out (LIFO) buffer modified from provided code for a first in-first out (FIFO) buffer.

In it's most basic sense, a buffer allows a user to shore data, and then retrieve the data later without saving the data to a specific name. Rather, the data is saved to a stack of data. The LIFO and FIFO are two of the most common types of buffers. buffers are a critical component in may applications. Normally, the buffer will be a part of a larger program.

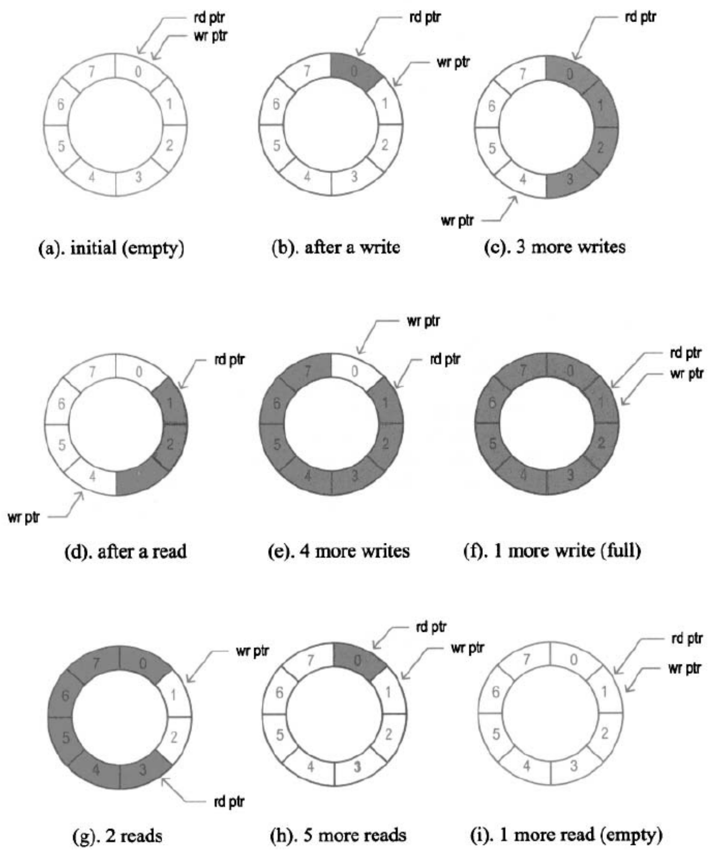
The first type of buffer is the FIFO buffer. Figure 1 and 2 show the basic schematic for a FIFO buffer. In the conceptual design shown in figure one, a data pack is written into the FIFO. FOr each data pack that is written into the FIFO, the stack gets larger. When it is time to read, the first data pack is read from the FIFO. Instead of an ever increasing line, the FIFO can also be thought of as having a circular implementation. This circular implementation is the original code provided for this project. Initially, the FIFO buffer is empty. From hear, data can only be written to the buffer. In the example shown in figure 2, four writes occur. At this point, the data can either be read or written. One of the difficulties with the circular FIFO buffer is with the rd and wr pointers. The position of these points have to be tracked as well as the position relative to each other. Initially, the pointers are at the same point, meaning that the buffer is full. Later, in step f, the pointers are again at the same position. This time the buffer is full. This tracking of the two pointers make the program more complicated and more time consuming. The benefit is that the pointer shows exactly where the next read or write occurs.

Figure 1: FiFO buffer Conceptual Diagram



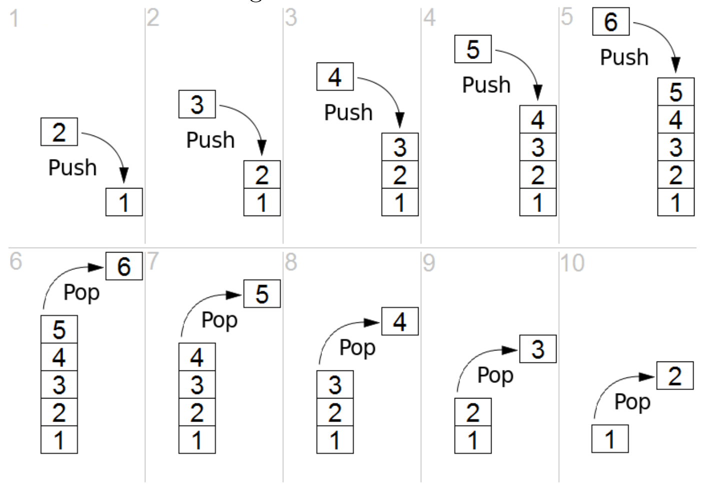
The second type of buffer is the LIFO buffer. Unlike the FIFO, where the first data written in is the first data read out, the LIFO buffer reads the last

Figure 2: FIFO buffer Circular



data written in. This is shown in figure 3. For an LIFO buffer, the read and write functions are commonly refered to as POP and PUSH, respectively. In the example shown, five seperate stages of data are written in. When the data is read, the last piece of data written will be read out. On of the advantages of the LIFO buffer is that it allows for only one pointer. In stead of stracking the postition of two pointers, only the position of one pointer needs to be known. If the pointer is at the bottom positioin, the stack is empty. If the pointer is at the top position, the stack is full. This simpler design will allos for easier programming and a quicker running program. The LIFO is also much faster than the traditional FIFO. In a FIFO buffer, the data can be put directly into the position and read from that position. In a FIFO buffer, the data has to be put in one side, and pushed through to the other side. For example, for a buffer that can have 12 stacks of data written in, with a read time of 1 ns and a write time of 1 ns, the whole read/write process will take 2 ns for a LIFO buffer. FOr a FIFO buffer, it will be written into position 0, and then have to be pushed through to position 11, or a total time of 13 ns. As can be seen, the LIFO has the potential to be much quicker.

Figure 3: LIFO buffer



2 Interface

To implement the LIFO buffer, a Verilog program was created that was to be run on Nexys 4 DDR boards. The read function was triggered using one button. Another button was used to trigger the write function. The reset button of the board was used as a reset to the circuit. Three switches were used for the data to be written. A total of 8 different LEDs were used in implementation. LED 0 to 2 show the data to be read. LED 6 shows if the buffer is empty, and LED 7 shows if the buffer is full. The remaining LEDs are set to 0. When the button for writing is pressed, the value of the switches is saved into the first available buffer position. When the read command button is pressed, the laset read data is displayed on the first three LEDs. The size of the read and write data could be increased, but this size was not changed from the provided code.

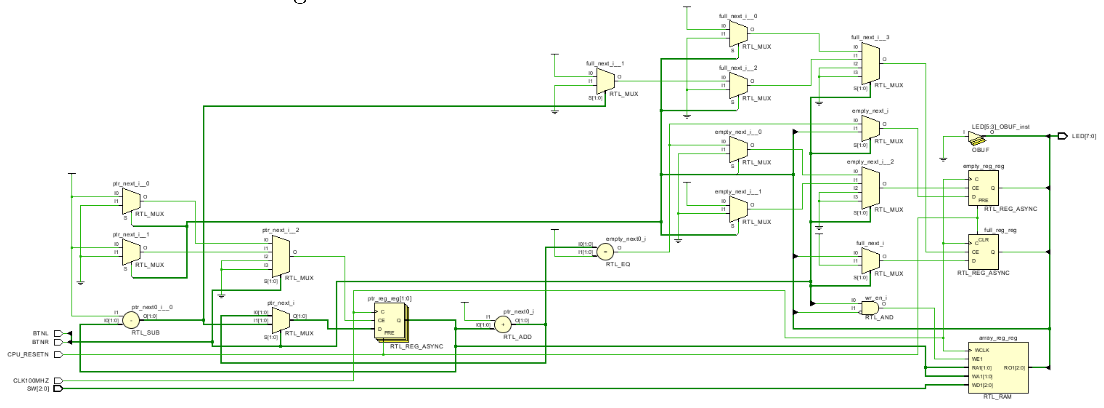
3 Design

In order to create a LIFO buffer, a few key pieces of code neeed to be created. The first is to write to the buffer when the write is enabled. The code must also be able to read from the buffer. The next key piece of code is the reset. The reset, as the name implies, will reset the system to the initial conditions. The next piece of code is the pointer, this determines what level the read or write will occur. The final critical piece of code is what to do depending on if no buttons are pushed, if only the read is pressed, if only the write is pressed, or finally if both the read and write are pressed. The next section will go into more detail of how the code was implemented and will also show a digital logic schematic of the system.

4 Implementation

The code for the LIFO is shown in appendix B. Appendix A shows the testbench for the LIFO buffer. The inputs and outputs of the code were discussed briefly above, and will be discussed in more detail in the FPGA Realization and Final Verification section. In order to create the program, a few other wires and registers were created. the array register is what holds the data. This register acts as the buffer. The various ptr registers act as the pointer. Either the current, previous, or next pointer. Finally, registers are used to determine if the buffer is full or empty. A wire is created to determine if the write command will be enabled. In the first section of the program, the write operation is executed. If the write command is executed and the buffer is not full, the data is written into the array. In order to complete the read function, the current value of the buffer is assigned to the read data. The final important piece of code is the determination of what to do when either the read or write commands are pressed. If nothing is engaged, then nothing changes in the program. If the read command is actiavated, the value of the full is set to zero, the pointer goes down by one, and the program checks to see if the buffer is empty. If the buffer is empty, it assigns a value of 1 to empty. If the write command is activated, the value of the empty is set to zero, the pointer goes up by one, and the program checks to see if the buffer is full. If the buffer is full, it assigns a value of 1 to full. This will then ensre that no more write commands will be allowed. The final case is if both the read and write are enabled. If this occurs, the pointer stays in the same position, and the write value goes directly to the read value. Figure 4 shows a schematic of how the logic chips are connected.

Figure 4: LIFO Schematic



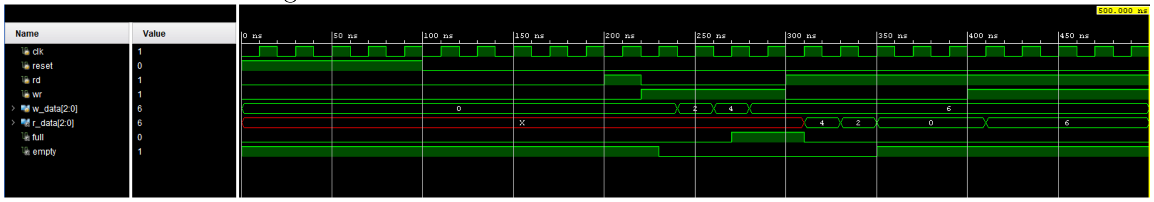
5 Test Bench Design

A test bench was created to try to account for any possible user interaction. Intially, the whole system was reset. This reset was to ensure that the conditions were set to the initial desired conditions. The program would still work the same without the reset stage, but it is good practice to ensure everything is going correctly. The next step is to try to read data from an empty buffer. The result should show that no data can be read. The next step is to write to the buffer. After the buffer is full, one more write command is given. This final write command should not work, meaning that the final value to be written will not be written to the buffer. To ensure this, the data is read until the buffer is empty. The value read should be the last data in the buffer, not the data after the buffer is filled. The final step is to active a read and a write simultaneously. The value of the read and the write should be exactly the same.

6 Simulation

Figure 5 shows the simulation results from the LIFO buffer. The results are exaclty as was expected. Initially, the system was reset. No data was being read, resulting in values of X, and no data was written. The buffer is also shown to be empty. After that, the read is activated. Since there is no data in the buffer, the value still reads X and the buffer is still empty. Next, the buffer is filled with 0, then 2, then 4. After the buffer is filled, the value of 6 is written. Notice at this point the buffer is now full. When the read is activated, it reads the values in inverse order of when they were written into the buffer. The read shows 4, then 2, then 0. Notice, the buffer is now empty, and the value of 6 was not read. Finally, the read and write are activated at the same time. Notice the read and write are the same value, 6.

Figure 5: LIFO Simulation Results



7 FPGA Realization and Final Verification

To implement the code, a few changes were made to the code used for simulation. This code will not be attached to this report, but will be included with the report. The first change was to change the 5 different inputs. The clk was changed to the internal clock of the nexys 4 DDR. THe next change was the reset was changed to the reset button on the board. The read and write commands were changed to two different buttons, BTNL, and BTNR. When BTNL is pressed, the read command is activated. When BTNR is pressed, the write command is activated. The final change for the input is to change the input write data to switches 0,1,2. This allows for numbers between 0 and 7 to be written into the buffer. If larger numbers are required, then the number of write switches can be increased. The outputs were also changed. LED 7 is used to show if the buffer is full or not. If the buffer is full, the LED will light, otherwise it will remain unlit. Similarily, LED 6 shows if the buffer is empty. LED 0,1,and 2 are used to show the data read from the buffer. The number of LEDs of the read will always be the same number as the number of switches for the write data. If the number of bits is changed for the write, the number of LEDs for the read will also change. Finally, the remaining LEDs between the read data and the empty indicator are set off. In order to ensure that the correct values will be shown , each of the switches must be fully up or down, and the button must be pressed firmly for a few seconds. A debounce circuit was used in order to ensure that the correct values were obtained from the switches and button.

8 Conclusions

In conclusion, code for a first in-first out buffer was modified to create code for a last in-first out buffer. The LIFO buffer is fast to program and faster to run as compared with the FIFO buffer. The LIFO buffer reads the last value written into the buffer. The FIFO buffer reads the first value written into the buffer. The FIFO buffer was created using Verilog and tested on a Nexys 4 DDR board using 2 buttons, 3 switches, and 8 LEDs. The two buttons determine if read or write will be activated. The switches work as the data to be written. Three LEDs work as the data read from the buffer. Two fo the final LEDs are used to determine if the buffer is full or empty.

One of the best take aways from this lab was being able to learn about the LIFO buffer and construct the LIFO buffer. Since the LIFO and FIFO buffers are commonly used in more complex circuits, it is very important to have a great understanding of exactly how these buffers work. Another value lesson learned was the ability to look at a piece of code written by someone else, understand what the was doing and why they were doing it, and finally to be able to edit said program to be able to create the program I need to create. This is an important skill that will help save time, as compared with writing code from scratch. One of the things I would change with this program is to decrease the amount of coding used. The point uses four different variables. This number should be decrease. Finally, I would make sure that when the buffer is empty, the data read is all X's.

A LIFO Testing

```
// Listing 4.21
module Lifo_test;

    //input wire clk, CPU_RESETN,
    //input wire BTNR,BTNL,
    //input wire [3:0] SW,
    //output wire [7:0] LED
    reg clk;
    reg reset;
    reg rd;
    reg wr;
    reg [2:0] w_data;
    wire [2:0] r_data;
    wire full;
    wire empty;


   // signal declaration
   //wire [1:0] db_btn;

   // debounce circuit for btn[0]
   //debounce btn_db_unit0
   //    (.clk(clk), .reset(CPU_RESETN), .sw(BTNL),
   //     .db_level(), .db_tick(db_btn[0]));
   // debounce circuit for btn[1]
   //debounce btn_db_unit1
   //    (.clk(clk), .reset(CPU_RESETN), .sw(BTNR),
   //     .db_level(), .db_tick(db_btn[1]));
   // instantiate a 2^2-by-3 fifo
   Lifo #(.B(3), .W(2)) Lifo_unit
      (.clk(clk), .reset(reset),
      .rd(rd), .wr(wr), .w_data(w_data),
      .r_data(r_data), .full(full), .empty(empty));
   // disable unused leds


    initial begin

    // Initialize Inputs

    w_data  = 4'h0;

    rd  = 1'b0;
    wr  = 1'b0;

    reset  = 1'b1;

    clk  = 1'b0;

    #100;
    reset     = 1'b0;
    #100;
    rd = 1'b1;
    #20;
    rd =1'b0;
    wr      = 1'b1;

    w_data  = 4'h0;

    #20;

    w_data = 4'h2;

    #20;

    w_data = 4'h4;

    #20;

    w_data = 4'h6;

    #20;

    rd  = 1'b1;
    wr = 1'b0;
    #100;
    wr = 1'b1;
    #100;
    $finish;

  end

     always #10 clk = ~clk;

endmodule
```

B LIFO Code

```
// Listing 4.20
module Lifo
   #(
    parameter B=3, // number of bits in a word
              W=2  // number of address bits
   )
   (
    input wire clk, reset,
    input wire rd, wr,
    input wire [B-1:0] w_data,
    output wire empty, full,
    output  wire [B-1:0] r_data
   );

   //signal declaration
   reg [B-1:0] array_reg [2**W-1:0];  // register array
   reg [W-1:0] ptr_reg, ptr_next;
   reg [W-1:0] ptr_prev, ptr_succ;
   reg full_reg, empty_reg, full_next, empty_next;
   wire wr_en;


   // body
   // register file write operation
   always @(posedge clk)
      if (wr_en)
         array_reg[ptr_reg] <= w_data;
   // register file read operation
   assign r_data = array_reg[ptr_reg];
   // write enabled only when FIFO is not full
   assign wr_en = wr & ~full_reg;

   // fifo control logic
   // register for read and write pointers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            ptr_reg <= 3;
            full_reg <= 1'b0;
            empty_reg <= 1'b1;
         end
      else
         begin
```

```
            ptr_reg <= ptr_next;
            full_reg <= full_next;
            empty_reg <= empty_next;
         end

   // next-state logic for read and write pointers
   always @*
   begin
      // successive pointer values
      ptr_succ = ptr_reg - 1;
      ptr_prev = ptr_reg + 1;
      // default: keep old values
      ptr_next = ptr_reg;
      full_next = full_reg;
      empty_next = empty_reg;
      case ({wr, rd})
         // 2'b00:  no op
         2'b01: // read
            if (~empty_reg) // not empty
               begin
                  ptr_next = ptr_prev;
                  full_next = 1'b0;
                  if (ptr_next==3)
                     empty_next = 1'b1;
               end
         2'b10: // write
            if (~full_reg) // not full
               begin
                  ptr_next = ptr_succ;
                  empty_next = 1'b0;
                  if (ptr_next==0)
                     full_next = 1'b1;
               end
         2'b11: // write and read
            begin
               ptr_next = ptr_reg;
            end
      endcase
   end

   // output
   assign full = full_reg;
   assign empty = empty_reg;

endmodule
```