Lab 5- Babbage Difference Engine

Cody Joy

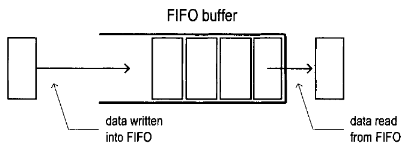
1 Introduction

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.

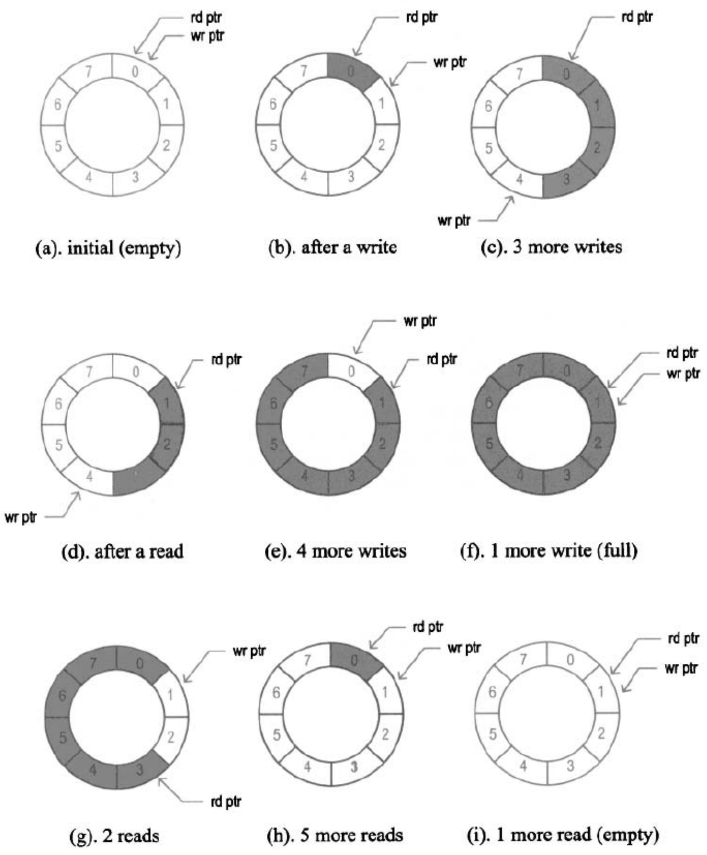
The first type of buffer is the FIFO buffer. Figure 1 and 2 show the basic schematic for a FIFO buffer. In the conceptual design shown in figure one, a data pack is written into the FIFO. FOr each data pack that is written into the FIFO, the stack gets larger. When it is time to read, the first data 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



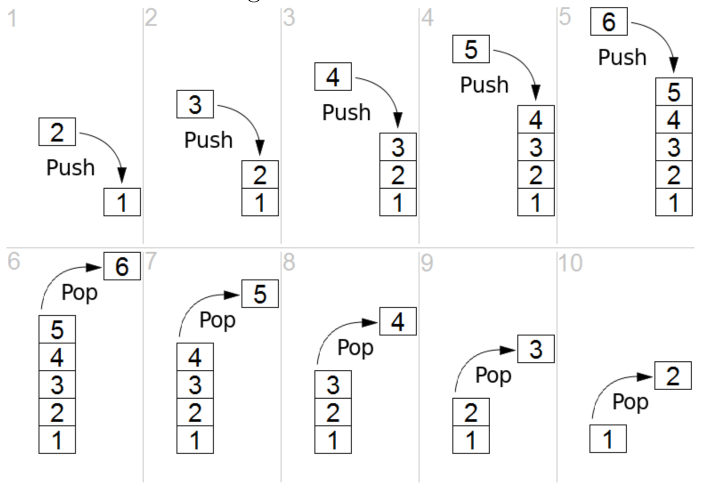
The second type of buffer is the LIFO buffer. Unlike the FIFO, where the first data written in is the first data read out, the LIFO buffer reads the last

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. To complete the first part of the assignment, the following equations had to be coded into a second order babbage program and a third order babbage program: $f(n) = 2n^2+3n+5$ and $h(n) = n^3+2n^2+2n+1$. To run the program, the 16 switches were used to find positions 0 to 15. If switch 0 is active, then position 0 will be displayed. If switch 1 is active, then position 1 will be displayed, etc. The results are displayed on the 16 LEDs. The output value is in binary. This output value will need to be converted to decimal to compare with the calculated value or converted to hexidecimal to compare with the simulated value.

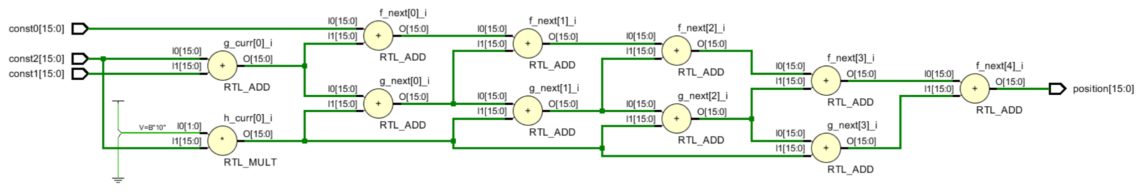
3 Design

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.

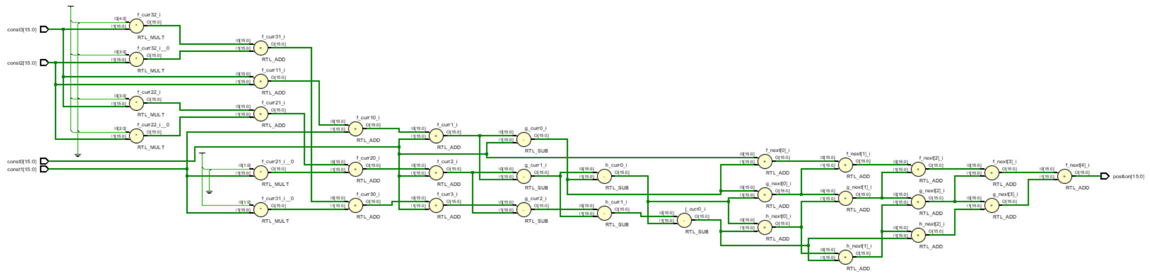
Figure 4: Second Order Babbage Schematic



5 Test Bench Design

A test bench was created to try to account for any possible user interaction. Four seperate equations were used for the test bench. Two equations for the second order polynomial and two equations for the third order polynomial. Two

Figure 5: Third Order Babbage Schematic

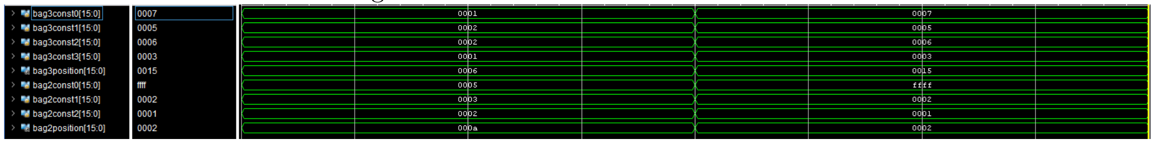


of the equations are required to test. These questions are: $f(n) = 2n^2 + 3n + 5$ and $h(n) = n^3 + 2n^2 + 2n + 1$. Two other equations were tested as well to show that the programmed Babbage difference engine works for any second or third order polynomial and the program wasn't created to specifically solve for one equation. The other tested equations are: $f(n) = n^2 + 2n - 1$ and $h(n) = 3n^3 + 6n^2 + 5n + 7$.

6 Simulation

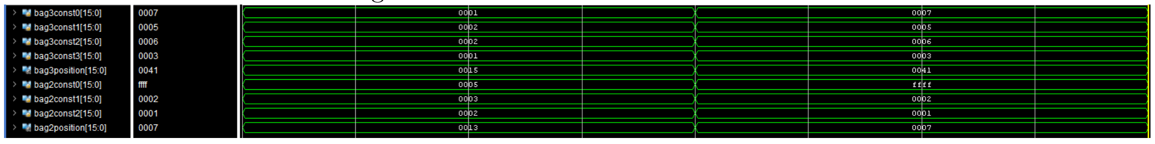
Four seperate simulations were completed. Since the position is dependent upon a given parameter, a new simulation must be completed for each position. In order to sufficiently verify the circuits using the test bench but not conducting too many simulations, four positions were calculated, 1,2,3, and 4. The following four figures show the simulations for position 1, position 2, position 3, and position 4.

Figure 6: Position 1



```
bag3const0[15:0]    0007    0001    0007
bag3const1[15:0]    0005    0002    0005
bag3const2[15:0]    0006    0002    0006
bag3const3[15:0]    0003    0001    0003
bag3position[15:0]  0015    0006    0015
bag2const0[15:0]    ffff    0005    ffff
bag2const1[15:0]    0002    0003    0002
bag2const2[15:0]    0001    0002    0001
bag2position[15:0]  0002    000a    0002
```

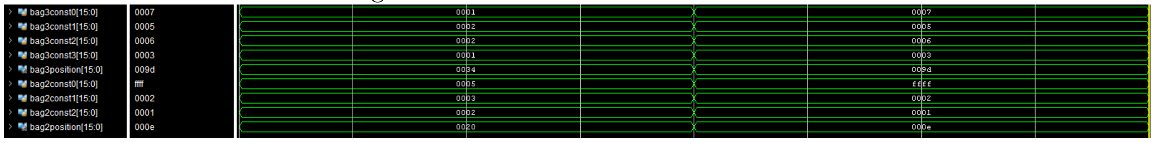
Figure 7: Position 2



```
bag3const0[15:0]    0007    0001    0007
bag3const1[15:0]    0005    0002    0005
bag3const2[15:0]    0006    0002    0006
bag3const3[15:0]    0003    0001    0003
bag3position[15:0]  0041    0015    0041
bag2const0[15:0]    ffff    0005    ffff
bag2const1[15:0]    0002    0003    0002
bag2const2[15:0]    0001    0002    0001
bag2position[15:0]  0007    0013    0007
```

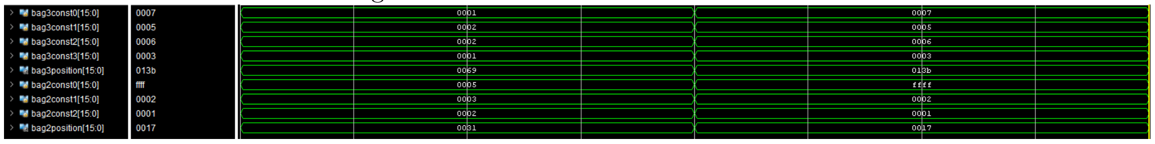
To begin, the first second order polynomial will be analyzed. In order to calculate the four positions, the values 1,2,3, and 4 are substituted in for n

Figure 8: Position 3



```
> bag3const0[15:0]    0007    0001    0007
> bag3const1[15:0]    0005    0002    0005
> bag3const2[15:0]    0006    0002    0006
> bag3const3[15:0]    0003    0001    0003
> bag3position[15:0]  009d    0034    009d
> bag2const0[15:0]    ffff    0005    ffff
> bag2const1[15:0]    0002    0003    0002
> bag2const2[15:0]    0001    0002    0001
> bag2position[15:0]  000e    0020    000e
```

Figure 9: Position 4



```
> bag3const0[15:0]    0007    0001    0007
> bag3const1[15:0]    0005    0002    0005
> bag3const2[15:0]    0006    0002    0006
> bag3const3[15:0]    0003    0001    0003
> bag3position[15:0]  013b    0069    013b
> bag2const0[15:0]    ffff    0005    ffff
> bag2const1[15:0]    0002    0003    0002
> bag2const2[15:0]    0001    0002    0001
> bag2position[15:0]  0017    0031    0017
```

and then calculated. The results are: $f(1) = 10$, $f(2) = 19$, $f(3) = 32$, and $f(4) = 49$. All values are in decimal. All simulation results are in binary or Hex. The Hex results are: $f(1) = A$, $f(2) = 13$, $f(3) = 20$, and $f(4) = 31$. When these hex values are converted to decimal, the simulation values are exactly the same as the calculated values.

Next, the second second order polynomial will be analyzed. The calculated results are: $f(1) = 2$, $f(2) = 7$, $f(3) = 14$, and $f(4) = 23$. The Hex simulation results are: $f(1) = 2$, $f(2) = 7$, $f(3) = E$, and $f(4) = 17$. When these hex values are converted to decimal, the simulation values are exactly the same as the calculated values.

The first third order polynomial will be analyzed next. The calculated results are: $f(1) = 6$, $f(2) = 21$, $f(3) = 52$, and $f(4) = 105$. The Hex simulation results are: $f(1) = 6$, $f(2) = 15$, $f(3) = 34$, and $f(4) = 69$. When these hex values are converted to decimal, the simulation values are exactly the same as the calculated values.

Next, the second third order polynomial will be analyzed. The calculated results are: $f(1) = 21$, $f(2) = 65$, $f(3) = 157$, and $f(4) = 315$. The Hex simulation results are: $f(1) = 15$, $f(2) = 41$, $f(3) = 9D$, and $f(4) = 13B$. When these hex values are converted to decimal, the simulation values are exactly the same as the calculated values.

7 FPGA Realization and Final Verification

According to the instruction for this lab, only the first set of test benchs will be implemented. Using the Nexys 4 DDR board, the constants will be set to a specific value. The position will be chosen by the 16 LEDs. LED 0 will choose position 0, LED 1 will choose position 1, all the way up to LED 15 will choose position 15. The same test bench conditions were tested. The results were the exact same as the simulation and the calculated value.

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 Babbage Testing

```
`timescale 1ns / 1ps
module Babbage_Difference_Engine_Test;
    reg [15:0] bag3const0;
    reg [15:0] bag3const1;
    reg [15:0] bag3const2;
    reg [15:0] bag3const3;
    wire[15:0] bag3position;
    reg [15:0] bag2const0;
    reg [15:0] bag2const1;
    reg [15:0] bag2const2;
    wire[15:0] bag2position;

            Third_Order_Babbage#(4) Bab3(
                .const0(bag3const0),
                .const1(bag3const1),
                .const2(bag3const2),
                .const3(bag3const3),
                .position(bag3position)
                );



            Second_Order_Babbage#(4) Bab2(
                .const0(bag2const0),
                .const1(bag2const1),
                .const2(bag2const2),
                .position(bag2position)

                );





        initial
        begin
        bag3const0 <= 1;
        bag3const1 <= 2;
        bag3const2 <= 2;
        bag3const3 <= 1;
        bag2const0 <= 5;
        bag2const1 <= 3;
        bag2const2 <= 2;
        #20;
        bag3const0 <= 7;
        bag3const1 <= 5;
        bag3const2 <= 6;
        bag3const3 <= 3;
        bag2const0 <= -1;
        bag2const1 <= 2;
        bag2const2 <= 1;
        #20;
        $finish;
        end


endmodule
```

B Second Order Babbage

```
`timescale 1ns / 1ps
module Second_Order_Babbage#(parameter Maxxvalue = 5)(
    input [15:0] const0,
    input [15:0] const1,
    input [15:0] const2,
    output [15:0] position

    );

    wire [15:0] f_curr [Maxxvalue:0];
    wire [15:0] f_next [Maxxvalue:0];
    wire [15:0] g_curr [Maxxvalue:0];
    wire [15:0] g_next [Maxxvalue:0];
    wire [15:0] h_curr [Maxxvalue:0];

    assign f_curr[0] = const0;
    assign g_curr[0] = const2 + const1;
    assign h_curr[0] = 2*const2;

    generate
    genvar i;
    for(i=0;i<Maxxvalue+1;i=i+1) begin:bab
        assign f_next[i] = f_curr[i] + g_curr[i];
        assign g_next[i] = g_curr[i] + h_curr[i];
        assign f_curr[i+1] = f_next[i];
        assign g_curr[i+1] = g_next[i];
        assign h_curr[i+1] = h_curr[i];
    end
    endgenerate
    assign position = f_curr[Maxxvalue];
endmodule
```

C Third Order Babbage

```
`timescale 1ns / 1ps
module Third_Order_Babbage#(parameter Maxxvalue = 5)(
    input [15:0] const0,
    input [15:0] const1,
    input [15:0] const2,
    input [15:0] const3,
    output [15:0] position
    );

    wire [15:0] f_curr [Maxxvalue:0];
    wire [15:0] f_next [Maxxvalue:0];
    wire [15:0] g_curr [Maxxvalue:0];
    wire [15:0] g_next [Maxxvalue:0];
    wire [15:0] h_curr [Maxxvalue:0];
    wire [15:0] h_next [Maxxvalue:0];
    wire [15:0] j_curr [Maxxvalue:0];
    wire [15:0] f_curr0;
    wire [15:0] f_curr1;
    wire [15:0] f_curr2;
    wire [15:0] f_curr3;
    wire [15:0] g_curr0;
    wire [15:0] g_curr1;
    wire [15:0] g_curr2;
    wire [15:0] h_curr0;
    wire [15:0] h_curr1;
    wire [15:0] j_curr0;

    assign f_curr0 = const0;
    assign f_curr1 = const3 + const2 + const1 + const0;
    assign f_curr2 = 8*const3 + 4*const2 + 2*const1 +
       const0;
    assign f_curr3 = 27*const3 + 9*const2 + 3*const1 +
       const0;
    assign g_curr0 = f_curr1-f_curr0;
    assign g_curr1 = f_curr2-f_curr1;
    assign g_curr2 = f_curr3-f_curr2;
    assign h_curr0 = g_curr1-g_curr0;
    assign h_curr1 = g_curr2-g_curr1;
    assign j_curr0 = h_curr1-h_curr0;

    generate
    genvar i;
    for(i=0;i<Maxxvalue+1;i=i+1) begin:bab
```

```
            assign f_curr[0] = f_curr0;
            assign g_curr[0] = g_curr0;
            assign h_curr[0] = h_curr0;
            assign j_curr[0] = j_curr0;
            assign f_next[i] = f_curr[i] + g_curr[i];
            assign g_next[i] = g_curr[i] + h_curr[i];
            assign h_next[i] = h_curr[i] + j_curr[i];
            assign f_curr[i+1] = f_next[i];
            assign g_curr[i+1] = g_next[i];
            assign h_curr[i+1] = h_next[i];
            assign j_curr[i+1] = j_curr[i];
        end
        endgenerate
        assign position = f_curr[Maxxvalue];
endmodule
```