CPE 480 Assignment 2
Multi-Cycle LOGICK

Mark Noblin, Tanner Wilkerson, CJ Vanderpool

***Abstract*—The goal of this assignment was to utilize the assembly code written in a previous assignment to develop and produce a multi-cycle processor design. The multi-cycle design should be able to execute most of the commands encoded with in the assembly using Verilog programming as a source for logic.**

I. GENERAL APPROACH

A. Assignment Specifications

The processor to be developed in this assignment needs to take care of executing a specific set of commands from the assembly encoding. Our group decided to use a fresh encoding that was more tailored to the needs of the assignment. The commands that this assignment focuses on executing are those that do not require the Logarithmic Number System, or LNS, that will be implemented later on. This means that the processor should be able to execute ad, an, or, no, eo, mi, sr, br, jr, li, si, co, lo, st, and sy. The encoding also has a macro developed for the la command, which was discussed in the last assignment. This operation, however, does not require anything from our processor as it simply tells the processor the proper combination of li and si to perform.

The advice given by the assignment is to stick to a top-down approach to designing the processor. Our group chose to do so, while also sticking to the ideology of breaking up our code chunks into seperate modules. This allows us to specify as much as possible what exactly verilog builds when the entire program is synthesized. It helps to avoid unecessary or messy synthesized blocks and helps in easing the stress of testing.

Our design makes use of a few very important structure blocks to achieve the needed functionality. We have an ALU that is capable of handling those operations that specifically require arithmetic to be performed. We also have a mainmem and regfile block that can not only read and write but perform shifts to the left and right as well. We a block currently being called the oracle that specifies the structure of the Finite State Machine, or FSM, used to determine the execution of instructions. Lastly, we have a few smaller blocks, such as the MDR, MAR, A, Y, and condition blocks that help in the movement and execution of data onto our bus.

The last step after designing all of this is our testing method. The first step in testing was to develop small code blocks capable of testing the logic of each of our instructions upon their execution using the operations we have available. For example, we might use ad to add two numbers and store them into memory, then use co to compare this to a predetermined result and see if this is correct. The testing program then would use a br operation to avoid a sy call that would halt the program if that result was incorrect. This provides a streamlined method for determining which operations function properly and which need work.

B. Code Implementation

Our code implementation for this project is roughly 400 lines, so for the sake of brevity, some examples will be given and important code will be addressed in this section.

The first piece of code to consider is the ALU. This unit is the heart of the processor, as most operations that can be performed are done so using operations built into the ALU. In looking at our encoding, the following operations are those that are specifically made to work with our ALU:

```
ad $.d, $.s, $.t := 0:4 .d:4 .s:4 .t:4
an $.d, $.s, $.t := 1:4 .d:4 .s:4 .t:4
or $.d, $.s, $.t := 2:4 .d:4 .s:4 .t:4
no $.d, $.s := 3:4 .d:4 .s:4 0:4
eo $.d, $.s, $.t := 4:4 .d:4 .s:4 .t:4
```

These five operations are defined within a case statement inside the ALU. The case statement is controlled by an input reg called c. The ALU receives this from the oracle module during execution of specific operations. The ALU operates on operands a and b, which are both given as inputs as well. Finally, after the ALU has performed calculation, it gives 2 outputs. One is of one word size, called out, that is the result of the calculation. The other is an 8 bit output of the condition result, which determines the state that the condition register should now be in based on the result of the calculation. This condition output is always calculated and returned in the event that it is needed at any point by the next operations. The code for the ALU in its entirety is as follows:

```
module alu(out, a, b, c, cond);
  input `word a, b;
  input [2:0] c;
  output reg `word out;
  output reg [7:0] cond;

  always@ (c) begin
    case(c)
          0: out = a + b;
          1: out = a & b;
          2: out = a | b;
          3: out = ~b;
          4: out = a ^ b;
    endcase
```

```
  end

  always@ (a, b)begin
    if (a>b) cond = 8'b00001111;
    else if (b>a) cond = 8'b01101001;
    else        cond = 00010001;
  end
endmodule
```

The next piece of code to consider is the Regfile Module. This module is going to hold the needed register values to be passed onto the bus when operations call for them. The module for this regfile is capable of taking in a clk variable for synchronization purposes, an in value for data, a control value, and a sel value for selecting one of the available registers. It only has one output for passing back out a fetched register value. The Regfile is capable of not only reading and writing values, but shifting the values left and right as well. These operations are all performed through a case statement that comprises most of the body of the module. The code for this Regfile module can be seen as follows:

```
module regfile(clk, out, in, c, sel);
  input clk;
  input `word in;
  input [3:0] c, sel;
  output `word out;
  reg `word r `regsize;

  always@ (posedge clk)begin
        case(regc)
        0: r[sel] <= r[sel];
        1: r[sel] <= in;
        2: r[sel] <= r[sel] << 1;
        3: r[sel] <= r[sel] >> 1;
  endcase
  end
  assign out = r[sel];
endmodule
```

The Mainmem module is very similar to the Regfile module seen above. It lacks a few of the fancier feature that the Regfile has as it does not have a use for them. Instead, the Mainmem only has the ability to read and write to and from its contents, with a single control input to determine which of the two to do. the code for the Mainmem module can be seen as follows:

```
module mainMem(clk, write, datain,
dataout, addr);
  input clk, write;
  input `word datain, addr;
  output `word dataout;

  reg `word memory `memsize;

  always@ (posedge clk) begin
        if (write) memory[addr] <= datain;
        dataout <= memory[addr];
  end

endmodule
```

Next, the load-store-shift registers, or lss_reg module is defined. This module aims to build all of the registers that aren't included in the Regfile module. It behaves similarly to the Regfile as well, including the shifting ability as well. It has an in and out for data movement, as well as an input c for control. The code for the lss_reg is as follows:

```
module lss_reg(clk, out, in, c)
  input [1:0] c;
  input `word in;
  output `word out;

  always@ (posedge clk)begin
    case(c)
        0: out <= out;
        1: out <= in;
        2: out <= out << 1;
        3: out <= out >> 1;
        endcase
  end
endmodule
```

II. Testing and Issues

A. Testing

Our testing is split up into a few different methods in order to ensure as complete of a testing routine as we can get. In this section, we'll step through all of these methods briefly and explain the general practice for testing using these methods.

The first method used is contained in the $Testing.txt$ file. This file contains multiple small blocks that aim to test the logic correctness of the operations defined in our assembly code. Each one has its own derived process for testing, however many of the operations have similar structure. for example, many of the operations tested from the ALU will place test operands into the operation to be tested, get a result from the operation, and use compare to

B. Known Issues

Our biggest known issue at the moment is that for one reason or another, our processor does not work. We currently have a very large portion of the code done, but there is no promise that it functions currently. As it stands at the moment of writing this, our issue lies in the state machine of our processor. Either some of our operations are defined incorrectly, or something larger is wrong with the design.