RISC-V CPU

Ramy Shehata-900222862

Omar Ganna-900222646

Mohamed Khaled-900225303

Spring 2024

Department of Computer Science and Engineering

The American University in Cairo

1. Introduction

This report will discuss the implementation of a RISC-V architecture on an Artix-7 FPGA using the Basys3 board. This CPU is designed to execute machine level code for a RISC-V CPU, where the implementation is done on Verilog. Our primary objective was to develop a CPU where every instruction is executed in exactly one clock cycle. This CPU supports a subset of the RV32i instruction set, excluding the informed instructions such as ECALL, EBREAK and FENCE instruction. The scope of this report will include how we designed the CPU Datapath, implementing the control unit, utilizing BRAM for program and data memory, and interfacing with the 7-segment display and the source codes of our Verilog Codes.

2. Design and Implementation

The CPU Datapath design is a critical aspect of this project, as it determines the flow of data and the execution of instructions within the CPU. Our goal is to create a single-cycle CPU, where every instruction completes in one clock cycle. This section outlines the block diagram of the CPU Datapath and provides detailed descriptions of each module within the Datapath.

2.1. CPU Datapath Design

Our Datapath design is done for a Single Cycle implementation where each instruction is done in 1 clock cycle. The block diagram represents the overall structure of the CPU Datapath. It includes the key components such as the program unit, the RegFile, the ALU where most instructions' main arithmetic logic is executed, and control units such as the main control unit and the branch control unit, as well as including specific control signals for branching jumping and loading and storing. The block diagram is included in our GITHUB under Datapath Drawing, will also be included in the appendix below.

2.2. Control Unit

The control unit is where the main control signals come out to drive the whole CPU. This is done by taking the opcode of the instruction that is processed by the program unit. And then it processes the needed control signal for each instruction.

For R Format→ the opcode is 51 and this triggers the ALUOp signal and doesn't trigger any other signal except writeToReg.

For I Format→ It checks if it is 3 or 19, if it is three it processes the Load word control signals such as memToReg and MemRead. If it is 19, it will process the ALUSrc so that it will take from the Immediate Generator.

For S Format→ The opcode is 33 and it will do the MemWrite control signal.

For SB Format, it will do the branch control signal for the branching control unit.

For LUI, AUIPC and Jumping instructions, there are specific control signals for each to make them work, such as AUIPC and Jump.

After the Processing of each opcode, then it will activate the rest of the needed blocks to do the needed instructions. Below is our Verilog Synthesized design of the Control Unit.

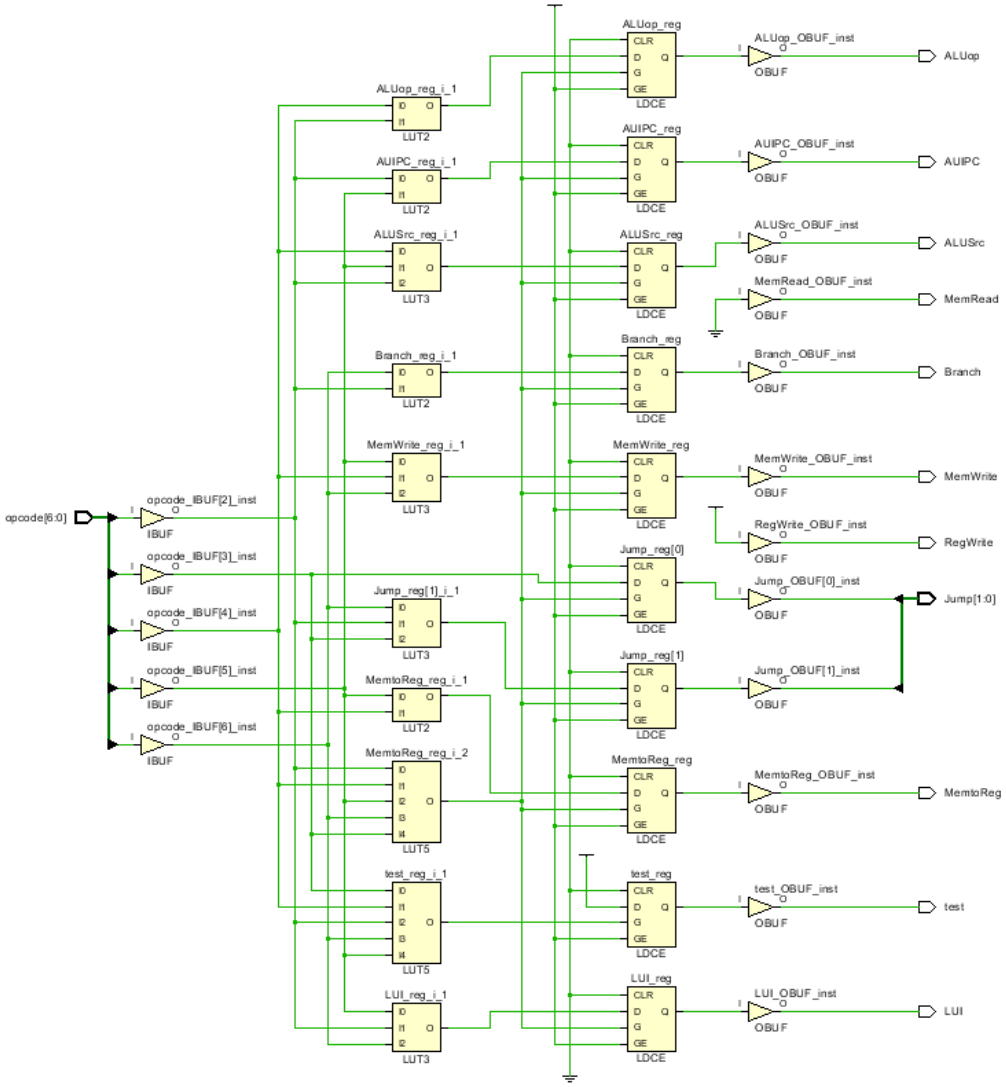


Figure 1: Control Unit Circuit

2.3. Clock Management

For the clock, we used two specific clocks by a clock divider not MMCM as we were familiar with how we implement a clock divider. In addition, we used two clocks, a 1Hz clock to showcase the instruction on the 7 Seg Display in the FPGA board and a 200Hz clock for the multiplexed display in the FPGA board. The reason a 1HZ clock was used was mainly so that the output of each instruction can be seen with our own eyes to illustrate that our implementation works as a Single Cycle Implementation on the FPGA. However, our code can withstand a 10MHZ clock normally. However, we didn't test the full 100MHZ clock of the FPGA to see if the instructions work correctly.

2.4. Memory Design

For our Memory, we used two different BRAMs, mainly a program ROM that holds the instructions of the assembly code, these instructions are given in a byte addressable format where each byte is specified by one slot. For example, Memory [0] holds the first 8 bits of the instructions and Memory [3:0] represents a word. This is done by little endian as in any RISC-V CPU. The other RAM is where we store values in our memory for load and store words, these are stored in the memory, which will be given by a separate file called datafile.txt. Each BRAM can hold up to 4KBs of memory. Below is our current representation of the Program Rom and the RAM:

Program ROM:

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  //* Module: ROM.v
4  //* Project: RISCV_CPU
5  //* Author: Omar Ahmed Ganna,  omarganna@aucegypt.edu
6  //* Description: Built the ROM (program data) which is used to store instructions
7  //* Change history: 15/05/24:  Created the Program data
8  //////////////////////////////////////////////////////////////////////////////////
9
10 module program_ROM #(parameter width = 8,
11 parameter depth = 4096,
12 parameter filename = "assembly.txt",
13 parameter address_width = $clog2(depth))
14 (
15 input wire  clk,
16 input wire [address_width-1:0] address,
17 output reg  [31:0] data
18 );
19 reg [width-1:0] memory [0:depth-1];
20 initial begin
21     if (filename != 0) begin
22         $readmemb("assembly.txt", memory);
23     end
24 end
25 always @(posedge clk) begin
26 data[7:0]=memory[address];
27 data[15:8]=memory[address+1];
28 data[23:16]=memory[address+2];
29 data[31:24]=memory[address+3];
30 end
31 endmodule
32
```

RAM:

```
`timescale 1ns / 1ps
/************************
 *
 * Module: memory_data.v
 * Project: RISCV_CPU
 * Author: Mohamed Abdelfatah Khaled, mohamedabdelfatah572@aucegypt.edu
 * Description: This module is the memory_data, which is responsible for
 * store and read the data.
 * Change history: 15/05/24:  Created the memory_data module
 *
 *
 ************************/
module memory_data # (parameter WIDTH=8,
    parameter DEPTH=4096,
    parameter INIT_F="",
    parameter ADDRW=$clog2(DEPTH))
    (
    input wire  clk,
    input wire  write_data,
    input wire read_data,
    input wire  [ADDRW-1:0] addr,
    input wire  [WIDTH-1:0] data_in,
    output reg  [31:0] data_out
    );
reg [WIDTH-1:0] memory [0:DEPTH-1];

initial begin
           $readmemb(INIT_F, memory);
end

always @(posedge clk) begin
       if (write_data)
           memory[addr] <= data_in;
        else if(read_data)
           data_out<=memory[addr];
end
```

In addition to the codes, we also take the assembly code from an external file using the $readmemb Function that takes the bits from the txt file and assigns it to the memory.

2.5. I/O Interface Design

For our I/O design, we used only the 7 Segment display and the FPGA's clock to output each instruction in the display, as was said in the clock section, we used a 200HZ clock for the multiplexed display and 1Hz to display each instruction. Below is a picture of one of the outputs on the FPGA using one of the testcases that will be discussed in the Testing Section.

Our usage of the FPGA is extremely minimized since we are only reliant on the FPGA's 7 segment display for our output. Otherwise, the other resources will not benefit us in our implementation of the RISC-V CPU on the FPGA.

2.6. Instruction Implementation

For the instructions, we used simplified test cases to test each format correctly. We first manually did every instruction in our testcases in machine code in a 32bit format. After that we made them in 8-bit format so that the ROM can read the file correctly and address each byte with the correct bits of that specific word. So each instruction in the ROM

is equal to 4 bytes in the memory as illustrated above from Mem[n+3,n] where in multiples of 4.

3. Test Cases and Validation

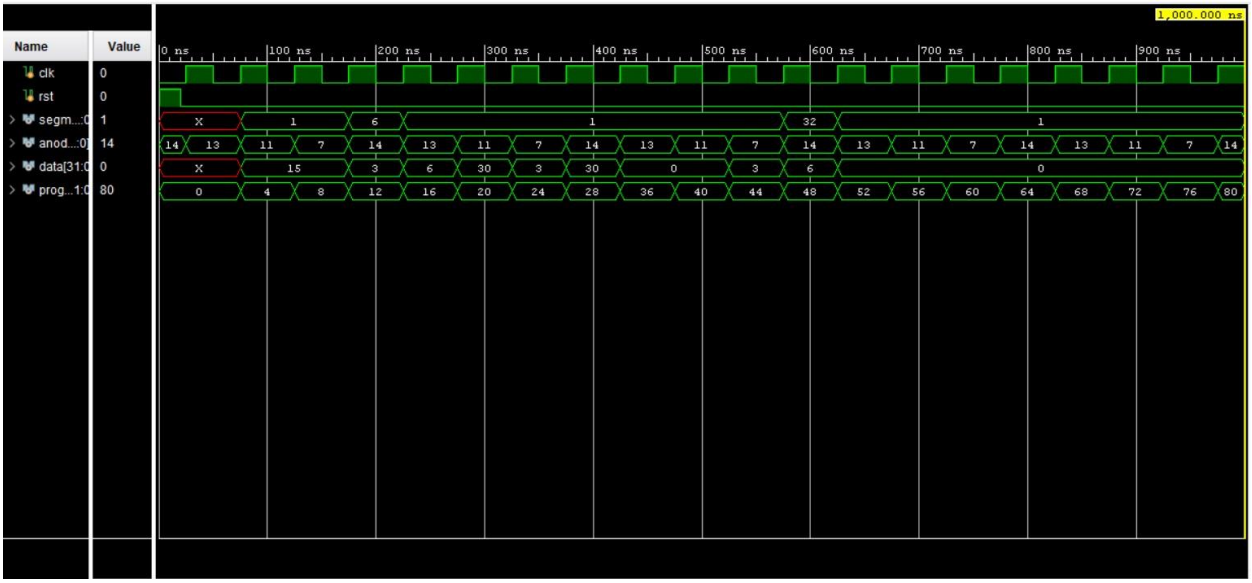
This section details our test cases and showcases the testbenches we created for smaller, simplified versions of the test cases to ensure the correct functionality of the instructions we implemented.

The first test case is designed to validate basic arithmetic and logical operations, including addition, subtraction, and the OR function. The test case sequence begins by loading a value into a register using the ADDI instruction, followed by adding the contents of two registers, subtracting the contents of another two registers, and performing a logical OR operation. Below is the implementation of the test case, represented in an 8-bit instruction format as discussed earlier.

Here is the following machine code:

```
10010011
00000001
11110000
00000000
10010011
00000000
00110000
00000000
00010011
00000010
01100000
00000000
10110011
10000010
00110000
00000000
00110011
10000001
01000010
01000000
10110011
11100011
00100010
00000000
```

And here is the testbench simulation



Although the final output is correct, some of the outputs in the instructions are not fully correct due to the clock issue which is discussed in the implementation issues section.

Next testcases test the bitwise functions such as XOR, ORI. ANDI and so on. It includes many I format instructions to ensure that they work as intended.

Below is the byte format and the testbench simulation.

For our third test case, we include some branching and shifting instructions to ensure that they also work correctly.

Below is the byte format and the testbench simulation.

4. Implementation issues

During our process of developing the single-cycle RV32i CPU, we encountered several significant issues, some of which are still unresolved at the time of writing this report. This section details the challenges we faced, the impact of these issues on our project, and the steps we are taking to address them.

4.1 Clock Synchronization Issues

One of the primary issues we encountered is related to clock synchronization. Specifically, the data in the first two instructions does not correctly sync with the clock. This misalignment causes instruction overlaps, leading to incorrect register values and outputs that deviate from the expected results. This problem has been particularly challenging to

debug, as it introduces a cascading effect that impacts subsequent instructions and their execution.

4.2 Program Counter (PC) Problems

The program counter has been another critical area of concern. The synchronization bug mentioned above causes the program counter to occasionally jump to incorrect instructions. This issue is especially problematic in the branching unit, where the program counter sometimes skips to the wrong instruction address. To mitigate this, we have had to insert an extra empty instruction for branching to work correctly. However, this is a temporary fix, and a more robust solution is needed to ensure accurate instruction flow.

4.3 Store and Jump Instructions

At the time of writing, the implementation of the store and jump instructions remains incomplete. The store instruction has not yet been fully tested and debugged, which affects our ability to write data back to memory accurately.

4.4 Instruction Formats

While most of the I-type, R-type, and SB-type instructions work as intended, there are still some issues with the S-type and U-type instructions. These instructions require rigorous testing to ensure their correct implementation.

5. Individual Contributions

With this massive project, all of us in the group contributed equally to the projects, in fact we all were together in the lab during the debugging phase for a few days. Therefore, individual contributions were mainly made together with no one taking a larger role than the other. For example, the 3 teammates would spend hours on the calling platform to discuss the Datapath diagram seen in the Github Repo. In addition, both Omar and Mohamed wrote most of the modules with Ramy assisting in a couple of them during debugging and testing. In addition, Ramy was responsible for writing the current report. However, Mohamed and Omar spent more time in debugging phase than Ramy due to distance problems in Ramy's end. However, Ramy was always on the phone and updated with the features and assisting in debugging from home if needed. In addition, all members contributed to the testcases done in the report above.

6. Conclusion

In conclusion, this project has involved the design and implementation of a single-cycle RV32i CPU on an Artix-7 FPGA using the Basys3 board. Our primary objective was to develop a CPU where each instruction is executed in exactly one clock cycle, excluding

specific system instructions, counters, and synchronization instructions. Throughout the project, we focused on creating a robust CPU datapath, an effective control unit, efficient memory management using BRAM, and a functional I/O interface.

The project allowed us to delve deeply into the architecture and operation of a RISC-V CPU, translating theoretical concepts into a practical, working model. Despite the challenges we encountered, such as clock synchronization issues, program counter problems, and incomplete implementation of certain instructions, we made significant progress. Our iterative approach to testing and validation helped identify and address many bugs, although some issues remain unresolved.

The detailed test cases we developed demonstrated the CPU's capability to perform basic arithmetic and logical operations, as well as more complex tasks involving branching and memory access. These tests were crucial in validating the functionality of our design and ensuring that our implementation adhered to the principles of a single-cycle CPU.

Moving forward, further work is needed to refine our implementation, particularly in addressing the remaining clock synchronization issues and completing the store and jump instructions. Additionally, rigorous testing of the S-type and U-type instructions is required to ensure their correct functionality.

Overall, this project has been a valuable learning experience, providing us with a deeper understanding of CPU design and FPGA implementation. The knowledge and skills we gained will be instrumental in our future endeavors in digital design and computer architecture. We are confident that with continued effort and refinement, our single-cycle RV32i CPU will achieve full functionality and reliability.