CSCE 3301 – Computer Architecture

Project 1: SINGLE CYCLE CPU

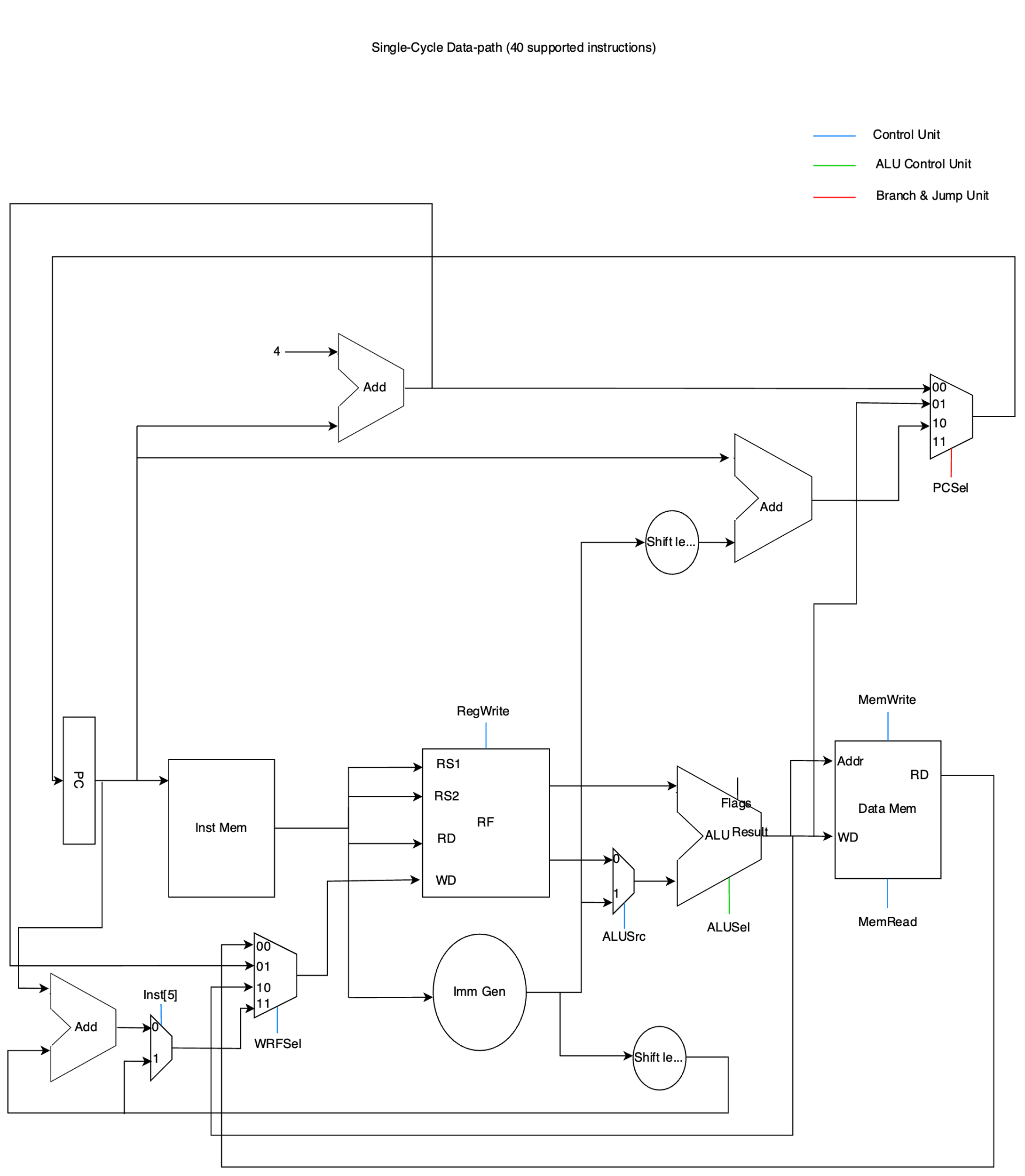
Youssef Elmahdy - 900212370

Hussein Heggi - 900213220

1. Project Objectives:

The primary goal of this project is to develop a complete single-cycle processor compliant with the RV32I standard, capable of executing 40 distinct instructions, including FENCE, ECALL, and EBREAK. In this context, FENCE and ECALL are treated as no-operation (nop) instructions, while EBREAK serves as a stopping command.

1. Data Path of CPU



1. Implementation

ALU Control Unit:

In the provided Verilog code, the ALU Control Unit takes ALUop, funct3, and funct7 as inputs to determine the operation mode for the ALU. It outputs ALUSel, a 4-bit signal that directs the ALU to perform the correct operation, such as addition, subtraction, or bitwise operations. This unit examines the ALUop to distinguish between different types of instructions (e.g., arithmetic, branch) and uses funct3 and funct7 to fine-tune the exact ALU operation required.

ALU:

The ALU in the given code performs arithmetic and logical operations based on the ALUSel input. It handles basic arithmetic operations like addition (ADD and SUB using two's complement for subtraction) and logical operations (AND, OR, XOR, shift left, shift right, signed shift right). The ALU outputs the result of these operations (ALU\_out) and sets flags such as zero (Zflag), carry (Cflag), overflow (Vflag), and sign (Sflag) to indicate specific conditions post-operation.

Branch Unit:

This unit uses the zero (Zflag), carry (Cflag), overflow (Vflag), and sign (Sflag) flags along with the branch instruction specifics (funct3) to make branching decisions (Branch\_Bit). For instance, it evaluates whether a branch should occur based on comparisons like equal (beq), not equal (bne), less than (blt), greater than or equal (bge), and their unsigned counterparts, thereby directing the program flow control.

Control Unit:

The Control Unit decodes the opcode of instructions to set control signals for the processor's operation. It determines the nature of the instruction (R-type, I-type, load, store, branch, etc.) and sets the signals like branch, jump, MemRead, MemWrite, ALUSrc, and RegWrite. These control signals are critical for guiding the processor's behavior, orchestrating which operations are to be executed in each cycle.

Data Memory:

Accessed during load and store operations, the Data Memory in the Verilog code uses the ALU\_out as the address. It operates based on control signals MemRead and MemWrite: if MemRead is active, it outputs data from the specified memory address; if MemWrite is active, it writes the input data to the specified address. This module represents the implementation of the processor's memory system, handling the storage and retrieval of data as the program executes.

Data Path:

The Data Path in the provided code interconnects all the functional units of the processor, including the ALU, the register file, the control unit, and memory. It ensures the proper routing of data between these components, managing the execution of instructions according to the control signals and the current state of the processor. The Data Path handles the immediate generation, PC calculations, branching decisions, and the updating of register values, embodying the processor's architecture and operational logic.

1. Difficulties & Solutions

Working on the single-cycle processor project faced several hurdles. A major challenge was the need to use Vivado, which meant we meant we had to come physically to the lab, so we stayed for 13 hours to complete it which was quite demanding. Debugging the Verilog code was also tough, as it involved dealing with complex parts of the processor like the ALU and Control Unit, and making sure they worked well together. Getting all parts of the processor to sync up correctly took a lot of trial and error.

1. Testing

We used a single test program to test 21 instructions out of the 32, as the rest were trivial and were tested before in the lab

* Test Program:

mem[0] = 32'b00000000100000001000000100010011; //addi x2,x1,8

mem[1] = 32'b00000000000000010000000110110011; //add x3, x2, x0

mem[2] = 32'b00000000000000001010001000000011; //lw x4,0(x1)

mem[3] = 32'b00000000010000000010011000100011; //sw x4, 12(x0)

mem[4] = 32'b00000000000000010000011001100011; //beq x2, x0, 12

mem[5] = 32'b00000000001000011111001010110011; //and x5, x3, x2

mem[6] = 32'b00000000000000101110010010110011; //or x9, x5, x0

mem[7] = 32'b00000000000000001000010001100011; //beq x1,x0,8

mem[8] = 32'b01000000000100010000000000110011; //sub x0, x2, x1

mem[9] = 32'b00000000001000010001010000010011; //slli x8, x2, 2

mem[10] = 32'b00000001100000000000000011101111; //jal x1, 24

mem[16] = 32'b00000000001111101000011110110111; //lui x15, 1000

mem[17] = 32'b00000000000001100100101000010111; //auipc x20, 100

mem[18] = 32'b01000000111110100000101010110011; //sub x21, x20, x15

mem[19] = 32'b00000000111100000110010001100011; //bltu x0, x15, 8

mem[20] = 32'b00000000000000000000000010110011; //add x1, x0, x0

mem[21] = 32'b00000000111110100010101100110011; //slt x22, x20, x15

mem[22] = 32'b00000000010000000001010001100011; //bne x0, x4, 8

mem[24] = 32'b00000000000000000001011001100011; //bne x0,x0,12

mem[25] = 32'b11111111111111111111010100110111; //lui x10,1048575

mem[26] = 32'b00000000010001010101010110010011; //srli x11, x10, 4

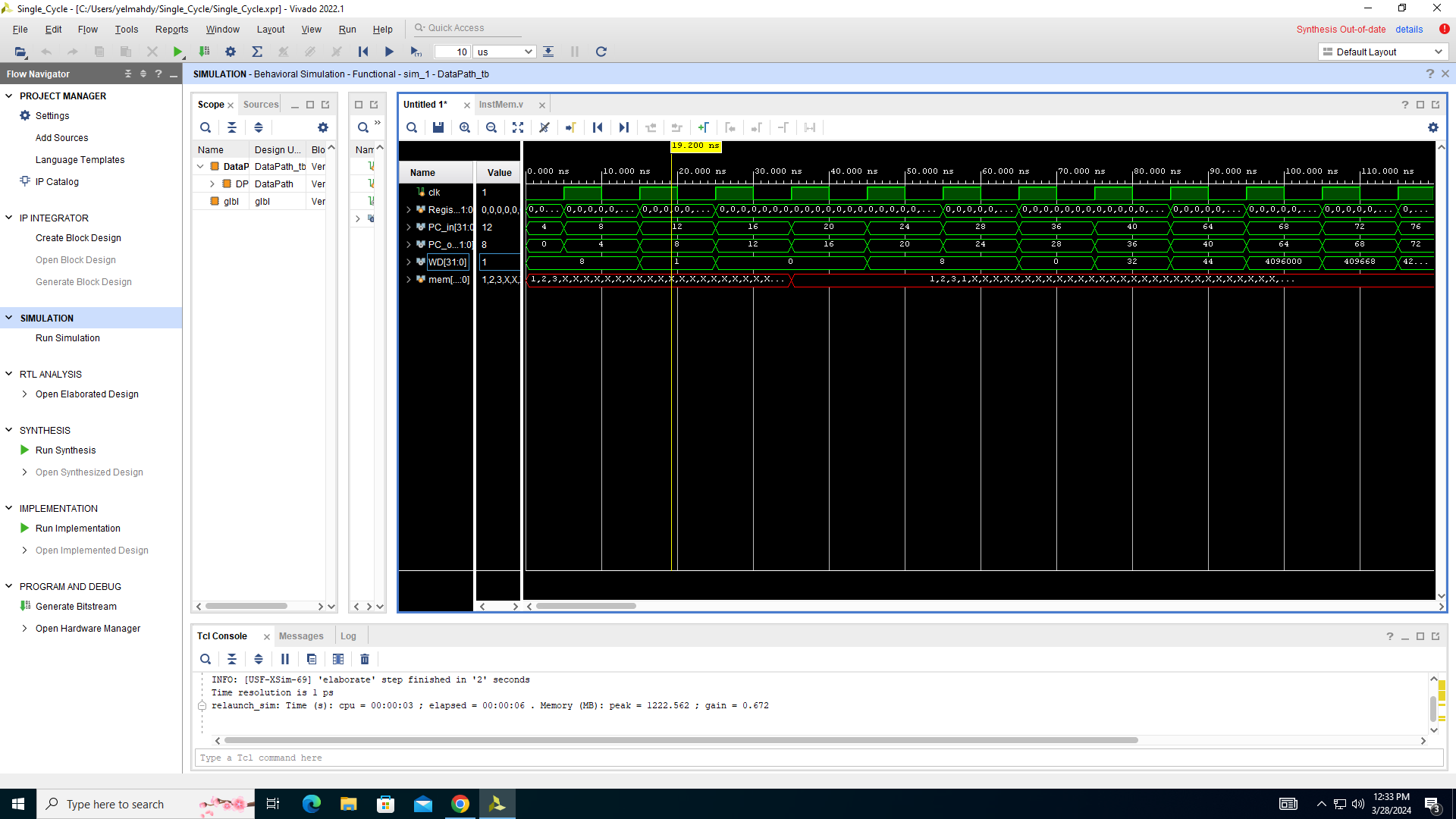
mem[27] = 32'b01000000010001010101011000010011; //srai x12, x10, 4

mem[28] = 32'b00000000000000011100001110010011; //xori x7, x3, 0

mem[29] = 32'b00000000101000011011010000010011; //sltiu x8, x3, 10

mem[30] = 32'b00000000000001111000000001100111; //jalr x0, 0(x15)

* Simulation:



A screenshot of a computer

Description automatically generated