Single Cycle rv32i Processor

CSCE 3301: Computer Architecture

The American University in Cairo

Youssef Elhagg - 900211812

Mariam ElGhobary - 900211608

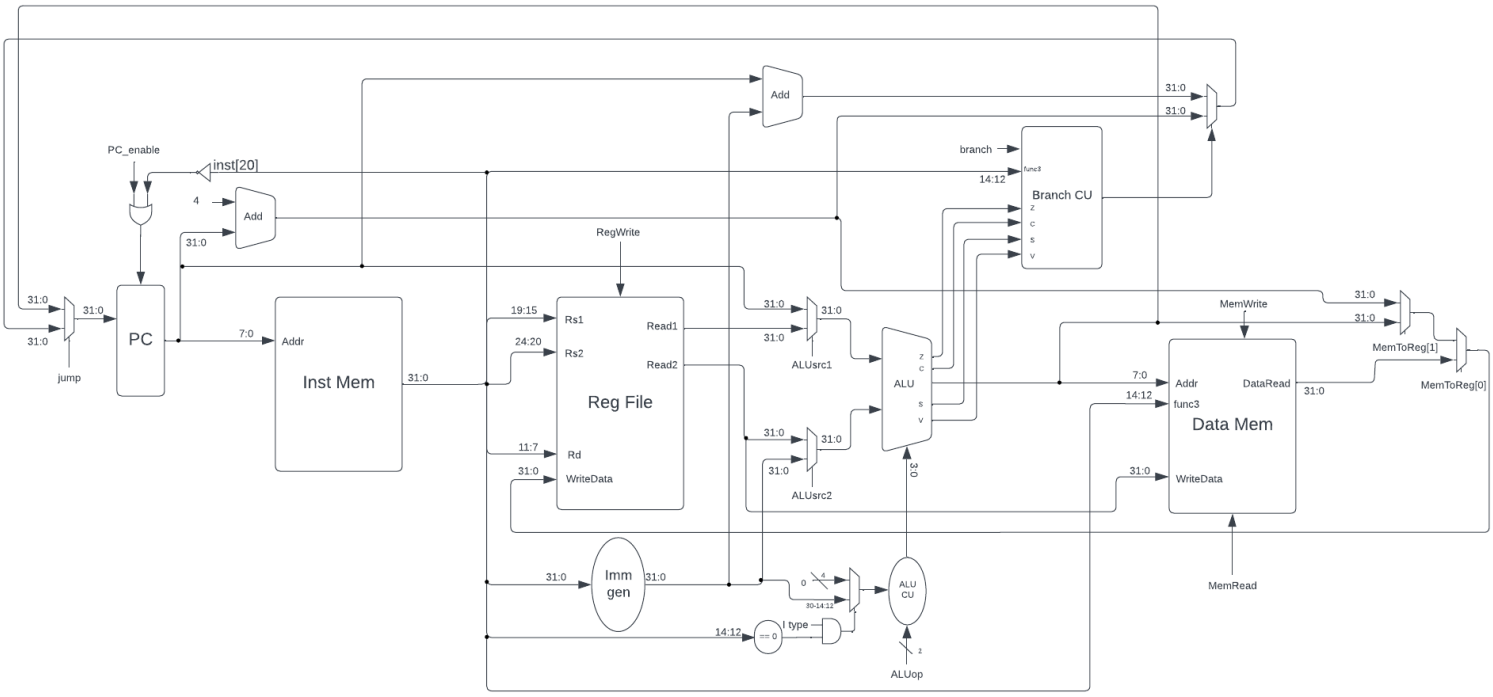
Dr. Cherif Salama

1. Milestone Objectives

In this milestone, we sought to use Verilog to create a single cycle CPU supporting all forty RV32I instructions as described in the specifications except ECALL, EBREAK, and FENCE instructions. We interpreted ECALL and FENCE instructions as no-op instructions and EBREAK as a halting instruction that ends program execution. Furthermore, we sought to ensure the correctness and completeness of our code through extensive testing.

2. Project Design

Our design utilizes two separate memories, one for data and one for instructions, and is a non-pipelined single-cycle processor. The datapath for our processor is presented below:



3. Milestone Implementation

Our datapath implementation was segmented into numerous modules which functioned as the building blocks for our processor. As well as creating certain modules from scratch, we utilized and adapted modules previously written in the Computer Architecture Lab course and integrated some modules provided by Dr.Cherif to ease the implementation process. The main modules are the following:

Control Unit: The control unit is responsible for generating control signals utilized by the other modules using the instruction opcode. These include but are not limited to: branch, memRead, memWrite, regWrite, PC_enable, memToReg, ALUop as well as other multiplexor selects.

Branch Control Unit: Takes in the 4 flags generated by the ALU alongside the branch signal and func3 and identifies whether or not the branch is to be taken and assigns the output accordingly. The output is used as a mux select which alongside the PC multiplexor and the jump signal from the control unit determine what the PC's next address will be.

PC: Our program counter was implemented as a 32-bit register taking a clock and an asynchronous reset signal as input and outputting the address of the upcoming instruction to be executed. The PC, if not halted, can either increment its current address by 4, increment by an immediate or increment by a shifted immediate. Which address is outputted by the PC next is determined by the branch and PC multiplexores with their respective select signals. The PC also

has a load input which when set to 1 allows the PC to update to the next address, else the program is halted which enables the implementation of EBREAK.

Instruction Memory: A byte-addressable memory using little endian addressing storing up to 256 bytes. The memory takes in an 8-bit address and outputs a 32-bit instruction.

Register File: Contains 32 32-bit registers and has 2 read-ports, 1 write-port and an asynchronous reset. Addresses to read from and write to are extracted from the instruction outputted by the instruction memory.Writing is synchronous and is determined by the control signal RegWrite. Writing only takes place when RegWrite is set to 1.

Immediate Generator: This module was provided by Dr.Cherif and is responsible for identifying and reordering immediate bits. Immediate length and bit order vary depending on the instruction type and so this module ensures the correct extraction of the immediate from all instruction types. The immediate generator also carries out any required shifts relating to immediate extraction. This includes shifting B and J-type immediates one bit to the left as they are stored as halfwords as well as shifting 12 bits to the left as required by U-type instructions.

ALU: The ALU was also provided by Dr. Cherif and is responsible for the execution of all arithmetic and logic operations on its 2 inputs. The selection of the operation executed depends on the 4-bit ALU select which is generated by the ALU control unit. It also takes in the 5-bits that contain any immediates utilized by the R and I-Type shift operations. The computed result is outputted as 32-bits alongside the carry flag, zero flag, sign flag and overflow flag.

ALU Control Unit: This is a control unit responsible for determining the appropriate ALU select to determine which operation is carried out by the ALU. It takes in ALUop which is a 2-bit signal generated by the control unit as well func3 and func7 from the original instruction and accordingly generates the correct select.

Data Memory: A byte-addressable memory using little endian addressing storing up to 256 bytes. The memory takes in an 8-bit address and outputs 32-bits or stores data as a byte, halfword or word. The control signals MemRead and MemWrite enable the reading and writing from and to memory respectively.

4. Difficulties Encountered

The primary difficulty was in the inflexibility of the working times and location. Since Vivado is only accessible through the restricted number of computers in the university laboratories, and since online simulators such as CloudV are currently out of order, a lot of work had to be done without testing in order to ensure timely progress. Consequently collections of modules were tested together at once which led to slightly lengthier debugging. This was tackled through the construction of various test cases each testing specific aspects of the code separately in addition to us writing our code in a clear and organized manner, tracing back errors was fairly quick and straight-forward.

We also encountered a quite unfortunate circumstance whereby after we completed our coding and testing the most recent files were accidentally overwritten. Nevertheless, we thankfully kept

record of the errors we fixed throughout the days during which we worked on the project and so completing our program for the second time was thankfully relatively smooth.

5. Program Testing

We created our own test cases by writing programs in assembly on RARS and then dumping the machine code and inserting it in our instruction memory. Some test cases were focused on testing a certain subset of the forty instructions whilst the others were more purposeful programs such as test case 2.

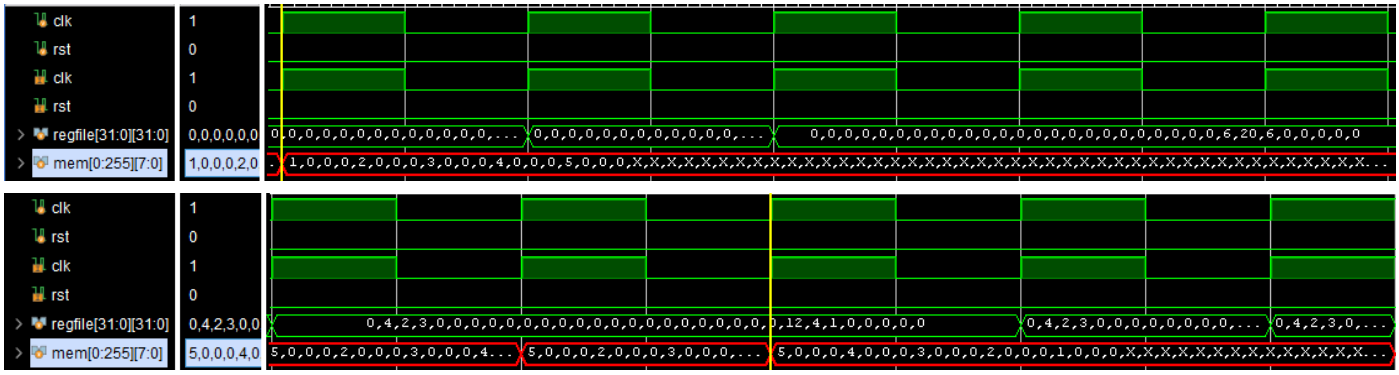
A total of 9 tests were conducted and are shown below:

Test case 1: checking that previous instructions constructed in the lab work

{mem[3], mem[2], mem[1], mem[0]} = 32'b000000000000_00000_010_00001_0000011; //lw x1, 0(x0)
{mem[7], mem[6], mem[5], mem[4]} = 32'b0000000_00000_00001_110_00100_0110011; //or x4, x1, x0
{mem[11], mem[10], mem[9], mem[8]} = 32'b0000000_00100_00000_010_01100_0100011; //sw x4, 12(x0)

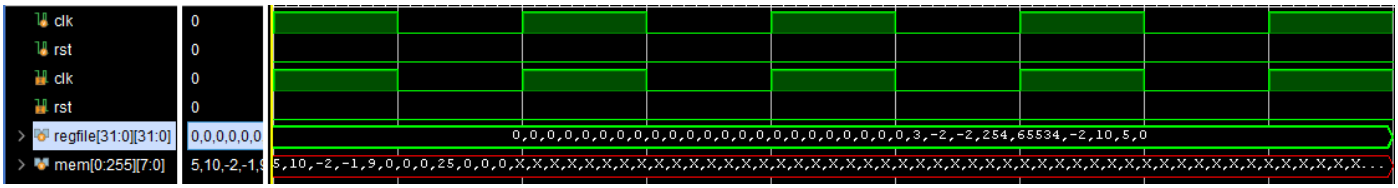
Test case 2: storing values from 1-5 then swapping them 5-1

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000100000000001010010011; //addi t0, zero, 1 # x = 1
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000000000000000001100010011; //addi t1, zero, 0 # i = 0
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000011000000000001110010011; //addi t2, zero, 6 # y = 6
{mem[15], mem[14], mem[13], mem[12]} = 32'b00000000011100101000101001100011; //beq t0, t2, endStore # i == y?
{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000010100110010000000100011; //sw t0, 0(t1) # mem[i] = x
{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000010000110000001100010011; //addi t1, t1, 4 # i+=4
{mem[27], mem[26], mem[25], mem[24]} = 32'b00000000000100101000001010010011; //addi t0, t0, 1 # x++
{mem[31], mem[30], mem[29], mem[28]} = 32'b11111110000000000000100011100011; //beq zero, zero, store # loop back
{mem[35], mem[34], mem[33], mem[32]} = 32'b00000000000000000000001010010011; //addi t0, zero, 0 # i = 0
{mem[39], mem[38], mem[37], mem[36]} = 32'b00000000000000000000001100010011; //addi t1, zero, 0 # *i
{mem[43], mem[42], mem[41], mem[40]} = 32'b00000001000000000000001110010011; //addi t2, zero, 16 # *(4-i)
{mem[47], mem[46], mem[45], mem[44]} = 32'b00000000001100000000111000010011; //addi t3, zero, 3 # x = 3
{mem[51], mem[50], mem[49], mem[48]} = 32'b00000011110000101000001001100011; //beq t0, t3, endSwap # i == x?
{mem[55], mem[54], mem[53], mem[52]} = 32'b00000000000000110010111010000011; //lw t4, 0(t1) # temp1 = mem[i]
{mem[59], mem[58], mem[57], mem[56]} = 32'b00000000000000111010111100000011; //lw t5, 0(t2) # temp2 = mem[4-i]
{mem[63], mem[62], mem[61], mem[60]} = 32'b00000001110100111010000000100011; //sw t4, 0(t2) # mem[4-i] = temp1
{mem[67], mem[66], mem[65], mem[64]} = 32'b00000001111000110010000000100011; //sw t5, 0(t1) # mem[i] = temp2
{mem[71], mem[70], mem[69], mem[68]} = 32'b00000000000100101000001010010011; //addi t0, t0, 1 # i++
{mem[75], mem[74], mem[73], mem[72]} = 32'b00000000010000110000001100010011; //addi t1, t1, 4 # *(i+1)
{mem[79], mem[78], mem[77], mem[76]} = 32'b11111111110000111000001110010011; //addi t2, t2, -4 # *(4-i-1)
{mem[83], mem[82], mem[81], mem[80]} = 32'b11111110000000000000000011100011; //beq zero, zero, swap # loop back



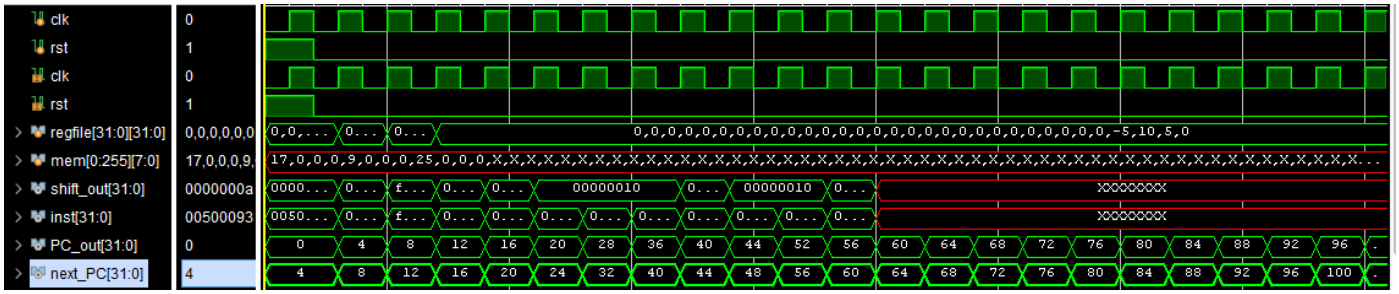
Test case 3: testing all remaining load and store instructions

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000010100000000000010010011; //addi x1, x0, 5 # x1 = 5
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000010100000000000100010011; //addi x2, x0, 10 # x2 = 10
{mem[11], mem[10], mem[9], mem[8]} = 32'b11111111111100000000000110010011; //addi x3, x0, -2 # x3 = -2
{mem[15], mem[14], mem[13], mem[12]} = 32'b11111111111100000100001000010011; //addi x8, x1, -2 # x8 = 3
{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000000010000001000000100011; //sw x1, 0(x0) # mem[0:3] = 5
{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000000010000000000000100011; //sb x2, 1(x0) # mem [0] = 5; mem[1:3] = 10
{mem[27], mem[26], mem[25], mem[24]} = 32'b00000000000011000000010001001100011; //sh x3, 2(x0) # mem[0] = 5; mem[1] = 10;
mem[2:3] = -2
{mem[31], mem[30], mem[29], mem[28]} = 32'b00000000000010000001010010000000011; //lhu x4, 2(x0) # x4 = 65534
{mem[35], mem[34], mem[33], mem[32]} = 32'b00000000000010000001000010100000011; //lbu x5, 2(x0) # x5 = 254
{mem[39], mem[38], mem[37], mem[36]} = 32'b00000000000010000000010011000000011; //lh x6, 2(x0) # x6 = -2
{mem[43], mem[42], mem[41], mem[40]} = 32'b00000000000010000000000001110000011; //lb x7, 2(x0) # x7 = -2



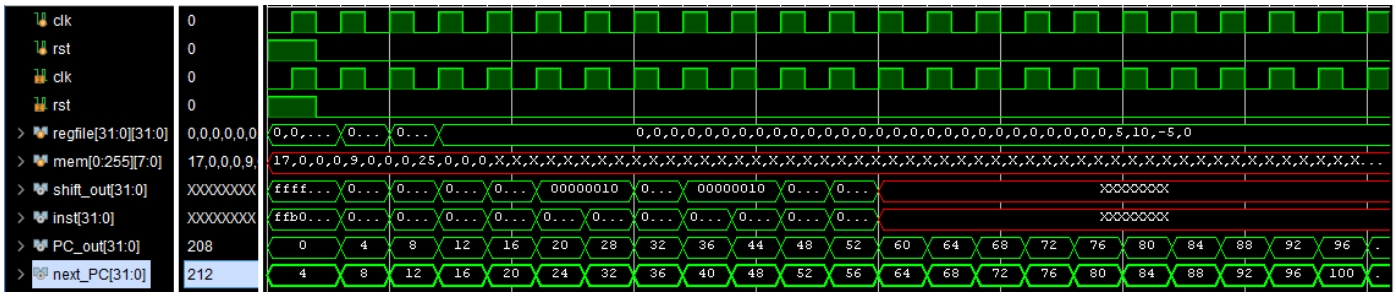
Test case 4: testing all branch instructions

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000010100000000000010010011; //addi x1, x0, 5
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000010100000000000100010011; //addi x2, x0, 10
{mem[11], mem[10], mem[9], mem[8]} = 32'b11111111110110000000000110010011; //addi x3, x0, -5
{mem[15], mem[14], mem[13], mem[12]} = 32'b00000000000100000100001000110011; //beq x1, x2, l1 # not taken
{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000000110000100101000110011; //bne x1, x3, l2 # taken
{mem[27], mem[26], mem[25], mem[24]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[31], mem[30], mem[29], mem[28]} = 32'b00000000000010001110001000110011; //blt x3, x1, l3 # taken
{mem[35], mem[34], mem[33], mem[32]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[39], mem[38], mem[37], mem[36]} = 32'b00000000000010001111001000110011; //bltu x3, x1, l4 # not taken
{mem[43], mem[42], mem[41], mem[40]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[47], mem[46], mem[45], mem[44]} = 32'b00000000001100001101010001100011; //bge x1, x3, l5 # taken
{mem[51], mem[50], mem[49], mem[48]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[55], mem[54], mem[53], mem[52]} = 32'b00000000001100001111010001100011; //bgeu x1, x3, l6 # not taken
{mem[59], mem[58], mem[57], mem[56]} = 32'b00000000000000000000000000110011; //add x0, x0, x0



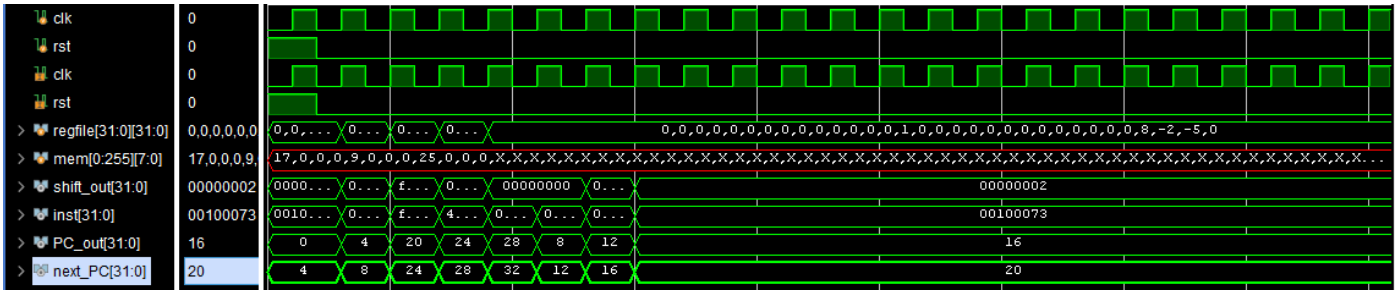
Test case 5: 2nd test for all branch instructions

{mem[3], mem[2], mem[1], mem[0]} = 32'b11111111101100000000000010010011; //addi x1, x0, -5
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000101000000000000100010011; //addi x2, x0, 10
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000001010000000000110010011; //addi x3, x0, 5
{mem[15], mem[14], mem[13], mem[12]} = 32'b00000000000100000100001000110011; //beq x1, x2, l1 # not taken
{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000000110000100101000110011; //bne x1, x3, l2 # taken
{mem[27], mem[26], mem[25], mem[24]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[31], mem[30], mem[29], mem[28]} = 32'b00000000000010001110001000110011; //blt x3, x1, l3 # not taken
{mem[35], mem[34], mem[33], mem[32]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[39], mem[38], mem[37], mem[36]} = 32'b00000000000010001111001000110011; //bltu x3, x1, l4 # taken
{mem[43], mem[42], mem[41], mem[40]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[47], mem[46], mem[45], mem[44]} = 32'b00000000001100001101010001100011; //bge x1, x3, l5 # not taken
{mem[51], mem[50], mem[49], mem[48]} = 32'b00000000000000000000000000110011; //add x0, x0, x0
{mem[55], mem[54], mem[53], mem[52]} = 32'b00000000001100001111010001100011; //bgeu x1, x3, l6 #taken
{mem[59], mem[58], mem[57], mem[56]} = 32'b00000000000000000000000000110011; //add x0, x0, x0



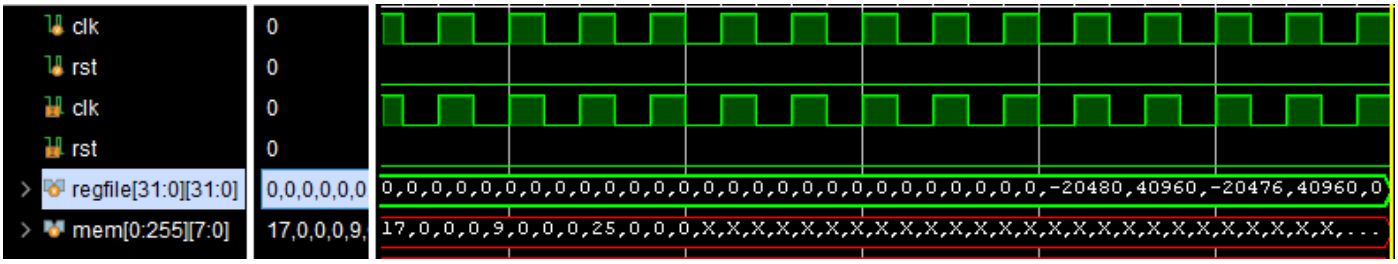
Test case 6: testing JAL, JALR, and system calls

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000100000000100010010011; //addi x17, x0, 1
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000001000000000000000111101111; //jal x3, l1
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000000000000000000001110011; //ecall
{mem[15], mem[14], mem[13], mem[12]} = 32'b00000001000100000000000000001111; //fence
{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000000010000000000001110011; //ebreak
{mem[23], mem[22], mem[21], mem[20]} = 32'b11111111101100000000000010010011; //addi x1, x0, -5
{mem[27], mem[26], mem[25], mem[24]} = 32'b01000000001000001101000100010011; //srai x2, x1, 2
{mem[31], mem[30], mem[29], mem[28]} = 32'b00000000000000001100000001100111; //jalr x0, x3, 0



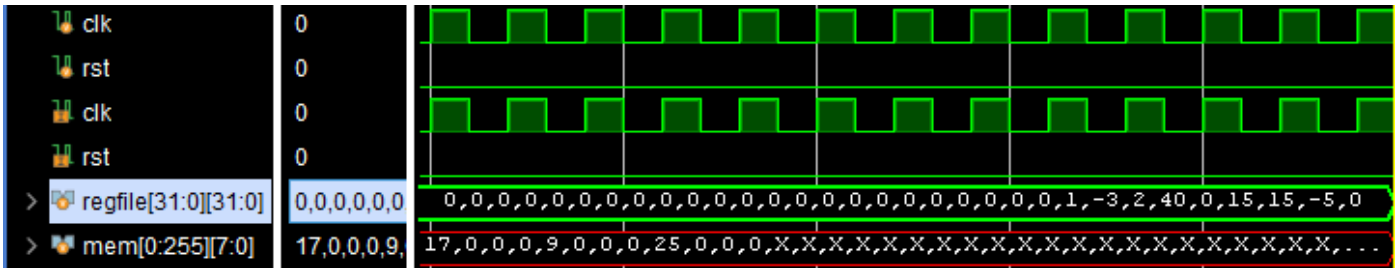
Test case 7: testing U-type

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000000001010000010010111; //auipc x1, 10
{mem[7], mem[6], mem[5], mem[4]} = 32'b11111111111111111011000100010111; //auipc x2, -5
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000000000001010000110110111; //lui x3, 10
{mem[15], mem[14], mem[13], mem[12]} = 32'b11111111111111111011001000110111; //lui x4, -5



Test case 8: testing all Arithmetic-I instructions

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000101000000000000010010011; //addi x1,x0,10 #x1=10
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000101000001100000100010011; //xori x2,x1,5 #x2=15
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000101000001110000110010011; //ori x3,x1,5 #x3=15
{mem[15], mem[14], mem[13], mem[12]} = 32'b00000000101000000111001000010011; //andi x4,x0,10 #x4=0
{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000000100000100100100010011; //slli x5,x1,2 #x5=40
{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000000100000110100100010011; //srli x6,x1,2 #x6=2
{mem[27], mem[26], mem[25], mem[24]} = 32'b11111111110110000000000010010011; //addi x1, x0, -5 #x5=-5
{mem[31], mem[30], mem[29], mem[28]} = 32'b01000000000100000110100111010011; //srai x7,x1,1 #x7=-3
{mem[35], mem[34], mem[33], mem[32]} = 32'b00000000101000001010010000010011; //slti x8,x1,10 #x8=1
{mem[39], mem[38], mem[37], mem[36]} = 32'b00000000101000001011010010010011; //sltiu x9,x1,10 #x9=0



Test case 9: testing all R-type instructions

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000101000000000000010010011; //addi x1, x0, 10 #x1=10
{mem[7], mem[6], mem[5], mem[4]} = 32'b11111111101100000000000100010011; //addi x2, x0, -5 #x2=-5
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000001000001000000110110011; //add x3, x1, x2 #x3=5
{mem[15], mem[14], mem[13], mem[12]} = 32'b01000000001000001000001000110011; //sub x4, x1, x2 #x4=15
{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000001000001100001010110011; //xor x5, x1, x2 #x5=-15

{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000001000001110001100110011; //or   x6, x1, x2 #x6=-5
{mem[27], mem[26], mem[25], mem[24]} = 32'b00000000001000001111001110110011; //and x7, x1, x2 #x7=10
{mem[31], mem[30], mem[29], mem[28]} = 32'b00000000101000010001010000110011; //sll x8, x2, x1 #x8=-5120
{mem[35], mem[34], mem[33], mem[32]} = 32'b00000000101000010101010010110011; //srl x9, x2, x1 #x9=4194303
{mem[39], mem[38], mem[37], mem[36]} = 32'b01000000101000010101010100110011; //sra x10,x2, x1 #x10=-1
{mem[43], mem[42], mem[41], mem[40]} = 32'b00000000001000001010010110110011; //slt x11,x1,x2 #x11=0
{mem[47], mem[46], mem[45], mem[44]} = 32'b00000000001000001011011000110011; //sltu x12,x1,x2 #x12=1