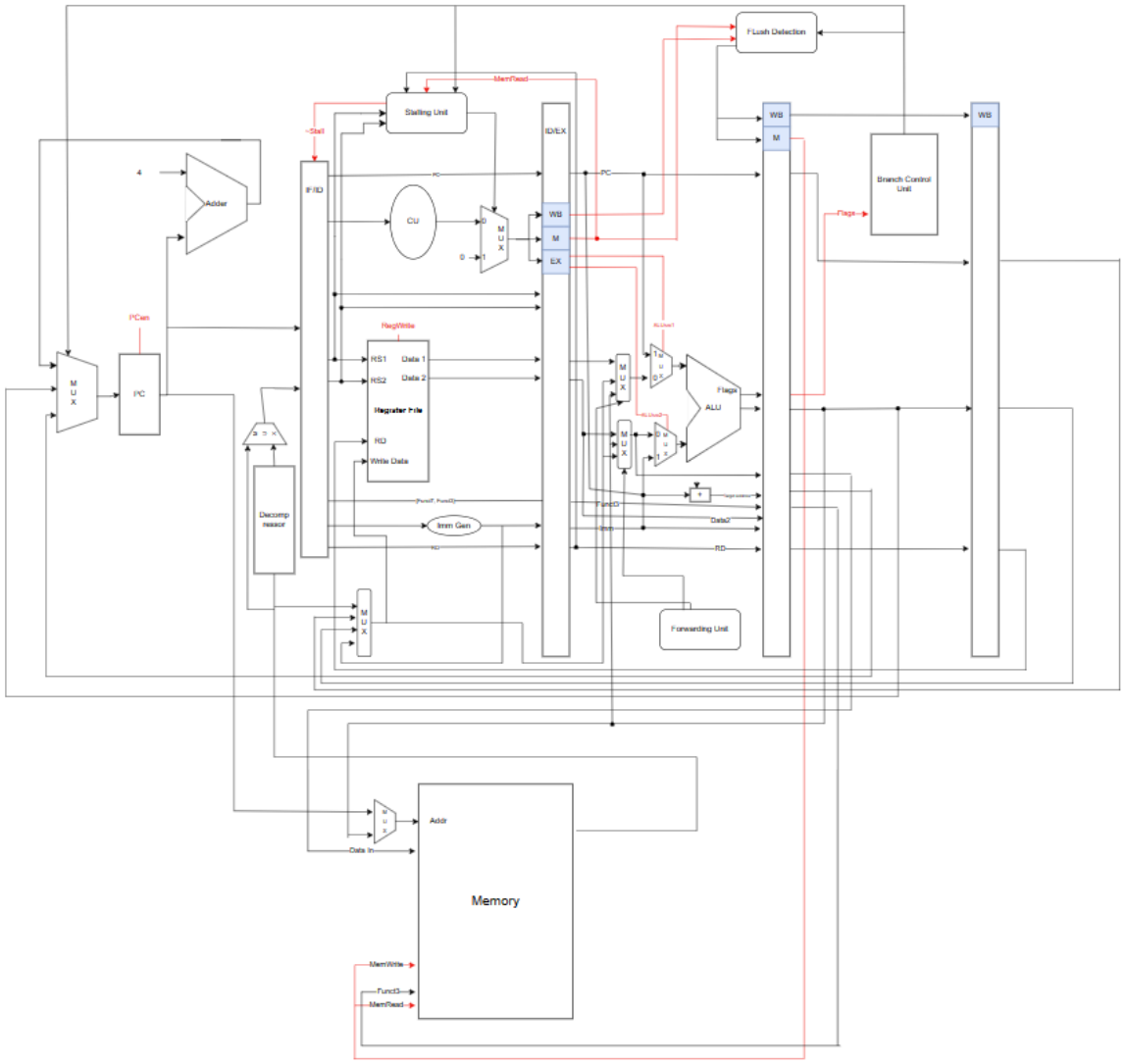
CSCE 3301 – Computer Architecture
Project 1: PIPELINED CPU
Nour Kasaby - 900211955
Nadine Safwat - 900212508

1. Project Objectives
   The objective of this project is to create a fully functional RV32-IC pipelined processor with full hazard handling and one memory for both instructions and data

2. Design of the project



** Clearer Image attached in submission

1. Implementation
   - PC: Implemented as a register that increments by 4 if the PCen signal is on, it will also check the BranchAnd signal and the Jump signal to either add the immediate of the ALU_out to the PC instead of adding 4 (To branch or jump).

   - Single Memory: A very simple register file of size 256 with the assumption that instructions are stored first and data is stored immediately afterwards. We handled the structural hazard of having one memory by creating a signal called Using_Mem which is on if we need to read or write from memory, and if this signal is on the required signals are chosen and the pipeline is stalled for one cycle, otherwise the signals required for reading an instruction are sent and there is no stall. To account for the data not starting at 0 an offset is added accordingly.

   - RegisterFile: We pass in the read and write sources which we can extract from the Instruction we get from the InstMem module. We will reset all registers to 0 if the rst signal is on,

otherwise, using the given inputs we will always read the two sources but only write to the RD if the WriteReg signal is on.

- ALU: The ALU takes in the chosen ALU_A as A and ALU_B as B and ALUSel as S. The ALU will carry out arithmetic or logical operations on the two inputs A and B according to the ALUSel provided. It will also assign all the flags which will be used in the Branch Control Unit (All flags are generated by subtracting B from A).

- Control Units: (Not displayed in datapath for simplicity)
    - Control Unit: The control unit assigns the control signals of each instruction based on the last bits of the opcode.

    - ALU Control Unit: The ALU control unit will receive func3 of every instruction and the ALUOp and accordingly it will assign the ALUSel which would decide which ALU operation to be carried out in the ALU.

    - Branch Control Unit: The branch control unit will take in func3 and flags of the branch instructions and will assign the BranchAnd according to the specified flag comparisons for each branch type.

    - Forwarding Unit: Will receive the source registers from the execute stage and the return register of the write back stage as well as the write signals for the register from both the memory and write back stage, then a series of comparisons are done to determine of forwarding is needed and from where and signals forward A and Forward B are assigned accordingly. These signals will be used as selection lines for 2 muxes that assign the input to the ALU sources.

    - Stalling Unit: Will receive the source register from the decode stage and the return register from the execute stage as well as the memory read signal in the execute stage and based on the required comparisons will set the stall signal to 1. If the stall signal is high the IF/ID register will not load, the PC will not increment and the control signals going into the ID/EX register will be zeroed.

    - Flushing Muxes: The flushing unit is a series of muxes that will check if a branch or jump is required and will insert a NOP instruction into the IF/ID register and zero the control signals going into both the ID/EX and EX/MEM register.

- Decompressor: We created a decompressor to handle the compressed format of our supported instructions by decompressing them to their original 32 bit format. We have tested all the instructions separately in a testbench; however, we did not have time to incorporate it with our test cases or come up with test cases that contain a mixture of both compressed and decompressed instructions.

3. Difficulties Encountered and Solutions

Predictably our biggest difficulty encountered was attempting to implement the single memory as there were a few signals that kept getting lost from us and, at some point, our store instructions refused to work. However, after a lot of testing we decided to deal with the structural hazard by stalling the pipeline everytime there is a memory use.

Initially we also had a few problems trying to follow our pipeline in our testing but this was overcome with time as we got more and more used to understanding which signal came from which stage.

The datapath was also extremely complex to draw and many signals were not shown simply because there was no space for them

A major difficulty for us was time management as debugging always seemed to take too much time.

4. Testing Procedure

We split the testing of the 40 instructions on a couple of test programs, each test program testing a few different instructions. We did 4 test programs in total aside from the one provided from the lab, three of which were programs that we wrote simply to test whether the instructions work or not rather than being an actual function which does a certain operation and one test program was a function that writes an array in memory and swaps it.

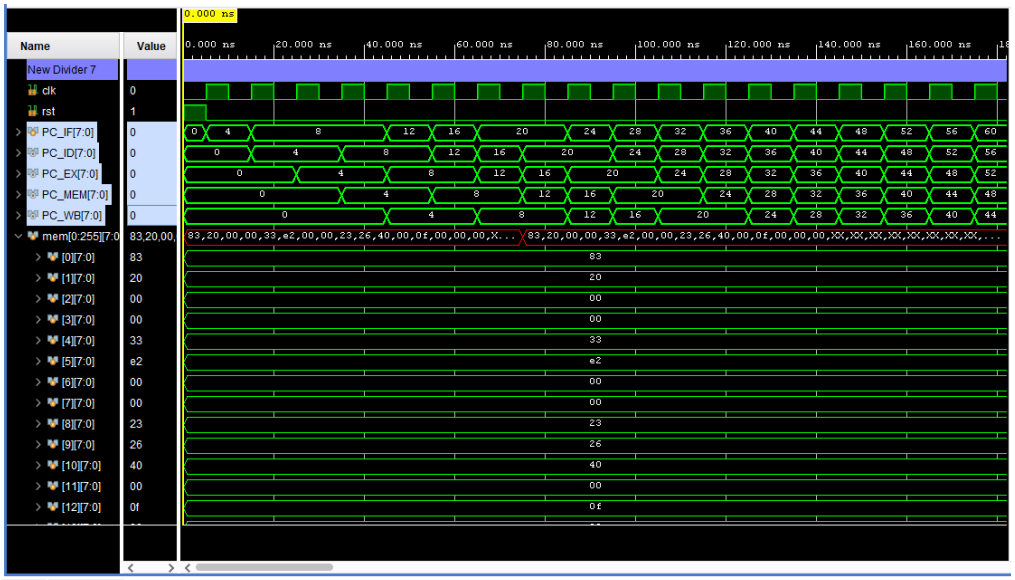
- Test program 1:

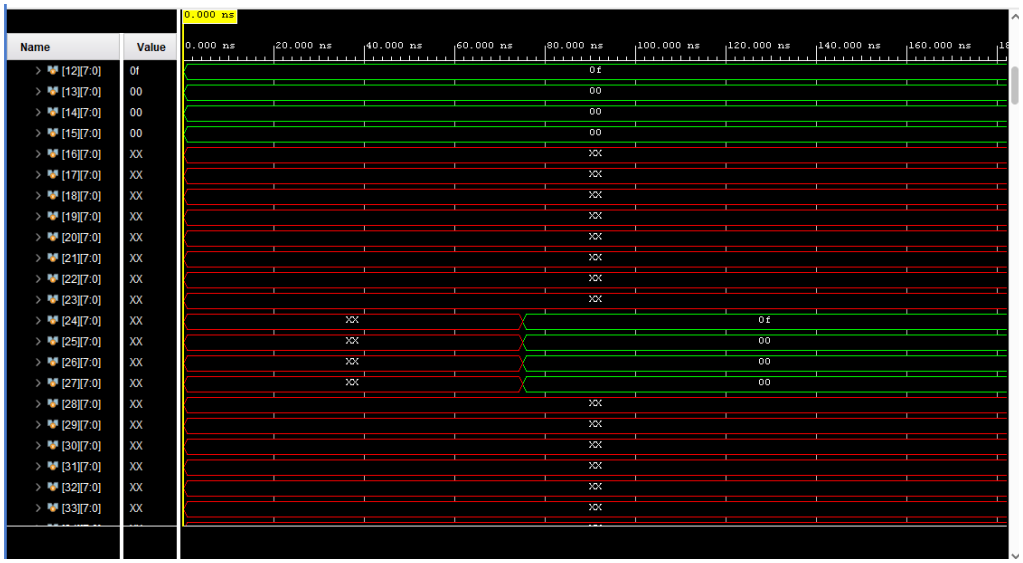
  (INST)

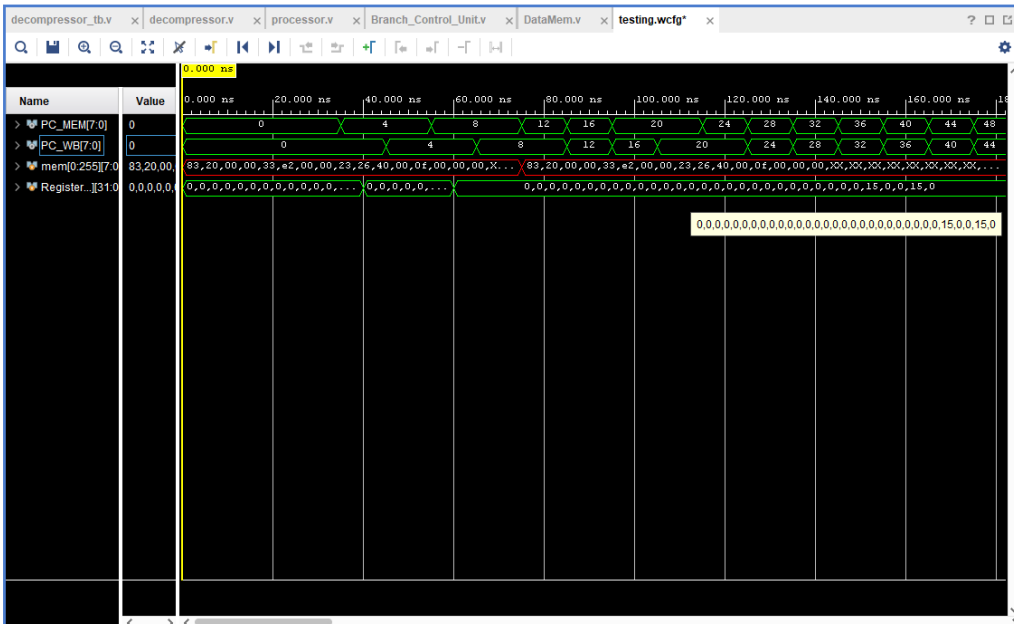
  {mem[3], mem[2], mem[1], mem[0]} = 32'b0000000000000_00000_010_00001_0000011; //lw x1, 0(x0) # x1 = mem[0:3]
  {mem[7], mem[6], mem[5], mem[4]} = 32'b0000000_00000_00001_110_00100_0110011; //or x4, x1, x0 # x4 = x1
  {mem[11], mem[10], mem[9], mem[8]} = 32'b0000000_00100_00000_010_01100_0100011; //sw x4, 12(x0) # mem[12:15] = x4

  (DATA)

  {mem[15], mem[14], mem[13], mem[12]} = 32'd15;

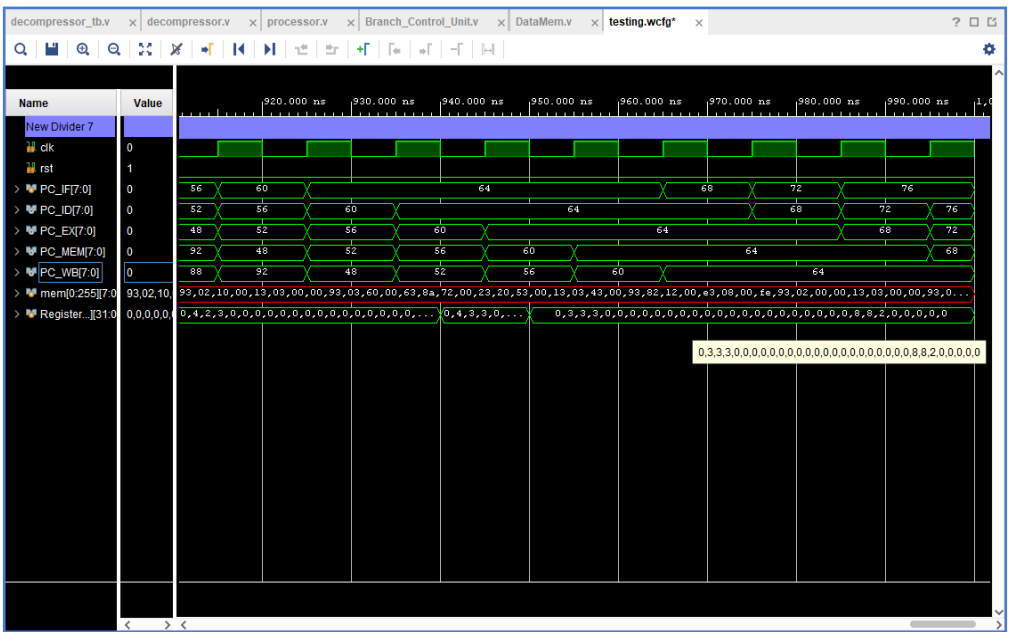


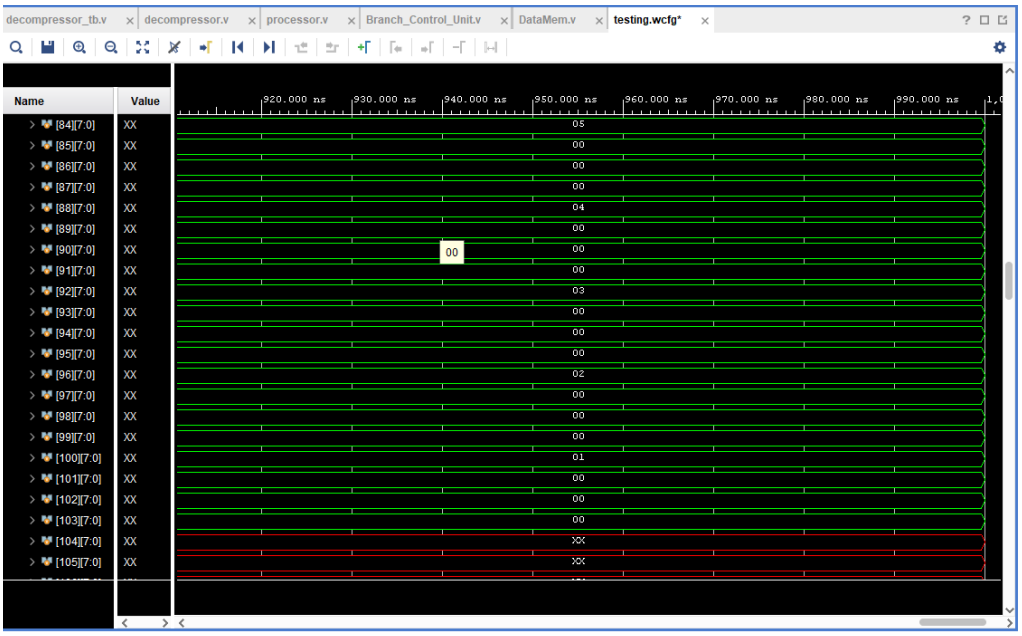




- Test program 2:

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000010000000000001010010011; //addi t0, zero, 1 # x = 1
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000000000000000000001100010011; //addi t1, zero, 0 # i = 0
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000011000000000000001110010011; //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'b00000000000010010100000101010010011; //addi t0, t0, 1 # x++
{mem[31], mem[30], mem[29], mem[28]} = 32'b11111111000000000000100011100011; //beq zero, zero, store # loop back
{mem[35], mem[34], mem[33], mem[32]} = 32'b00000000000000000000000001010010011; //addi t0, zero, 0 # i = 0
{mem[39], mem[38], mem[37], mem[36]} = 32'b00000000000000000000000001100010011; //addi t1, zero, 0 # *i
{mem[43], mem[42], mem[41], mem[40]} = 32'b00000001000000000000000001110010011; //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'b00000000000000011001011010000011; //lw t4, 0(t1) # temp1 = mem[i]
{mem[59], mem[58], mem[57], mem[56]} = 32'b00000000000000011101011110000011; //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'b00000000000010010100000101010010011; //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





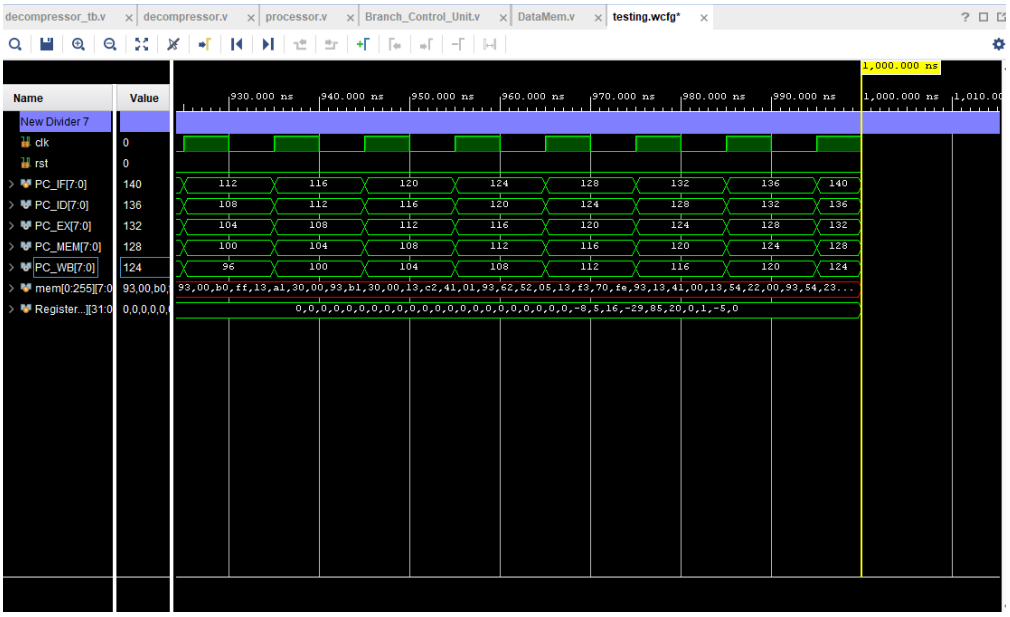
- Test Program 3:

Assembly:

```
addi x1, x0, -5 #X1 = -5
slti x2, x1, 3 #x2 = 1
sltiu x3,x1, 3 #x3 = 0
xori x4, x3, 20 #x4 = 20
ori x5, x4, 85 #x5 = 85
andi x6, x1, -25 #x6 = -29
slli x7, x2, 4 #X7 = 16
srli x8, x4, 2 #x8 = 5
srai x9, x6, 2 #x9 = -8
```

Machine:

{mem[3], mem[2], mem[1], mem[0]} = 32'b11111111101100000000000010010011;
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000000110000101000100010011;
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000000110000101100011010011;
{mem[15], mem[14], mem[13], mem[12]} = 32'b 00000001010000011100001000010011;
{mem[19], mem[18], mem[17], mem[16]} = 32'b00000101010100100110001010010011;
{mem[23], mem[22], mem[21], mem[20]} = 32'b11111110011100001111001100010011;
{mem[27], mem[26], mem[25], mem[24]} = 32'b00000000010000010001001110010011;
{mem[31], mem[30], mem[29], mem[28]} = 32'b00000000000100010010101000010011;
{mem[35], mem[34], mem[33], mem[32]} = 32'b01000000000100011010101001010011;



- Test 4:

Assembly:

```
lbu x1, 0(x0) #x1 = 24
lh x2, 4(x0) #x2 = 2
lb x3, 8(x0) #x3 = -126
lhu x4, 12(x0) #x4 = 3
sll x5, x1, x2 #x5 = 96
slt x6, x3, x1 #x6 = 1
sltu x7, x3, x1 #x7 = 0
xor x8, x6, x5 #x8 = 97
srl x9, x3, x2 #x9 = 1073741792
sra x10,x3, x2 #x10 = -32
```

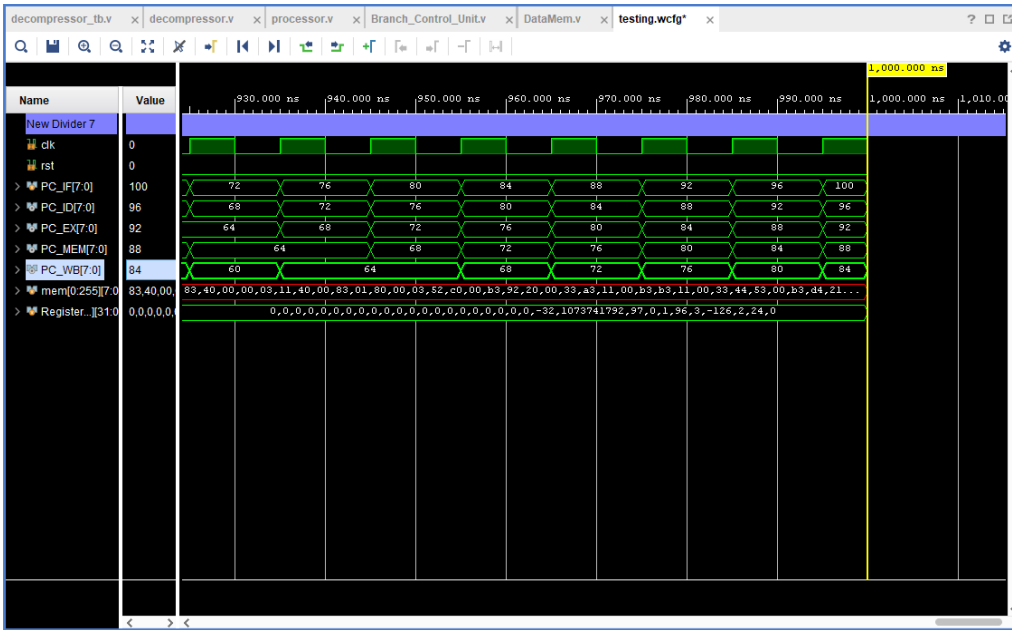
Machine Code:

//INST

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000000000100000010000011;
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000010000000001000100000011;
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000100000000000000110000011;
{mem[15], mem[14], mem[13], mem[12]} = 32'b00000000110000000101001000000011;
{mem[19], mem[18], mem[17], mem[16]} =32'b00000000001000001001001010110011;
{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000000100011010001100110011;
{mem[27], mem[26], mem[25], mem[24]} = 32'b00000000000100011011001110110011;
{mem[31], mem[30], mem[29], mem[28]} = 32'b00000000010100110100010000110011;
{mem[35], mem[34], mem[33], mem[32]} = 32'b00000000001000011101010010110011;
{mem[39], mem[38], mem[37], mem[36]} = 32'b01000000001000011101010100110011;

//DATA

{mem[39], mem[38], mem[37], mem[36]} = 32'b00000000000000000000000000011000;
{mem[43], mem[42], mem[41], mem[40]} = 32'b00000000000000000000000000000010;
{mem[47], mem[46], mem[45], mem[44]} = 32'b00000000000000000000000010000010;
{mem[51], mem[50], mem[49], mem[48]} = 32'b00000000000000000000000000000011;



- Test 5:

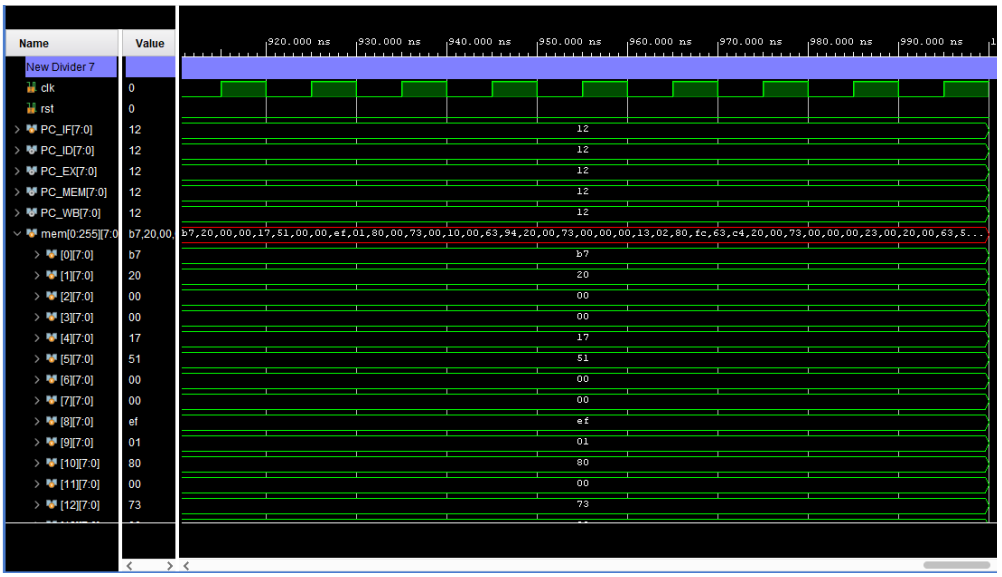
Assembly:

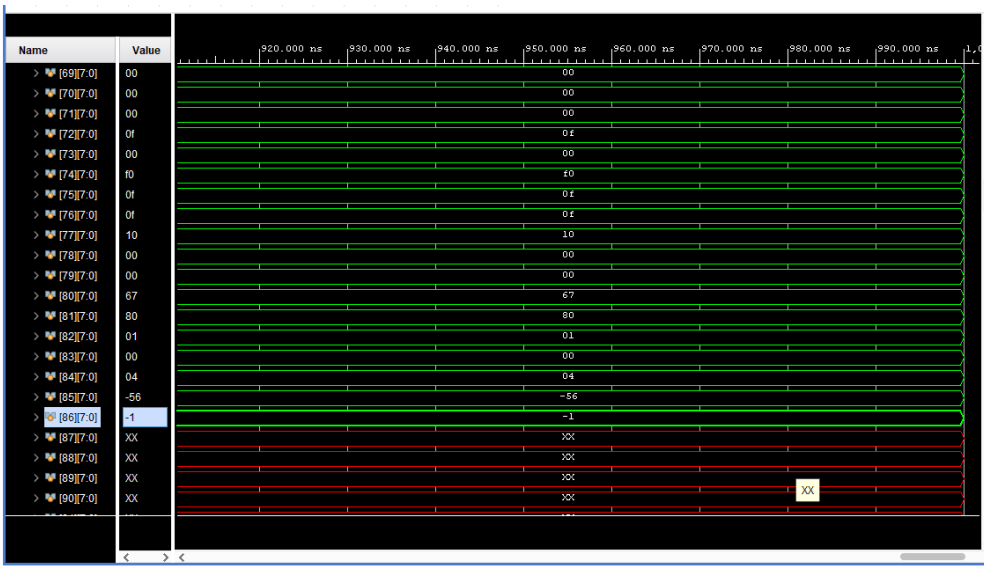
```
lui x1, 2 #x1 = 8192
auipc x2, 5 #x2 = 20484
jal x3, 8 # x3 = 12 ; PC => 16
ebreak #Halt PC
bne x1, x2, 8 #PC => 24
ecall
addi x4, x0, -56 #x4 = -56
blt x1, x2, 8 #PC => 36
ecall
```

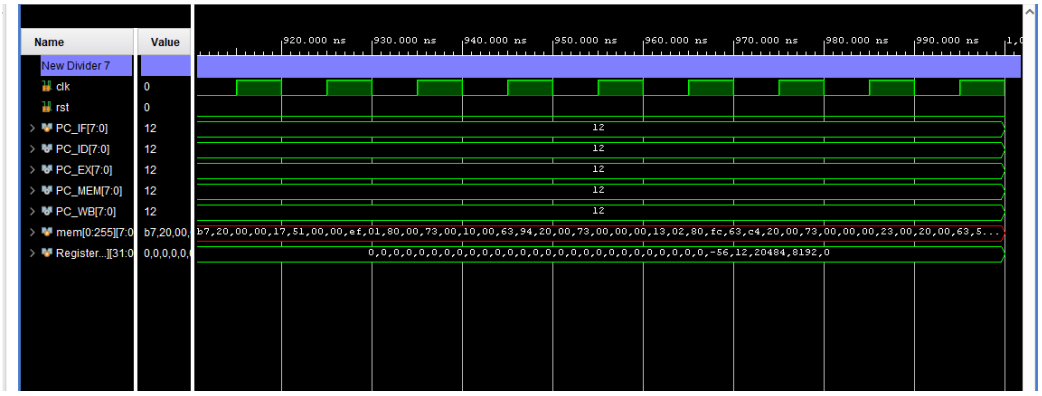
```
sb x2, 0(x0)  #mem[0] = 4
bge x2, x1, 8 #PC => 48
ecall
bltu x1, x4, 8 #PC => 56
ecall
bgeu x4, x2, 8 #PC => 64
ecall
sh x4, 1(x0)   #mem[2:1] = -56
ecall                  #nop
fence                  #nop
fence.i                #nop
jalr x0, x3, 0 #PC => 12
```

Machine Code:

{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000000000010000010110111;
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000000000000101000100010111;
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000100000000000000111101111;
{mem[15], mem[14], mem[13], mem[12]} = 32'b00000000000010000000000000001110011;
{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000001000001001010001100011;
{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000000000000000000000001110011;
{mem[27], mem[26], mem[25], mem[24]} = 32'b11111100100000000000001000010011;
{mem[31], mem[30], mem[29], mem[28]} = 32'b00000000001000001100010001100011;
{mem[35], mem[34], mem[33], mem[32]} =32'b 00000000000000000000000001110011;
{mem[39], mem[38], mem[37], mem[36]} = 32'b00000000001000000000000000100011;
{mem[43], mem[42], mem[41], mem[40]} = 32'b00000000000010001010101000100011;
{mem[47], mem[46], mem[45], mem[44]} = 32'b00000000000000000000000001110011;
{mem[51], mem[50], mem[49], mem[48]} = 32'b00000000010000001110010001100011;
{mem[55], mem[54], mem[53], mem[52]} = 32'b00000000000000000000000001110011;
{mem[59], mem[58], mem[57], mem[56]} =32'b00000000001000100111010001100011;
{mem[63], mem[62], mem[61], mem[60]} = 32'b00000000000000000000000001110011;
{mem[67], mem[66], mem[65], mem[64]} =32'b00000000010000000010000010100011;
{mem[71], mem[70], mem[69], mem[68]} =32'b00000000000000000000000001110011;
{mem[75], mem[74], mem[73], mem[72]} =32'b00001111111100000000000000001111;
{mem[79], mem[78], mem[77], mem[76]} = 32'b00000000000000000001000000001111;
{mem[83], mem[82], mem[81], mem[80]} = 32'b00000000000000011000000001100111;




Name
Value
920.000 ns
930.000 ns
940.000 ns
950.000 ns
960.000 ns
970.000 ns
980.000 ns
990.000 ns
[69][7:0] 00
[70][7:0] 00
[71][7:0] 00
[72][7:0] 0f
[73][7:0] 00
[74][7:0] f0
[75][7:0] 0f
[76][7:0] 0f
[77][7:0] 10
[78][7:0] 00
[79][7:0] 00
[80][7:0] 67
[81][7:0] 80
[82][7:0] 01
[83][7:0] 00
[84][7:0] 04
[85][7:0] -56
[86][7:0] -1
[87][7:0] XX
[88][7:0] XX
[89][7:0] XX
[90][7:0] XX
XX


Name
Value
920.000 ns
930.000 ns
940.000 ns
950.000 ns
960.000 ns
970.000 ns
980.000 ns
990.000 ns
New Divider 7
clk 0
rst 0
PC_IF[7:0] 12
PC_ID[7:0] 12
PC_EX[7:0] 12
PC_MEM[7:0] 12
PC_WB[7:0] 12
mem[0:255][7:0] b7,20,00,
Register...][31:0] 0,0,0,0,0,
b7,20,00,00,17,51,00,00,ef,01,80,00,73,00,10,00,63,94,20,00,73,00,00,00,13,02,80,fc,63,c4,20,00,73,00,00,00,23,00,20,00,63,5...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-56,12,20484,8192,0