**Project 1 MS3 Report**

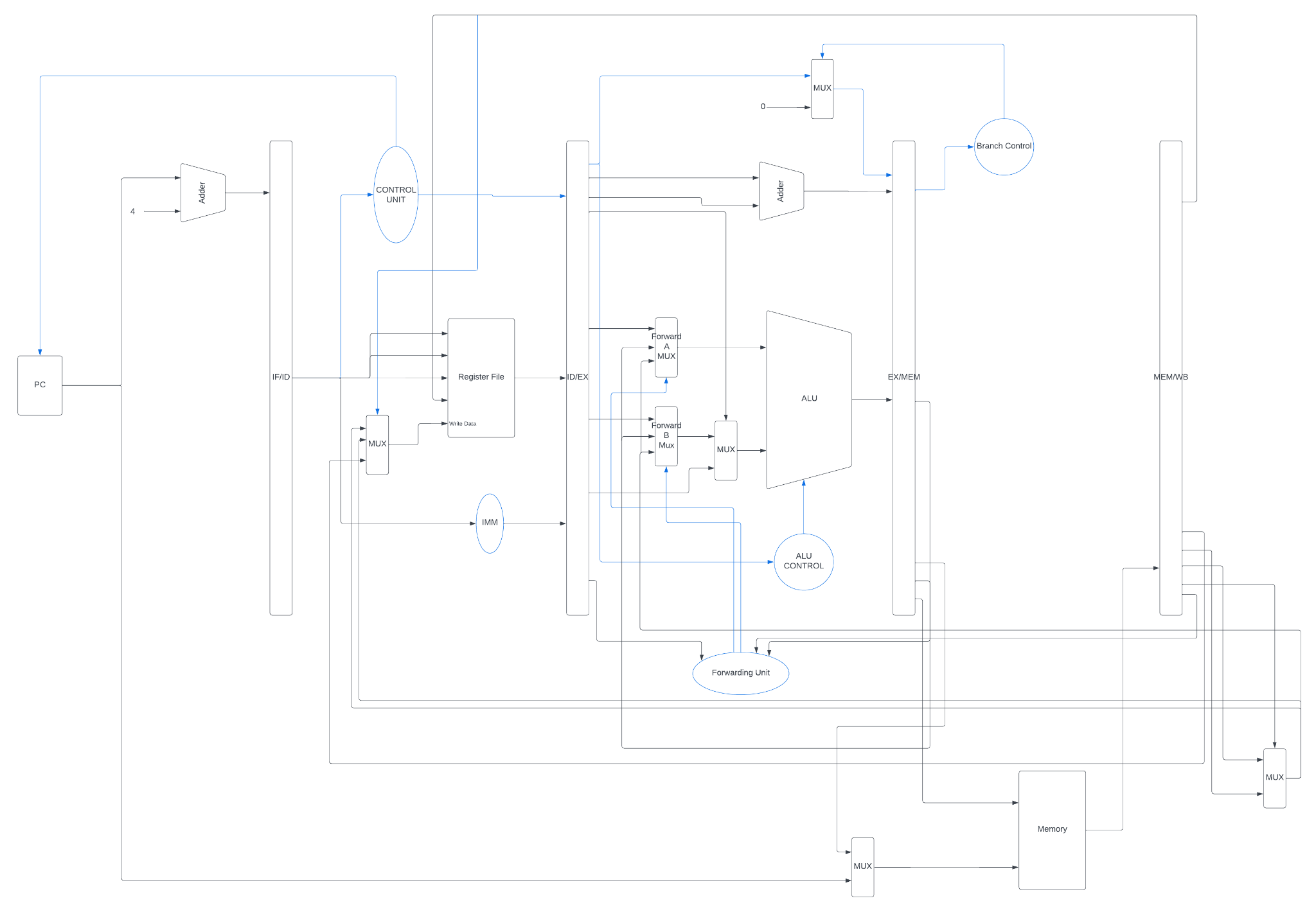
*Ramy Badras - 900194248 Zein Noureddin - 900212591*

***Project Objectives***

Implementing a pipelined RISC-V processor with a single memory for data and instructions that supports the RV32I base integer instruction set.

***Block Diagram & Datapath***

The block diagram for the single cycle datapath can be found below.

******

***Implementation***

In this milestone, we took our modules from the previous milestone and started by adding the pipeline registers, which we took from the modules in lab 7. Needless to say, we had to add several signals to the registers, such as those coming out of the newly added branch control unit, the signals we added to the control unit to accommodate the 40 instructions, the flags coming out from the ALU, etc.

Following that, we added the implementations for the forwarding unit, the hazard detection unit, and the flushing and ensured that their signals were propagated correctly.

After that, we merged the data and instruction memories into a single memory, at which point we realized that we don’t need the hazard detection unit nor the flushing in the IF/ID stage and so we removed that.

Below are listed the modules that were used in our project:

* *Nbit\_reg*: a module representing an n-bit register with load control and reset control signals.
* *n\_bit\_4x1\_mux*: a 4 to 1 multiplexer module with parameter N to change the input size.
* *n\_bit\_2x1\_mux*: a 2 to 1 multiplexer module with parameter N to change the input size.
* *rca*: a ripple carry adder with parameter N to change the input size (the sizes of the numbers to be added).
* *DataMem*: the single memory used in the processor. It takes in MemRead and MemWrite signals (to distinguish between when we want to read and write in the case of using the data memory part of it), the address of the memory that we want to read from/write to, the value that we want to write (in case of MemWrite), funct3 to distinguish between different types of loading/storing instructions (sw/sh/sb, lw/lh/lb/lbu/lhu), and it outputs either the instruction we’re reading or the data we want to load.
* *control\_unit*: this is the processor’s main control unit. It takes in as input the instruction and outputs many signals to be passed to several modules in the processor.
* *reg\_file*: a three-ported register file which holds the 32 registers (x0 → x31). It has parameters N and M representing the size of the registers and the size of the addresses of the registers. It reads in two source registers, a destination register, the data to write to the destination register, the regWrite signal, as well as the clock and reset signals, and it outputs the two read registers.
* *rv32\_ImmGen*: immediate generator which takes in the instruction and outputs the ordered 32-bit immediate.
* *n\_bit\_shift\_left\_1*: shifting module that takes in a number and shifts it by 1 bit to the left. It has parameter N to control the size of the number to be shifted.
* *forwarding\_unit*: detects the need to forward values that have yet to be written to registers instead of using the values read from the register file in the decoding stage. It follows the conditions specified in lab 8.
* *ALU\_control\_unit*: the control unit for the Arithmetic and Logic Unit of the processor. It takes in the 3-bit ALUOp coming from the control unit and the instruction, and it outputs the ALU selection line.
* *prv32\_ALU*: the Arithmetic and Logic Unit of the processor. It takes as input two numbers, the shifting amount, and the ALU selection line, and it outputs a number and several flags (carry flag, zero flag, overflow flag, and sign flag).
* *BranchControlUnit*: a control unit responsible for determining whether a branch should occur or not. It takes in the funct3 of an instruction, the corresponding flags coming out of the ALU, and two signals coming out of the control unit (jump and branch). It outputs a signal, branchCUOut, which indicates whether a branch should be taken or not.

***Difficulties Encountered***

There isn’t really something specific to mention here. Of course, that is not to say that the process was easy. Integrating the pipeline registers with the single cycle module was hard, keeping track of all size differences between the pipelines from the lab and the pipelines accommodating additional signals was hard, merging the two memories into a single one and figuring out when to read from each was hard. There is no particular algorithm that we followed to overcome these difficulties, it was a continuous process of debugging and deeply inspecting the modules and flow of wires.

***Bonus Features***

* We implemented a program generator that generates sample test programs to test the processor. We had several restrictions in the logic of the program so as to avoid creating meaningless programs as much as possible. This includes ensuring that whenever an instruction was written, the source registered in it had already been initialized with a value from a previous instruction. We also ensure that the program does not start with an EBREAK instruction. We also ensure that the program terminates with an EBREAK instruction. Lastly, we provide an option to omit JAL and JALR or storing and loading instructions from the generated program. The program also generates the hexadecimal file containing the instructions in the format readable by Vivado using the readmemh command. The program can be found in the folder with the name “TestGenerator.py”
* We also implemented the following additional instructions from the RV32M instruction set and tested them: mul, mulh, mulhu, mulhsu, and div.

***Testing***

The programs we used for testing are provided below along with screenshots of their outputs. The output was validated either by comparing to a known simulator such as Venus or by tracing.

*Default lab program:*

{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'b000000000100\_00000\_010\_00010\_0000011 ; //lw x2, 4(x0)

{mem[11], mem[10], mem[9], mem[8]} = 32'b000000001000\_00000\_010\_00011\_0000011 ; //lw x3, 8(x0)

{mem[15], mem[14], mem[13], mem[12]} = 32'b0000000\_00010\_00001\_110\_00100\_0110011 ; //or x4, x1, x2

{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000001100100000010001100011; //beq x4, x3, 4

{mem[23], mem[22], mem[21], mem[20]} = 32'b0000000\_00010\_00001\_000\_00011\_0110011 ; //add x3, x1, x2

{mem[27], mem[26], mem[25], mem[24]} = 32'b0000000\_00010\_00011\_000\_00101\_0110011 ; //add x5, x3, x2

{mem[31], mem[30], mem[29], mem[28]} = 32'b0000000\_00101\_00000\_010\_01100\_0100011; //sw x5, 12(x0)

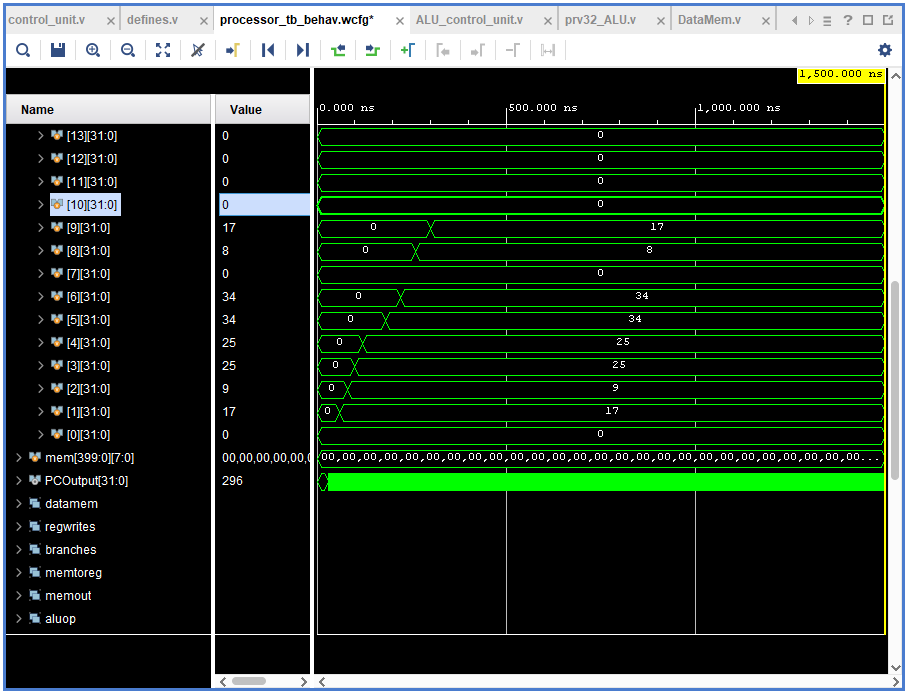
{mem[35], mem[34], mem[33], mem[32]} = 32'b000000001100\_00000\_010\_00110\_0000011 ; //lw x6, 12(x0)

{mem[39], mem[38], mem[37], mem[36]} = 32'b0000000\_00001\_00110\_111\_00111\_0110011 ; //and x7, x6, x1

{mem[43], mem[42], mem[41], mem[40]} = 32'b0100000\_00010\_00001\_000\_01000\_0110011 ; //sub x8, x1, x2

{mem[47], mem[46], mem[45], mem[44]} = 32'b0000000\_00010\_00001\_000\_00000\_0110011 ; //add x0, x1, x2

{mem[51], mem[50], mem[49], mem[48]} = 32'b0000000\_00001\_00000\_000\_01001\_0110011 ; //add 9, x0, x1

Output:**

*Program that tests branching and jumps:*

{mem[3], mem[2], mem[1], mem[0]} = 32'h00500093; // addi x1, x1, 5

{mem[7], mem[6], mem[5], mem[4]} = 32'hffc00113; // addi x2, x2, -4

{mem[11], mem[10], mem[9], mem[8]} = 32'h00108463; // beq x1, x1, 8

{mem[15], mem[14], mem[13], mem[12]} = 32'h00a00193; // addi x3, x0, 10

{mem[19], mem[18], mem[17], mem[16]} = 32'h00b00213; // addi x4 x0 11

{mem[23], mem[22], mem[21], mem[20]} = 32'h00209463; // bne x1 x2

{mem[27], mem[26], mem[25], mem[24]} = 32'h00a00193; // addi x3 x0 10

{mem[31], mem[30], mem[29], mem[28]} = 32'h00c00213; // addi x4 x0 12

{mem[35], mem[34], mem[33], mem[32]} = 32'h0020d463; // bge x1 x2 8

{mem[39], mem[38], mem[37], mem[36]} = 32'h00a00193; // addi x3 x0 10

{mem[43], mem[42], mem[41], mem[40]} = 32'h00d00213; // addi x4 x0 13

{mem[47], mem[46], mem[45], mem[44]} = 32'h0020e463; // bltu x1 x2 8

{mem[51], mem[50], mem[49], mem[48]} = 32'h00a00193; // addi x3 x0 10

{mem[55], mem[54], mem[53], mem[52]} = 32'h00e00213; // addi x4 x0 14

{mem[59], mem[58], mem[57], mem[56]} = 32'h00114463; // blt x2, x1, 8

{mem[63], mem[62], mem[61], mem[60]} = 32'h00a00193; // addi x3, x0, 10

{mem[67], mem[66], mem[65], mem[64]} = 32'h00f00213; // addi x4, x0, 15

{mem[71], mem[70], mem[69], mem[68]} = 32'h00117463; // bgeu x2, x1, 8

{mem[75], mem[74], mem[73], mem[72]} = 32'h00a00193; // addi x3, x0, 10

{mem[79], mem[78], mem[77], mem[76]} = 32'h01000213; // addi x4, x0, 16

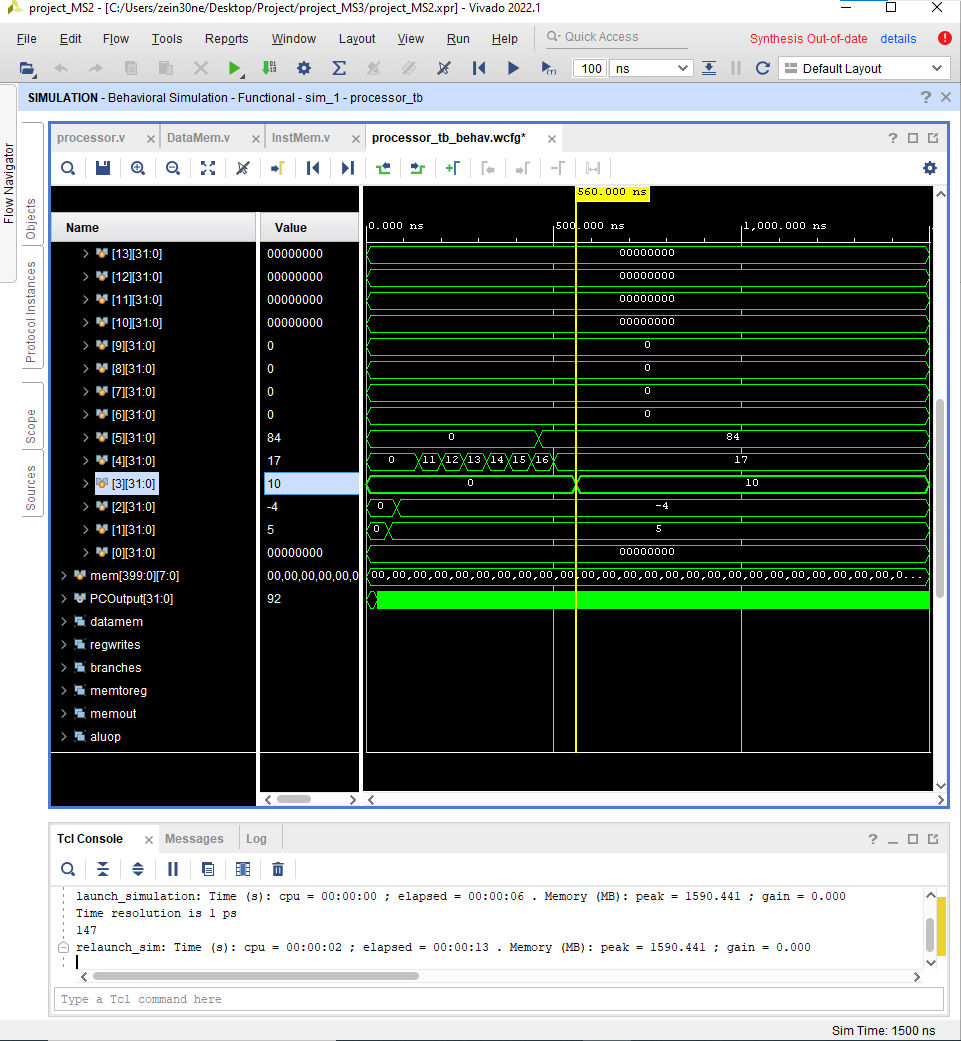
{mem[83], mem[82], mem[81], mem[80]} = 32'h008002ef; // jal x5, 8

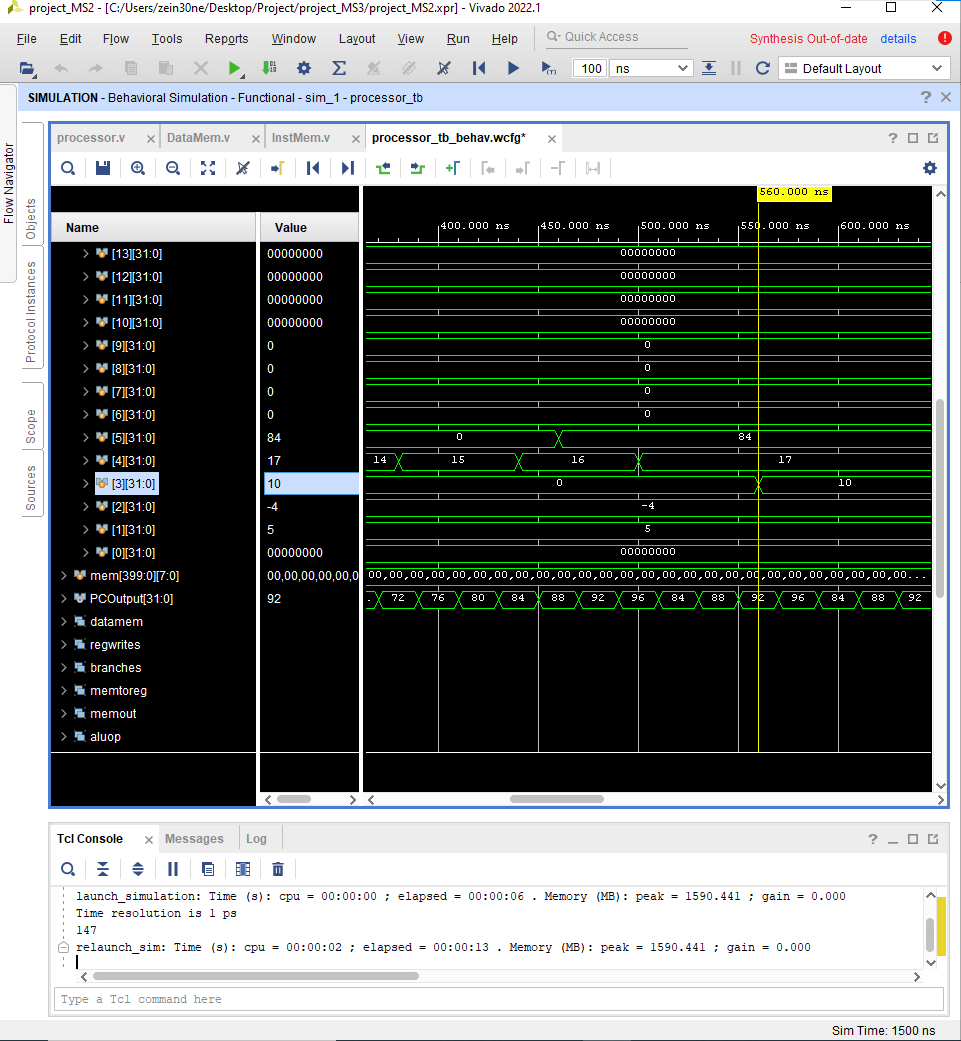
{mem[87], mem[86], mem[85], mem[84]} = 32'h00a00193; // addi x3, x0, 10

{mem[91], mem[90], mem[89], mem[88]} = 32'h01100213; // addi x4, x0, 17

{mem[95], mem[94], mem[93], mem[92]} = 32'h00028067; // jalr x0, 0(x5)

Output:





*Program that tests loading and storing and Ecall:*

{mem[3], mem[2], mem[1], mem[0]} =32'b00000001001110001000000010110111; // lui x1, 5000

{mem[7], mem[6], mem[5], mem[4]} =32'b00000000000100001000000010010011; // addi x1, x1, 1

{mem[11], mem[10], mem[9], mem[8]} =32'b00000000000100000010001000100011; // sw x1, 4(x0)

{mem[15], mem[14], mem[13], mem[12]}=32'b00000000000100000001010000100011; // sh x1, 8(x0)

{mem[19], mem[18], mem[17], mem[16]}=32'b00000000000100000000010100100011; // sb x1, 10(x0)

{mem[23], mem[22], mem[21], mem[20]}=32'b00000000010000000010111110000011; // lw x31, 4(x0)

{mem[27], mem[26], mem[25], mem[24]}=32'b00000000010000000001111100000011; // lh x30, 4(x0)

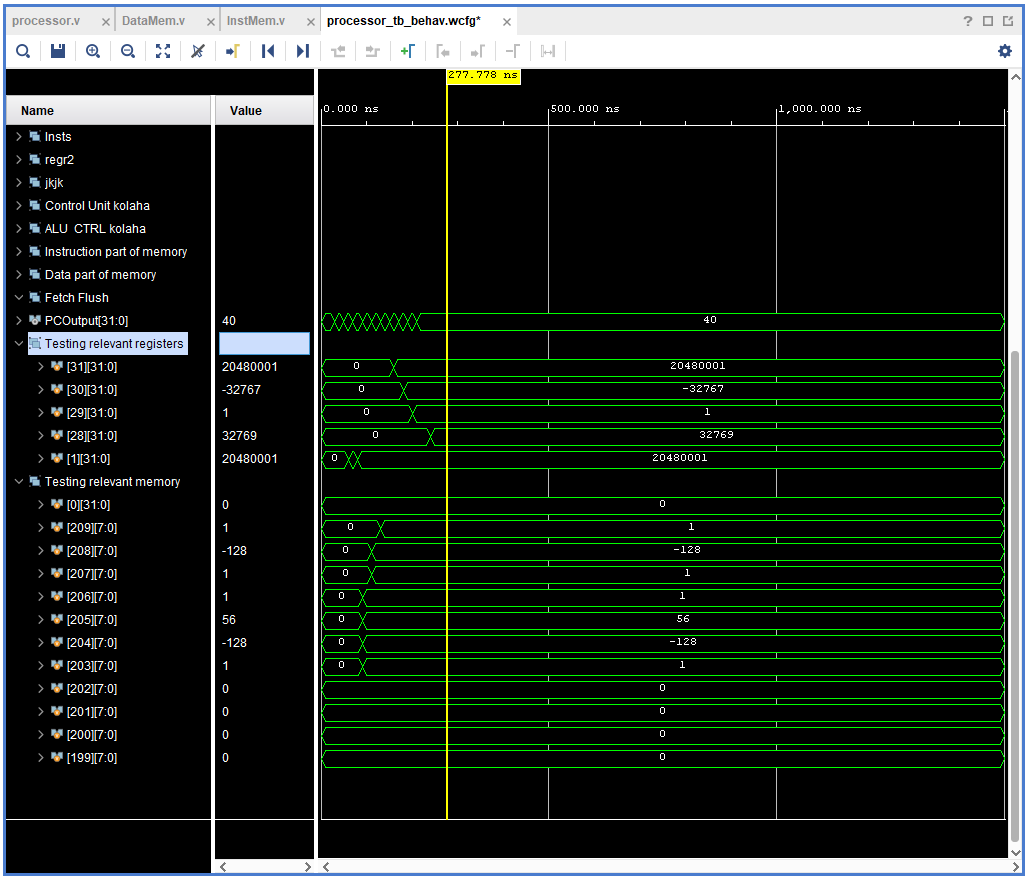
{mem[31], mem[30], mem[29], mem[28]}=32'b00000000010000000000111010000011; // lb x29, 4(x0)

{mem[35], mem[34], mem[33], mem[32]}=32'b00000000000000000000000001110011; // ECALL

{mem[39], mem[38], mem[37], mem[36]}=32'b00000000010000000101111000000011; // lhu x28, 4(x0)

{mem[43], mem[42], mem[41], mem[40]}=32'b00000000000100000000000001110011; // EBREAK

Output:



*Program that tests SLT and SLTU:*

{mem[3], mem[2], mem[1], mem[0]} =32'b11111111000000000000101100010011; // addi x22, x0, -16

{mem[7], mem[6], mem[5], mem[4]} =32'b00000000111000000000101110010011; // addi x23, x0, 14

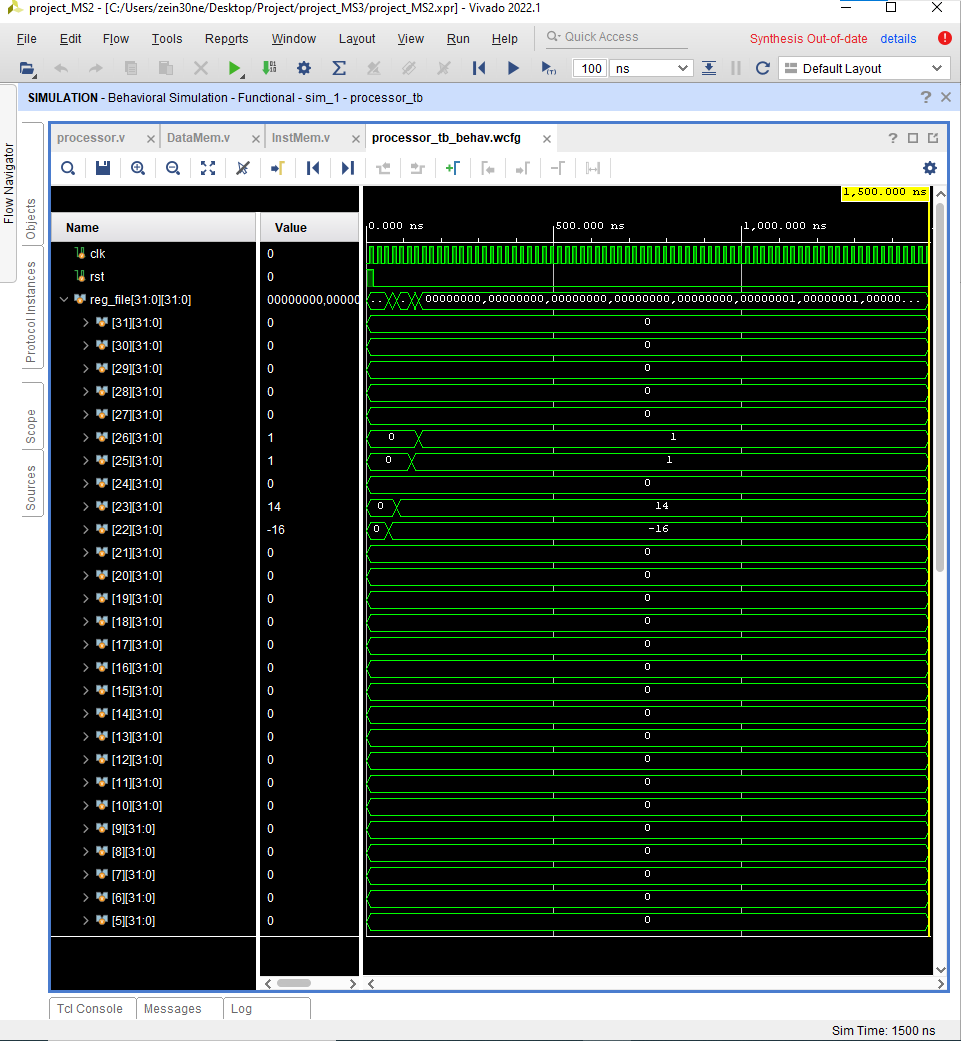
{mem[11], mem[10], mem[9], mem[8]} =32'b00000001011010111010110000110011; // slt x24, x23, x22 (should be 0)

{mem[15], mem[14], mem[13], mem[12]}=32'b00000001011110110010110010110011; // slt x25, x22, x23 (should be 1)

{mem[19], mem[18], mem[17], mem[16]}=32'b00000001011010111011110100110011; // sltu x26, x23, x22 (should be 1)

{mem[23], mem[22], mem[21], mem[20]}=32'b00000001011110110011110110110011; // sltu x27, x22, x23 (should be 0)

Output:



*Program that tests I-format and a few R-format instructions (includes data dependency):*

{mem[3], mem[2], mem[1], mem[0]} =32'b11111111000000000000101100010011; // addi x22, x0, -16

{mem[7], mem[6], mem[5], mem[4]} =32'b00000000000110110101101110010011; // srli x23, x22, 1

{mem[11], mem[10], mem[9], mem[8]} =32'b01000000000110110101110000010011; // srai x24, x22, 1

{mem[15], mem[14], mem[13], mem[12]}=32'b00000000000100000000110010010011; // addi x25, x0, 1

{mem[19], mem[18], mem[17], mem[16]}=32'b01000001100110110101110100110011; // sra x26, x22, x25

{mem[23], mem[22], mem[21], mem[20]}=32'b00000000101010110100110110010011; // xori x27, x22, 10

{mem[27], mem[26], mem[25], mem[24]}=32'b00000001100110110100111000110011; // xor x28, x22, x25

{mem[31], mem[30], mem[29], mem[28]}=32'b00000001100110110101111010110011; // srl x29, x22, x25

{mem[35], mem[34], mem[33], mem[32]}=32'b00000001001110001000000010110111; // lui x1, 5000

{mem[39], mem[38], mem[37], mem[36]}=32'b00010010110000000010001000010011; // slti x4, x0, 300

{mem[43], mem[42], mem[41], mem[40]}=32'b11111111101100000010001010010011; // slti x5, x0, -5

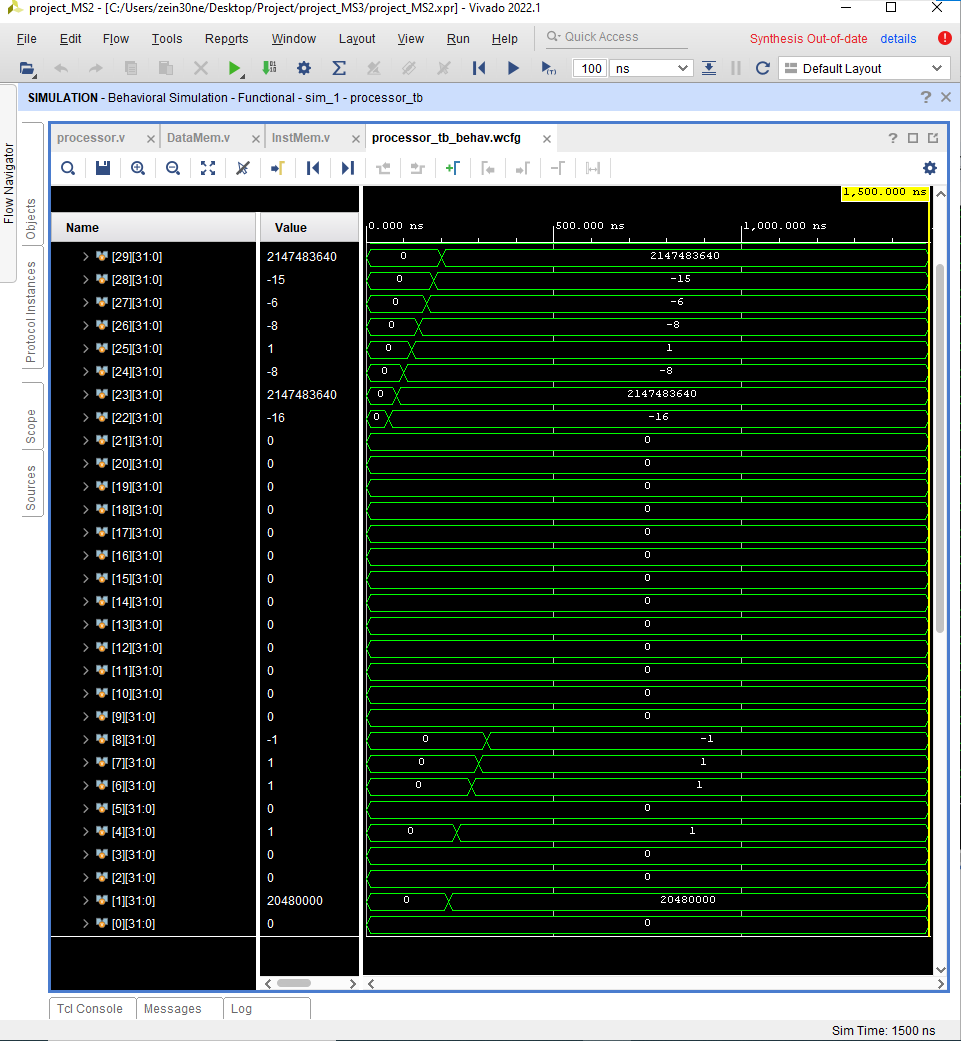
{mem[47], mem[46], mem[45], mem[44]}=32'b11111111101100000011001100010011; // sltiu x6, x0, -5

{mem[51], mem[50], mem[49], mem[48]}=32'b00010010110000000011001110010011; // sltiu x7, x0, 300

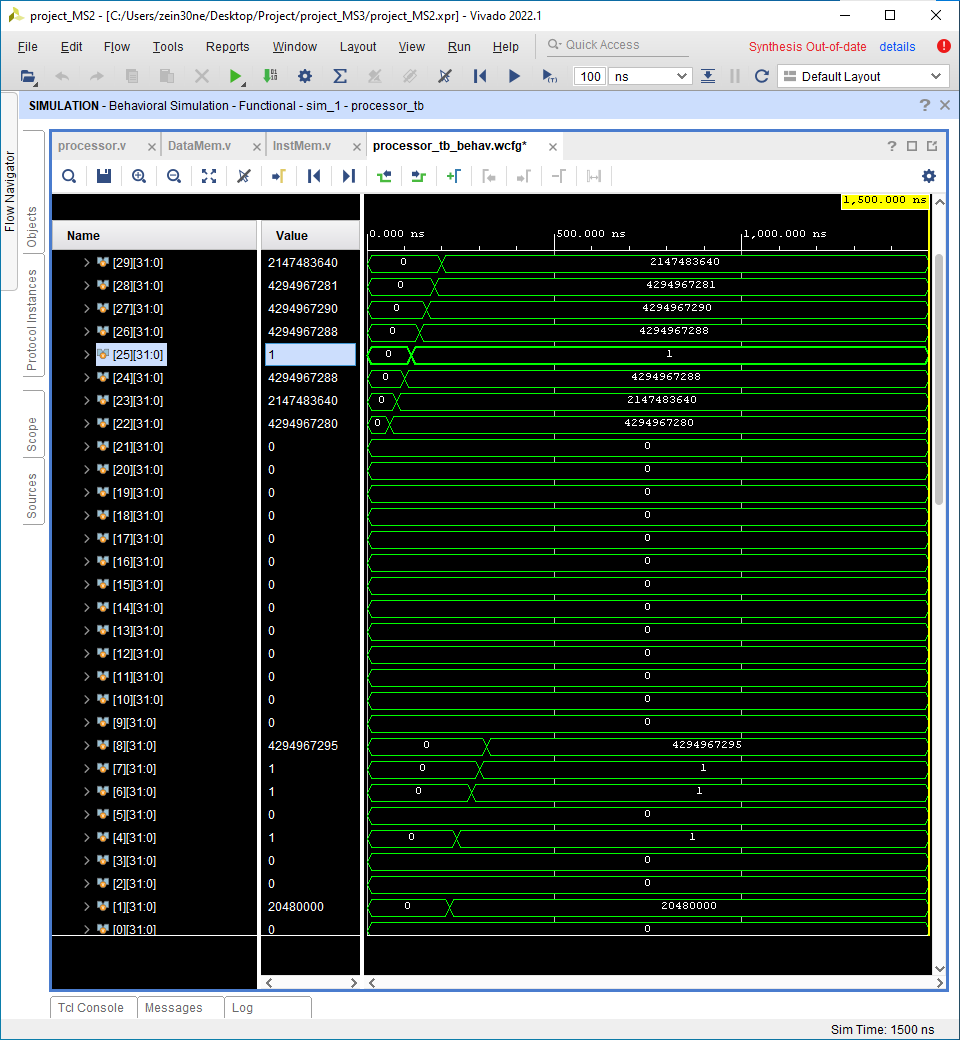
{mem[55], mem[54], mem[53], mem[52]}=32'b11111111111100000110010000010011; // ori x8, x0, -1

{mem[59], mem[58], mem[57], mem[56]}=32'b00000000000100001111010010010011; // andi x9, x1, 1

Output with signed radix:



Output with unsigned radix:



*Program that tests remaining R-format instructions:*

{mem[3], mem[2], mem[1], mem[0]} = 32'hff600093; // addi x1, x0, -10

{mem[7], mem[6], mem[5], mem[4]} = 32'h00f00113; // addi x2, x0, 15

{mem[11], mem[10], mem[9], mem[8]} = 32'h00100f93; // addi x31, x0, 1

{mem[15], mem[14], mem[13], mem[12]} = 32'h002081b3; // add x3, x1, x2

{mem[19], mem[18], mem[17], mem[16]} = 32'h40110233; // sub x4, x2, x1

{mem[23], mem[22], mem[21], mem[20]} = 32'h402082b3; // sub x5, x1, x2

{mem[27], mem[26], mem[25], mem[24]} = 32'h0ff0000f; // fence

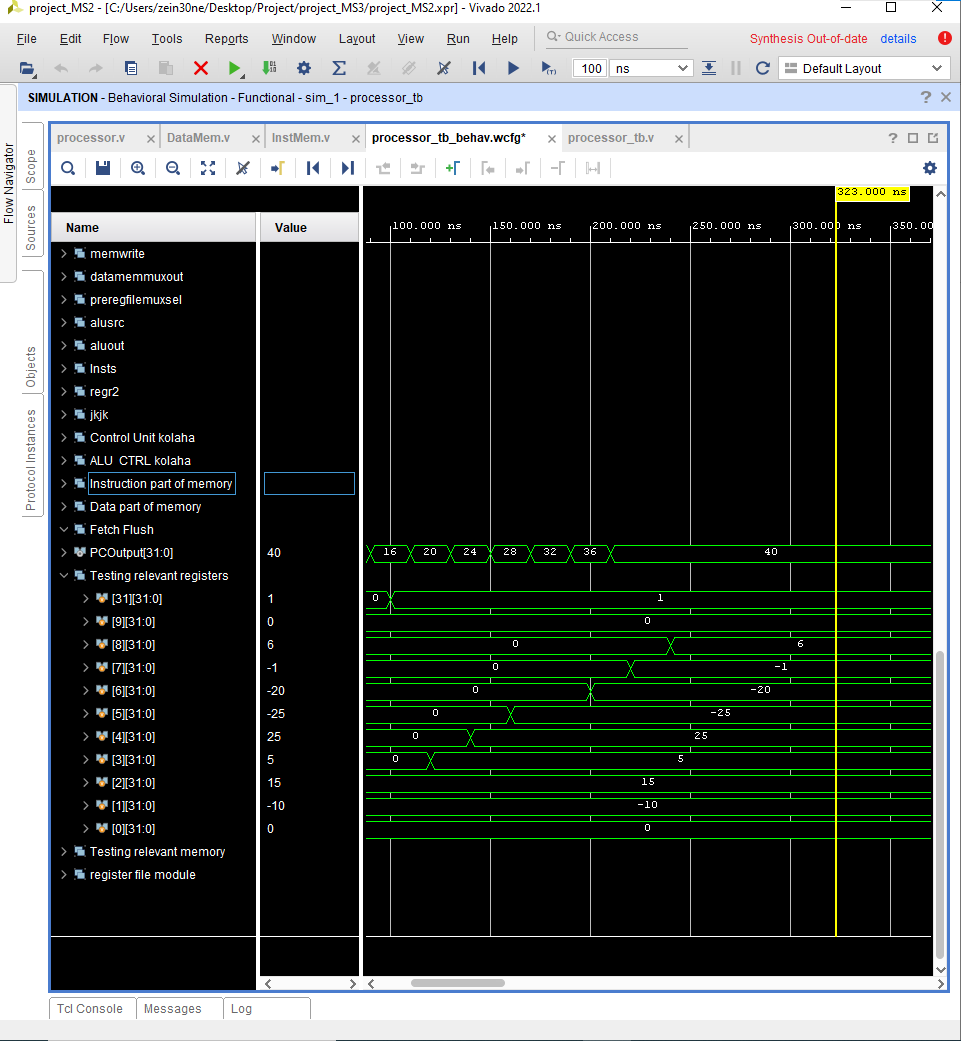
{mem[31], mem[30], mem[29], mem[28]} = 32'h01f09333; // sll x6, x1, x31

{mem[35], mem[34], mem[33], mem[32]} = 32'h001163b3; // or x7 x2, x1

{mem[39], mem[38], mem[37], mem[36]} = 32'h00117433; // and x8, x2, x1

{mem[43], mem[42], mem[41], mem[40]} =32'h00100073; // ebreak

{mem[47], mem[46], mem[45], mem[44]} =32'hfff00493; // addi x9, x0, -1

Output:

*Program that tests multiplication instructions and div instruction:*

{mem[3], mem[2], mem[1], mem[0]} = 32'hfb800093; //addi x1, x0, -72

{mem[7], mem[6], mem[5], mem[4]} = 32'h00c00113; //addi x2, x0, 12

{mem[11], mem[10], mem[9], mem[8]} = 32'h021101b3; //mul x3, x2, x1

{mem[15], mem[14], mem[13], mem[12]} = 32'h0220c233; //div x4, x1, x2

{mem[19], mem[18], mem[17], mem[16]} = 32'h022092b3; //mulh x5, x1, x2

{mem[23], mem[22], mem[21], mem[20]} = 32'h0220a333; //mulhsu x6, x1, x2

{mem[27], mem[26], mem[25], mem[24]} = 32'h0220b3b3; //mulhu x7, x1, x2

Output:

