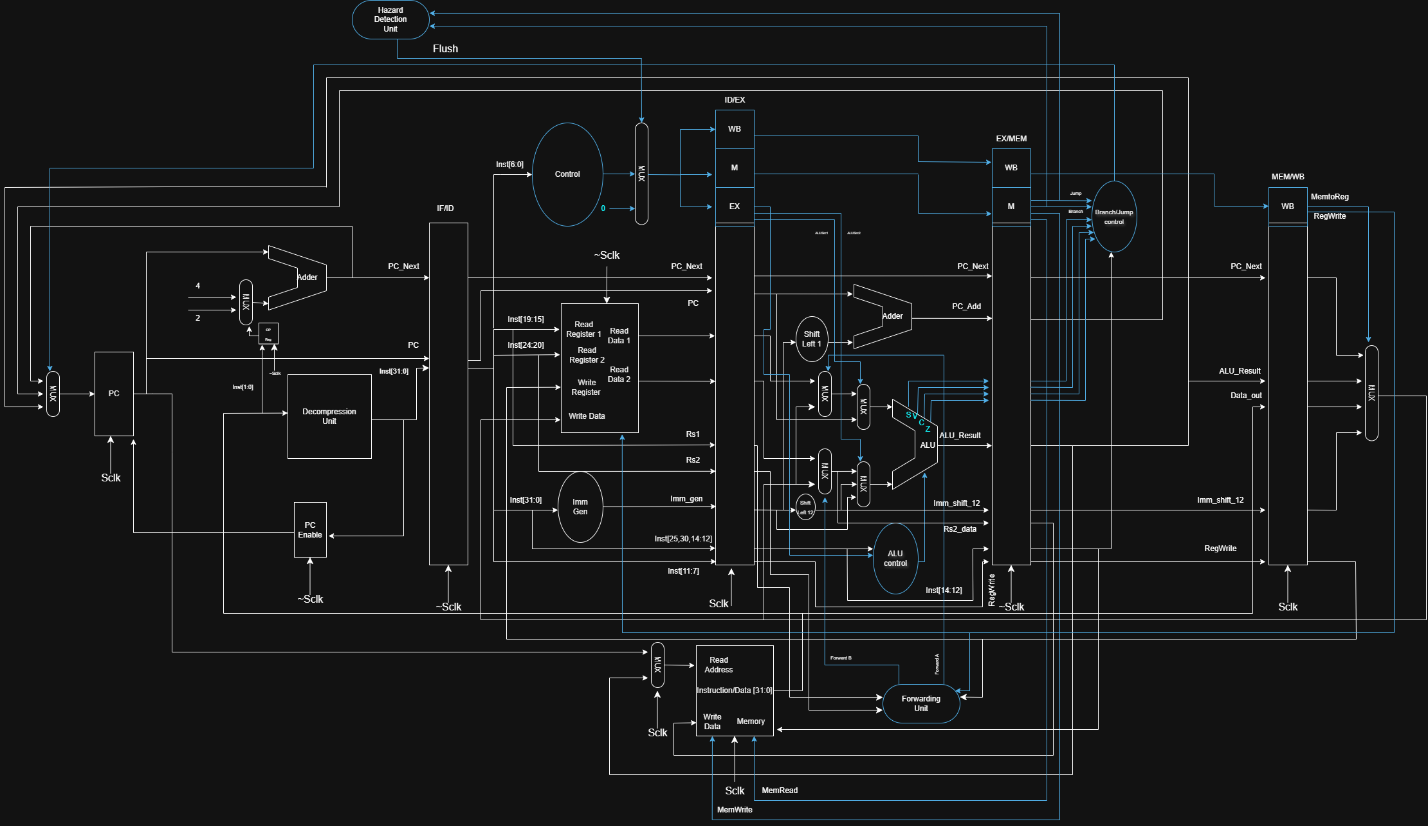
**Pipelined CPU Report**

**Seif ElAnsary:900221511**

**Nadine ElGarem**

**Block Diagram**



This diagram demonstrates the 3-stage pipeline that the professor discussed in class. It supports all RiscV32ICM instructions except div and rem, for there exists a problem in with signed division that I could not solve. The pipeline supports forwarding from the MEM/WB stage to the ID/EX stage and has Branch/jump hazard flushing at the ID stage. The instruction per 2 cycles removes almost all hazards, making the CPU quite reliable. The tests below demonstrate its ability.

**Test Waveforms:**

This website <https://luplab.gitlab.io/rvcodecjs/#q=c.addi4spn&abi=false&isa=AUTO> was used for testing the instructions and generating the testcases. Many of their default C & M instructions were used as my test instructions.

***C Extension***

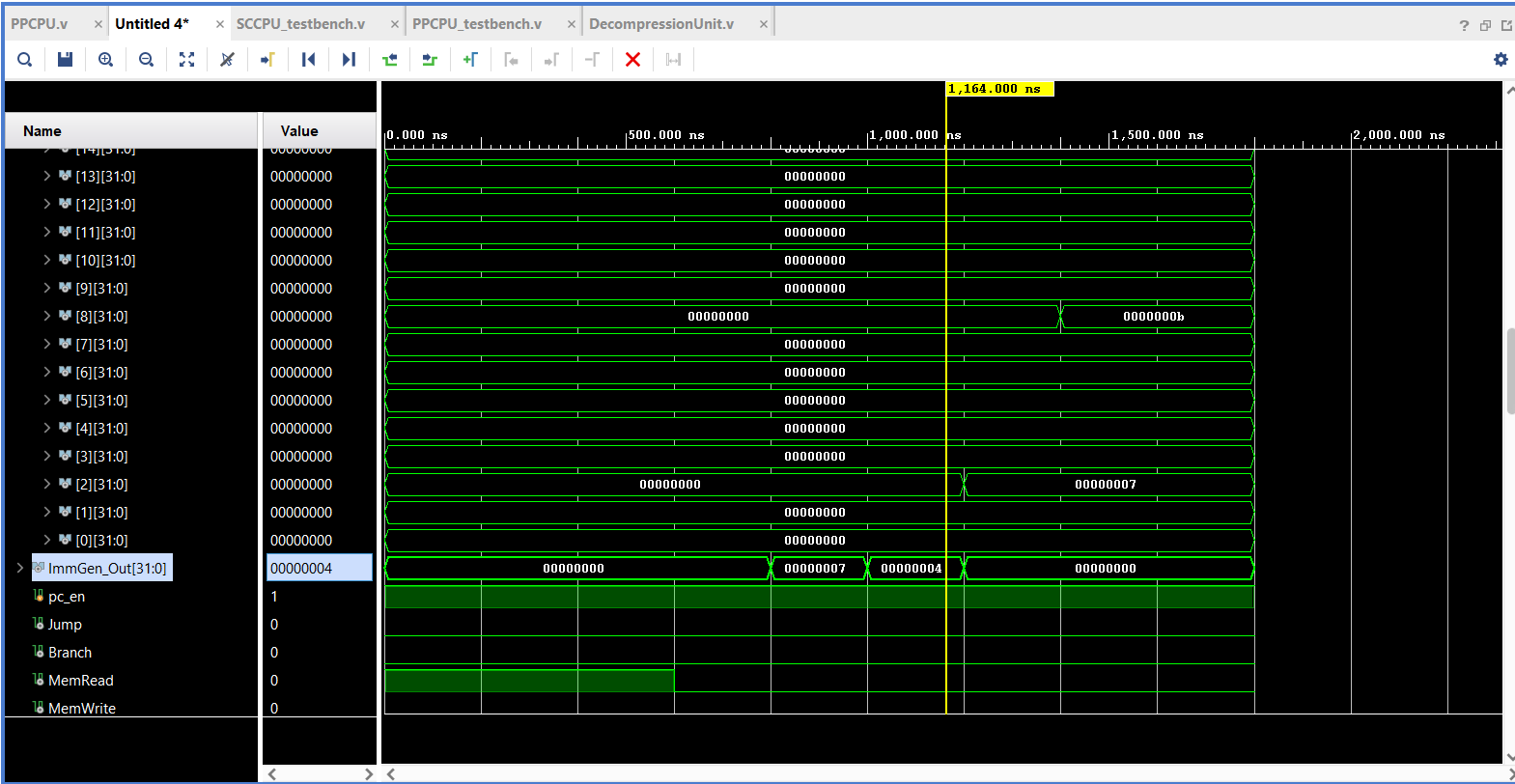
**C.ADDI4SPN**

**c.addi4spn x8, 4 (Expands to addi x8, x2, 4)**

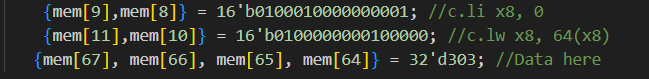
**Binary:** **0000000001000000**

**Hexadecimal:** **0x0040**

**x2 = 7**

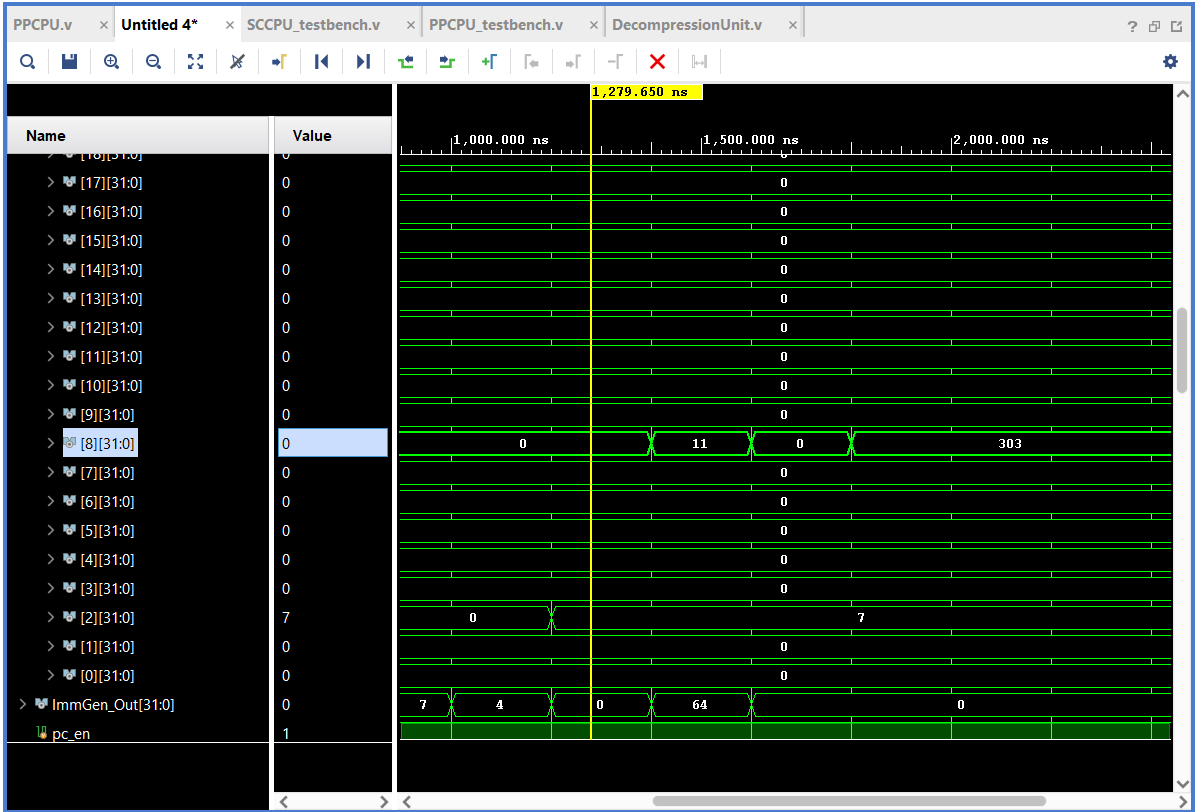
****

**C.LW**

**c.lw x8, 64(x8) (Expands to lw x8, 64(x8))**

**Binary:** **0100000000100000**

**Hexadecimal:** **0x4020**

****

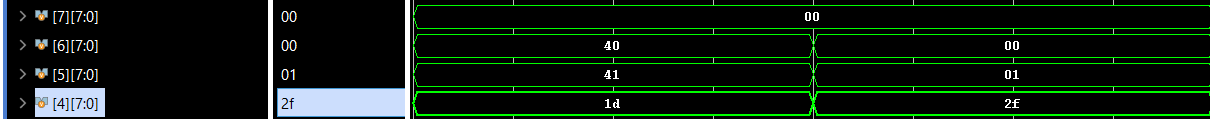
**C.SW**

**c.sw x8, 4(x9) (Expands to sw x8, 4(x9))**

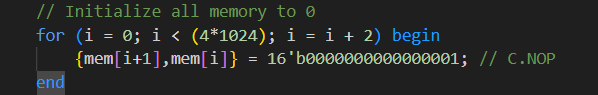
**Binary:** **1100000011000000**

**Hexadecimal: 0xc0c0**

**x9 = 0 x8=303**

**  
  
0x00012f = 303**

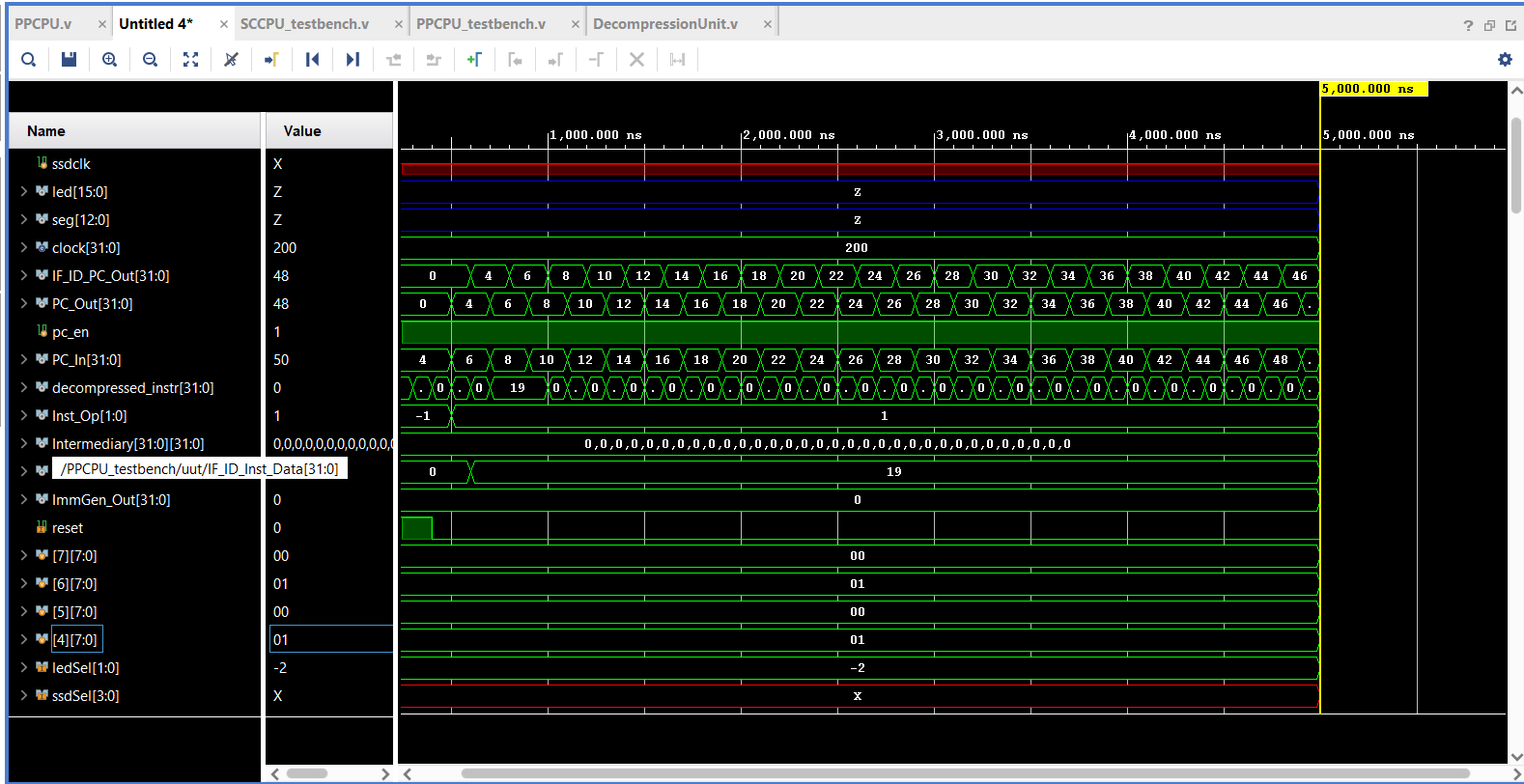
**C.NOP**

**c.nop (Expands to addi, x0, x0, 0)**

**Binary: 0000000000000001**

**Hexadecimal: 0x0001**

**Entire memory is C.NOPs**



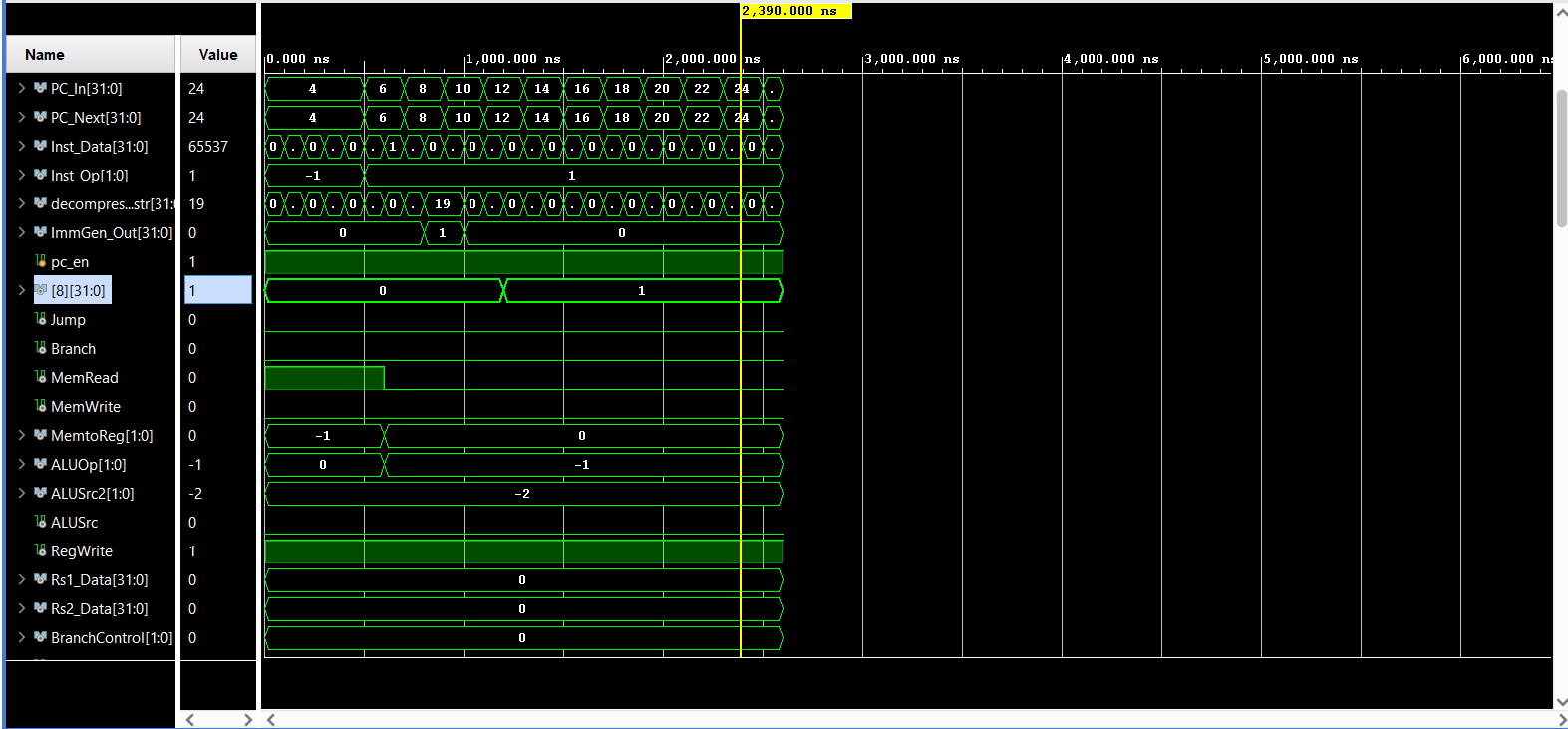
**C.ADDI**

**c.addi x8, 1 (addi, x8, x8, 1)**

**Binary:** **0000010000000101**

**Hexadecimal: 0x0405**

**x8=0**

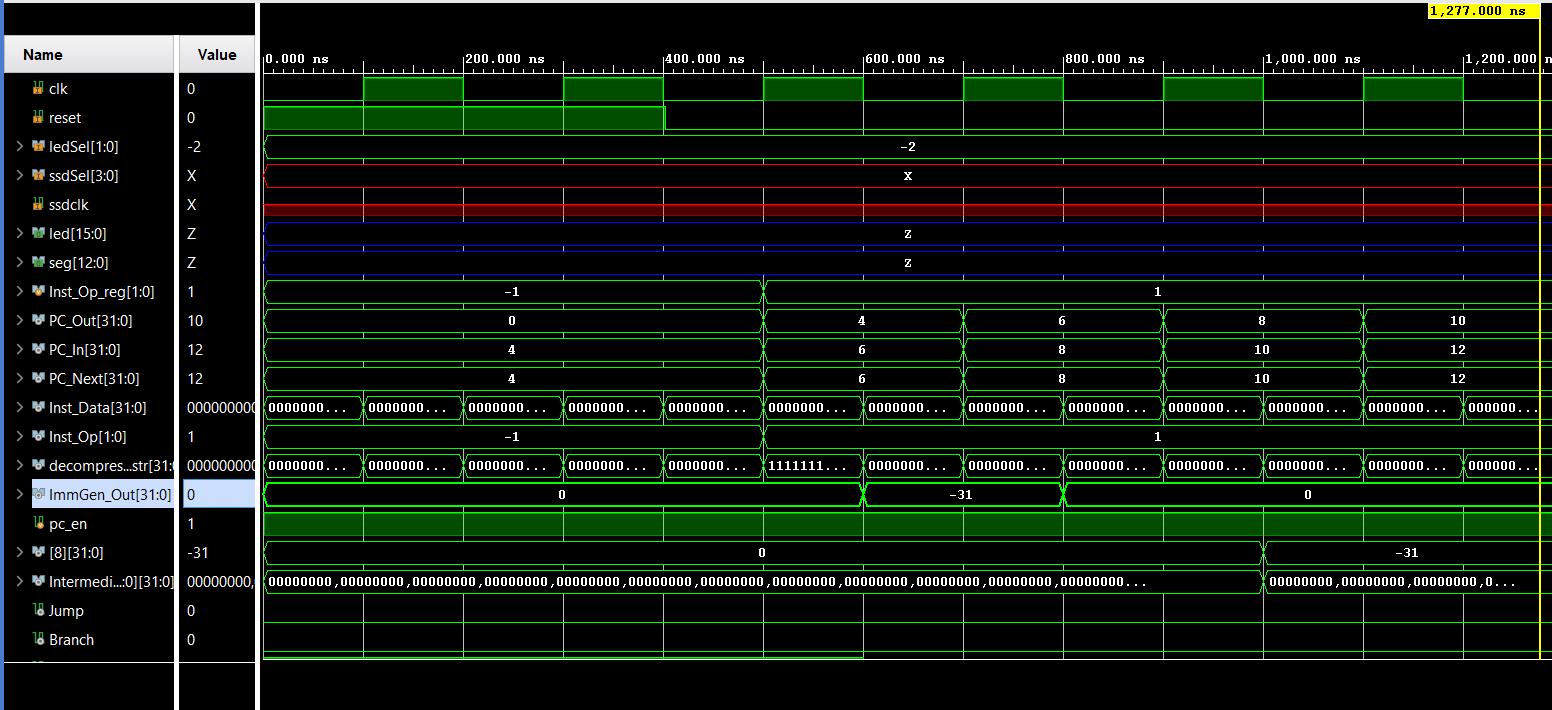
****

**c.addi x8, -31**

**Binary: 0001010000000101**

**Hexadecimal: 0x1405**

**x8=0**

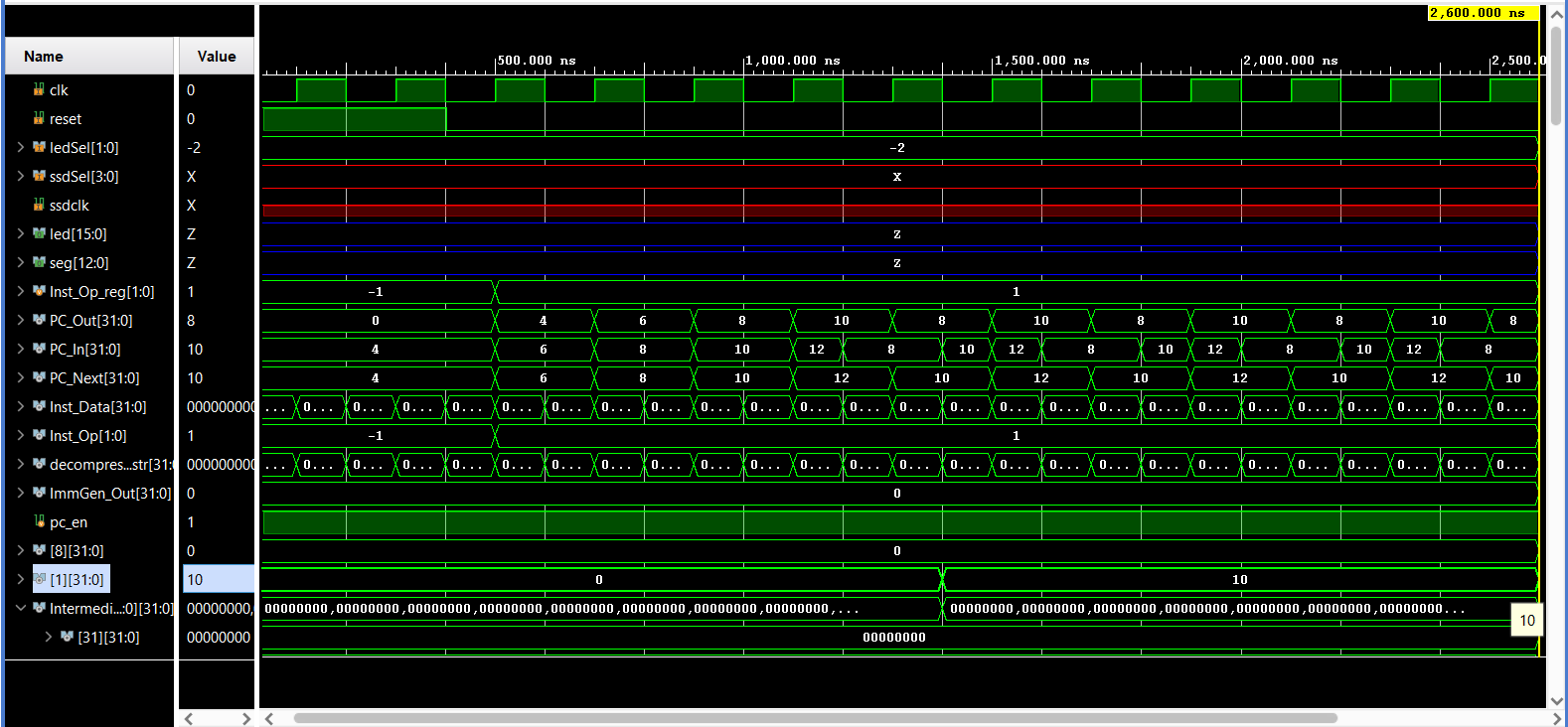
****

**C.JAL**

**c.jal 0 (Expands to jal x1, 0)**

**Binary: 0010000000000001**

**Hexadecimal: 0x2001**

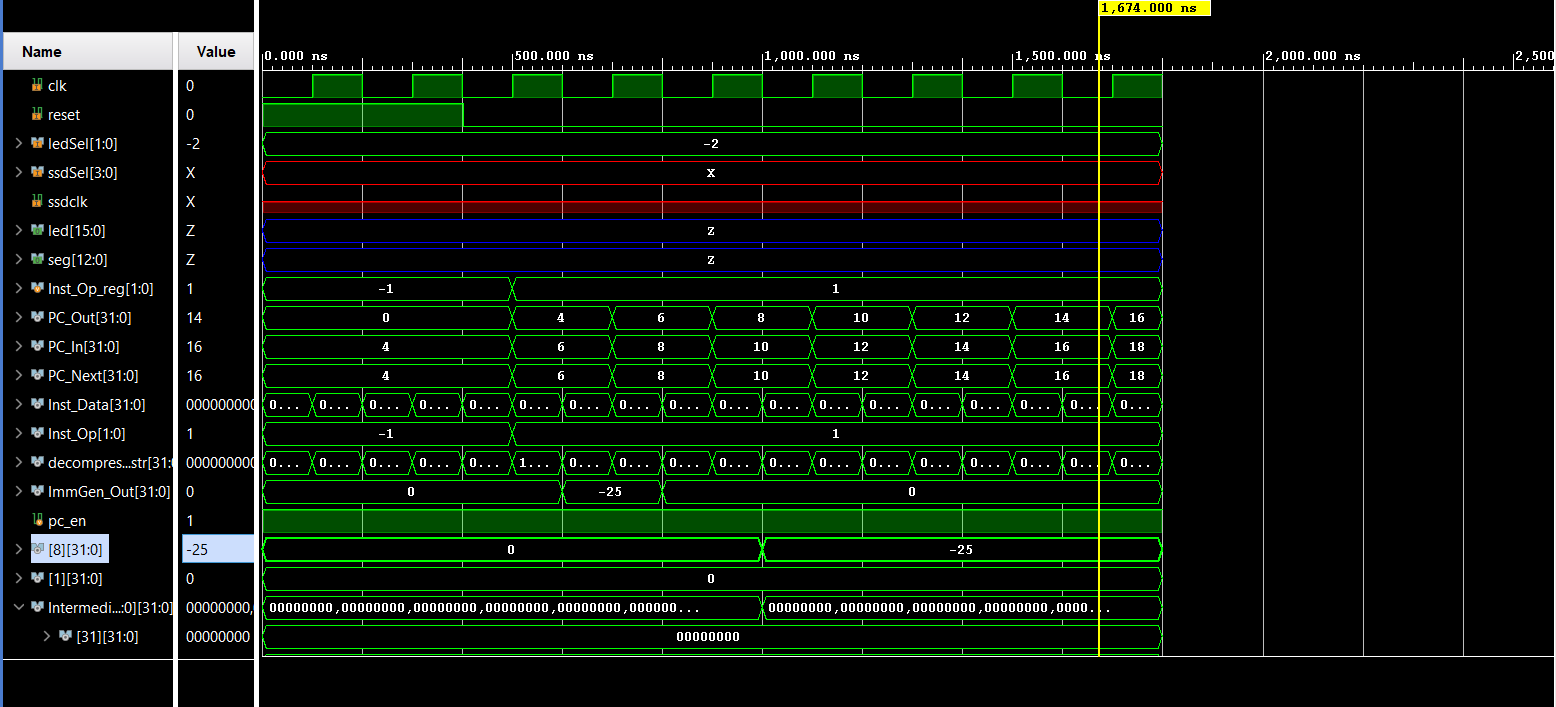
****

**C.LI**

**c.li x8, -25 (Expands to addi x8, x0, -25)**

**Binary: 0101010000011101**

**Hex: 0x541d**

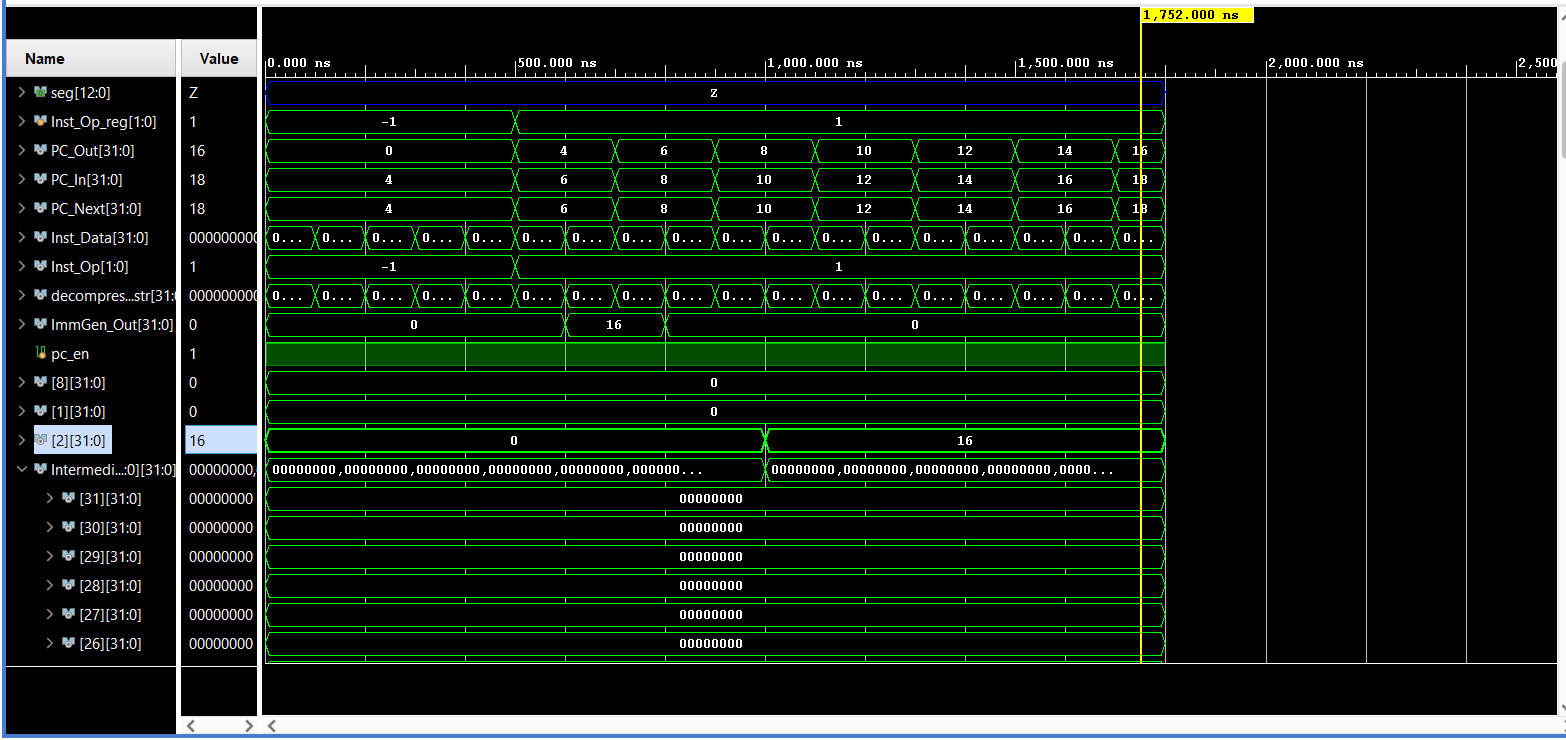
****

**C.ADDI16SP**

**c.addi16sp 16 (Expands to addi, x2, x2, 16)**

**Binary: 0110000101000001**

**Hexadecimal: 0x6141**

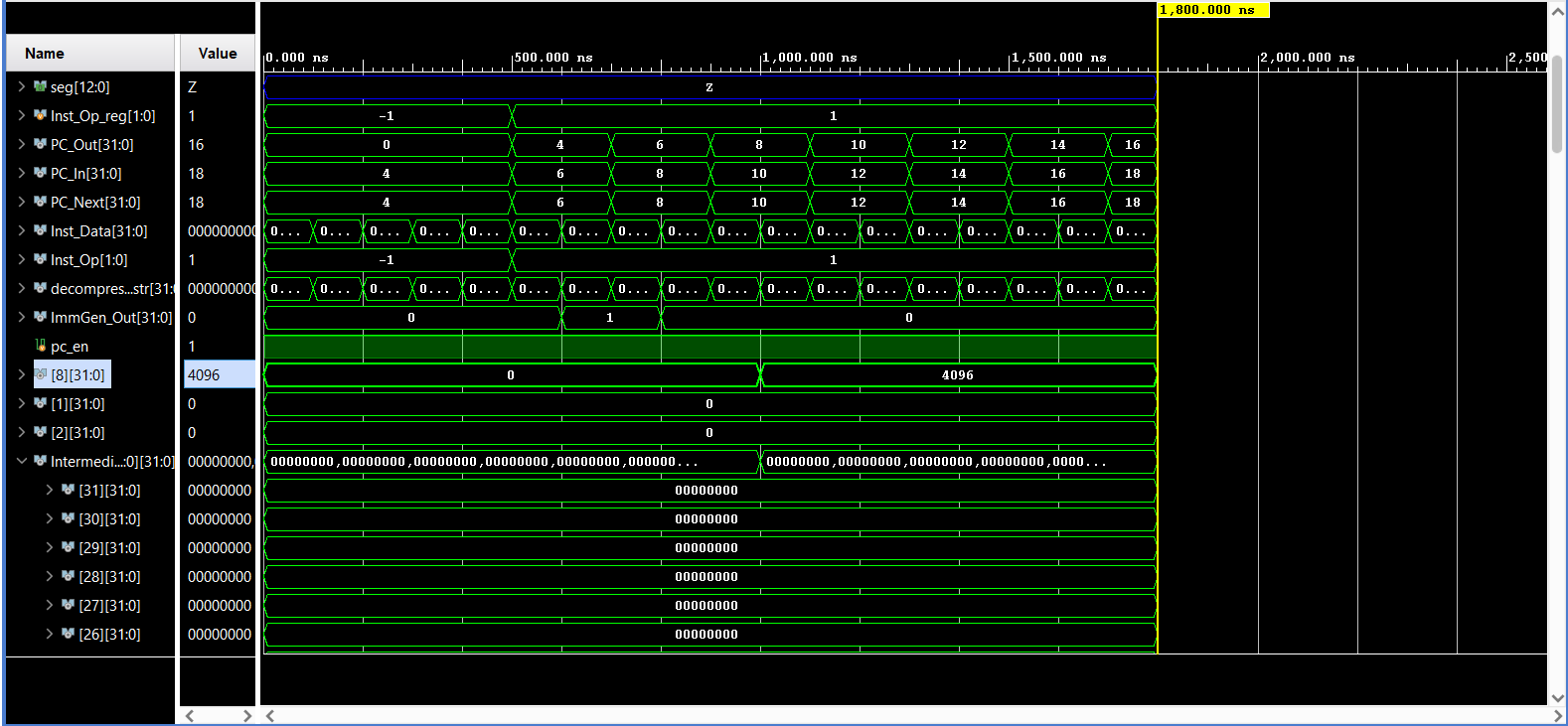
****

**C.LUI**

**c.lui x8, 1 (Expands to lui x8, 1)**

**Binary: 0110010000000101**

**Hexadecimal: 0x6405**

****

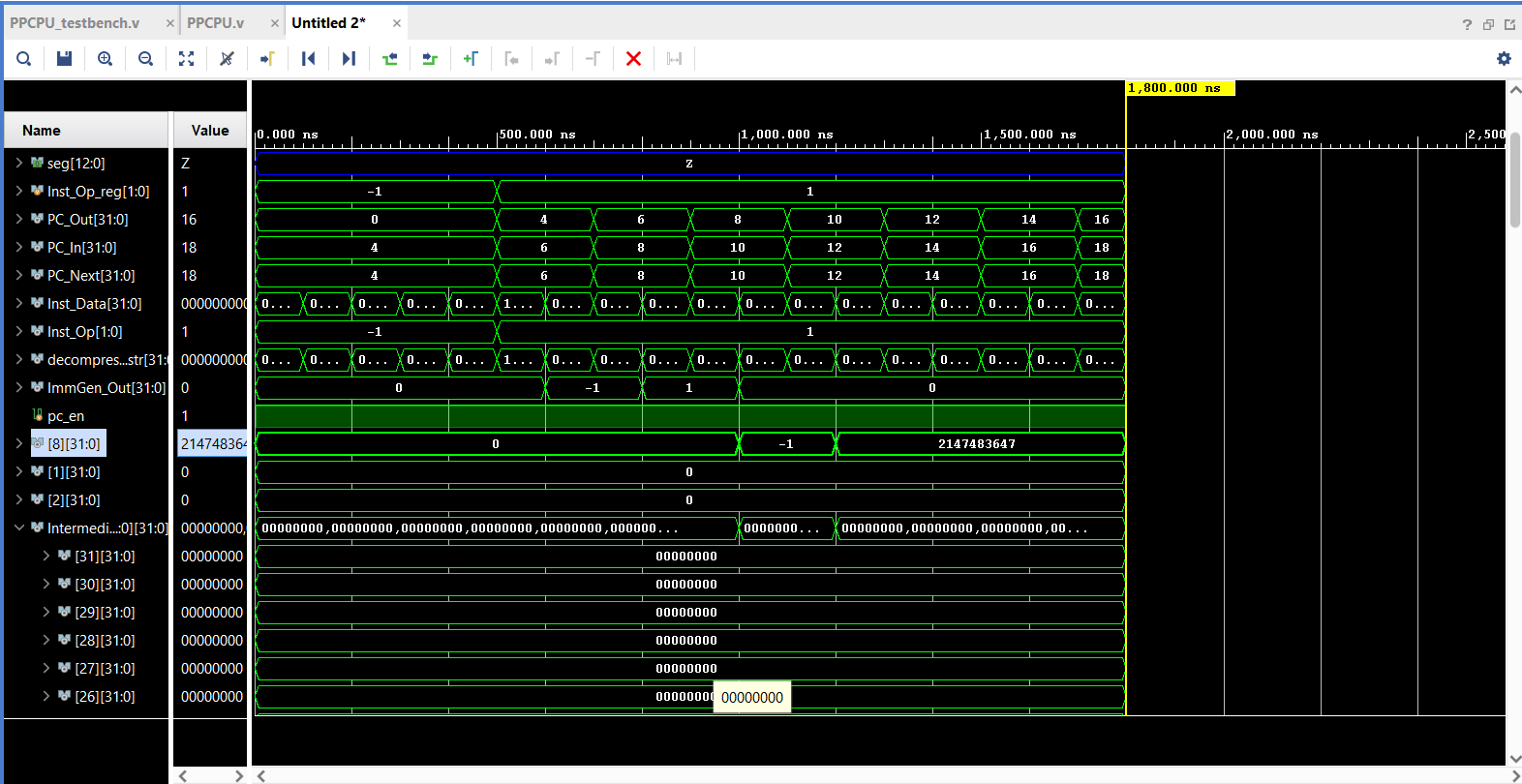
**C.SRLI**

**c.srli x8, 1 (expands to srli x8, x8, 1)**

**Binary: 1000000000000101**

**Hexadecimal: 0x8005**

**x8=-1 x8 now should be SIGNED INT MAX which is 2^30-1**

****

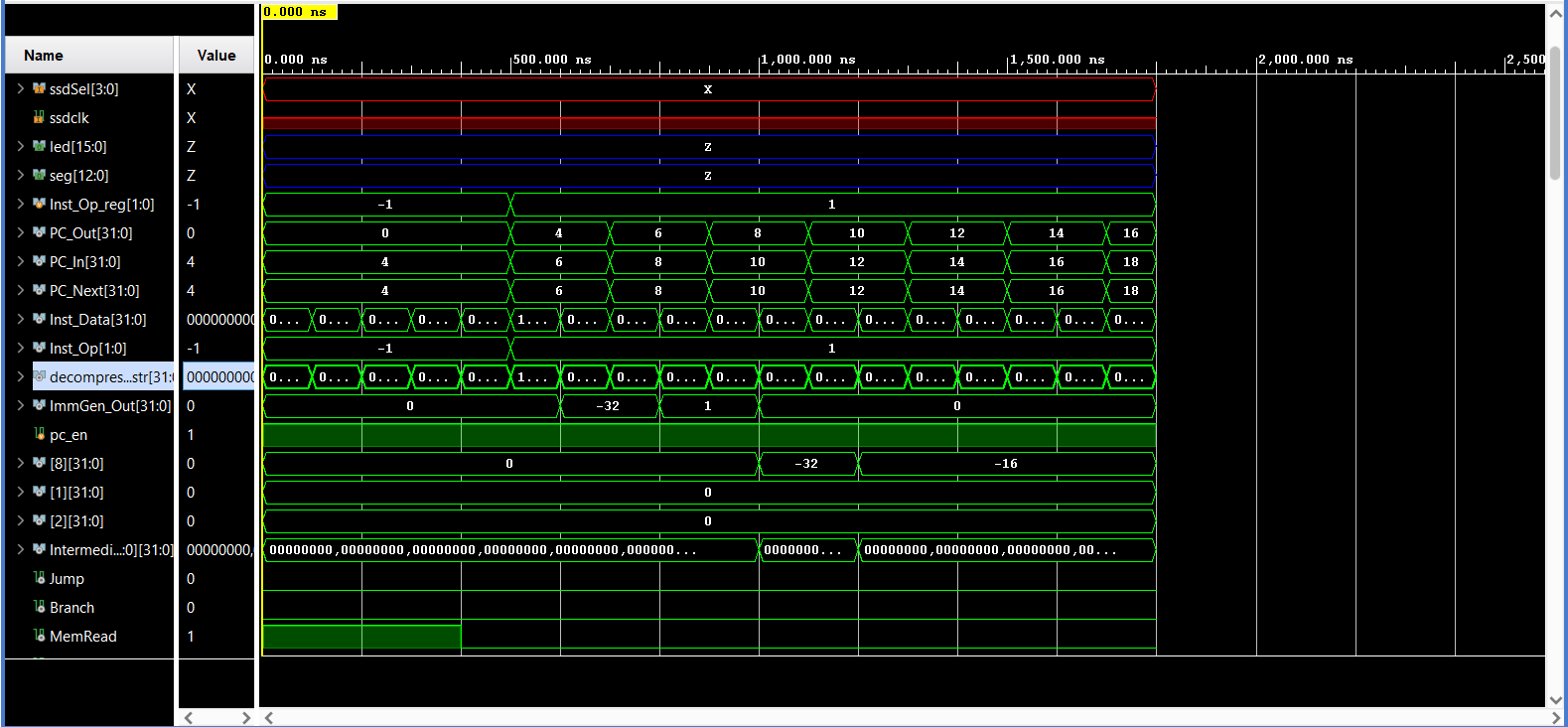
**C.SRAI**

**c.srai x8, 1 (expands to srai, x8, x8, 1)**

**Binary: 1000010000000101**

**Hexadecimal: 0x8405**

**x8=-32 Will be -16**

****

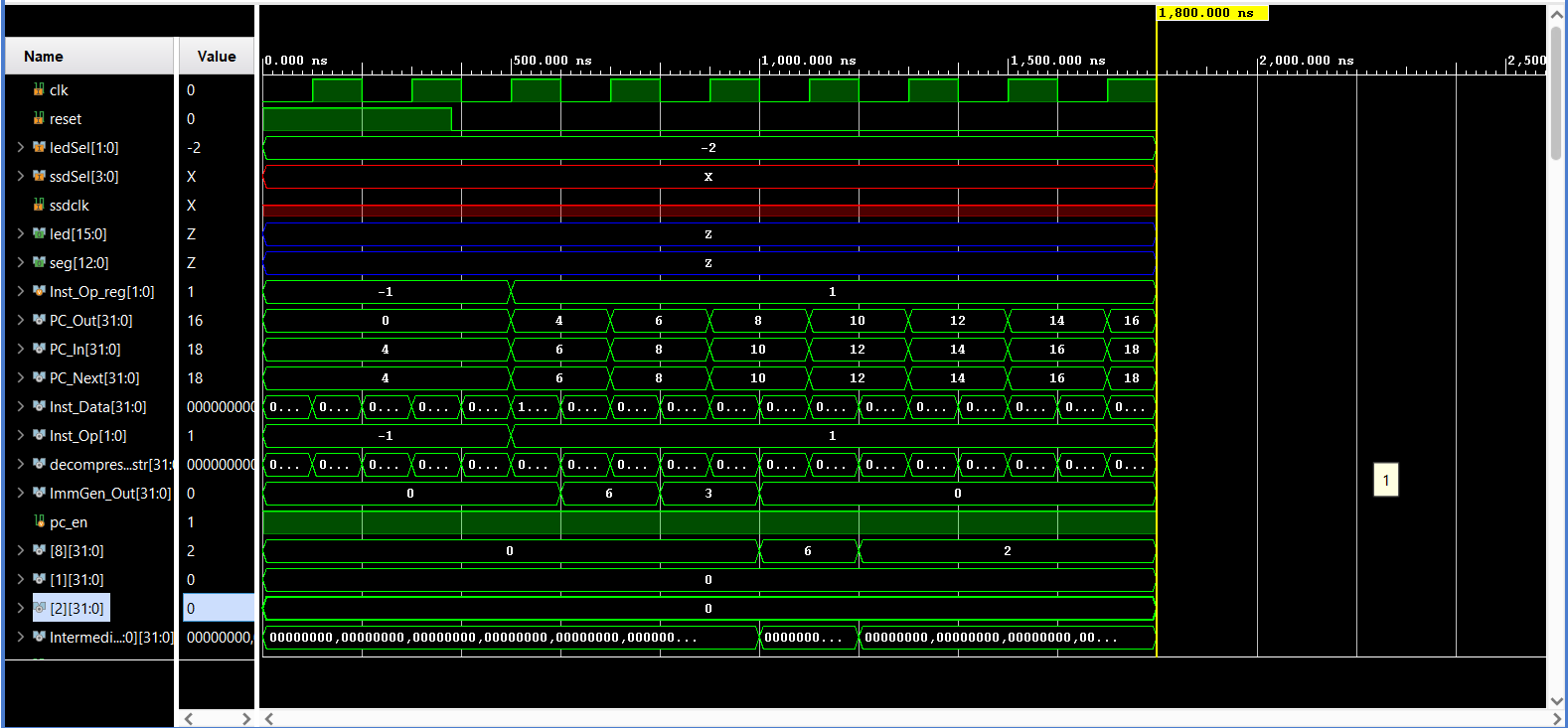
**C.ANDI**

**c.andi x8, 3 (Expands to andi x8, x8, 3)**

**Binary: 1000100000001101**

**Hexadecimal: 0x880d**

**x8 = 6 6 & 3 =2**

****

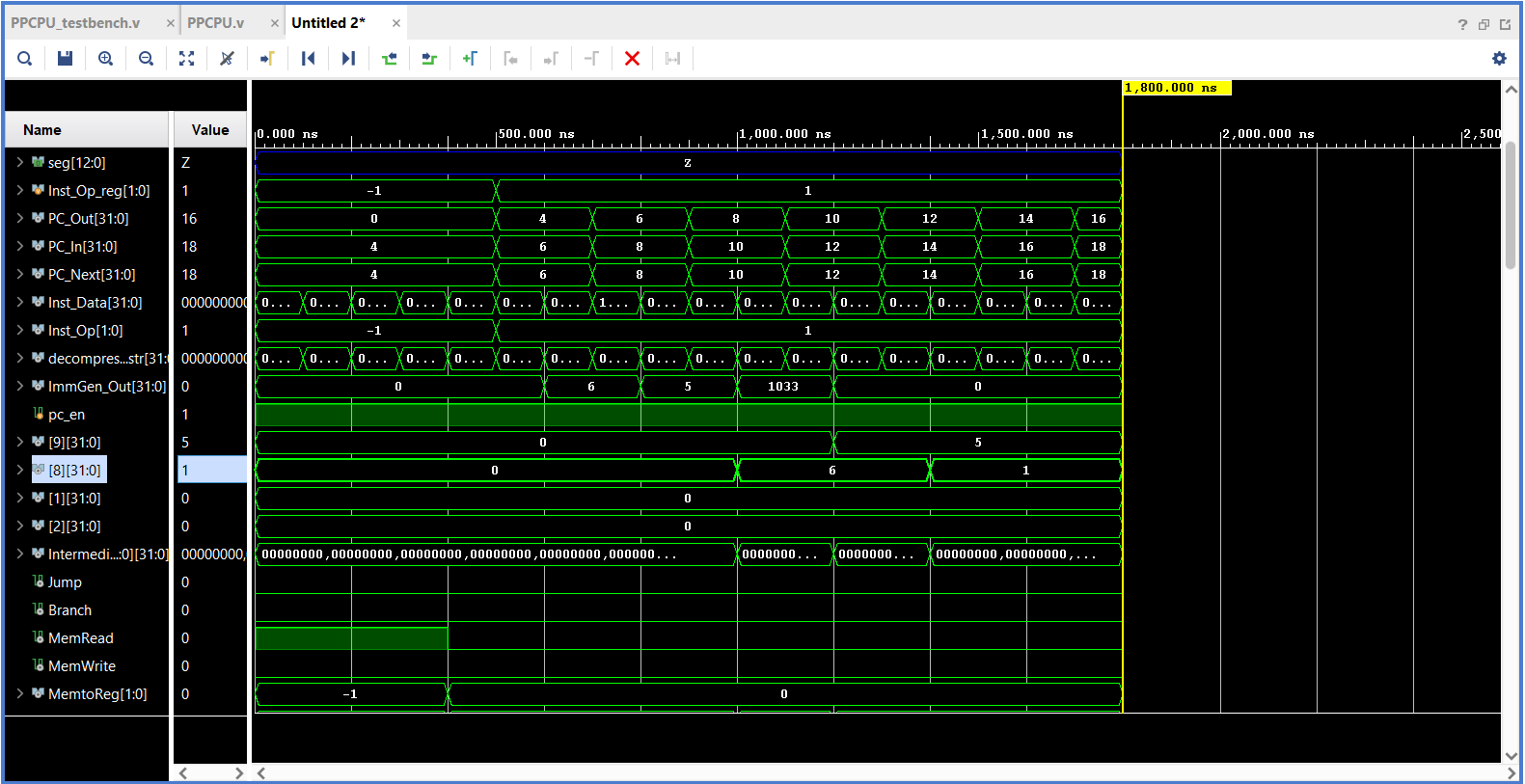
**C.SUB**

**c.sub x8, x9 (Expands to sub x8, x8, x9)**

**Binary: 1000110000000101**

**Hexadecimal: 0x8c05**

**x8 = 6 x9=5**

****

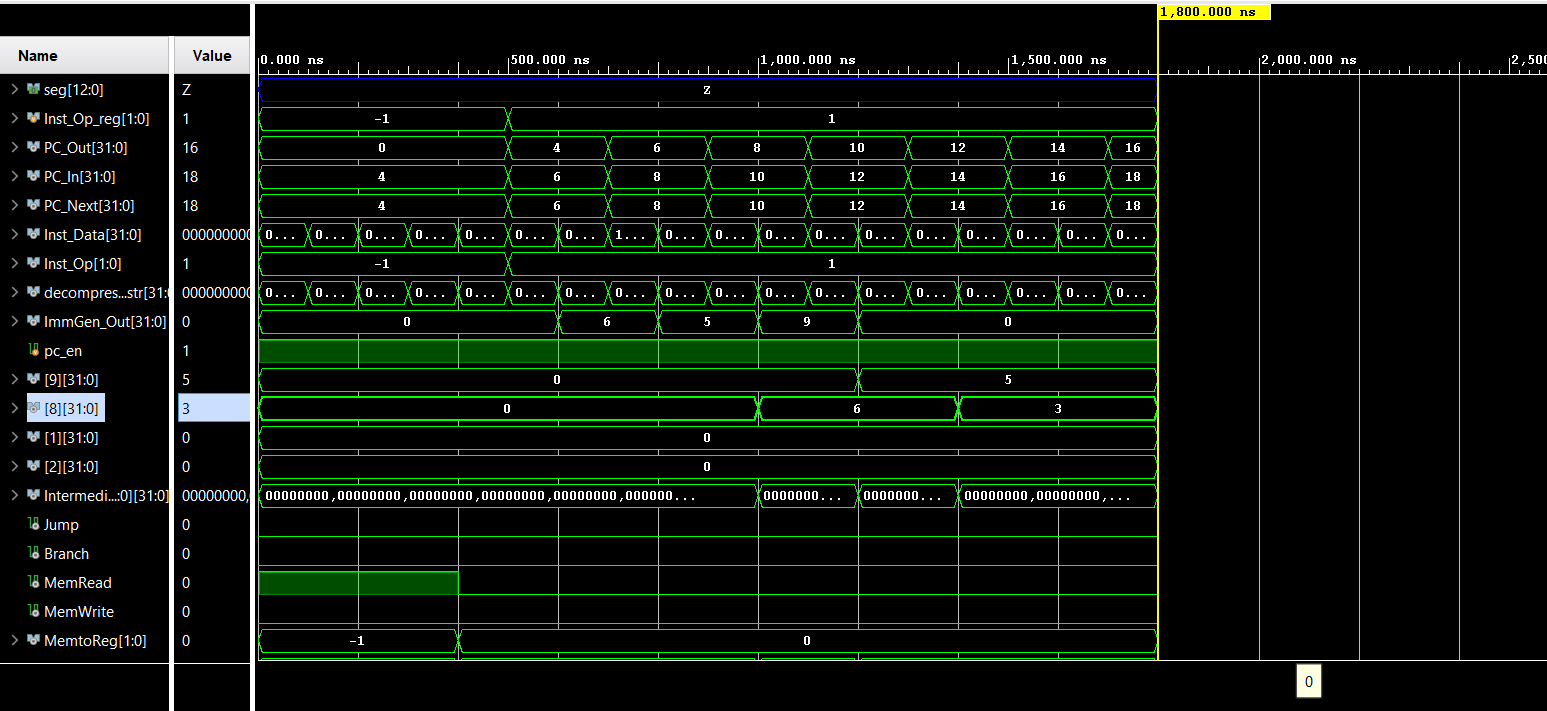
**C.XOR**

**c.xor x8, x9 (Expands to xor x8, x8, x9)**

**Binary: 1000110000100101**

**Hexadecimal: 0x8c25**

**x8 = 6 x9=5 6 ^ 5 = 3**

****

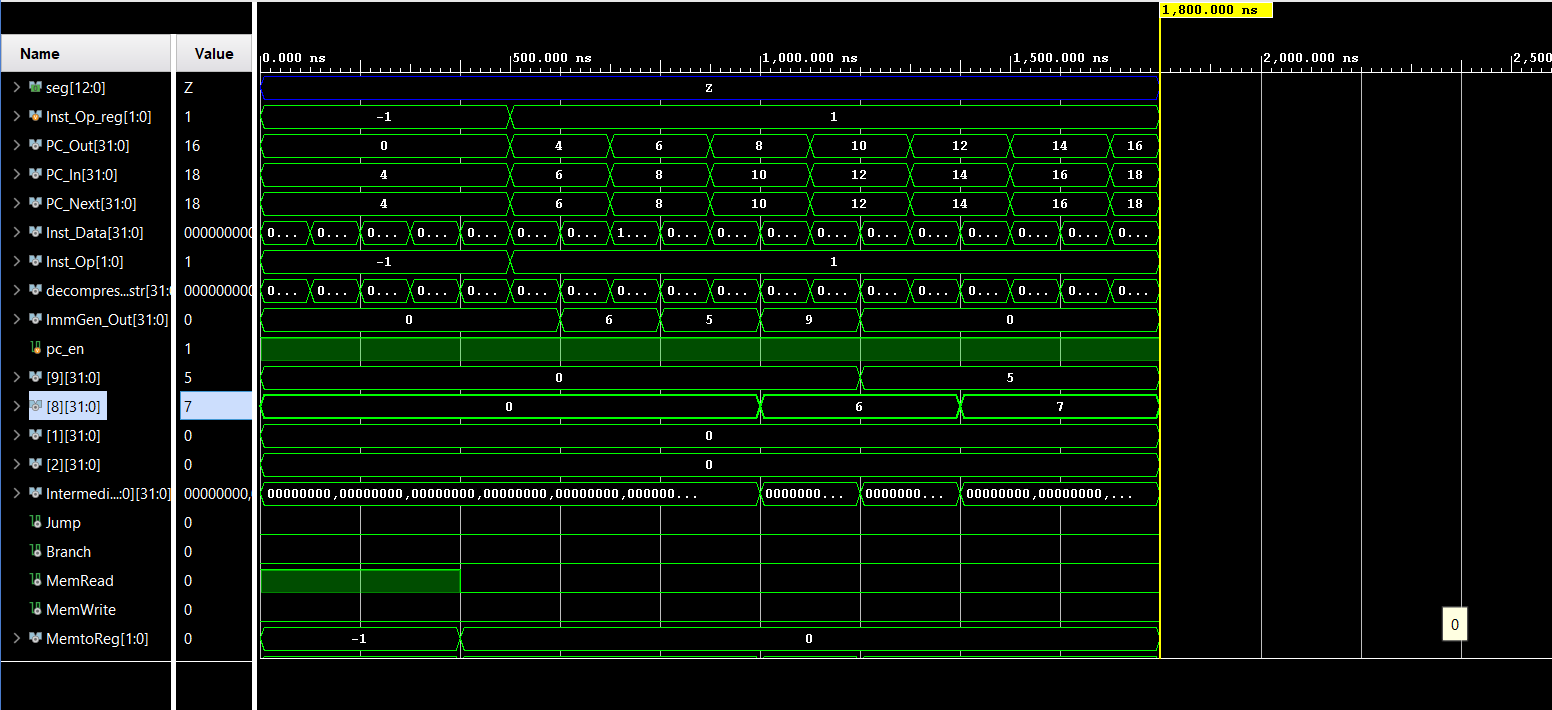
**C.OR**

**c.or x8, x9 (Expands to or x8, x8, x9)**

**Binary: 1000110001000101**

**Hexadecimal: 0x8c45**

**x8 = 6 x9=5 6 | 5 = 7**

****

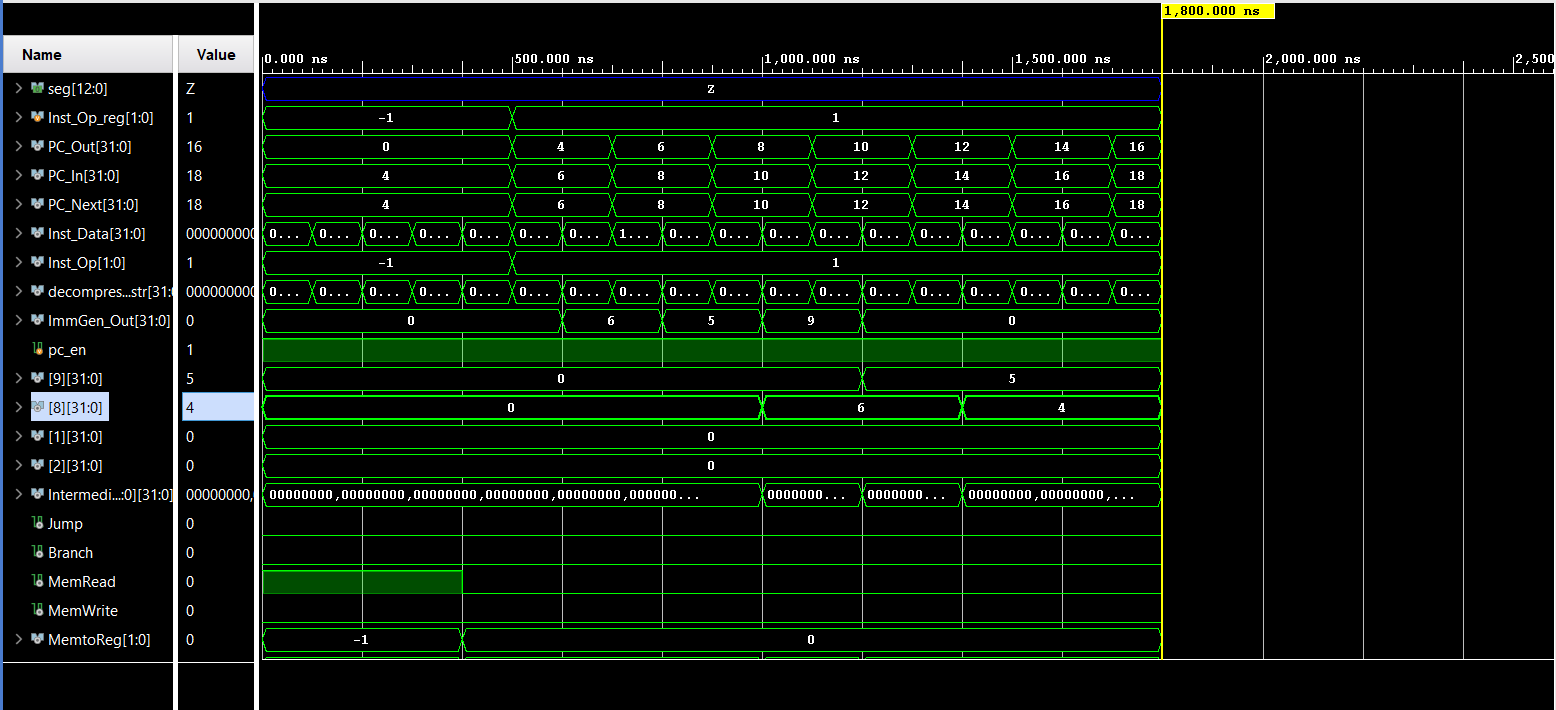
**C.AND**

**c.and x8, x9 (Expands to and x8, x8, x9)**

**Binary: 1000110001100101**

**Hexadecimal: 0x8c65**

**x8 = 6 x9=5 6 & 5 = 4**

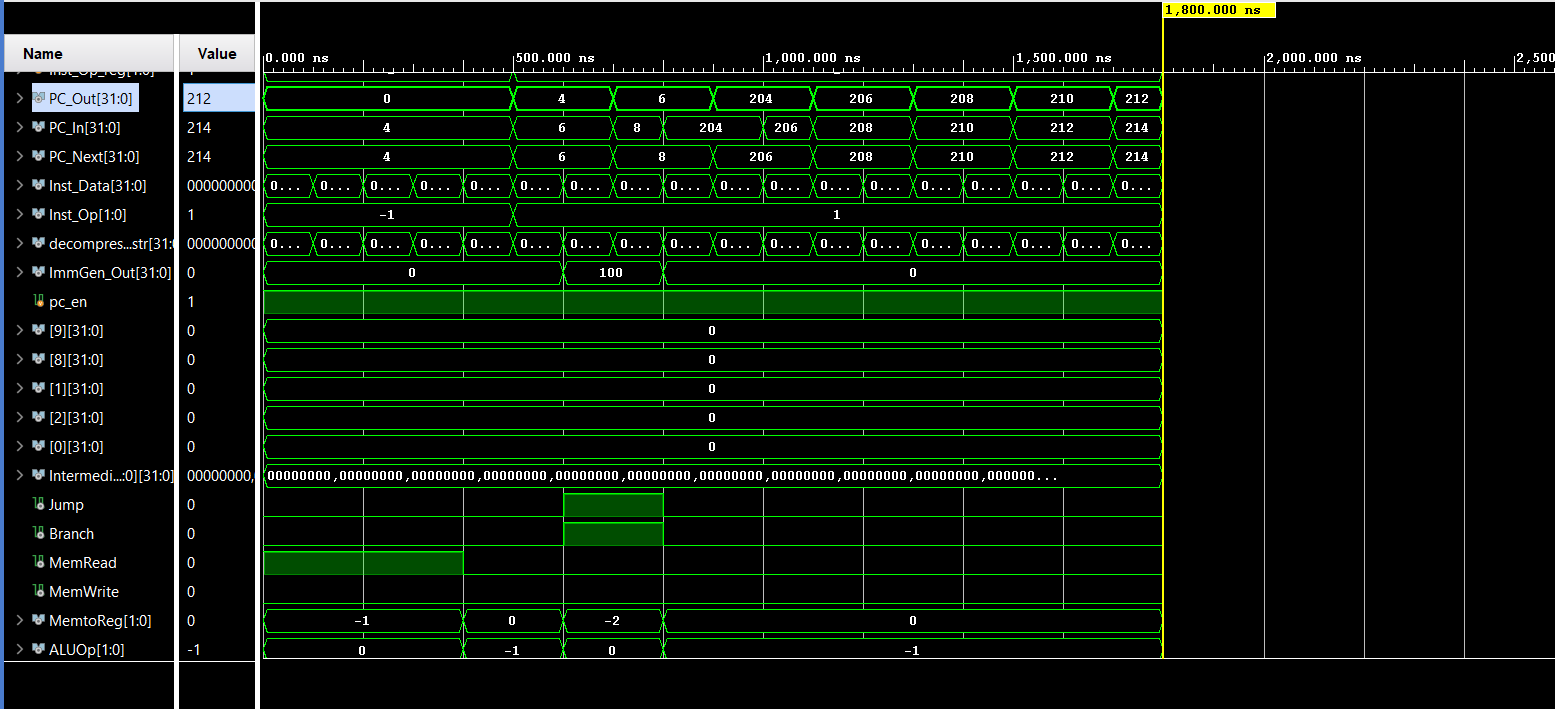
****

**C.J**

**c.j 200 (Expands to jal, x0, 200)**

**Binary: 1010000011100001**

**Hexadecimal: 0xa0e1**

****

**C.BEQZ**

**c.beqz x8, 32 (Expands to beq x8, x0, 32)**

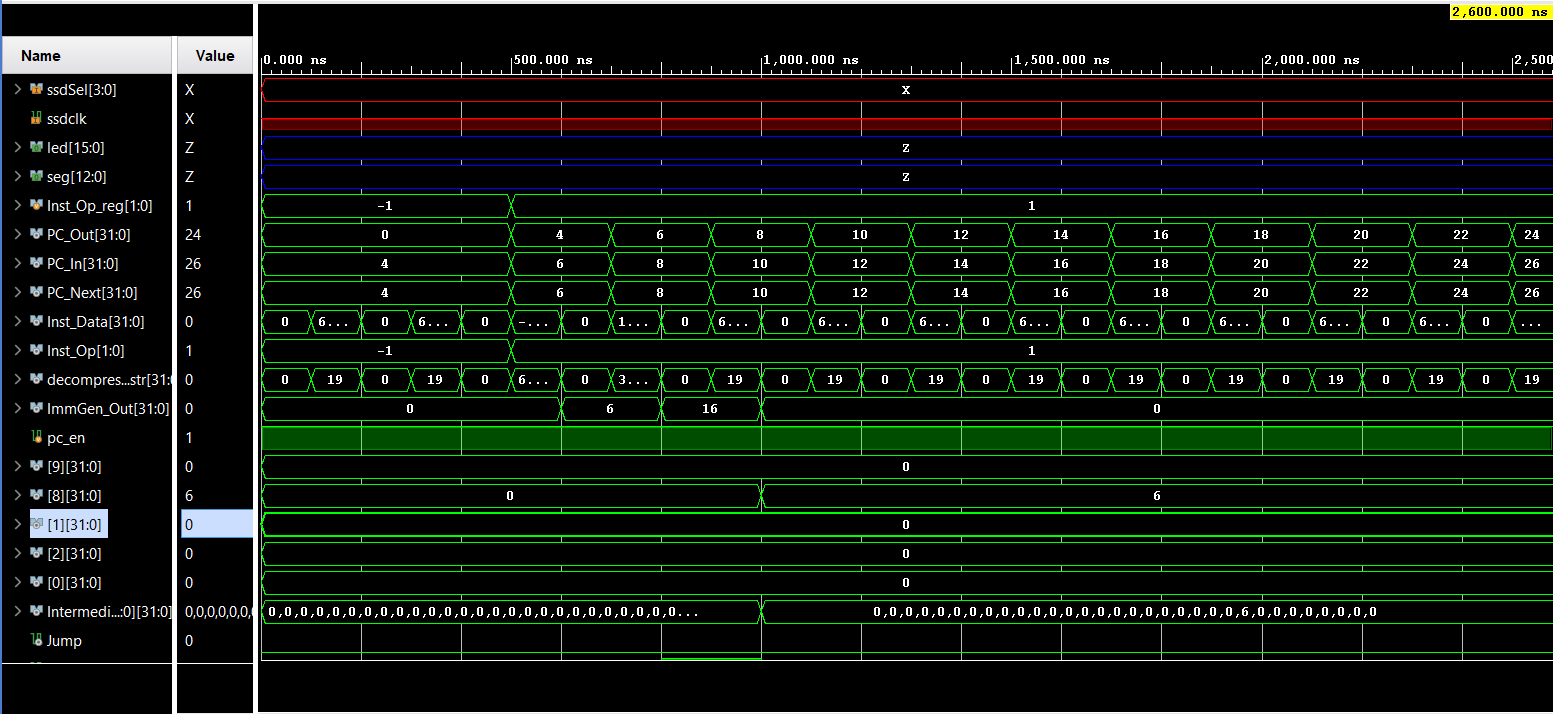
**Binary: 1100000000000101**

**Hexadecimal: 0xc005**

**x8 = 0**

****

**x8 = 6**

****

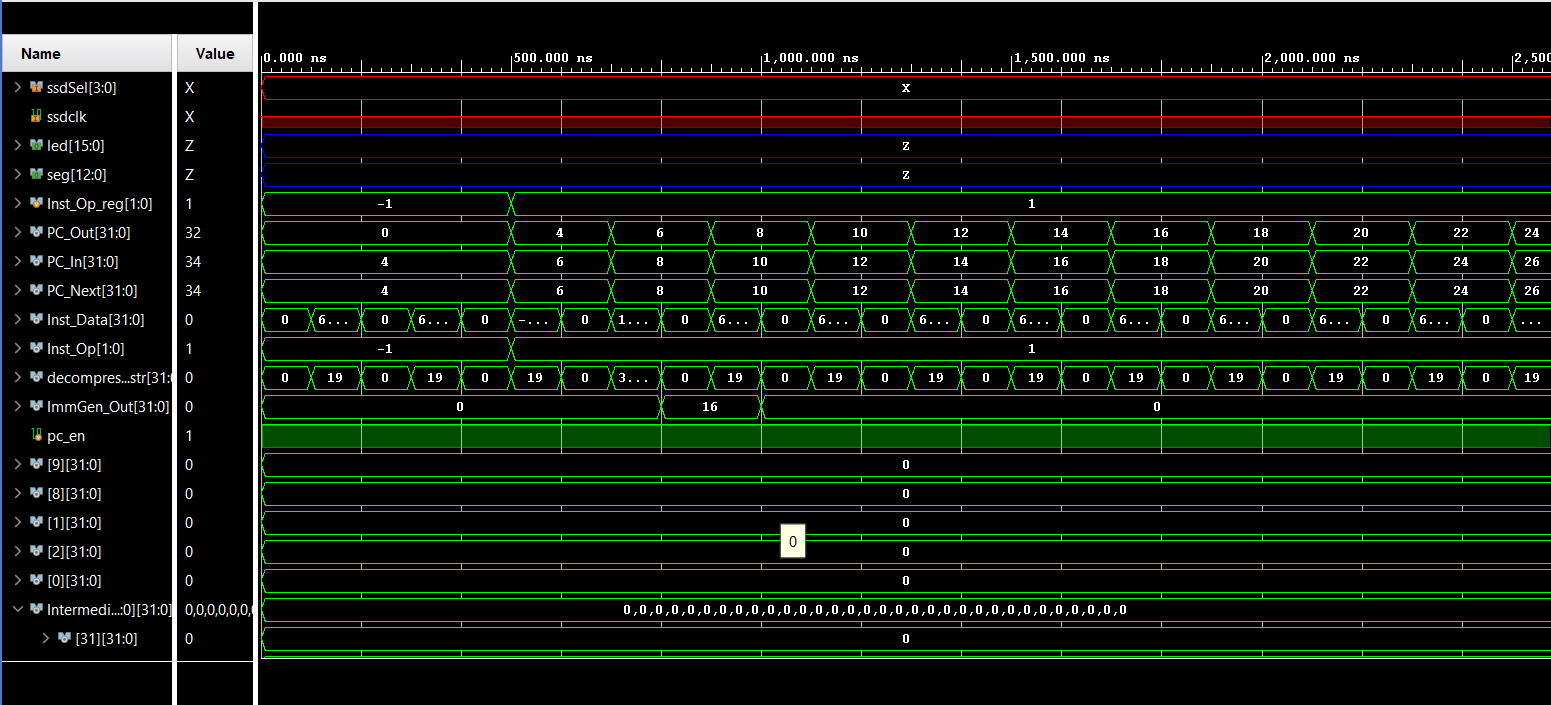
**C.BNEZ**

**c.bnez x8, 32 ( bne x8, x0, 32)**

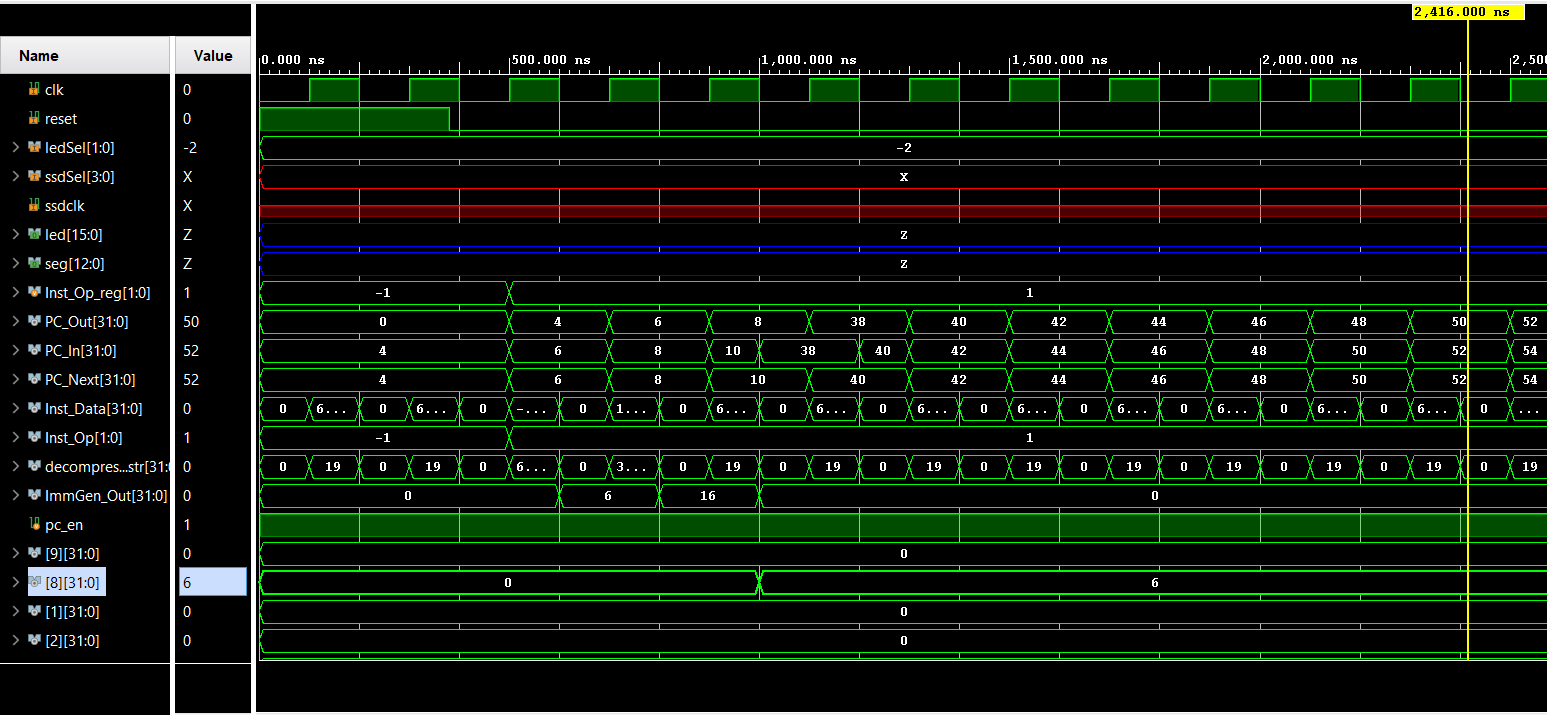
**Binary: 1110000000000101**

**Hexadecimal: 0xe005**

**x8 = 0**

****

**x8 = 6**

****

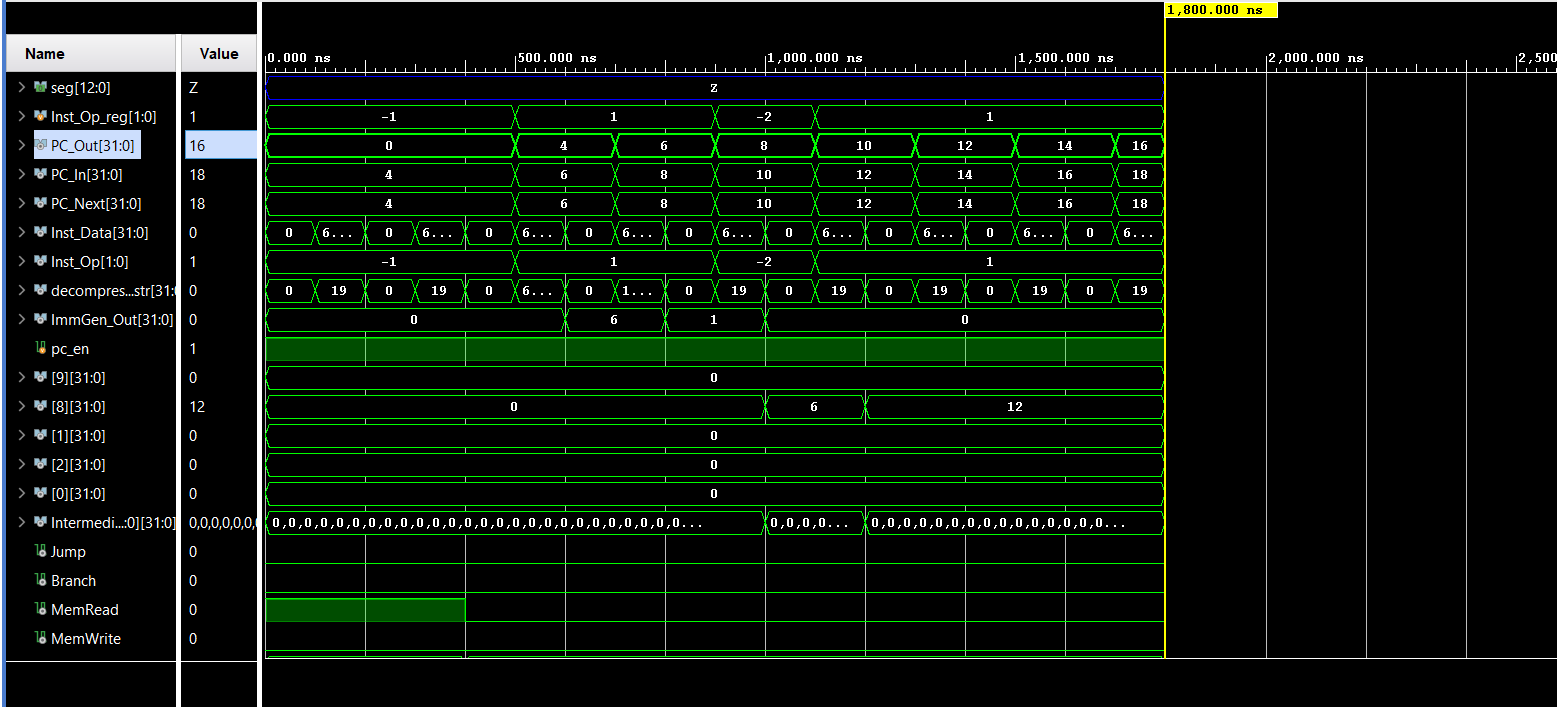
**C.SLLI**

**c.slli x8, 1 (Expands to x8, x8, 1)**

**Binary: 0000010000000110**

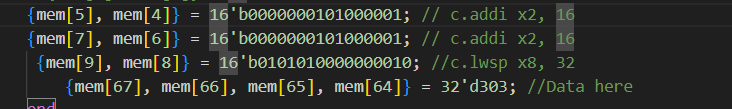
**Hexadecimal: 0x0406**

**x8 = 6**

****

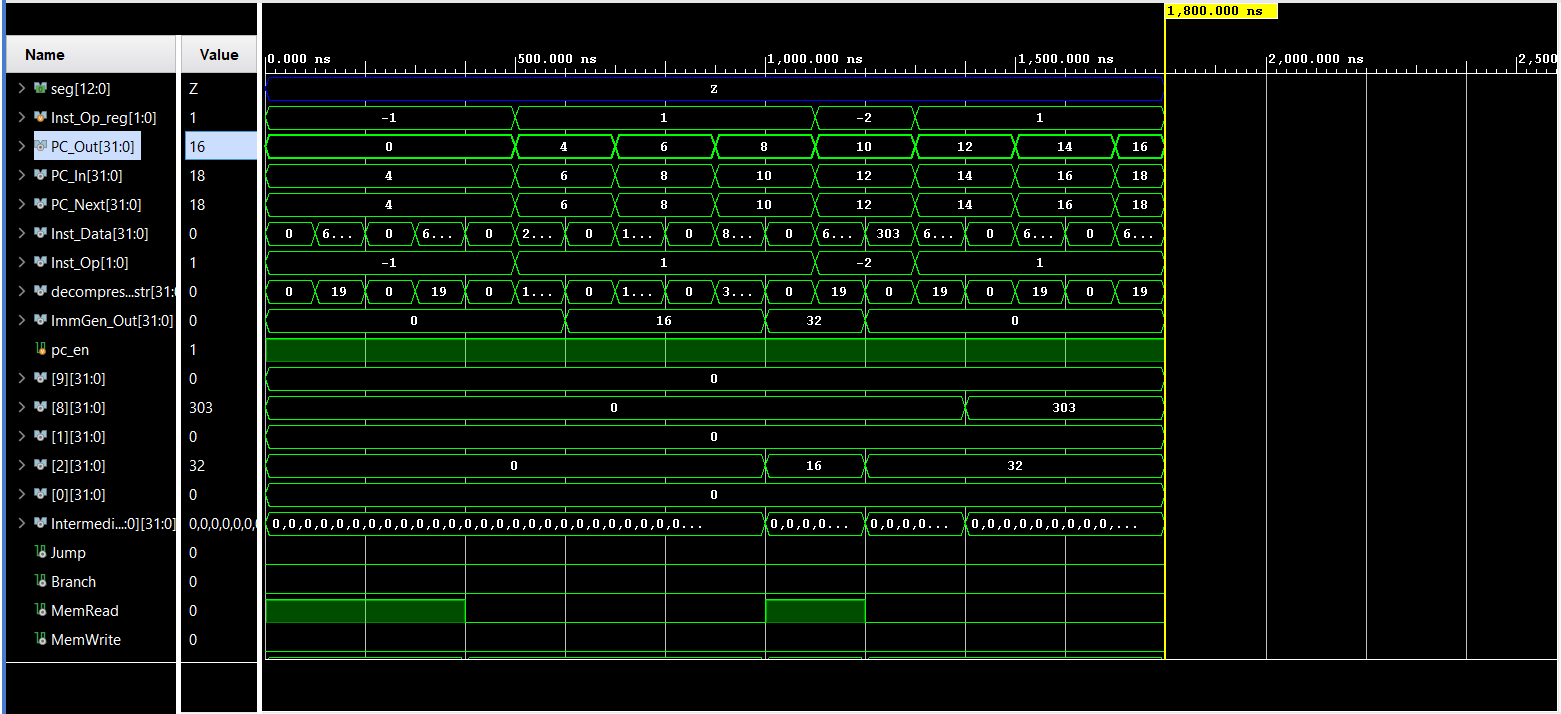
**C.LWSP**

**c.lwsp x8, 32 (Expands to lw x8, 32(x2))**

**Binary: 0101010000000010**

**Hexadecimal: 0x5402**

**x2 = 32**

****

**C.JR**

**c.jr x2 (Expands to jalr x0, 0(x2))**

**Binary: 1000000100000010**

**Hexadecimal: 0x8102**

**X2 = 32**

****

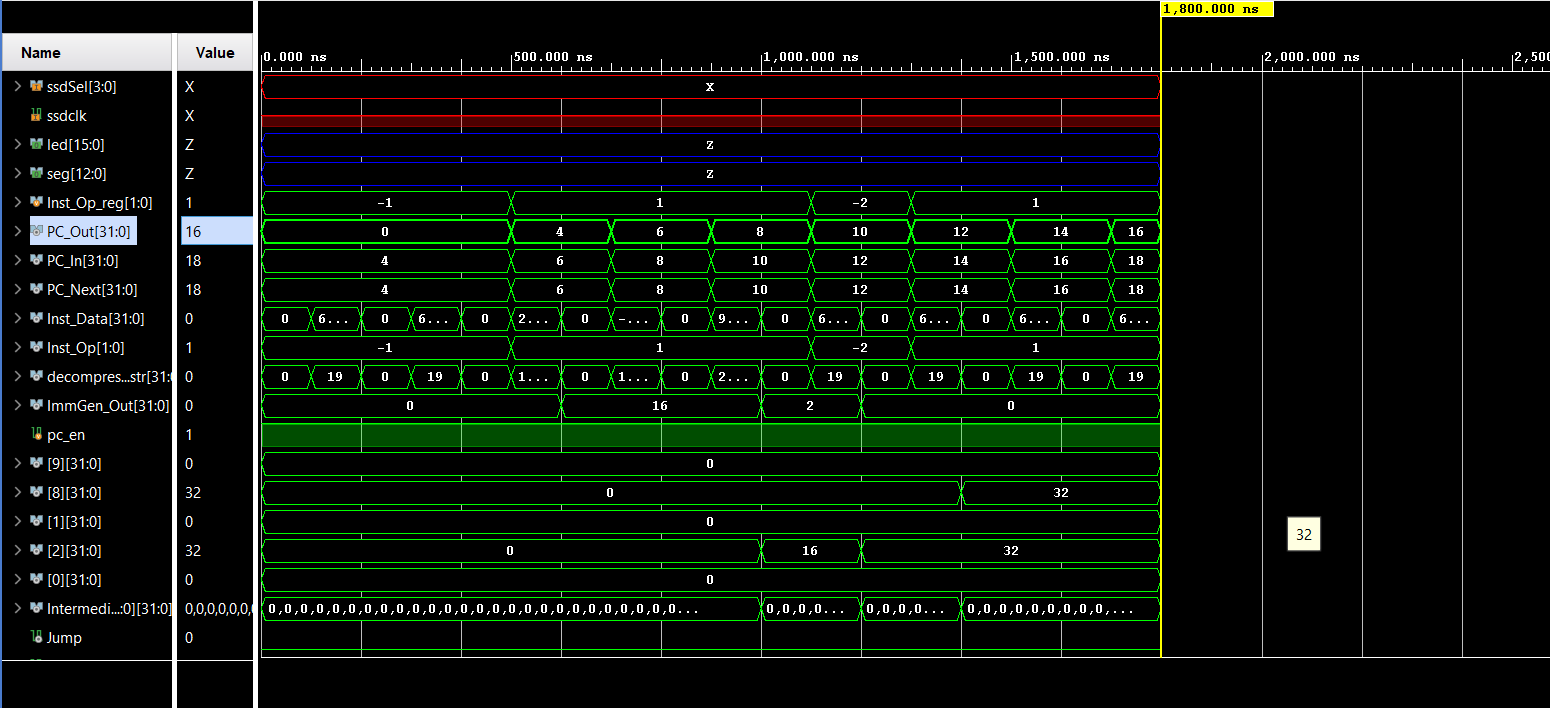
**C.MV**

**c.mv x8, x2 (Expands to add x8, x0, x2)**

**Binary: 1000010000001010**

**Hexadecimal: 0x840a**

**x2 = 32**

****

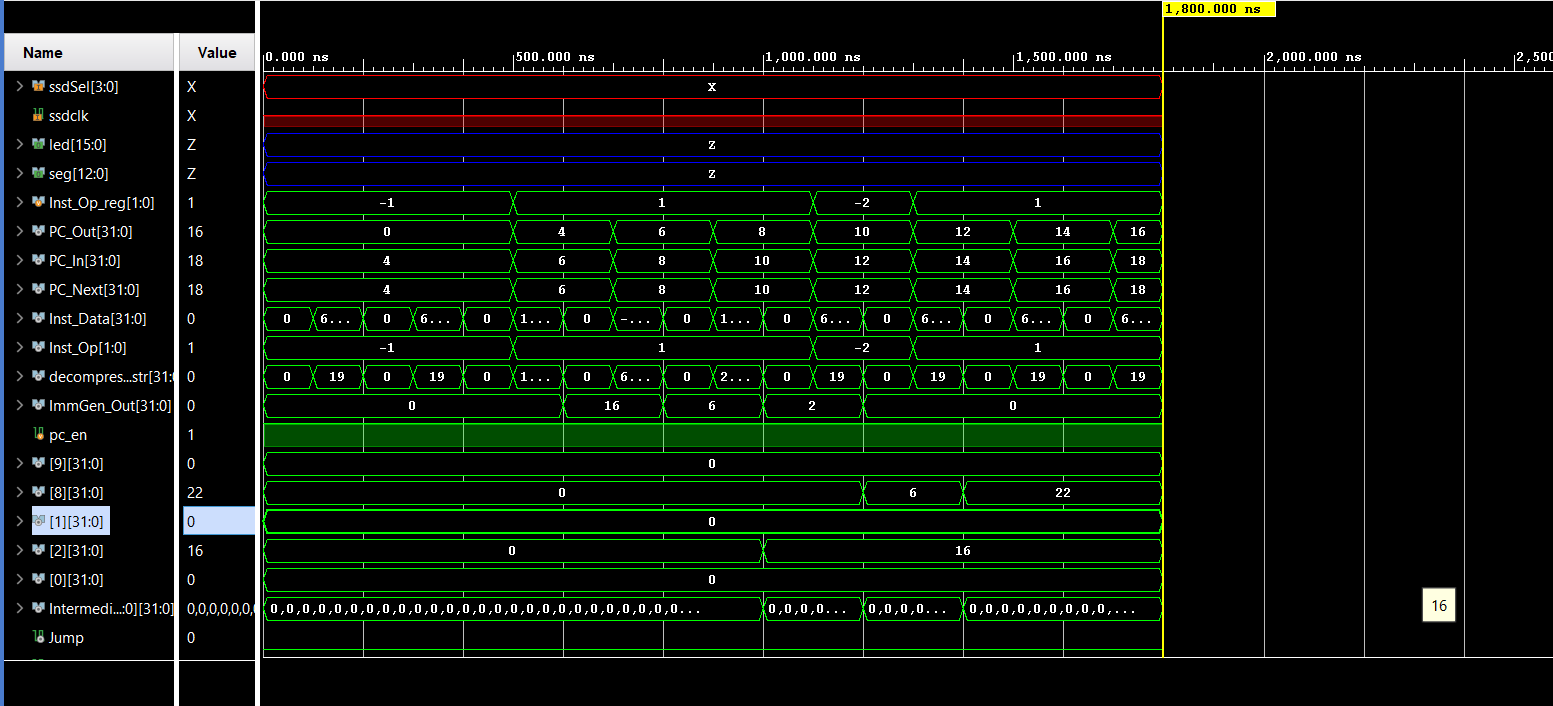
**C.ADD**

**c.add x8, x2 (Expands to add x8, x8, x2)**

**Binary: 1001010000001010**

**Hexadecimal: 0x940a**

**x8 = 6 x12=16 16+6 = 22**

****

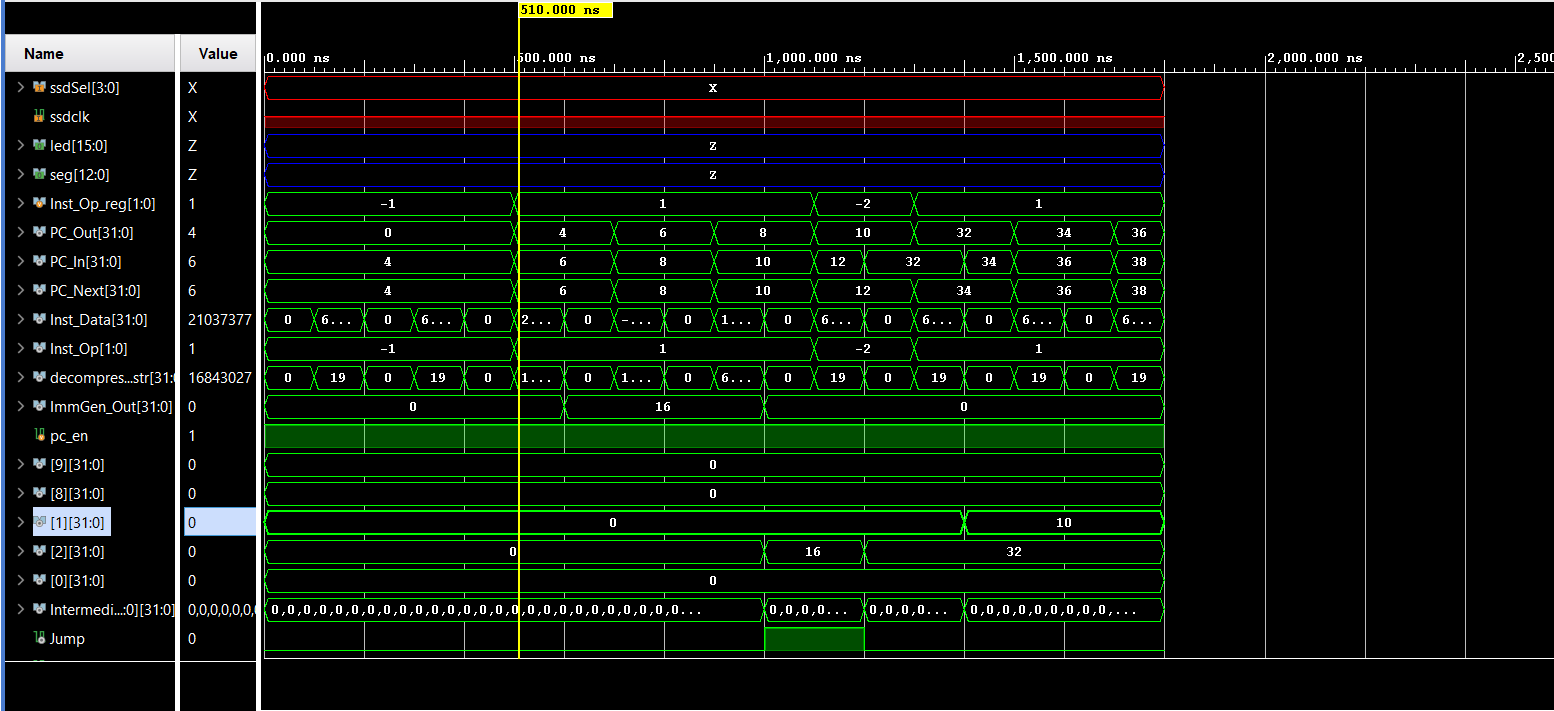
**C.JALR**

**c.jalr x2 (Expands to jalr x1, 0(x2))**

**Binary: 1001000100000010**

**Hexadecimal: 0x9102**

**x2 = 32**

****

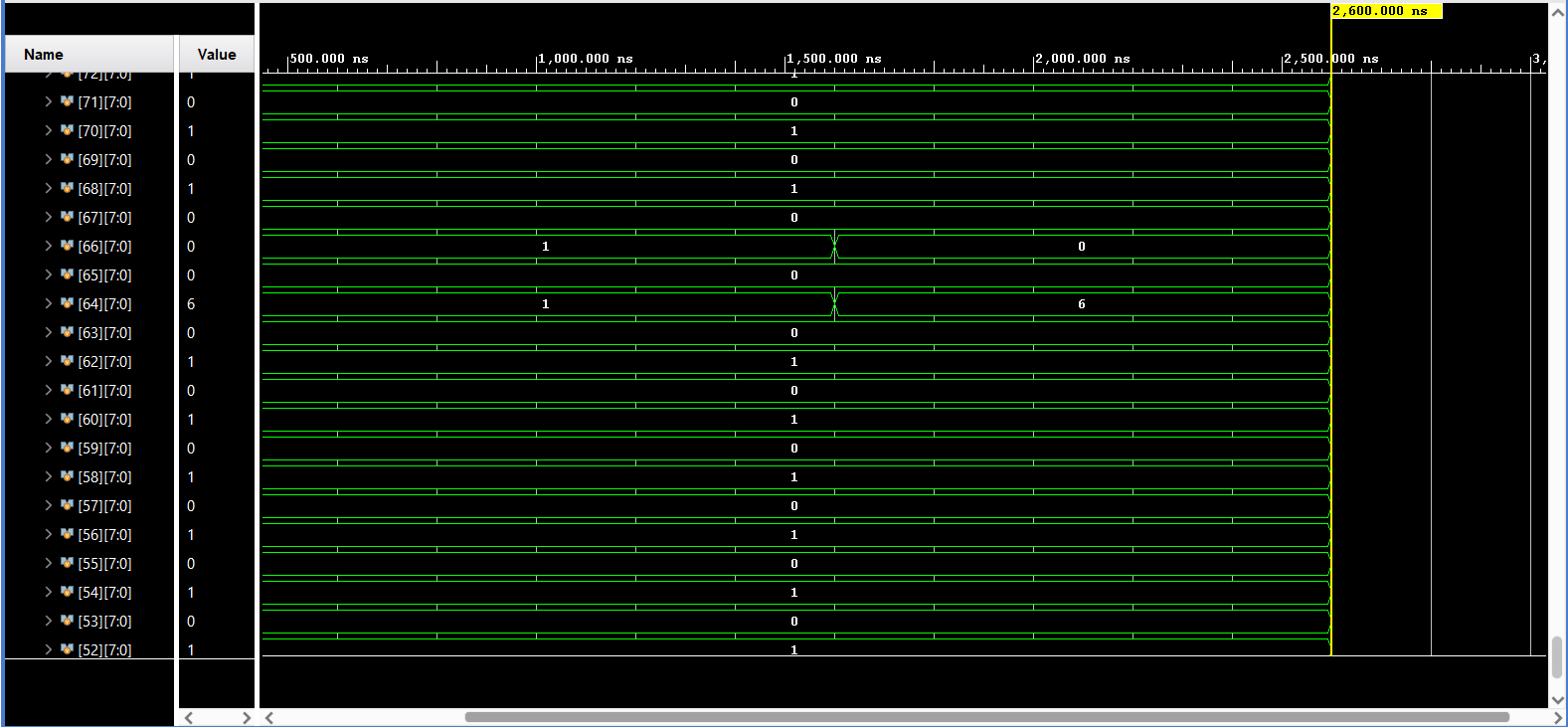
**C.SWSP**

**c.swsp x8, 32 (Expands to sw x8, 32(x2))**

**Binary: 1101000000100010**

**Hexadecimal: 0xd022**

**x8 = 6 x2 = 32 mem[64:67] = 32’d6**

****

***M Extension***

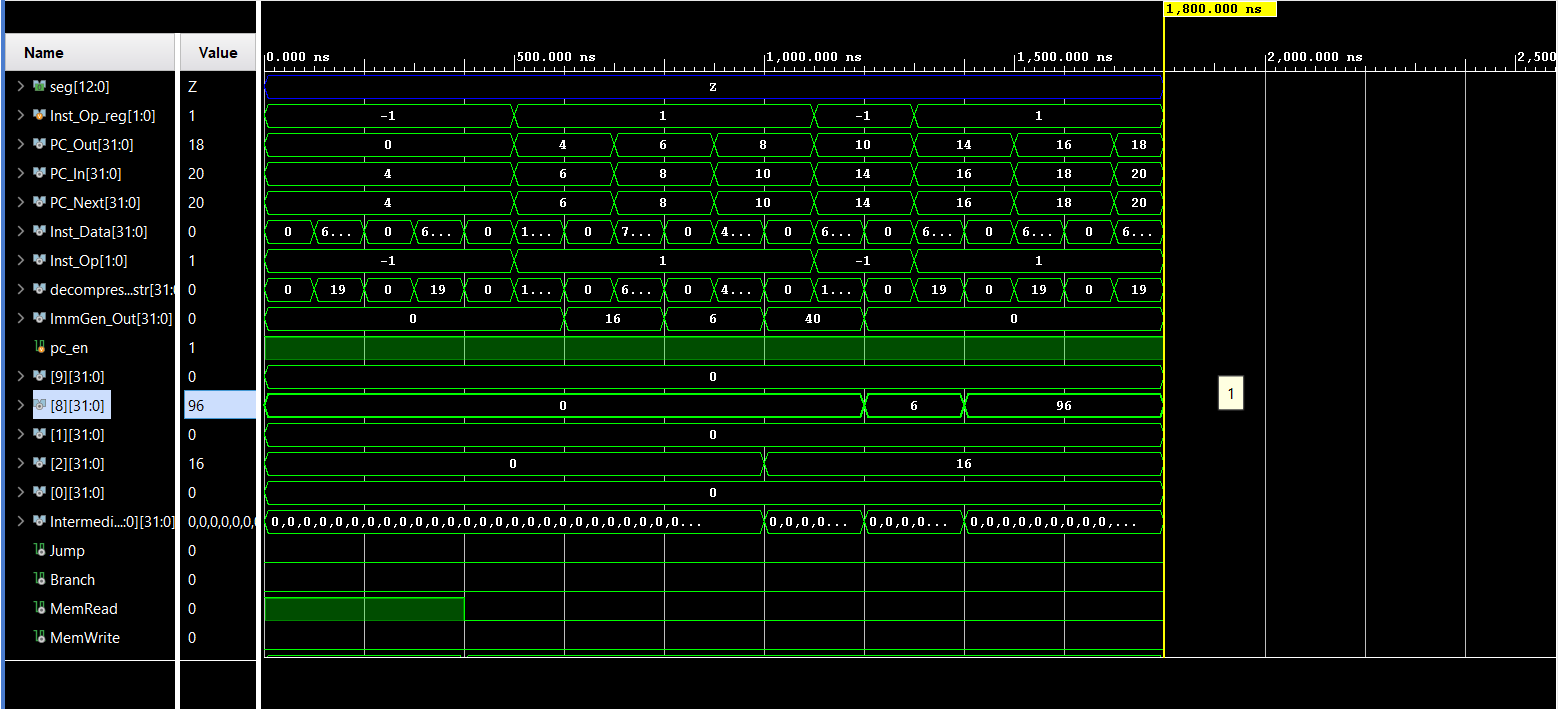
**MUL**

**mul x8, x2, x8**

**Binary:** **00000010100000010000010000110011**

**Hexadecimal:** **0x02810433**

**x8 = 6 x2 = 16 16 \* 6 = 96**

****

**MULH**

**mulh x8, x2, x8**

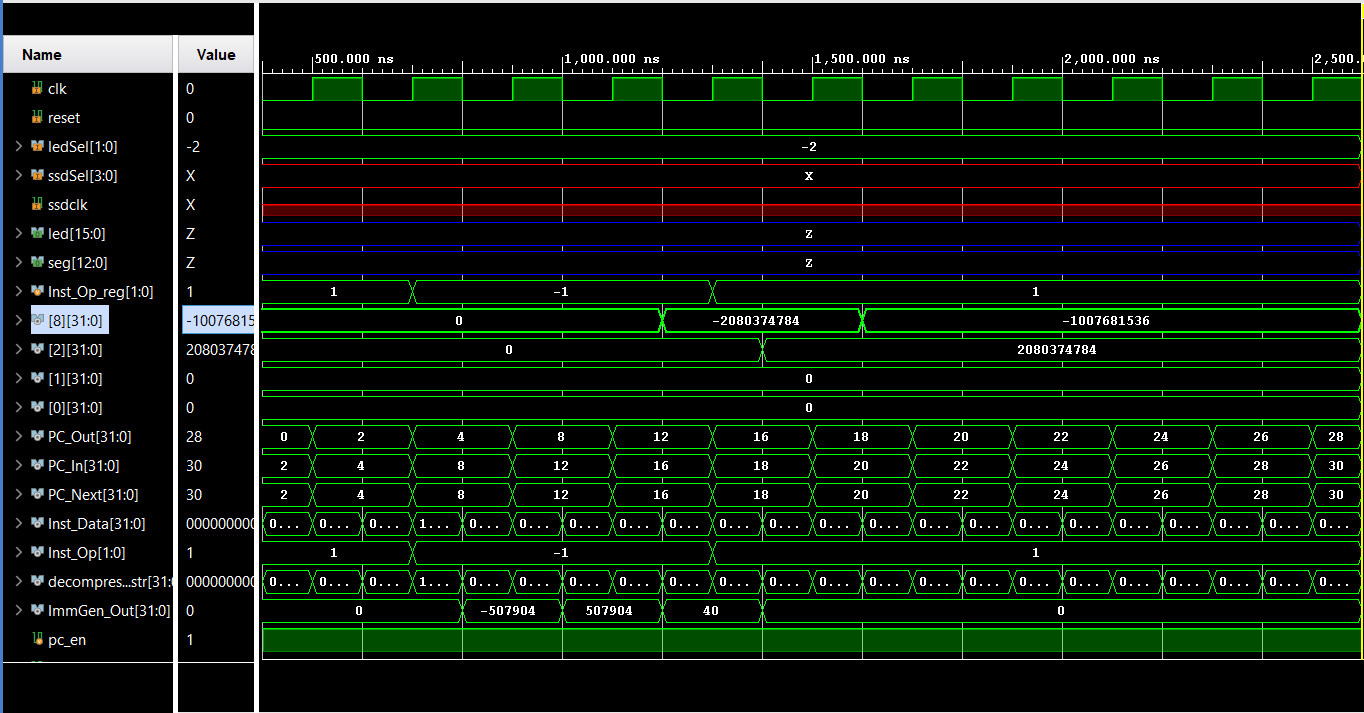
**Binary: 00000010100000010001010000110011**

**Hexadecimal: 0x02811433**

**x8 = -2080374784 x2=2080374784 -(2080374784^2) =**

**-4,327,959,241,903,046,656**

**-4,327,959,241,903,046,656/2^32 = -1,007,681,536 (approximately)**

****

**MULHSU**

**mulhsu x8, x2, x8**

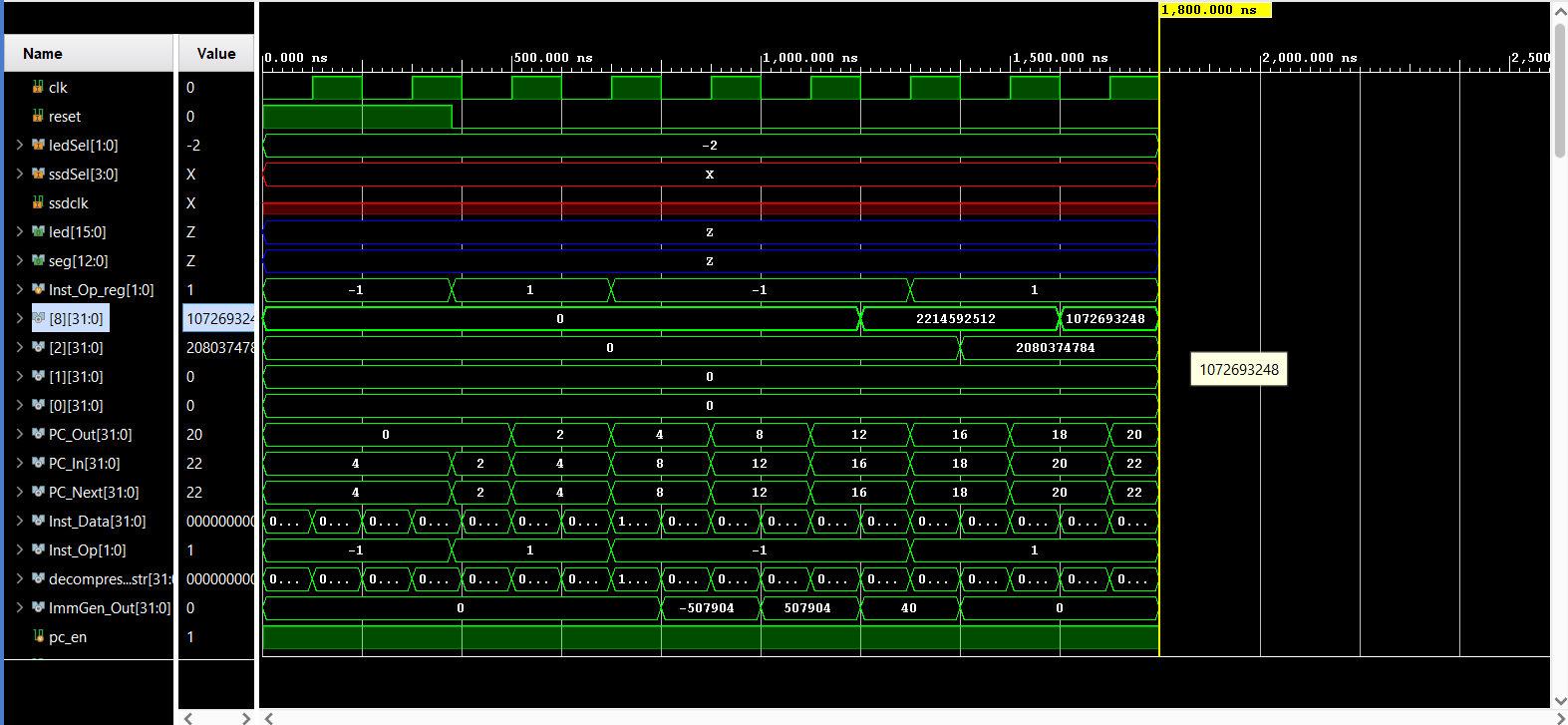
**Binary: 00000010100000010010010000110011**

**Hexadecimal: 0x02812433**

**x8 = 2214592512(-2080374784) x2=2080374784**

**2214592512\*2080374784=4,607,182,418,800,017,408**

**4,607,182,418,800,017,408/2^32 = 1,072,693,248**

****

**MULHU**

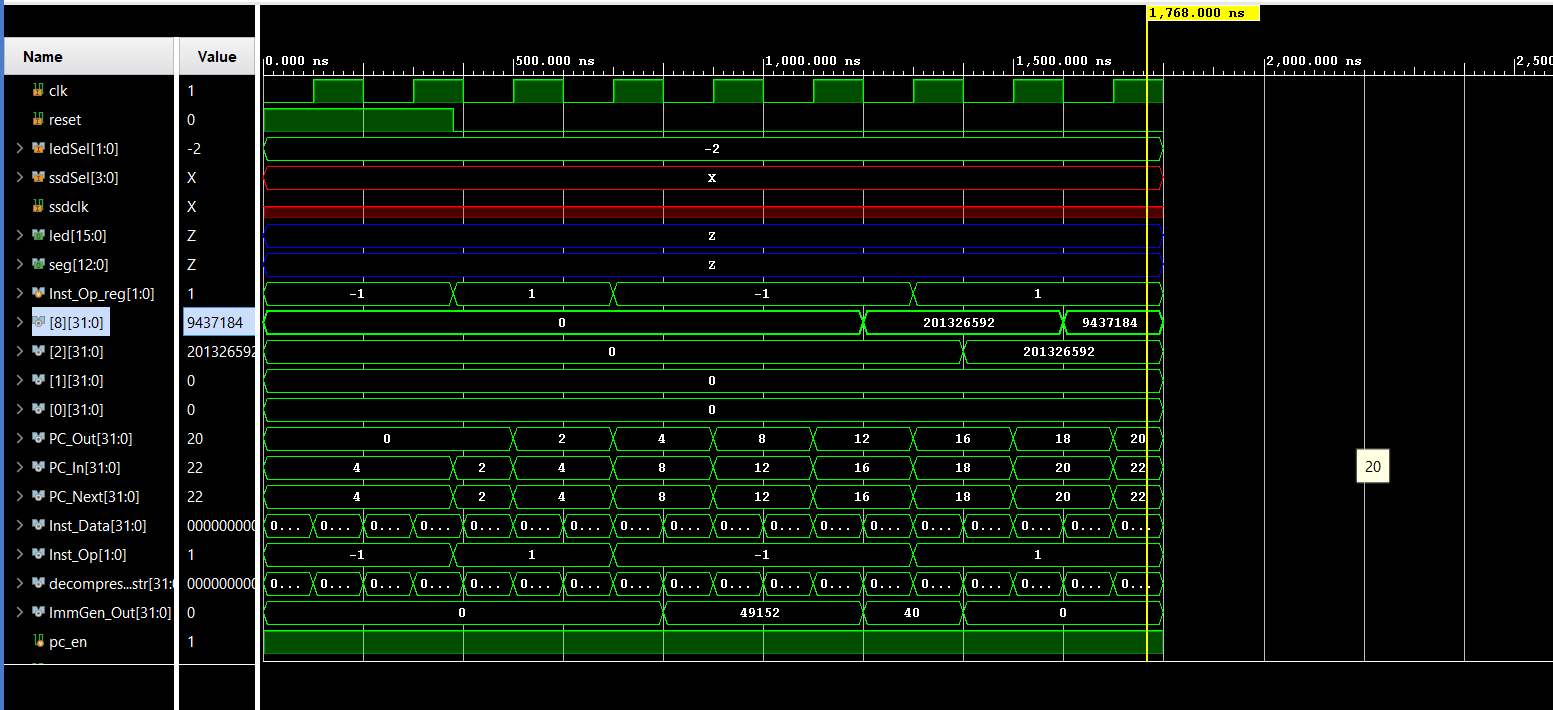
**mulhu x8, x2, x8**

**Binary: 00000010100000010011010000110011**

**Hexadecimal: 0x02813433**

**x8 = 201326592 x2=** **201326592 (201326592)^2 = 40,532,396,646,334,464**

**40,532,396,646,334,464/2^32 = 9,437,184**

****

**DIV**

**Does not work**

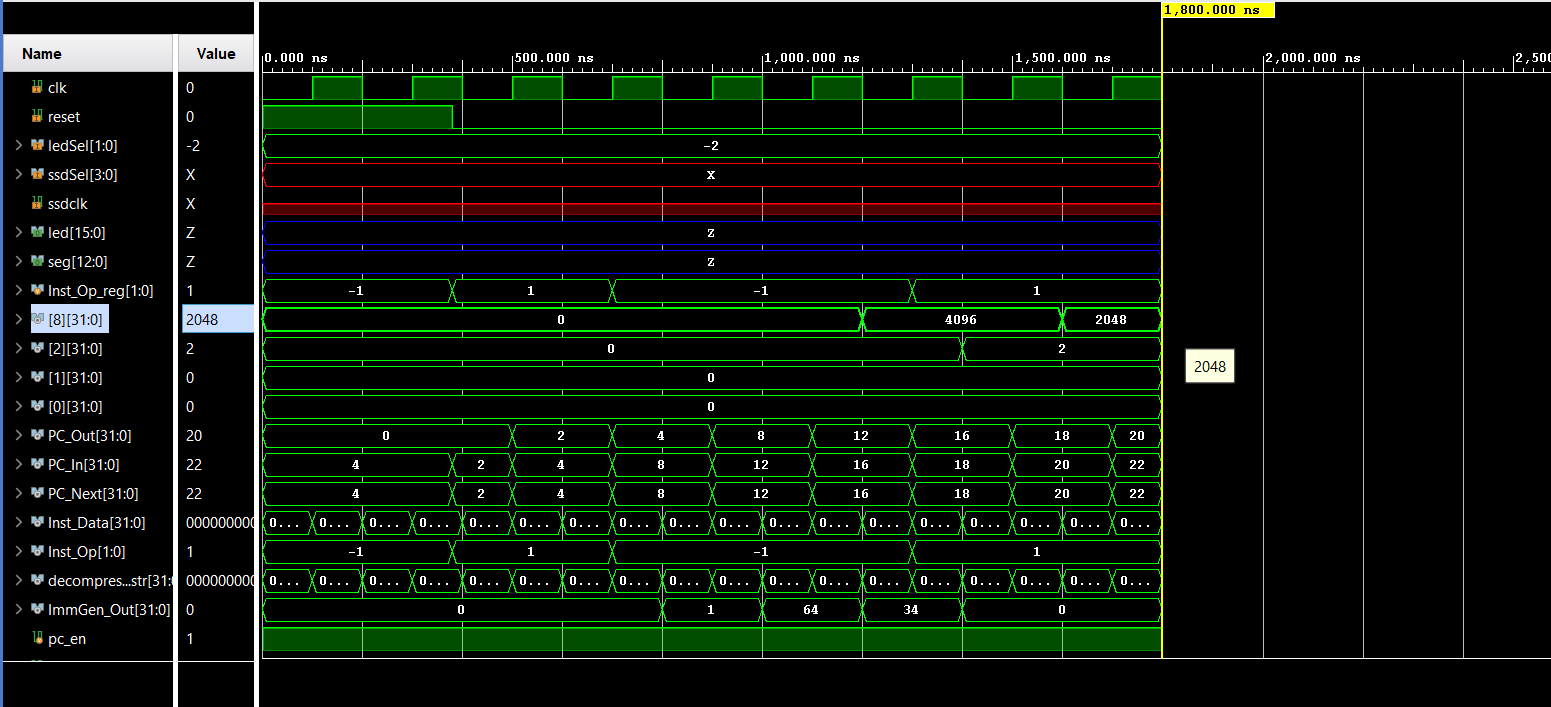
**DIVU**

**divu x8, x8, x2**

**Binary: 00000010001001000101010000110011**

**Hexadecimal: 0x02245433**

**X8 = 4096 x2 = 2 4096/2 = 2048**

****

**REM**

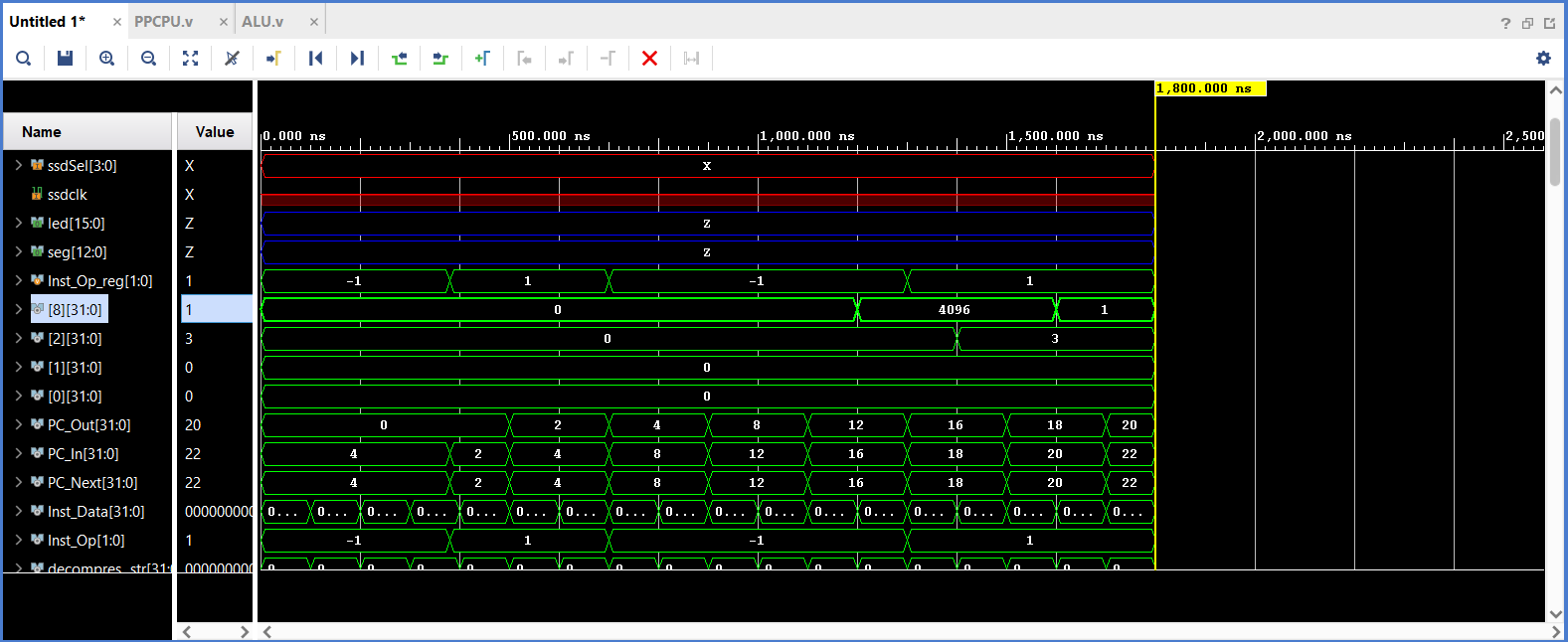
**Does not work**

**REMU**

**remu x8, x8, x2**

**Binary: 00000010001001000111010000110011**

**Hexadecimal: 0x02247433**

****

***Standard Instructions***

**Complete TestCase**

/ Load Instructions

mem[0] = 32'b00000000000000000010000010000011; // lw x1, 0(x0) // x1: 0 -> 1, PC: 0 -> 4 // we load the only initalized memory which hsa a value of 1 // PC: 0 -> 4

mem[1] = 32'b00000000000000000001000100000011; // lh x2, 0(x0) // x2: 0 -> 1, PC: 4 -> 8// we load the only the first 2 bytes of initalized memory which hsa a value of 1, and sign extend with(0) most signifcant bit // PC: 4 -> 8

mem[2] = 32'b00000000000000000000000110000011; // lb x3, 0(x0) // x3: 0 -> 1, PC: 8 -> 12// we load the only the first 1 byte of initalized memory which hsa a value of 1, and sign extend with(0) most signifcant bit // PC: 8 -> 12

mem[3] = 32'b00000000000000000100001000000011; // lbu x4, 0(x0) // x4: 0 -> 1, PC: 12 -> 16// we load the only the first 2 bytes of initalized memory which hsa a value of 1, and sign extend with(0) // PC: 12 -> 16

mem[4] = 32'b00000000000000000101001010000011; // lhu x5, 0(x0) // x5: 0 -> 1, PC: 16 -> 20// we load the only the first 1 bytes of initalized memory which hsa a value of 1, and sign extend with(0) // PC: 16 -> 20

// Store Instructions

// Immediate Arithmetic/Logic

mem[8] = 32'b00000000010100000000001100010011; // addi x6, x0, 5 // x6: 0 -> 5, PC: 32 -> 36, since all registers are int=itially set to 9

mem[9] = 32'b00000000101000110010001110010011; // slti x7, x6, 10 // x7: 0 -> 1, since (5 < 10) so x7 will have a value of 1, PC: 36 -> 40

mem[10] = 32'b00000000101000110011010000010011; // sltiu x8, x6, 10 // x8: 0 -> 1 since (5 < 10) so x7 will have a value of 1 here value of x is trated as unsigned but this will not make a difference here, PC: 40 -> 44

mem[11] = 32'b00000000001100110100010010010011; // xori x9, x6, 3 // x9: 0 -> 6 we take the xor of (5 ^ 3) which is 101 011 so we ge 110 which is 6, PC: 44 -> 48

mem[12] = 32'b00000000001100110110010100010011; // ori x10, x6, 3 // x10: 0 -> 7 we take the or of (5 | 3) which is 101 011 so we ge 111 which is 7, PC: 48 -> 52

mem[13] = 32'b00000000001100110111000010010011; // andi x1, x6, 3 // x1: 1 -> 1 we take the or of (5 & 3) which is 101 011 so we ge 001 which is 1, PC: 52 -> 56

mem[14] = 32'b00000000001000110001000100010011; // slli x2, x6, 2 // x2: 1 -> 20 shift the left bit twice (5 << 2) which is the equivalent to multiplying 5\*2\*2, PC: 56 -> 60

mem[15] = 32'b00000000001000010101000110010011; // srli x3, x2, 2 // x3: 1 -> 5 shift the left bit twice (20 >> 2) which is the equivalent to dividing 20/(2\*2)=5 , PC: 60 -> 64

mem[16] = 32'b11111111111100000000001000010011; // addi x4, x0, -1 // x4: 1 -> 0xFFFFFFFF (Since this is asigned number -1 has the format 0xFFFFFFFF), PC: 64 -> 68

mem[17] = 32'b01000000001000100101001010010011; // srai x5, x4, 2 // x5: 1 -> 0xFFFFFFFF (-1 >> 2) the result will not chnage since the sign was perserved because this is srai, PC: 68 -> 72

// Register-Register Arithmetic/Logic

mem[18] = 32'b00000000000100110000001110110011; // add x7, x6, x1 // x7: 1 -> 6 simply add x6, x1 note that x6 ws updated to 5 ar addi instruction(5 + 1), PC: 72 -> 76

mem[19] = 32'b01000000000100111000010000110011; // sub x8, x7, x1 // x8: 1 -> 5 simply subtraction x7 was updated in step above, (6 - 1), PC: 76 -> 80

mem[20] = 32'b00000000000100111001010010110011; // sll x9, x7, x1 // x9: 6 -> 12 (6 << 1), PC: 80 -> 84

mem[21] = 32'b00000000100000111010010100110011; // slt x10, x7, x8 // x10: 7 -> 0 here we check if x7<x8 and set the boolean value of the comparison to x10(6 < 5 = false), PC: 84 -> 88

mem[22] = 32'b00000000000100111011000010110011; // sltu x1, x7, x1 // x1: 1 -> 0 same as above snice both numbers are treated as unsigned to ffffff wil be treated as a huge positive value, but here both values are positive (6 < 1 = false), PC: 88 -> 92

mem[23] = 32'b00000000100000111100000100110011; // xor x2, x7, x8 // x2: 20 -> 3 take the xor of (6 ^ 5) 110 101 which is 011, PC: 92 -> 96

mem[24] = 32'b00000000000101001101000110110011; // srl x3, x9, x1 // x3: 5 -> 6 shift to right by 1 which is equivalent to dividng by 2 so we have (12 >> 1)=6, PC: 96 -> 100

mem[25] = 32'b01000000000100100101001000110011; // sra x4, x4, x1 // x4: 0xFFFFFFFF -> 0xFFFFFFFF (-1 >> 1) doesn't change since sign was perserved, PC: 100 -> 104

mem[26] = 32'b00000000100000111110001010110011; // or x5, x7, x8 // x5: 0xFFFFFFFF -> 7 we simply take the oring of 110, 101 which is equal to 111(6 | 5), PC: 104 -> 108

mem[27] = 32'b00000000100000111111001100110011; // and x6, x7, x8 // x6: 5 -> 4 we take anding of 110, 101 which 100 which is 4 ,(6 & 5), PC: 108 -> 112

// Upper Immediates

mem[28] = 32'b00000000000000000001000010110111; // lui x1, 1 // x1: 0 -> 4096 , we load the 20 bits from immediate and them by 12 so we get (1 << 12), PC: 112 -> 116

mem[29] = 32'b00000000000000000001000100010111; // auipc x2, 1 // x2: 3 -> 4212 we add pc to immediate shifted by 12 to the left (116 + 1 << 12), PC: 116 -> 120

// Jumps

mem[30] = 32'b00000000100000000000000111101111; // jal x3, 8 // x3: 5 -> 124 we store the next instruction in x3 which is pc+4, then we adds offset to pc, PC: 120 -> 128

mem[31] = 32'; // addi x0, x0, 0 // NOP, PC: 128 -> 132, we just insert a dummy instruction that wll not be executed to make sure that we tested all instructions

mem[32] = 32'b00000000000000011000001001100111; // jalr x4, 0(x3) // x4: 0xFFFFFFFF -> 136 (PC+4) here we store 0(x3) which is 124 as the new Pc, PC: 132 -> 124 (x3)

// Branches (all test taken case, offset=8, with dummy NOP), here operands are adjusted so that branching is gurantted

mem[31] = 32'b00000000000100001000010001100011; // beq x1, x1, 8 // PC: 124 -> 132 (x1 == x1), which will always be true

mem[32] = 32'; // addi x0, x0, 0 // NOP, PC: 132 -> 136 we just insert a dummy instruction that wll not be executed to make sure that we tested all instructions

mem[33] = 32'b00000000100000111001010001100011; // bne x7, x8, 8 // PC: 136 -> 144 (6 != 5), so we branch

mem[34] = 32'; // addi x0, x0, 0 // NOP, PC: 144 -> 148 we just insert a dummy instruction that wll not be executed to make sure that we tested all instructions

mem[35] = 32'b00000000011101000100010001100011; // blt x8, x7, 8 // PC: 148 -> 156 (5 < 6), swapped x7, x8

mem[36] = 32'; // addi x0, x0, 0 // NOP, PC: 156 -> 160 we just insert a dummy instruction that wll not be executed to make sure that we tested all instructions

mem[37] = 32'b00000000100000111101010001100011; // bge x7, x8, 8 // PC: 160 -> 168 (6 >= 5)

mem[38] = 32'; // addi x0, x0, 0 // NOP, PC: 168 -> 172 we just insert a dummy instruction that wll not be executed to make sure that we tested all instructions

mem[39] = 32'b00000000011101000110010001100011; // bltu x8, x7, 8 // PC: 172 -> 180 (5 < 6), swapped x7, x8

mem[40] = 32'; // addi x0, x0, 0 // NOP, PC: 180 -> 184 we just insert a dummy instruction that wll not be executed to make sure that we tested all instructions

mem[41] = 32'b00000000100000111111010001100011; // bgeu x7, x8, 8 // PC: 184 -> 192 (6 >= 5)

mem[42] = 32'; // addi x0, x0, 0 // NOP, PC: 192 -> 196 we just insert a dummy instruction that wll not be executed to make sure that we tested all instructions

mem[5] = 32'b00000000000100000010001000100011; // sw x1, 4(x0) // mem[4-7]: 0 -> 1, PC: 20 -> 24, so simply mem[4] will be one the rest will be 0 // PC: 20 -> 24

mem[6] = 32'b00000000001000000001010000100011; // sh x2, 8(x0) // mem[8-9]: 0 -> 1, PC: 24 -> 28, we only store 2 bytes, so the other 2 bytes of memory will not be updated here mem[8] will be 1 and mem[9]is 0 // PC: 24 -> 28

mem[7] = 32'b00000000001100000000011000100011; // sb x3, 12(x0) // mem[12]: 0 -> 1, PC: 28 -> 32 we only store 1 bytes, so the other 3 bytes of memory will not be updated here mem[12] will be 1 // PC: 28 -> 32

