32 Bit Multicycle CPU – MIPS

Niklas Kunkel (Boston University)

EC413 – Computer Organization

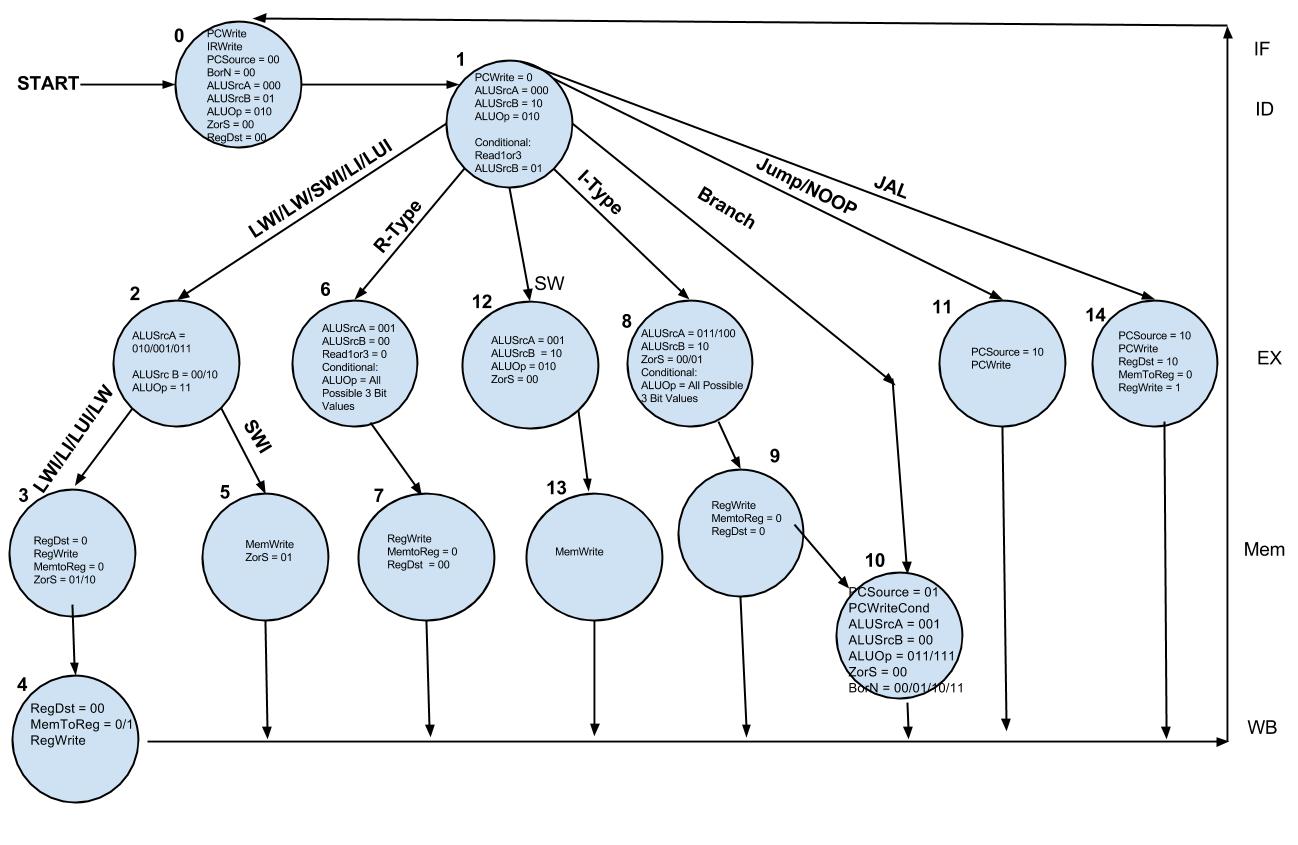
U47792740

12/08/13

Objective:

To create a 32 bit Multicycle Bare MIPS Processor in order to better understand the inner workings of a computer. Speed efficiency was a critical objective. Support for as many MIPS Assembly Language instructions as possible was also a driving goal. Area optimization was also attempted by deleting unnecessary hardware.

# State Machine



# Stage Advancement by Instruction Type

R Type = Stage 0 -> Stage 1 -> Stage 6-> Stage 7 - > Stage0

I-Type = Stage 0 -> Stage 1 -> Stage 8 -> Stage 9 -> Stage 10 -> Stage0

Branch = Stage 0 -> Stage 1 -> Stage 10  -> Stage0

Jump/NOOP = Stage 0 -> Stage 1 -> Stage 11 -> Stage 0

JAL = Stage 0 -> Stage 1 -> Stage 14 -> Stage 0

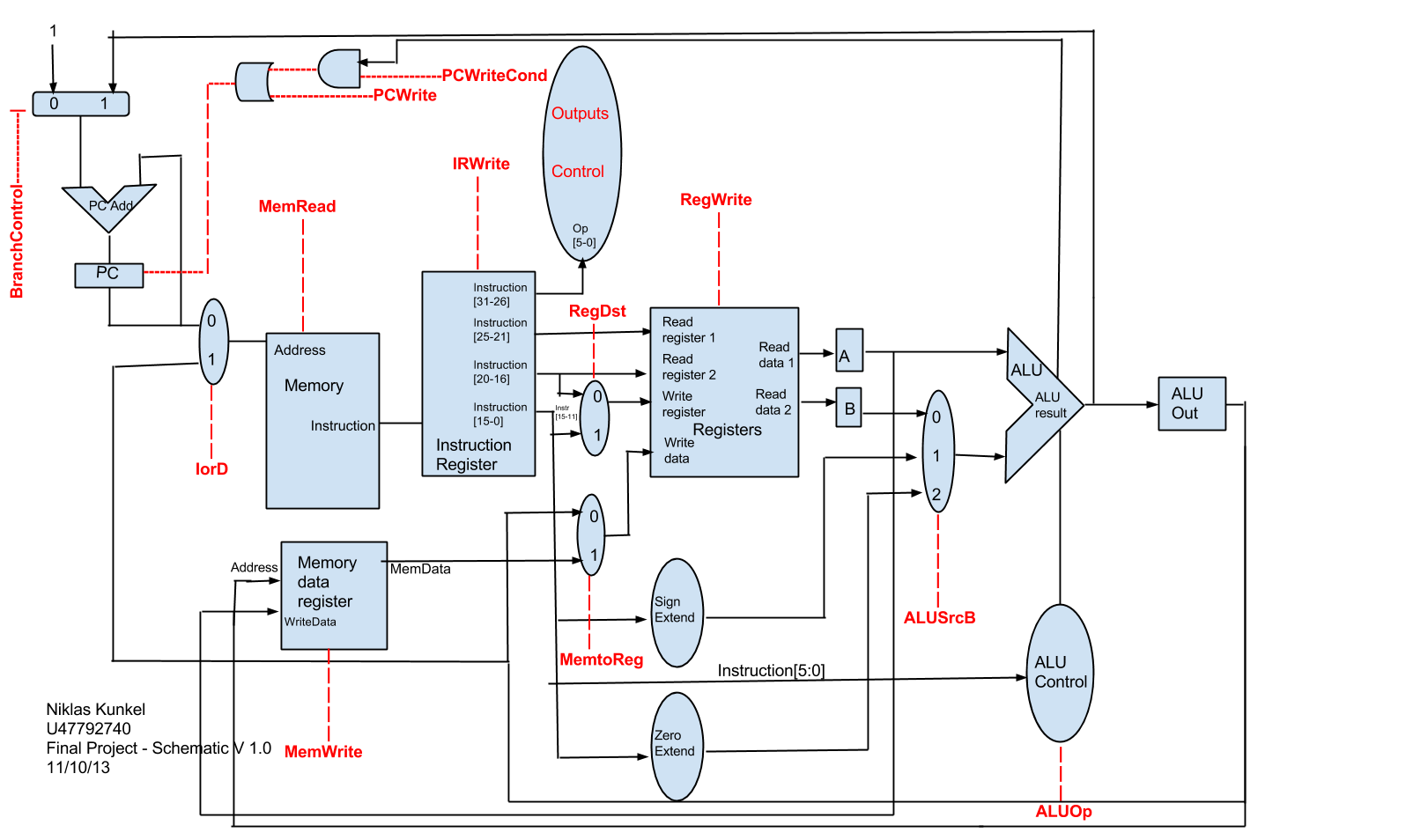
SWI = Stage 0 -> Stage 1 -> Stage 2 -> Stage 5 -> stage0

SW = Stage 0 -> Stage 1 -> Stage 12 -> Stage 13 - > Stage0

LW/LWI/LI/LUI = Stage 0 -> Stage 1 -> Stage 2 -> Stage 3 -> Stage 4 -> Stage0

# Hardware Configuration

**The Original Hardware configuration as per Milestone 1 can be seen here:**

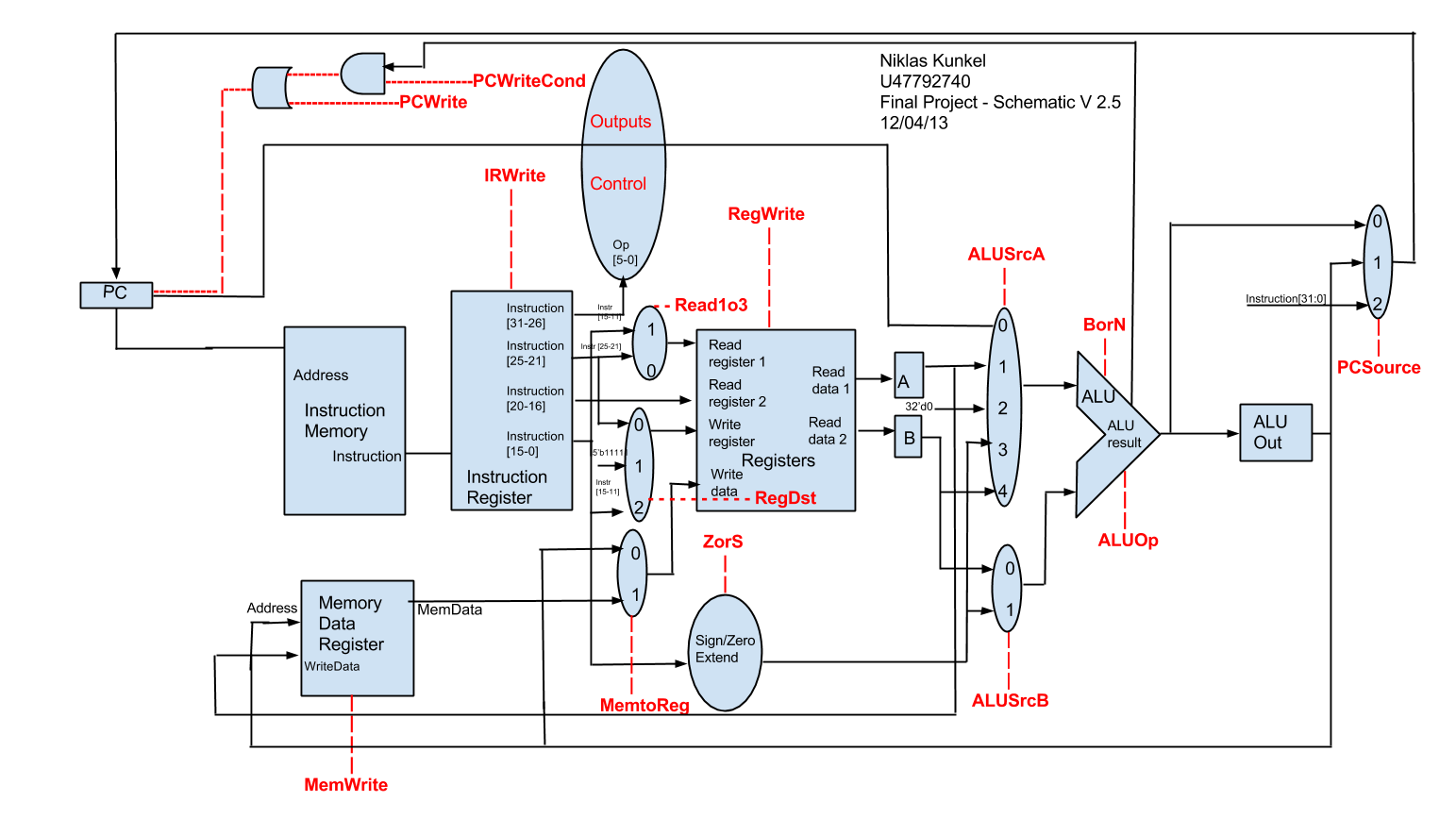


**Hardware Design Changes**

The original design stressed to minimize complexity. In comparison to the second revision, the first revision (shown above) lacked an ALUSrcA MUX and a PCSource MUX. Instead program counter incrementation was going to be done through a dedicated second full adder with a MUX called BranchControl (with it’s own newly created controller value) deciding whether PC would be incremented by 4 or by an ALU output value. This hardware design choice though making the hardware implementation look less cluttered, resulted in difficulties during implementation of branch and jump logic. It was seen as a necessary change to redesign the hardware to a more complex state, to be able to handle all commands that needed to be supported. ALU Control was fused into the MUX, and Zero and Sign extender modules were homogenized into a single unit. Consequently some unneeded controller values such as MemRead were removed.

Debugging stages revealed some serious flaws with the existing model, and in response many more multiplexers were extended to accommodate special corner-case scenarios. This saw the expansion of ALUSrcA to a 3 bit control value.

**Below is the Final Hardware Implementation:**



# Verilog Hardware Design Implementation

**Logic\_Control**

This is the Controller for the CPU. It handles all instruction inputs, and cycles through states synced to a clock based off the instruction type. Each state has corresponding changes to control values which feed into the rest of the hardware to ensure correct execution of commands. These control values are used to choose values in MUXs, allow writing of data to registers and memory, and toggle alterations of the program counter.

**ALU + ALU\_Controller All-In-One**

The ALU is comprised of a variable Full Adder. It functions behaviorally and combines the oftentimes separated ALU Controller and ALU Module into a single unit to minimize excessive hardware. It bases its ALUOp from the state machine assigned values and is fed in from the ALUSrcA and ALUSrcB multiplexer. In order to accomdate all the different functionaility required of the ALU, we extended the ALUOp to a 3 bit A zero flag is output for branching purposes to assign whether two input values are equivalent.

**ALU\_Out**

This register is used to keep the hardware in sync. It’s used to funnel the output from the ALU into the PCSource MUX as well as WriteData for writing to Memory and the MemToReg mux to write to Registers. For purposes of altering the program counter at the exact moment in the cycle, output from the ALU is sometimes taken directly rather than the ALU\_Out register.

**Sign\_Logical\_Extend (ZorS)**

This is a combined zero and sign extender. I added an extra controller value called ZorS (Zero Or Sign) to toggle, whether the immediate value is sign or zero extended dependent on the state machine identifying which type of command is input.

**PC\_Reg**

Keeps track of the current program count. Since our Instruction Memory is divided in words being equivalent to 1 byte instead of the regular 4 bytes, it will increment by 1 byte after each instruction. Branch, Jump, and JAL all work on the PC\_Reg.

**PC\_Mux (PCSource)**

Uses the control line PCSource to select between the Jump Instruction, the result coming directly out of the ALU, or the output of the ALU\_Out register.

**Write\_Reg\_Mux (RegDst)**

This MUX is controlled by the select control line RegDst. It selects the value of the value of the Register we’re going to be writing to between the Instr\_25\_21 and Instr\_15\_11 lines.

**Read1Or3\_Mux (Read1o3)**

This Mux is controlled by the select control line Read1o3. It select the value of the ReadReg1 input for the General Purpose Register from either the first 5 or second set of 5 bits after the 6 bit Opcode in an Instruction.

**Write\_Data\_Mux (MemToReg)**

Uses MemToReg Control to select either the data coming out of Memory or the data coming out of the ALU output register . This data in turn is then passed along to be used as WriteData for our General Purpose Registers.

**Reg\_A**

This is a temporary container value for the output of ReadData1. It passes along its value into the ALUSrcA MUX.

**Reg\_B**

This is a temporary container value for the output of ReadData2. It passes along its value into the ALUSrcB MUX.

**Reg\_A\_Mux (ALUSrcA MUX)**

This Mux is used to select for the input A to the ALU. It will either be the value output from Register A (Reg\_A), the value output from Register B, the program counter, or the Sign/Zero Extended Immediate input.

**Reg\_B\_Mux (ALUSrcB MUX)**

This Mux is used to select for the input B to the ALU. It will either be the value output from Register B (Reg\_B), the Sign/Zero Extended Immediate input, or

**Memory\_Data (DMEM)**

This module comprises our “Memory”. You can load words from memory into registers, or save words (and immediates) into memory. The write is toggled by a MemWrite control. Where something gets input/output from Memory is determined by the address input. WriteData is fed as an input from the ALU\_Out Register.

**Mem\_Data\_Reg**

Serves as an interim register to hold the values coming out of Memory in a process such as LW. The value then gets shuffled on to a Mux to write it as data to a Register. This Register is created more as a placeholder to stall 1 value while the rest of the hardware executes its own processes, in order for the command to continue in sync.

**Instruction\_Memory (IMEM)**

This is the default Instruction Memory module provided. It holds all of our instructions, which are fetched according to a program counter. Instructions begin at 0 and continue until all assigned instructions are executed at which point the last command will continue to loop. Programs 1, 2, and 3 provide a comprehensive test bench for execution of all possible supported commands.

**Instruction\_Splitter (IReg)**

This module takes as input the instruction output from the Instruction Memory and serves to break it up into chunked component outputs. The first 6 bit which are fed to the controller serve the Opcode. The next 5 bits serve as the destination register. The following 5 bits designate a register to look up for a value.

In an R-Type instruction the next 5 bits will be the other register value used for the instruction, whereas in an I-Type instruction the value is a 15 bit immediate. IRWrite is used to toggle, whether the instruction sitting as input is chunked into output data wires, to the rest of the hardware, or made to wait. This is so we don’t start a new instruction before our current instruction has finished.

**Gen\_Purpose\_Reg**

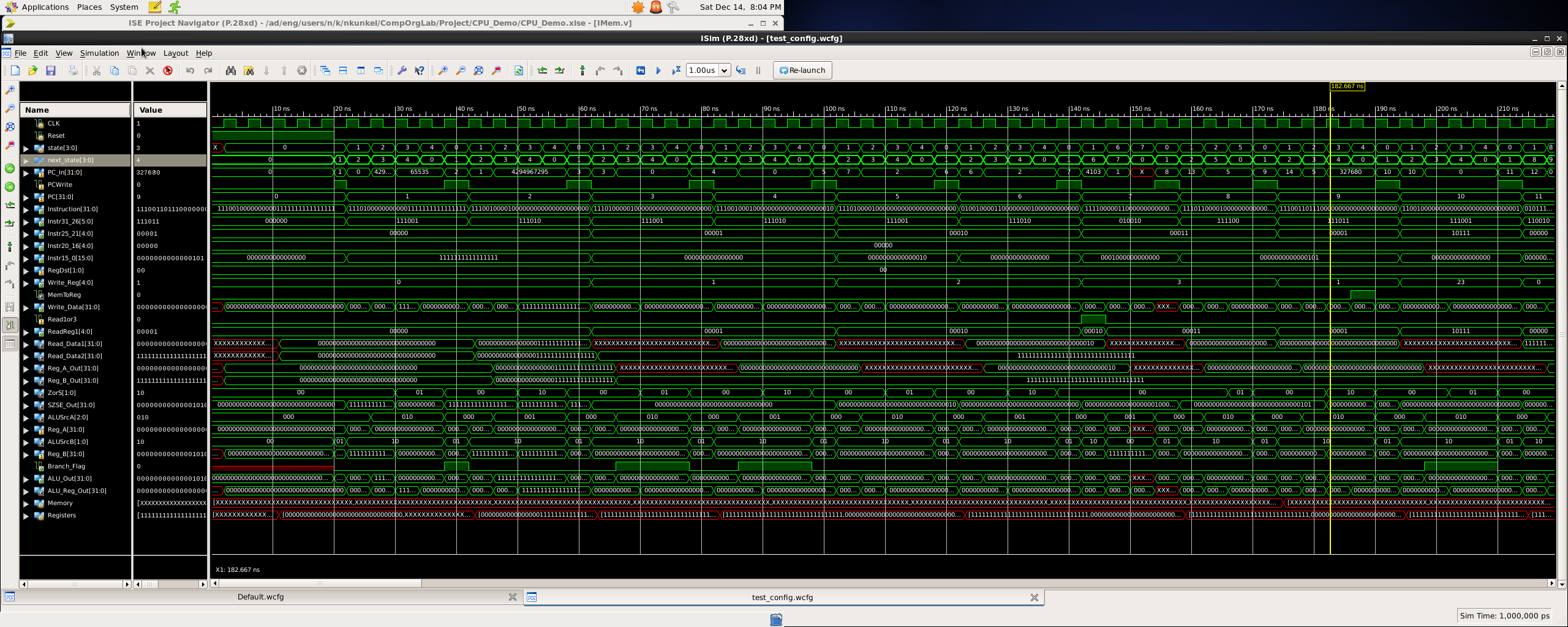
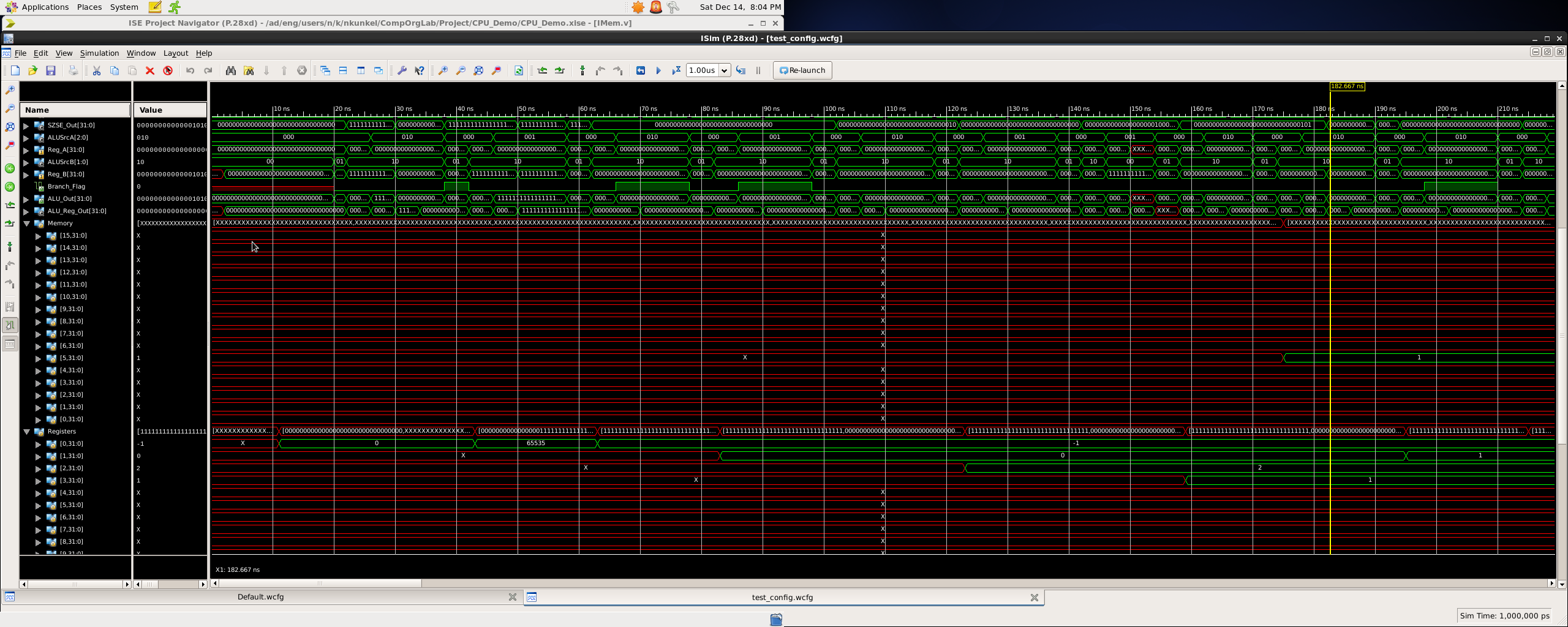
These are our General Purpose Registers which hold all the registers our MIPS commands can assign or grab values from. Registers are labeled R0 through R31.

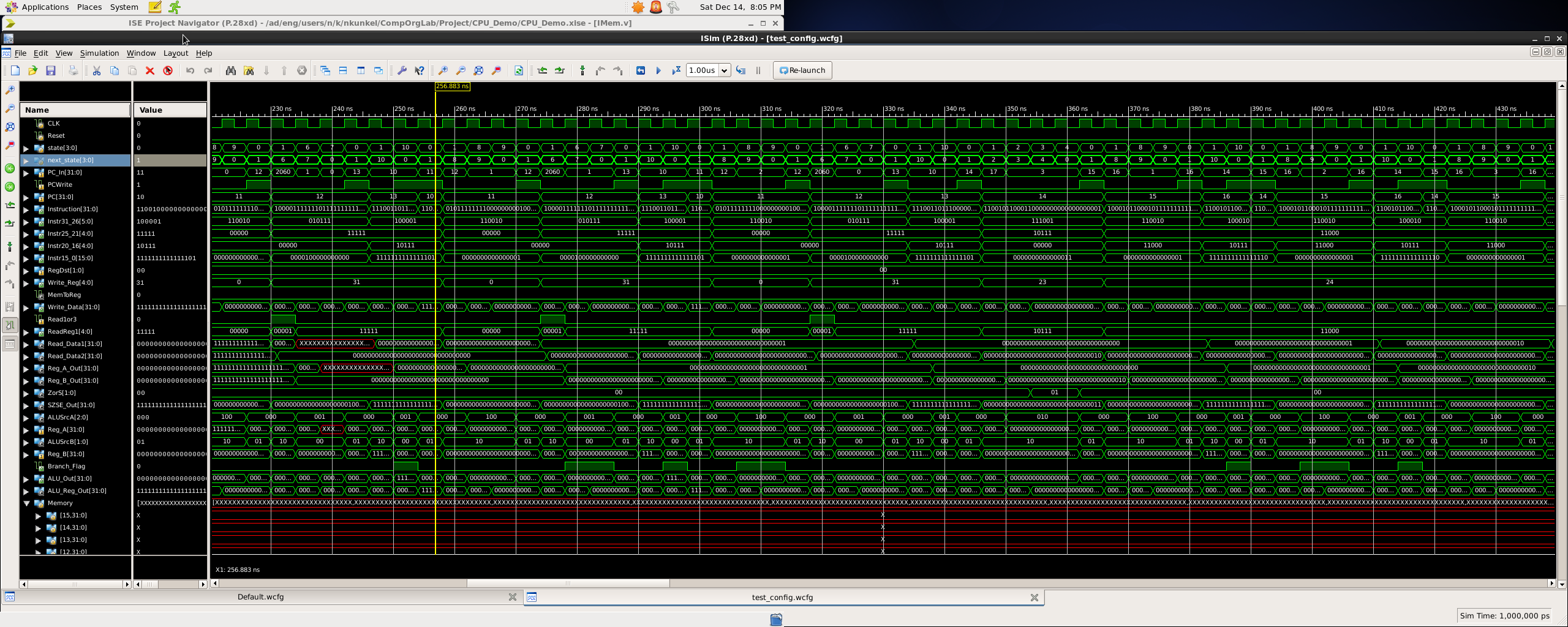
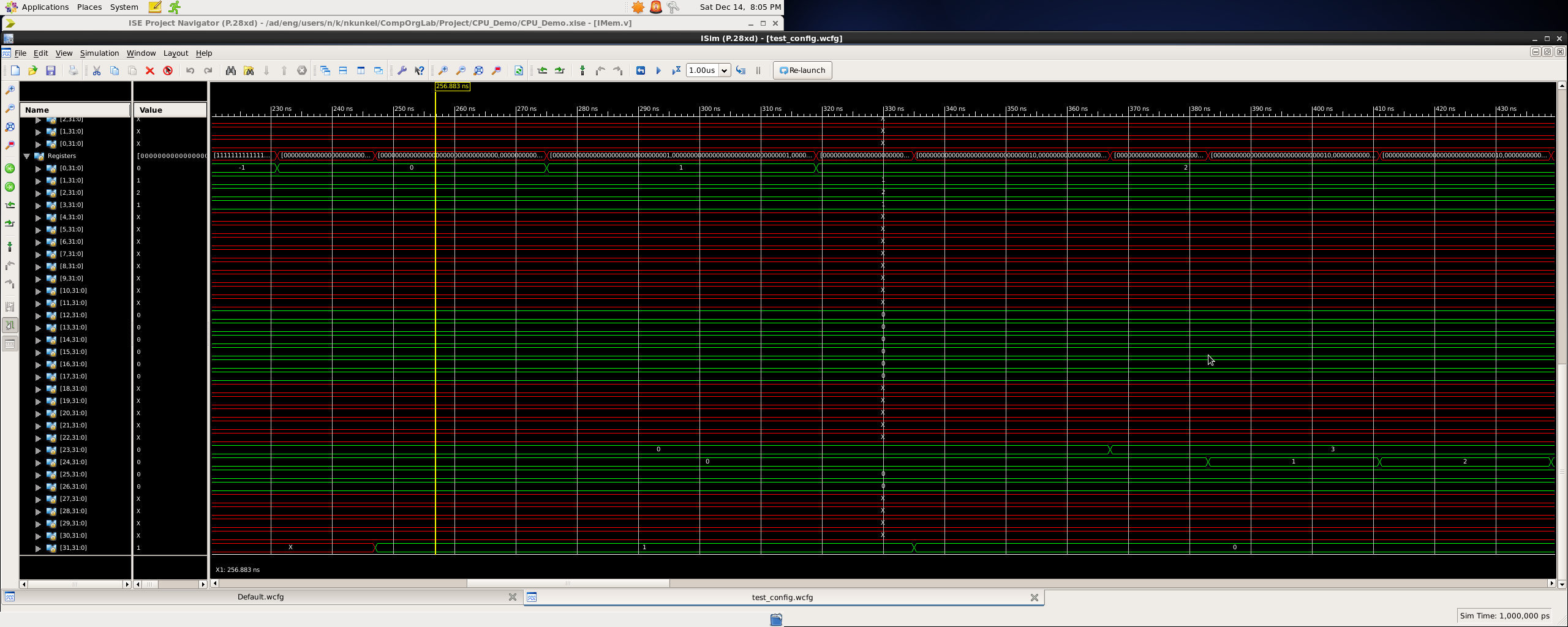
# Instruction List and Corresponding Types

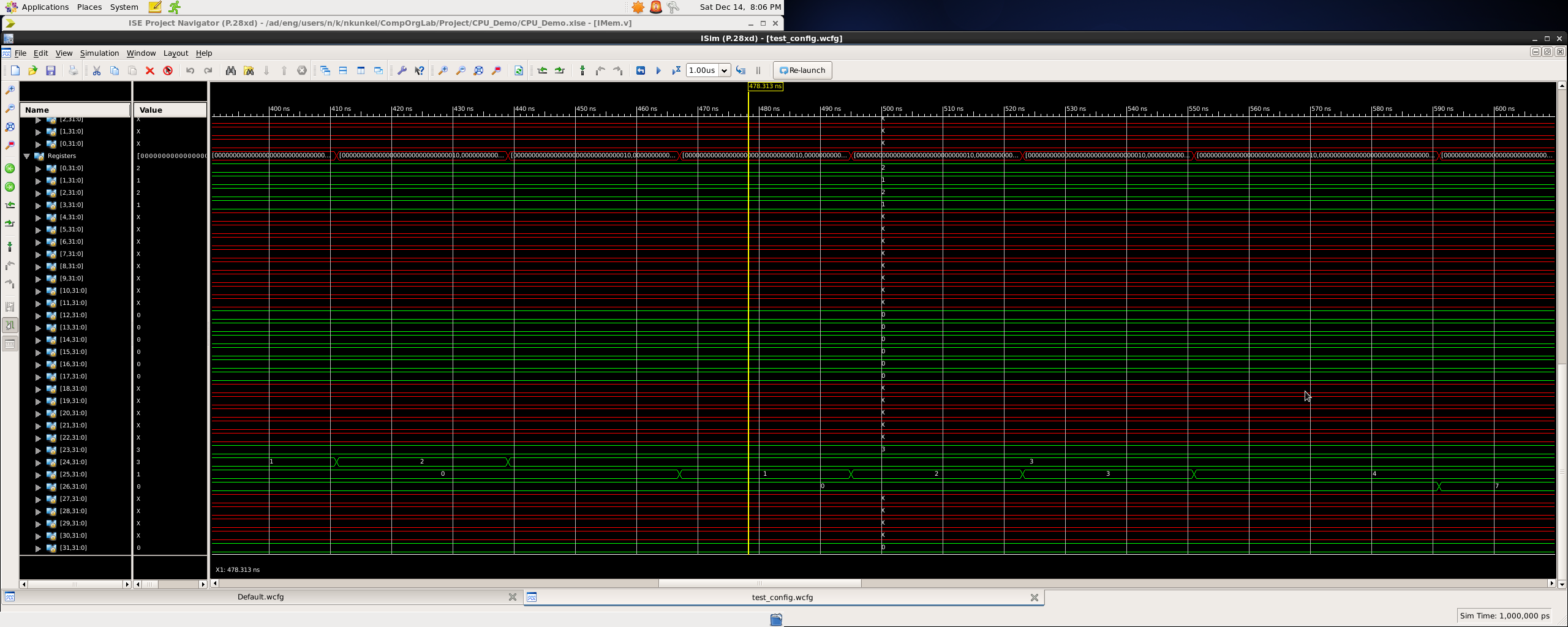
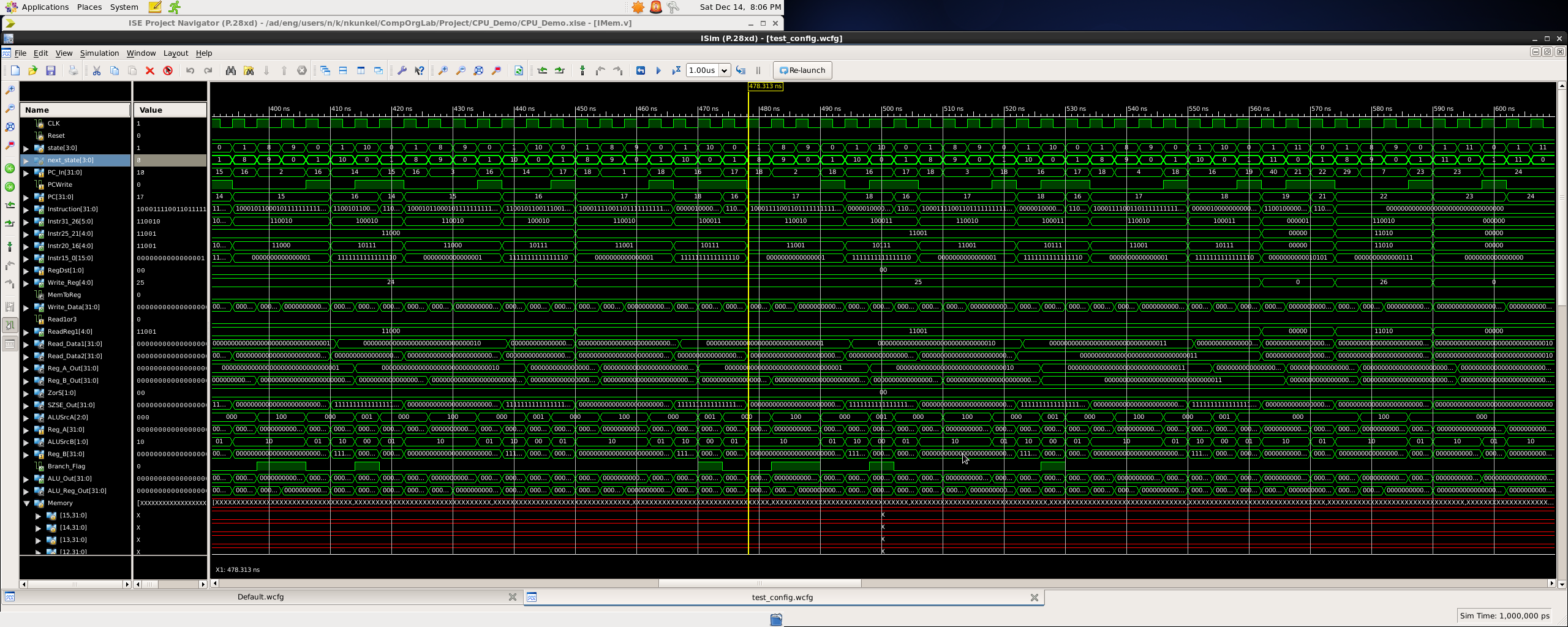
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Op5** | **Op4** | **Op3** | **Op2** | **Op1** | **Op0** | **Output** | **Function Name** | **Type** |
| 0 | 0 | 0 | 0 | 0 | 0 | N/A | **NOOP** | NOOP |
| 0 | 0 | 0 | 0 | 0 | 1 | PC <- PC[31:26] || Limm | **J** | J-Type |
| 0 | 0 | 0 | 0 | 1 | 1 | PC =PC[31:26]; R31= PC+4 | **JAL** | J-Type |
| 0 | 1 | 0 | 0 | 0 | 0 | R1 = R2 | **MOV** | R-type  Logical |
| 0 | 1 | 0 | 0 | 0 | 1 | R1 = ~R2 | **NOT** | R-type  Logical |
| 0 | 1 | 0 | 0 | 1 | 0 | R1 = R2 + R3 | **ADD** | R-type  Arithmetic |
| 0 | 1 | 0 | 0 | 1 | 1 | R1 = R2 - R3 | **SUB** | R-type  Arithmetic |
| 0 | 1 | 0 | 1 | 0 | 0 | R1 = R2 | R3 | **OR** | R-type  Logical |
| 0 | 1 | 0 | 1 | 0 | 1 | R1 = R2 & R3 | **AND** | R-type  Logical |
| 0 | 1 | 0 | 1 | 1 | 0 | R1 = R2 ^ R3 | **XOR** | R-type  Logical |
| 0 | 1 | 0 | 1 | 1 | 1 | R1 = 1 if R2 < R3,  else 0 | **SLT** | R-Type  Arithmetic |
| 1 | 0 | 0 | 0 | 0 | 0 | IF(R1==R2)  Then PC <- PC+SE(Imm)  Else PC <- PC + 1 | **BEQ** | Branch Type (I) Arithmetic |
| 1 | 0 | 0 | 0 | 0 | 1 | IF(R1 != R2)  Then PC <- PC+SE(Imm)  Else PC <- PC + 1 | **BNE** | Branch Type (I)  Arithmetic |
| 1 | 0 | 0 | 0 | 1 | 0 | IF(R1 < R2)`  Then PC <- PC+SE(Imm)  Else PC <- PC + 1 | **BLT** | Branch Type (I)  Arithmetic |
| 1 | 0 | 0 | 0 | 1 | 1 | IF(R1 <= R2)  Then PC <- PC+SE(Imm)  Else PC <- PC + 1 | **BLE** | Branch Type (I)  Arithmetic |
| 1 | 1 | 0 | 0 | 1 | 0 | R1 = R2 + SE(Imm) | **ADDI** | I-Type  Arithmetic |
| 1 | 1 | 0 | 0 | 1 | 1 | R1 = R2 - SE(Imm) | **SUBI** | I-Type  Arithmetic |
| 1 | 1 | 0 | 1 | 0 | 0 | R1 = R2 | ZE(Imm) | **ORI** | I-Type  Logical |
| 1 | 1 | 0 | 1 | 0 | 1 | R1 = R2 & ZE(Imm) | **ANDI** | I-Type  Logical |
| 1 | 1 | 0 | 1 | 1 | 0 | R1 = R2 ^ ZE(Imm) | **XORI** | I-Type  Logical |
| 1 | 1 | 0 | 1 | 1 | 1 | R1 = 1 if R2 < SE(Imm)  Else 0 | **SLTI** | I-Type  Arithmetic |
| 1 | 1 | 1 | 0 | 0 | 1 | R1[15:0] = ZE(Imm) | **LI** | I-Type  Logical |
| 1 | 1 | 1 | 0 | 1 | 0 | R1[31:16] = ZE(Imm) | **LUI** | I-Type  Logical |
| 1 | 1 | 1 | 0 | 1 | 1 | R1 = M[ZE(Imm)] | **LWI** | I-Type  Logical |
| 1 | 1 | 1 | 1 | 0 | 0 | M[ZE(Imm)] = R1 | **SWI** | I-Type  Logical |
| 1 | 1 | 1 | 1 | 0 | 1 | R1 = M[R2+C] | **LW** | I-Type |
| 1 | 1 | 1 | 1 | 1 | 0 | M[R2+C] = R1 | **SW** | I-Type |

# Wavelengths of Testbench:

**Program 1**

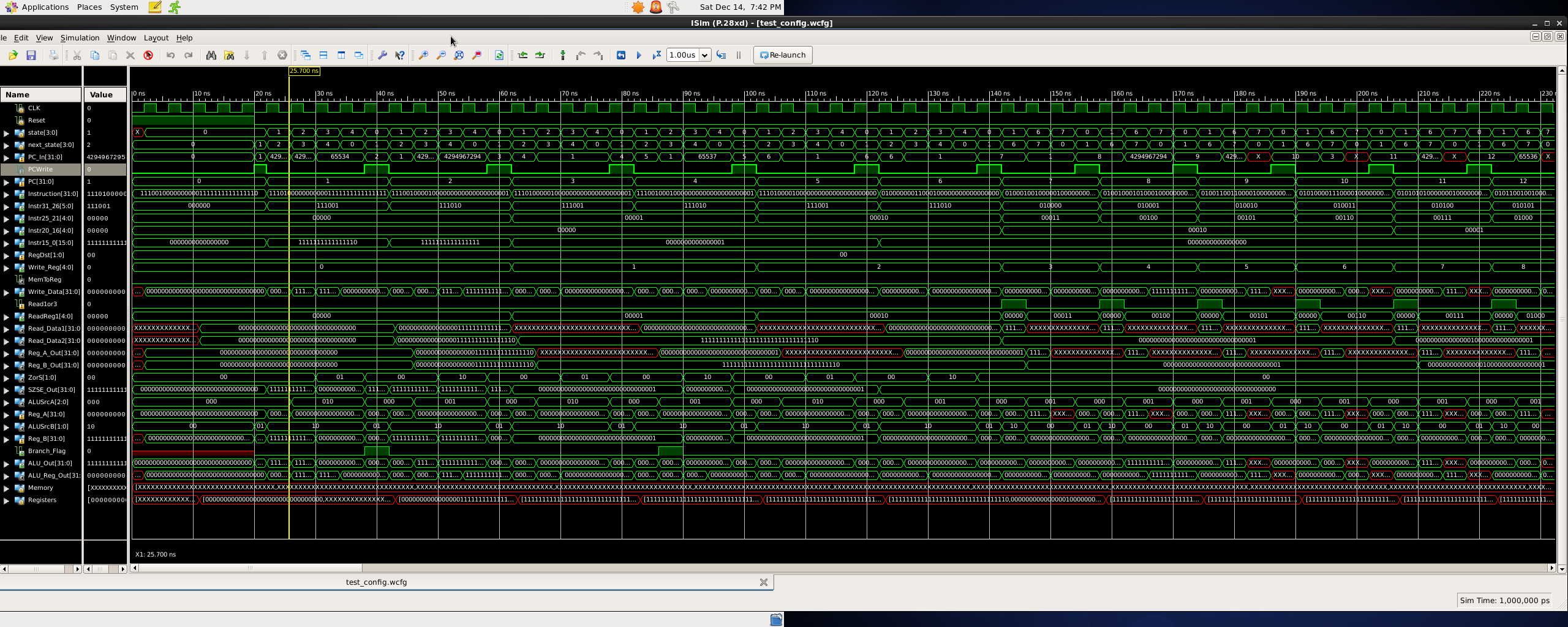
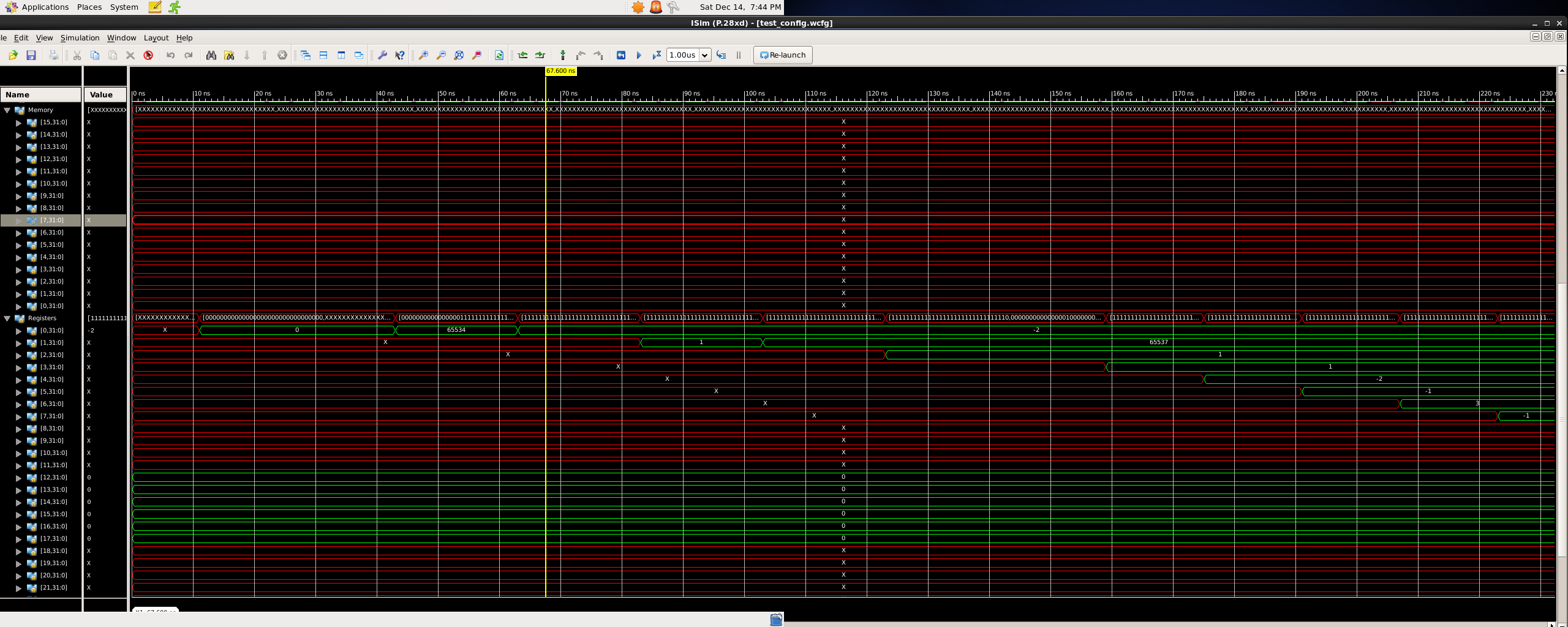
****

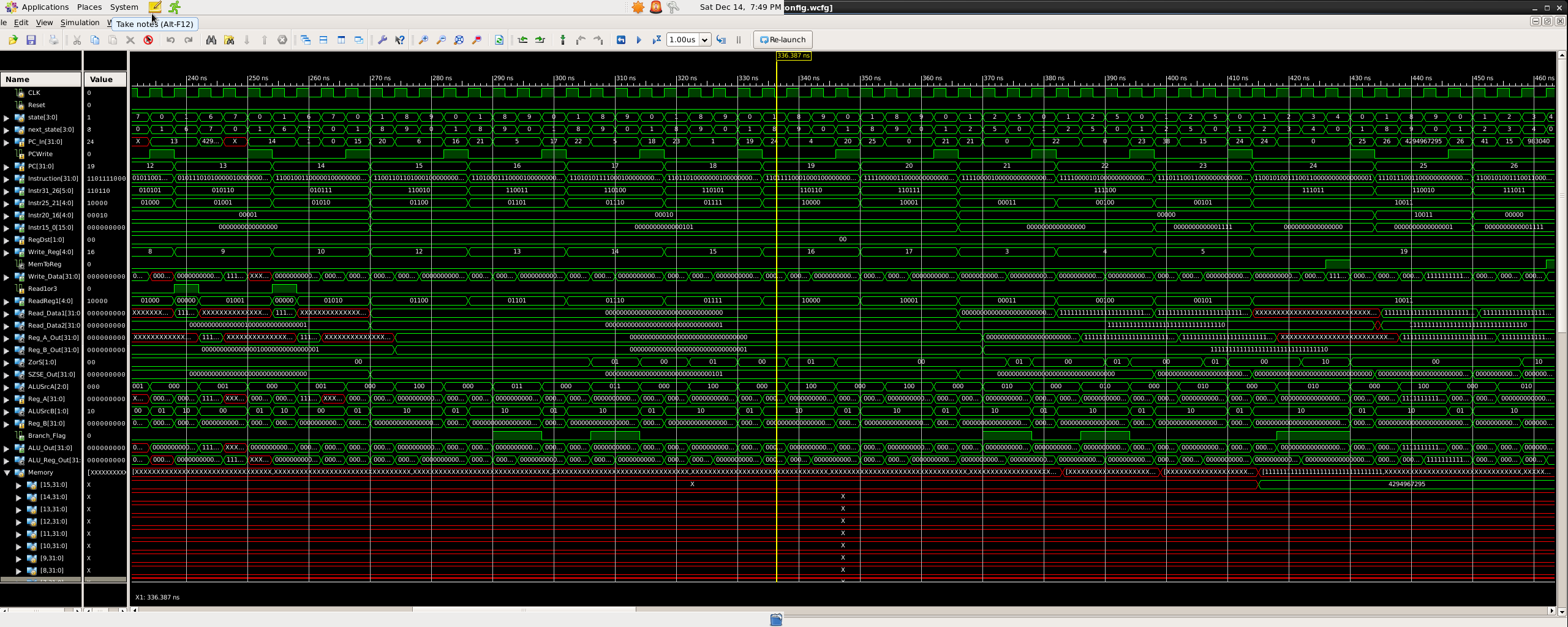
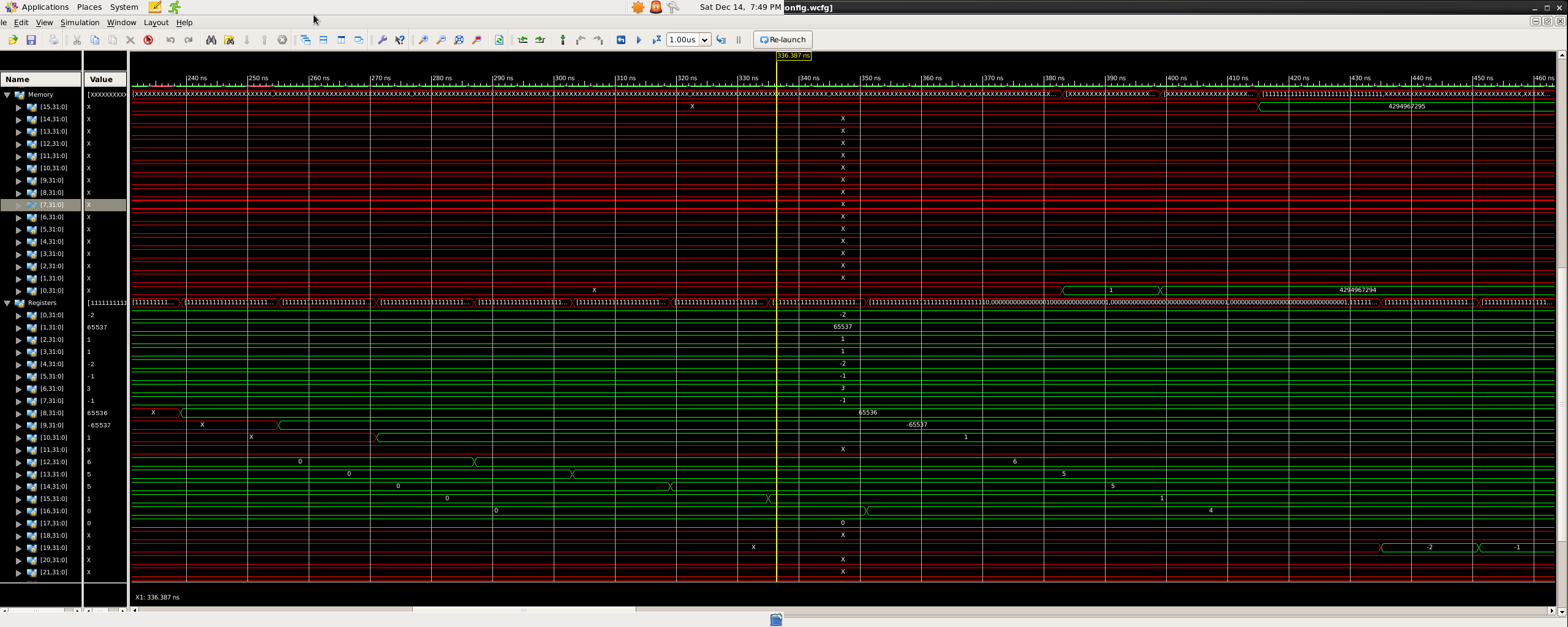




**Program 2**

**Program 2**

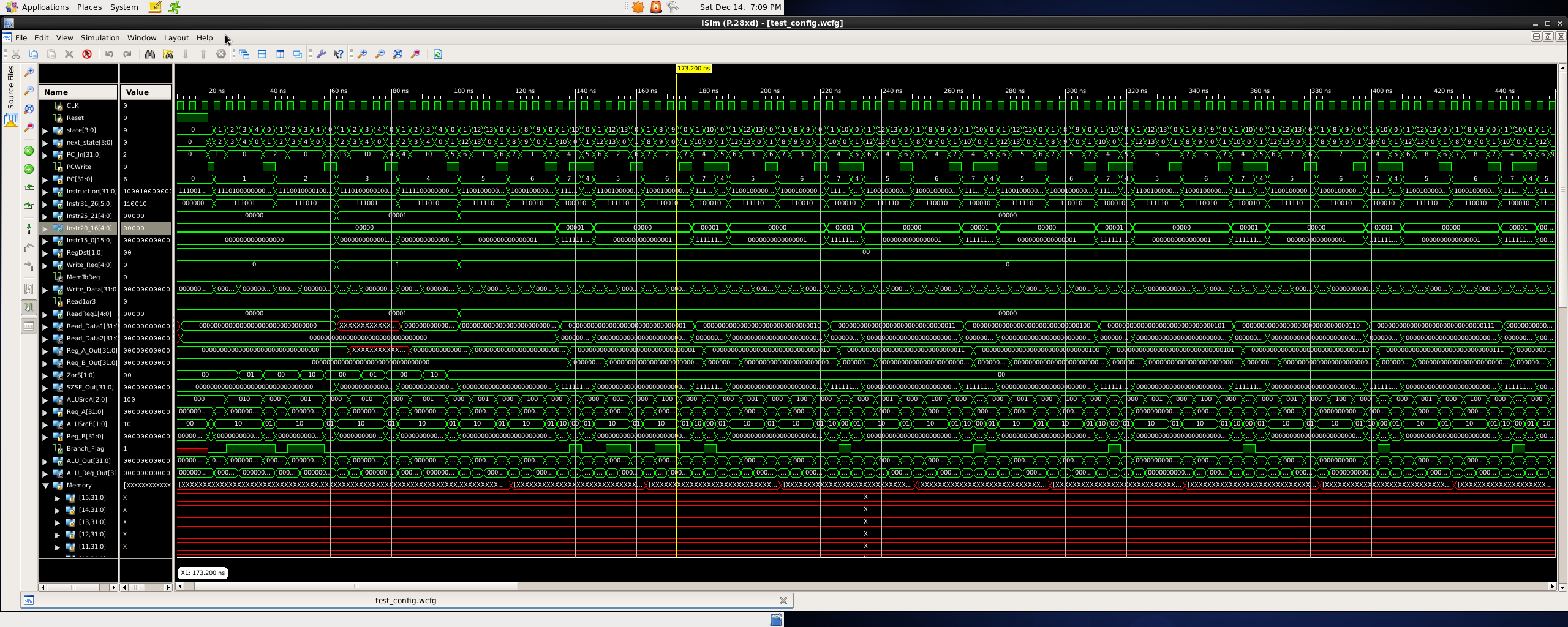


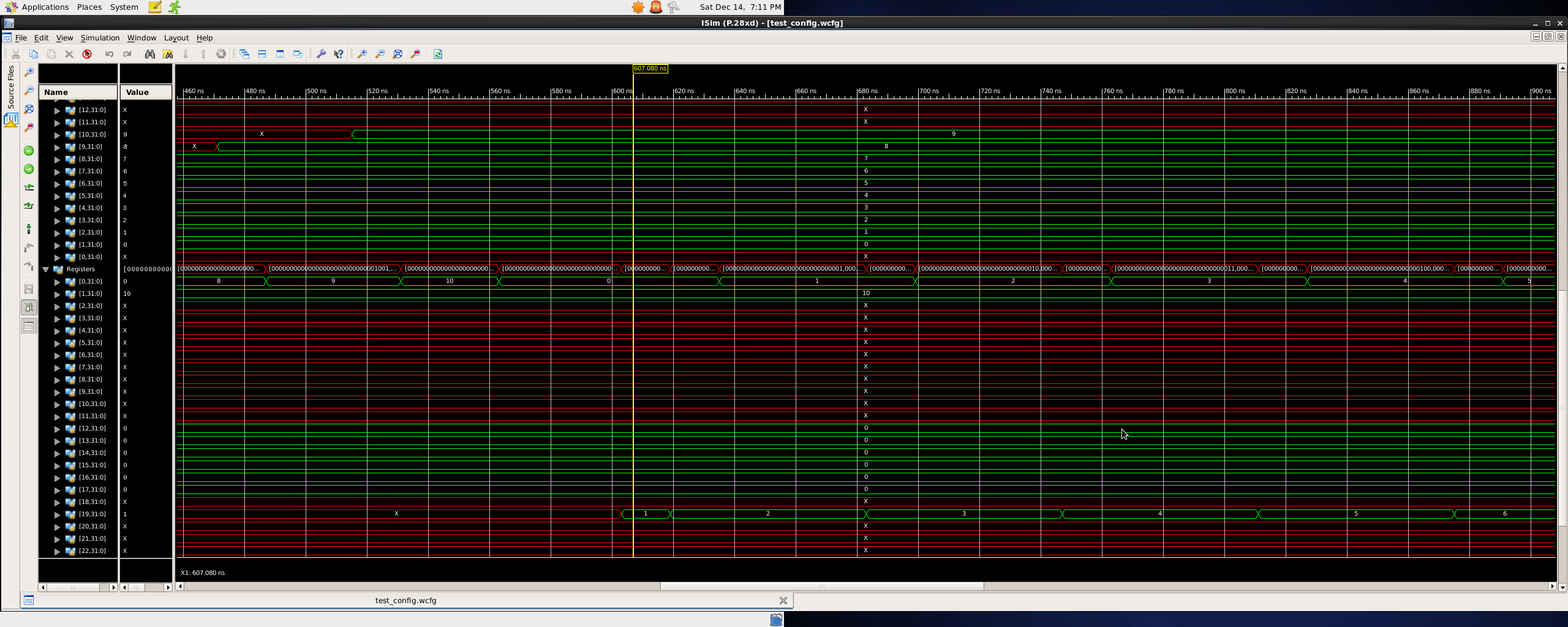
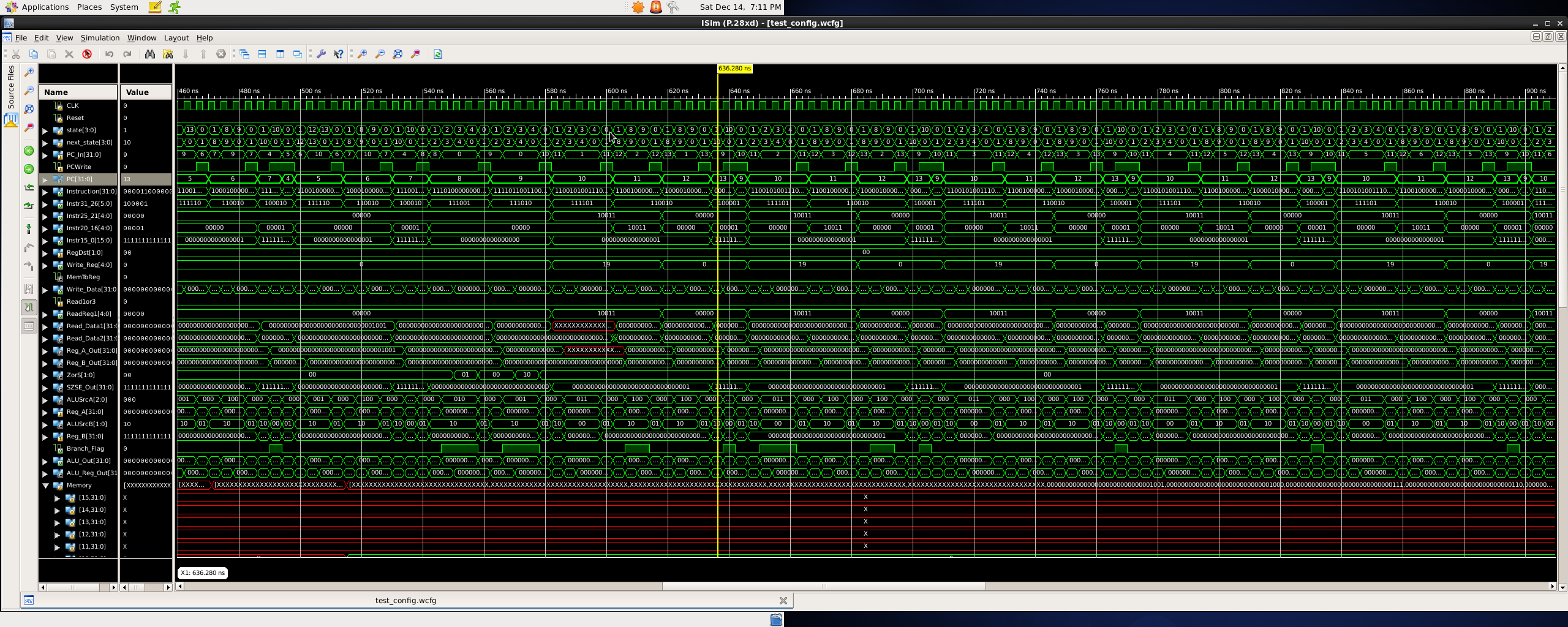


**Program 3 w/ JAL instruction appending to end**

- no memory/reg needed

for this instruction set for this range





\*Might not to be too legible in document format. Attached are all screenshots in .png format.

Naming convention = P#\_###\_###\_Empty/Mem

Where:

P# stands for Programm # (1,2,3) being tested

### stands for wavelength time at beginning

### stands for wavelength time at end

Empty/Mem –

Empty stands for datapath values

Mem stands for Memory and Register Values