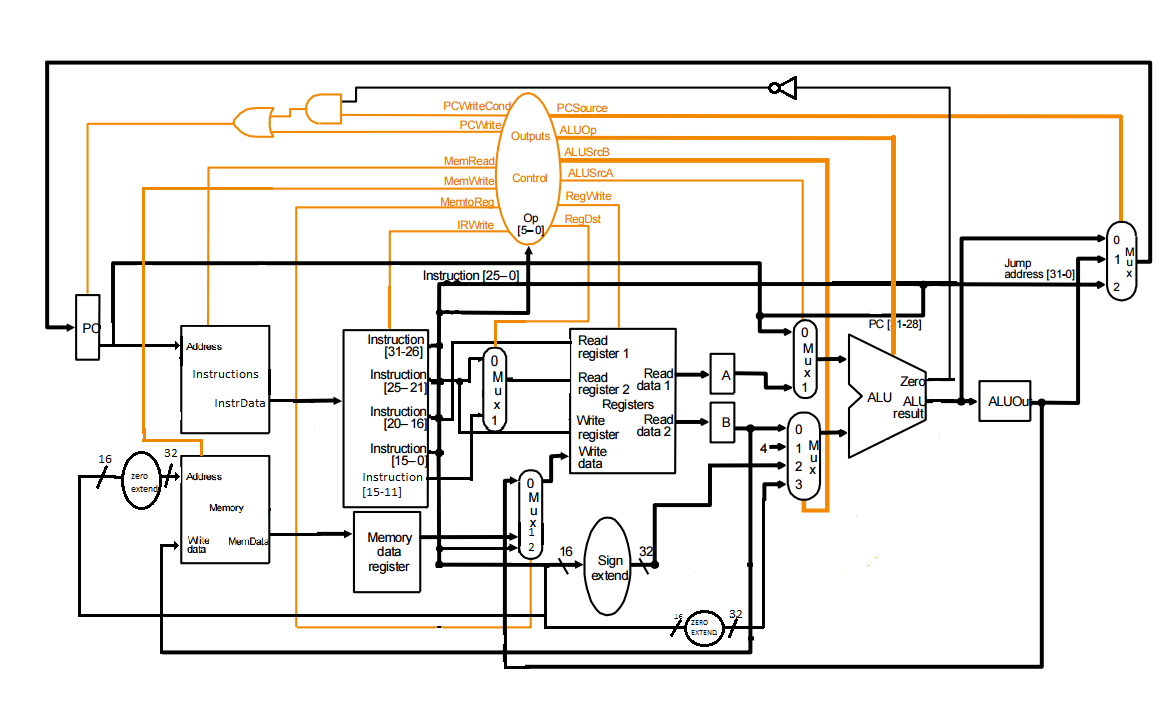
Allison Durkan and Hannah Hebert

EC413

**Final Project: MIPS Mulit-cycle CPU Design**

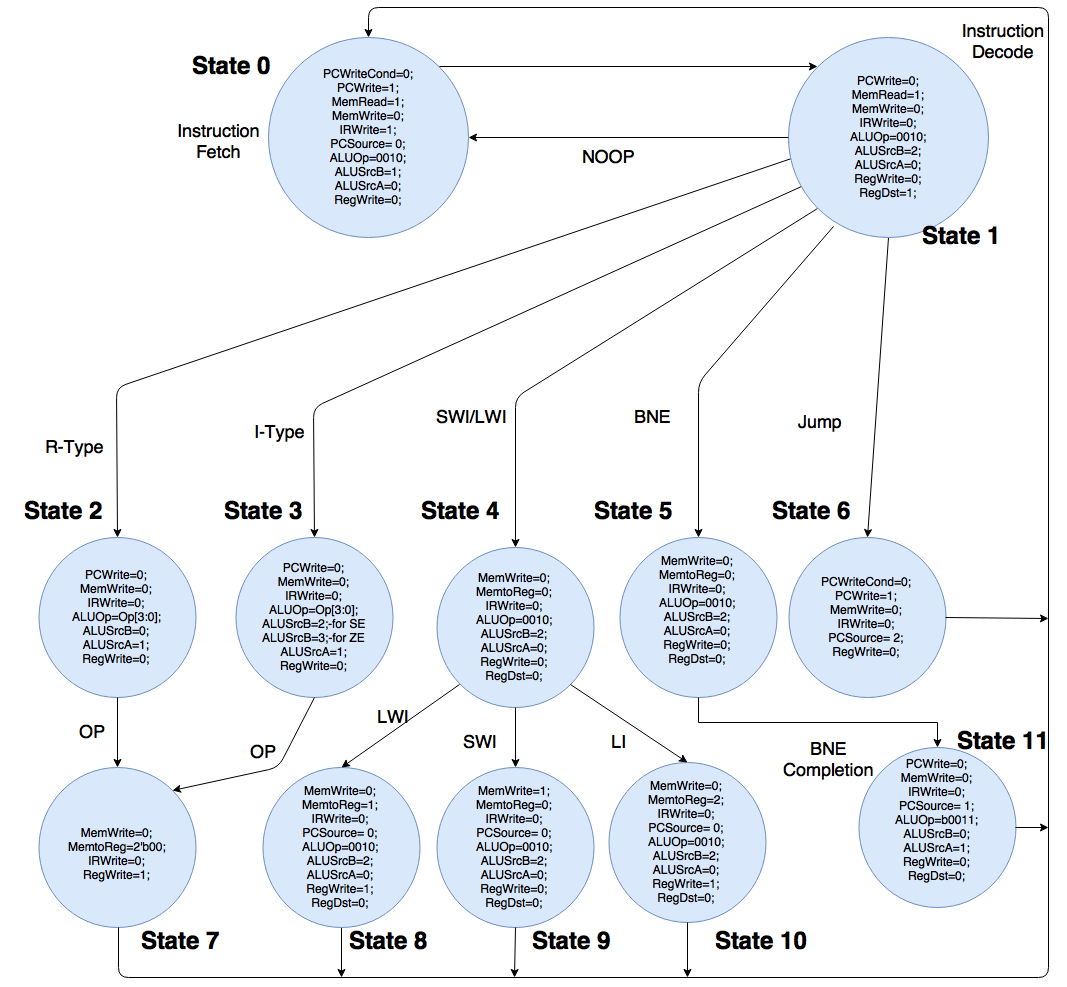
To implement a 32-bit multicycle CPU we designed both a datapath and a control that supports a set of MIPS assembly instructions. The finalized design is shown below.

**Datapath**

The datapath is the collection of hardware needed to implement the MIPS instructions. As instructions are executed, they move through a series of registers and multiplexor selections in order to read/write to both memory and the register file. Our datapath supports R-type and I-type ALU instructions as well as loading immediates, storing immediates, a conditional branch, and a jump instruction. The finalized design for the hardware path is shown below.

**Controller**

The controller sets a series of enable/disable signals as well as multiplexor selections in defined states that allows the instructions to move through the datapath. In order to implement this, we began with a state diagram shown below.

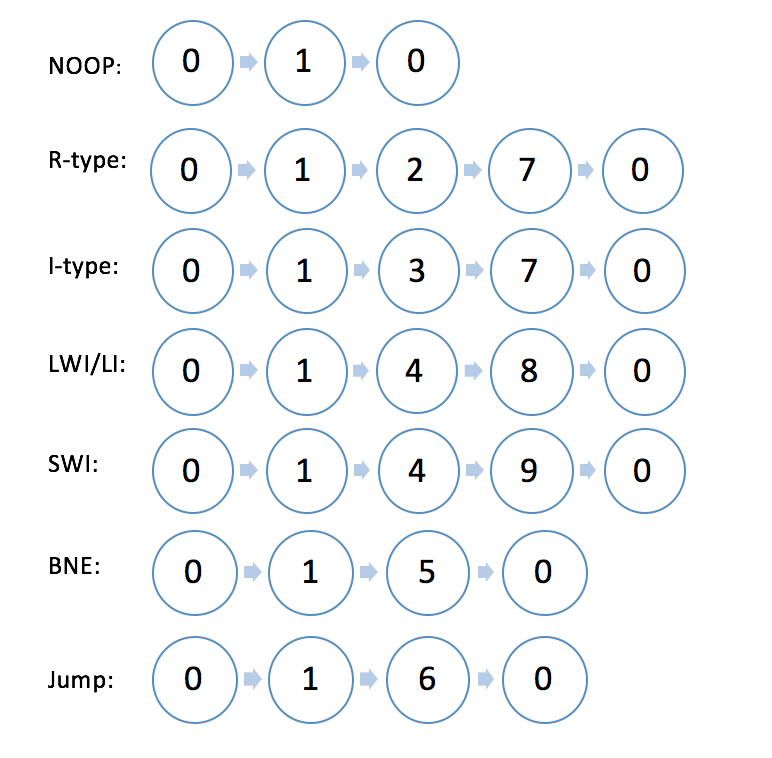


In our controller, we use a switch statement to move through states that are used to set the value of the control logic lines. In each state, the values from the control lines are set to correspond with what our system is using. Then, the next state variable we use is changed, and the overarching switch case is gone through again. At the top of that case statement, the state is set to the next state value, which allows us to keep track of the state of the machine.

The control lines that go into multiplexers include RegDst, MemToReg, ALUSrcA, ALUSrcB, ALUOp, and PCSrc. RegDst is set to 0 or 1 based on whether read register 2 will be either R1 or R3, respectively. MemToReg decides if the write data input is from register B or from data memory. ALUSrcA either selects the program counter or read data 1. ALUSrcB, in order from 0 to 3 based on the select, selects register b, the value 4 to increment the program counter, the sign extended immediate, or the zero extended immediate value. ALUOp is a 4-bit select based on the opcodes given to us in the project documentation. PCSrc is either the ALU output (the incremented program counter), the value of the ALU register, or the 32-bit address for a jump instruction, with selects in that order.

The other control lines not described previously are in charge of enabling reading and writing from the different registers in the datapath. The names of those control lines describe what they enable.

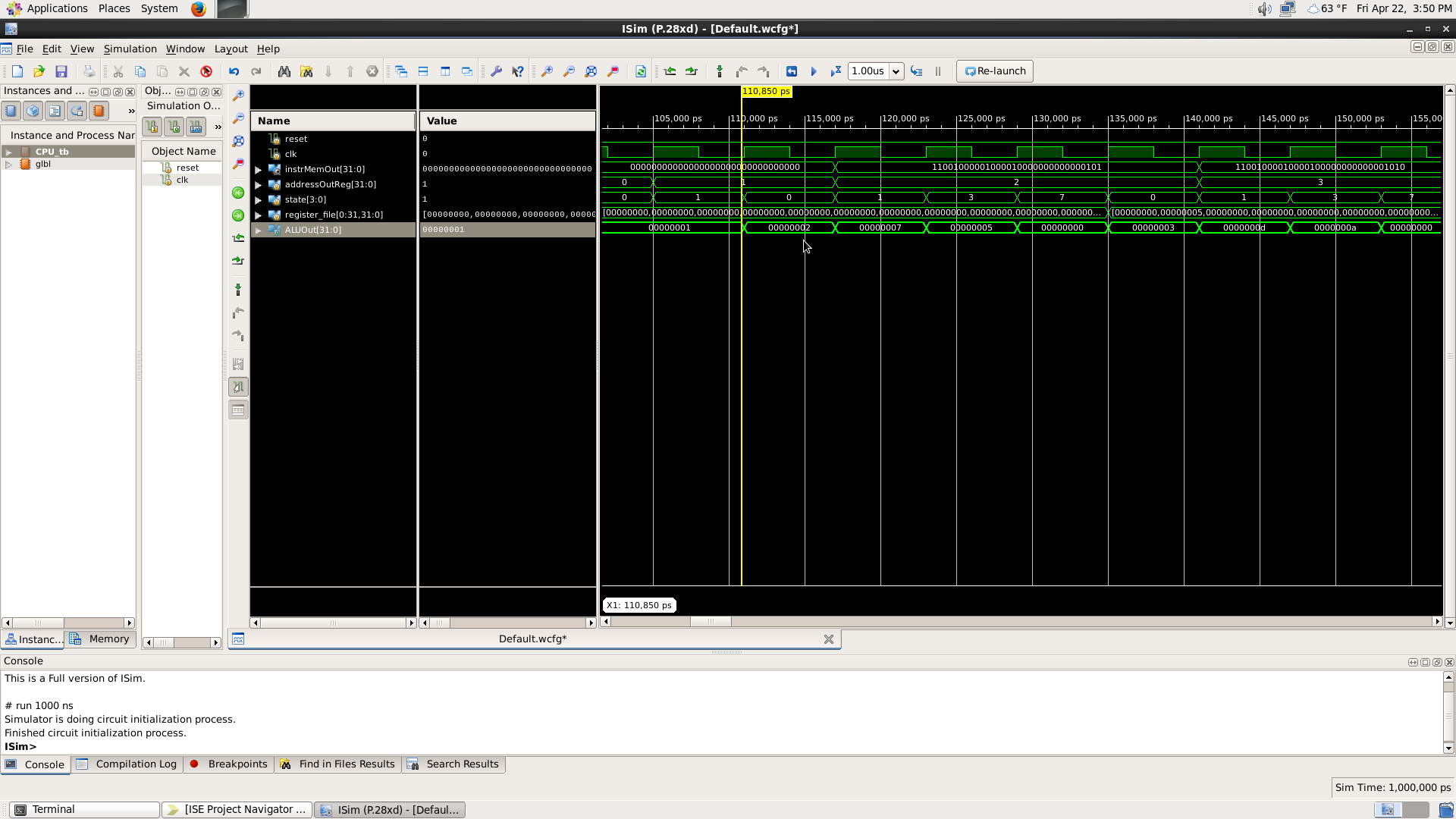
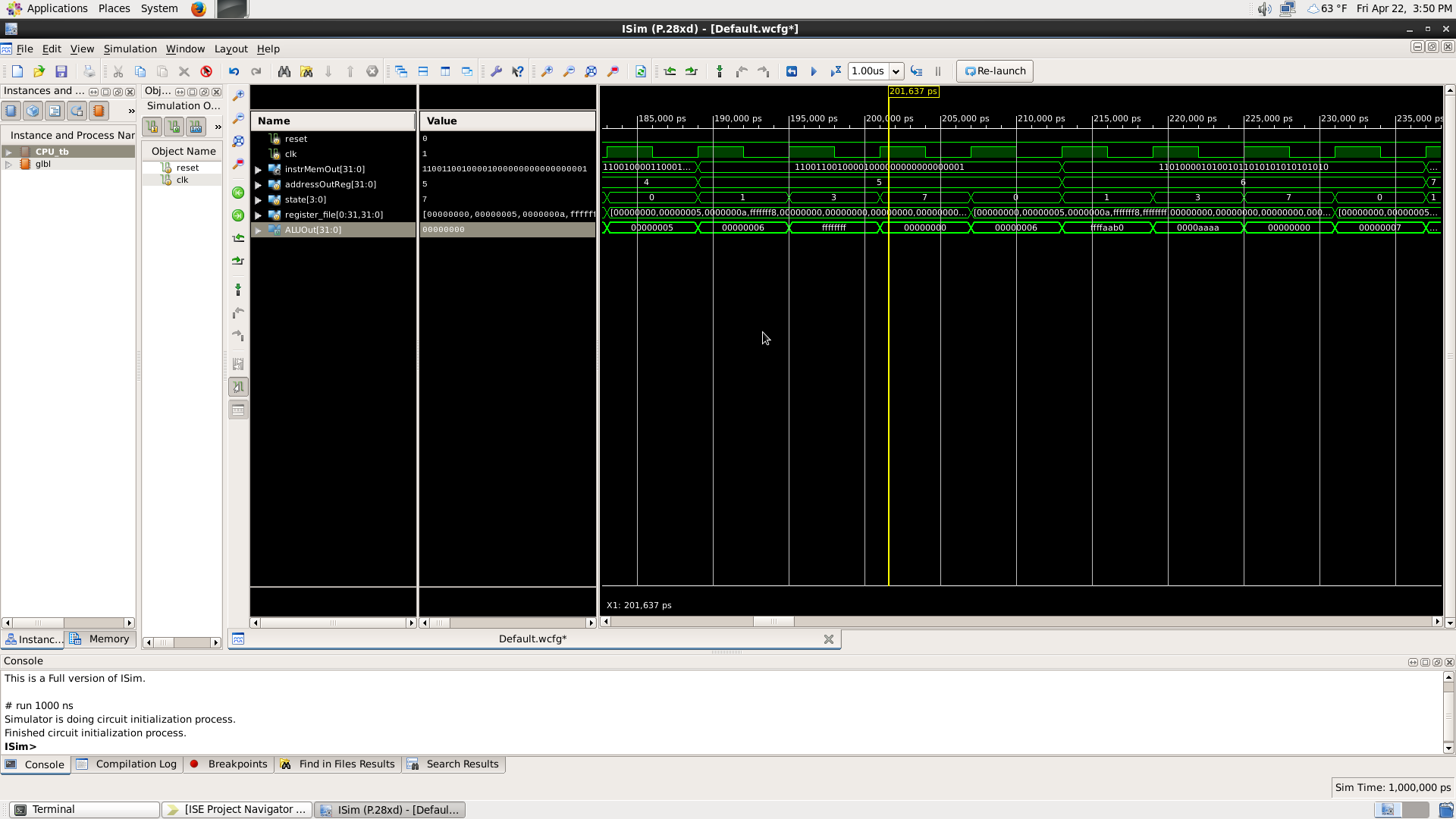
For each instruction, a specified state path was created. The state path and number of cycles required of each instruction is shown below.



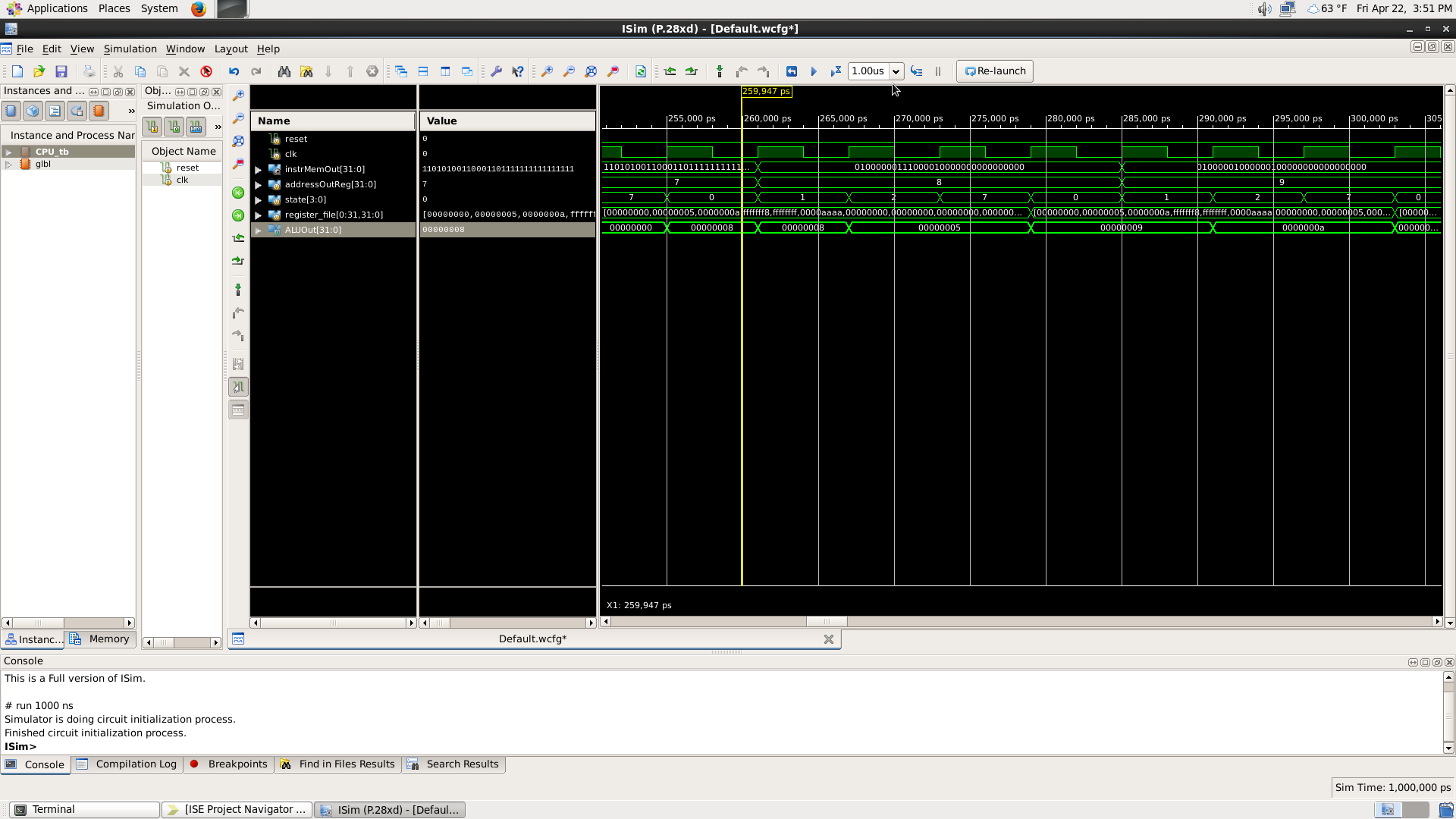
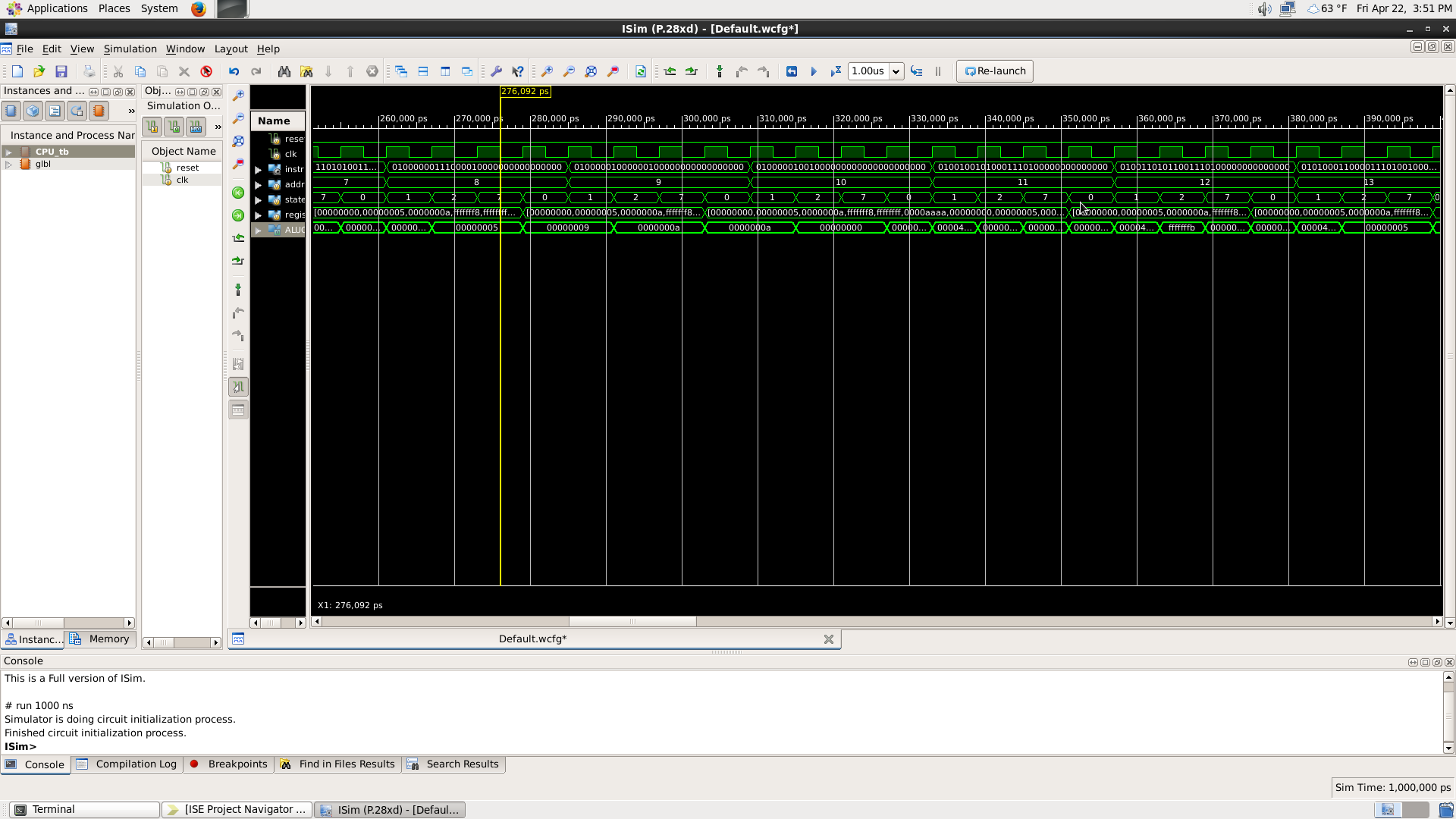
**Verilog Testing/Impelmentation**

To incorporate both the control module and the datapath module into one full working system, we made instances of both modules in our CPU module, which only had inputs that were the clock and reset lines. The wires inside that module connected the instances of the control and datapath in order for the two to work concurrently. Our final schematic, state diagram, control line table, waveforms and state transitions are depicted below.

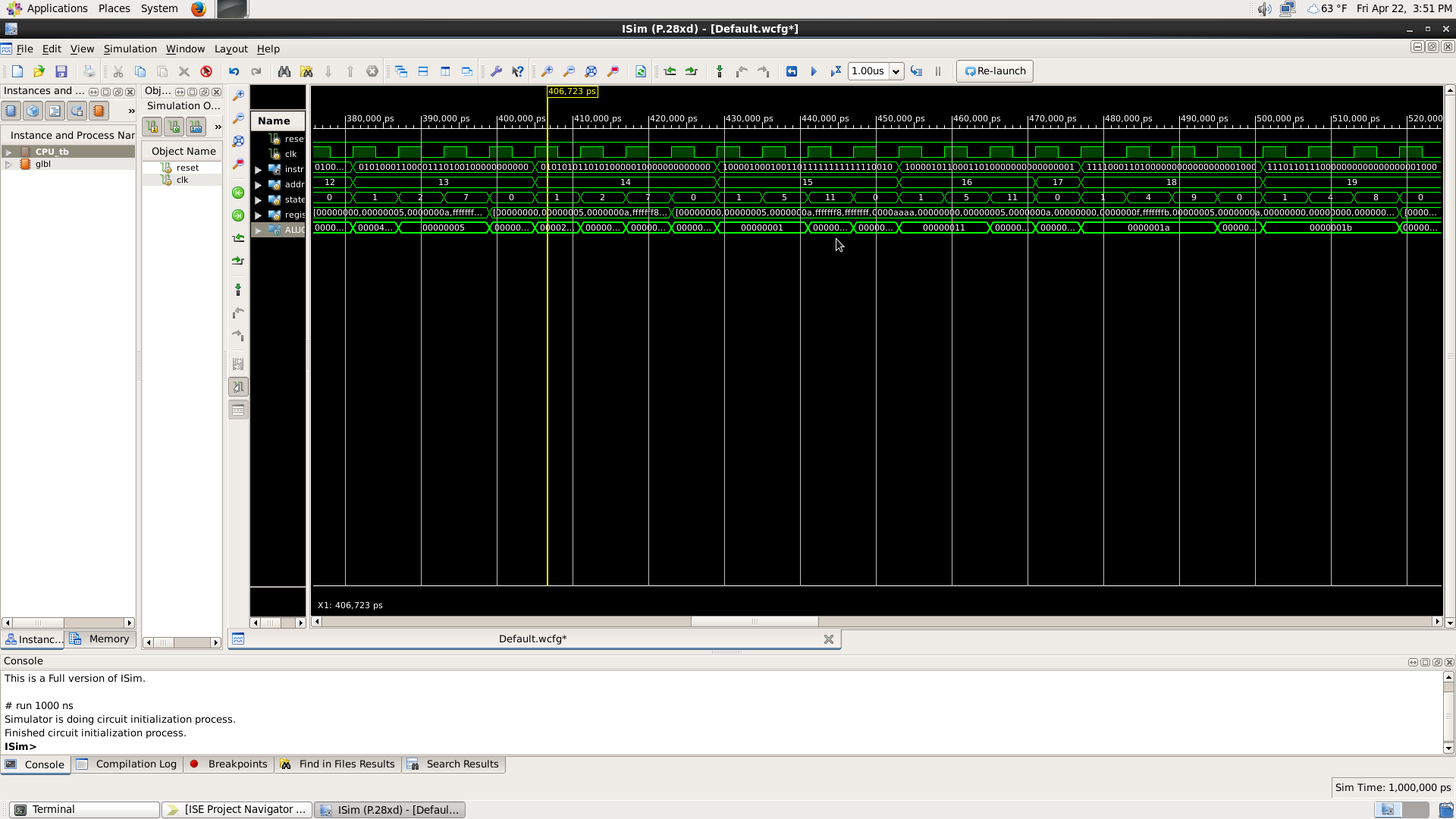
In our CPU testbench, instructions are executed based on the IMem.v file following an initial sytem reset. The resulting waveforms are shown below. The waveforms shown include the current instruction being executed, the program counter (instrMemOut), the state of the CPU, the result of the ALU, and the register file. The value of all these signals are updated on the positive edge of the input clock cycle.

I-type instructions(PC=1🡪PC=6):

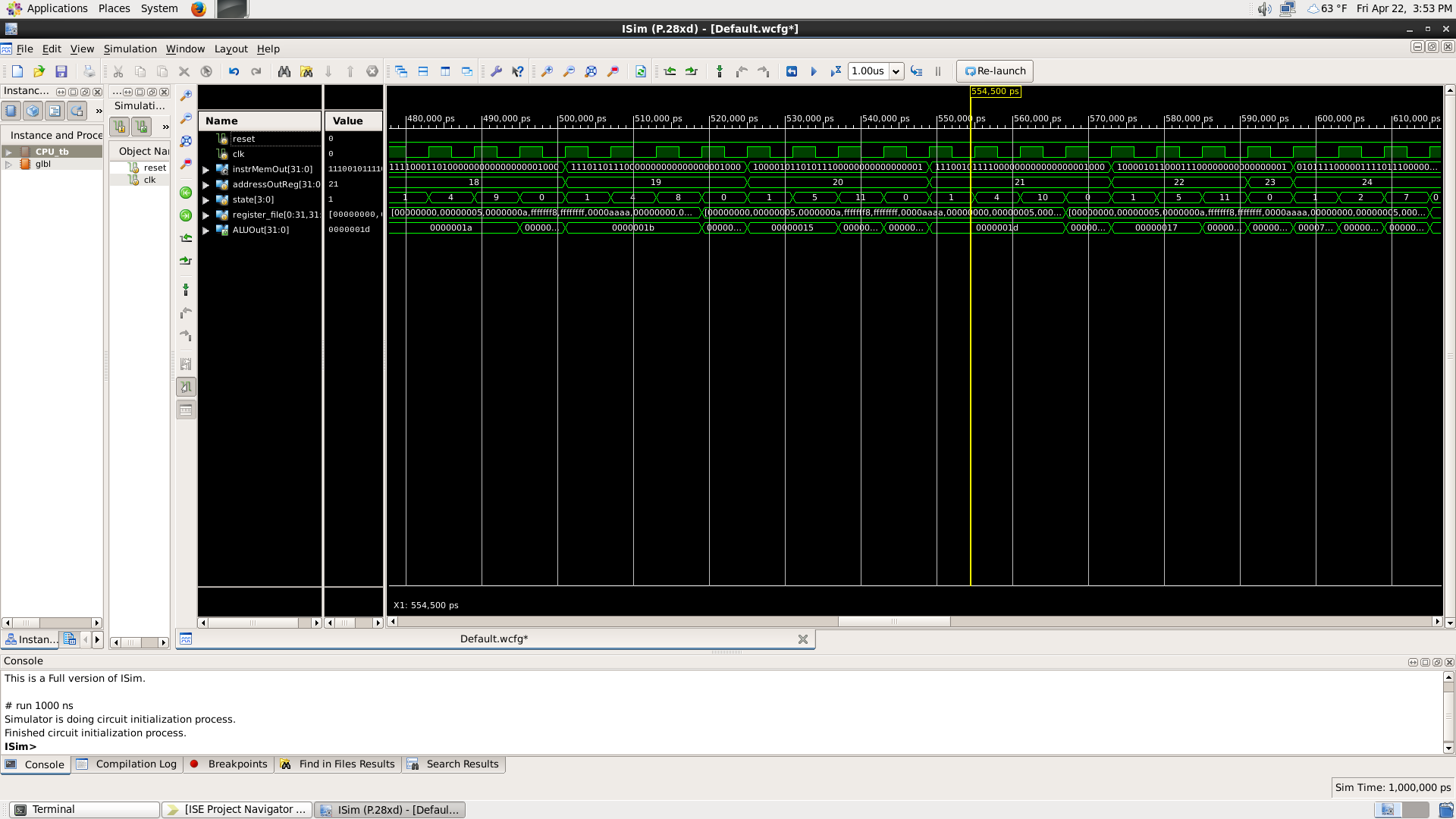
As the program counter is increased, the instruction changes and the register file is updated accordingly.

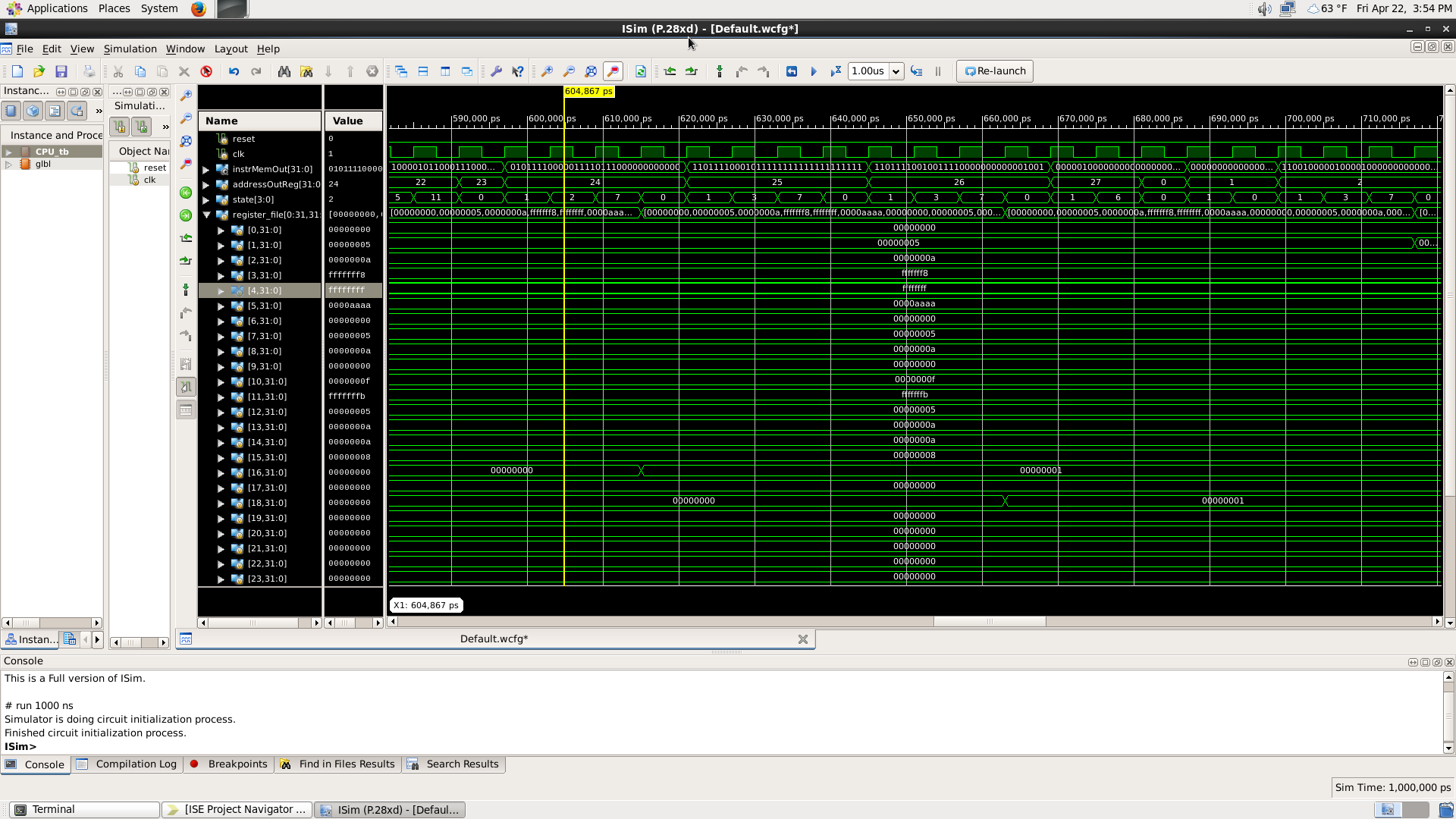
R-type instructions (PC=7🡪PC=13):

As the program counter is increased, the instruction changes and the register file is updated accordingly.

BNE, SWI and LWI instructions:

In the PC=14 instance of BNE, the program does not branch and moves to instruction 15. In the PC=15 instance of BNE, the program branches and moves to instruction 17 (SWI).

LI, BNE instructions:

SLTI and Jump instructions:

When the instruction hits the jump instruction, the program restarts at PC=0. In this waveform it is also clear that the register file has been updated correctly.

**Notable Changes from Original Design**

Between the original designs and the final working system, we made some drastic changes to our state machine. Firstly, we combined the SWI and LWI original state. From instruction decode both instructions go to our State 4 which branches into two separate states based on whether it is a load or store instruction. Secondly, we combined the operation state of I and R type instructions, meaning that I and R types load the data into the register file and differently, but the actual operation occurs in the same way for both types, in State 7. We also fine-tuned the control select lines for the BNE and Jump instructions.

Some changes we made in our hardware include adding an inverter to the zero line from the ALU, which was important for our BNE logic. We also added a zero extend from the output line of the instruction register into the Source B multiplexer, in order to zero extend immediate values so we would not have issues with the values in our ALU. Finally, we extended the multiplexer going into the Write Data port on the register file, with the third option being the immediate value. This required us to extend the MemtoReg control line by one bit.