Description: https://lh4.googleusercontent.com/bpfGlrXcwL2InVVTEmmoD_RjmplIRjQ_qTei0xro8dNAn1_LWfEzxzwbA6Ph-12qiEAOhlUhNU8lPPq-x6jsbRG0r59XzrcxgbNNAZbAQvOMofzDXY8UgF2dYyKOfh8XmWHIBgv7CYrN-Z6srCJdzCEa-Cl_sPh8A67eEpwJKJBpnFroi2ANV9WYy4jU1nQkymy8v_s

**“Final PROJECT”**

**COURSE :**

**Computer Architecture And**

**Logic Design**

**SUBMITTED TO :**

**DR.Irum Matloob**

**SUBMITTED BY :**

**Rabia Batool (2021-BSE-064)**

**Sawaira Saeed(2021-BSE-067)**

**Hafsah Tahir(2021-BSE-067)**

**SECTION :**

**B**

[1. Installation: 3](#_Toc156158485)

[2. Arithmetic logic unit (ALU): 7](#_Toc156158486)

[Program Explanation: 7](#_Toc156158487)

[FOR AND: 8](#_Toc156158488)

[For OR : 8](#_Toc156158489)

[For Add: 9](#_Toc156158490)

[For XOR: 10](#_Toc156158491)

[NOR: 11](#_Toc156158492)

[For Sub: 12](#_Toc156158493)

[Set less than: 13](#_Toc156158494)

[Shift left: 14](#_Toc156158495)

[Shift right: 15](#_Toc156158496)

[Shift right Arithmetic: 16](#_Toc156158497)

[Signed add: 17](#_Toc156158498)

[Signed sub: 18](#_Toc156158499)

[3. Mux : 19](#_Toc156158500)

[2-to-1 multiplexer: 19](#_Toc156158501)

[Program Explanation: 19](#_Toc156158502)

[Set bit 0: 20](#_Toc156158503)

[Set bit 1: 21](#_Toc156158504)

[4. Sign extension: 21](#_Toc156158505)

[Program Explanation: 21](#_Toc156158506)

[Simulation: 22](#_Toc156158507)

[5. Instruction memory: 22](#_Toc156158508)

[Program Explanation: 22](#_Toc156158509)

[6. Control unit: 24](#_Toc156158510)

[Program Explanation: 24](#_Toc156158511)

[I type 25](#_Toc156158512)

[Jump: 25](#_Toc156158513)

[OPCODE for instructions 25](#_Toc156158514)

[Load: 26](#_Toc156158515)

[Store: 27](#_Toc156158516)

[Jump: 27](#_Toc156158517)

[7. Data memory: 27](#_Toc156158518)

[Program Explanation: 27](#_Toc156158519)

[Simulation: 28](#_Toc156158520)

[8. Program counter: 29](#_Toc156158521)

[Program Explanation: 29](#_Toc156158522)

[Simulation: 29](#_Toc156158523)

[9. Register file: 29](#_Toc156158524)

[Program Explanation: 29](#_Toc156158525)

[Simulation: 30](#_Toc156158526)

[10. CPU : 31](#_Toc156158527)

[Program Explanation: 31](#_Toc156158528)

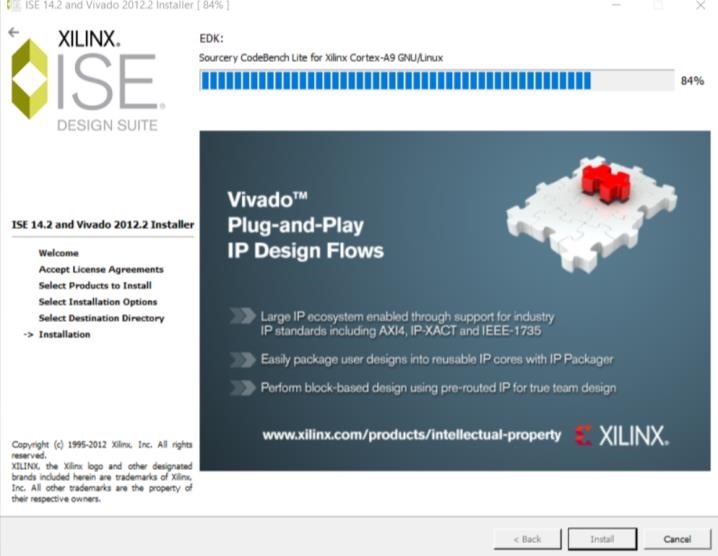
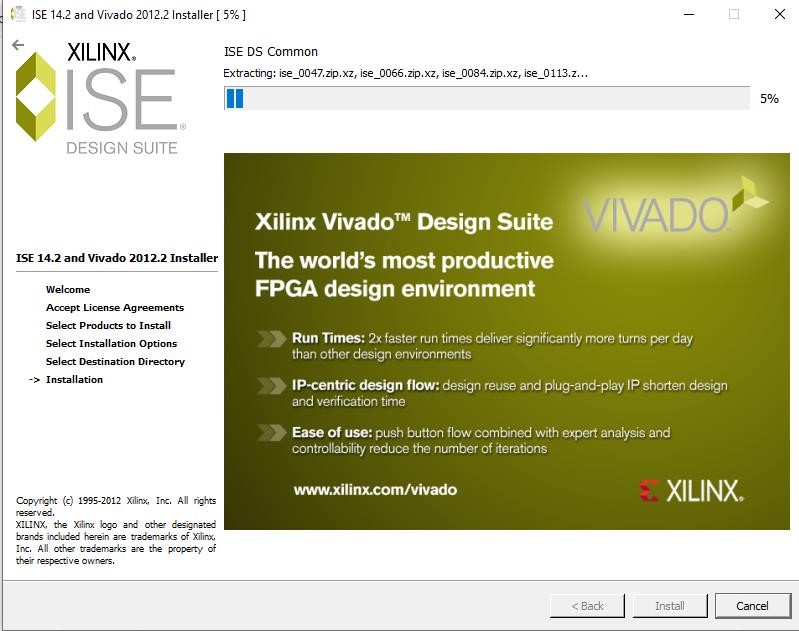
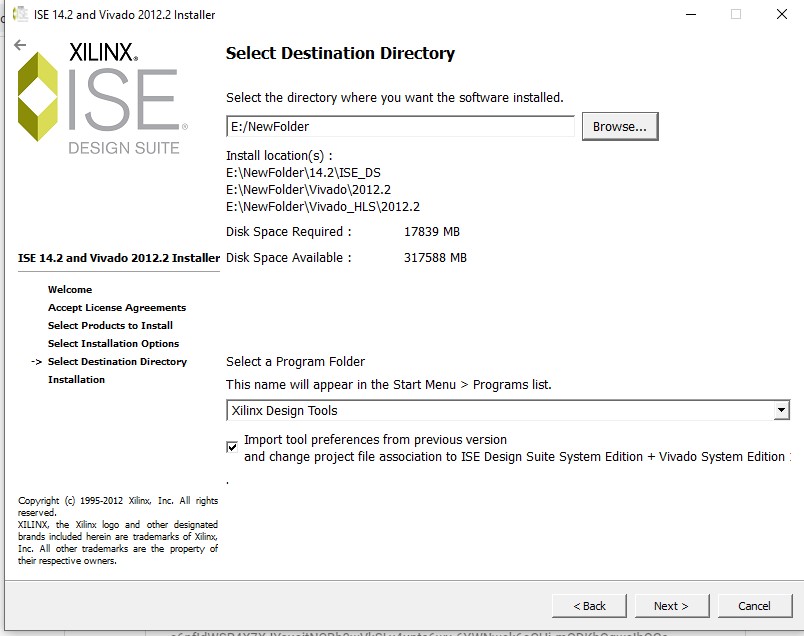
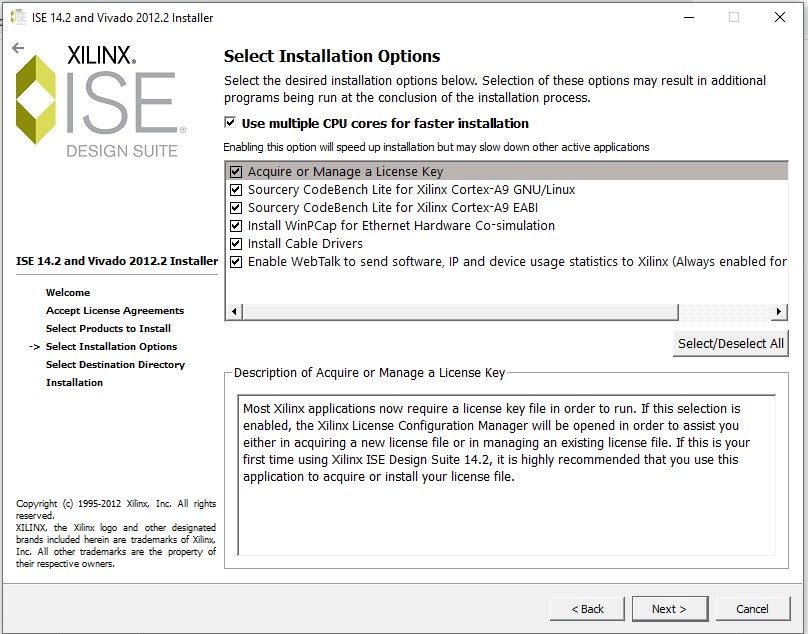
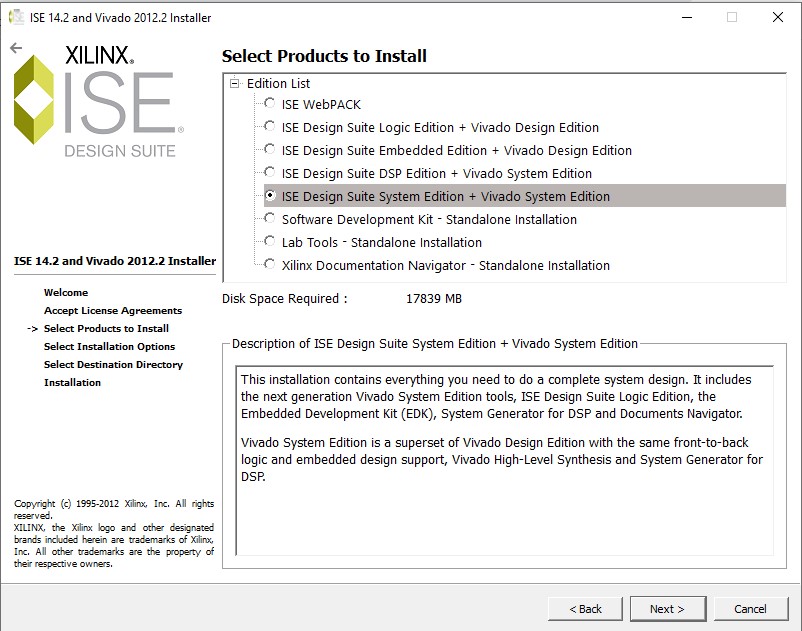
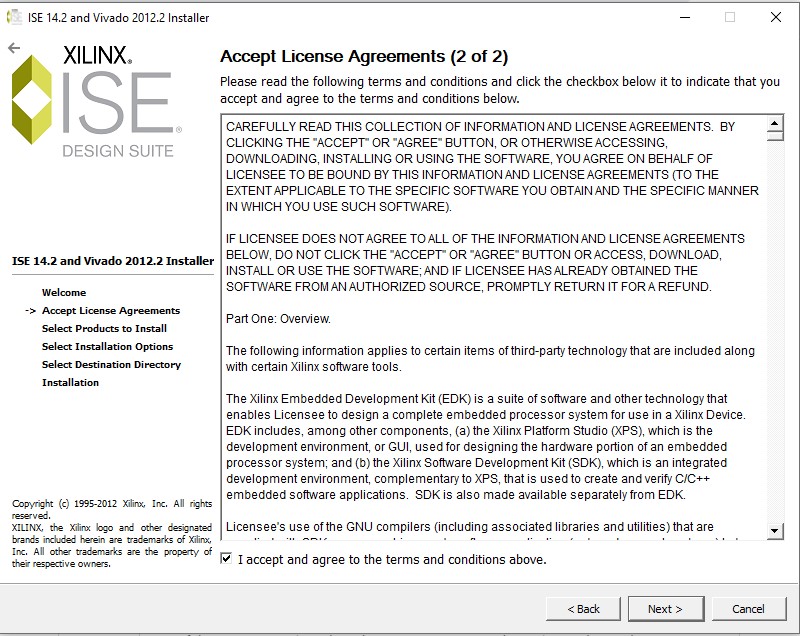
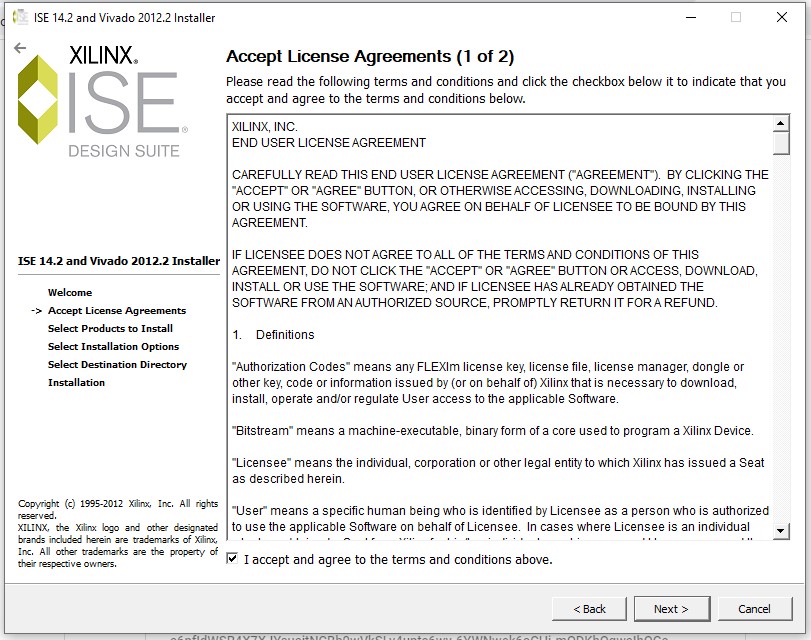
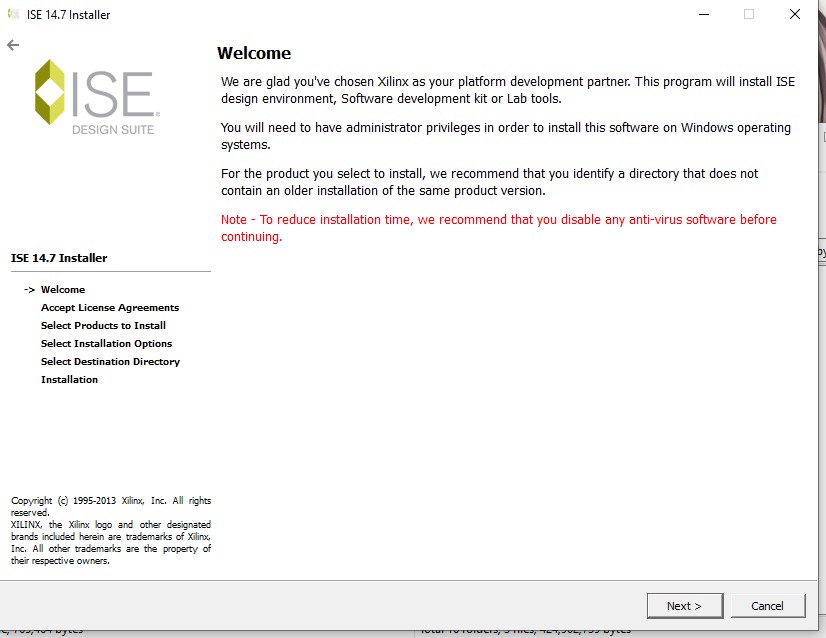
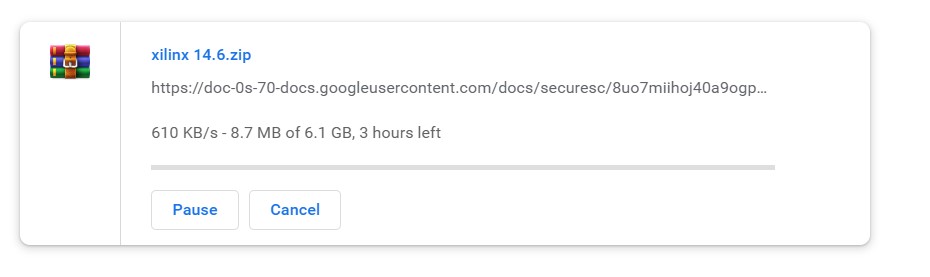
[Simulation: 32](#_Toc156158529)

[11. TB CPU: 32](#_Toc156158530)

[Program Explanation: 32](#_Toc156158531)

[Simulation: 33](#_Toc156158532)

# Installation:



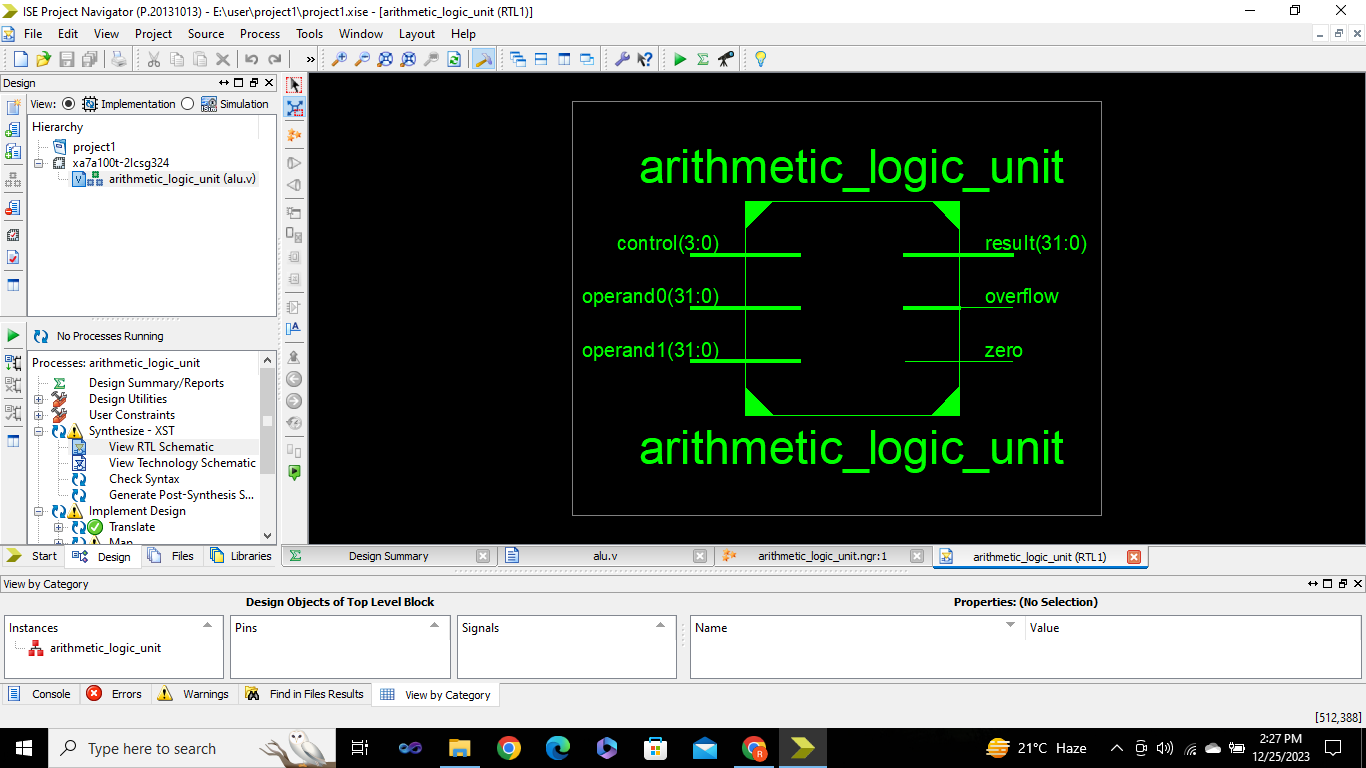
# Arithmetic logic unit (ALU):

## Program Explanation:

The overall simulation of this code will take input for control, operand0, and operand1 and then use the case statement to decide the operations to be performed, and then it will compute the result and put the output to the respective output pins.

|  |  |
| --- | --- |
| ALU Control Lines | Functions |
| **0000** | AND |
| **0001** | OR |
| **0010** | Add |
| **0110** | Subtract |
| **0111** | Set on less than |
| **1100** | NOR |

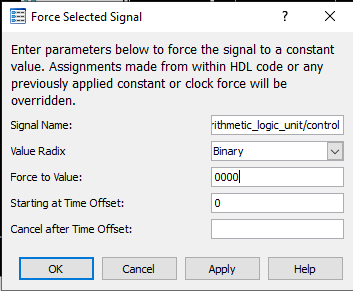
Implementation:

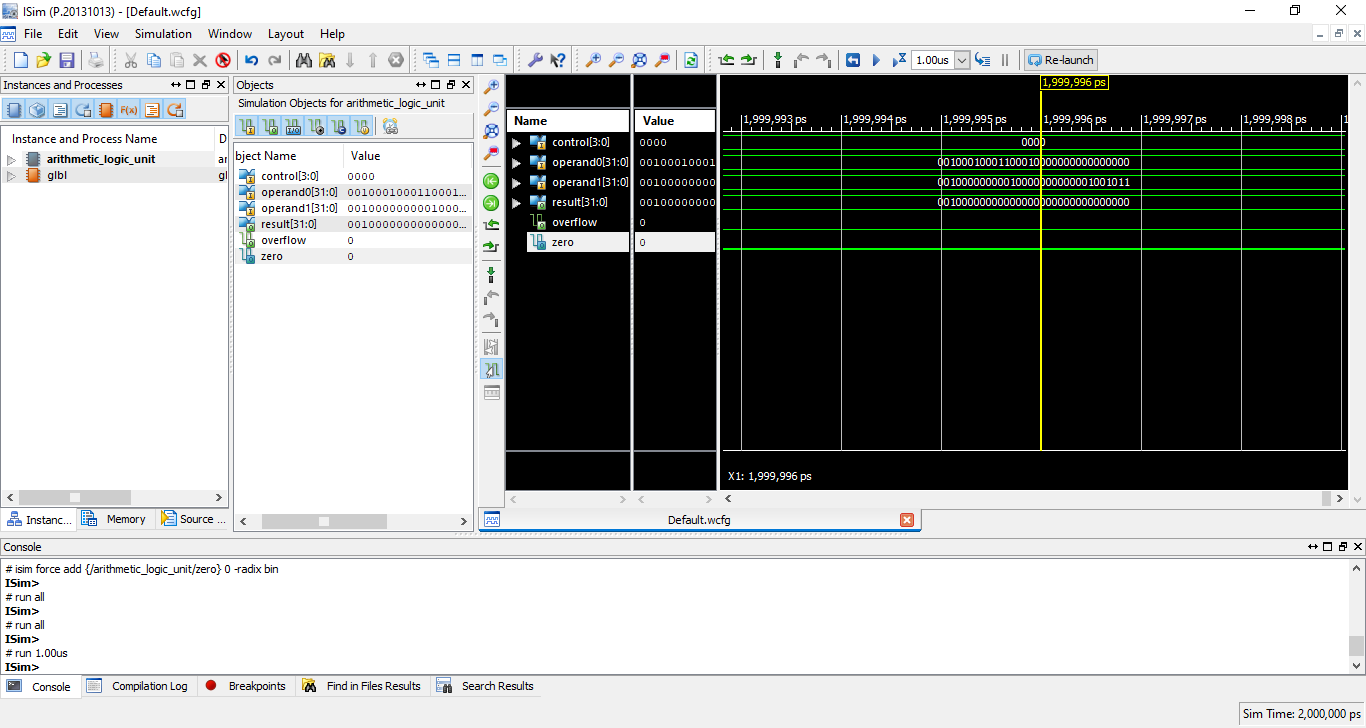


Simulation:

## FOR AND:

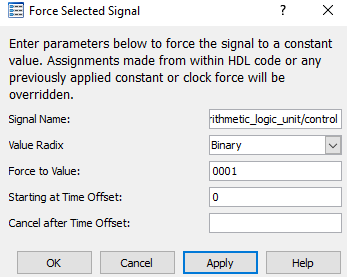
If the control bit is “**0000**”**, AND** operation will performed on the operands. It will compare each bit of the 2 operands. If the bit of both operands are 1 then the result bit will be “**1**”, otherwise result bit will be “**0**”

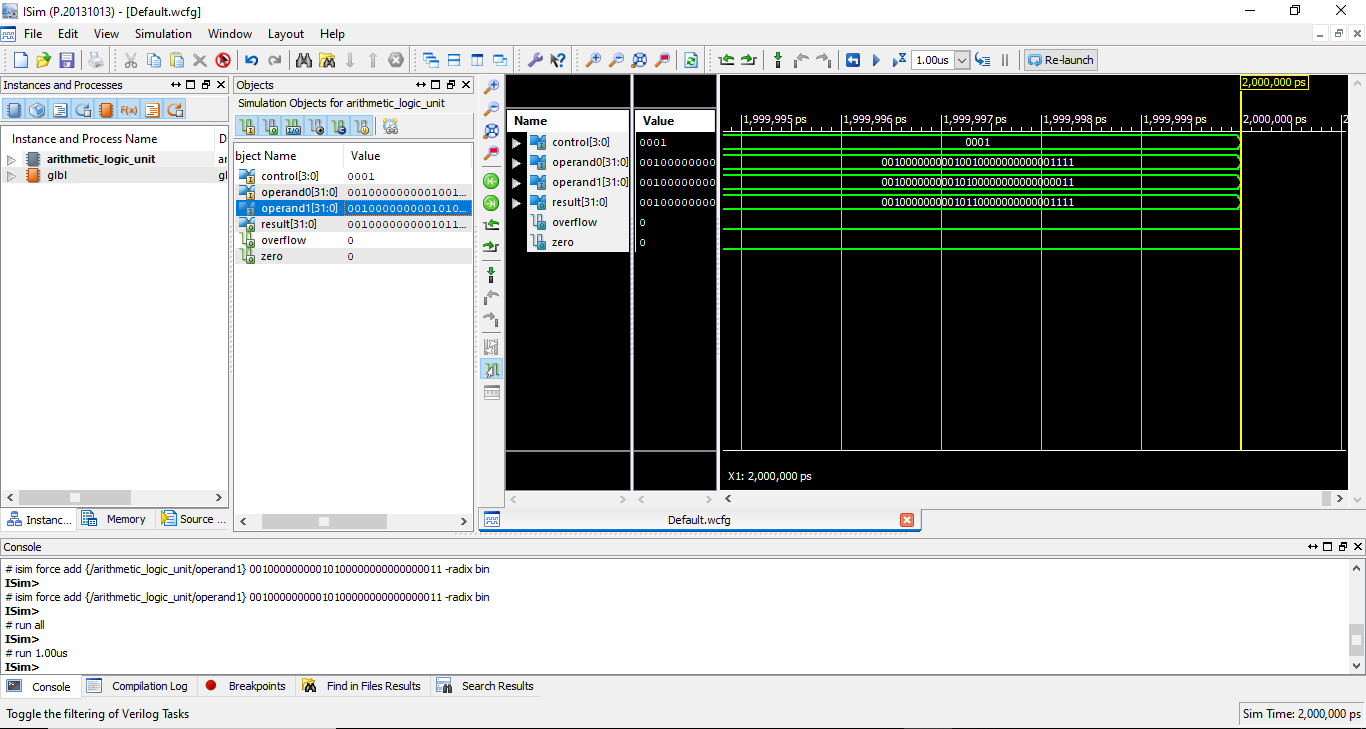




## For OR :

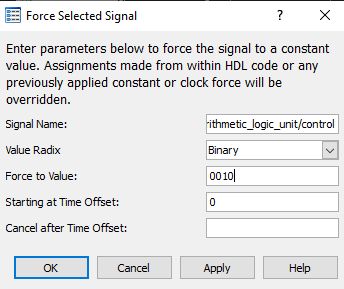
If the control bit is “**0001**”**, OR** operation will performed on the operands. It will compare each bit of the 2 operands. If the bit of both operands are “**0**” then the result bit will be “**0**”, otherwise result bit will be “**1**”

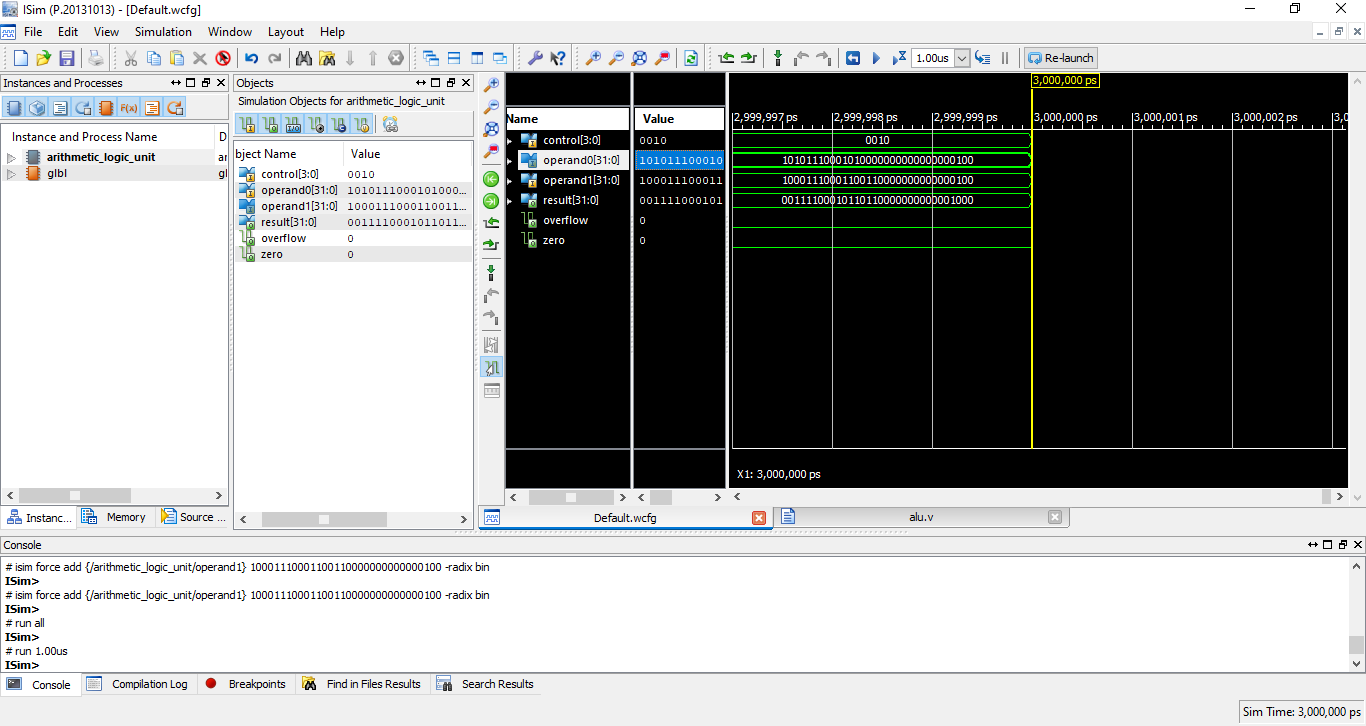




## For Add:

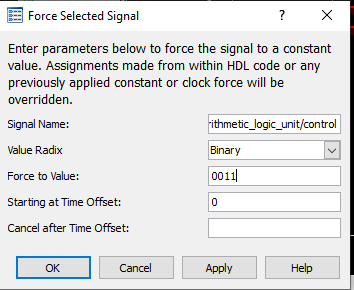
If the control bit is “**0010**”**, ADD** operation will performed on the operands. It will add the both operands. If the bit of both operands are 0 then the result bit will be “**0**”, otherwise result bit will be “**1**” but if the bit of both operands are “**1**”, then the result bit will be “**0**” and “**1**” will be taken as a carry for the next bit. This will form a new number that is the result of addition of the two operands.

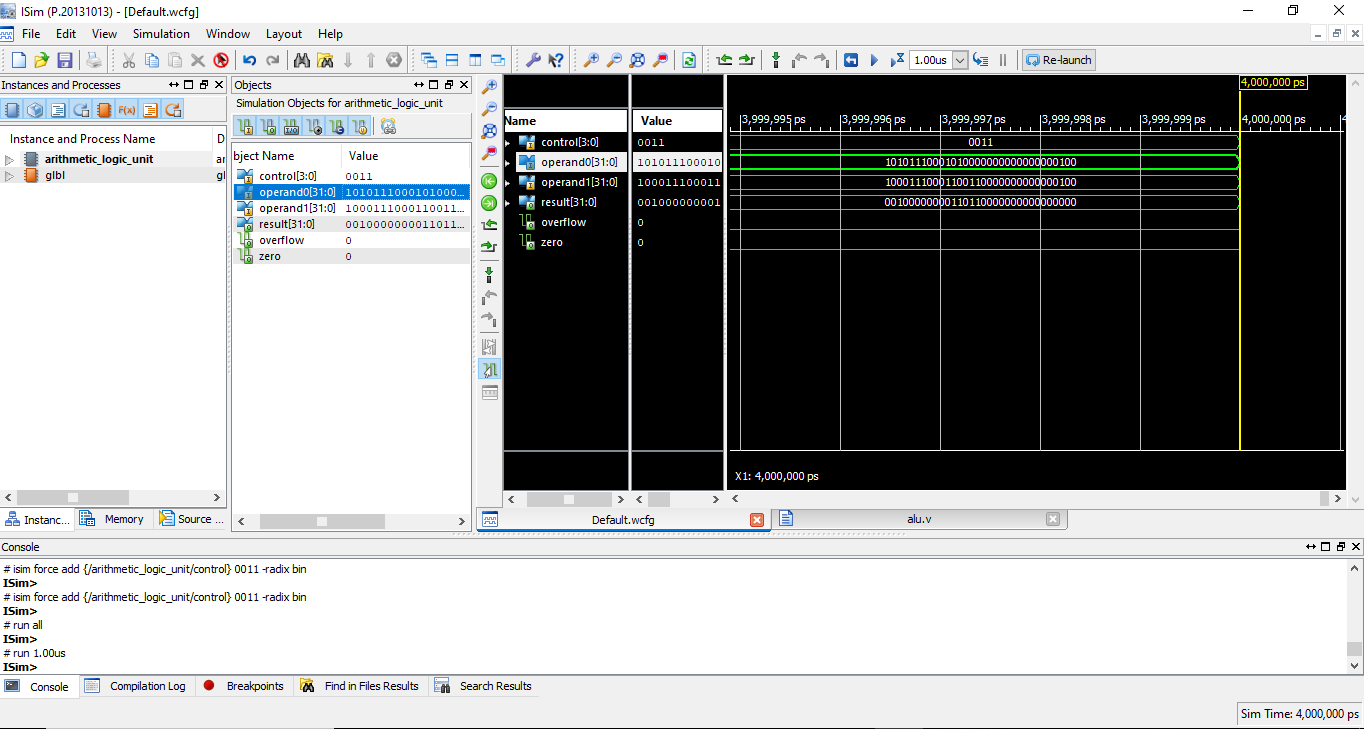




## For XOR:

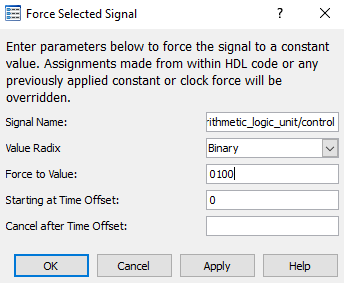
If the control bit is “**0011**”**, XOR** operation will performed on the operands. It will compare each bit of the 2 operands. If the bit of both operands are opposite then the result bit will be “**1**”, and if they are same then resultant bit will be “**0**”

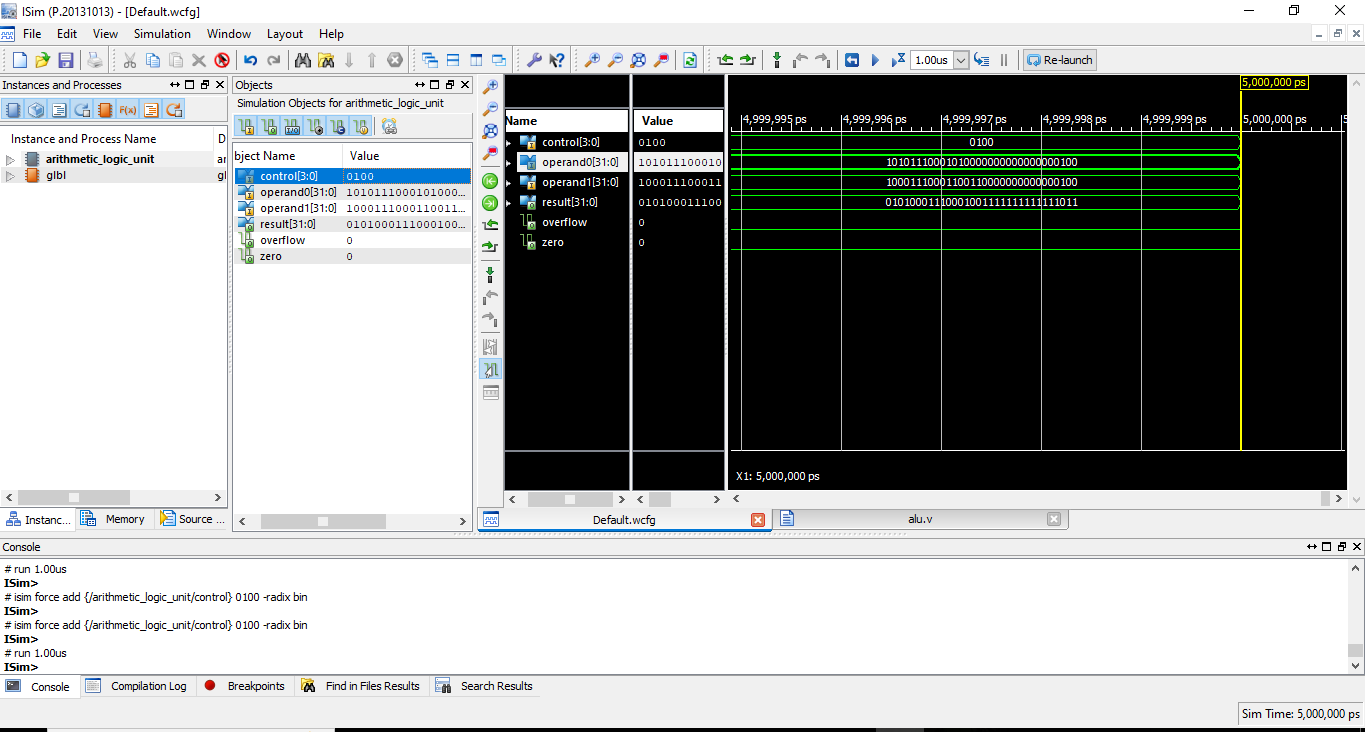




## NOR:

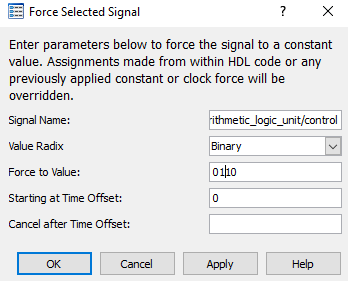
If the control bit is “**0100**”**, NOR** operation will performed on the operands. It will compare each bit of the 2 operands. If the bit of both operands are “**0**” then the result bit will be “**1**”, otherwise result bit will be “**0**”

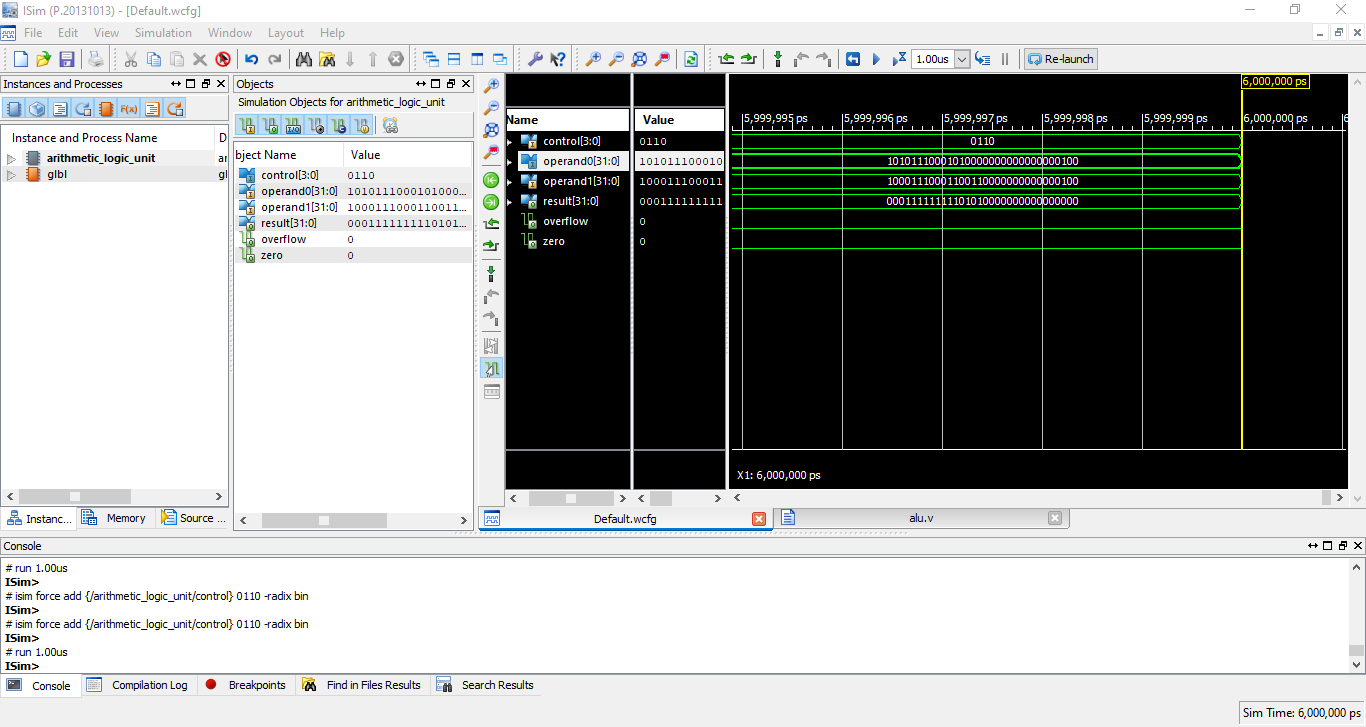




## For Sub:

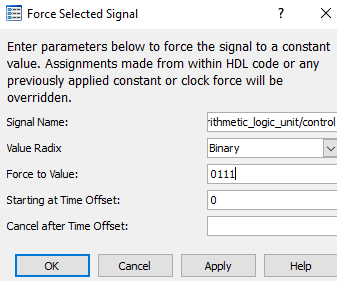
If the control bit is “**0110**”**, Subtract** operation will performed on the operands. It will add the both operands. If the bit of both operands are 0 then the result bit will be “**0**”, otherwise result bit will be “**1**” but if the bit of both operands are “**1**”, then the result bit will be “**0**” and “**1**” will be taken as a carry for the next bit. This will form a new number that is the result of addition of the two operands.

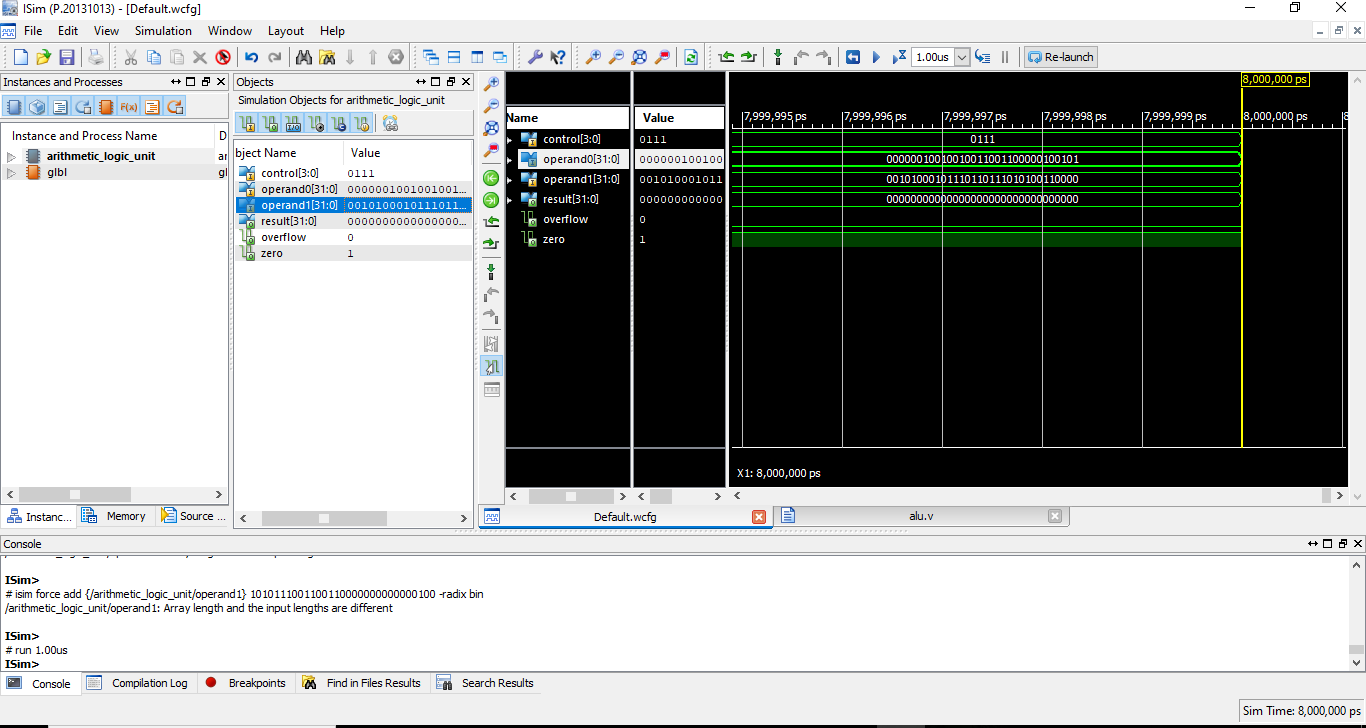




## Set less than:

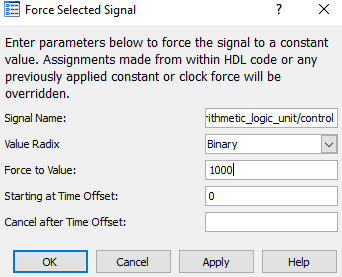
If the control bit is “**0111**”**, less than** operation will performed on the operands.

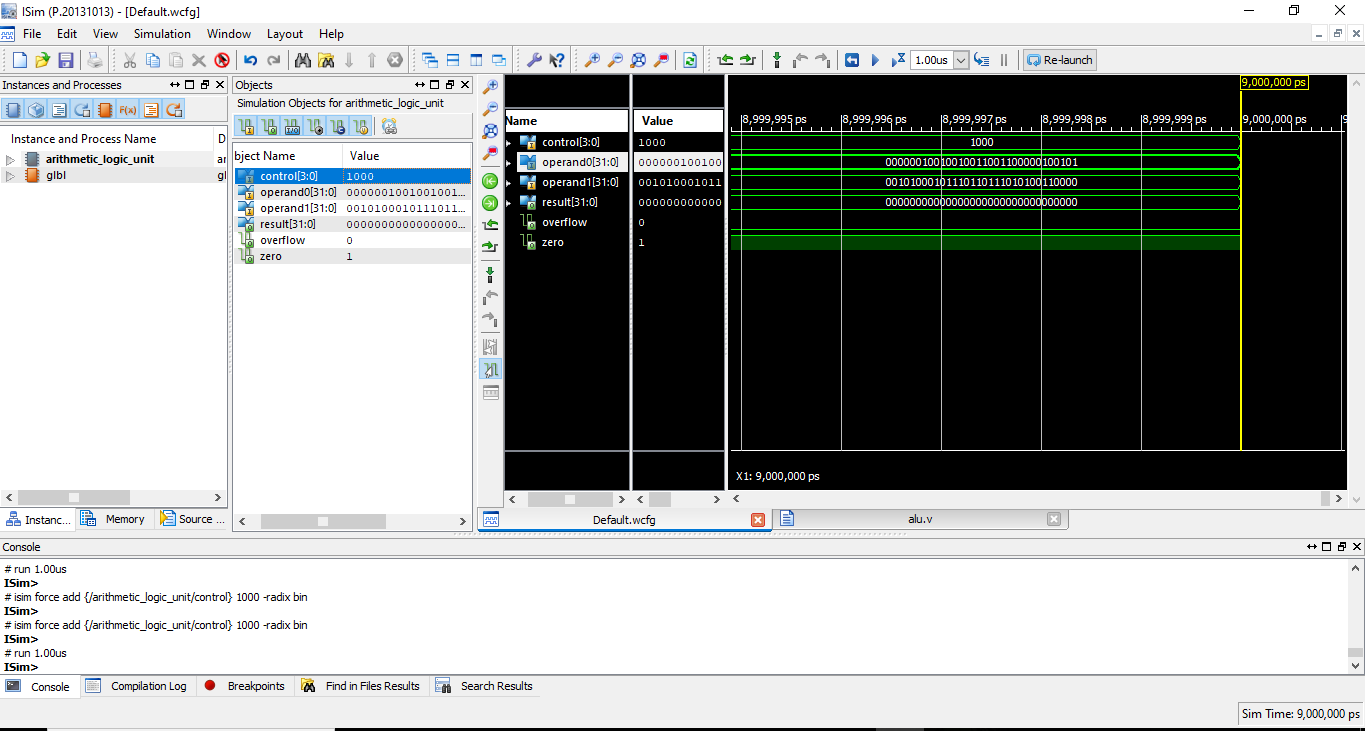




## Shift left:

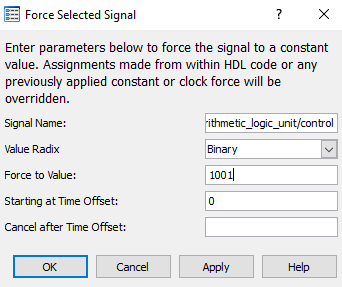
If the control bit is “**1000**”**, Shift left** operation will performed on the operands.

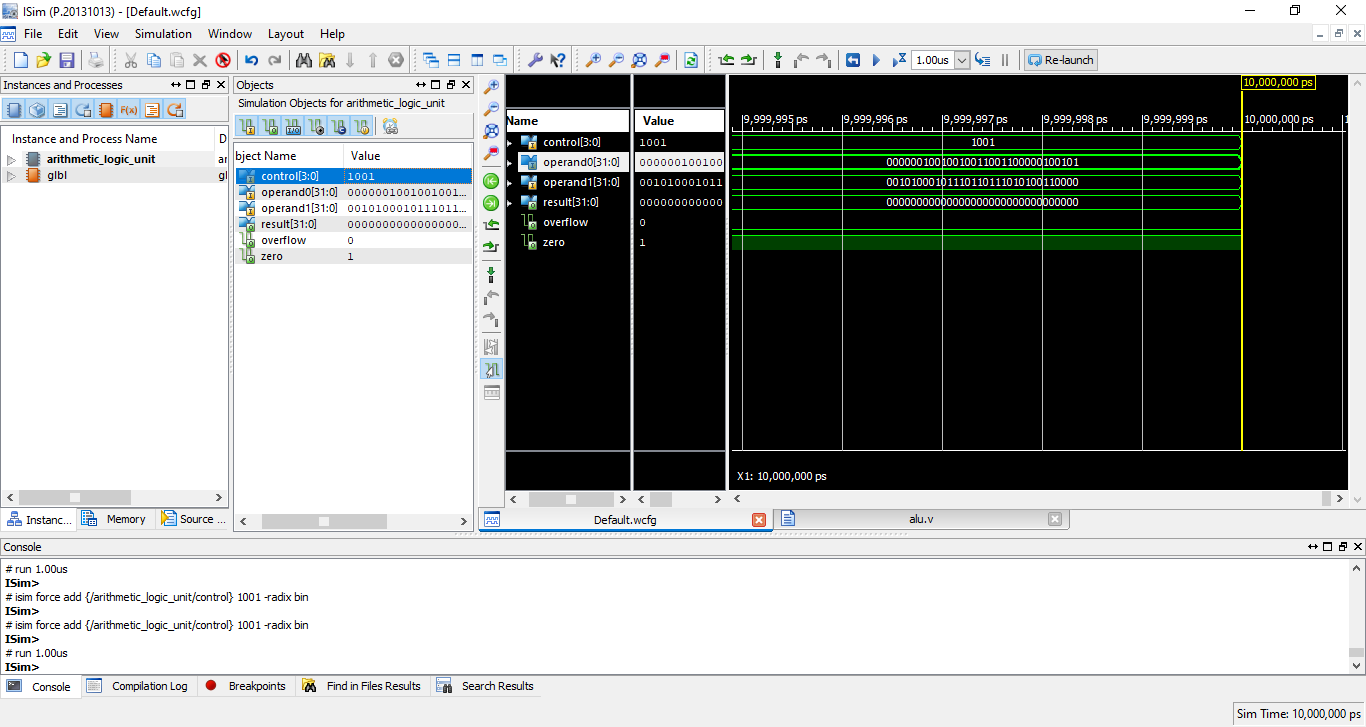




## Shift right:

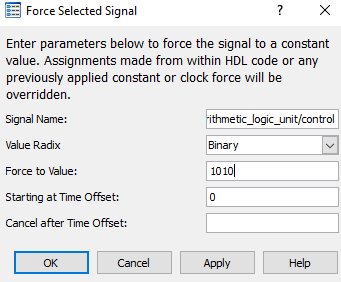
If the control bit is “**1001**”**, Shift right** operation will performed on the operands.

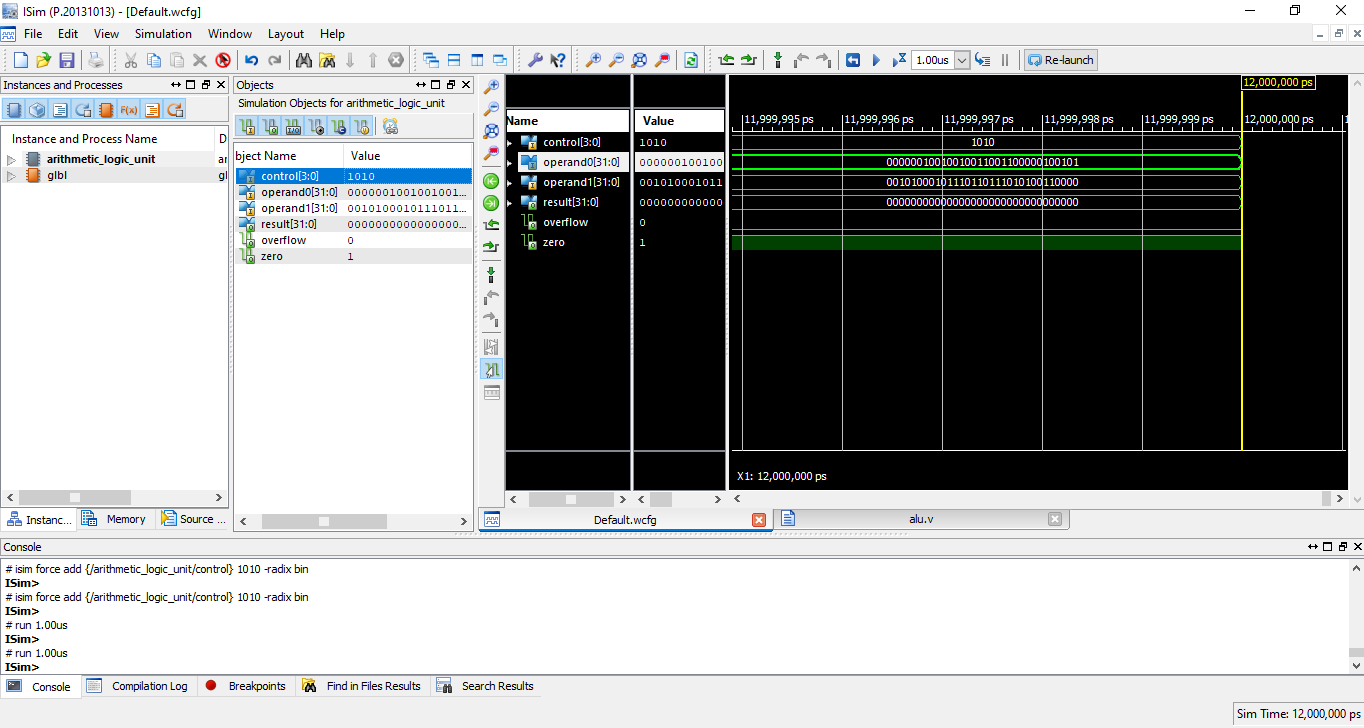




## Shift right Arithmetic:

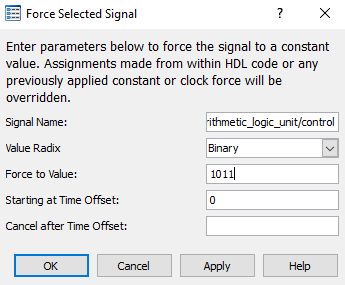
If the control bit is “**1010**”**, Shift right arithematic** operation will performed on the operands.





## Signed add:

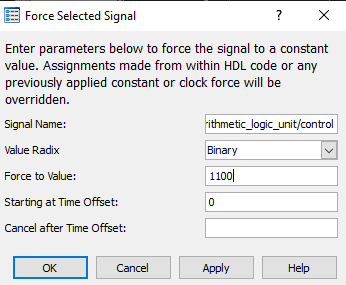
Two operands are given as an input and control bits 1011 will perform signed add operation.

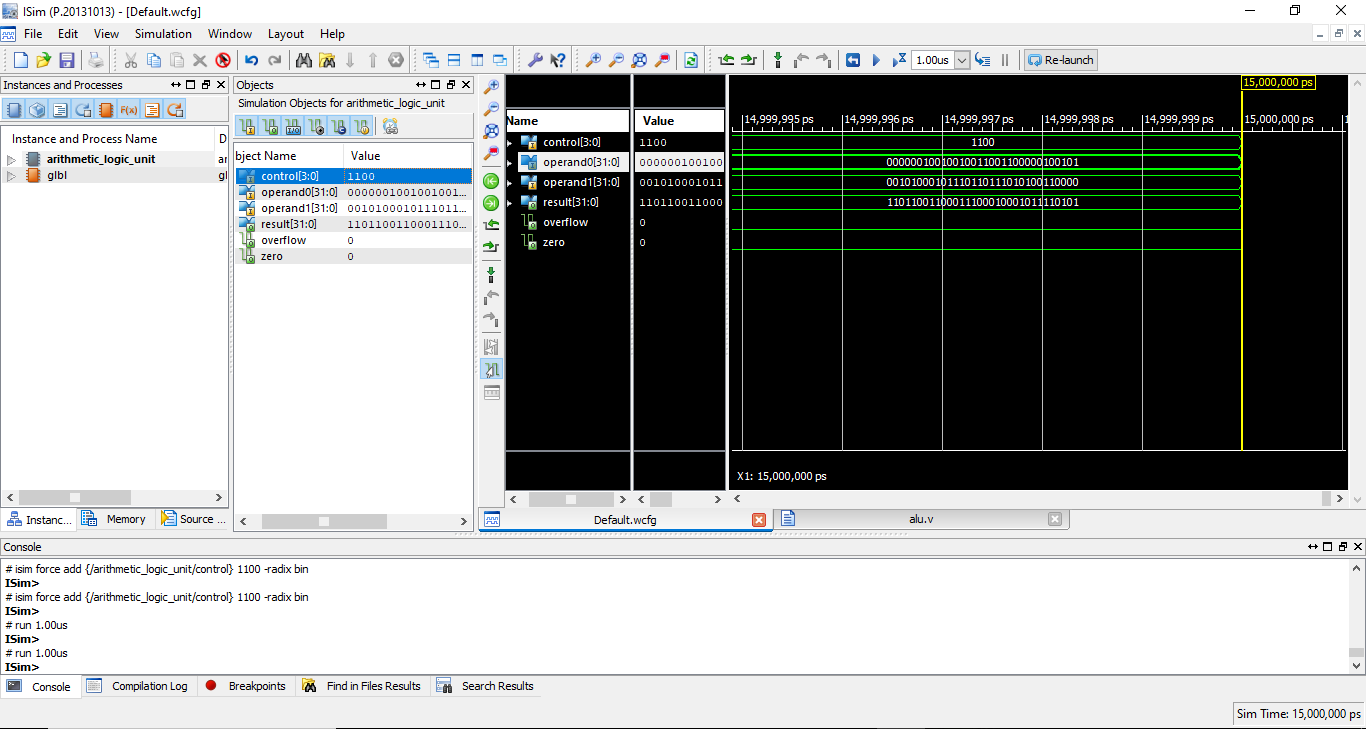




## Signed sub:

Two operands are given as an input and if control bits are 1100, then signed subtract operation will be performed.





# Mux :

# 2-to-1 multiplexer:

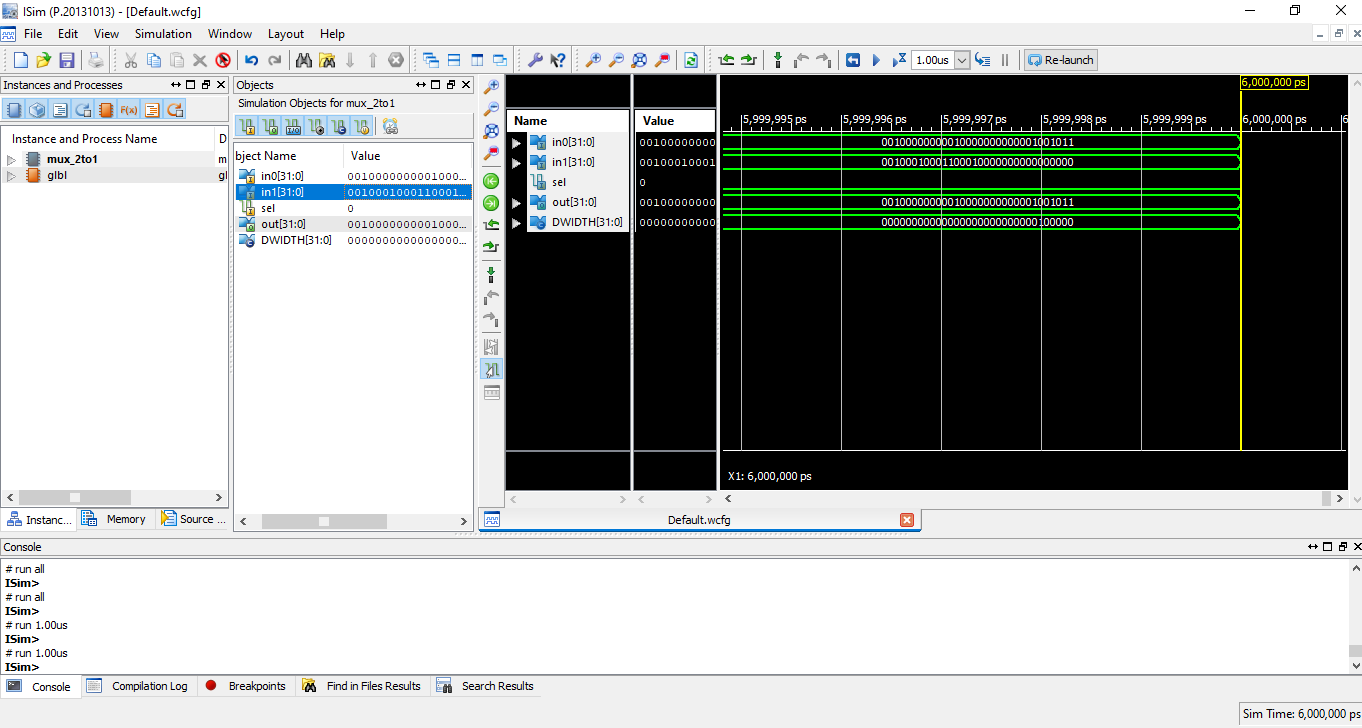
## Program Explanation:

This is a Verilog code for a 2-to-1 multiplexer (mux) module. A multiplexer is a digital circuit that selects one of several inputs and forwards the selected input to its output. The mux\_2to1 module takes in three inputs: in0, in1, and set, and has one output, out. The parameter DWIDTH is defined as 32, and it's used to define the width of the in0, in1 and out input and output ports. The set input is a single-bit value that determines which of the two inputs, in0 or in1, will be forwarded to the output, out. When set is 0, in0 is selected and forwarded to out. When set is 1, in1 is selected and forwarded to out.

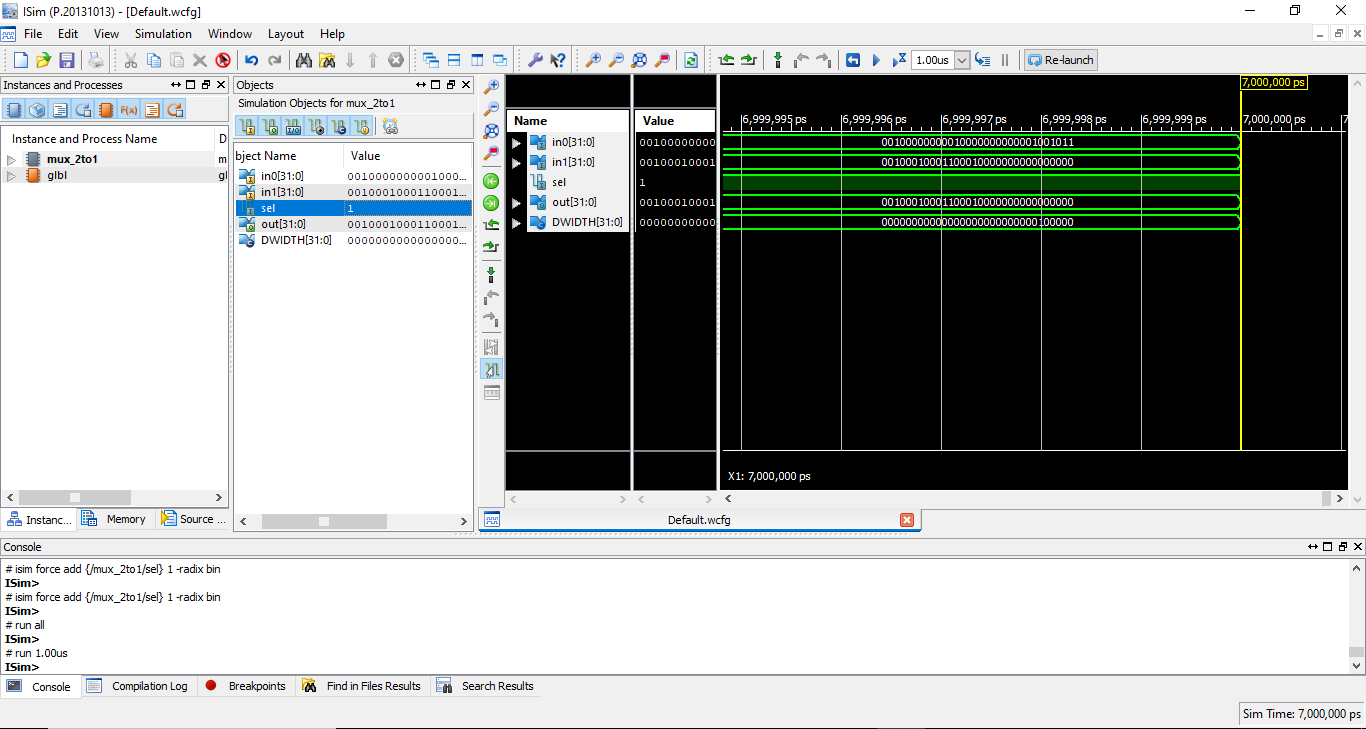
Implementation:



## Set bit 0:



## Set bit 1:

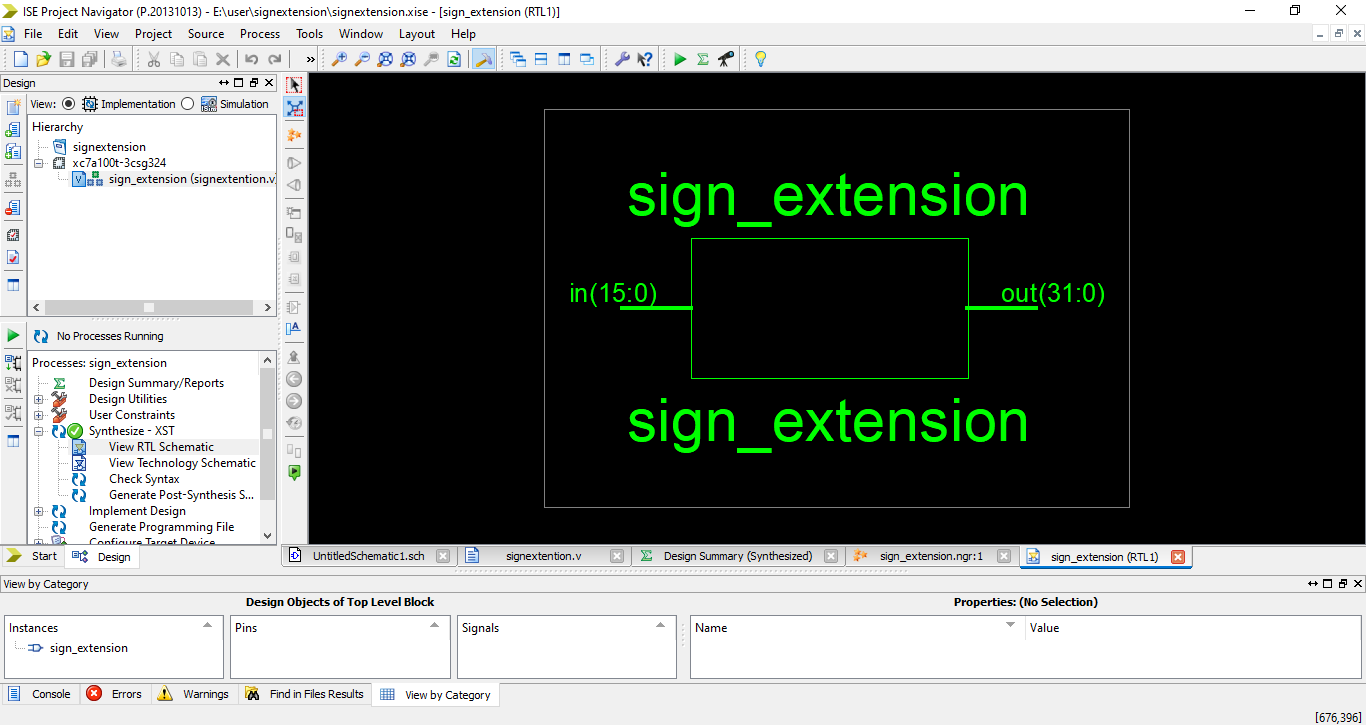


# Sign extension:

## Program Explanation:

Simulation of the sign\_extension module is a process of testing the behavior of the design under different scenarios. During the simulation, the values of the input port 'in' are set according to the test bench. The assign statement in the combinatorial logic block uses the input value to generate the value of the output port 'out'. The output value, out, is a replication of the most significant bit of the input value (MSB) of the input in, followed by the value of in. If MSB of input is 1, output is negative and if MSB is 0, output is positive. It will help to extend the bit width of input and to preserve the sign of number.

Implementation:



## Simulation:

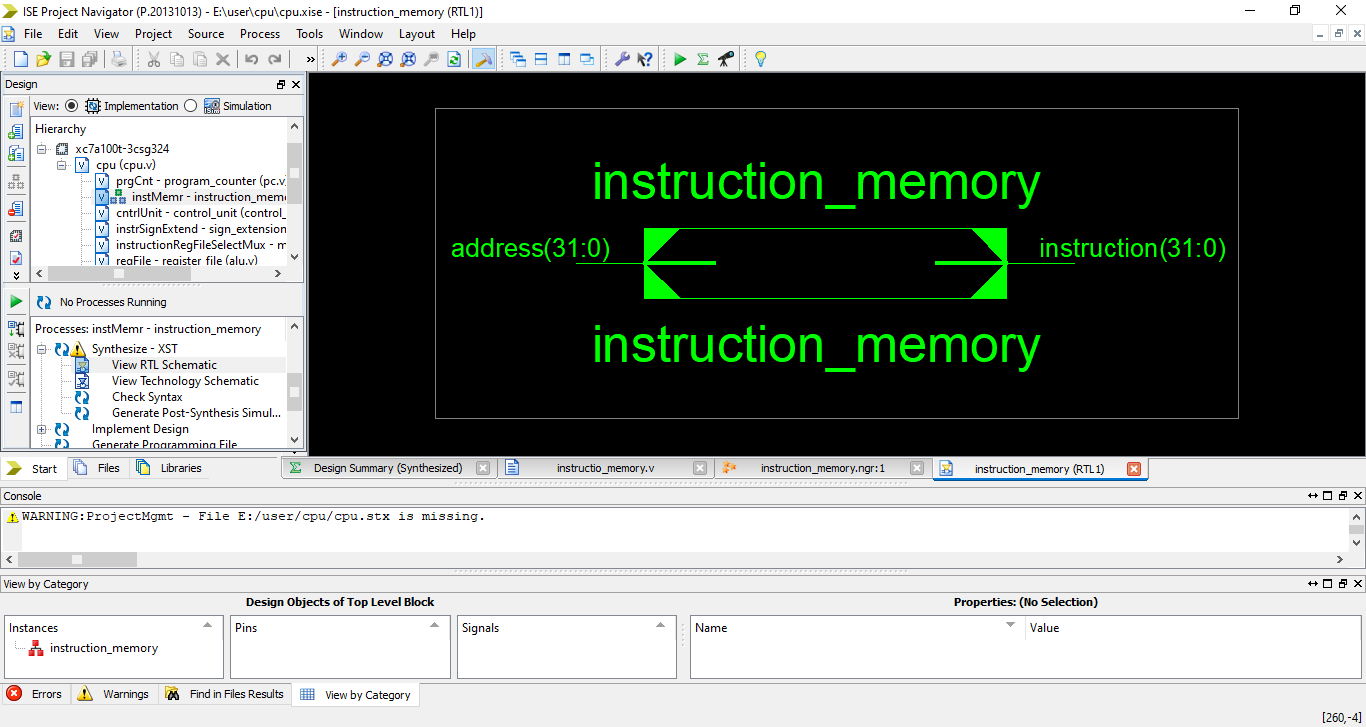
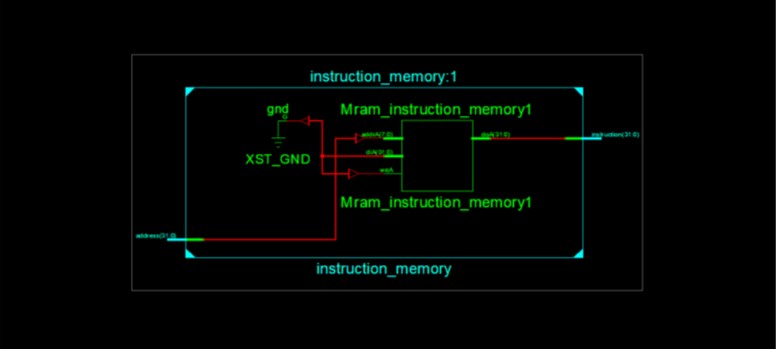


# Instruction memory:

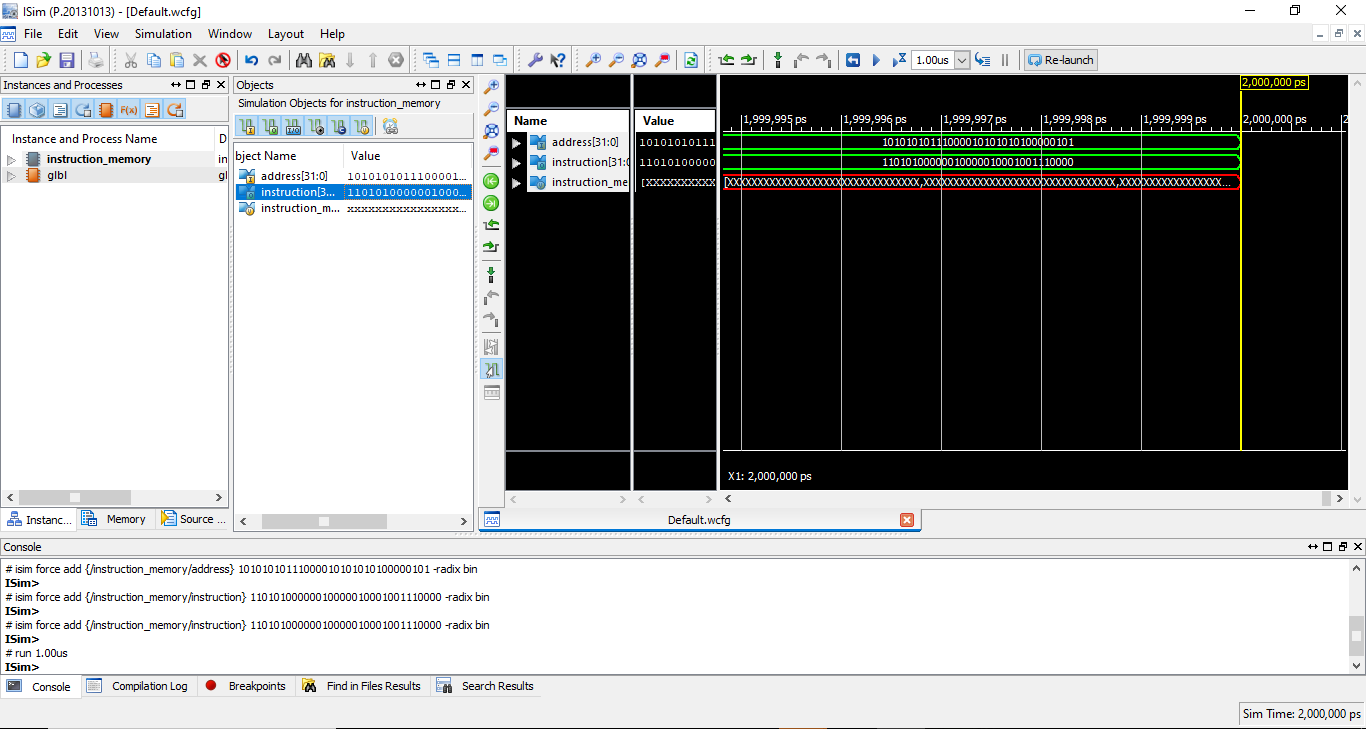
## Program Explanation:

This is a Verilog code for an instruction memory module. Instruction memory is used to store the instructions that a CPU or other digital device will execute. The code defines an instruction memory module that takes in two inputs: address, and instruction. The address input is a 32-bit value used to specify the location in memory of the instruction that is going to be fetched. The instruction output is a 32-bit value representing the instruction stored at the specified address. The code then goes on to define 1 register named instruction\_memory of 32-bit size

Implementation:

Simulation:



# Control unit:

## Program Explanation:

Control unit: The control unit of the central processing unit regulates and integrates the operations of the computer. It selects and retrieves instructions from the main memory in proper sequence and interprets them so as to activate the other functional elements of the system at the appropriate moment.

**R-type:**

* Address and immediate is 0.
* Rs, rt, rd and shamt will be extracted from the instruction

**I-type:**

* Address will be zero and immediate will be extracted from the instruction automatically.
* Rs will be extracted from the instruction, rt will store the address and rd and shamt will be 0.

**J-type:**

* 26 bits’ address will be extracted from the instruction and immediate will be zero.
* Rs, rt, rd, function bits and shamt will be 0

**Signals:**

1. **data\_mem\_wren**

When it is 0000 data cannot be stored in the memory. For I-type for load case data\_mem\_wren is 0000 and for store case it is 1111. For I-type and J-type it will be 0000.

1. **reg\_file\_wren**

It determines whether the branch signal is occurred or not. If branch signal is occurred it is 1 otherwise it is 0.

1. **reg\_file\_dmux\_select**

**For 0** it means variation has been occurred in loaded word and if it is 1 it means there is no variation in loaded word.

1. **reg\_file\_rmux\_select**

Will be 1 for R-type and 0 for I-type and J-type according to the code.

1. **alu\_mux\_select**

It is 1 for I-type and 0 for J-type and J-type.

1. **alu\_control**

Determines which operation is to be applied on the operand(s).

1. **PC**

Holds the address of the next instruction. In the output below the branch and jump has not occurred so the default part of the code is running that is 000.

## I type

If the opcode is other than 000000, 000010, 000011 then Address will be assigned 00000000000000000000000000 bits and the immediate will be assigned instruction [15:0] bits. **rs** will be assigned bits of instruction[25:21]. **rt** will be assigned bits of instruction[20:16]. **rd** will be 00000 because in I type, there is no need of rd. shamt will be assigned bits of instruction[10:6]. Type will be **010** and the funct bit will be bits of instruction [5:0];

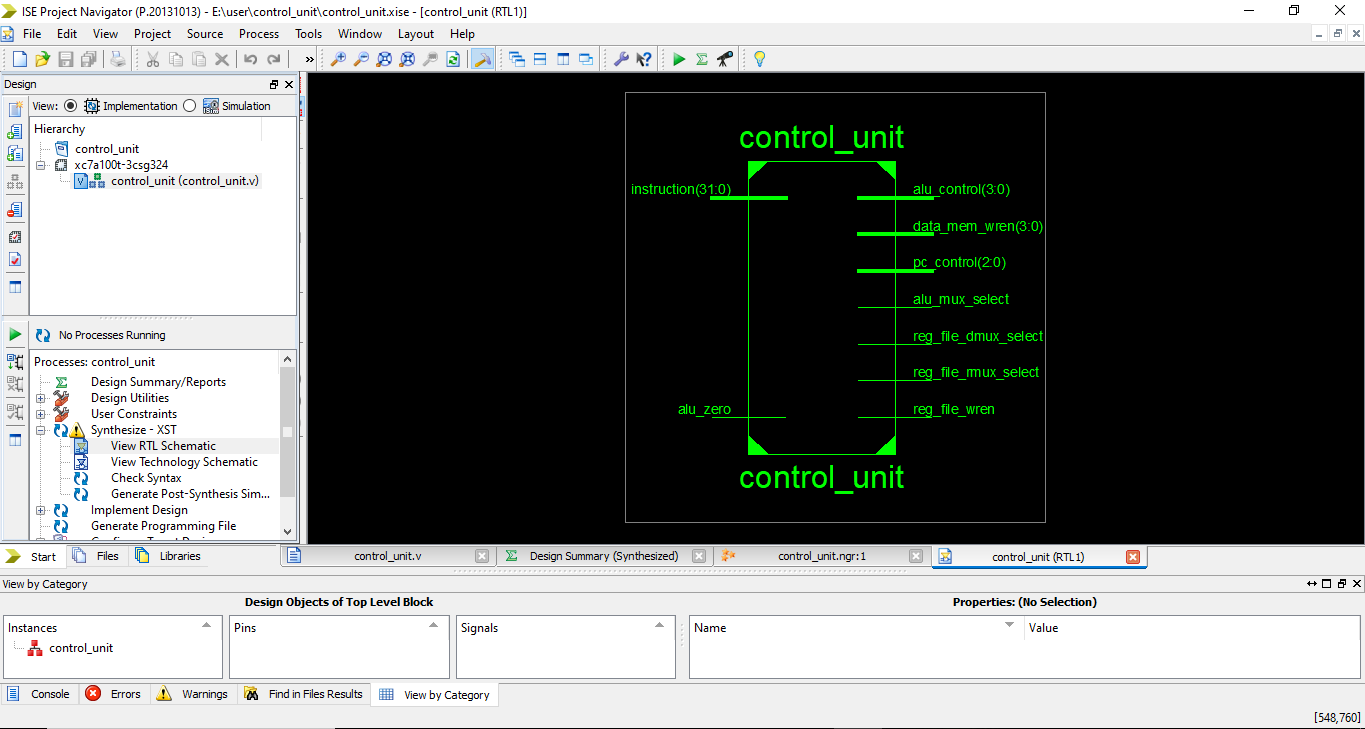
## Jump:

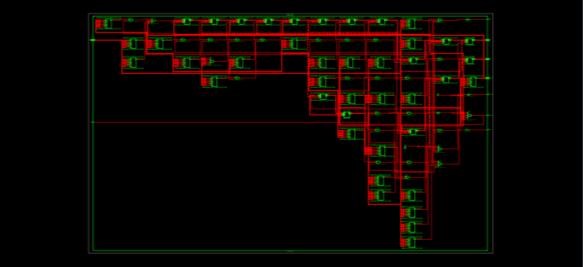
If the opcode is either 000010 or 000011, then Address will be assigned instruction [25:0] bits and the immediate will be assigned 0000000000000000 bits. **rs** will be assigned 00000 bits. **rt** will be assigned 00000 bits. **rd** will be 00000. shamt will be assigned 00000 bits. Type will be **100** and the funct bit will be 000000.

## OPCODE for instructions

|  |  |
| --- | --- |
| load | 100011 |
| store | 101011 |
| jump | 000010 |
| branch | 000100 |

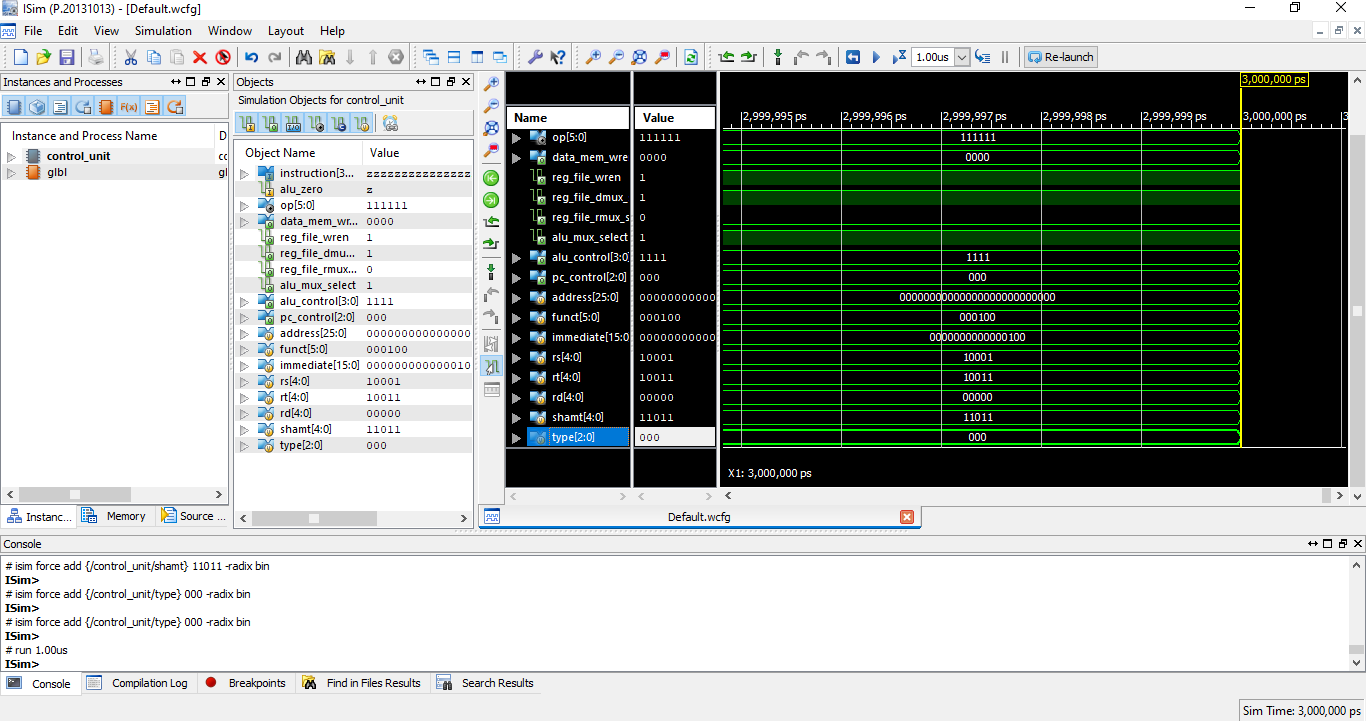
Implementation:



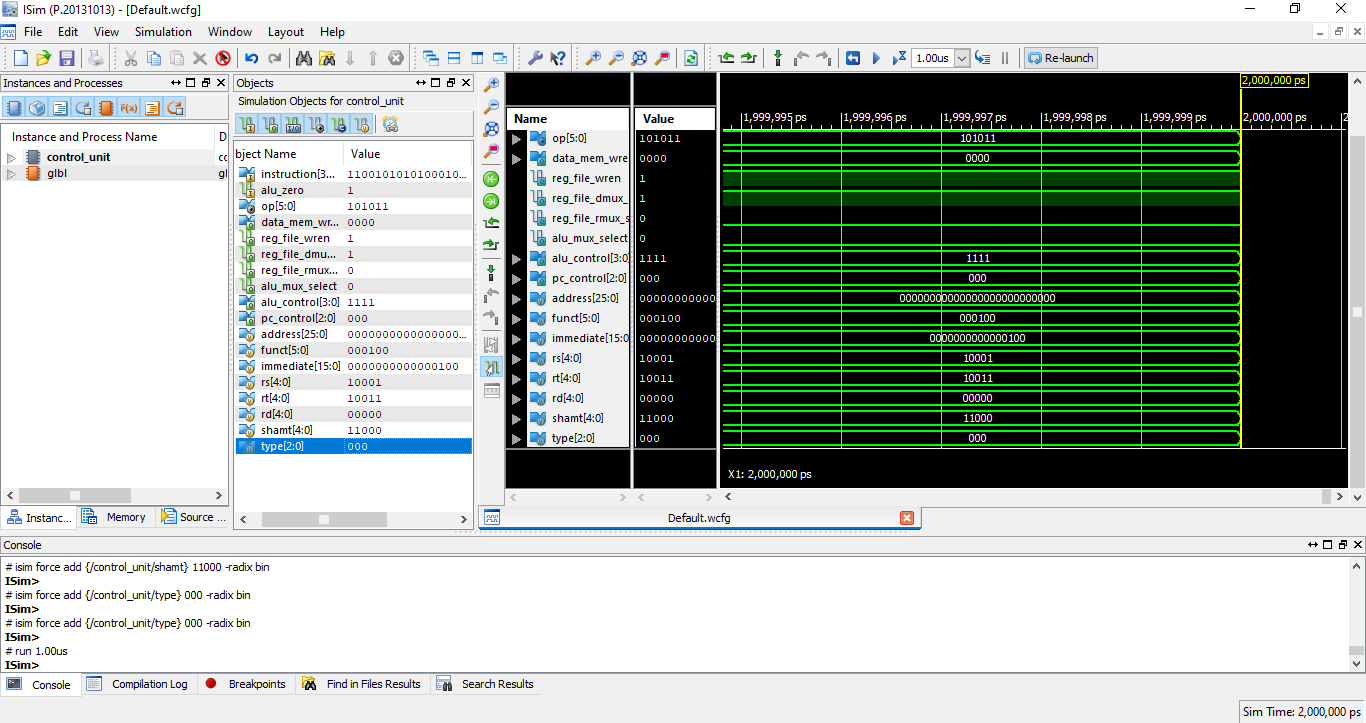


Load:

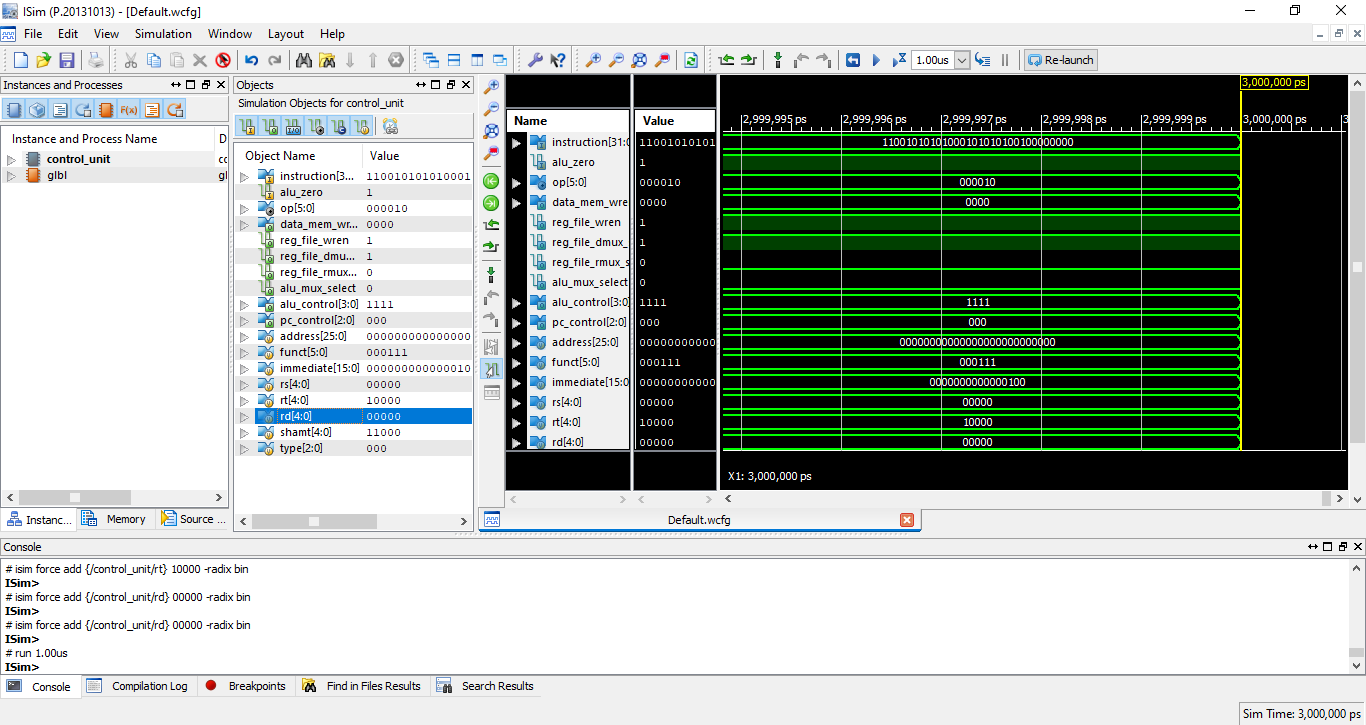
## Load:



## Store:



## Jump:



# Data memory:

## Program Explanation:

This is a Verilog code for a data memory module. Data memory is used to store data that can be read and written to by a CPU or other digital device.

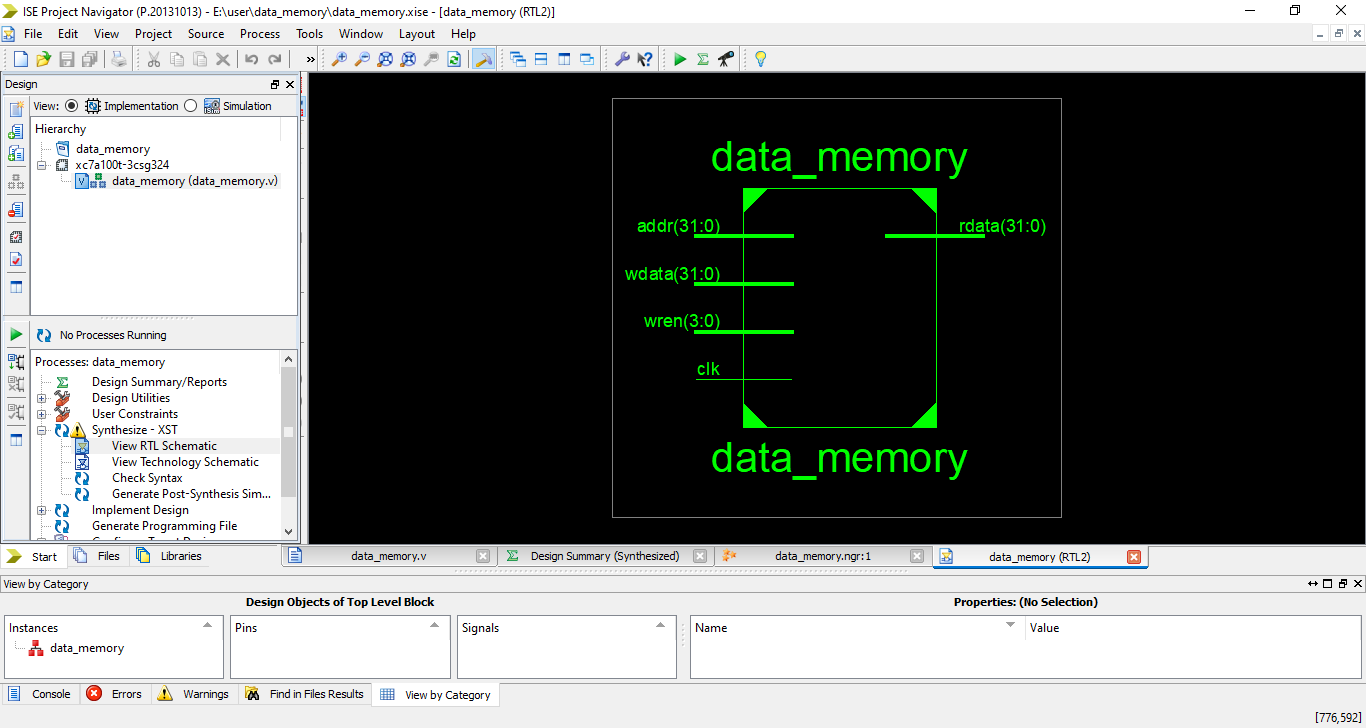
• The code defines a data memory module that takes in four inputs: clk, addr, wdata, and wren. The clk input is the clock signal that synchronizes the memory operations.

• The addr input is a 32-bit address used to specify the location in memory from which data should be read or to which data should be written.

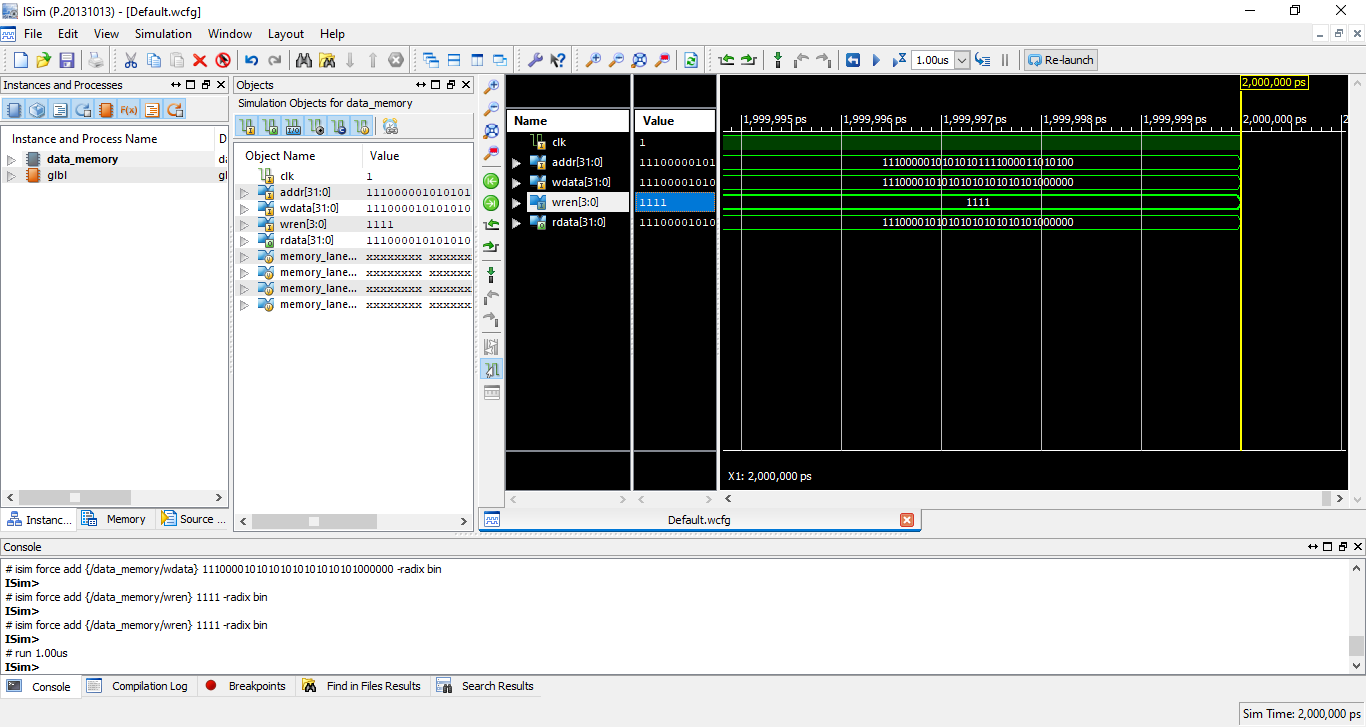
• The wdata input is a 32-bit value that will be written to memory if the wren input is high.

• The wren input is a 4-bit value that is used to enable writing to specific bytes of the memory. The module has one output port, rdata, which is 32-bit value read from memory at the specified address. The code then goes on to define 4 registers named memory\_lane0,memory\_lane1,memory\_lane2,memory\_lane3

Implementation:



## Simulation:



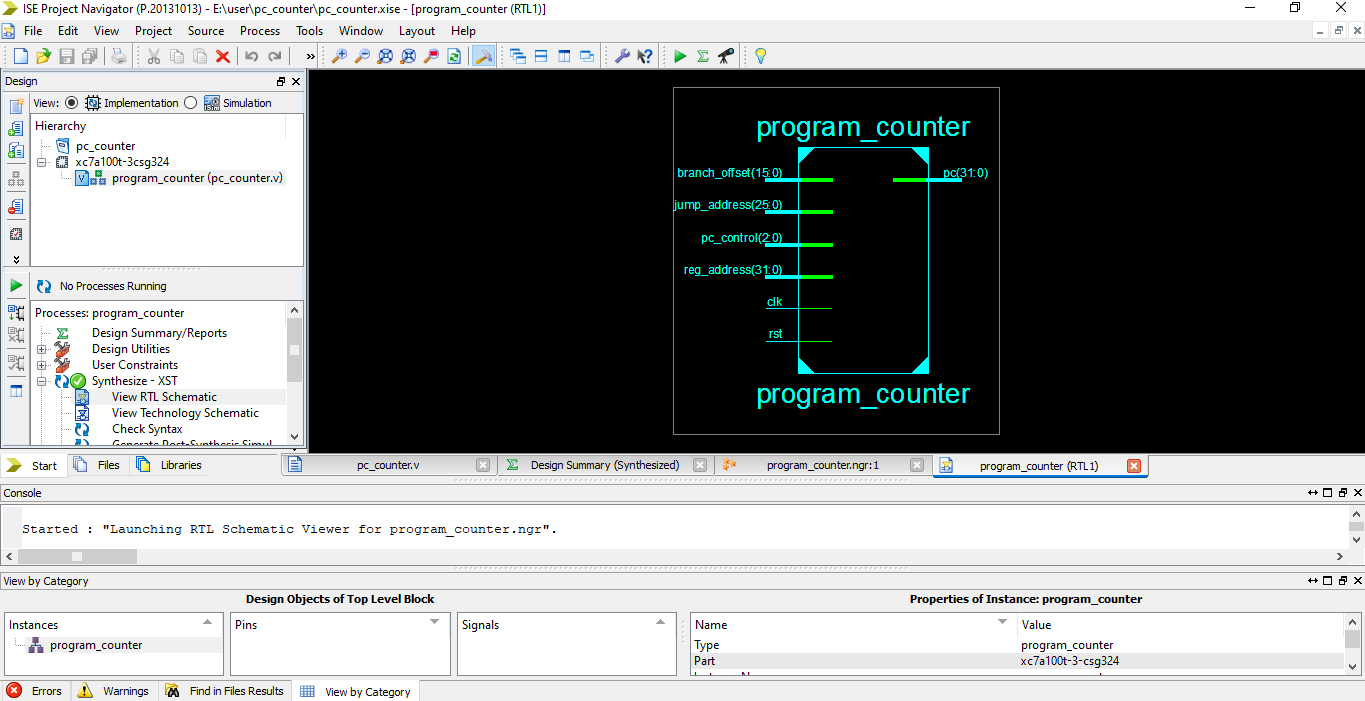
# Program counter:

## Program Explanation:

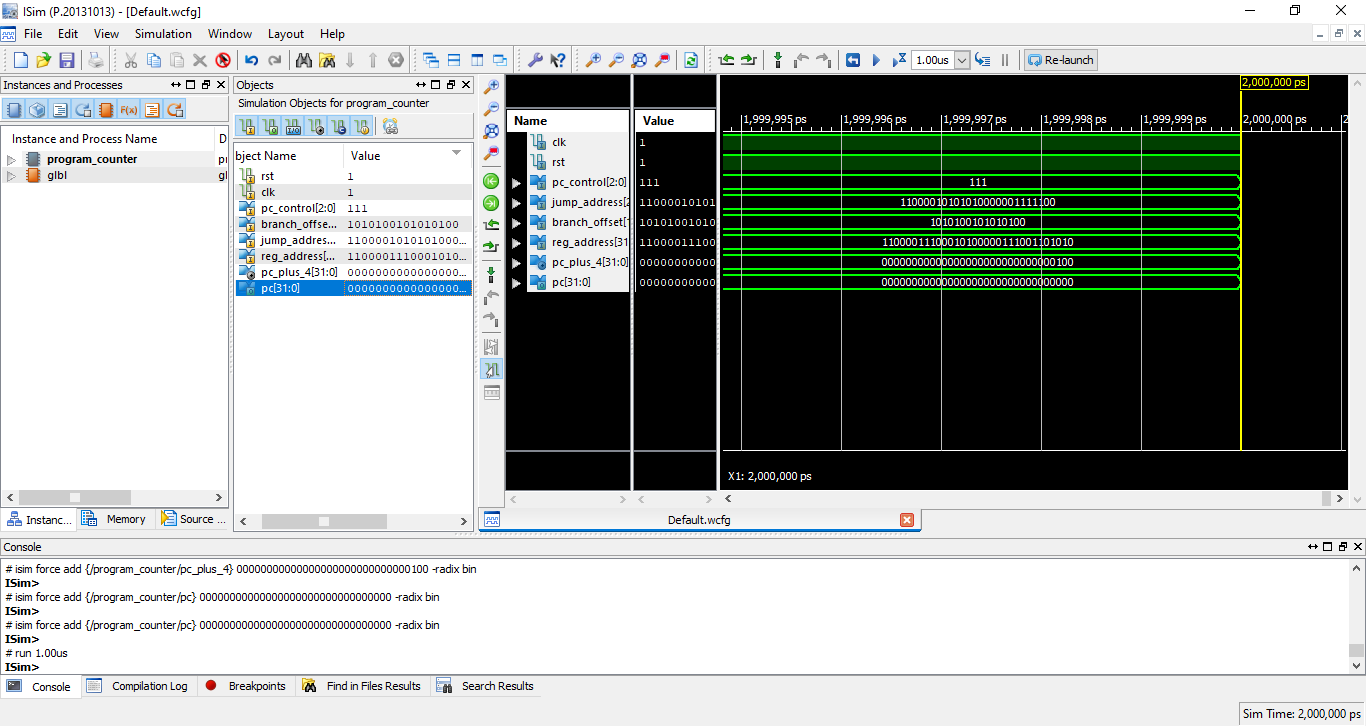
In program counter code the **pc\_control**, **jump\_address**, **branch\_offset** and **reg\_address** are set as the inputs and output is the value of pc.

Output will be generated according to pc control value. **Pc\_control** will determine that for which instruction pc value will be incremented.

Implementation:



## Simulation:



# Register file:

## Program Explanation:

Register\_file The "register\_file" module is a design that models a register file, which is a component commonly used in processors and other digital systems to store data. The module has several inputs and outputs that are used to read and write data to the register file. The inputs to the module include:

• "**clk":** This is the clock signal that is used to synchronize the operations of the register file.

**• "raddr0" and "raddr1":** These are the 5-bit read addresses for the two read ports of the register file. They specify the addresses of the registers that should be read.

• "**waddr":** This is the 5-bit write address that specifies the location in the register file where the data should be written.

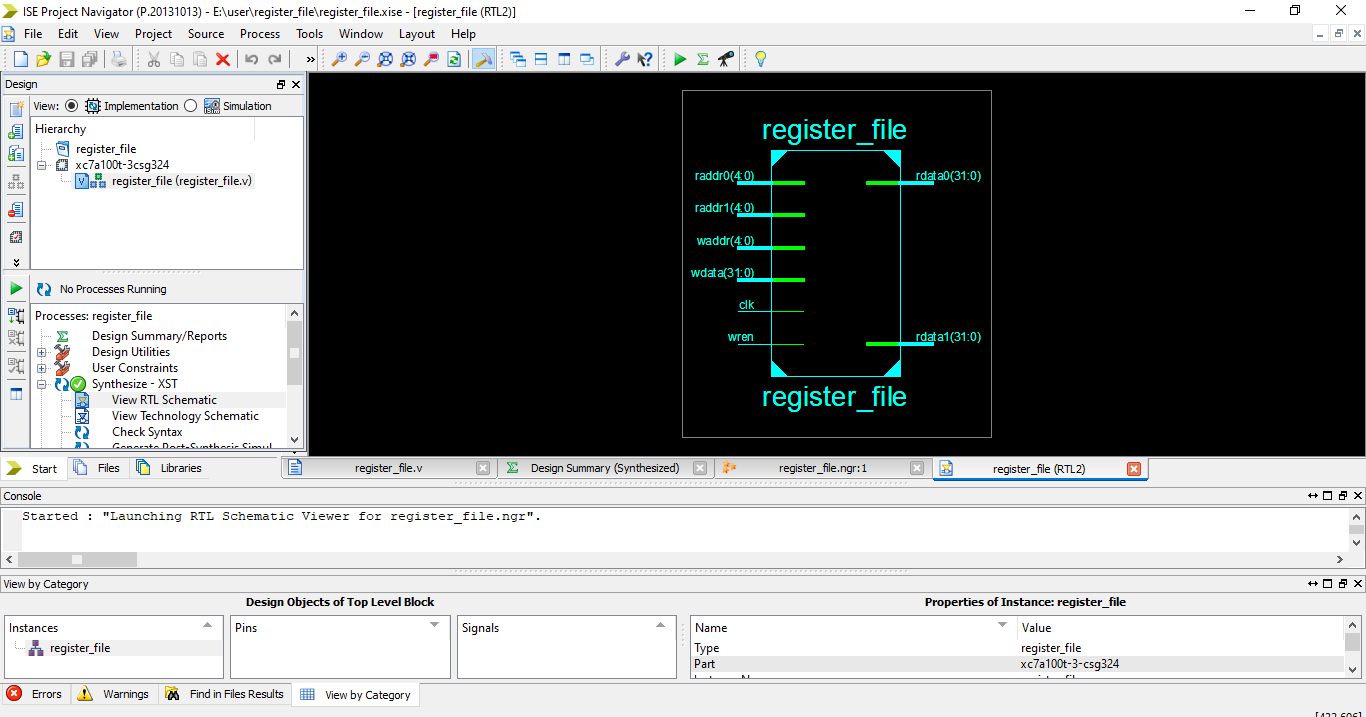
• "**wdata":** This is the 32-bit value that should be written to the specified address in the register file.

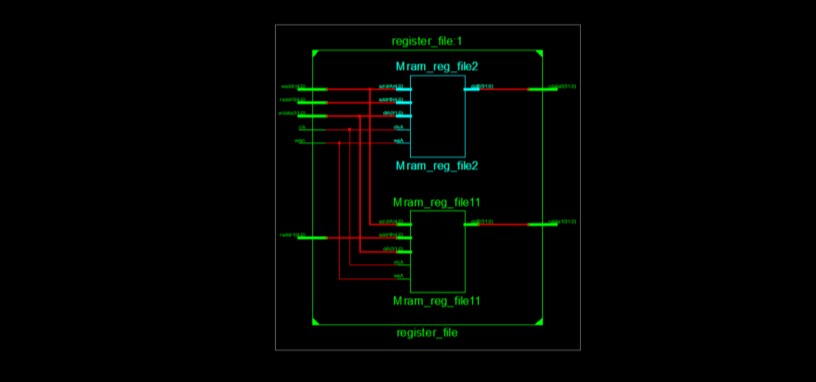
• **"wren":** This is a write enable signal. The register file only writes data when this signal is high.

The outputs of the module include:

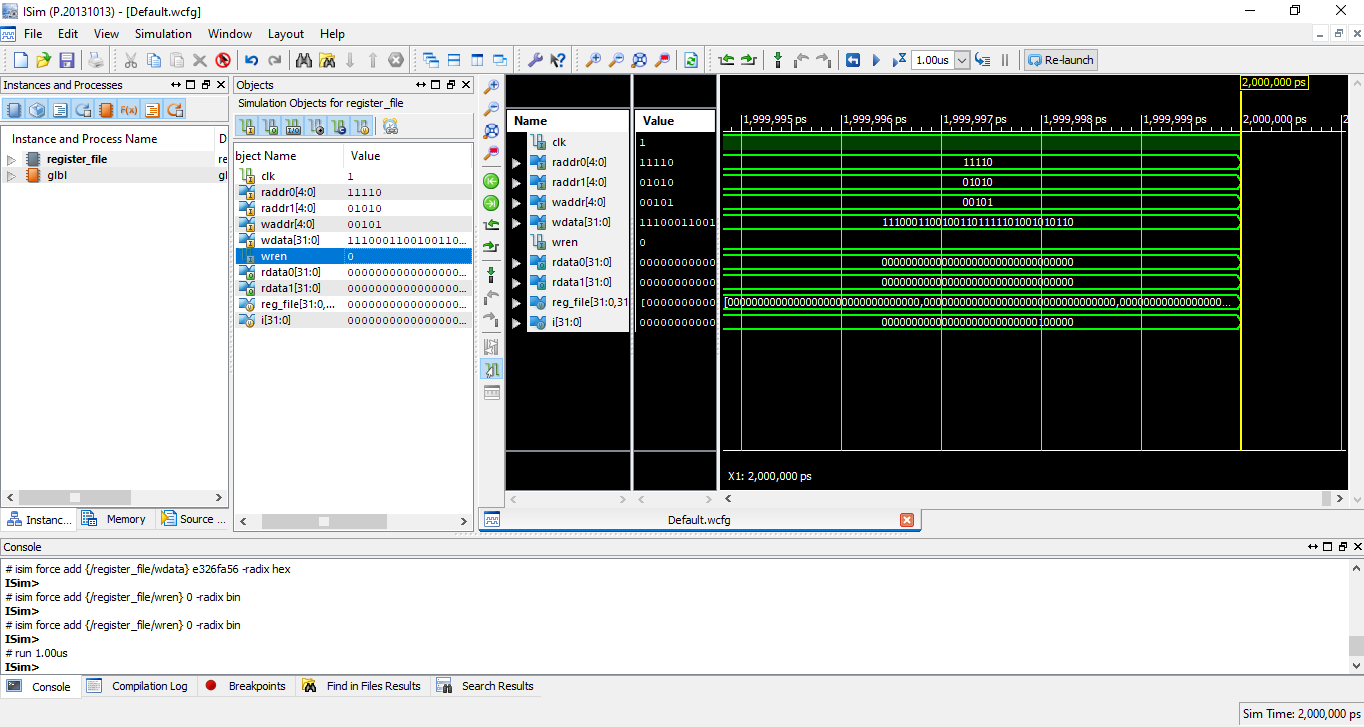
• "**rdata0" and "rdata1":** These are the 32-bit values that are read from the register file at the addresses specified by "raddr0" and "raddr1" respectively.

Implementation:





## Simulation:

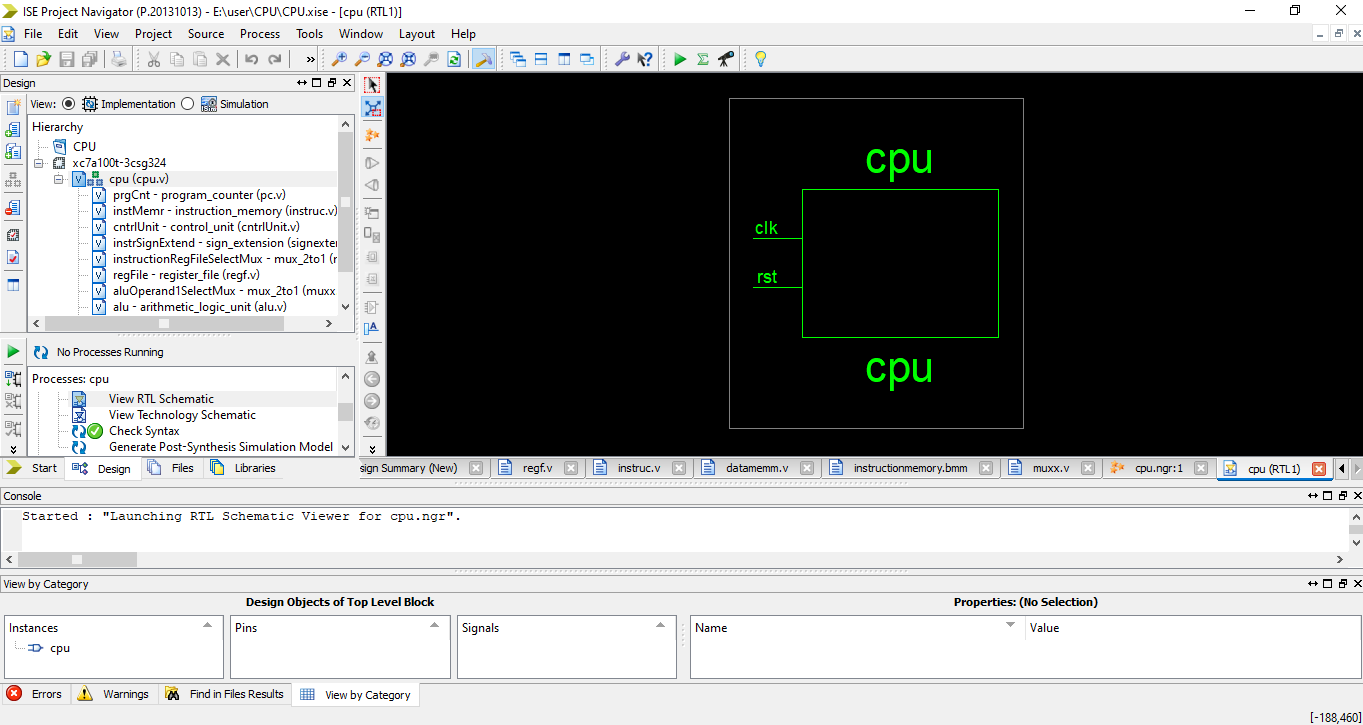


# CPU :

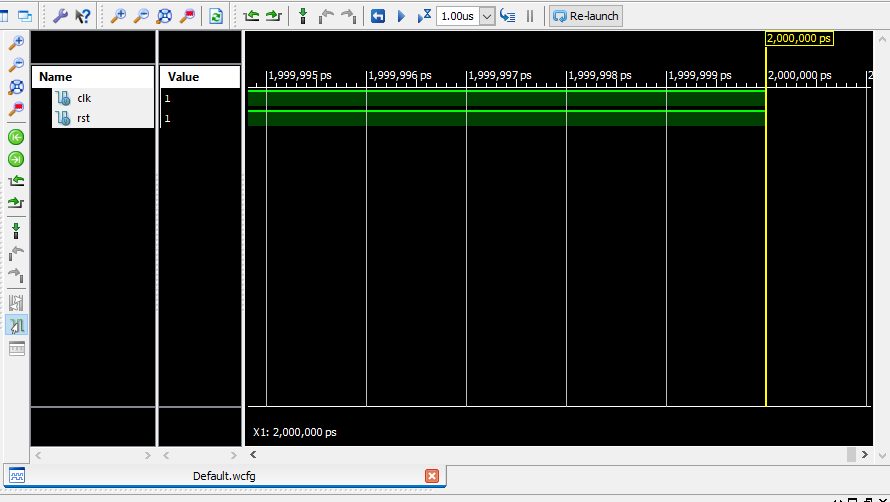
## Program Explanation:

All the code files are inserted and clk and rst are set and given as an input accordingly. Cpu will read all the files and simulate all of them and will give output.

Implementation:



## Simulation:

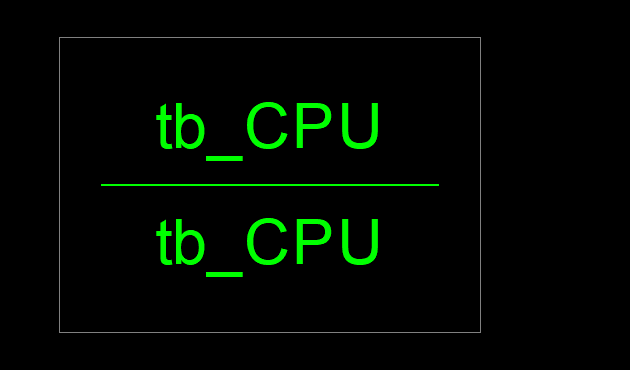


# TB CPU:

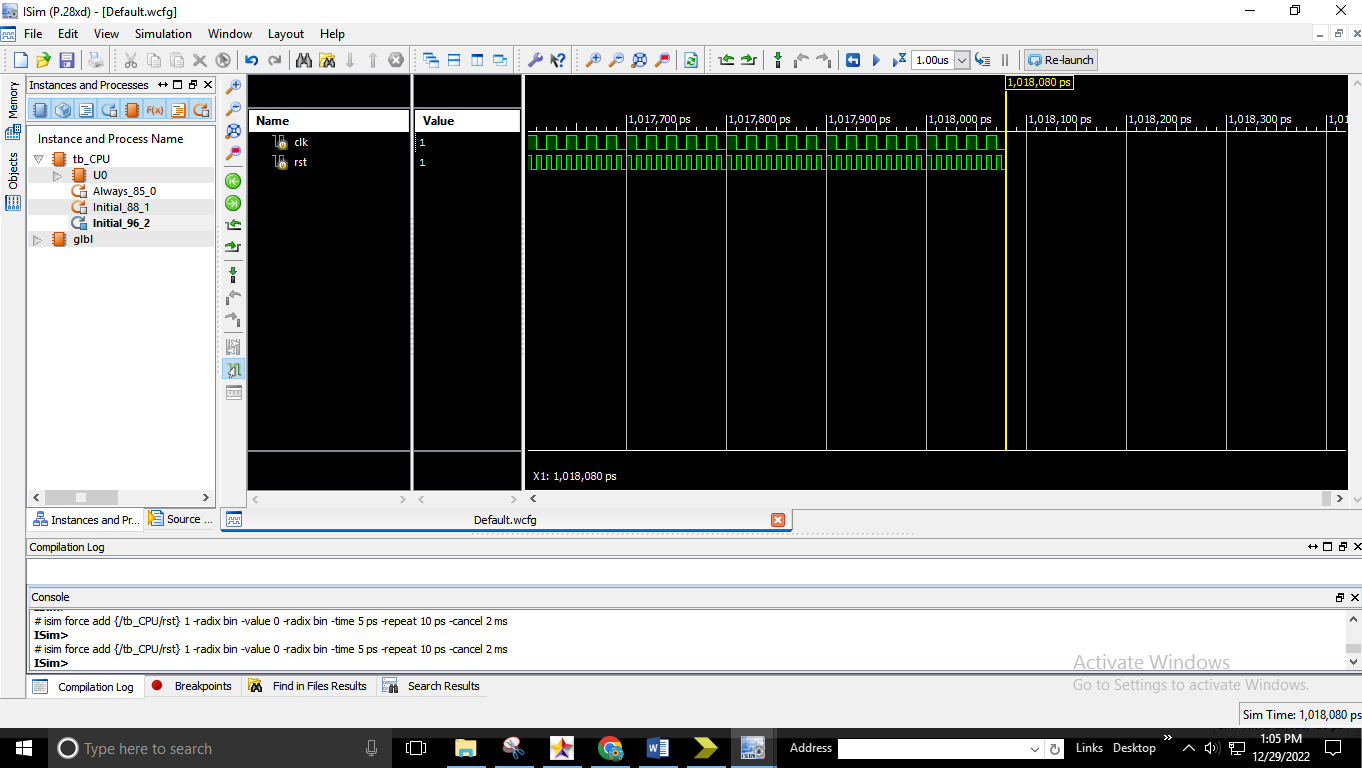
## Program Explanation:

In Tb\_CPU simulation clock and reset are set. In this reset is reset is fully synchronize with the clock. Synchronous reset means that reset is sampled with respect to clock. In other words, when reset is enabled, it will not be effective till the next active clock edge.

Implementation:



## Simulation:

****