# Assignment 7: KGP-RISC

Course: Computer Organization and Architecture Laboratory (CS39001)


**Indian Institute of Technology Kharagpur**

West Bengal, India, Pin-721302

## Group No. - 19

## Parth Tusham | 19CS30034

## Shashwat Shukla | 19CS10056

# Instruction set

| Class | Instruction | Usage | Meaning |
|---|---|---|---|
| Arithmetic | Add | add rs,rt | $rs \leftarrow (rs) + (rt)$ |
| | Comp | comp rs,rt | $rs \leftarrow$ 2's Complement $(rs)$ |
| | Add immediate | addi rs,imm | $rs \leftarrow (rs) + imm$ |
| | Complement Immediate | compi rs,imm | $rs \leftarrow$ 2's Complement $(imm)$ |
| Logic | AND | and rs,rt | $rs \leftarrow (rs) \wedge (rt)$ |
| | XOR | xor rs,rt | $rs \leftarrow (rs) \oplus (rt)$ |
| Shift | Shift left logical | shll rs, sh | $rs \leftarrow (rs)$ left-shifted by $sh$ |
| | Shift right logical | shrl rs, sh | $rs \leftarrow (rs)$ right-shifted by $sh$ |
| | Shift left logical variable | shllv rs, rt | $rs \leftarrow (rs)$ left-shifted by $(rt)$ |
| | Shift right logical | shrl rs, rt | $rs \leftarrow (rs)$ right-shifted by $(rt)$ |
| | Shift right arithmetic | shra rs, sh | $rs \leftarrow (rs)$ arithmetic right-shifted by $sh$ |
| | Shift right arithmetic variable | shrav rs, rt | $rs \leftarrow (rs)$ right-shifted by $(rt)$ |
| Memory | Load Word | lw rt,imm(rs) | $rt \leftarrow mem[(rs) + imm]$ |
| | Store Word | sw rt,imm,(rs) | $mem[(rs) + imm] \leftarrow (rt)$ |
| Branch | Unconditional branch | b L | goto L |
| | Branch Register | br rs | goto (rs) |
| | Branch on less than 0 | bltz rs,L | if$(rs) < 0$ then goto $L$ |
| | Branch on flag zero | bz rs,L | if $(rs) = 0$ then goto $L$ |
| | Branch on flag not zero | bnz rs,L | if$(rs) \neq 0$ then goto $L$ |
| | Branch and link | bl L | goto L; 31 $\leftarrow$ (PC)+4 |
| | Branch on Carry | bcy L | goto L if Carry = 1 |
| | Branch on No Carry | bncy L | goto L if Carry = 0 |

# Instruction Decoding

| Instruction | Opcode | Function code |
|---|---|---|
| **Add** | **000** | **0000** |
| **Xor** | | **0001** |
| **And** | | **0010** |
| **Comp** | | **0011** |
| **SHLL** | | **0100** |
| **SHRL** | | **0101** |
| **SHLLV** | | **0110** |
| **SHLRV** | | **0111** |
| **SHRA** | | **1000** |
| **SHRAV** | | **1001** |
| **Compi** | **001** | **0000** |

| | | |
|---|---|---|
| **Addi** | | **0001** |
| **LW** | **010** | **0000** |
| **SW** | | **0001** |
| **Unconditional branch** | **011** | **0000** |
| **Branch on less than zero** | | **0010** |
| **Branch on flag zero** | | **0001** |
| **Branch on flag not zero** | | **0011** |
| **Branch and link** | | **0100** |
| **Branch on carry** | | **0101** |
| **Branch on no carry overflow** | | **0101** |
| **Branch register** | **100** | **000** |

## Instruction format

**1) opcode: 000**

| OPCODE | rs | rt | shamt | func | offset |
|---|---|---|---|---|---|
| 3 | 5 | 5 | 5 | 4 | 10 |

**2) opcode: 001**

| OPCODE | rs | immediate | func |
|---|---|---|---|
| 3 | 5 | 22 | 2 |

**3) opcode: 010**

| OPCODE | rs | rt | immediate | func |
|---|---|---|---|---|
| 3 | 5 | 5 | 18 | 1 |

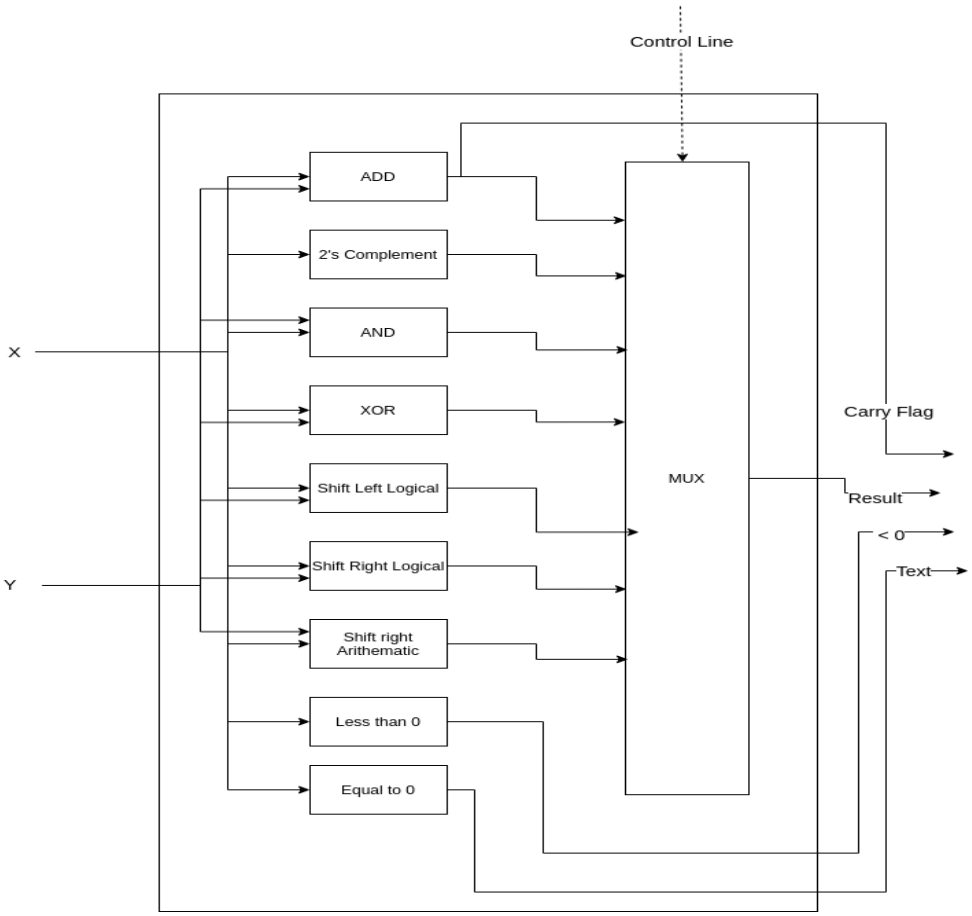
**4) opcode: 011**

| OPCODE | Label | func |
|---|---|---|
| 3 | 25 | 4 |

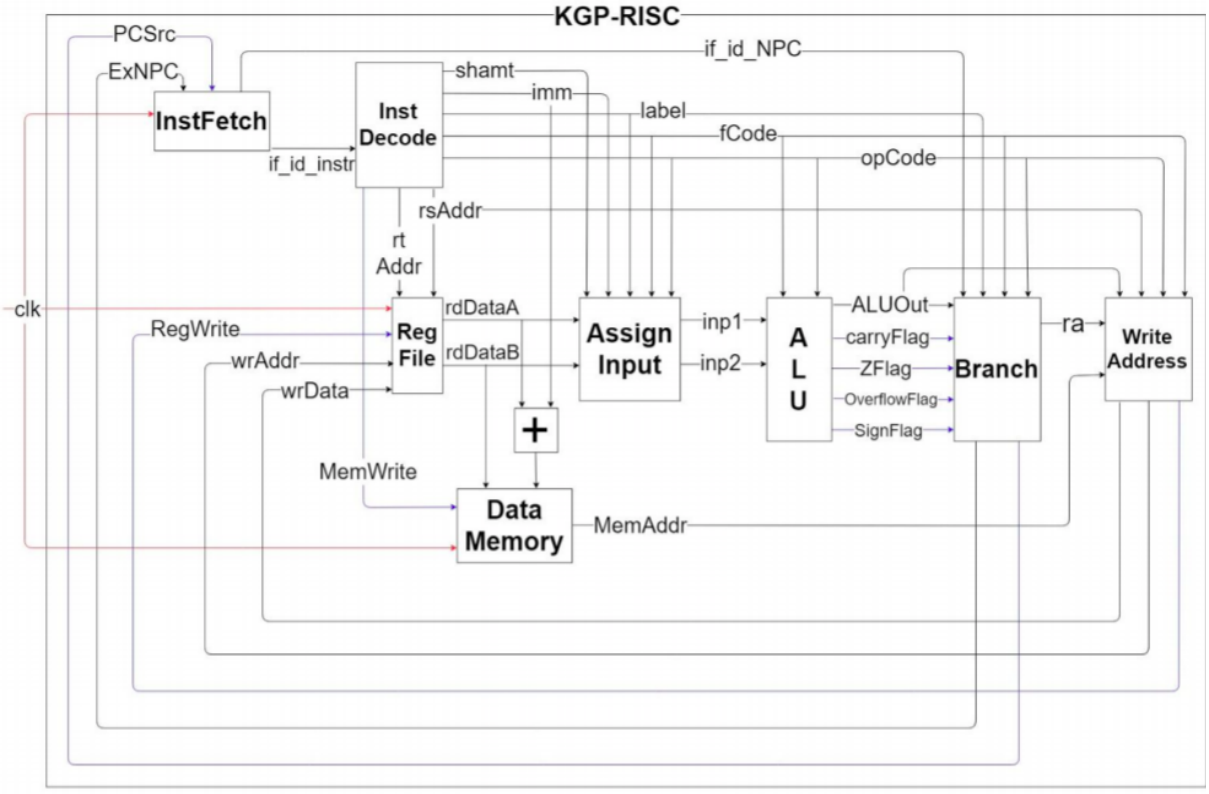
**5) opcode: 100**

| OPCODE | rs | Label |
|---|---|---|
| 3 | 5 | 24 |

# Architecture




**ALU**




**PROCESSOR**

**Note:** Block RAMs have a peculiar issue in that they have a significant delay in fetching data (around 0.5 clock cycles) from the RAM. Hence, we have divided the input clock into two parts, a slower one and a faster one. The faster one is eight times faster and is used to fetch data from Block RAM whereas the slower one is used for other modules

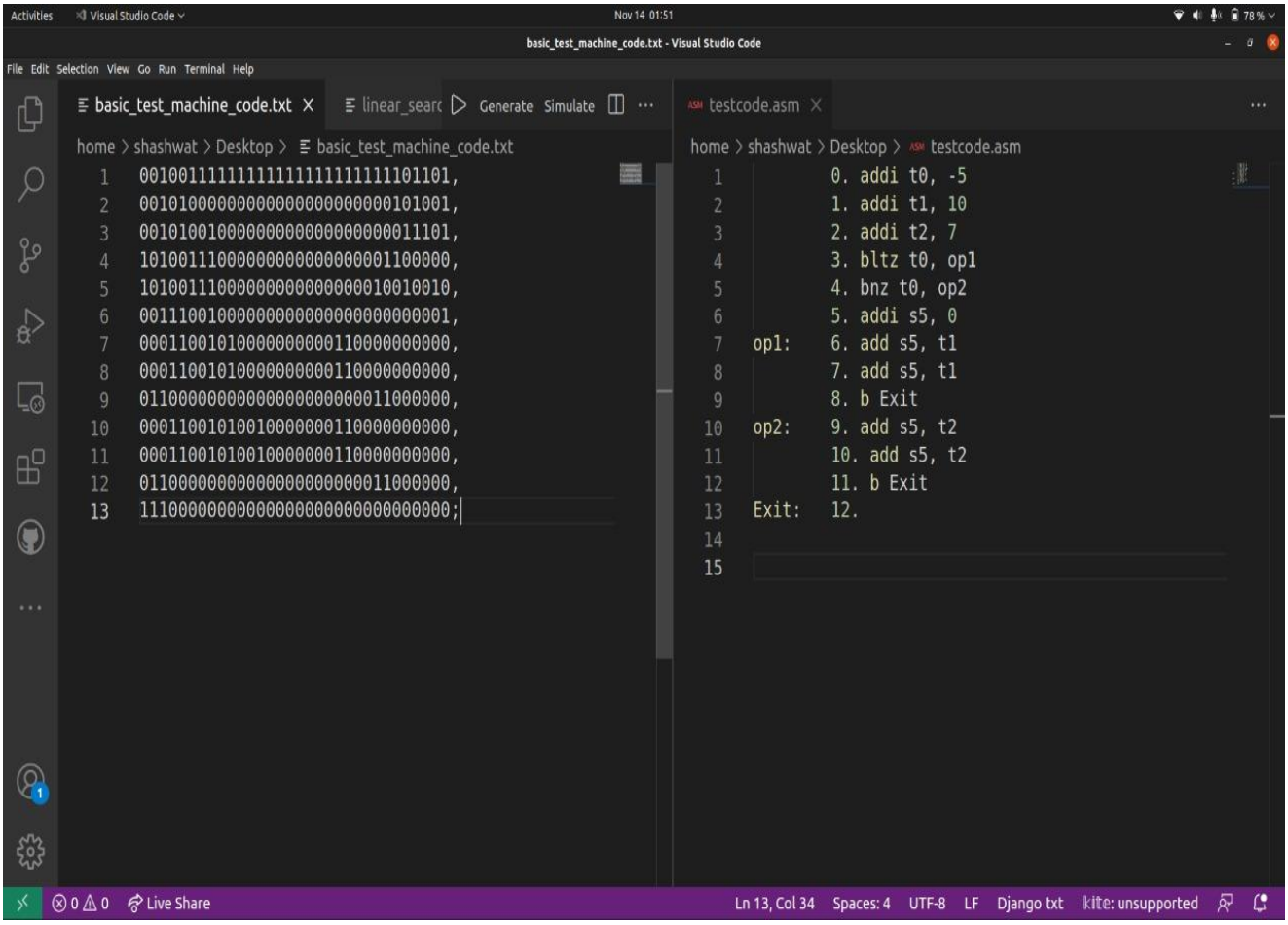
# Design Summary

- **Instruction Fetch Module**: Used to fetch the instruction from the BRAM based on the PC
- **Instruction Decoder Module:** Splits the instruction into opcode, fcode, rsAddress, rtAddress, imm, label, and shamt. Also, it decides if we need to write anything in Data Memory or not using MemWrite.
- **RegFile32x32**: Register file, used to write into and read from registers. Allows two registers to be read at a time.
- **assignInputs**: Based on the opcode and fcode, this module assigns what two inputs will be present in the ALU. for example, in arithmetic instructions, it is rs and rt whereas in shift instructions it is rs and shamt.
- **ALU:** Based on opcode, fcode, and inputs, computes the flags and Output of ALU.
- **Branch**: Decides the value of PCSrc (flag) and ExNPC from the opcode and fcode, hence adding the branching part to the processor.

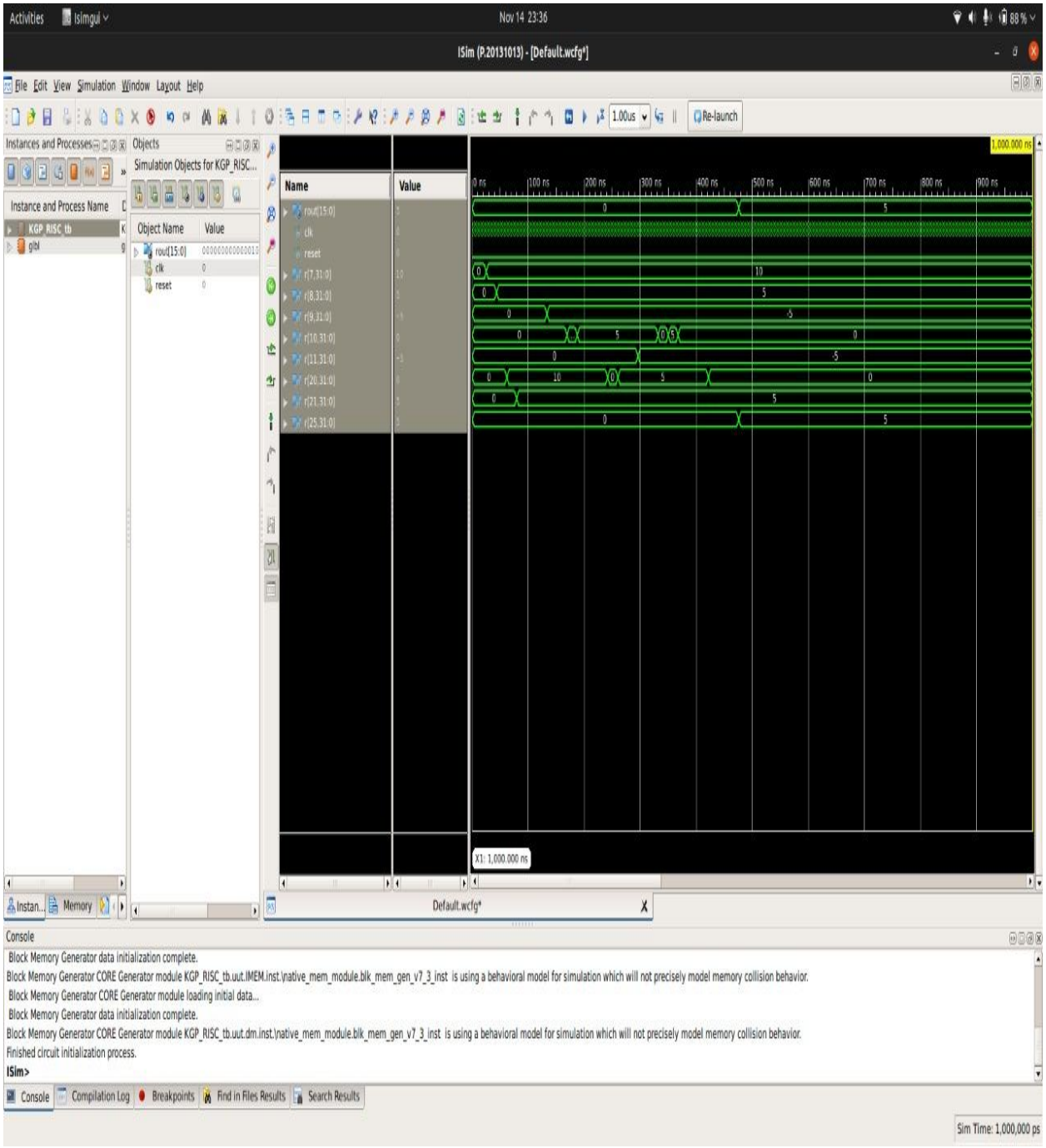
- **writeAddress**: Decides the values of Address of the register, data, and the RegWrite flag which is used as input to write data into ALU
- **Data Memory**: Block RAM is used to load and store data elements.

# Simulation Screenshot

## 1) GCD of 5 and 10




Assembly code and instruction code


Activities
Isimgui
Nov 14 23:36
ISim (P.20131013) - [Default.wcfg*]
File Edit View Simulation Window Layout Help
Re-launch
Instances and Processes
Objects
Simulation Objects for KGP_RISC...
Instance and Process Name
KGP_RISC_tb
glbl
Object Name
Value
rout[15:0]
clk
reset
Name
Value
r[7,31:0]
r[8,31:0]
r[9,31:0]
r[10,31:0]
r[11,31:0]
r[20,31:0]
r[21,31:0]
r[25,31:0]
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
1,000.000 ns
X1: 1,000.000 ns
Default.wcfg*
Console
Block Memory Generator data initialization complete.
Block Memory Generator CORE Generator module KGP_RISC_tb.uut.IMEM.inst.\native_mem_module.blk_mem_gen_v7_3_inst is using a behavioral model for simulation which will not precisely model memory collision behavior.
Block Memory Generator CORE Generator module loading initial data...
Block Memory Generator data initialization complete.
Block Memory Generator CORE Generator module KGP_RISC_tb.uut.dm.inst.\native_mem_module.blk_mem_gen_v7_3_inst is using a behavioral model for simulation which will not precisely model memory collision behavior.
Finished circuit initialization process.
ISim>
Console
Compilation Log
Breakpoints
Find in Files Results
Search Results
Sim Time: 1,000,000 ps

simulation

prac.cpp | pra2.cpp

```cpp
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    else if (b == 0)
        return a;
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}

```

Algorithm used

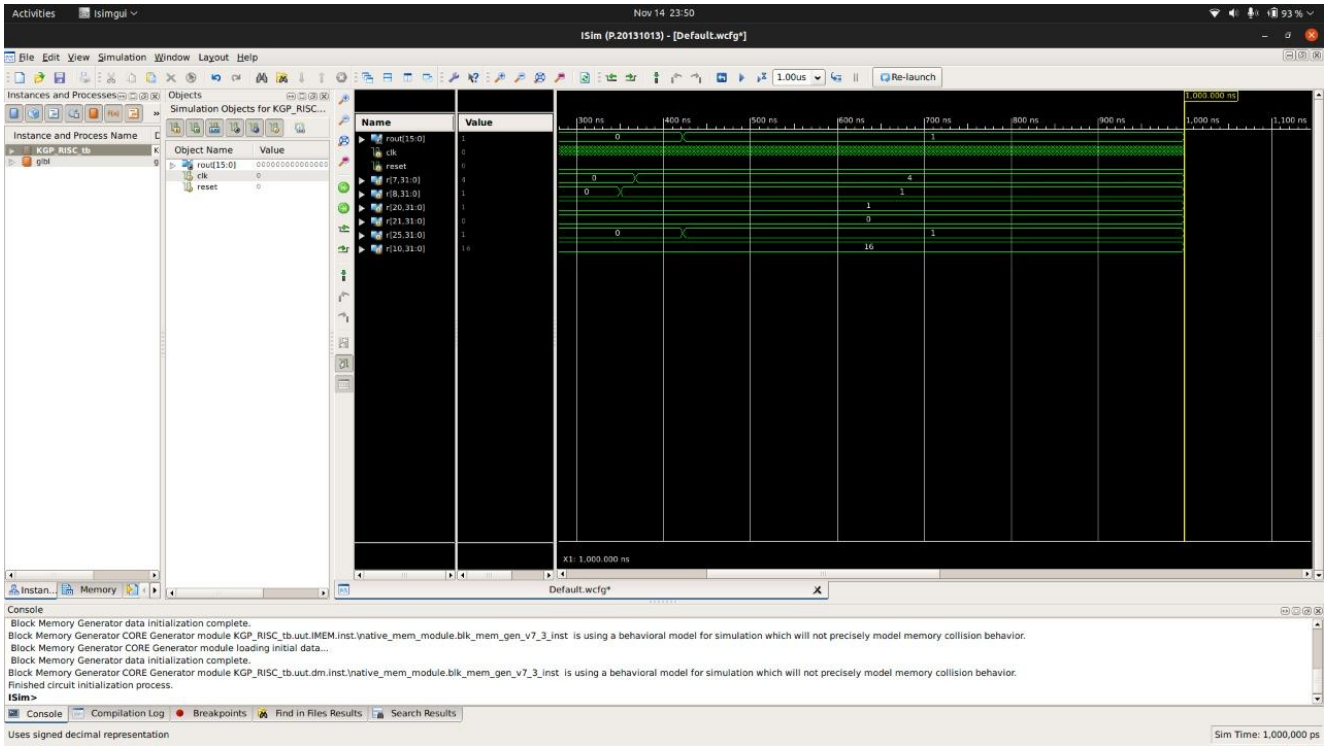
## 2) Linear search

AssemblyCode.asm

```
            0.  addi $t0 , 0
            1.  add $t1 , $zero
            2.  addi $s2 , 10
            3.  comp $s3 , $s2
            4.  addi $s3 , 1
            5.  lw $s0 , 10( $zero )
            6.  comp $s1 , $s0
            7.  addi $s1 , 1
            8.  add $s5 , $zero
LoopLabel : 9.  add $t2 , $t1
            10. add $t2 , $s3
            11. bz Exit
            12. add $s7 , $zero
            13. lw $t3 , 0($t0)
            14. add $t4 , $t3
            15. add $t4 , $s1
            16. bz Match
            17. add $s7 , $zero
            18. addi $t1 , 1
            19. addi $t0 , 4
            20. b LoopLabel
            21. add $s7 , $zero
Match :     22. addi $s5 , 1
            23. b Exit
            24. add $s7 , $zero
Exit :      25.
```

MachineCode.cpp

```
00100111000000000000000000000001,
00001000000000000000110000000000,
00110110000000000000000000101001,
00010111101100000000100000000000,
00110111000000000000000000000101,
01010100000000000000000000010100,
00010101101000000000100000000000,
00110101000000000000000000000101,
00011001000000000000110000000000,
00001001010000000000110000000000,
00001001101110000000110000000000,
01100000000000000000000110010001,
00011011000000000000110000000000,
01001010001110000000000000000000,
00001011010100000000110000000000,
00001001101010000000110000000000,
01100000000000000000000101100001,
00011011000000000000110000000000,
00101000000000000000000000000101,
00100111000000000000000000010001,
00100000000000000000000010010000,
00011011000000000000110000000000,
00111001000000000000000000000101,
01100000000000000000000110010000,
00011011000000000000110000000000,
11100000000000000000000000000000;
```




- The above assembly language code is written according to the assembly language specification provided to us. The result of the Linear Search is stored in the register $s5, which corresponds to the 26th register with an address of 25 (11001) in our Memory. The Register holds a Zero Value in case there is No Match and a Non-Zero value in case there is a Match. The corresponding Machine Language Instruction Sequence is generated according to the encoding of each instruction in our language specification, with appropriate values assigned to the opCode, rs, rt, shamt, imm, label, code depending on the instruction and the registers used. The last instruction with the first three bits set indicates an opCode of 7, which does not fall in any class of our specification, hence leading to the termination of the program. In the above Screenshot, each Assembly Language Instruction ( left panel ) has its equivalent 32-bit Binary Representation in the Machine Language ( right panel ).