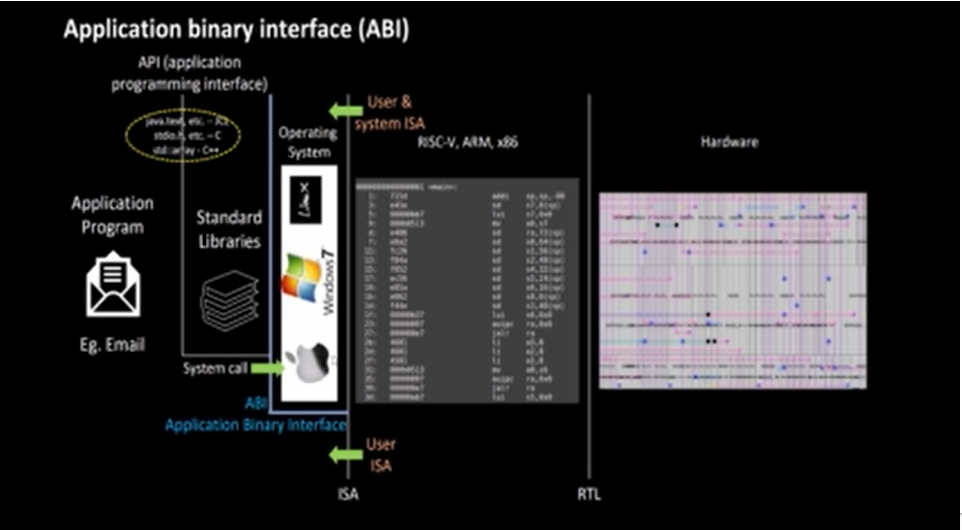
**APPLICATION BINARY INTERFACE**

For users when talking about interface in terms of any application, what matters is the appearance and functionality. Any application programmer has to write multiple layers of programme for an application to run on hardware.



Through the above diagram we realise the process which is going on

1.For first building interface we need to use some standard libraries which can be through Java, C++ or C this interface is called Application Programming Interface which can access OS and std.lib via another interface.

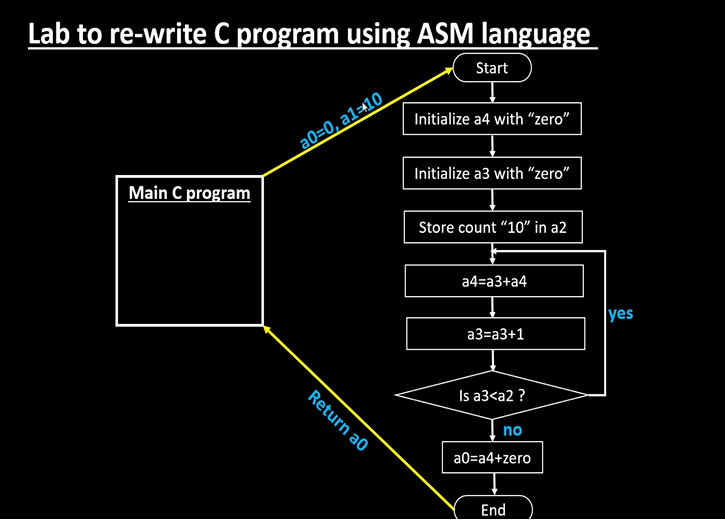
2. From Operating system in order to reach the machine language we will be using ISA. Parts of ISA which are accessible to programmer is called USER ISA and parts of ISA which are accessible to OS are called User and System ISA.

3. For this architecture to be implemented on hardware we have interface called RTL which implements specifications.

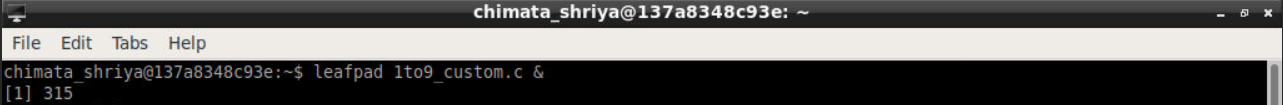
4. Through system call the programmer can access parts (registers of RISC-V architecture) of OS. And this interface is called as Application Binary Interface (System Call Interface) because if an application programmer wants to access hardware resources of processors then it has to do via registers only.

**TO WRITE C PROGRAM OF COMPUTING SUM OF NUMBERS 1 TO 9 USING ASM LANGUAGE**

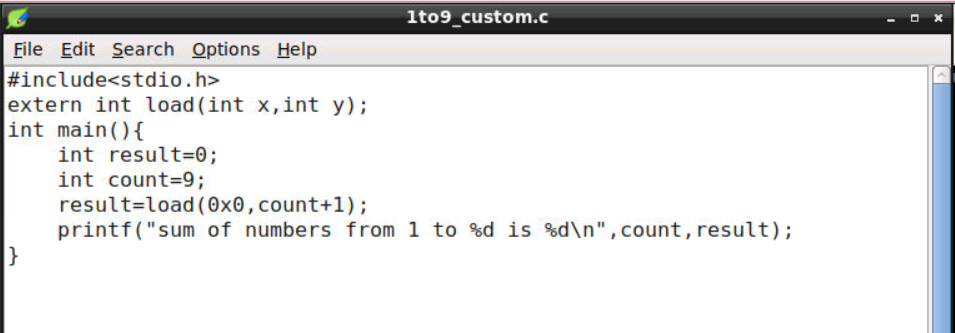
1. **ALGORTITHM**



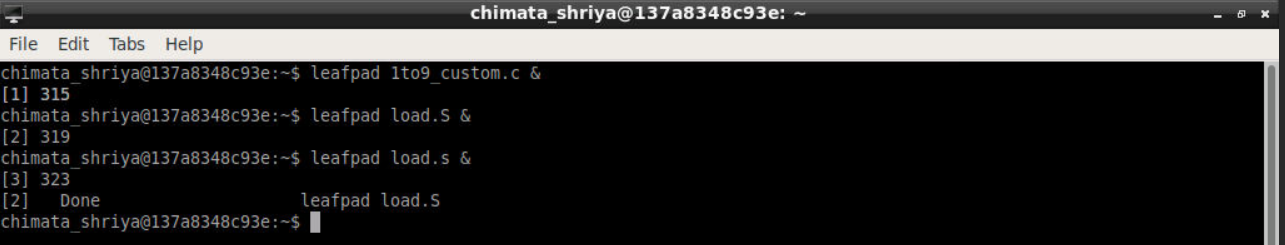
**STEP 1:** Opening a leafpad



**STEP 2:** Writing a C programme for sum up to 1 to 9 numbers



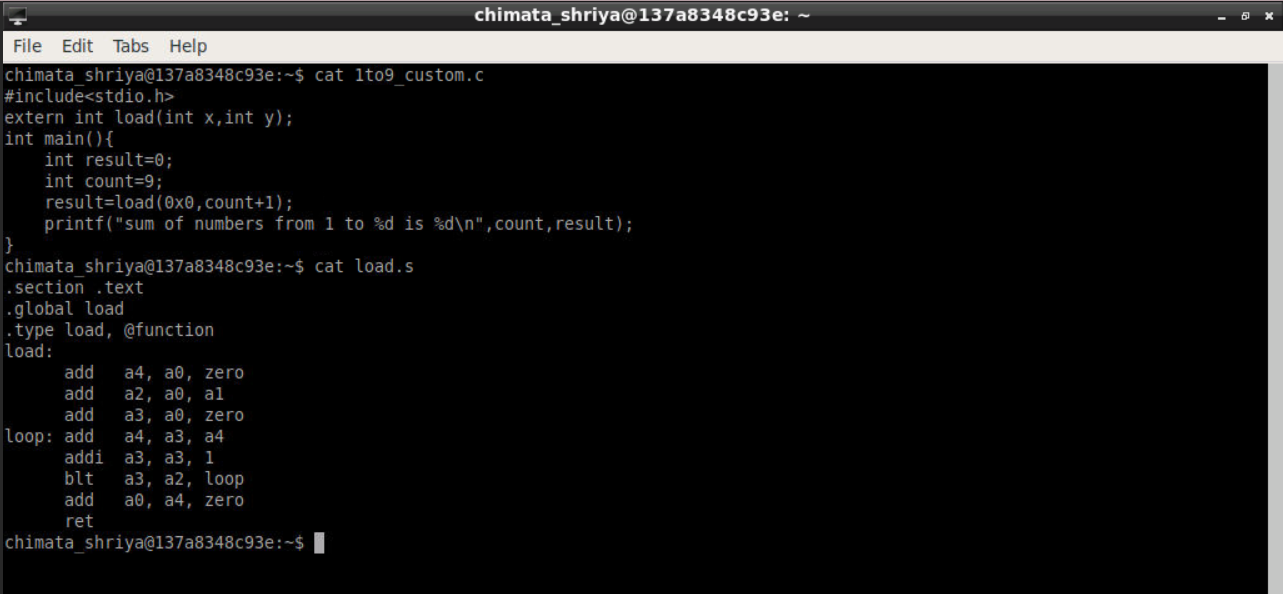
**STEP 3:** Opening another window for load of this programme where same programme will be written but in assembly language



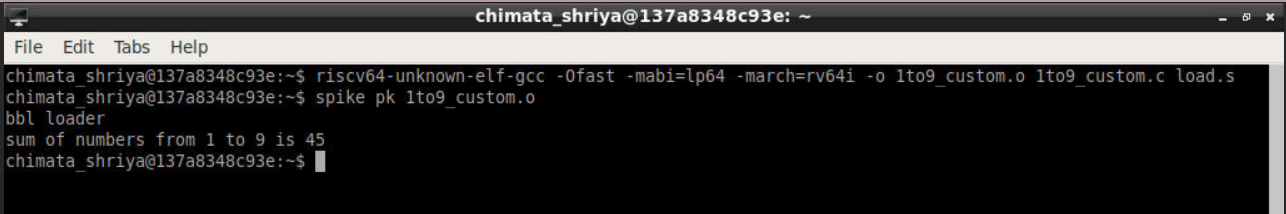
**STEP 4:** Writing the same programme in assembly language



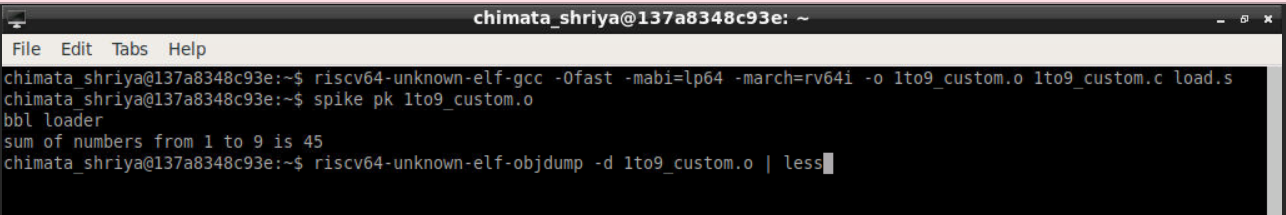
**STEP 5:** using cat to show both programs in terminal itself



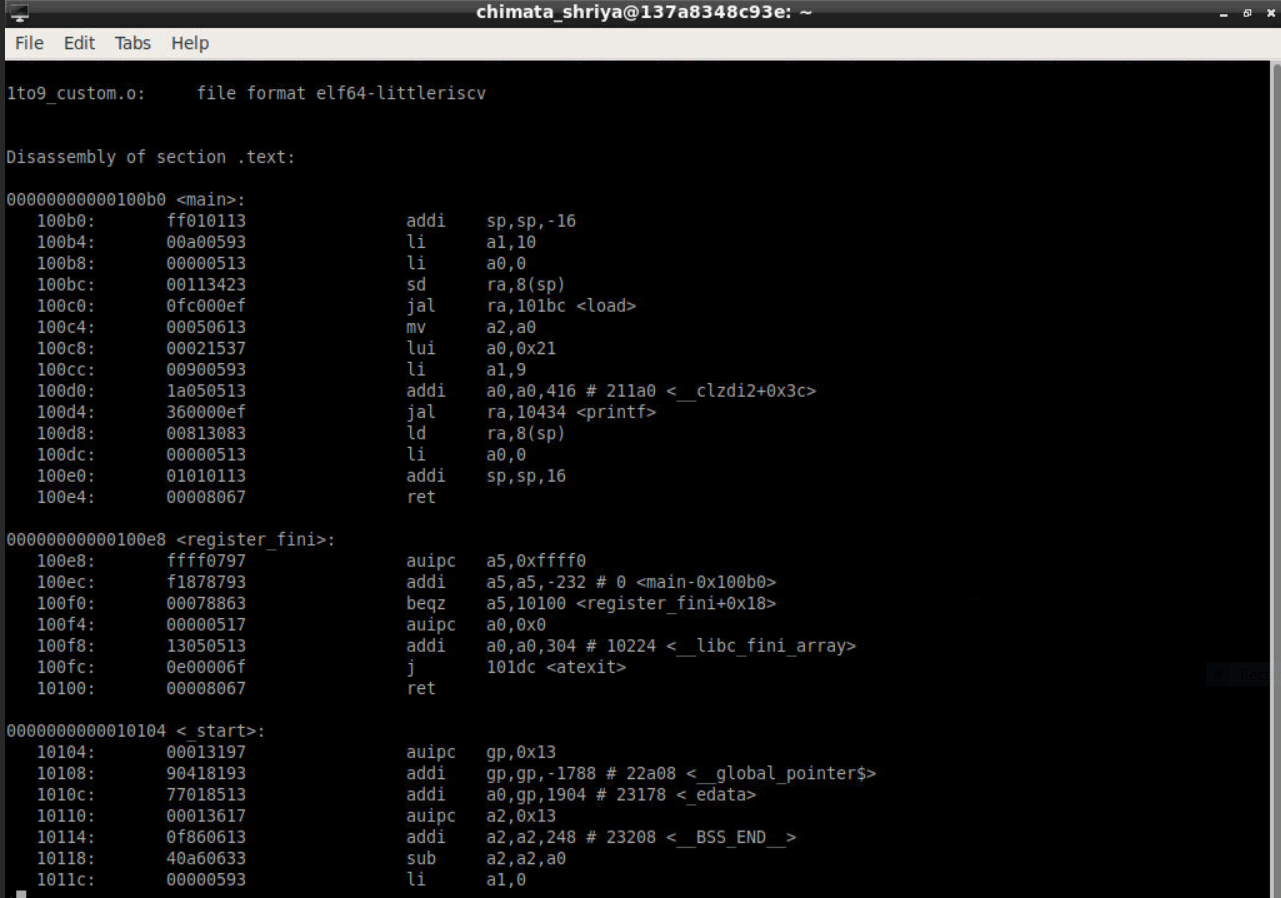
**STEP 6:** Running the programme in RISCV and checking the answer which matches exactly as the previous one. Sum of numbers 1 to 9 is 45.



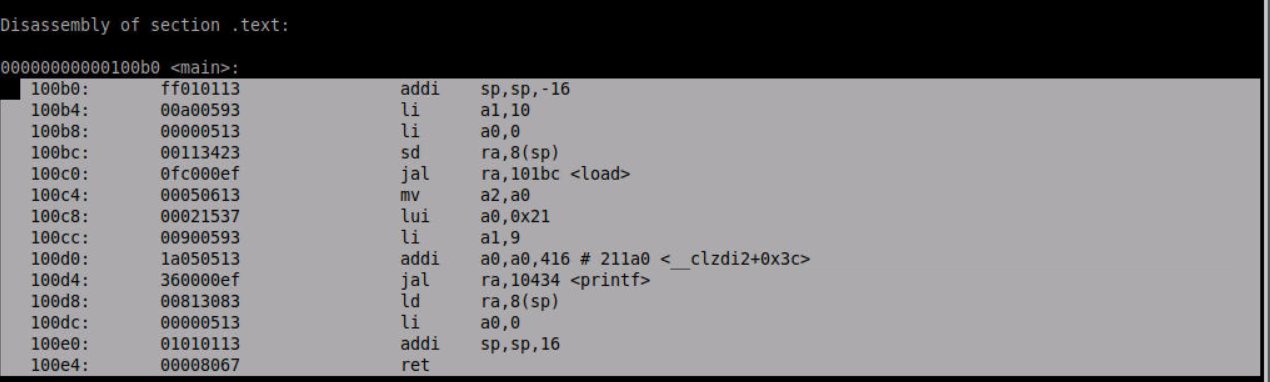
**STEP 7:** Using objdump and pipe less command to monitor disassembly window

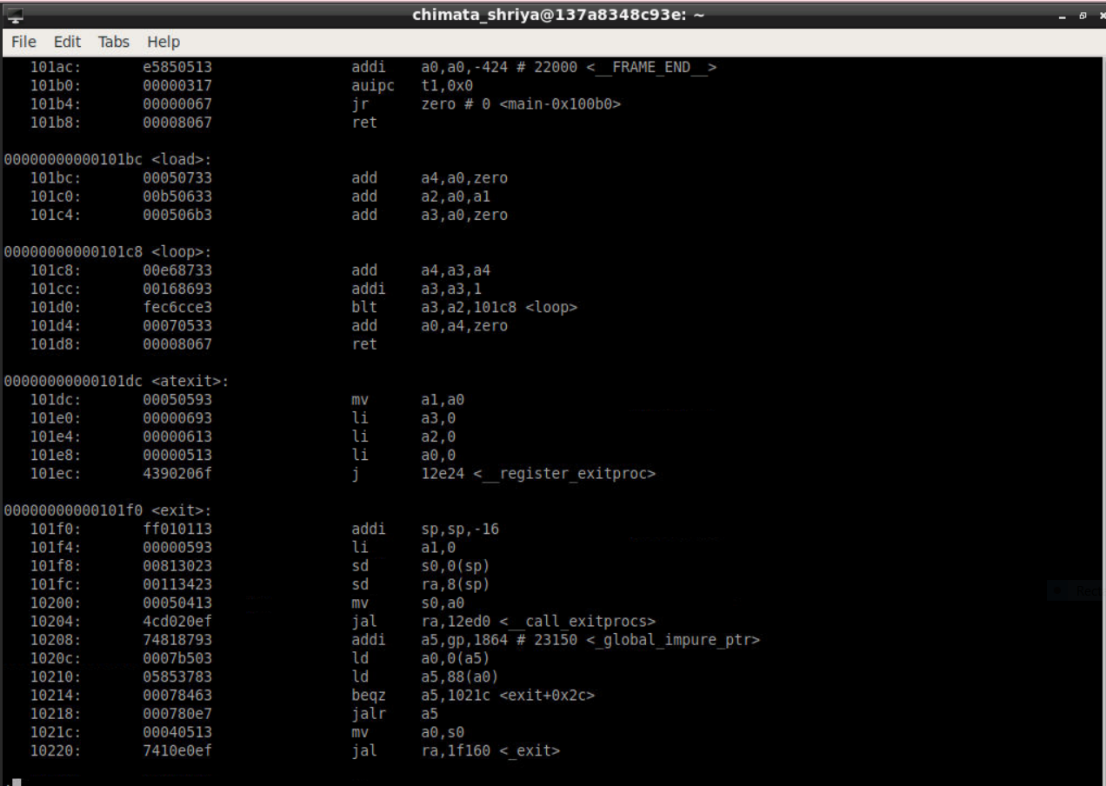


**STEP 8:** Observing main in the disassembly window

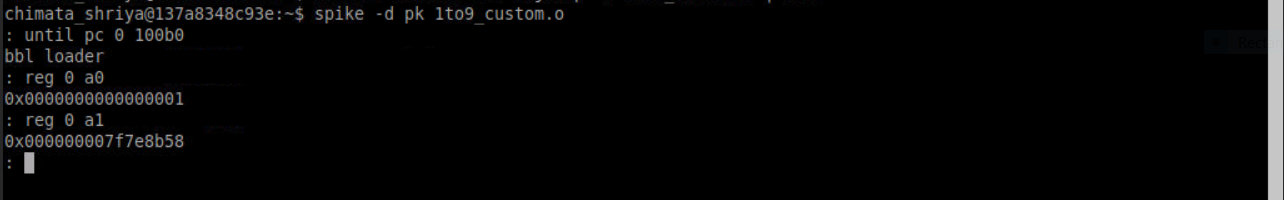


**STEP 9:** The programme is running as we wrote it and wanted it to .

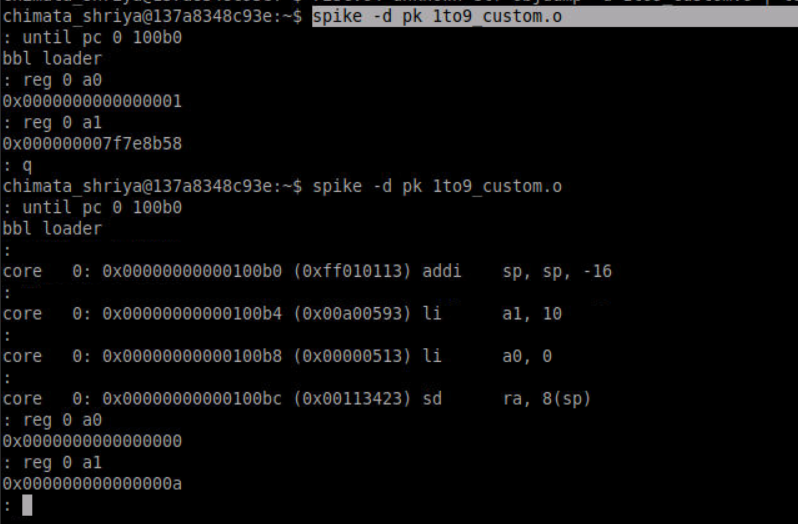


**STEP 10:** To be able to check memory loaction of loop and load subroutine 

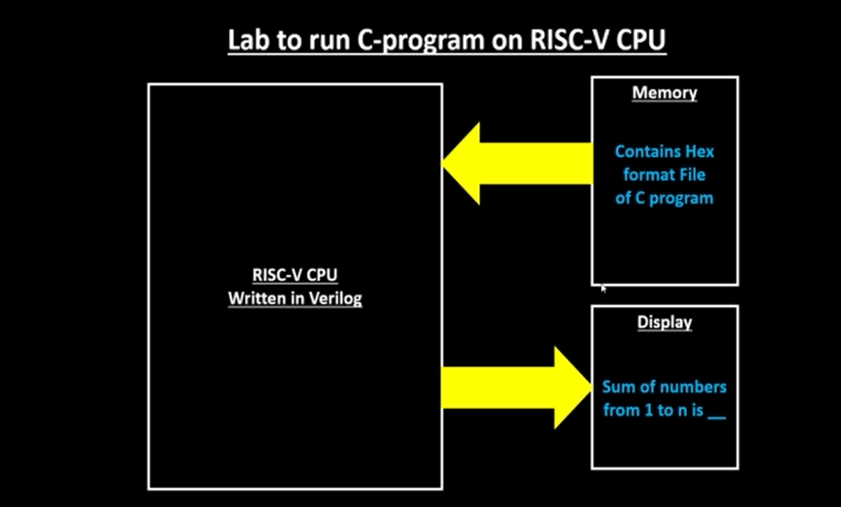
**STEP 11:** To run the program from 0 till 100b0 and finding values of registers a0 and a1



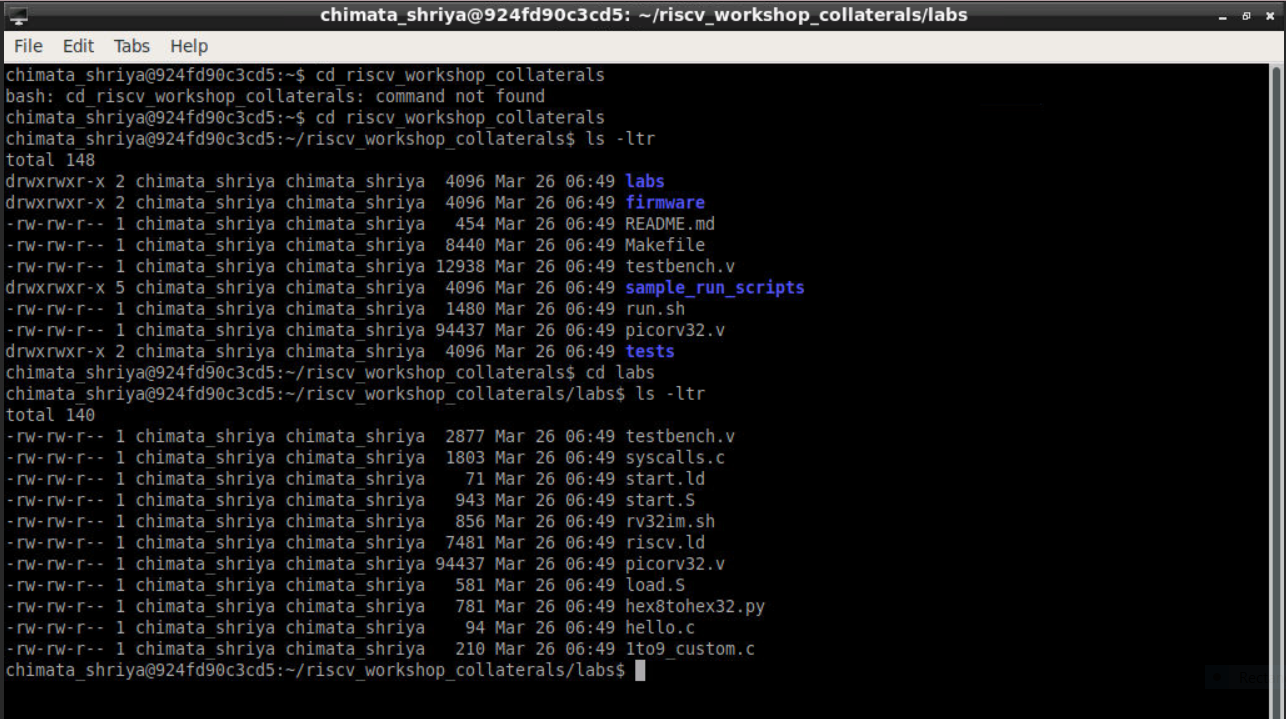
**STEP 12:** To run the program from 0 till 100b0c and finding values of registers a0 and a1



**TO RUN A C PROGRAM ON RISCV-CPU**



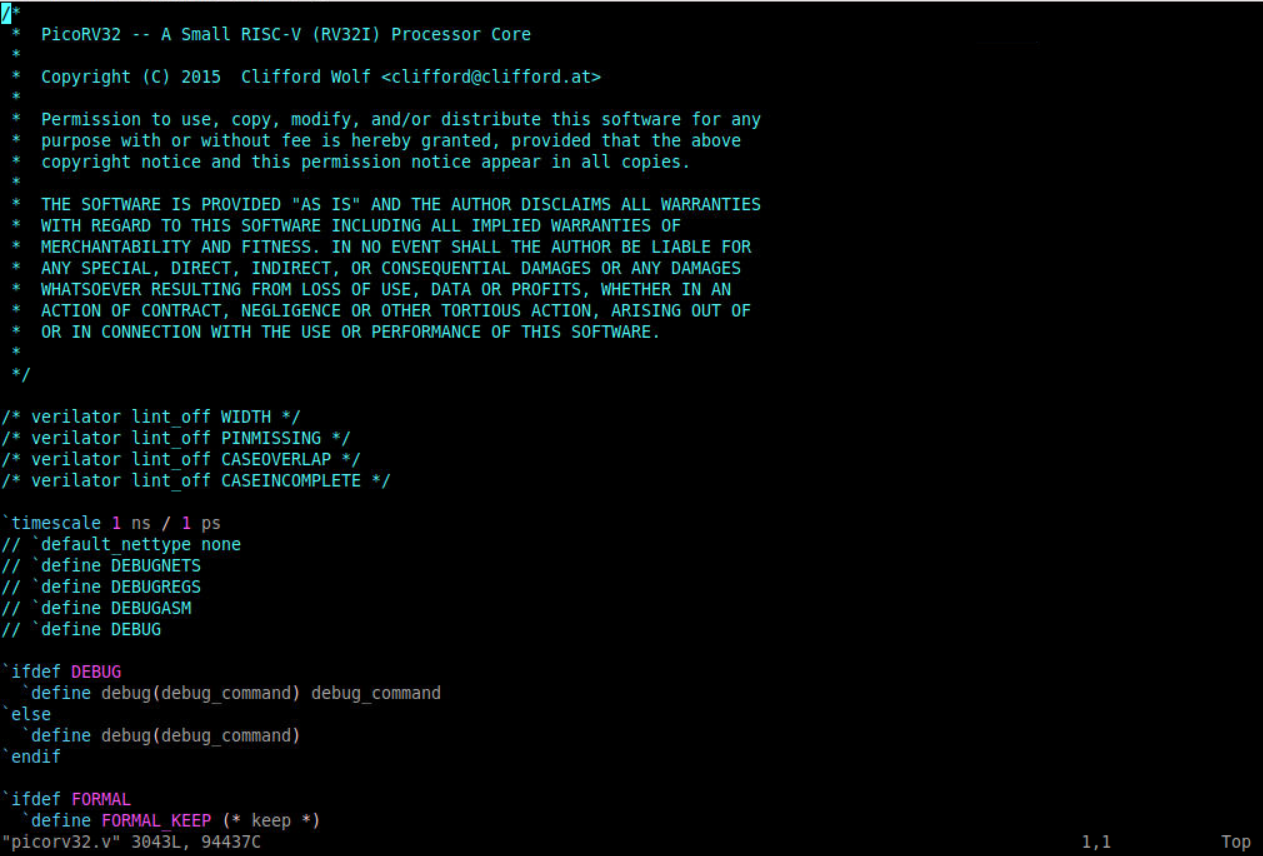
**STEP 1:** cd and then writing <https://github.com/kunalg123/riscv_workshop_collaterals.git> to downlaod some basic labs.

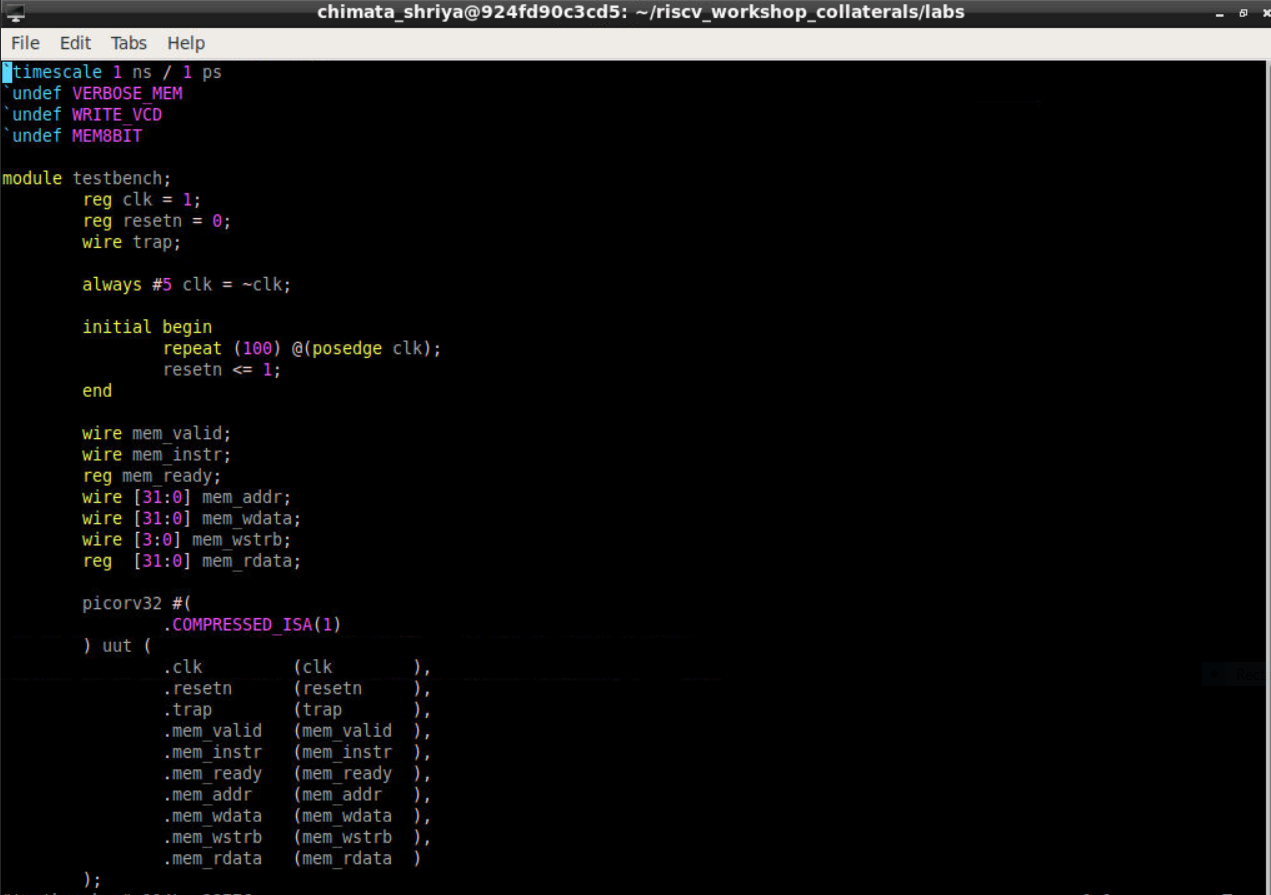


**STEP 2:** picorv32.v is for our RISCV CPU whereas testbench is for verifying entire thing where picor v32 is unit under test(uut) and it is the place where we give all the initial conditions. Testbench is where we read the hex files into memory.

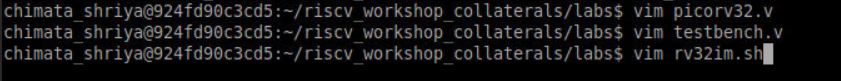


**STEP 3:** After running the above two commands one after the other we will get picorv32 and testbench which look as shown below

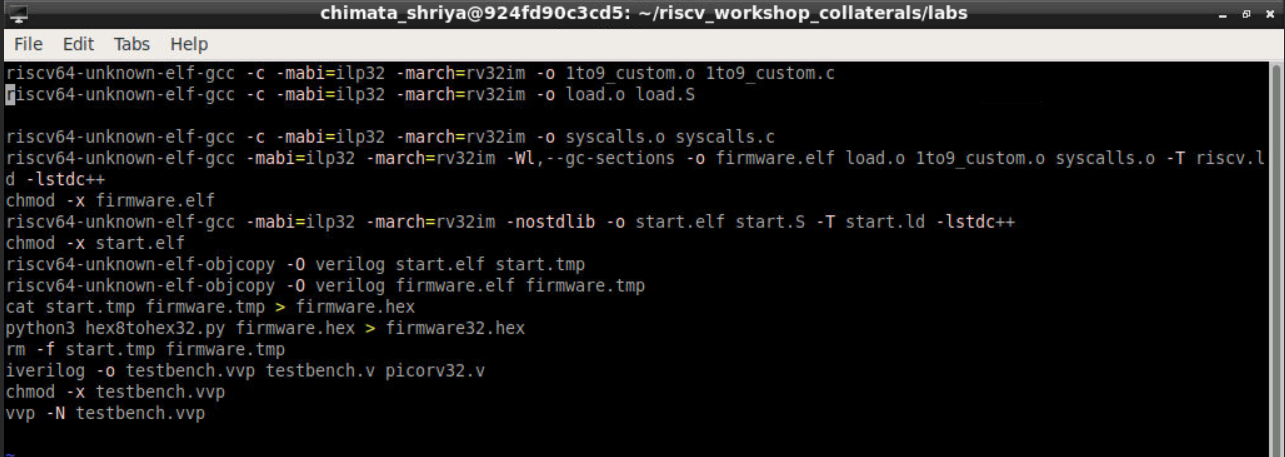




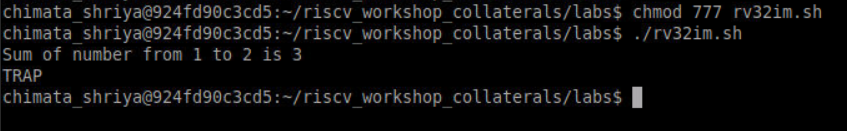
**STEP 4:** Now we would be running scripts which will help in conversion into hex file and then load it into CPU for running.



**STEP 5:** We can observe mabi and march for rv32i here

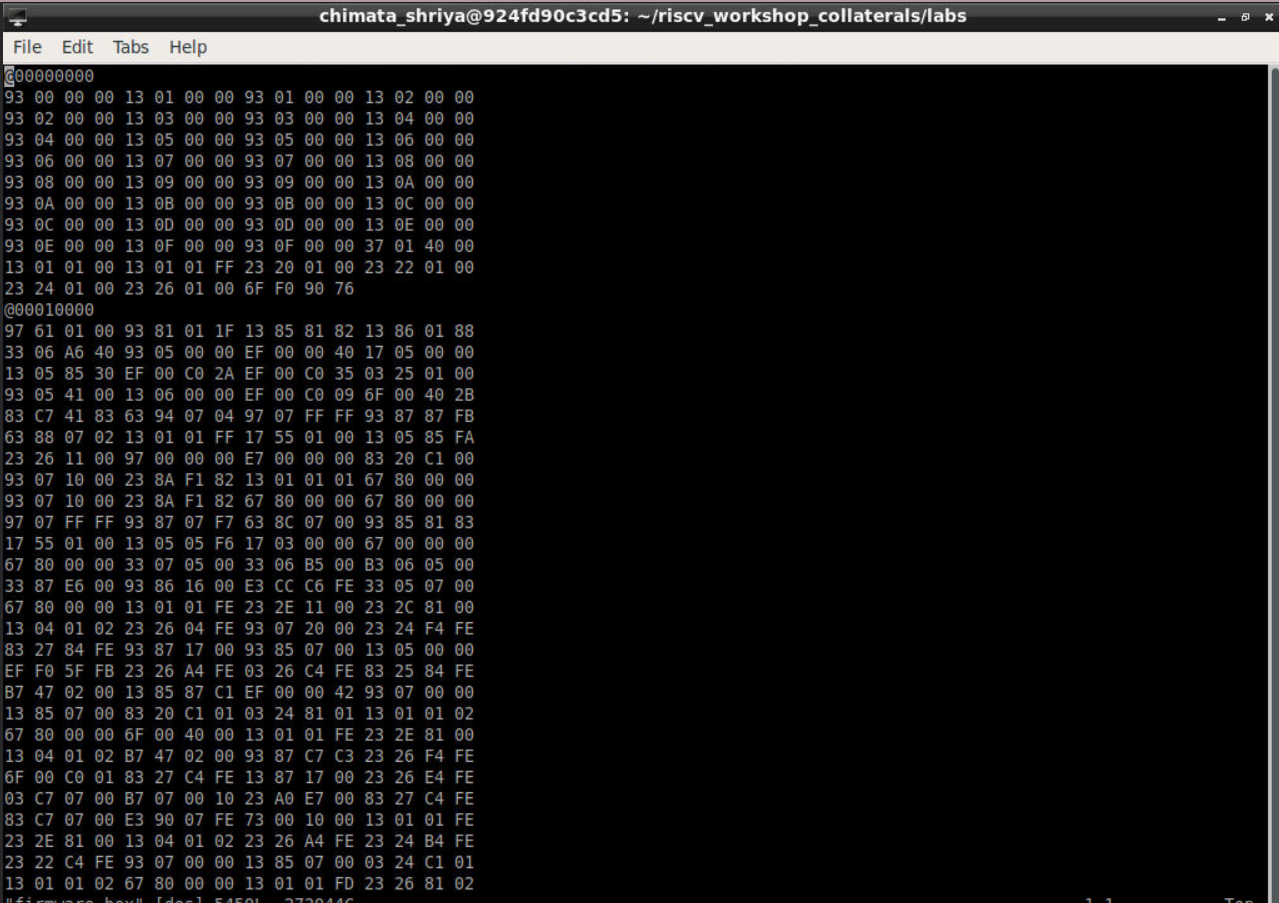


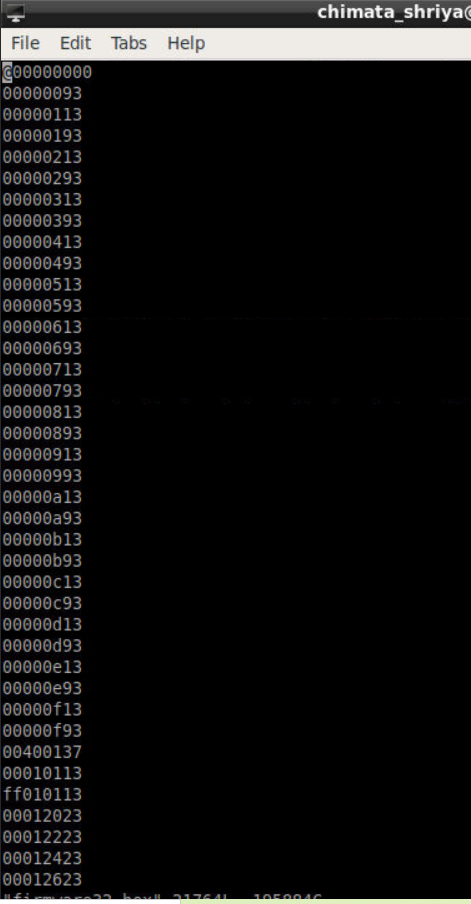
**STEP 6:** For running this we use the following commands. We get output that is sum of 1 to 2 will be 3 and rightly so



**STEP 7:** The script will create a hex file to access the hex file we use following command



**STEP 8:** We see bitstream which gets loaded into firmware. Using vim firmware32.hex will give how exactly bitstream looks like



**SUMMARY**

So basically, the testbench will read the memory and that is how it will get loaded inside testbench and the it will be used by picorv32(uut) and this is being processed under core and finally displays output