NAND TO TETRIS

J Component Project Report for the course

CSE2005 Operating Systems

by

Maulishree Awasthi (19BCE1864)
Rekha V(19BCE1871)
Katya Pandey(19BCE1312)

Submitted to

Dr. Phrangboklang Lyngton Thangkhiew

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER ENGINEERING
VELLORE INSTITUTE OF TECHNOLOGY
CHENNAI - 600127

June 2021

Table of Contents

ABSTRACT

Nand to Tetris is a well known learning that helps us build some very basic foundations of computer science. It directs you to build a virtual computer using the very elementary unit the NAND gate. The journey starts with elementary logic gates, out of which we build a chipset. Then we wire this chipset in an ingenious way to establish the computer architecture of the HACK platform. This computer only speaks machine language, and our next step is to write the software stack that makes it speak an object oriented high level language, as well as the Operating System that this language relies on. The project uses some of the coolest computer science algorithm and application of data structures. The project evolves loosely connected fragments to a structured knowledge tree, with substantial knowledge about the inner workings of all components.

INTRODUCTION

NAND TO TETRIS restores the big picture and demystifies the integrated design and function of computer systems. Using a modular series of 12 projects, the gradual construction of a complete working computer system is achieved. Starting with simple NAND gates, the students build a general-purpose hardware platform and a modern software hierarchy, yielding a simple but surprisingly powerful computer.

The project focuses on the building of modern, full-scale computer system - hardware and software - from the ground up. It compliments one of the most important and basic feature of the computer science foundation – Abstraction.

The project not only focuses on maximizing the concepts of computer science but also creates the thrill of creating something from almost nothing. Thus the project is also quite popular and interesting for non CS major students. Nand to Tetris gives such learners a hands-on coverage of most of the important ideas and techniques in applied computer science, focusing on computer architecture, compilation, and software engineering, in one course. Nand to Tetris also provides a hands-on overview of key data structures and algorithms, as they unfold in the context of 12 captivating hardware and software development projects.

The computer built in Nand to Tetris is real, and it works. Typically, learners implement their evolving computer on the supplied hardware simulator, which is a software tool running on the learner's PC (that's how hardware engineers actually build computers).

THEORY

The entire journey is divided into two sub-journeys, with the first one comprising of building the hardware layer of the platform and the second, more demanding one, comprising of the platform's software hierarchy. Consequently, the second journey requires good knowledge of an object oriented high level language such as Java or Python.

In this project we have built a modern computer system from the ground up. This project will take on from constructing elementary logic gates to fully functional general-purpose computer. Starting with NAND gate we will be able to build hardware like ALU, memory and assembler. After that we will be building modern software hierarchy, designed to enable the translation and execution of object-based, high level languages on a bare-bone computer hardware system.

We have implemented a virtual machine and a compiler for a JAVA like programming language and at last we will develop a basic operating system that closes the gap between the high level language and the underlying hardware platform.

METHODS

3.1 EXPERIMENTAL SETUP

The project consists of 12 mini projects. The first six projects will be around building hardware of the computer. The later six will be around building software of the computer.

The basic constitutional unit is the NAND gate. We have built building basic units like gates, general purpose registers, etc. to fully functional units like ALU, assembler, compiler, etc. only using NAND gates.

The hardware projects will be built on a personal computer using a simple Hardware Description Language (HDL) and a supplied Hardware Simulator.In computer engineering, a **hardware description language** (**HDL**) is a specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits. A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis and simulation of an electronic circuit. A hardware description language looks much like a programming language such as C or ALGOL; it is a textual description consisting of expressions, statements and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time.

The software projects (assembler, virtual machine, and compiler for a simple object-based language) can be developed in any high-level programming language like JAVA or python. A mini-OS will also be built, using the high-level language.

3.2 PROCEDURE

NAND TO TETRIS restores the big picture and demystifies the integrated design and function of computer systems. Using a modular series of 12 projects, the gradual construction of a complete working computer system is achieved.

The entire journey is divided into two sub-journeys, with the first one comprising of building the hardware layer of the platform and the second, more demanding one, comprising of the platform's software hierarchy.

Project 1:

- In this module we learnt how Boolean functions can be physically implemented using logic gates.
- Learnt to hardware description language (HDL) to simulate the chips.
- In this module we have build some gates like AND, OR, XOR. And also built MUX and DMUX using the previously built gates.
- The gates build in this module will be helpful to build other circuits in upcoming modules.

Project 2:

- In this module we have built chips to add two binary numbers.
- First we built half adder and then full adder. Next we designed the n-bit adders, 16-bit incremented, counters, two's complement method.
- With the previously built chips in this module we have also built the ALU which performs a whole set of arithmetic and logical operation.

Project 3:

- In this module, we have gradually built the main memory unit, i.e., the Random-Access Memory (RAM).
- We begin by building the elementary flip-flops, and registers (1-bit and 16 bit). Then we proceed to make memory chips of varying storage capacity and the program counter.
- The above mentioned chipset along with the previously built chips, lead up to the Random Access Memory (RAM) unit.

Project 4:

- In this module, we learnt about the assembly language and also saw how native binary code executes on the hardware platform.
- We have written two simple low-level assembly programs, mul.asm (computes multiplication of R0 and R1) and fill.asm (illustrates I/O handling).
- Next, we used a supplied CPU Emulator (a computer program) to test and execute our programs.

Project 5:

- This module deals with the computer architecture. For this system we have used Von Neumann Architecture.
- Like any other computer, the HACK computer also has a predefined fetch execute cycle and executes one instruction in one clock cycle.
- In this module we also consolidate all the parts built so far to form the central processing unit (CPU).

Project 6:

- Building the assembler to convert the assembly code to equivalent binary code.
- Challenges:
  1. Instructions: converting A and C instruction to 16-bit machine instruction.
  2. White spaces and comments: ignore them
  3. Symbols: Creating Symbol table which consists of symbol and value pairs.
- The assembly process:
  1. Initialization: Construct an empty symbol table and then add the pre-defined symbols to the symbol table.
  2. First pass: Scan for labels and add symbol value pair to the symbol table.
  3. Second pass: Scan for variables and add symbol value pair to the symbol table.
- Proposed Software Architecture
  1. Parser: unpacks each instruction into its underlying fields.
  2. Code: translates each field into its corresponding binary value.
  3. Symbol Table: manages the symbol table.
  4. Main: initializes the I/O files and drives the process.

Project 7 and Project 8:

- The virtual machine language is learnt. The VM language arithmetic commands, control command, function call and return commands are learnt.
- The VM translator is built in this module. The VM-Translator is built in java language. The VM-Translator converts the VM code into the assembly language code.
-

Project 9 and Project 10:

- The jack language is high level language. The jack program language syntax and semantics are learnt. The jack language is similar to java language with different syntax and procedures. In this module the compiler is built.

- The compiler converts the jack level language program into VM code. The compiler is developed using java.

Project 11:

In this the programs are written in jack level language. Then it is converted into VM code using the compiler. The VM code is loaded into the VM-Emulator to check the working of the program. In the same manner 3 to 5 programs are written in jack language and tested using VM-Emulator.

Project 12:

In this module the Mini-OS developed. The mini-OS consists of 8 main programs each performing some tasks. The 8 programs are:

- Array: Implements the array type and array related operations.
- String: Implements the string type and string related operations.
- Keyboard: Handles user input from the key-board.
- Screen: Handles the graphic output to the screen.
- Output: Handles the text output to the screen.
- Math: Provides basic mathematical operations.
- Memory: Handles the memory operations.
- Sys: Provides the execution-related services.

RESULTS

This project consists of 12 modules. First six modules are developing the hardware of the computer and the last six modules are developing the software hierarchy. The last module in this project is very important which is used to build the mini-OS.

Simulation of first project:

First starting to built the simple gates like AND, OR, XOR, DMUX, MUX, Half-adder and Full-adder etc. using the NAND gate. And the hardware descriptive language is used for building the chips. Here the implementation of building AND gate from the NAND gate is shown in figure 1.2. The implementation of AND gate HDL code is shown in the figure 1.2. The code of AND gate is loaded into the HDL software simulator. The output from simulating the code is shown in the figure 1.3. In the same manner the other gates are developed.

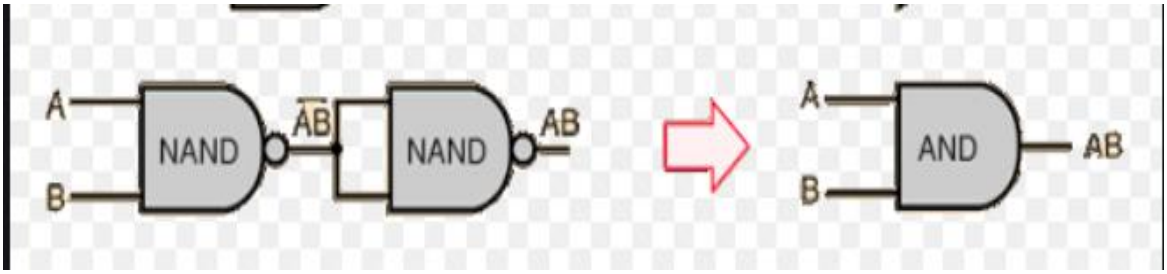


Figure 1.1- NAND to AND conversion

```
CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    |
        Nand(a=a,b=b,out=anandb);
        Nand(a=anandb,b=anandb,out=out);

}
```

Figure 1.2- AND gate HDL code.

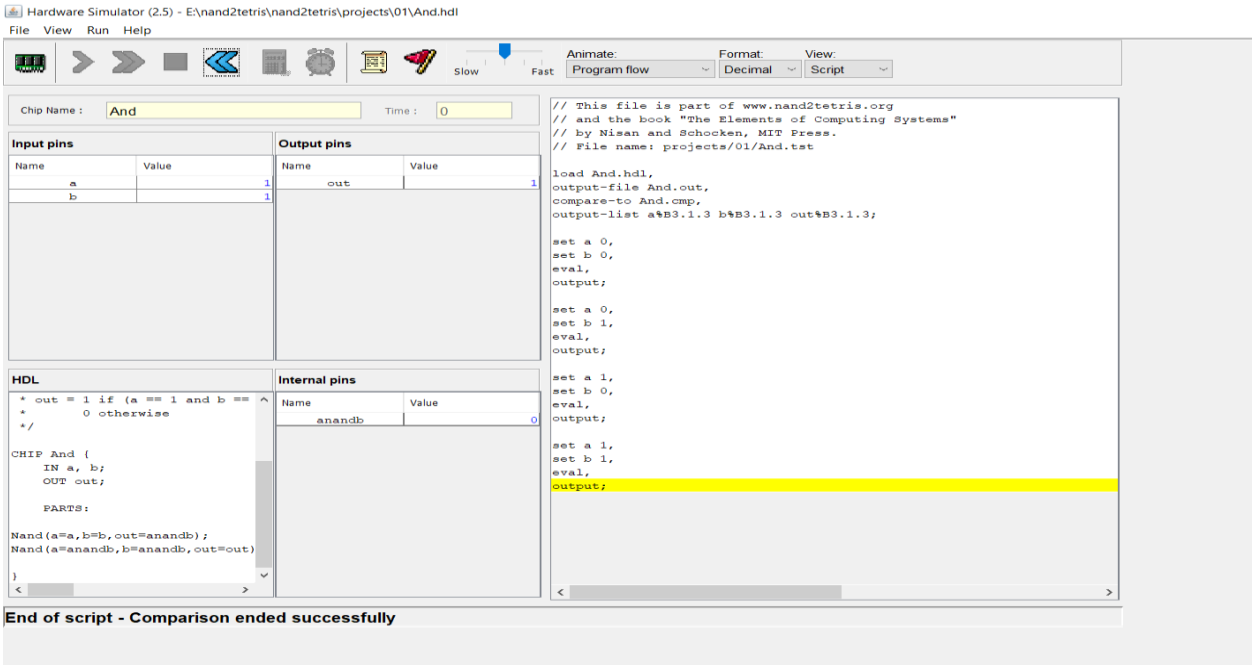


Figure 1.3- Successful HDL simulation of AND gate.

Simulation of second project:

Next module is building of ALU. The figure 1.4 shows the implementation diagram. The ALU does the simple computation works like addition, subtraction, bitwise AND, bitwise OR, 2's complement etc. And the figure 1.5 shows the implementation of the ALU in HDL simulator.

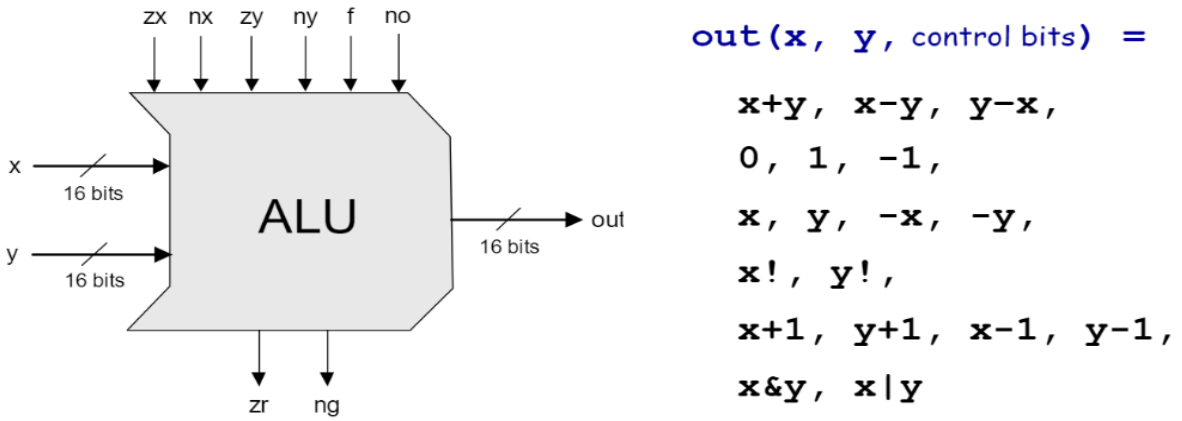


Figure 1.4- ALU diagram.

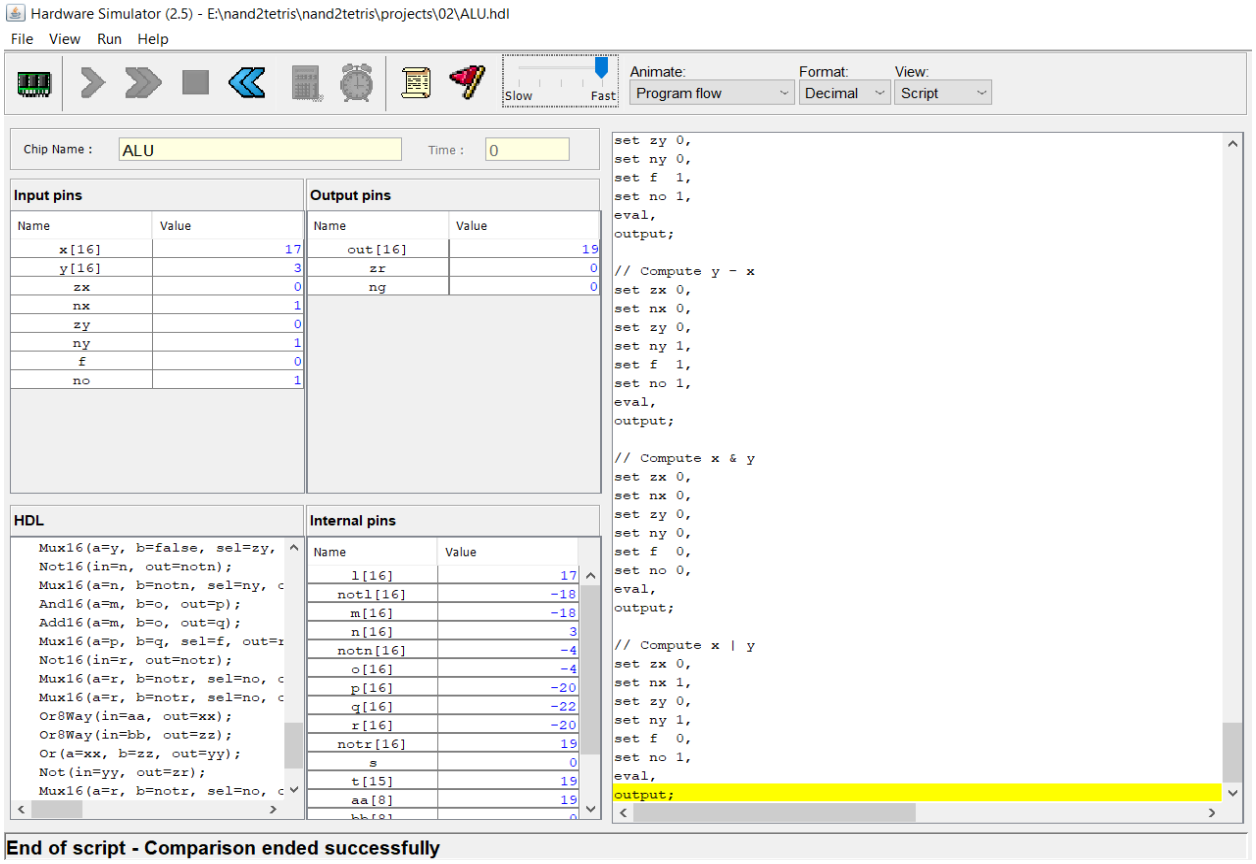


Figure 1.5- ALU simulation.

Simulation of third project:

In the next module initially the 1-bit register is built using the D-Flip Flop and MUX. Figure 1.6 shows the implementation of 1-bit register. With the 1-bit register the 16-bit registers are built. And using the 16-bit registers the RAM8, RAM64, RAM512, RAM4k, RAM16k are built. In this module the Program counter is also built.

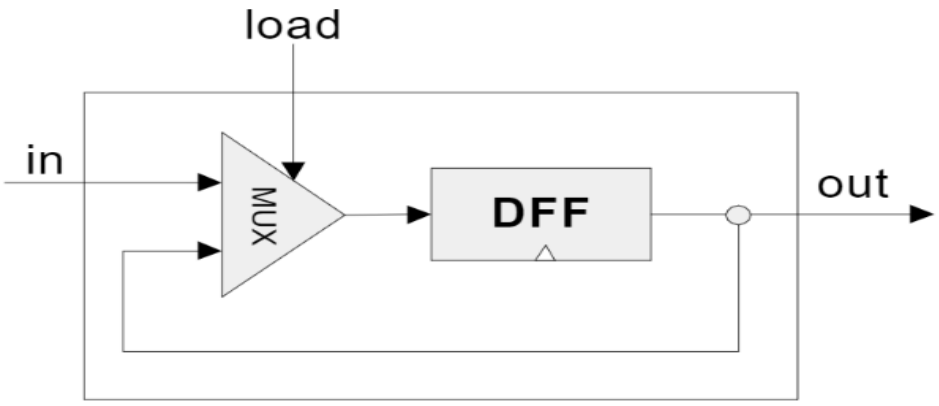


Figure 1.6- 1-bit register implementation diagram.

Simulation of fourth project:

In this module machine language instructions are introduced. The hack assembly code basics are learnt and the simple programs are executed using the CPU Emulator. The multiplication of two number program is written in assembly language and executed. The written program is loaded into the CPU Emulator and executed. The figure 1.7 shows the execution of multiplication of two numbers. The two numbers used in multiplication is 10 and 2. The output is stored in the RAM[2] position which is 20. So the multiplication program is working properly.

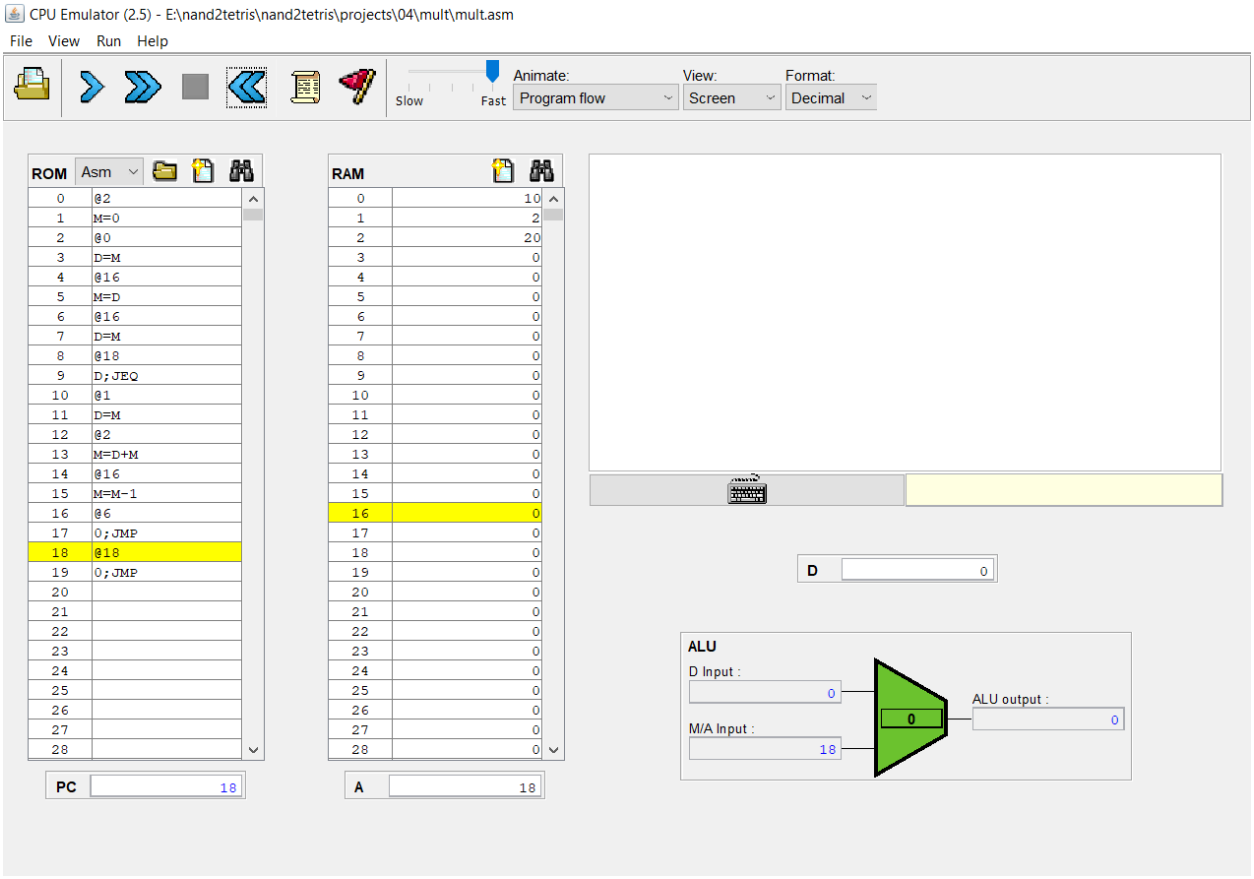


Figure 1.7- Multiplication of two numbers 10 and 2.

Simulation of fifth project:

The next module the CPU is built using the Von Neumann architecture. The figure 1.8 shows the CPU external diagram. The CPU contains the ALU, two registers and counter PC. The CPU code is done in the HDL and executed using the HDL simulator. The code is loaded into the simulator and executed. The figure 1.9 shows the successful execution of CPU.

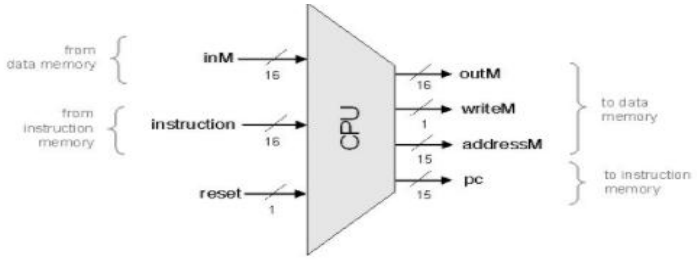


Figure 1.8- CPU external diagram.

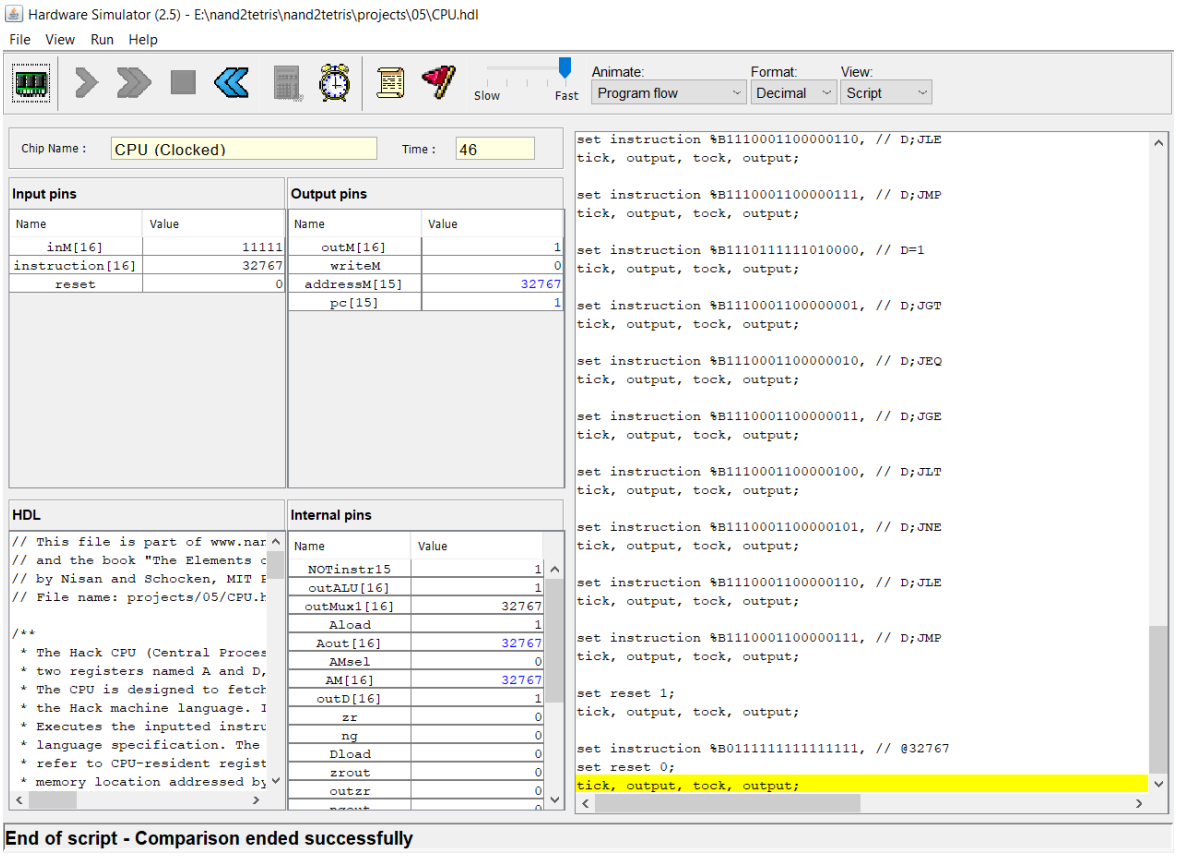


Figure 1.9- CPU execution in HDL Simulator.

Next the hardware computer is built which consists of CPU, ROM and RAM. The figure 1.10 shows the chip implementation of the hardware Computer. The code is done in HDL. The figure 1.11 shows the code of hardware Computer. The code is executed in HDL simulator. The figure 1.12 shows the successful execution of the Hardware computer.

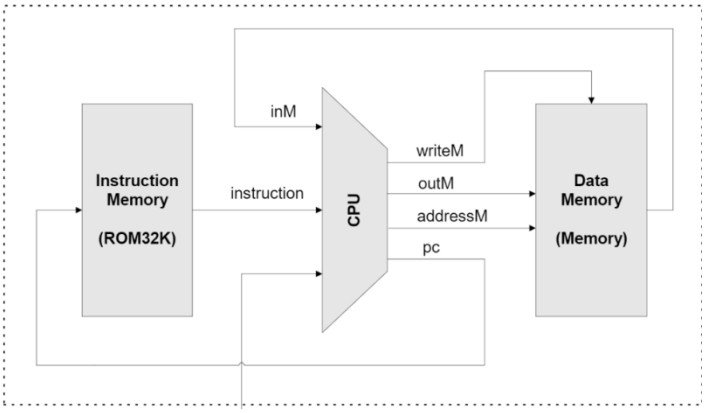


Figure 1.10- The Schematic diagram of Hardware Computer.

```
CHIP Computer {

    IN reset;

    PARTS:
    // Put your code here:
        ROM32K(address=pc,out=instruction);
    CPU(inM=memOut,instruction=instruction,reset=reset,outM=outM,writeM=writeM,addressM=addressM,pc=pc);
    Memory(in=outM,load=writeM,address=addressM,out=memOut);


}
```

Figure 1.11- Code for Hardware Computer.

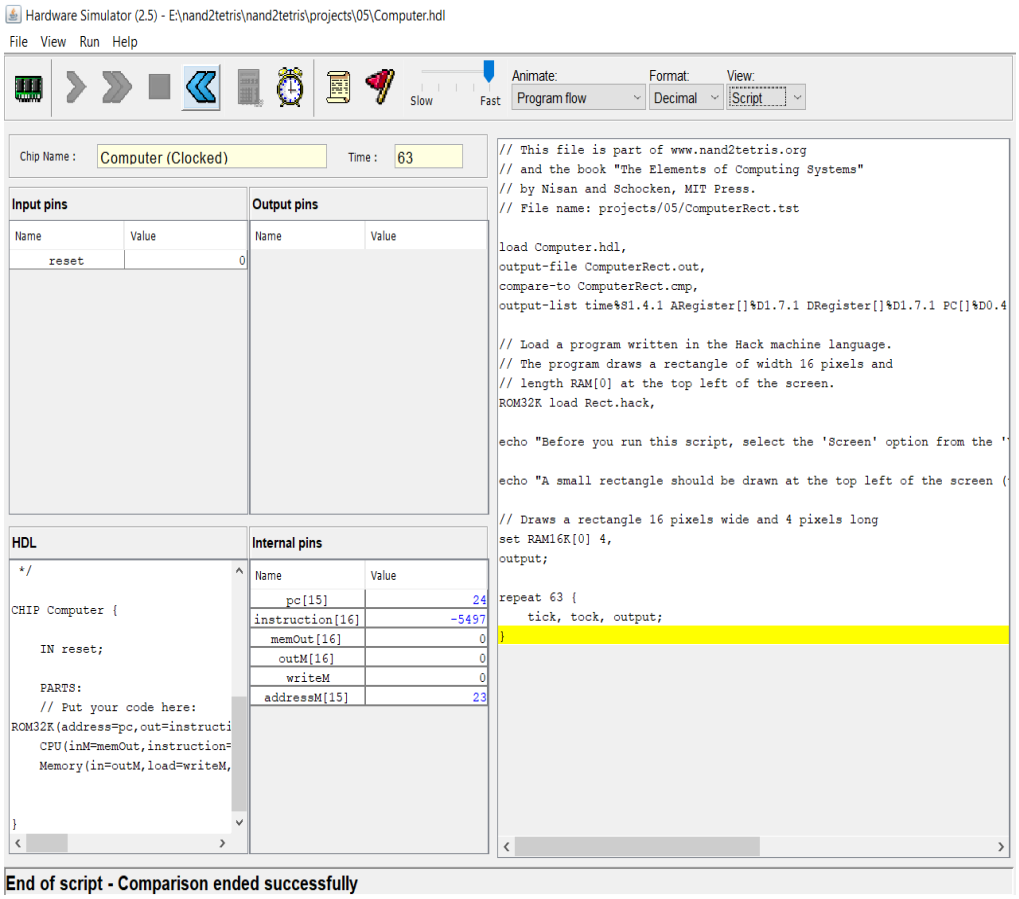


Figure 1.12- Execution of hardware Computer in HDL simulator.

Simulation of sixth project:

The next module is building of the Assembler which converts the assembly language into the binary form. The Assembler is built using the Java. The normal assembly language program is loaded into the assembler created and the binary format file is generated. And to check that the binary code generated is correct we use the Assembler simulator to compare. In the figure 1.13 the Assembler generated binary file is loaded into the comparison column to check whether the binary created by the Assembler Emulator and the Assembly created by us match. The figure 1.13 shows the successful comparison.

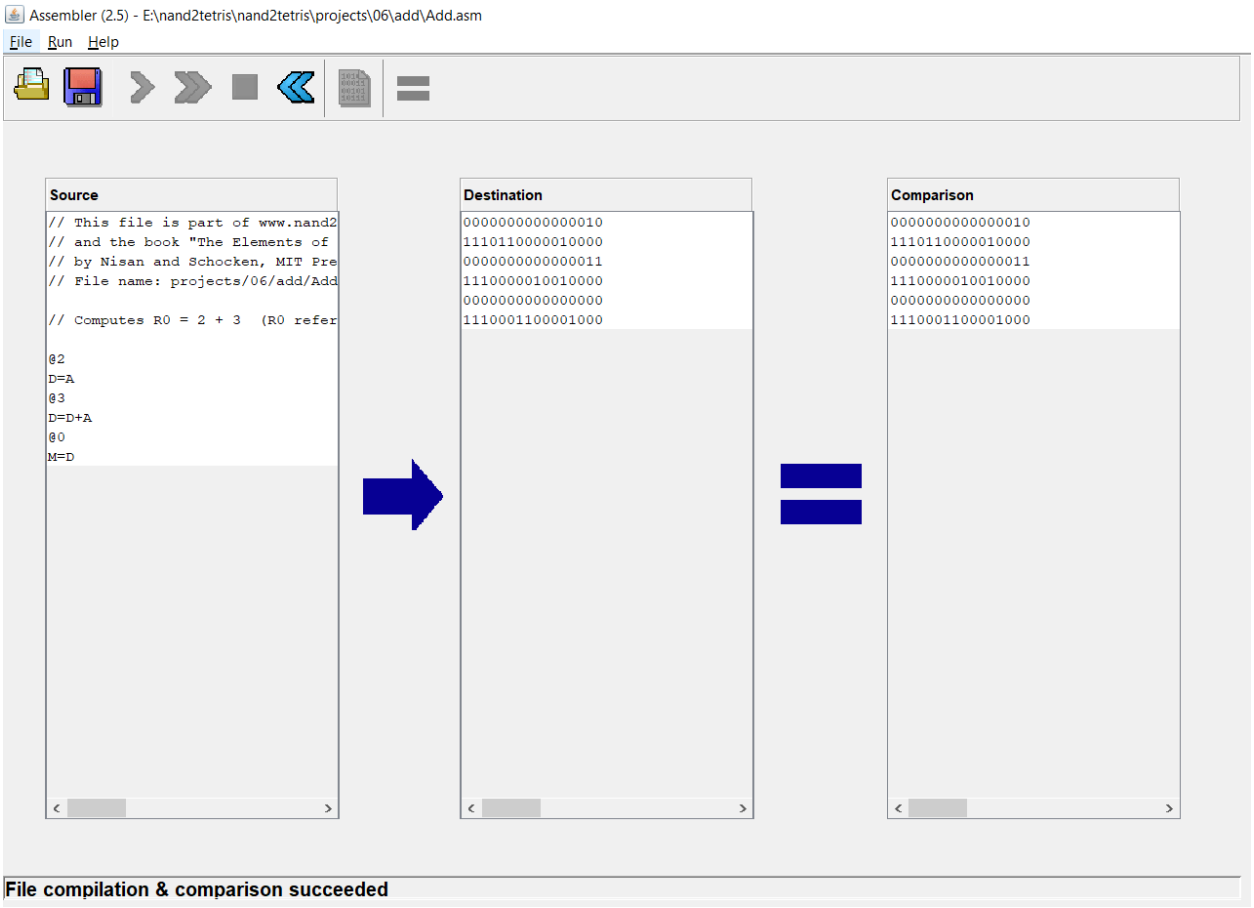


Figure 1.13- Add.asm file converted to binary.

Simulation of 7th and 8th project:

In this module the VM language is learnt and the VM-Translator is built. The VM-Translator converts the VM code to the Assembly level code. The VM-Translator is coded using the Java. For example, the Fibonacci series code is written in VM language and converted to assembly level code using the VM-Translator. To know that the VM-Translator is working properly, the generated Assembly level code is loaded into the CPU emulator. If the code gets executed without any error, then we can verify that the VM-Translator build is working properly. The figure 1.14 shows the VM code of Fibonacci series. The code is loaded into the VM-Translator created by us and the proper assembly level code is generated. This assembly level code is loaded into the CPU emulator. In figure 1.15, the Fibonacci series is stored from RAM[3000] to RAM[3005]. The figure 1.16 shows the Fibonacci series.

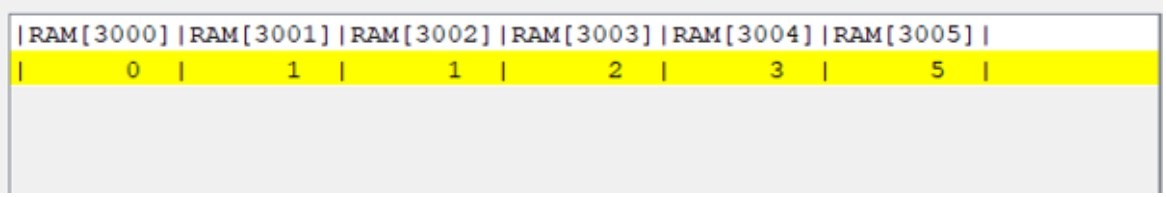


Figure 1.16- Fibonacci series.

```
push argument 1
pop pointer 1           // that = argument[1]

push constant 0
pop that 0              // first element in the series = 0
push constant 1
pop that 1              // second element in the series = 1

push argument 0
push constant 2
sub
pop argument 0          // num_of_elements -= 2 (first 2 elements are set)

label MAIN_LOOP_START

push argument 0
if-goto COMPUTE_ELEMENT // if num_of_elements > 0, goto COMPUTE_ELEMENT
goto END_PROGRAM        // otherwise, goto END_PROGRAM

label COMPUTE_ELEMENT

push that 0
push that 1
add
pop that 2              // that[2] = that[0] + that[1]

push pointer 1
push constant 1
add
pop pointer 1           // that += 1

push argument 0
push constant 1
sub
pop argument 0          // num_of_elements--

goto MAIN_LOOP_START

label END_PROGRAM
```

Figure 1.14- Fibonacci Series code in VM language.

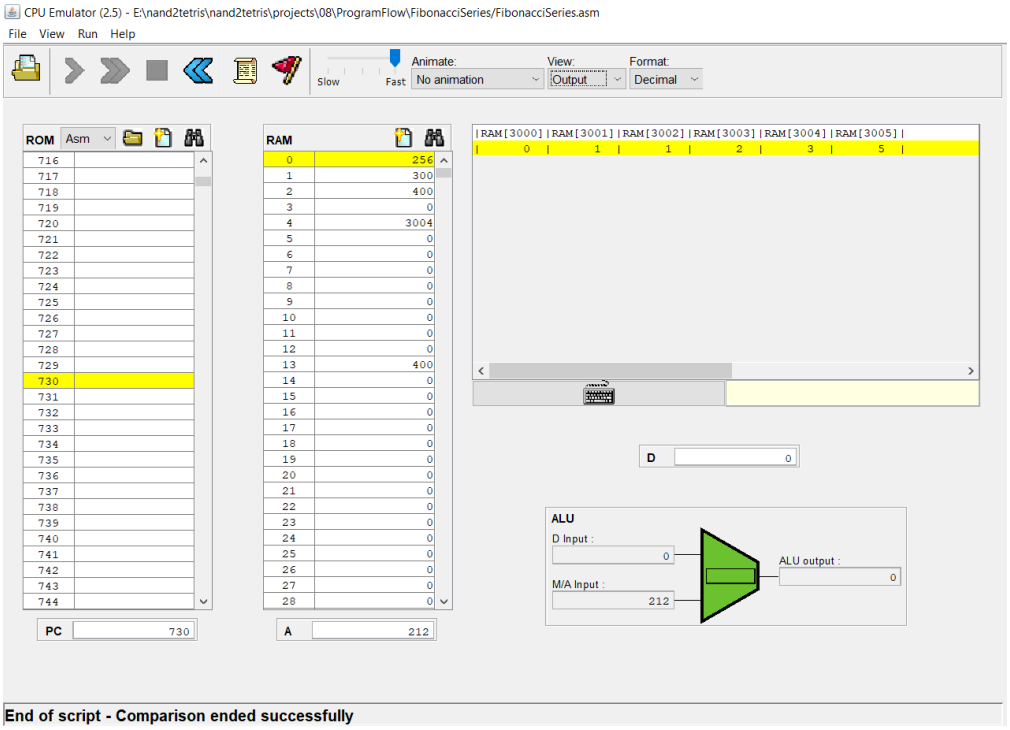


Figure 1.15- The execution of Fibonacci series in CPU emulator.

Simulation of 9th and 10th project:

In the next module the Jack language is learnt. Jack is a high-level programming language. In this module the compiler is built to convert the high-level Jack program to VM language. The jack program is loaded into the compiler and the compiler generates the VM code. The VM code is loaded into the VM Emulator to check whether it is working properly. If the VM code is working properly then the Compiler built is working properly or else there is a mistake in building the compiler. For this we can take an example program conversion of number to binary written in jack language. The program is shown below.

```
class Main {

   function void main() {
            var int value;
      do Main.fillMemory(8001, 16, -1); // sets RAM[8001]..RAM[8016] to -1
      let value = Memory.peek(8000);    // reads a value from RAM[8000]
      do Main.convert(value);           // performs the conversion
      return;
   }

   /** Converts the given decimal value to binary, and puts
    *  the resulting bits in RAM[8001]..RAM[8016]. */
   function void convert(int value) {
         var int mask, position;
         var boolean loop;

         let loop = true;
         while (loop) {
            let position = position + 1;
            let mask = Main.nextMask(mask);

            if (~(position > 16)) {

               if (~((value & mask) = 0)) {
                  do Memory.poke(8000 + position, 1);
               }
               else {
                  do Memory.poke(8000 + position, 0);
               }
            }
            else {
```

```
            let loop = false;
          }
        }
        return;
    }

    /** Returns the next mask (the mask that should follow the given mask). */
    function int nextMask(int mask) {
        if (mask = 0) {
          return 1;
        }
        else {
          return mask * 2;
        }
    }

    /** Fills 'length' consecutive memory locations with 'value',
     * starting at 'startAddress'. */
    function void fillMemory(int startAddress, int length, int value) {
      while (length > 0) {
        do Memory.poke(startAddress, value);
        let length = length - 1;
        let startAddress = startAddress + 1;
      }
      return;
    }
}
```

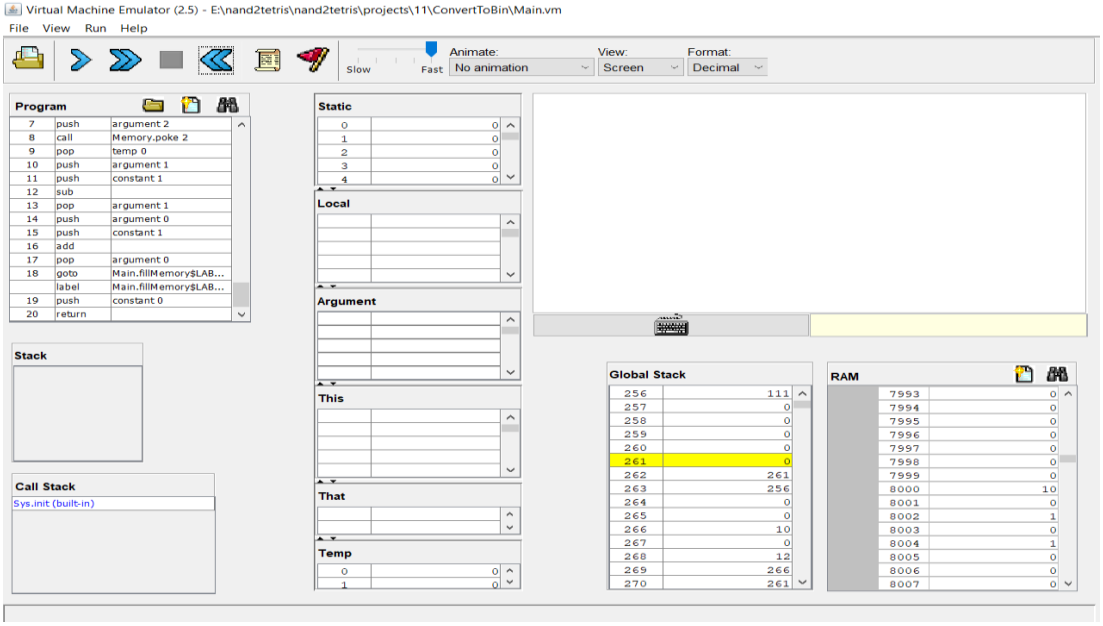
VM language program:

```
function Main.main 1
push constant 8001
push constant 16
push constant 1
neg
call Main.fillMemory 3
pop temp 0
push constant 8000
call Memory.peek 1
pop local 0
push local 0
call Main.convert 1
pop temp 0
push constant 0
return
function Main.convert 3
```

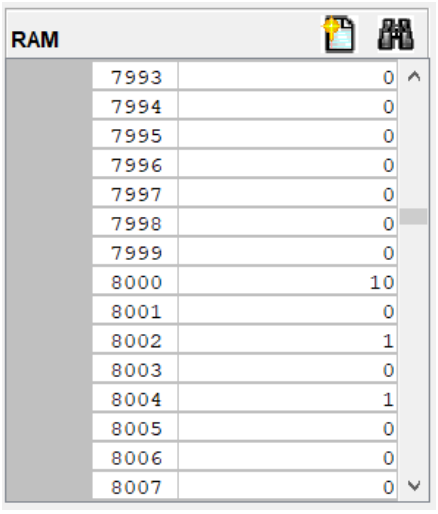
```
push constant 0
not
pop local 2
label LABEL_1
push local 2
not
if-goto LABEL_0
push local 1
push constant 1
add
pop local 1
push local 0
call Main.nextMask 1
pop local 0
push local 1
push constant 16
gt
not
not
if-goto LABEL_2
push argument 0
push local 0
and
push constant 0
eq
not
not
if-goto LABEL_4
push constant 8000
push local 1
add
push constant 1
call Memory.poke 2
pop temp 0
goto LABEL_5
label LABEL_4
push constant 8000
push local 1
add
push constant 0
call Memory.poke 2
pop temp 0
label LABEL_5
goto LABEL_3
label LABEL_2
push constant 0
pop local 2
```

```
label LABEL_3
goto LABEL_1
label LABEL_0
push constant 0
return
function Main.nextMask 0
push argument 0
push constant 0
eq
not
if-goto LABEL_6
push constant 1
return
goto LABEL_7
label LABEL_6
push argument 0
push constant 2
call Math.multiply 2
return
label LABEL_7
function Main.fillMemory 0
label LABEL_9
push argument 1
push constant 0
gt
not
if-goto LABEL_8
push argument 0
push argument 2
call Memory.poke 2
pop temp 0
push argument 1
push constant 1
sub
pop argument 1
push argument 0
push constant 1
add
pop argument 0
goto LABEL_9
label LABEL_8
push constant 0
return
```

OUTPUT from the VM-EMULATOR:



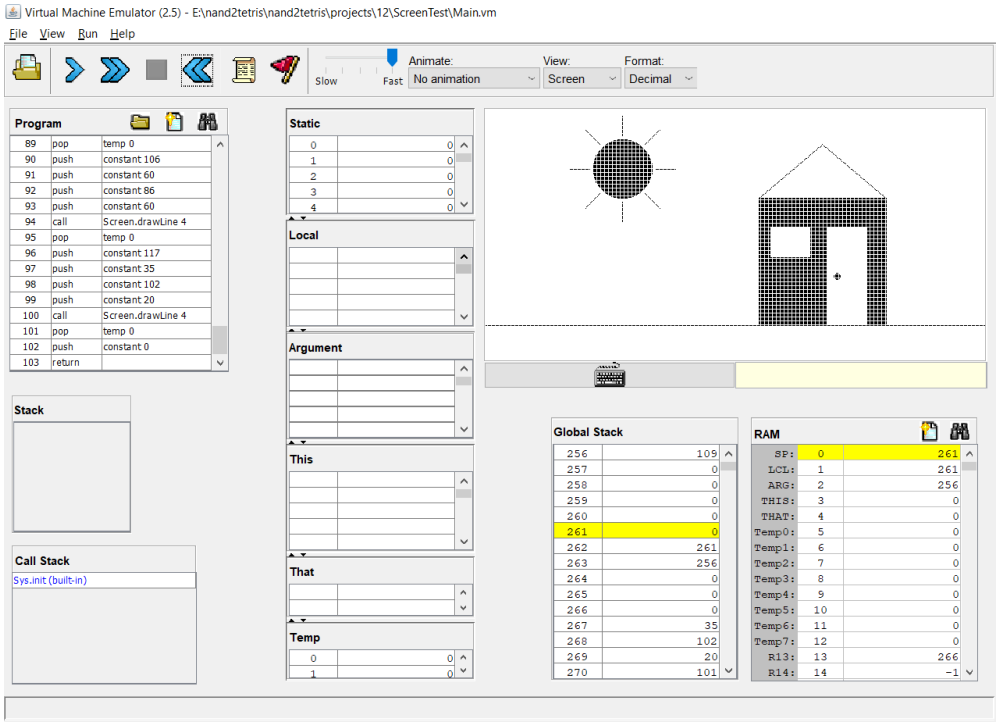
The numeric value entered is 10. The binary value is found from RAM[8001] to RAM[8016].



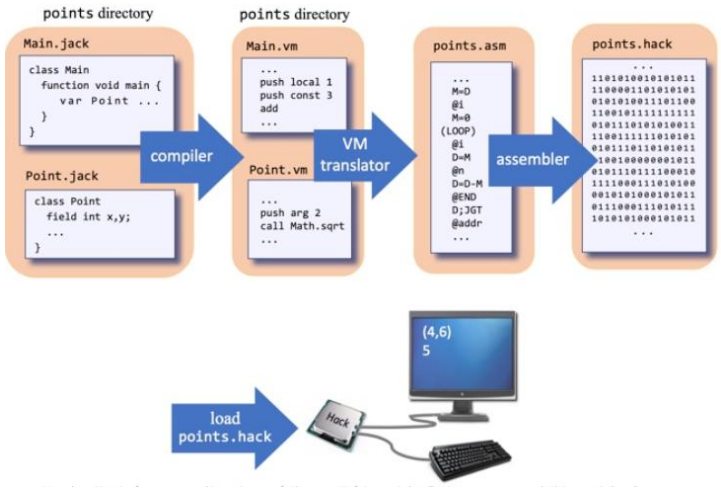
The binary value of 10 is 01010 is shown in the figure above from RAM[8001] to RAM[8005].

Simulation of 12th project:

The next module is building of the mini-OS. The consist of 8 functionalities which are Array allocation and de-allocation, Math functions, Keyboard, Memory allocation, Output handling, Screen functionalities, String functions and sys function. These programs are coded in jack language. And converted into VM language using the compiler and the VM code is executed in the VM-Emulator. For example, the Screen handles the graphic output of the screen. The Screen.vm code is loaded into the VM-Emulator and executed. The figure below shows the output from execution. From the figure the house like picture is printed in the screen after the execution. From this we can say that the Screen functionalities are working properly. All the other functionalities are tested in the same manner.



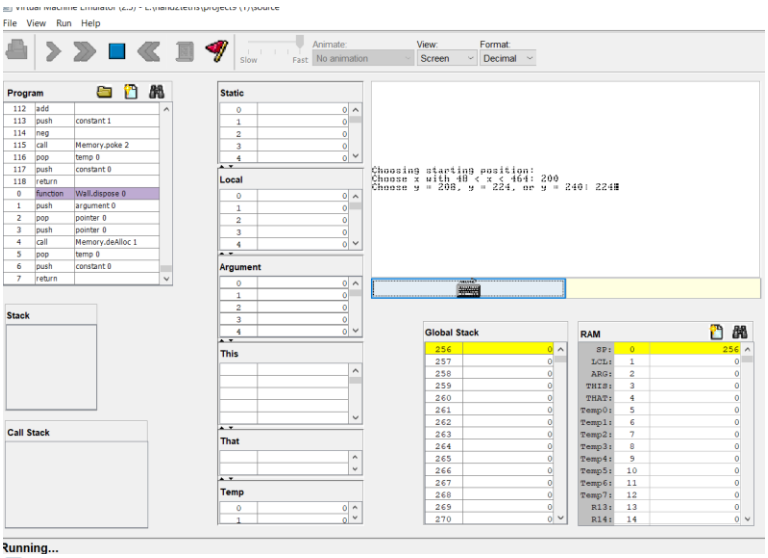
The figure shown below shows the implementation of the program point in jack language and how the computer works.

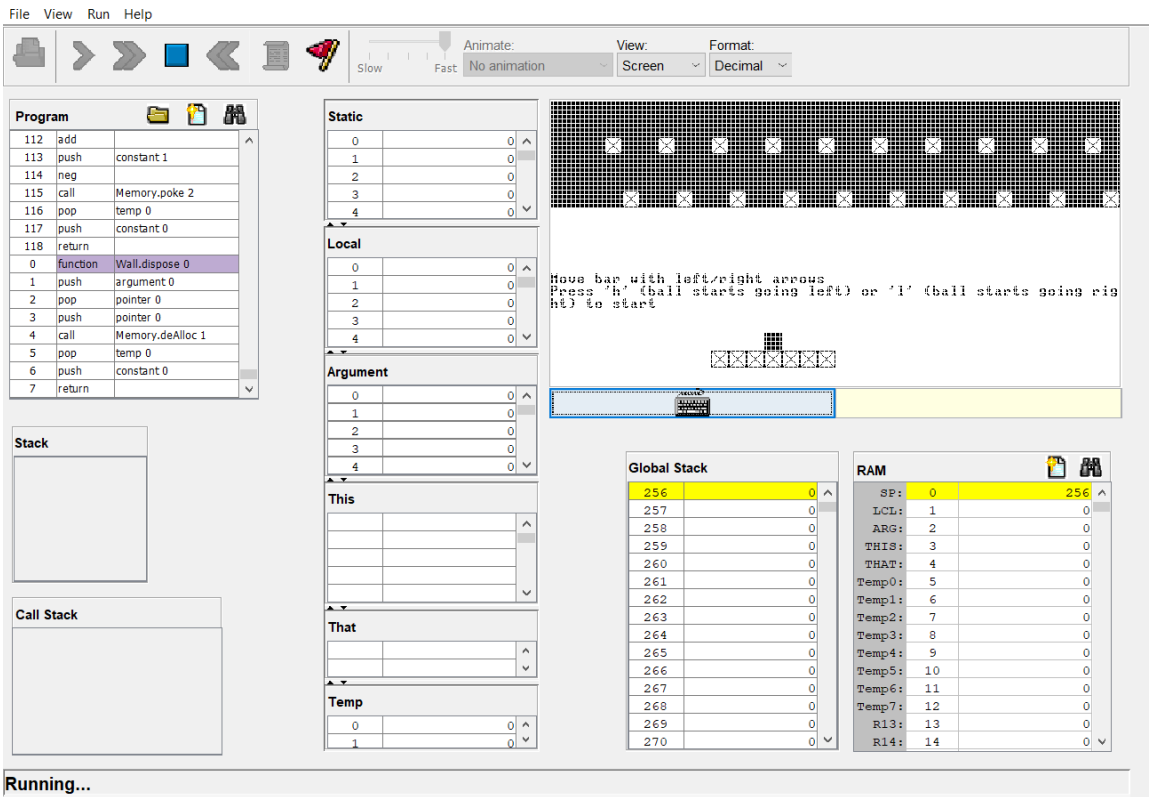


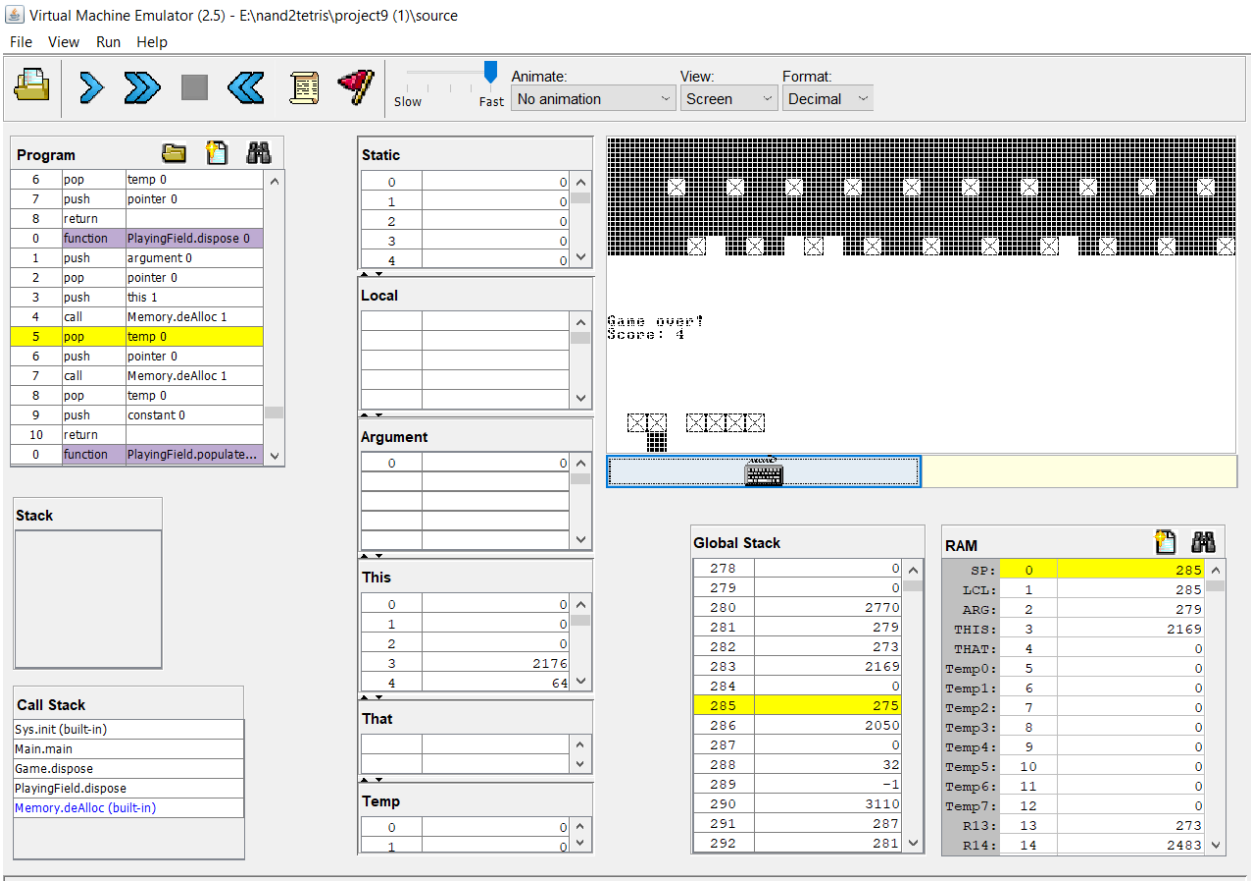
GAME IMPLEMENTATIONS:

GAME 1: BLOCK BREAKER

This game breaks the blocks using the ball. The blocks contain some walls. The pad i.e., the moving bar moves left and right directions so that the player doesn't miss the ball. Score is calculated on the basis of no. of blocks popped.







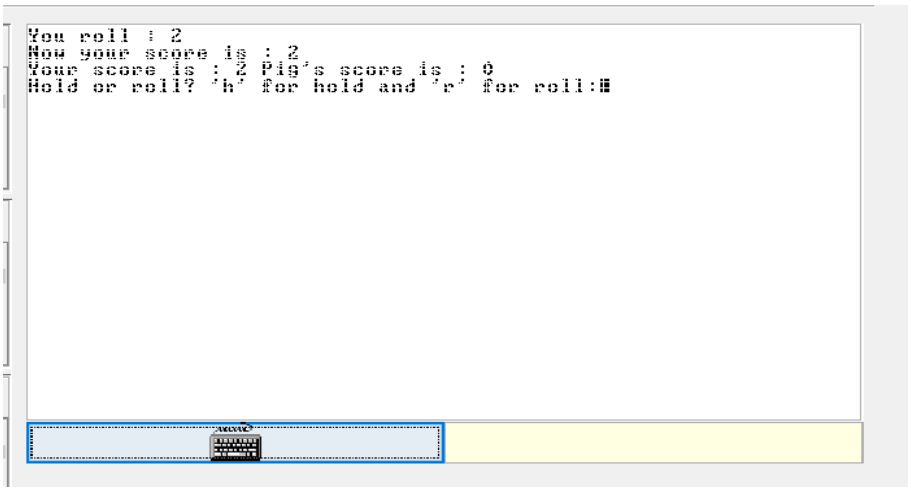
- Player misses the ball and the game ends.
- The score is displayed.

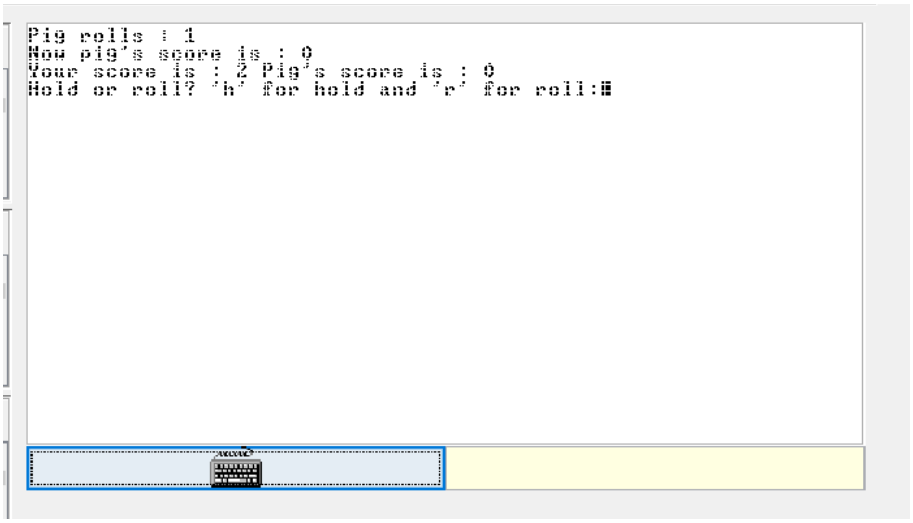
GAME 2: PIG DICE

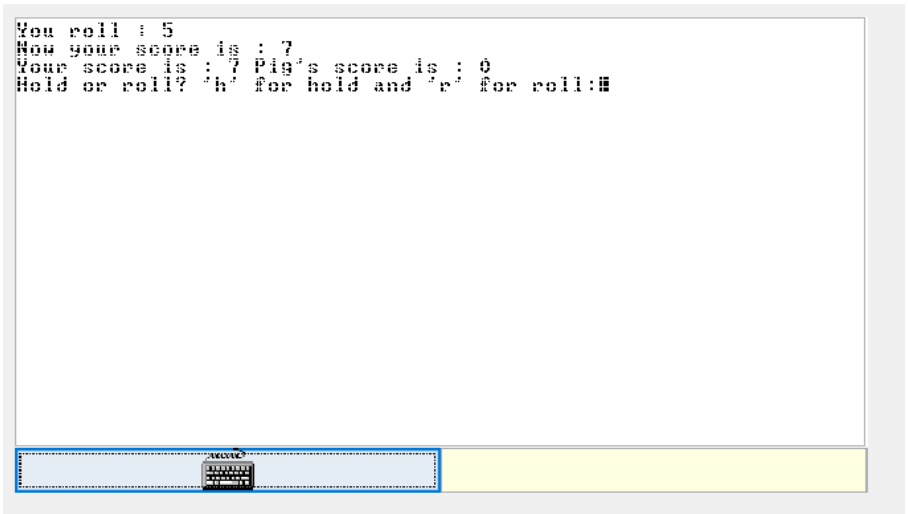
The user will play against the computer.
On a turn, a player rolls the die repeatedly until either:

- A 1 is rolled
- The player chooses to hold (stop rolling)

If a 1 is rolled, that player's turn ends and no points are earned.
If the player chooses to hold, all of the points rolled during that turn are added to his or her score.







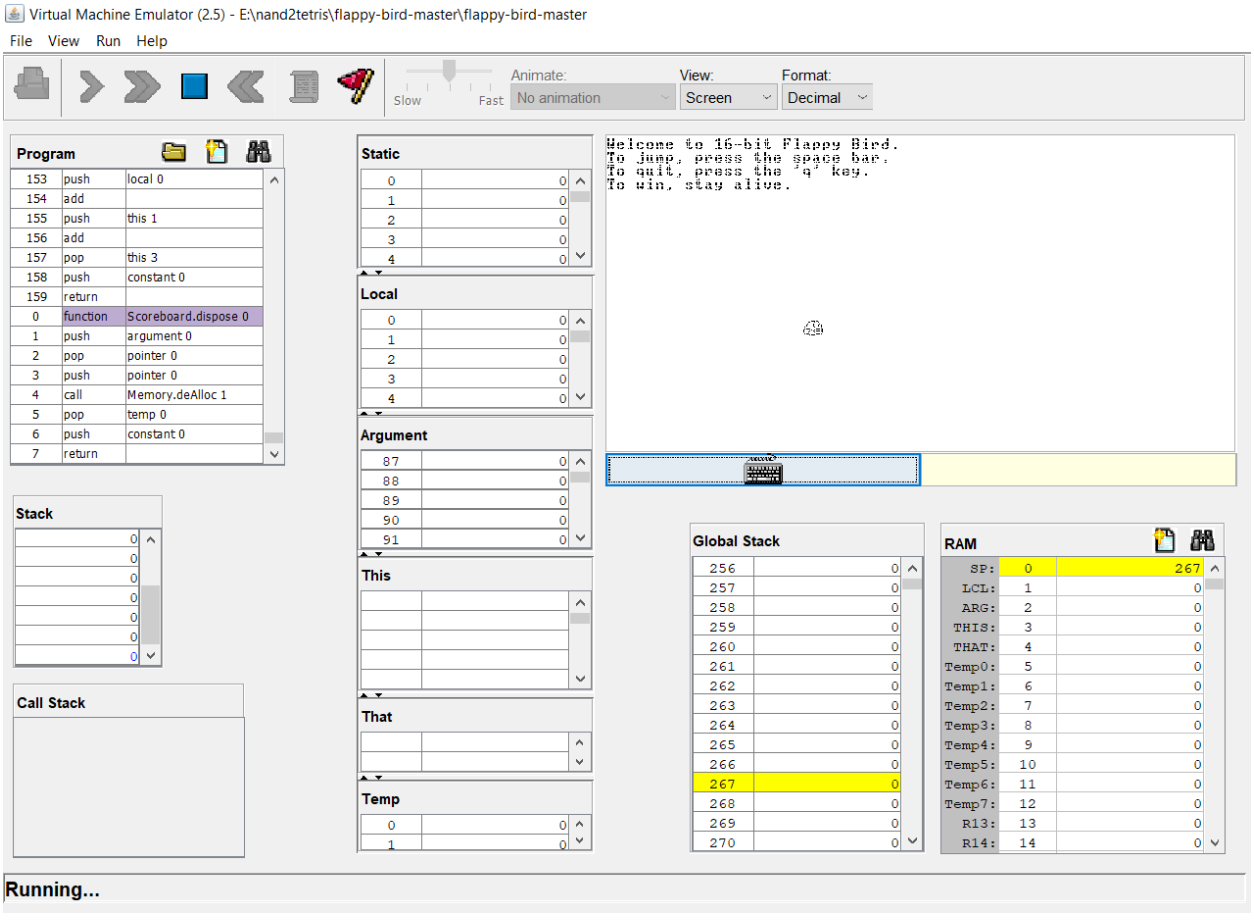
- When a player reaches a total of 20 or more points, the game ends and that player is the winner.

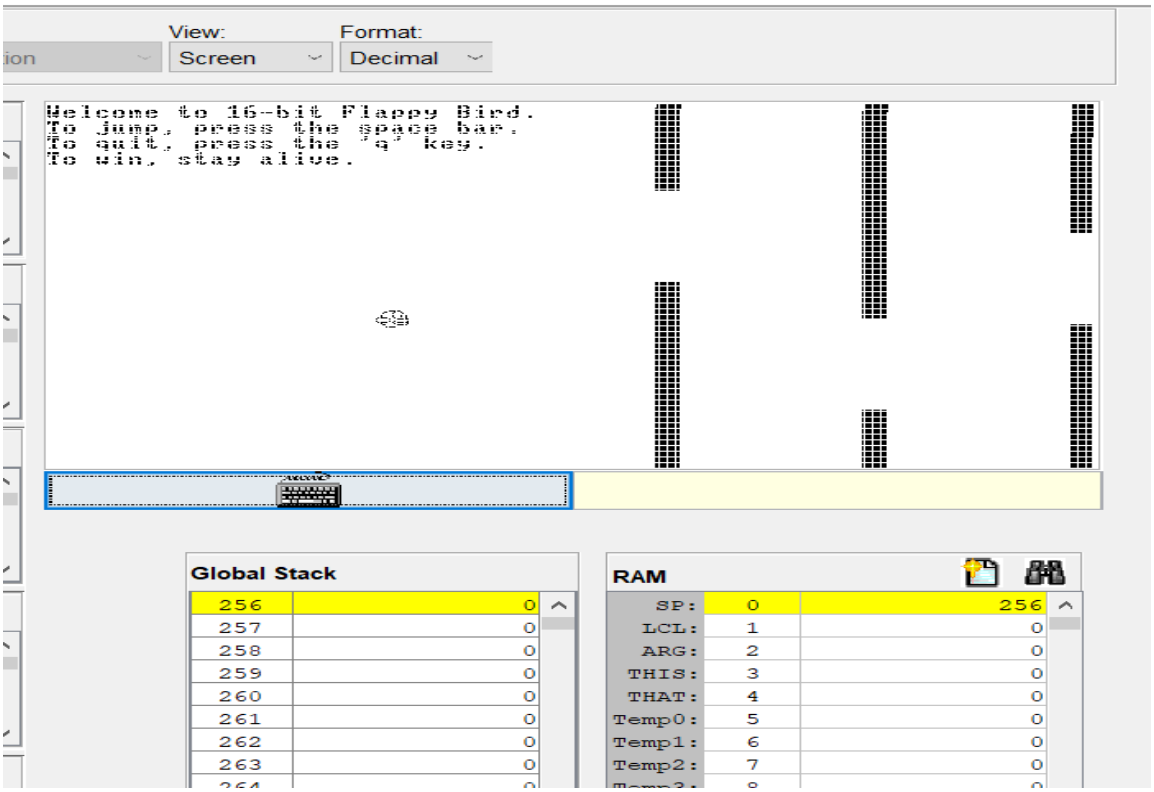
GAME 3: FLAPPY BIRD

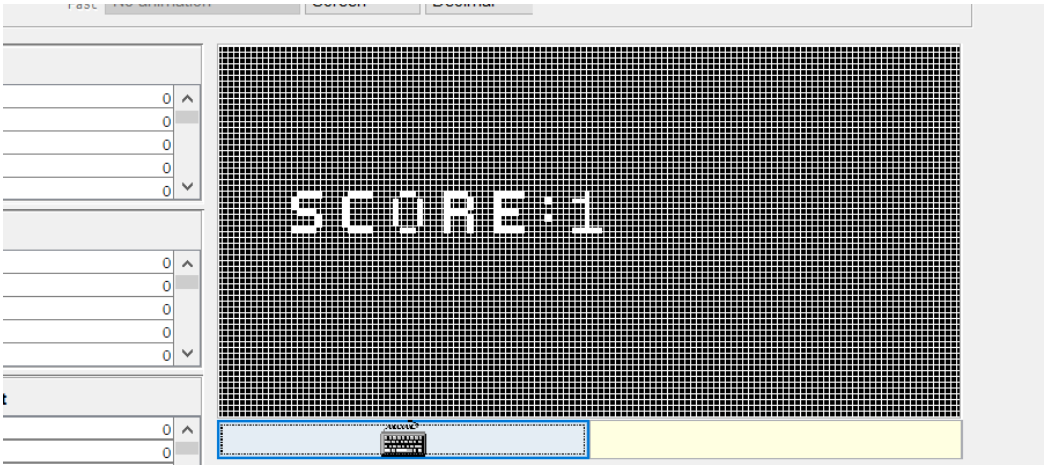
Flappy bird game is single player game where the player controls the bird who is persistently moving to the right. The player earns one point on passing through a pair of pipes.

In order to not lose the game the player needs to:

a. Protect the bird from colliding into any of the pipes.
b. Not soaring too high or diving too low.

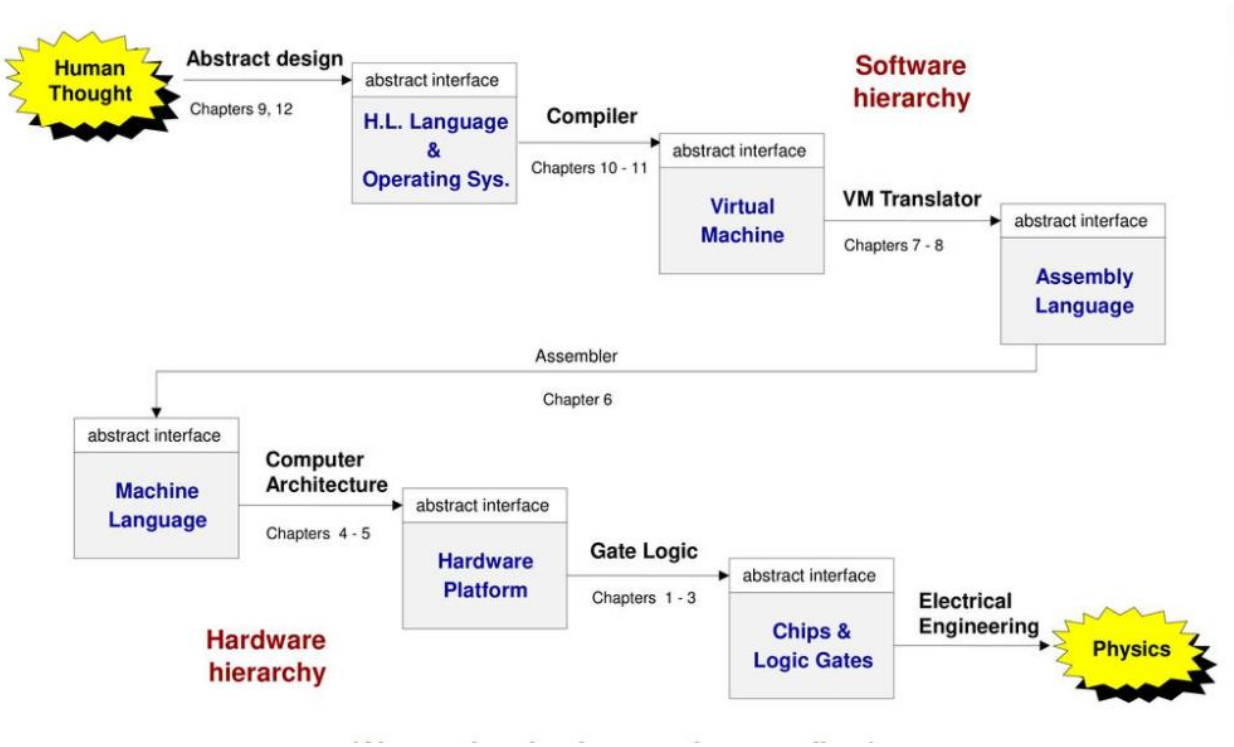






- Score is displayed in the screen.

CONCLUSION



The figure shown above shows the total block diagram of the project. This project is done in the bottom-up approach from building the elementary gates using NAND gates to building the Mini-OS. This project involves some of the coolest algorithms, data structures and techniques of computer science that will help us build a modern computer from the very first principles. Using the modern computer, any GAME can be built in the Jack high level program and executed.

ACKNOWLEDGEMENT

I would like to express my special thanks of gratitude to my teacher Dr. Phrangboklang Lyngton Thangkhiew , who gave me the golden opportunity to do this wonderful project of Operating Systems on "NAND TO TETRIS", who also helped me in completing my project. I came to know about so many new things I am really thankful to them. **Secondly,** I would also like to thank my project partners who helped me a lot in finalizing this project within the limited time frame.

REFERENCES

- **https://www.nand2tetris.org/**