

ELEMENTS OF COMPUTING-II

A THESIS

Submitted by

Ganesh Sundhar S
(CB.EN.U4AIE22017)

Karthigai Selvam
(CB.EN.U4AIE22025)

Shruthikaa C
(CB.EN.U4AIE22047)

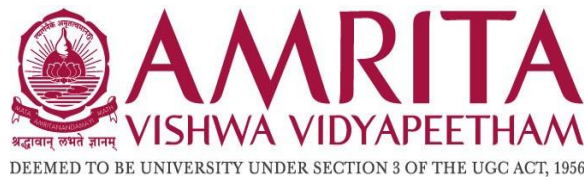
Bhavya Sainath
(CB.EN.U4AIE22055)

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

CSE(AI)



Centre for Computational Engineering and Networking

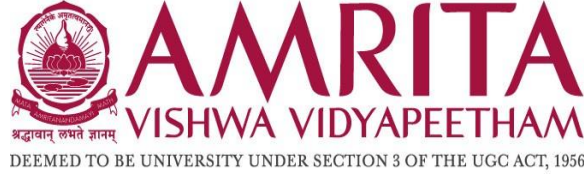
AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112 (INDIA)

JULY - 2023

**AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112**



BONAFIDE CERTIFICATE

This is to certify that the thesis entitled “Final Project” submitted by Ganesh Sundhar S (CB.EN.U4AIE22017), Karthigai Selvam (CB.EN.U4AIE22025), Shruthikaa V (CB.EN.U4AIE22047) and Bhavya Sainath (CB.EN.U4AIE22055), for the award of the Degree of Bachelor of Technology in the “CSE(AI) ” is a bonafide record of the work carried out by her under our guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

Ms. Sreelakshmi K
Project Guide

Dr. K.P.Soman
Professor and Head CEN

Submitted for the university examination held on 03-07-2023

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112

DECLARATION

We, Ganesh Sundhar S (CB.EN.U4AIE22017), Karthigai Selvam (CB.EN.U4AIE22025), Shruthikaa V (CB.EN.U4AIE22047) and Bhavya Sainath (CB.EN.U4AIE22055), hereby declare that this thesis entitled “Final Project”, is the record of the original work done by me under the guidance of Ms. Sreelakshmi K, Assistant Professor, Centre for Computational Engineering and Networking, Amrita School of Artificial Intelligence, Coimbatore. To the best of my knowledge this work has not formed the basis for the award of any degree/diploma/ associate ship/fellowship/or a similar award to any candidate in any University.

Place: Coimbatore

Date: 03-07-2023

Signature of the Student

CONTENTS

1 Introduction	4
1.1 Abstract	4
1.2 Problem Statement	5
1.3 Objectives	6
1.4 Introduction	7
2 Part A – Implement hack computer	8
2.1 Introduction	8
2.2 Methodology	9
2.3 CPU	12
2.4 Memory	18
2.5 Memory Allocation	20
2.6 Building of Computer	23
2.7 Flow Chart	26
2.8 Memory Flow Chart	27
2.9 CPU HDL Code	28
2.10 HDL Code for building memory	20
2.11 Testing of hack computer	32
3 Part B – Tic Tac Toe	33

3.1 Code in Java(Single Player)	34
3.1.1 Initialisation Steps	34
3.1.2 Main function	34
3.1.3 Move Function	38
3.1.4 Possibility function	41
3.1.5 Display Function	43
3.1.6 Over Function	43
3.1.7 Win Function	45
3.2 Code in Java(Double Player)	46
3.3 Code in Jack	48
3.3.1 Main File	48
3.3.2 Game File	48
3.3.3 Setup File	50
3.3.4 Single File	64
3.3.5 Double File	77

3.4 Source code and Demonstration	81
4 Flappy Bird Game	82
4.1 Working of Game.	82
4.1.1 Basic Understanding.....	82
4.1.2 Functionalities of the Game.....	82
4.2 Implementation	84
4.2.1 Classes in Jack	84
4.3 Code Implementation	107
4.3.1 Code in Jack.....	107
4.4 Output	152
5 Rock, Paper and Scissors	153
5.1 Implementation of Classes	154
5.2 Code in Jack	155
5.3 Output	168
6 Conclusion	173
7 References	174
8 List of figures	175

1 INTRODUCTION :

1.1 ABSTRACT :

Part A : This project report summarizes our effort to build an entirely functional computer system as a component of the Nand2Tetris course. We began with the fundamentals of digital logic and progressively working our way up to the development of an entire computer system.

Part B : This part of the report summarizes on how to build a Tic Tac Toe game in Jack language based on our understanding of the various aspects of the jack language. Jack is often thought of as a precursor to Java. This particular build used 5 different class .i.e. 2 for program management, 1 for setup activities and 2 for the different playable modes available. This was implemented for both single player and double player modes.

Part C :

Section 1 : The Flappy Bird Game has been implemented in the nand2tetris software (VM Emulator). Using 7 different classes, Main, Game, Bird, Pipe, Mod, LCGRandom and Score the code is written in Jack Language. By checking the pixel collision between the birds and the pipes the game functions.

Section 2 :

1.2 PROBLEM STATEMENT:

1. Implement the HACK computer in HDL and test it.
 - Refer the link [Project 05 | nand2tetris](#) to implement computer.hdl.
 - The computer.hdl has to be tested as per the instructions given in the above link by using any .hack file.
2. Implement the Tic-Tac-Toe game in JACK
3. Implementing The Flappy Bird Game in Jack

1.3 OBJECTIVES

- The Tic Tac Toe game should allow the player to play in two modes, single and double player
- The Flappy Bird Game should function smoothly as the bird surpasses the pipes without collision when the user presses the key instructed.
- To run the hdl code of computer in nand2tetris software (Hardware Simulator)
- To write tst file for the Computer

1.4 INTRODUCTION :

This report paper is divided into three sections. The first part discusses the Computer, which is followed by the Tic-Tac-Toe Game and then Flappy Bird Game. According to the Von Neumann Architecture the computer system is built and implemented. The Von Neumann architecture consists of a single, shared memory for programs and data, a single bus for memory access, an arithmetic unit, and a program control unit.

The CPU is the brain of a computer, containing all the circuitry needed to process input, store data, and output results. The CPU is constantly following instructions of computer programs that tell it which data to process and how to process it. A CPU (Central Processing Unit) is the primary component of a computer system that performs most of the processing tasks. It fetches instructions from memory, decodes them, performs the required calculations and operations, and stores the results back to memory.

This report provides a detailed explanation on the construction of The Computer and it's working. The vital components of the Computer including the CPU, Arithmetic Logic Unit, Memory Units are discussed along with their hdl codes to be implemented in the Hack Platform.

The game engine needs to capture user input, such as pressing a button, to make the bird flap its wings. It should respond to these input events and update the game state accordingly. It must keep track of the current state of the game, including the position of the bird, the position and movement of the pipes, the score, and whether the game is over or still in progress.

It will update and maintain this game state as the game progresses This also involves drawing the bird, the pipes, and any other visual elements. In the Nand2Tetris framework, graphics rendering is achieved by manipulating the pixels on the computer's screen memory.

2 PART A - IMPLEMENT HACK COMPUTER :

2.1 INTRODUCTION :

This project report summarises our ambitious effort to build an entirely functional computer system as a component of the Nand2Tetris course. We undertook a wonderful journey throughout the course of this project, beginning with the fundamentals of digital logic and progressively working our way up to the development of an entire computer system.

2.2 METHODOLOGY:

In the context of the Nand2Tetris course, the term "Compute" refers to the culmination of the course project. It represents the central processing unit (CPU) of a simplified computer system that is designed and implemented from scratch using basic logic gates.

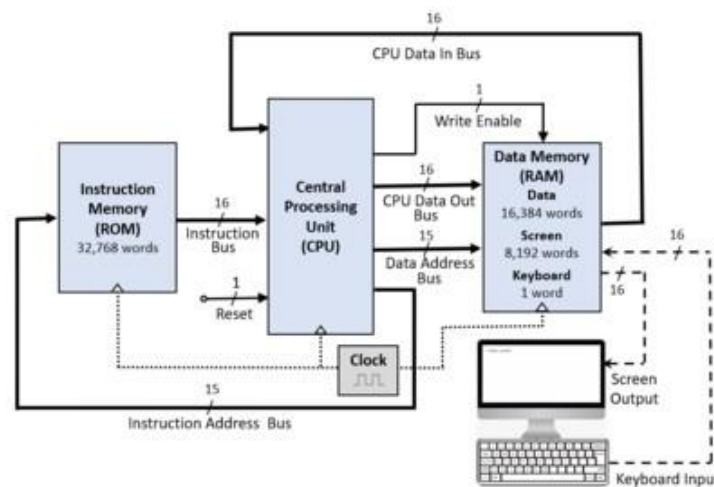


Figure 1 : Hack Computer

In the NAND 2 Tetris course, the computer chip contains multiple interconnected components, including:

- i) Arithmetic Logic Unit (ALU): The ALU conducts arithmetic (addition and subtraction) and logical (AND, OR, and NOT) operations on binary data.
- ii) Control Unit: The control unit coordinates and oversees the CPU's instruction execution. It decodes instructions and creates control signals to regulate data flow and chip activities.
- iii) Registers: Registers are temporary storage units within the CPU that hold data while it is being processed. They serve as a repository for operands, intermediate results, and memory addresses.
- iv) Memory: Random access memory (RAM) and read-only memory (ROM) are both included on the computer chip. RAM

- stores and retrieves data, whereas ROM stores predetermined instructions and data that remain intact during programme execution.
- v) Input and Output (I/O): The computer chip communicates with input devices (such as a keyboard) and output devices (such as a display) to enable interaction with the user and the outside world.
 - vi) Buses are communication routes that allow data and control signals to be exchanged between different components of a computer chip.

Students express the behaviour and structure of the components in the Hardware Description Language (HDL), which is used to build and implement the computer chip. They begin with logic gates and work their way through numerous projects, gradually developing the CPU, memory, and other components.

Students obtain a detailed understanding of how a computer system works from the transistor level up to a working CPU capable of interpreting instructions by completing the Computer chip project in the NAND to Tetris course. This hands-on experience teaches you about computer architecture, digital logic design, and low-level programming.

- As part of the NAND to Tetris course, you'll need the following main components to build a comprehensive computer system:

1. Central Processing Unit (CPU): The CPU is the computer's central processing unit, and it is in charge of executing instructions and conducting computations. It is made up of a control unit, an ALU (Arithmetic Logic

Unit), and registers.

2. Memory: Memory is where data and instructions that the CPU can access are stored. RAM (Random Access Memory) and ROM (Read-Only Memory) are both included. RAM is a type of volatile memory that may be read and written to, whereas ROM is a type of non-volatile memory that holds pre-defined instructions and data.

2.3 CPU:

- CPU is made up of logic gates like Nand, Mux16, Register, ALU (Arithmetic Logic unit) and PC (Program Counter) logic gates where, ALU makes the function part and Program Counter works like a register that selected the load, reset, increment components and change the output accordingly
- CPU has a length of 16 – bit long that executes the output as the following accordingly

Architecture of CPU :

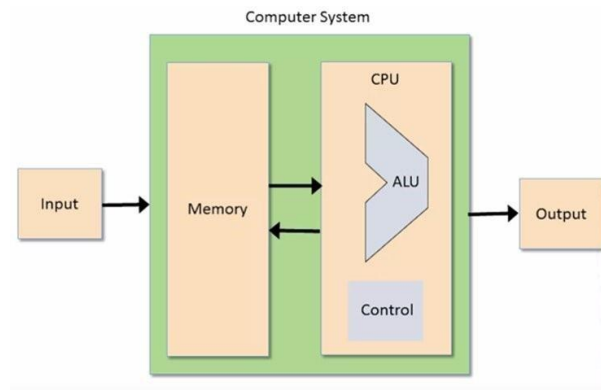


Figure 2: Von Neumann Architecture [6]

A CPU (Central Processing Unit) is the brain of a computer and is responsible for executing instructions and performing arithmetic and logical operations. The structure of a CPU

typically contains the following components:

1. The control unit retrieves instructions from memory and decodes them in order to determine the operation to be executed. It also handles the overall execution of the programme and governs the flow of data and signals between various sections of the CPU.
2. The ALU performs arithmetic and logical operations like as addition, subtraction, comparison, and bitwise operations. These operations are carried out in accordance with the instructions received from the control unit.
3. Registers: Registers are compact, quick storage units that are used to store interim results and other data. Each register on the CPU serves a distinct purpose, such as the programme counter.

(PC), the stack pointer (SP), and the general-purpose registers (GPRs). The memory part contains Register of A type and D type

1. Cache: Cache is a small amount of fast memory used to store frequently used data, allowing the CPU to access this data quickly and efficiently.

2. Bus Interface Unit (BIU): The BIU acts as an intermediary between the CPU and the main memory, allowing the CPU to fetch instructions and data from memory.

The CPU in the NAND2Tetris course is a simplified version of a real-world CPU, known as the Hack computer. It operates on a 16-bit instruction set architecture (ISA), meaning that each instruction is represented by a 16-bit binary value. The course introduces three main types of instructions: A-instructions, C-instructions, and jump instructions.

1. A-instructions (Addressing Instructions): A-instructions are used to set the value of the A register, which is a 16-bit register used for memory access. The A-instructions start with the symbol "@" followed by a memory address or a constant value. For example, @42 sets the A register to the value 42, while @SCREEN sets it to the memory address of the screen.

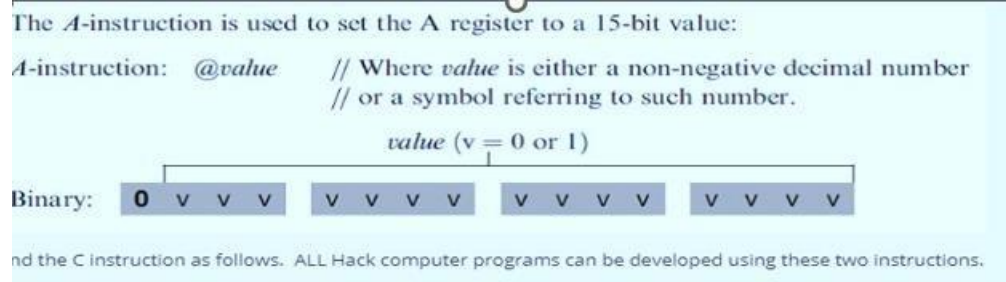


Figure 3 : Hack Instructions (Part 1)

2. **C-instructions (Computation Instructions):** C-instructions perform computations and control flow operations. They consist of three main parts: destination, computation, and jump. The destination part specifies where the computation result should be stored, the computation part specifies the operation to be performed, and the jump part determines whether to jump to a different instruction based on a condition.

The *C*-instruction is the programming workhorse of the Hack platform—the instruction that gets almost everything done. The instruction code is a specification that answers three questions: (a) what to compute, (b) where to store the computed value, and (c) what to do next? Along with the *A*-instruction, these specifications determine all the possible operations of the computer.

C-instruction: `dest=comp;jump` // Either the *dest* or *jump* fields may be empty.
 // If *dest* is empty, the “=” is omitted;
 // If *jump* is empty, the “;” is omitted.



Figure 4s : Hack Instructions (Part 2)

The general format of a *C*-instruction is: **dest=comp;jump**. The *dest* field is optional and specifies the destination register or memory location to store the computation result. The *comp* field specifies the computation to be performed, such as addition, bitwise operations, or logical operations. The *jump* field is optional and determines whether to perform a jump operation based on a condition.

3. **Jump Instructions:** Jump instructions are used to alter the control flow of the program

j1 (out < 0)	j2 (out = 0)	j3 (out > 0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If out > 0 jump
0	1	0	JEQ	If out = 0 jump
0	1	1	JGE	If out ≥ 0 jump
1	0	0	JLT	If out < 0 jump
1	0	1	JNE	If out ≠ 0 jump
1	1	0	JLE	If out ≤ 0 jump
1	1	1	JMP	Jump

Figure 5 : Jump instructions

based on certain conditions. They consist of a prefix "J" followed by a condition. Examples of jump instructions include `JGT` (jump if the last computation result is greater than zero), `JEQ` (jump if the last computation result is zero), and `JMP` (unconditional jump to a specified instruction address).

In the context of the NAND2Tetris course, students develop an assembler program that translates a higher-level assembly language into the binary machine code of the Hack computer. This assembler converts human-readable mnemonics, such as `@42` or `D=M`, into their corresponding binary representations.

Overall, understanding and implementing the different types of instructions in the CPU is a crucial part of building a functional computer architecture in the NAND2Tetris course, enabling the execution of programs written in the Hack assembly language.

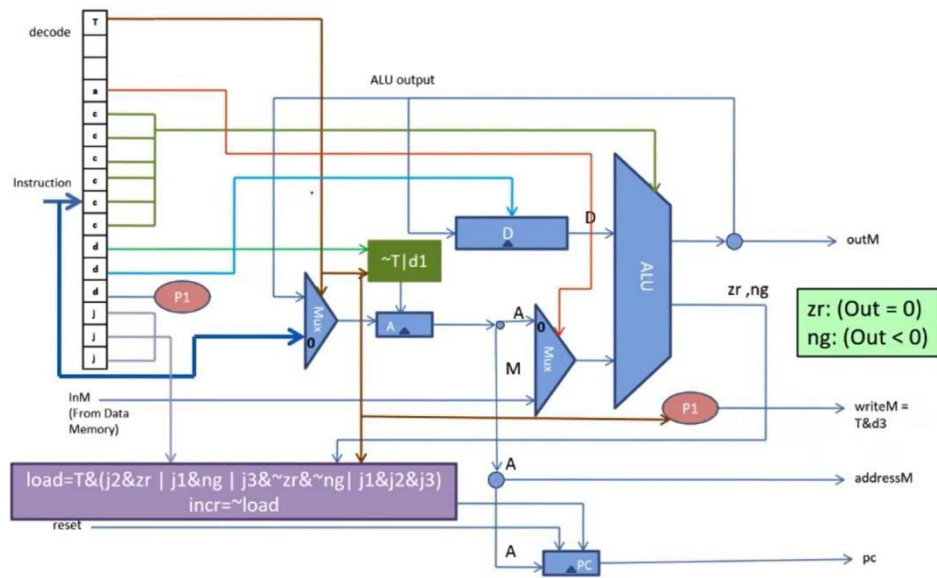
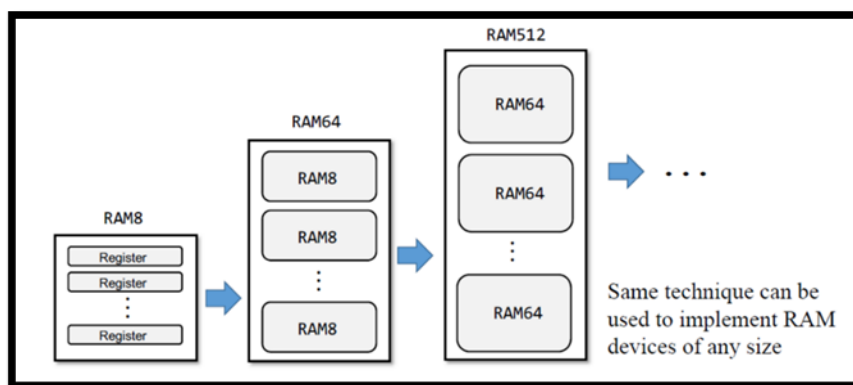


Figure 6 : Hack CPU

2.4 MEMORY:

The memory part of the NAND to Tetris course involves designing and implementing memory units to store and retrieve data within the computer system. Here's an overview of the memory components you'll typically encounter in the course:

1. **Bit:** The smallest unit of memory in a computer, representing a single binary digit (0 or 1).
2. **Register:** A group of sequential bits used to store and manipulate data within the computer. In the course, you'll typically work with 16-bit registers.
3. **RAM (Random Access Memory):** RAM is a type of volatile memory that allows for both reading and writing data. In the context of the course, you'll design a RAM chip capable of storing multiple registers.
4. **Memory Address:** Each memory location in RAM is assigned a unique address, which is used to identify and access specific data. In the case of the course, you'll typically work with a 15-bit memory address.



5. **Memory Module:** A memory module combines multiple RAM chips to provide a larger storage capacity. In the course, you'll construct a memory module capable of storing 16K (16,384) 16-bit values.

6. **Memory Access:** The process of reading or writing data from/to a specific memory location. You'll design circuitry that allows the computer's CPU to access and interact with the memory module.

2.5 MEMORY ALLOCATION :

In the NAND2Tetris course, the concept of keyboard memory mapping is used to handle input from the keyboard in the Hack computer architecture. The Hack computer is a simple computer system designed as part of the course, and it utilizes a memory-mapped I/O approach for handling input and output devices.

Memory mapping involves associating specific memory addresses with certain devices or functions. In the case of the Hack computer's keyboard, certain memory addresses are designated to represent different keys or key combinations on the keyboard. This allows the computer to read input from the keyboard by accessing the corresponding memory address.

The keyboard memory mapping in the Hack computer is as follows:

- Memory Address `KBD` (0x6000):

This memory address is reserved for reading the status of the keyboard. When the computer reads from this address, it retrieves the status of the keyboard, indicating whether a key has been pressed or not.

- Memory Address `SCREEN` (0x4000):

Although not directly related to keyboard mapping, it is worth mentioning that the `SCREEN` memory address is used for displaying output on the screen. It represents the virtual

memory address of the screen in the Hack computer architecture.

To read input from the keyboard, the Hack computer's software would typically check the value at the `KBD` memory address. If the value indicates that a key has been pressed, the software can retrieve the corresponding key code or character associated with that key.

The exact implementation of handling keyboard input and mapping keys to their corresponding memory addresses would be done in the software layer, typically in the form of an operating system or a program written in the Hack assembly language. The software would monitor the keyboard status and perform the necessary actions based on the keys pressed.

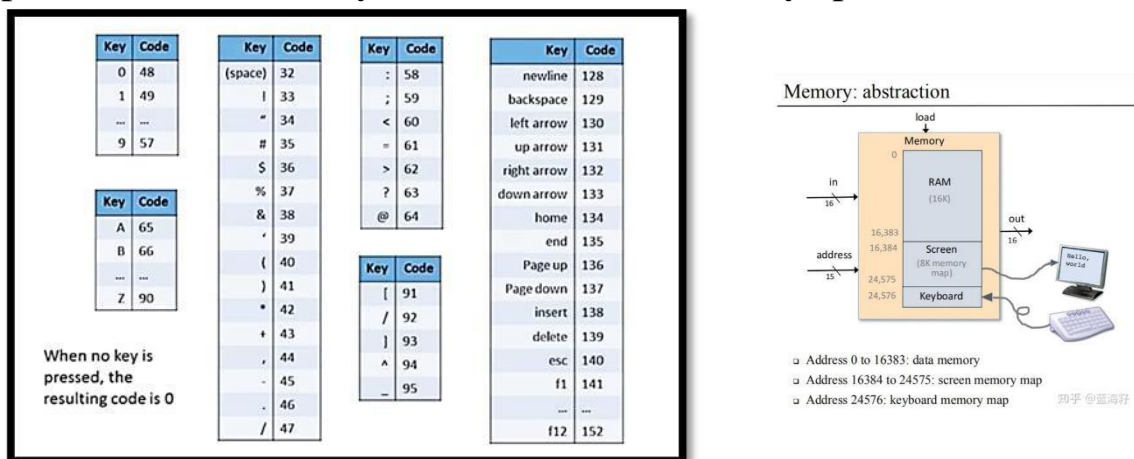


Figure 7 : Hack Character set

By using memory mapping for the keyboard, the Hack computer can treat input from the keyboard as if it were reading from memory addresses, providing a straightforward and consistent approach for handling keyboard input in the system.

During the memory part of the course, you'll learn about concepts like addressing, data storage, data retrieval, and the implementation of memory modules using logic gates. You'll

also develop an understanding of memory organization, including concepts like word size, byte addressing, and memory addressing modes.

To implement the memory components, you'll utilize the HDL (Hardware Description Language) learned in the course to design the necessary circuitry, including multiplexers, decoders, and other combinational and sequential logic circuits. You'll then test and verify the functionality of the memory units using the provided software tools and simulated hardware platforms.

By the end of the memory part, you should have a fully functional memory module that can store and retrieve data within your computer system. This will serve as an essential component in the overall architecture of the computer you're building as part of the NAND to Tetris course.

2.6 BUILDING OF COMPUTER :

To build a computer in the NAND2Tetris course, you would need to write HDL (Hardware Description Language) code to design and implement the various components of the computer architecture. HDL is a specialized language used for describing and simulating digital circuits. Here's a high-level overview of the components you would typically need to implement in HDL:

1. Logic Gates:

The basic building blocks of digital circuits are logic gates. In the NAND2Tetris course, you start by implementing elementary logic gates, such as NAND, AND, OR, and XOR gates. These gates serve as the foundation for building more complex components.

2. Arithmetic Logic Unit (ALU):

The ALU is responsible for performing arithmetic and logical operations. It takes input from registers and performs operations like addition, subtraction, bitwise operations, and comparisons. You would need to write HDL code for the ALU, considering the specific operations and bit widths

required by the computer architecture.

3. Registers:

Registers are used for storing data temporarily. In the Hack computer architecture, there are several types of registers, including the A register, D register, and memory registers (RAM). You would need to write HDL code to describe these registers and their functionalities.

4. Memory:

The Hack computer uses Random Access Memory (RAM) for storing instructions and data. The memory is organized into a large addressable space, and you would need to design and implement the memory module using HDL code. The course introduces the concept of a memory chip, which is essentially a collection of registers.

5. CPU:

The CPU (Central Processing Unit) is the heart of the computer. It fetches instructions, performs calculations, and controls the flow of data. To build the CPU, you would need to combine the ALU, registers, memory, and other components into a cohesive unit. The CPU executes

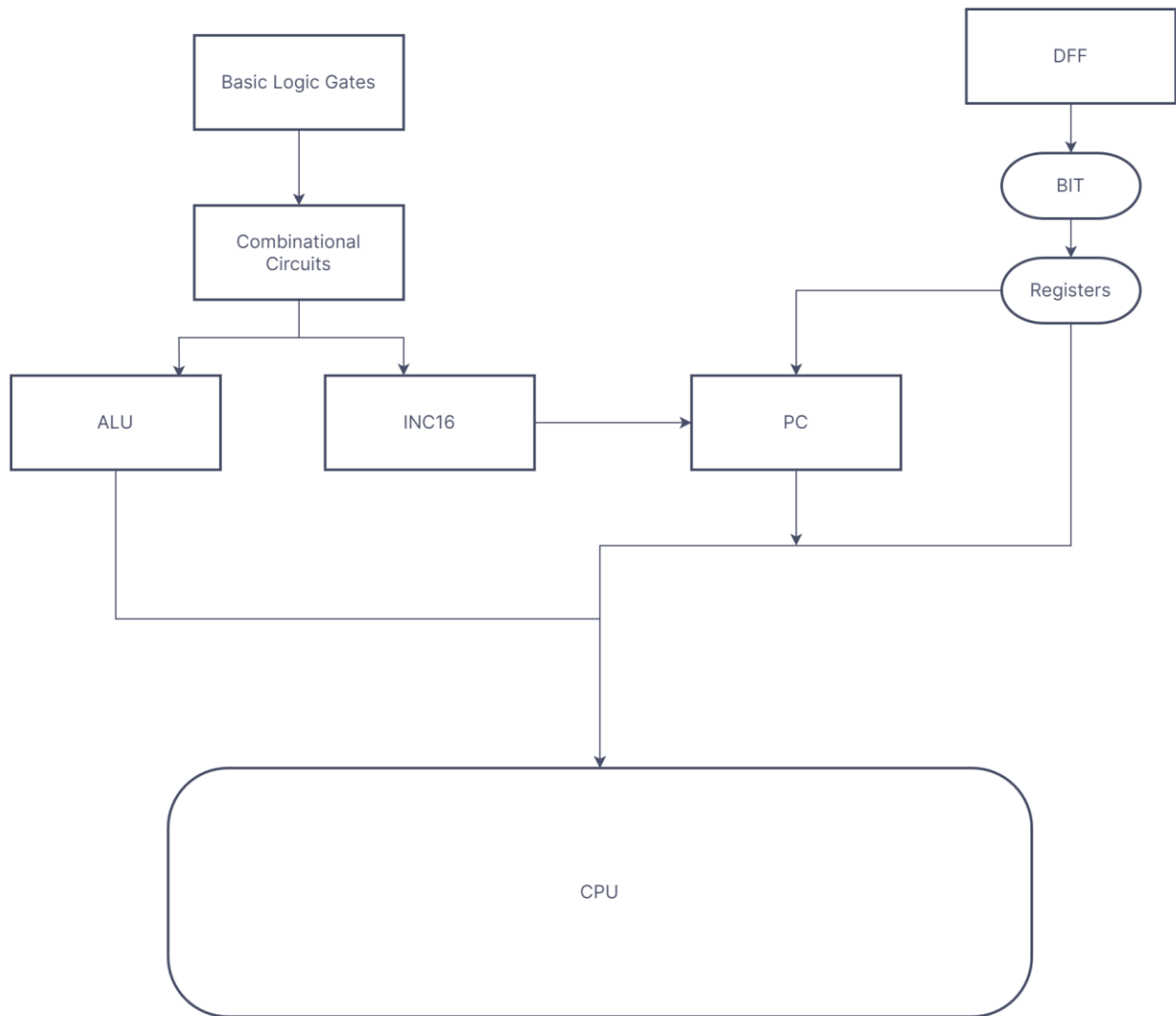
instructions based on the architecture's instruction set, which includes A-instructions, C-instructions, and jump instructions.

6. Control Logic:

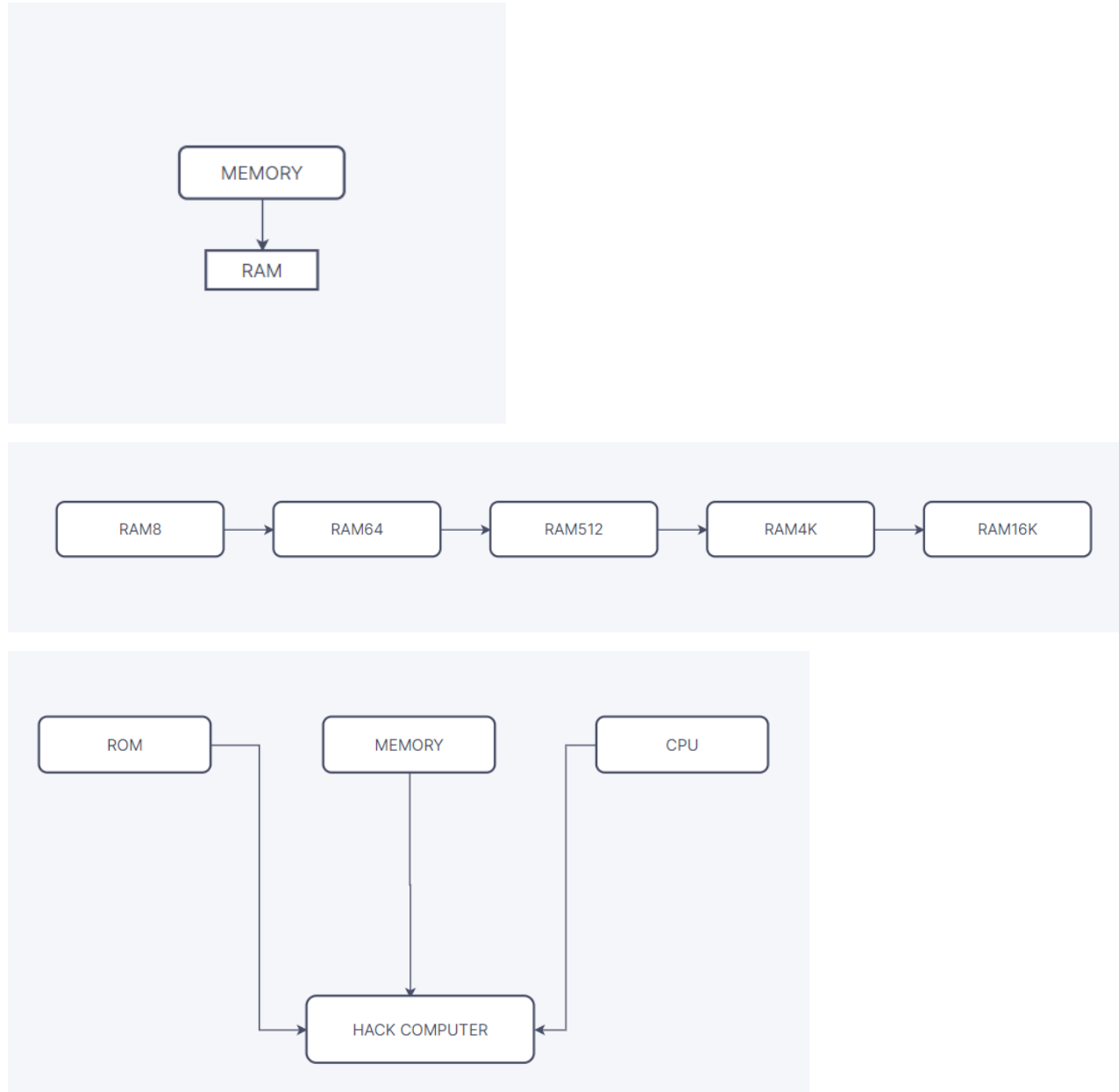
The control logic coordinates the operation of the CPU and ensures that the correct signals are generated for each instruction. It controls the flow of data, performs instruction decoding, and generates control signals to activate the appropriate components. Writing HDL code for the control logic involves implementing finite state machines and conditional logic to execute instructions.

These are some of the essential components you would need to implement in HDL to build a computer in the NAND2Tetris course. The course provides a step-by-step guide and incremental projects to help you gradually build and test each component. It's worth noting that the course provides a hardware simulator called the Hardware Simulator (often referred to as the "toolsuite"), which allows you to simulate and test your HDL designs at each stage of the computer's construction.

2.7 FLOW CHART:



2.8 MEMORY FLOW CHART:



After Building all these components we will then build CPU with HDL code as follows –

2.9 CPU HDL CODE :

```
CHIP CPU {
```

```
    IN inM[16],      // M value input (M =  
contents of RAM[A])    instruction[16],
```

```
// Instruction for execution    reset;
```

```
    // Signals whether to re-start the current  
// program (reset==1) or continue executing  
// the current program (reset==0).
```

```
    OUT outM[16],    // M  
value output    writeM,  
// Write to M?  addressM[15],  
// address of next instruction
```

PARTS:

```
    // Put your code here:  
Not(in=instruction[15],out=Ains);  
Not(in=Ains,out=Cins);  
Or(a=Ains,b=instruction[5],out=Aload);  
Mux16(a=instruction,b=fout,sel=Cins,out=Ain);  
ARegister(in=Ain,load=Aload,out=Aout,out[0..14]=address  
M);  
And(a=Cins,b=instruction[4],out=Dreg);
```

```

DRegister(in=fout,load=Dreg,out=Dout);
Mux16(a=Aout,b=inM,sel=instruction[12],out=secin);

ALU(x=Dout,y=secin,zx=instruction[11],nx=instruction[10]
,zy=instruction[9],ny
=instruction[8],f=instruction[7],no=instruction[6],out=outM,
out=fout,zr=zr,ng=ng);

And(a=instruction[2],b=ng,out=less);

And(a=instruction[1],b=zr,out=equal);

Not(in=zr,out=Notzr);

Not(in=ng,out=Notng);

And3(a=instruction[0],b=Notzr,c=Notng,out=great);

Or3(a=less,b=equal,c=great,out=preload);


And(a=Cins,b=preload,out=load);

Not(in=load,out=inc);


PC(in=Aout,reset=reset,load=load,inc=inc,out[0..14]=pc);

And(a=instruction[3],b=Cins,out=writeM);

}

```


2.10 HDL CODE BUILDING MEMORY :

```
CHIP RAM16K {
```

```
    IN in[16], load, address[14];
```

```
    OUT out[16];
```

PARTS:

```
// Put your code here:
```

```
DMux4Way(in=load,sel=address[12..13],a=o1,b=o2,c=o3,d=o4);
```

```
RAM4K(in=in,address=address[0..11],load=o1,out=s1);
```

```
RAM4K(in=in,address=address[0..11],load=o2,out=s2);
```

```
RAM4K(in=in,address=address[0..11],load=o3,out=s3);
```

```
RAM4K(in=in,address=address[0..11],load=o4,out=s4);
```

```
Mux4Way16(a=s1,b=s2,c=s3,d=s4,sel=address[12..13],out=out);
```

```
}
```

For RAM of Length (no of addresses) 16k

Since RAM is Mad of Bit and Bit is made of Mux and DFF

HDL code for BIT is – The HDL code for HACK Computer :

CHIP Bit {

IN in, load;

OUT out;

PARTS:

// Put your code here:

Mux(a=o1,b=in,sel=load,out=o2);

DFF(in=o2,out=o1,out=out);

}

Finally Hack Computer is Madeup of mainly these two components

CHIP Computer {

IN reset;

PARTS:

// Put your code here:

ROM32K(address=pc,out=instruction);

CPU(inM=inM,reset=reset,instruction=instruction,writeM=load,outM=outM,pc=pc,addressM=address);

Memory(in=outM,load=load,address=address,out=inM);

}

2.11 TESTING OF HACK COMPUTER CODE :

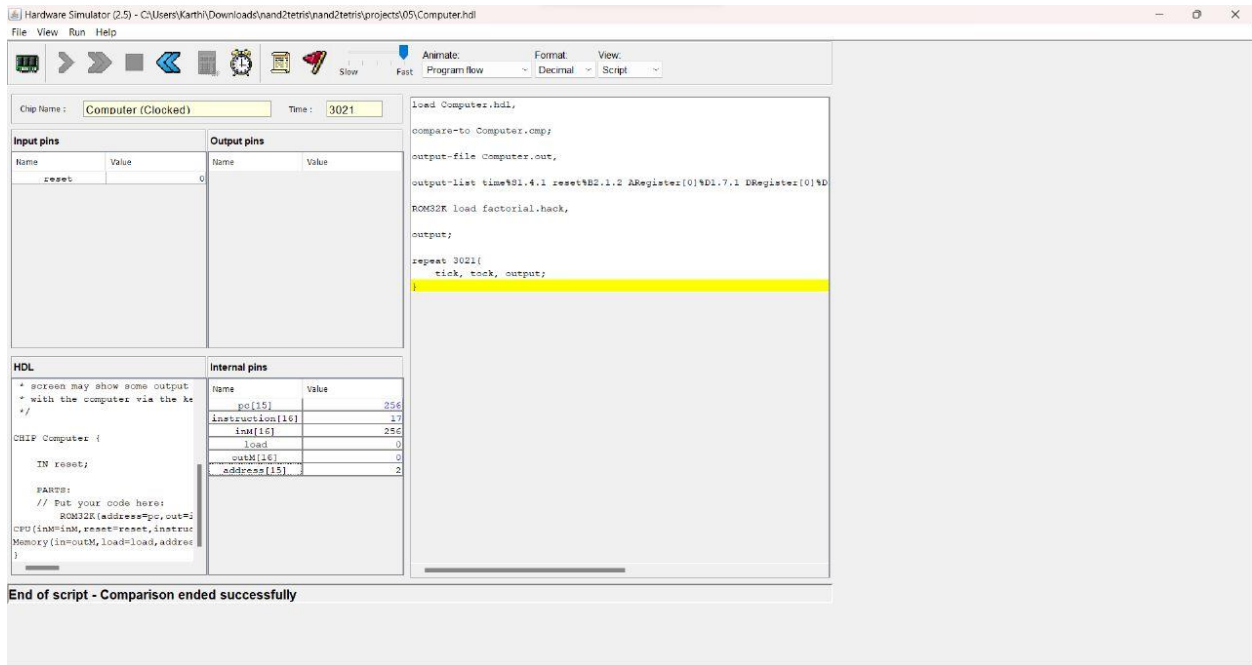


Figure 8: Computer output

3 PART B - TIC TAC TOE :

The Part – B of this project is to build a completely working Tic Tac Toe game using Jack High Level Language and implement it using the vm emulator. Also, we will convert these files to hack files(binary) using the Compiler, VM Translator and Assembler that we previously built in our assignments .We have built both single player and double player modes in this project. The code for all these is first done using Java to check if the algorithm works properly.

Tic tac toe is a game played by 2 people where a player plays as 'X' and the other player plays as 'O'. The game consists of a 3 x 3 board. Both the players have alternate turns to play. The first player to get 3 of his/her marks in a straight line(row, column or diagonal) wins the game.

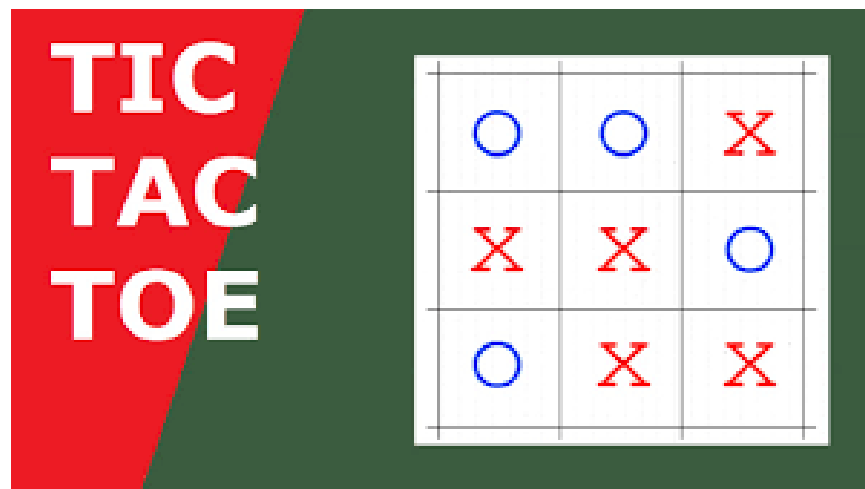


Figure 9 : Tic Tac Toe

In Single player mode, we will be playing as either 'X' or 'O' against the computer. The computer uses a recursive backtracking algorithm to make moves according to the moves by the player. This will find its best move by checking the probability of winning. It will place 'X' or 'O' in the square where the possibility of winning will be high. The possibility of winning is calculated using a recursive function. The double player mode is made for two players to play together at the same time and test their skills.

3.1 CODE IN JAVA (SINGLE PLAYER) :

3.1.1 Initialisation steps :

This part of the code uses only 1 inbuilt library from the util class that is used to take input from the user with the help of Scanner class and the Scanner(System.in) constructor. We define the public class with class name as the file name and a opening curly bracket ('{'). Then, we define the 'static char computer' to define a static variable of type char to store whether the computer is 'X' or 'O'.

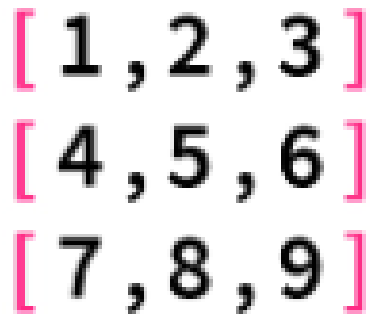
```
import java.util.*;  
  
public class Computer {  
  
    static char computer = ' ';
```

3.1.2 Main function :

We go for the main function now. In that, we define a variable of the scanner class in order to take input from the user. Secondly, we define a 2D char array of size 3 X 3 to act as the board. Then, we declare a char variable called player and get whether the player plays as 'X' or 'O'. Then, we define a boolean variable 'GameOver' to exit the loop if the game is over.

Now, we initialise the values of the board as ' '. This will be used in the condition to check if the place is already used. Now, we set the value of computer opposite to that of the player and player to 'X' by default as 'X' starts the game first. Now, we initialise a while loop with condition '!(GameOver)'. This is done to exit the loop when the game is over. Inside the while loop, we define a if-else loop. We check if the player is computer. If yes, we call a function called move that returns the board after placing the move of the computer onto the board. Then, we call the

function over to check if the game is over. Then, we change the value of the *player* variable and exit the loop. In the else loop, we first print the board and ask for the location to play from the user. We get the location from 1 to 9.



```
[ 1 , 2 , 3 ]
[ 4 , 5 , 6 ]
[ 7 , 8 , 9 ]
```

Figure 10 : Location values for insertion

Now, if the location is in between the range, we accept that or else print a error statement. We then get the row by dividing the (location-1) by 3. Then column by taking modulus of (location-1) with 3. We check if the value is ' '. If so, we place the player value in that location. Else, we print an error. Then, we check for GameOver, change the player value and exit the loop.

Now, outside the while loop, we print the board and then check if any player has won. If so, it prints the player won (computer or user) and if not, it prints '*GAME : DRAW*'.

```
public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);
    char[][] board = new char[3][3];
    char player=' ';
    System.out.print("Do you want to play as x(1) or o(2) : ");
    int pl = sc.nextInt();
    boolean GameOver = false;

    for(int i=0;i<board.length;i++){
        for(int j=0;j<board.length;j++){
            board[i][j] = ' ';
        }
    }
}
```

```

    }
}

if(pl==1){
    player = 'x';
    computer = 'o';
}
else if(pl == 2){
    player = 'x';
    computer = 'x';
}
else{
    System.out.println("ERROR : Enter the correct value");
    System.exit(1);
}

while(!(GameOver)){
    if(player == computer){
        board = move(board,computer);
        GameOver = over(board,player);
        if(player == 'x' ){
            player = 'o';
        }
        else{
            player = 'x';
        }
    }
    else{
        display(board);
        System.out.print("Enter the place to be entered : ");
        int val = sc.nextInt(),row,col;

        if(val >= 1 && val<=9 ){
            row = (val-1)/3;
            col = (val-1)%3;

```

```

        if(board[row][col] == ' '){
            board[row][col] = player;

            GameOver = over(board,player);
            if(player == 'x' ){
                player = 'o';
            }
            else{
                player = 'x';
            }
        }
        else{
            System.out.println("ERROR : The location is already
used");
        }
    }
    else{
        System.out.println("ERROR : Invalid Location to place a
value");
    }
}

if(player == computer){
    if(computer == 'o'){
        player = 'x';
    }
    else{
        player = 'o';
    }
}

display(board);

```



```

    if(win(board,computer)){
        System.out.println("You have lost the game");
    }
    else if(win(board,player)){
        System.out.println("Congragulations! You have won the game");
    }
    else{
        System.out.println("Game : DRAW");
    }
}

```

3.1.3 Move function :

Inside the move function, we define a temp variable to get the value of the player. We initially check if the computer could win in any of the next move. If so, we give priority to that area and place our computer value at that place and return the board. Next, we check if the player could win in the next move. If so, we place our computer value at that point and return the board. If none of these conditions get satisfied, we check how many empty spaces are there, in order to create an array of that size to store the possibilities of winning in each. Then, we run for loops again to call the possibility function in each and store the value of the possibilities in an array. In each of these, we enter the value into the board and give it inside the respective functions. Then, we replace it to its initial value so that rest of the recursive part doesn't get affected. This is known as *Backtracking*. We then find the max value from that array and place the computer value at that point and return the value. This ensures that the computer plays the best move possible.

```

public static char[][] move(char[][] board,char player){

    char temp;
    if(computer == 'x'){
        temp = 'o';
    }
}

```

```

    }
    else{
        temp = 'x';
    }

    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            if(board[i][j] == ' '){
                board[i][j] = computer;
                if(win(board,computer)){
                    board[i][j] = computer;
                    return board;
                }
                board[i][j] = ' ';
            }
        }
    }
}

```

```

    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            if(board[i][j] == ' '){
                board[i][j] = temp;
                if(win(board,temp)){
                    board[i][j] = computer;
                    return board;
                }
                board[i][j] = ' ';
            }
        }
    }
}

```

```

int k=0;
for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){

```

```

        if(board[i][j] == ' '){
            k++;
        }
    }
}

int[] possible = new int[k];
int k1=0;

for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        if(board[i][j] == ' '){
            board[i][j] = player;
            int pos = possibility(board,player);
            possible[k1] = pos;
            k1++;
            board[i][j] = ' ';
        }
    }
}

int max = possible[0];
int index = 0;
for (int i = 0; i < possible.length; i++)
{
    if (max < possible[i])
    {
        max = possible[i];
        index = i;
    }
}

k=0;
for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){

```

```

        if(board[i][j] == ' '){
            if(k==index){
                board[i][j] = player;
                return board;
            }
            k++;
        }
    }
}

return board;
}

```

3.1.4 Possibility function :

Now, we go for the possibility function. We check if all the locations are filled. If so, we check if the computer wins or the player wins or it's a draw. For computer we return 1, for player we return -1 and for draw we return 0. These get added in each recursion step and denotes the final probability. In the else loop, we change the player value and place it at all possible values using a for loop, and then call the probability function again. This adds the value of the current probability to its recursive caller. Then, we return the final output.

```

public static int possibility(char[][] board,char player){

    int k=0;
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            if(board[i][j] != ' '){
                k++;
            }
        }
    }
}

```

```

if(k==9){
    char temp;
    if(computer == 'x'){
        temp = 'o';
    }
    else{
        temp = 'x';
    }

    if(win(board,computer)){
        return 1;
    }
    else if(win(board,temp)){
        return -1;
    }
    else{
        return 0;
    }
}
else{
    if(player == 'x'){
        player = 'o';
    }
    else{
        player = 'x';
    }
}

int pos=0;

for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        if(board[i][j] == ' '){
            board[i][j] = player;
            pos += possibility(board, player);
            board[i][j] = ' ';
        }
    }
}

```

```

    }
}
return pos;
}
}

```

3.1.5 Display function :

This is used to display the board. We run two for loops to basically print the board values in a matrix format and use '|' and '-----' for convenience while playing the game.

```

public static void display(char[][] board) {

    System.out.println("-----");

    for(int i=0;i<board.length;i++){
        System.out.print("|");
        for(int j=0;j<board.length;j++){
            System.out.print(board[i][j] + "|");
        }
        System.out.println("\n-----");
    }
}

```

3.1.6 Over function :

We check if the current player has won. The winning conditions are the player should have entered the same value either row wise or column wise. This is done using for loops. Also, the player could win diagonal wise. We check for all these conditions. The other exit condition is when the board is completely filled. We check for all these using basic for loops.

```

public static boolean over(char[][] board,char player){

    for(int i=0;i<3;i++){
        if(board[i][0] == player && board[i][1] == player &&
board[i][2] == player){
            return true;
        }
    }

    for(int i=0;i<3;i++){
        if(board[0][i] == player && board[1][i] == player &&
board[2][i] == player){
            return true;
        }
    }

    for(int i = 0 ; i<=2 ; i=i+2){
        if(board[0][i] == player && board[1][1] == player &&
board[2][2-i] == player){
            return true;
        }
    }

    int k=0;
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            if(board[i][j] != ' '){
                k++;
            }
        }
    }

    if(k==9){
        return true;
    }
}

```

```
    return false;
}
```

3.1.7 Win function :

This includes the same condition as over function without the last case .i.e. when the board is completely full as the player cannot win if the board is completely filled and there is no straight line in their moves. We then close the class.

```
public static boolean win(char[][] board,char player){
    for(int i=0;i<3;i++){
        if(board[i][0] == player && board[i][1] == player &&
board[i][2] == player){
            return true;
        }
    }

    for(int i=0;i<3;i++){
        if(board[0][i] == player && board[1][i] == player &&
board[2][i] == player){
            return true;
        }
    }

    for(int i = 0 ; i<=2 ; i=i+2){
        if(board[0][i] == player && board[1][1] == player &&
board[2][2-i] == player){
            return true;
        }
    }
    return false;
}
```


3.2. CODE IN JAVA (DOUBLE PLAYER) :

Now, we are going to implement the game in Double player mode. We start with implementing the same library, defining the main class, the scanner and the board. We initialise the board values as ' '. Then, we define the player 'X' as default as 'X' starts first always and also the GameOver variable. We then initialise a while loop with condition '!(GameOver)'. Inside that, we ask for the location of the value to be entered. We then check if the location lies in the range of [1,9]. If not, we print an error statement. Then, we check if the location is already used. If so, we print error. If not, we enter the value in the board and check if the game is over. If yes, the loop will exit. Then, we change the current player. The final part of the main function is to print if the person wins or if the game ends in a draw.

```
public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);
    char[][] board = new char[3][3];
    int i,j,row,col,val;

    for(i=0;i<board.length;i++){
        for(j=0;j<board.length;j++){
            board[i][j] = ' ';
        }
    }

    boolean GameOver = false;
    char player = 'X';
    while(!GameOver){
        display(board);
        System.out.print("Enter the place to be entered : ");
        val = sc.nextInt();
        if(val >= 1 && val<=9 ){
            row = (val-1)/3;
```

```

        col = (val-1)%3;

        if(board[row][col] == ' '){
            board[row][col] = player;
            GameOver = over(board,player);
            player = (player=='X')? 'O' : 'X' ;
        }
        else{
            System.out.println("ERROR : The location is already
used");
        }
    }
    else{
        System.out.println("ERROR : Invalid Location to place a
value");
    }
}
for(i=0;i<3;i++){
    for(j=0;j<3;j++){
        if(board[i][j] == ' '){
            display(board);
            player = (player=='X')? 'O' : 'X' ;
            System.out.println("Player "+player+ " has won the game");
            return;
        }
    }
}
display(board);
System.out.println("Game Over : DRAW");
}

```

Next, we have the display and over functions. These are the same for both single and double player. With this, the game has been built using Java. Both the Java codes are given below.

3.3 CODE IN JACK:

3.3.1 Main File :

Inside the main class we define the function void main(). We have to define the Main class and then the main function. This is because Main.main is the entry point in jack language. Inside the main function, we define a variable 'g' of type Game. Then, we call g.go() and then the return statement.

```
class Main{  
    function void main(){  
        var Game g;  
  
        do g.go();  
        return;  
    }  
}
```

3.3.2 Game File :

In this, we define the variables 'set' of type Setup, check of type int, single of type Single, double of type Double. All these are used to set the game up and play the game. We first do clear screen in order to make sure that there is no previous values. Then, we call the documentation function. Then, we call the draw function. We use the check variable to ask the user whether it is a single player mode or double player mode and store the value. We have used 1 for Single player and 2 for double player. If it is 1, we call the set.clear_line() function and then the Single.run() function. Else, we check if it is 2 in a nested if-else loop, and if so, we will call the set.clear_line() and then the double.run() function. In the nested else loop, we print a error statement, wait for 2 seconds and then call go() again. After it exits the if-else loop, we wait for 3.5 seconds and the clear the screen. Then , we print the statement "click 'R' to restart the game". We then wait for a keypress with value 82 as the ASCII value of R is 82. Then,

we call the go() function again. This is done so that we can restart the code after the end of each and every game. Then, we end the go function with a return statement and the Game class with a '}'.

```
class Game{
    method void go(){
        var Setup set;
        var int check;
        var Single single;
        var Double double;

        do Screen.clearScreen();
        do set.documentation();
        do set.draw();
        let check = set.player_computer();

        if(check = 1){
            do set.clear_line();
            do single.run();
        }
        else{
            if(check = 2){
                do set.clear_line();
                do double.run();
            }
            else{
                do set.clear_line();
                do Output.printString("ERROR : Please enter either 1 or 2");
                do Sys.wait(2000);
                do go();
            }
        }

        do Sys.wait(3500);
        do Screen.clearScreen();
    }
}
```

```

do Output.moveCursor(11,17);
do Output.printString("CLICK 'R' TO RESTART THE GAME");

while(~(Keyboard.keyPressed() = 82)){
}

do go();

return;
}
}

```

3.3.3 Setup file :

The setup file contains the setup class which is the most important class in this specific project. This is because it contains all the useful functions such as draw, documentation, over, win, draw_x, draw_y, etc. We will now look at all these functions in detail.

3.3.3.1 Documentation :

This is to add the documentation line to our project. The documentation used in this project is the subject name and group name. The line is “Elements of computing systems – 2 : Group 14 | PART B”.

```

method void documentation(){
do Output.moveCursor(22,6);
do Output.printString("Elements of Computing Systems - 2 : Group
14 | PART B");
do Output.moveCursor(0,0);
return;
}

```

3.3.3.2 Mode :

This will ask the user if the mode is single player mode or double player mode.

```
method int player_computer(){
    var int check;
    let check = Keyboard.readInt("Single Player(1) | Double Player(2)
: ");
    return check;
}
```

3.3.3.3 Draw :

This function is to draw the grid for playing Tic Tac Toe. The coordinates of the rectangle were found using trial and error method. Then, 4 filled rectangles were drawn with those coordinates. This will act as the board.

```
method void draw(){
    do Screen.setColor(true);
    do Screen.drawRectangle(156,82,356,92);
    do Screen.drawRectangle(156,152,356,162);
    do Screen.drawRectangle(216,22,226,222);
    do Screen.drawRectangle(286,22,296,222);
    return;

}
```

3.3.3.4 Clear line :

This function will clear the line entered by drawing a rectangle with set colour true. This means erasing the line entirely. Then, it moves the cursor to (0,0) in order that the next line appears at that point.

```

method void clear_line(){
    do Screen.setColor(false);
    do Screen.drawRect(0,0,511,10);
    do Output.moveCursor(0,0);
    return;
}

```

3.3.3.5 Board :

This function is used to access the board that we play. This is done because there is no multi dimensional array in jack. For this purpose, we use array of arrays and access the value using a function. We return the value present at the row and column given.

```

method int board(Array arr,int x,int y){
    var int temp;
    var Array temp1;

    let temp1 = arr[x];
    let temp = temp1[y];

    return temp;
}

```

3.3.3.6 Set Board :

The set board function is used to set the value given at the row and column given by the user into the board. It returns the board afterwards.

```

method Array set_board(Array arr,int x,int y,int val){
    var Array temp;

    let temp = arr[x];
    let temp[y] = val;
}

```

```

let arr[x] = temp;

return arr;
}

```

3.3.3.7 Draw X :

This function draws the symbol 'X' at the location given by the user. The X is created using Bitmap editor from the folder 9 of nand2tetris.

IDC Herzliya / Efi Arazi School of Computer Science / Digital Systems Construction, Spring 2011 / Project 09 / Golan Parashi

Sokoban Bitmap Editor

This javascript applicaiton is used to generate highly optimized jack code for drawing a 16x16 bitmap to the screen.

Using the mouse, click the desired cell to mark/unmark it. You may use 90 degrees rotation and vertical mirroring by clicking the appropriate buttons.

When you are finished drawing, you may select function type and enter function's name.

Bitmap

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1																
2																
3																
4																
5																
6																
7																
8																
9																
10																
11																
12																
13																
14																
15																
16																

Function Type:

Function Name:

Generated Jack Code

```

function void draw(int location) {
    let memAddress = 16384+location;
    do Memory.poke(memAddress+0, 0);
    do Memory.poke(memAddress+32, 0);
    do Memory.poke(memAddress+64, 0);
    do Memory.poke(memAddress+96, 0);
    do Memory.poke(memAddress+128, 0);
    do Memory.poke(memAddress+160, 0);
    do Memory.poke(memAddress+192, 0);
    do Memory.poke(memAddress+224, 0);
    do Memory.poke(memAddress+256, 0);
    do Memory.poke(memAddress+288, 0);
    do Memory.poke(memAddress+320, 0);
    do Memory.poke(memAddress+352, 0);
    do Memory.poke(memAddress+384, 0);
    do Memory.poke(memAddress+416, 0);
    do Memory.poke(memAddress+448, 0);
    do Memory.poke(memAddress+480, 0);
    return;
}

```

Figure 11 : Bitmap ediitor

This will give us the code to draw a 16 x 16 entity in the jack screen. But, the size 16 x 16 is in itself small. So, we draw $\frac{1}{4}$ th of the entity(x and o) each time and combine them into one by changing the location variable. The location is the register number of the screen. There are a total of 8192 registers used by the hack screen. 32 in each row. So, we give the 1st part

as the location, the 2nd part as the location + 1, the 3rd as location + 32 and 4th as location + 33. This will place all the blocks in correct order and form the respective shape. We can also use the rotate right and vertical mirror options to get the other 4 blocks and both 'X' and 'O' are symmetric at their centers.

```
method void draw_x(int loc){  
  
    var int memAddress,location;  
    let location = location_fn(loc);  
  
    let memAddress = 16384+location;  
    do Memory.poke(memAddress+0, 2);  
    do Memory.poke(memAddress+32, 7);  
    do Memory.poke(memAddress+64, 14);  
    do Memory.poke(memAddress+96, 28);  
    do Memory.poke(memAddress+128, 56);  
    do Memory.poke(memAddress+160, 112);  
    do Memory.poke(memAddress+192, 224);  
    do Memory.poke(memAddress+224, 448);  
    do Memory.poke(memAddress+256, 896);  
    do Memory.poke(memAddress+288, 1792);  
    do Memory.poke(memAddress+320, 3584);  
    do Memory.poke(memAddress+352, 7168);  
    do Memory.poke(memAddress+384, 14336);  
    do Memory.poke(memAddress+416, -4096);  
    do Memory.poke(memAddress+448, -8192);  
    do Memory.poke(memAddress+480, -8192);  
  
    let memAddress = 16384+location+1;  
    do Memory.poke(memAddress+0, 16384);  
    do Memory.poke(memAddress+32, -8192);  
    do Memory.poke(memAddress+64, 28672);  
    do Memory.poke(memAddress+96, 14336);  
    do Memory.poke(memAddress+128, 7168);
```

```
do Memory.poke(memAddress+160, 3584);
do Memory.poke(memAddress+192, 1792);
do Memory.poke(memAddress+224, 896);
do Memory.poke(memAddress+256, 448);
do Memory.poke(memAddress+288, 224);
do Memory.poke(memAddress+320, 112);
do Memory.poke(memAddress+352, 56);
do Memory.poke(memAddress+384, 28);
do Memory.poke(memAddress+416, 15);
do Memory.poke(memAddress+448, 7);
do Memory.poke(memAddress+480, 7);
```

```
let memAddress = 16384+location+(32*16);
do Memory.poke(memAddress+0, -8192);
do Memory.poke(memAddress+32, -8192);
do Memory.poke(memAddress+64, -4096);
do Memory.poke(memAddress+96, 14336);
do Memory.poke(memAddress+128, 7168);
do Memory.poke(memAddress+160, 3584);
do Memory.poke(memAddress+192, 1792);
do Memory.poke(memAddress+224, 896);
do Memory.poke(memAddress+256, 448);
do Memory.poke(memAddress+288, 224);
do Memory.poke(memAddress+320, 112);
do Memory.poke(memAddress+352, 56);
do Memory.poke(memAddress+384, 28);
do Memory.poke(memAddress+416, 14);
do Memory.poke(memAddress+448, 7);
do Memory.poke(memAddress+480, 2);
```

```
let memAddress = 16384+location+(32*16)+1 ;
do Memory.poke(memAddress+0, 7);
do Memory.poke(memAddress+32, 7);
do Memory.poke(memAddress+64, 15);
do Memory.poke(memAddress+96, 28);
```

```

do Memory.poke(memAddress+128, 56);
do Memory.poke(memAddress+160, 112);
do Memory.poke(memAddress+192, 224);
do Memory.poke(memAddress+224, 448);
do Memory.poke(memAddress+256, 896);
do Memory.poke(memAddress+288, 1792);
do Memory.poke(memAddress+320, 3584);
do Memory.poke(memAddress+352, 7168);
do Memory.poke(memAddress+384, 14336);
do Memory.poke(memAddress+416, 28672);
do Memory.poke(memAddress+448, -8192);
do Memory.poke(memAddress+480, 16384);

return;
}

```

3.3.3.8 Draw O :

The Draw O function is also created in the same way as mentioned above.

```

method void draw_o(int loc){

var int memAddress,location;
let location = location_fn(loc);

let memAddress = 16384+location;
do Memory.poke(memAddress+0, -1024);
do Memory.poke(memAddress+32, -256);
do Memory.poke(memAddress+64, -128);
do Memory.poke(memAddress+96, 8128);
do Memory.poke(memAddress+128, 2016);
do Memory.poke(memAddress+160, 496);
do Memory.poke(memAddress+192, 248);
do Memory.poke(memAddress+224, 120);
do Memory.poke(memAddress+256, 60);

```

```
do Memory.poke(memAddress+288, 28);
do Memory.poke(memAddress+320, 30);
do Memory.poke(memAddress+352, 14);
do Memory.poke(memAddress+384, 15);
do Memory.poke(memAddress+416, 7);
do Memory.poke(memAddress+448, 7);
do Memory.poke(memAddress+480, 7);
```

```
let memAddress = 16384+location+1;
do Memory.poke(memAddress+0, 63);
do Memory.poke(memAddress+32, 255);
do Memory.poke(memAddress+64, 511);
do Memory.poke(memAddress+96, 1016);
do Memory.poke(memAddress+128, 2016);
do Memory.poke(memAddress+160, 3968);
do Memory.poke(memAddress+192, 7936);
do Memory.poke(memAddress+224, 7680);
do Memory.poke(memAddress+256, 15360);
do Memory.poke(memAddress+288, 14336);
do Memory.poke(memAddress+320, 30720);
do Memory.poke(memAddress+352, 28672);
do Memory.poke(memAddress+384, -4096);
do Memory.poke(memAddress+416, -8192);
do Memory.poke(memAddress+448, -8192);
do Memory.poke(memAddress+480, -8192);
```

```
let memAddress = 16384+location+(32*16);
do Memory.poke(memAddress+0, 7);
do Memory.poke(memAddress+32, 7);
do Memory.poke(memAddress+64, 7);
do Memory.poke(memAddress+96, 15);
do Memory.poke(memAddress+128, 15);
do Memory.poke(memAddress+160, 31);
do Memory.poke(memAddress+192, 30);
do Memory.poke(memAddress+224, 62);
```

```

do Memory.poke(memAddress+256, 124);
do Memory.poke(memAddress+288, 248);
do Memory.poke(memAddress+320, 496);
do Memory.poke(memAddress+352, 2016);
do Memory.poke(memAddress+384, 8128);
do Memory.poke(memAddress+416, -256);
do Memory.poke(memAddress+448, -1024);
do Memory.poke(memAddress+480, -4096);

let memAddress = 16384+location+1+(32*16);
do Memory.poke(memAddress+0, -8192);
do Memory.poke(memAddress+32, -8192);
do Memory.poke(memAddress+64, -8192);
do Memory.poke(memAddress+96, -4096);
do Memory.poke(memAddress+128, -4096);
do Memory.poke(memAddress+160, -2048);
do Memory.poke(memAddress+192, 30720);
do Memory.poke(memAddress+224, 31744);
do Memory.poke(memAddress+256, 15872);
do Memory.poke(memAddress+288, 7936);
do Memory.poke(memAddress+320, 3968);
do Memory.poke(memAddress+352, 2016);
do Memory.poke(memAddress+384, 1016);
do Memory.poke(memAddress+416, 255);
do Memory.poke(memAddress+448, 63);
do Memory.poke(memAddress+480, 15);
return;
}

```

3.3.3.9 General draw :

The grid has a 60 x 60 space for drawing either 'X' or 'O'. Then, the grid is drawn as a rectangle with width 10 and length 200 ($60*3 + 10*2$) pixels. The 'X' and 'O' has a size of 32 x 32 pixels. This is because we use four 16 x 16 blocks to build this.

3.3.3.10 Location :

This function is made to find the register location with the help of the location given between 1 to 9 as the input to the board. All the register location values are found using trial and error method.

```
method int location_fn(int loc){  
  
    var int location;  
  
    if(loc=1){  
        let location = (35*32) + 10;  
    }  
    if(loc=2){  
        let location = (35*32) + 15;  
    }  
    if(loc=3){  
        let location = (35*32) + 19 ;  
    }  
    if(loc=4){  
        let location = (105*32) + 10;  
    }  
    if(loc=5){  
        let location = (105*32) + 15;  
    }  
    if(loc=6){  
        let location = (105*32) + 19;  
    }  
    if(loc=7){  
        let location = (175*32) + 10;  
    }  
    if(loc=8){  
        let location = (175*32) + 15;  
    }  
    if(loc=9){
```

```

    let location = (175*32) + 19;
}

return location;
}

```

3.3.3.11 Win and Over functions :

The win and over functions are created the same way as we did in java. But here, we use 0 for ' ', 1 for 'X' and 2 for 'O'. This is because it is difficult to use character variables in java as there is no direct initialisation of char variables.

Over :

```

method Array set_board(Array arr,int x,int y,int val){
    var Array temp;

    let temp = arr[x];
    let temp[y] = val;
    let arr[x] = temp;

    return arr;
}

method boolean check(Array arr,int player){

    var int i,j,k;
    let i=0;

    while(~(i=3)){
        if(board(arr,i,0) = player){
            if(board(arr,i,1) = player){
                if(board(arr,i,2) = player){

```

```

        return true;
    }
}
}
let i = i+1;
}

let i=0;
while(~(i=3)){
    if(board(arr,0,i) = player){
        if(board(arr,1,i) = player){
            if(board(arr,2,i) = player){
                return true;
            }
        }
    }
    let i = i+1;
}

if(board(arr,0,0) = player){
    if(board(arr,1,1) = player){
        if(board(arr,2,2) = player){
            return true;
        }
    }
}

if(board(arr,0,2) = player){
    if(board(arr,1,1) = player){
        if(board(arr,2,0) = player){
            return true;
        }
    }
}
}

```



```

let i=0;
let k = 0;
while(~(i=3)){
    let j = 0;
    while(~(j=3)){
        if(~(board(arr,i,j) = 0)){
            let k = k+1;
        }
        let j = j+1;
    }
    let i = i+1;
}

if(k=9){
    return true;
}

return false;
}

```

Win :

```

method boolean win(Array arr,int player){
    var int i,j,k;
    let i=0;

    while(~(i=3)){
        if(board(arr,i,0) = player){
            if(board(arr,i,1) = player){
                if(board(arr,i,2) = player){
                    return true;
                }
            }
        }
        let i = i+1;
    }
}

```

```

    }

    let i=0;
    while(~(i=3)){
        if(board(arr,0,i) = player){
            if(board(arr,1,i) = player){
                if(board(arr,2,i) = player){
                    return true;
                }
            }
        }
        let i = i+1;
    }

    if(board(arr,0,0) = player){
        if(board(arr,1,1) = player){
            if(board(arr,2,2) = player){
                return true;
            }
        }
    }

    if(board(arr,0,2) = player){
        if(board(arr,1,1) = player){
            if(board(arr,2,0) = player){
                return true;
            }
        }
    }

    return false;
}

```

3.3.4 Single file :

3.3.4.1 Run :

We first declare and initialise the variables. Then, we get who is playing as the computer using the get function. We then set the player to 1 .i.e. 'X' to default as 'X' starts the round always. Now, we initialise a while loop to check if the Game is over and then set the location variable to the location given by the move function. Then, we check if the is 1 or 2 (X or O) and call the respective draw function. We then find the row and column of the input using the location given. We then update the board to that value. Then, we change the current player. In the else loop, we print the statements required to get the user input, and check if the location entered is between 1 to 9. If not, we print an error. If yes, we check if the location entered already has a value. If yes, we print an error. If not, we update the board to that value, check if game is over and then change the current player. Then, we exit the loop. Now, we check who won using the win function and store the player who won the game. We also print the string Game Over !!! at the top of the screen. We then display the player who won. If the computer has won, we print "OOPS!!! You have lost the Game". If the player has won, we print "Congragulations!!! You have won the game". If the game ends in a draw, we print "GAME : DRAW". Then, we return the run function.

```
method void run(){
    var Array row1,row2,row3;
    var Array board1;
    var boolean Game_Over,win1,win2;
    var int player,location,row,col,i,j,k,computer,temp;
    var Setup set;

    let Game_Over = false;
    let player = 1;

    let row1 = Array.new(3);
```

```

let row2 = Array.new(3);
let row3 = Array.new(3);
let board1 = Array.new(3);

let row1[0] = 0;
let row1[1] = 0;
let row1[2] = 0;
let row2[0] = 0;
let row2[1] = 0;
let row2[2] = 0;
let row3[0] = 0;
let row3[1] = 0;
let row3[2] = 0;
let board1[0] = row1;
let board1[1] = row2;
let board1[2] = row3;

let computer = get();
let player = 1;

while(~(Game_Over)){
  if(player = computer){
    let location = move(board1,player,computer);

    if(player = 1){
      do set.draw_x(location);
    }
    else{
      do set.draw_o(location);
    }

    let location = location-1;
    let row = location/3;
    let col = location - (3 * (location/3));
    let board1 = set.set_board(board1,row,col,player);
  }
}

```

```

let Game_Over = set.check(board1,player);

if(player = 1){
    let player = 2;
}
else{
    let player = 1;
}
}
else{
    let location = Keyboard.readInt("Your turn (Enter a number
between 1 to 9) : ");
    do set.clear_line();

    if((location<10) & (location>0)){

        let location = location-1;
        let row = location/3;
        let col = location - (3 * (location/3));

        if(set.board(board1,row,col) = 0){

            let location = location+1;
            let board1 = set.set_board(board1,row,col,player);

            if(player = 1){
                do set.draw_x(location);
            }
            else{
                do set.draw_o(location);
            }

            let Game_Over = set.check(board1,player);

```

```

        if(player = 1){
            let player = 2;
        }
        else{
            let player = 1;
        }
    }
    else{
        do Output.printString("ERROR : The entered location is
already used");
        do Sys.wait(2000);
        do set.clear_line();
    }

}
else{
    do Output.printString("ERROR : Enter a valid location");
    do Sys.wait(2000);
    do set.clear_line();
}
}

}

if(computer = 2){
    let temp = 1;
}
else{
    let temp = 2;
}

let win1 = set.win(board1,computer);
let win2 = set.win(board1,temp);
do Output.printString(" GAME OVER !!! ");

```

```

do Screen.setColor(false);
do Screen.drawRect(0,12,511,240);
do Output.moveCursor(11,20);

if(win1){
    do Output.printString("OOPS!!! You have lost the game");
    return;
}

do Output.moveCursor(11,17);

if(win2){
    do Output.printString("Congragulations!!! You have won the
game");
    return;
}

do Output.moveCursor(11,25);
do Output.printString("GAME : DRAW");

return;
}

```

3.3.4.2 Get :

The get function gets whether the player plays as 'X' or 'O' and then returns the value of what the computer should play as. This is done using a if-else loop. If the player plays as 'X', the computer plays as 'O' and *vice-versa*.

```

method int get(){
    var int computer;
    var Setup set;

```

```

    let computer = Keyboard.readInt("Do you want to play as x(1) or
o(2) : ");
    do set.clear_line();

    if(computer = 1){
        let computer = 2;
    }
    else{
        if(computer = 2){
            let computer = 1;
        }
        else{
            do Output.printString("Please enter a valid number");
            do Sys.wait(2000);
            do set.clear_line();
            let computer = get();
        }
    }

    return computer;
}

```

3.3.4.3 Move :

We initially declare the variables that we use. This is because variables can be declared in jack only at the starting. We then check if the computer has a chance of winning in either of the next move possible . If so, we place our move there. Next, we check if there is a possibility of the player winning. Is so, we give the next priority to it. Then, we check how many empty spaces are there. It specifies that there are those many possible moves. We store it in a variable and then call the function called gekko. We set the first line as “Thinking...” for the period when the computer calculates its move. Then, we delete it using the clear line function at the end of the function. We then move the cursor to (0,0). We then return the function. We give the number of empty spaces as an input

to the gecko function. This is because, we need an array of size k where k is the number of empty spaces. As it is not possible to define a variable after a sub routine declaration, we define it by creating a new sub routine.

```
method int move(Array board2,int player,int computer){
    var int temp,i,j,location,k;
    var Setup set;

    do Output.printString(" Thinking...");

    if(computer = 1){
        let temp = 2;
    }
    else{
        let temp = 1;
    }

    let i=0;
    let j=0;
    while(~(i=3)){
        let j=0;
        while(~(j=3)){
            if(set.board(board2,i,j) = 0){
                let board2 = set.set_board(board2,i,j,computer);
                if(set.win(board2,computer)){
                    let location = ((i*3)+j)+1;
                    return location;
                }
                let board2 = set.set_board(board2,i,j,0);
            }
            let j = j+1;
        }
        let i = i+1;
    }
}
```

```

let i=0;
let j=0;
while(~(i=3)){
  let j=0;
  while(~(j=3)){
    if(set.board(board2,i,j) = 0){
      let board2 = set.set_board(board2,i,j,temp);
      if(set.win(board2,temp)){
        let location = ((i*3)+j)+1;
        return location;
      }
      let board2 = set.set_board(board2,i,j,0);
    }
    let j = j+1;
  }
  let i = i+1;
}

```

```

let i = 0;
let j = 0;
let k = 0;
while(~(i=3)){
  let j=0;
  while(~(j=3)){
    if(set.board(board2,i,j) = 0){
      let k = k+1;
    }
    let j = j+1;
  }
  let i = i+1;
}

```

```

let location = gekko(board2,k,player,computer);
do set.clear_line();
do Output.moveCursor(0,0);

```

```
    return location;  
}
```

3.3.4.4 Gekko :

We initially declare all the variables and initialise an array with size k, where k is the number of empty spaces that we got from the move function. We now run 2 while loops by updating I and j as we don't have for loops in jack. We then fill in the array with the possibility values. For this purpose, we create a possibility function here also. We then find the index of the maximum value using a max variable and updating it while traversing through the array whenever the value inside the ith location of the array is greater than the current max value. We then get the location of the place where the max possibility is there. This is done by multiplying the row by 3 and adding that to the column. Then finally 1 is added to get the location. The location is then returned to the move function which is inturn returned to our run function by the move function.

```
method int gekko(Array board1,int k,int player,int computer){  
    var int location,i,j,k1,pos,max,index;  
    var Array values;  
    var Setup set;  
  
    let values = Array.new(k);  
  
    let k1 = 0;  
  
    let i=0;  
    let j=0;  
    while(~(i=3)){  
        let j=0;  
        while(~(j=3)){  
            if(set.board(board1,i,j) = 0){  
                let board1 = set.set_board(board1,i,j,player);  
                let pos = possibility(board1,player,computer);
```

```

        let values[k1] = pos;
        let k1 = k1+1;
        let board1 = set.set_board(board1,i,j,0);
    }
    let j = j+1;
}
let i = i+1;
}

let max = values[0];
let index = 0;

let i = 0;
while(~(i=k)){

    if(max < values[i]){
        let max = values[i];
        let index = i;
    }

    let i = i + 1;
}

let i = 0;
let j=0;
let k1=0;

while(~(i=3)){
    let j=0;
    while(~(j=3)){
        if(set.board(board1,i,j) = 0){
            if(k1=index){
                let location = ((i*3)+j) + 1;
                return location;
            }

```

```

        let k1 = k1+1;
    }
    let j = j+1;
}
let i = i+1;
}

return 0;
}

```

3.3.4.5 Possibility :

We first check if the board is completely filled using 2 while loops, as there is no for loop in this. We update a variable by 1 each time we find a filled location. If the filled location is equal to $9(3^2)$, we will check the winning of either the computer or the player. If the computer wins, we will return 1. If the player wins, we will return -1. If it is a draw, we will return 0. In the else loop, we will change the current player initially. Then we will run loops to set the value of the player in a different square in each iteration and calculate the possibility. We will iteratively and recursively add the possibilities of winning to a pos variable. Iteration is when the possibilities of the winning by placing in a different location for the same move. Recursion is when we call the possibility function again after updating the board to the new player in order to calculate the possibility of winning after the move is made. This is done until the board is full in each and every recursive loop and the possibility value is then returned. We also revert the board to its original value after each and every recursive call. This is where backtracking is involved. We then return the pos variable as the output of the possibility function.

```

method int possibility(Array board1,int player,int computer){
    var int i,j,k,temp,pos;
    var Setup set;

    let i = 0;

```

```

let j = 0;
let k = 0;
while(~(i=3)){
    let j=0;
    while(~(j=3)){
        if(~(set.board(board1,i,j) = 0)){
            let k = k+1;
        }
        let j = j+1;
    }
    let i = i+1;
}

if(k=9){

    if(computer = 1){
        let temp = 2;
    }
    else{
        let temp = 1;
    }

    if(set.win(board1,computer)){
        return 1;
    }
    else{
        if(set.win(board1,temp)){
            return -1;
        }
        else{
            return 0;
        }
    }
}
else{

```

```

    if(player = 1){
        let player = 2;
    }
    else{
        let player = 1;
    }

    let pos=0;

    let i = 0;
    let j = 0;
    while(~(i=3)){
        let j=0;
        while(~(j=3)){
            if(set.board(board1,i,j) = 0){
                let board1 = set.set_board(board1,i,j,player);
                let pos = pos + possibility(board1,player,computer);
                let board1 = set.set_board(board1,i,j,0);
            }
            let j = j+1;
        }
        let i = i+1;
    }

    return pos;

}
}

```

We then end the class and with this the single player mode with the computer gets over. Through this recursive backtracking algorithm, the computer decides its best move in each and every step in order to win.

3.3.5 Double File :

We define the run function inside the double class. We declare all the variables that we will be using for our purpose initially. We initialise the variables for the with the same name for the same purpose which we did earlier. We then initialise a while loop with condition ($\sim(\text{GameOver})$). This loop will run until the boolean GameOver variable becomes true. Inside that, we use a if-else loop to check whose turn it is. If the player variable is 1, it means that it is the player 'X' turn. So, we print 'Player X turn' and then ask for the location. If it is 2, it means that it is player 'O' turn. So, we print 'Player O turn' and then ask for the locations. Then, we use if-else loops the same way we did earlier to check if the location is correct and there is no value entered at that point already. Then, we update the board with the specific value and then check if the game is over. Then, we change the current player. Then, we will exit the loop. We then print 'GAME OVER!!!' as we come out of the loop. Then, we print who has won in the center of the screen. We check this using the win function. We return the run function inside the if-else loop itself. If it does not return, we will print "GAME : DRAW". Then, we will return the run function.

```
class Double{
  method void run(){
    var Array row1,row2,row3;
    var Array board1;
    var boolean Game_Over,win1,win2;
    var int player,location,row,col,i,j,k;
    var Setup set;

    let Game_Over = false;
    let player = 1;

    let row1 = Array.new(3);
    let row2 = Array.new(3);
    let row3 = Array.new(3);
    let board1 = Array.new(3);
```



```

let row1[0] = 0;
let row1[1] = 0;
let row1[2] = 0;
let row2[0] = 0;
let row2[1] = 0;
let row2[2] = 0;
let row3[0] = 0;
let row3[1] = 0;
let row3[2] = 0;
let board1[0] = row1;
let board1[1] = row2;
let board1[2] = row3;

while(~(Game_Over)){
    if(player = 1){
        let location = Keyboard.readInt("Player X turn (Enter a
number between 1 to 9) : ");
        do set.clear_line();
    }
    else{
        let location = Keyboard.readInt("Player o turn (Enter a number
between 1 to 9) : ");
        do set.clear_line();
    }

    if((location<10) & (location>0)){

        let location = location-1;
        let row = location/3;
        let col = location - (3 * (location/3));

        if(set.board(board1,row,col) = 0){

            let location = location+1;

```

```

    let board1 = set.set_board(board1,row,col,player);

    if(player = 1){
        do set.draw_x(location);
    }
    else{
        do set.draw_o(location);
    }

    let Game_Over = set.check(board1,player);

    if(player = 1){
        let player = 2;
    }
    else{
        let player = 1;
    }
}
else{
    do Output.printString("ERROR : The entered location is
already used");
    do Sys.wait(2000);
    do set.clear_line();
}

}
else{
    do Output.printString("ERROR : Enter a valid location");
    do Sys.wait(2000);
    do set.clear_line();
}
}

let win1 = set.win(board1,1);
let win2 = set.win(board1,2);

```

```

do Output.println(" GAME OVER !!! ");

do Screen.setColor(false);
do Screen.drawRect(0,12,511,240);
do Output.moveCursor(11,20);

if(win1){
    do Output.println("Player X has won the game");
    return;
}

if(win2){
    do Output.println("Player o has won the game");
    return;
}

do Output.moveCursor(11,25);
do Output.println("GAME : DRAW");
return;
}
}

```

With this, the entire code for our project Part B Tic Tac Toe Game is over.

3.4 SOURCE CODE AND DEMONSTATION :

JAVA

[Single Player](#)
[Double Player](#)

JACK

[Main.jack](#)
[Game.jack](#)
[Setup.jack](#)
[Single.jack](#)
[Double.jack](#)

VM

[Main.vm](#)
[Game.vm](#)
[Setup.vm](#)
[Single.vm](#)
[Double.vm](#)

OUTPUT

[Output Video](#)

4 FLAPPY BIRD GAME

4.1 WORKING OF GAME:

4.1.1 Basic Understanding of the Game:

In Flappy Bird, the player controls a bird character that must navigate through a side-scrolling game window with obstacles in the form of pipes. The objective is to guide the bird safely through the gaps between the pipes without colliding with them. The bird automatically moves forward, and the player's control is limited to tapping a button to make the bird flap its wings and gain upward momentum. Timing is crucial because the bird descends due to gravity when not flapping.

4.1.2 Functionalities of the Game:

User Input Handling: The game engine needs to capture user input, such as tapping the screen or pressing a button, to make the bird flap its wings. It should respond to these input events and update the game state accordingly.

Game State Management: The game engine must keep track of the current state of the game, including the position of the bird, the position and movement of the pipes, the score, and whether the game is over or still in progress. It will update and maintain this game state as the game progresses.

Graphics Rendering: The game engine needs to handle working of graphics on the screen. This involves drawing the bird, the pipes, and any other visual elements. In the Nand2Tetris framework, graphics rendering can be achieved by manipulating the pixels on the computer's screen memory.

Implementation of the Game:

- Creating classes to represent the various game objects, such as the bird, game, pipes, player, sprite, random number generator in the game.
- Defining the rules and logic of the Flappy Bird game. This includes handling collisions between the bird and the pipes, updating the bird's

position based on user input and gravity, calculating the score, and managing the game over conditions.

- Implementing User Interface to display the score and other messages.

4.2 IMPLEMENTATION:

4.2.1 Classes in Jack :

Main Class:

```
class Main {  
    function void main() {  
        var Game game;  
        var boolean run;  
        let game = Game.new(2);  
        do game.gamestart();  
        let run = true;  
        while (run) {  
            let run = game.run();  
        }  
        do game.dispose();  
        return;  
    }  
}
```

A variable game is created of the Game class. Boolean run is created which is given as the condition for while loop. Using the game instance, the run() method is called. The gamestart method is called without conditions to display the instructions. After the loop gets exited, dispose() function is called from the Game class.

Game Class:

```
field int speed;  
  
field Bird bird;  
field Pipe pipe1;  
field Pipe pipe2;  
field Pipe pipe3;  
field Score score;  
  
field char key;  
field char last_key;  
field boolean first_jump;  
field boolean exit;  
field boolean reset;  
field boolean quit;  
field int collision;
```

field int frame;

The class contains declaration of various fields representing different aspects of the game. These fields include variables for game speed, game objects (bird, pipes, score), internal states (key, last_key, first_jump, exit, reset, quit, collision), and profiling information (frames).

The **last_key** is used to keep track of the last key that was pressed by the player.

the **first_jump** field is used to indicate whether the player has performed the first jump in the game.

```
constructor Game new(int s) {  
let speed = s;  
let player = Player.new(8, 128);  
do Pipe.create(2, 50);  
let pipe1 = Pipe.new(614+((560/3)*0), 128);  
let pipe2 = Pipe.new(614+((560/3)*1), 128);  
let pipe3 = Pipe.new(614+((560/3)*2), 128);  
let score = Score.new();  
return this;  
}
```

The constructor **Game new(int s)** initializes the Game object and sets the delay (game speed) based on the input parameter **s**. It also allocates and initializes the game objects (player, pipes, score).

Three **Pipe** objects with different x-coordinates, placing them at specific positions on the screen is coded.

```
method void gamestart(){  
do Output.moveCursor(0,0);  
do Output.printString("Use < and > arrows to increase or  
decrease flapping of bird");  
do Output.println();  
do Output.printString("Press r to reset");  
do Output.println();  
do Output.printString("Press q to quit");
```



```

        do Output.println();
        do Output.printString("Press spacebar to play");
return;
}

method void inputs() {
let key = Keyboard.keyPressed();
//
if (key = 81) {
    let exit = true;
    let quit = true;
}
if (key = 82) { // R
    let exit = true;
    let reset = true;
}
if ((key = 32) & (~(last_key=32))) { // space
    do bird.jump();
    if (first_jump) {

        let first_jump = false;

        do LCGRandom.setSeed(frame);
        do Output.moveCursor(0,0);
        do Output.printString("                ");
        do Output.moveCursor(1,0);
        do Output.printString("                ");
        do Output.moveCursor(2,0);
        do Output.printString("                ");
        do Output.moveCursor(3,0);
        do Output.printString("                ");
        do Output.moveCursor(4,0);
        do Output.printString("                ");
        do Output.moveCursor(5,0);
        do Output.printString("                ");

```

```

do Output.moveCursor(10, 20);
do Output.printString("                ");

    }
    } else {
        if (((key = 130) & ~(last_key=130))) & (speed > 0)) { let speed =
speed - 1; }
        if ((key = 132) & ~(last_key=132))) { let speed = speed + 1; } //
right arrow
    }
    let last_key = key;
    return;
}

```

The **inputs()** method handles keyboard input and updates the game state accordingly. It captures the pressed key using **Keyboard.keyPressed()** and performs different actions based on the pressed key.

Pressing Q (ASCII value is 81) will set the **exit** and **quit** variables to true, indicating the player wants to quit the game. Pressing R (ASCII value is 82) will set the **exit** and **reset** variables to true, indicating the player wants to restart the game. It also checks that if it is the first jump in the game. If it is, the code inside the condition is executed, which includes updating the game state, replacing the title and instruction messages, and preparing the game for play.

Pressing the spacebar (ASCII value is 32) will make the player character jump by calling **bird.jump()**.

To execute the jump function it checks if the key pressed is 32 (space bar) and the previously pressed key is not 32. This ensures that the action inside the condition is executed only once when the spacebar is initially pressed, rather than continuously while it is held down.

If it is the first jump, the Boolean variable **first_jump** is set to false and starts the game.

LCGRandom.setSeed(frame) sets the seed value for the linear congruential generator (LCG) random number generator. The seed value determines the starting point of the sequence of random numbers

generated by the LCG. Here, it uses the frame variable as the seed, which is incremented each frame. This ensures that the sequence of random numbers generated by the game is different each time the game is played. In else branching we give debugging controls, where if the left arrow key is pressed and the last_key is not left arrow and the delay value is greater than zero, then delay value is decremented by 1.

To update the last pressed key, value of key is given to last_key.

```
method void handle_pipe(int collision) {
```

```
if (collision =2) { // Collision
```

```
let exit = true;}
```

```
else{
```

```
if (collision =1) { // Passing by
```

```
do score.increment();
```

```
    } }
```

```
    return;
```

```
}
```

Nextly, function for the pipe is defined.

method void handle_pipe(int collision) defines the method that takes an integer parameter called collision. The parameter represents the type of interaction with the pipe. It checks if the value of collision is equal to 2, which indicates a collision between the player and the pipe. If the condition is met the exit variable is set to else if collision is 1 it indicates that the player has successfully passed by a pipe without collision leading to increment in the score.

```
method void refresh() {
```

```
do bird.update();
```

```
if ((~first_jump)) {
```

```
do pipe1.update();
```

```
do pipe2.update();
```

```
do pipe3.update();
```

```
do score.show();
```

```

do handle_pipe(pipe1.gameover(bird.xP(),bird.yP()));
do handle_pipe(pipe2.gameover(bird.xP(),bird.yP()));
do handle_pipe(pipe3.gameover(bird.xP(),bird.yP()));    }
return;
}

```

The method refresh is to update the pipe objects and score. **do bird.update()** calls the **update** method on the **player** object. If it is not the first jump, meaning the game has started already, the sprite is updated. The three pipes are updated by the update() function. The show method is called on the score object.

The **handle_pipe** method and passes the result of **pipe1.gameover(player.x(),player.y())** as an argument. This expression checks if the player is inside the **pipe1** object's boundaries (collision detection) and returns a value indicating the type of interaction between the player and the pipe. Similarly, collision is detected for all 3 pipes.

```

method void framecount() {
    let frame = frame + 1;
    if (frame < 0) {
        let frame = 0;
    }
    return;
}

```

The **frame** variable in the code represents the number of frames that have elapsed in the game. The purpose of tracking the number of frames is typically to measure the passage of time or to synchronize game events.

In the framecount method, the frame variable is added by 1 and checked if it is lesser than 0. If the condition is true, it sets it to 0 to avoid integer overflow and checks if it the first jump

```

method boolean run() {

```

```

    let key = 0;
    let last_key = 0;
    let first_jump = true;
    let exit = false;
    let reset = false;
    let quit = false;
    let frames1 = 0;

```

```

do Output.moveCursor(10, 20);
do Output.printString(" G E T   R E A D Y ");

```

```

while (~exit) {
    do inputs();
    do refresh();

    do framecount();
    do Sys.wait(speed);
}
if ((~quit)) {
    do Output.moveCursor(10, 20);
do Output.printString(" G A M E O V E R ");
}

while ((~reset)) {
    let key = Keyboard.keyPressed();
    if (key = 81) {
        return false;
    }
    if (key = 82) {
        let reset = true;
    }
}
do reset();
return true;
}

```

The run method initialises key, last_key, first_jump, exit, rest, qit and frame.

When the game has not been exited, the defined method,

- **inputs()** is called to handle keyboard input and update relevant variables.
- **refresh()** is called to update the game logic and render the objects.
- **render_vitals()** is called to display profiling and timing information.
- **Sys.wait(delay)** pauses the execution for a certain duration specified by the **delay** variable, controlling the game speed.

Once, this while loop is exited, meaning the game is over, it checks if the user has quit or reset by checking the Boolean variables quit and reset. If it is not quitting,

- Displays “Game Over” message

The code enters another loop controlled by the reset variable. Within this loop, it waits for keyboard input to either quit the game by pressing “Q”

or reset the game by pressing “R”. If “Q” is pressed, the method returns false, indicating the game should exit. If “R” is pressed, the reset flag is set to true.

Finally, the reset() method is called to reset the game state, and the method returns true, indicating that the game should continue running.

```
method void reset() {  
do Screen.clearScreen();  
do bird.reset(128, 128);  
do pipe1.reset(614+((560/3)*0), 128);  
do pipe2.reset(614+((560/3)*1), 128);  
do pipe3.reset(614+((560/3)*2), 128);  
do score.reset();  
return;  
}
```

In reset method, the player is reset to position (128, 128) and the pipes are reset to the initial positions. The Screen is cleared and the score is reset.

```
method void dispose() {  
do player.dispose();  
do pipe1.dispose();  
do pipe2.dispose();  
do pipe3.dispose();  
do score.dispose();  
do Memory.deAlloc(this);  
return;  
}
```

The dispose function deallocates memory for the objects as java has no garbage collection and memory has to be disposed explicitly. As “this” contains the base address of the object, it is also given to deAlloc function.

Pipe Class:

```
static int xvelocity;  
static int fixedgap;  
field int xpositionition;  
field int ypositionition;  
field int time;  
field boolean pass;
```

The velocity by which the Sprite moves is xvelocity and the fixedgap represents the gap between the pipes.

```
function void create (int x, int y) {  
    let xvelocity = x;  
    let fixedgap = y;  
return;  
}
```

This function sets up the global pipe options by assigning the provided **x** and **y** values to the static variables xVel and yGap.

```
constructor Pipe new(int x, int y) {  
do setpipe(x, y);  
return this;  
}  
method void dispose() {  
do Memory.deAlloc(this);  
return;  
}
```

The constructor is called when a new Pipe object is created. It takes an **x** and **y** parameter to set the initial position of the pipe.

The setpipe method is then called to initialize the pipe with the provided position. dispose() function is defined to deallocate memory.

```
method void draw() {  
    // Draw next frame  
    do Screen.setColor(true);  
    // head bottom  
    do Screen.drawRect(Math.min(Math.max(xpositionition-
```

```

2,0),511), ypositionition+30,
Math.min(Math.max(xpositionition+44,0),511), ypositionition+31); //
top side
    do Screen.drawRectangle(Math.min(Math.max(xpositionition-
2,0),511), ypositionition+54,
Math.min(Math.max(xpositionition+44,0),511), ypositionition+55); //
bottom side
    do Screen.drawRectangle(Math.min(Math.max(xpositionition-
2,0),511), ypositionition+30, Math.min(Math.max(xpositionition-
1,0),511), ypositionition+54); // left side
    do
Screen.drawRectangle(Math.min(Math.max(xpositionition+44,0),511),
ypositionition+30, Math.min(Math.max(xpositionition+45,0),511),
ypositionition+55); // right side

    // body bottom
    do
Screen.drawRectangle(Math.min(Math.max(xpositionition,0),511),
ypositionition+54, Math.min(Math.max(xpositionition+1,0),511), 255);
// left wall
    do
Screen.drawRectangle(Math.min(Math.max(xpositionition+42,0),511),
ypositionition+54, Math.min(Math.max(xpositionition+43,0),511), 255);
// right wall

    // head top
    do Screen.drawRectangle(Math.min(Math.max(xpositionition-
2,0),511), ypositionition+(-fixedgap),
Math.min(Math.max(xpositionition+45,0),511), ypositionition+(-
(fixedgap-1))); // top
    do Screen.drawRectangle(Math.min(Math.max(xpositionition-
2,0),511), ypositionition+(-(fixedgap+25)),
Math.min(Math.max(xpositionition+45,0),511), ypositionition+(-
(fixedgap+24))); // bottom
    do Screen.drawRectangle(Math.min(Math.max(xpositionition-

```



```

2,0),511), ypositionition+(-(fixedgap+23)),
Math.min(Math.max(xpositionition-1,0),511), ypositionition+(-
fixedgap));    // left
do
Screen.drawRectangle(Math.min(Math.max(xpositionition+44,0),511),
ypositionition+(-(fixedgap+25)),
Math.min(Math.max(xpositionition+45,0),511), ypositionition+(-
fixedgap));    // right
    // body top
do
Screen.drawRectangle(Math.min(Math.max(xpositionition,0),511), 0,
Math.min(Math.max(xpositionition+1,0),511), ypositionition+(-
(fixedgap+25)));    // left
do
Screen.drawRectangle(Math.min(Math.max(xpositionition+42,0),511),
0, Math.min(Math.max(xpositionition+43,0),511), ypositionition+(-
(fixedgap+25))); // right

do Screen.setColor(false);
    // bottom
do
Screen.drawRectangle(Math.min(Math.max(xpositionition,0),511),
ypositionition+32,
Math.min(Math.max(xpositionition+(1+xvelocity),0),511),
ypositionition+53);    // clside
do
Screen.drawRectangle(Math.min(Math.max(xpositionition+46,0),511),
ypositionition+30,
Math.min(Math.max(xpositionition+(48+xvelocity),0),511),
ypositionition+55); // crside
do
Screen.drawRectangle(Math.min(Math.max(xpositionition+2,0),511),
ypositionition+56,
Math.min(Math.max(xpositionition+(4+xvelocity),0),511), 255);    //
clwall

```

```

    do
Screen.drawRectangle(Math.min(Math.max(xpositionition+44,0),511),
ypositionition+56,
Math.min(Math.max(xpositionition+(46+xvelocity),0),511), 255);    //
crwall
    // top
    do
Screen.drawRectangle(Math.min(Math.max(xpositionition,0),511),
ypositionition+(-(fixedgap+23)),
Math.min(Math.max(xpositionition+1,0),511), ypositionition+(-
(fixedgap+1)));    // clside
    do
Screen.drawRectangle(Math.min(Math.max(xpositionition+46,0),511),
ypositionition+(-(fixedgap+25)),
Math.min(Math.max(xpositionition+47,0),511), 1+(ypositionition+(-
fixedgap))); // crside
    do
Screen.drawRectangle(Math.min(Math.max(xpositionition+2,0),511), 0,
Math.min(Math.max(xpositionition+3,0),511), ypositionition+(-
(fixedgap+26)));    // clwall
    do
Screen.drawRectangle(Math.min(Math.max(xpositionition+44,0),511),
0, Math.min(Math.max(xpositionition+45,0),511), ypositionition+(-
(fixedgap+25)));    // crwall

return;
}

```

The draw() method is responsible for rendering the pipe on the screen. It uses a combination of rectangles to create the visual representation of the pipe.

```

do Screen.drawRectangle(Math.min(Math.max(xpositionition-
2,0),511), ypositionition+30,
Math.min(Math.max(xpositionition+44,0),511), ypositionition+31);
// top side

```

do Screen.drawRectangle indicates the start of the function call to **Screen.drawRectangle()**. The keyword **do** suggests that it's executing a command or function.

Math.min(Math.max(xposition-2,0),511) calculates the x-coordinate for the left side of the rectangle to be drawn. It subtracts 2 from the value of **xposition** and ensures that the result is not less than 0 or greater than 511. The **Math.max()** function ensures that the value is not less than 0, and the **Math.min()** function ensures that the value is not greater than 511.

yposition+30 calculates the y-coordinate for the top side of the rectangle to be drawn. It adds 30 to the value of **yposition**.

yposition+30 represents the y-coordinate of the top-left corner of the rectangle. It adds 30 to the **yposition** variable, which offsets the rectangle vertically from the **yposition** position.

Math.min(Math.max(xposition+44,0),511) calculates the x-coordinate for the right side of the rectangle to be drawn. It adds 44 to the value of **xposition** and ensures that the result is not less than 0 or greater than 511.

yposition+31 represents the y-coordinate of the bottom-right corner of the rectangle. It adds 31 to the **yposition** variable, representing the height of the rectangle. The resulting rectangle has a width of 47 pixels (44+2+1) and a height of 2 pixels.

Screen.drawRectangle(Math.min(Math.max(xposition-2,0),511), yposition+54, Math.min(Math.max(xposition+44,0),511), yposition+55) sets the bottom sides of the bottom part of the pipe's head.

The first parameter (**Math.min(Math.max(xposition-2,0),511)**) represents the x-coordinate of the top-left corner of the rectangle. It is calculated by subtracting 2 from the **xposition** value, ensuring that the resulting value is not less than 0 (using **Math.max(xposition-2, 0)**), and then limiting it to a maximum value of 511 (using **Math.min(..., 511)**). This ensures that the rectangle is not drawn outside the left boundary of the screen.

The second parameter **yposition+54** represents the y-coordinate of the top-left corner of the rectangle. It is obtained by adding 54 to the

ypositionition value. This positions the top side of the rectangle at a distance of 54 pixels below the **ypositionition** position.

The third parameter (**Math.min(Math.max(xpositionition+44,0),511)**) represents the x-coordinate of the bottom-right corner of the rectangle. It is calculated by adding 44 to the **xposition** value, ensuring that the resulting value is not less than 0 (using **Math.max(xpositionition+44, 0)**), and then limiting it to a maximum value of 511 (using **Math.min(..., 511)**). This ensures that the rectangle is not drawn outside the right boundary of the screen.

The fourth parameter **ypositionition+55** represents the y-coordinate of the bottom-right corner of the rectangle. It is obtained by adding 55.

The **Screen.drawRectangle()** function is being called to draw a rectangle on the screen. The rectangle's top-left corner is determined by the **xpositionition** and **ypositionition** variables, and its size is defined as a width of 44 pixels and a height of 1 pixel. The function ensures that the rectangle is drawn within the boundaries of the screen, with the x-coordinate clamped between 0 and 511 and the y-coordinate adjusted by adding 54 and 55 to the original **ypositionition** value. Similarly, the constraints are chosen for all borders for pipes and drawn in rest of the code.

```
method void updpipeline() {
    let time = time + 1;
    if (time = 5) {
        let time = 0;
    } else {
        let xpositionition = xpositionition - xvelocity;

        if (xpositionition < (-50)) {
            let pass = true;
            let xpositionition = 514;
            let ypositionition = LCGRandom.randRange(80,194);
        }
        if (xpositionition > 614) {
            let ypositionition = LCGRandom.randRange(80,194);
        }
    }
    return;
}
```

The variable **time** is incremented by 1 each time the **updpipeline()** method

is called. The next conditional statement checks if it is equal to 6. In that case, it is reset to 0. In the else part, **xpositionition** is updated as the variable gets decremented by the **xvelocity**. The next conditional statement checks The variable **time** is incremented by 1 each time. Similarly, the next conditional statement checks if the **xpositionition** value has exceeded the right boundary, represented by 614. If so, it generates a random value for **ypositionition** using the **LCGRandom.randRange()** function.

```
method void update() {
    do updpipeline();
    do draw();
    return;
}
```

```
method int x() {
    return xposition;
}
```

```
method int y() {
    return yposition;
}
```

The update method calls the previously defined methods. The **x()** and **y()** methods are accessors that returns the **xpositionition** and **ypositionition**.

```
method int gameover(int x, int y) {
    if ((x>(xpositionition-32))&(x<(xpositionition+40))) { // inside pipe
        region (left-right)
        if ((y>ypositionition)|(y<(ypositionition-(fixedgap+6)))) { // inside
            pipe boundary (bottom-top)
            return 2; // xpositionitionlision
        }
        if (pass) {
            let pass = false;
            return 1; // safe-inside-region first time
        }
        else {return 0;} // safe-inside-region other time
    }
    return 0; // not in pipe region
}
```

The above code is defined to check for collision, which means that the player is inside the pipe's boundaries. This conditional statement checks if the x-coordinate (**x**) of the player is within the range of **xpositionition**

- **32** and **xpositionition + 40**. This range defines the left and right boundaries of the "pipe" region. If the condition is true, the execution continues inside the **if** block.

Within the previous **if** block, this conditional statement checks if the y-coordinate (**y**) of the player is either greater than **yposition** (the top boundary of the pipe) or less than **ypositionition - (fixedgap + 6)** (the bottom boundary of the pipe).

If the player's coordinates are inside the pipe region and outside the pipe boundaries, the method returns the value 2, indicating a collision has occurred.

If the player's coordinates are inside the pipe region and inside the pipe boundaries, the code checks the value of the **pass** variable. If **pass** is **true**, it sets **pass** to **false** and returns the value 1, indicating that the player is inside the pipe region but in the first time of being in that region.

If the player's coordinates are inside the pipe region and inside the pipe boundaries, but it is not the first time, the method returns the value 0, indicating that the player is inside the pipe region in a subsequent time.

If the player's coordinates are inside the pipe region and inside the pipe boundaries, but it is not the first time, the method returns the value 0, indicating that the player is inside the pipe region in a subsequent time.

In summary, this code checks if the player's coordinates are inside a defined pipe region and returns different values based on the specific conditions met. It returns 2 if a collision occurs, 1 if the player is in the pipe region for the first time, 0 if the player is in the pipe region for subsequent times, and 0 if the player is not in the pipe region at all.

```
method void setpipe(int x, int y){  
    let xpositionition = x;  
    let ypositionition = y;  
    let time = -1;  
    let pass = true;  
    return;  
}
```

Class Mod :

```
function int mod(int a, int b) {  
    var int q;  
    var int r;  
    let q = (a / b);
```

```

    let r = a - (b * q);
    return r;
}

```

The **mod** class is declared, encapsulating the **mod** method. The **mod** method takes two integer parameters **a** and **b**. Two integer variables **q** and **r** are declared to store the quotient and remainder, respectively.

The method calculates the remainder of dividing **a** by **b**. It first calculates the quotient by dividing **a** by **b** and assigns it to the variable **q**. Then, it computes the remainder by subtracting the product of **b** and **q** from **a** and assigns it to the variable **r**. The method returns the value of **r**, which represents the remainder of **a** divided by **b**.

Class Bird:

```

field int row;
field int col;
field int velocity;
field int index;
field int fly;
field int time;
field int up;
field int down;

```

The variables take the same instances as declared earlier in Game class.

```

Constructor Bird new(int x, int y) {
    do reset(x, y);
    return this;
}

method void dispose() {
    do Memory.deAlloc(this);
    return;
}

method void jump() {
    let velocity = -4;
    if (~gravity) {
        let gravity = true;
    }
    return;
}

```

The vertical velocity is set to -4 giving an upward motion. When the gravity variable is not true, it sets it to become true.

```

method void updbird(){
let time=time+1;
if (time = 5) {
    let time = 0;
    let fly = fly + 1;
    if (velocity < 6){
        let velocity=velocity+1;
    }
    if (fly = 4){
        let fly = 0;
    }
}

if (down = 0){
    if( velocity > 0) {
        if(row < (221-velocity)){
            let row = row +velocity;
            let index = index + (velocity*32);
        }
    } else {
        if (row > -velocity){
            let row=row+velocity;
            let index= index+ (velocity*32);
        } else {
            let velocity = 0;
        }
    }
} else {
    if (up = 0) {
        if ((time = 1)) {
            let row = row - 1;
            let index = index - 32;
        }
        if (row < 118)
        {let up = 1;
        }
    }
    } else {
        if ((time = 1)) {
            let row = row + 1;
            let index = index + 32;
            if (row > 138) {let up = 0;}}
        }
    }
}
return;
}

```


In the `updbird` method, it updates the player's state and position based on game logic. Increments the **time** variable by 1. If **time** equals 5, resets **time** to 0, increments **fly** by 1, and increases **velocity** by 1 (up to a maximum of 6). If **fly** equals 4, it resets **fly** to 0.

If **gravity** is true, it applies gravity to the player's position and updates the **yposition**. If the velocity is greater than 0, the code block inside this condition is executed:

If the **yposition** is less than `221 - velocity`, the **yposition** and **position** variables are incremented by `velocity * 32`. If the **velocity** is not greater than 0, the code block inside this condition is executed, that if the **yposition** is greater than `-velocity`, the **yposition** and **position** variables are incremented by `velocity * 32`. If the **yposition** is not greater than `-velocity`, the **velocity** is set to 0, indicating that the player has reached the top or bottom of the game area and their vertical velocity should be reset.

```
method void render() {  
    if (fly = 0) {  
        do Flappy.flappy1(index); }  
    else {  
        if (fly = 1) {  
            do Flappy.flappy2(index); }  
        else {  
            if (fly = 2) {  
                do Flappy.flappy3(index); }  
            else {  
                if (fly = 3) {  
                    do Flappy.flappy2(index); }  
                }  
            }  
        }  
    }  
    return;  
}
```

The **render** method is defined, likely responsible for drawing the Flappy Bird character sprite on the screen. The code uses an if-else conditions to determine which sprite to draw based on the current **fly** value. If **fly** is equal to 0, it means the Flappy Bird character should be drawn with the sprite **draw_flappy1** at the specified **position**. This sprite represents one appearance of the character. If **fly** is not equal to 0, the code moves to the next level of if-else conditions. If **fly** is equal to 1, the Flappy Bird character should be drawn with the sprite **draw_flappy2** at the specified **position**. This sprite represents another appearance of the character.

```

method void update() {
    do time();
    do render();
    return;
}
method int x() {
    return xposition;
}
method int y() {
    return yposition;
}
method void reset(int x, int y) {
    let xposition = (x*16);
    let yposition = y;
    let velocity = 0;
    let position = (x+(32*yposition));
    let fly = 0;
    let time = 0;
    let gravity = false;
    let animation_direction_up = true;
    return;
}

```

An update method and reset method is defined alongwith, two accessor methods m() and n().

Score Class:

```

field int total;
field int v1;
field int v2;
field int v3;

constructor Score new() {
    do reset();
    return this;
}

method void dispose() {
    do Memory.deAlloc(this);
    return;
}

```

The **Score** class is declared with four fields: **total**, **v1**, **v2**, and **v3**. These fields store the score values and individual digits of the score. The **Score** class has a constructor **new()** that initializes a new instance of the **Score** class by calling the **reset()** method. The **Score** class has a **dispose()** method that deallocates memory.

```

method void render(int x, int v) {
    if (v = 0) {do Flappy.draw_digit0(x+512);} else {
    if (v = 1) {do Flappy.draw_digit1(x+512);} else {
    if (v = 2) {do Flappy.draw_digit2(x+512);} else {

```

```

    if (v = 3) {do Flappy.draw_digit3(x+512);} else {
    if (v = 4) {do Flappy.draw_digit4(x+512);} else {
    if (v = 5) {do Flappy.draw_digit5(x+512);} else {
    if (v = 6) {do Flappy.draw_digit6(x+512);} else {
    if (v = 7) {do Flappy.draw_digit7(x+512);} else {
    if (v = 8) {do Flappy.draw_digit8(x+512);} else {
    if (v = 9) {do Flappy.draw_digit9(x+512);} else {
    }}}}}}}}}
    return;
}

```

The **render** method maps the individual digits (0 to 9) to their respective digit image sprites using the **Flappy** class. The method takes two parameters: **x** for the horizontal position on the screen and **v** for the value of the digit. Depending on the value of **v**, the corresponding digit image sprite is drawn at the specified horizontal position on the screen.

```

method void show() {
    if (total > 99) {
        do render(16,v1);
        do render(14,v2);
        do render(12,v3);
    } else {
        if (total > 9) {
            do render(16,v1);
            do render(14,v2);
        }
        else {
            if (total > 0) {do render(15,v1);}
        }
    }
    return;
}

```

If **total** is greater than 99, it means the score is a three-digit number, and the digits are rendered individually at specific horizontal positions using the **render** method.

If **total** is between 10 and 99, it means the score is a two-digit number. In this case, the first and second digits are rendered at specific positions using the **render** method.

If **total** is between 1 and 9, it means the score is a single-digit number, and only the first digit is rendered.

```

method void increment() {
    let total = total + 1;
    let v1 = v1 + 1;
}

```

```

    if (v1 > 9) {
        let v1 = 0;
        let v2 = v2 + 1;
        if (v2 > 9) {
            let v2 = 0;
            let v3 = v3 + 1;
        }
    }
    return;
}

```

The **increment** method is responsible for incrementing the score and updating the individual digits accordingly. The **total** field is incremented by 1. The **v1** field (the ones digit) is incremented by 1. If **v1** becomes greater than 9 (indicating a carryover to the tens digit), **v1** is reset to 0. The **v2** field (the tens digit) is incremented by 1. If **v2** becomes greater than 9 (indicating a carryover to the hundreds digit), **v2** is reset to 0. The **v3** field (the hundreds digit) is incremented by 1.

```

method int value() {
    return total;
}

method void reset() {
    let v1 = 0;
    let v2 = 0;
    let v3 = 0;
    let total = 0;
    return;
}

```

The value method returns the total score and reset method resets all three values and total to zero.

LCGRandom Class:

```

class LCGRandom {
    static int seed;
    static int A;
    static int M;
    static int Q;
    static int R;

    function void setSeed(int newSeed) {
        let seed = newSeed;
        if(seed=0) {
            let seed=1;
        }
        let A=219;
        let M=32749;
        let Q=M/A;
    }
}

```

```

    let R=Mod.mod(M,A);
    return;
}

/* returns a random int in range 0..(M-1) inclusive */
function int rand() {
    var int test;
    let test=(A*(Mod.mod(seed,Q)))-(R*(seed/Q));
    if(test<0) {
        let seed=test+M;
    }
    else {
        let seed=test;
    }
    return seed;
}

/* returns a random int in range low..high inclusive */
function int randRange(int low, int high) {
    var int scale;
    let scale = (M / (high - low + 1));
    return (LCGRandom.rand() / scale) + low;
}
}

```

seed: An integer variable that represents the current seed value for the random number generator.

A, M, Q, R: Integer variables used as constants in the LCG algorithm.

setSeed(int newSeed) Function:

This function is used to set the seed value of the random number generator.

It takes an integer parameter **newSeed** representing the new seed value. Inside the function, the local variable **seed** is assigned the value of **newSeed**.

If **seed** equals 0, the local variable **seed** is reassigned the value 1.

The constants **A, M, Q,** and **R** are assigned their respective values for the LCG algorithm.

The function ends with a **return;** statement.

rand() function generates a random integer using the LCG algorithm.

It doesn't take any parameters.

Inside the function, a local variable **test** is declared to store an intermediate value.

The intermediate value is calculated based on the LCG algorithm formula: **test = (A * (seed % Q)) - (R * (seed / Q)).**

If **test** is less than 0, the local variable **seed** is updated to **test + M**.

Otherwise, the local variable **seed** is updated to **test**.

The function ends with a **return seed;** statement.

randRange(int low, int high) function generates a random integer within a given range.

It takes two integer parameters **low** and **high**, representing the lower and

upper bounds of the range.
 Inside the function, a local variable **scale** is declared to calculate the scaling factor for the range.
 The scaling factor is calculated as $M / (\text{high} - \text{low} + 1)$.
 The function then calls the **rand()** function to generate a random integer, and divides it by the scaling factor.
 Finally, the function adds the lower bound **low** to the result and returns it as the final random integer.

Flappy Class:

The Flappy class has the bitmapped codes for drawing the skin/states of the bird as well as the score board using the built in `memory.poke()` function.

Bitmap

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1																
2																
3																
4																
5																
6																
7																
8																
9																
10																
11																
12																
13																
14																
15																
16																

Rotate right

Vertical Mirror



Figure 12 : Sprite and numbers

4.3 CODE IN JACK :

```
class Main {  
    function void main() {  
        var Game game;  
        var boolean run;  
        let game = Game.new(2);  
        do game.gamestart();  
        let run = true;  
        while (run) {  
            let run = game.run();  
        }  
        do game.dispose();  
        return;  
    }  
}
```

```
class Game {  
  
    field int speed;  
  
    field Bird bird;  
    field Pipe pipe1;  
    field Pipe pipe2;  
    field Pipe pipe3;  
    field Score score;  
  
    field char key;  
    field char last_key;  
    field boolean first_jump;  
    field boolean exit;  
    field boolean reset;  
    field boolean quit;
```

```
field int collision;
```

```
field int frames1;
```

```
constructor Game new(int d) {  
    let speed = d;
```

```
    let bird = Bird.new(8, 128);
```

```
    do Pipe.create(2, 50);
```

```
    let pipe1 = Pipe.new(614+((560/3)*0), 128);
```

```
    let pipe2 = Pipe.new(614+((560/3)*1), 128);
```

```
    let pipe3 = Pipe.new(614+((560/3)*2), 128);
```

```
    let score = Score.new();
```

```
    return this;
```

```
}
```

```
method void gamestart(){
```

```
    do Output.moveCursor(0,0);
```

```
        do Output.printString("Use < and > arrows to increase or  
decrease flapping of bird");
```

```
        do Output.println();
```

```
        do Output.printString("PRESS R TO RESET");
```

```
        do Output.println();
```

```
        do Output.printString("PRESS Q TO QUIT");
```

```
        do Output.println();
```

```
        do Output.printString("PRESS SPACEBAR TO PLAY");
```

```
return;
```

```
}
```

```
method void dispose() {
```

```
    // dispose of (deallocate) objects
```

```
    do bird.dispose();
```



```

do pipe1.dispose();
do pipe2.dispose();
do pipe3.dispose();
do score.dispose();
do Memory.deAlloc(this);
return;
}

```

```

method void inputs() {
    let key = Keyboard.keyPressed();

    if (key = 81) {
        let exit = true;
        let quit = true;
    }
    if (key = 82) {
        let exit = true;
        let reset = true;
    }
    if ((key = 32) & (~(last_key=32))) {
        do bird.jump();
        if (first_jump) {

            let first_jump = false;

            do LCGRandom.setSeed(frames1);
            do Output.moveCursor(0,0);
            do Output.printString("                    ");
            do Output.moveCursor(1,0);
            do Output.printString("                    ");
            do Output.moveCursor(2,0);
            do Output.printString("                    ");
            do Output.moveCursor(3,0);
            do Output.printString("                    ");

```

```

do Output.moveCursor(4,0);
do Output.printString("                ");
do Output.moveCursor(5,0);
do Output.printString("                ");
do Output.moveCursor(10, 20);
do Output.printString("                ");

    }
} else {

    if (((key = 130) & ~(last_key=130))) & (speed > 0)) { let speed =
speed - 1; }

    if ((key = 132) & ~(last_key=132))) { let speed = speed + 1; } //
right arrow
    }

    let last_key = key;
    return;
}

method void handle_pipe(int collision) {
    if (collision=2) { // Collision
        let exit = true;
    } else {
        if (collision=1) { // Passing by
            do score.increment();
        }
    }
    return;
}

```

```

method void refresh() {

    do bird.update();

    if ((~first_jump)) {

        do pipe1.update();
        do pipe2.update();
        do pipe3.update();

        do score.show();

        do handle_pipe(pipe1.gameover(bird.xP(),bird.yP()));
        do handle_pipe(pipe2.gameover(bird.xP(),bird.yP()));
        do handle_pipe(pipe3.gameover(bird.xP(),bird.yP()));

    }
    return;
}

```

```

method void framecount() {
    let frames1 = frames1 + 1;
    if (frames1 < 0) {
        let frames1 = 0;
    }
    return;
}

```

```

method boolean run() {

    let key = 0;
    let last_key = 0;
    let first_jump = true;

```

```

let exit = false;
let reset = false;
let quit = false;
let frames1 = 0;

do Output.moveCursor(10, 20);
do Output.printString(" G E T   R E A D Y ");

while (~exit) {

    do inputs();
    do refresh();

    do framecount();
    do Sys.wait(speed);
}
if ((~quit)) {
    do Output.moveCursor(10, 20);
do Output.printString(" G A M E   O V E R ");
}

while ((~reset)) {
    let key = Keyboard.keyPressed();
    if (key = 81) {
        return false;
    }
    if (key = 82) {
        let reset = true;
    }
}
do reset();
return true;
}

```

```

method void reset() {
    do Screen.clearScreen();
    do bird.setbird(8, 128);
    do pipe1.setpipe(614+((560/3)*0), 128);
    do pipe2.setpipe(614+((560/3)*1), 128);
    do pipe3.setpipe(614+((560/3)*2), 128);
    do score.reset();
    return;
}
}

```

```

class Pipe {
    static int xvelocity;
    static int fixedgap;

    field int xposition;
    field int yposition;
    field int time;
    field boolean pass;
}

```

```

function void create(int x, int y) {
    let xvelocity = x;
    let fixedgap = y;
    return;
}

constructor Pipe new(int x, int y) {
    do setpipe(x,y);
    return this;
}

```

```

method void dispose() {
    do Memory.deAlloc(this);
    return;
}

```

```

method void draw(){

```

```

    do Screen.setColor(true);
    // head bottom
    do Screen.drawRect(Math.min(Math.max(xposition-2,0),511),
yposition+30, Math.min(Math.max(xposition+44,0),511),
yposition+31);
    do Screen.drawRect(Math.min(Math.max(xposition-2,0),511),
yposition+54, Math.min(Math.max(xposition+44,0),511),
yposition+55);
    do Screen.drawRect(Math.min(Math.max(xposition-2,0),511),
yposition+30, Math.min(Math.max(xposition-1,0),511), yposition+54);
    do
Screen.drawRect(Math.min(Math.max(xposition+44,0),511),
yposition+30, Math.min(Math.max(xposition+45,0),511),
yposition+55);

    do Screen.drawRect(Math.min(Math.max(xposition,0),511),
yposition+54, Math.min(Math.max(xposition+1,0),511), 255);
    do
Screen.drawRect(Math.min(Math.max(xposition+42,0),511),
yposition+54, Math.min(Math.max(xposition+43,0),511), 255);

    do Screen.drawRect(Math.min(Math.max(xposition-2,0),511),
yposition+(-fixedgap), Math.min(Math.max(xposition+45,0),511),
yposition+(-(fixedgap-1)));
    do Screen.drawRect(Math.min(Math.max(xposition-2,0),511),
yposition+(-(fixedgap+25)), Math.min(Math.max(xposition+45,0),511),

```

```

yposition+(-(fixedgap+24)));
    do Screen.drawRect(Math.min(Math.max(xposition-2,0),511),
yposition+(-(fixedgap+23)), Math.min(Math.max(xposition-1,0),511),
yposition+(-fixedgap));
    do
Screen.drawRect(Math.min(Math.max(xposition+44,0),511),
yposition+(-(fixedgap+25)), Math.min(Math.max(xposition+45,0),511),
yposition+(-fixedgap));

    do Screen.drawRect(Math.min(Math.max(xposition,0),511), 0,
Math.min(Math.max(xposition+1,0),511), yposition+(-(fixedgap+25)));
    do
Screen.drawRect(Math.min(Math.max(xposition+42,0),511), 0,
Math.min(Math.max(xposition+43,0),511), yposition+(-(fixedgap+25)));

    do Screen.setColor(false);

    do Screen.drawRect(Math.min(Math.max(xposition,0),511),
yposition+32, Math.min(Math.max(xposition+(1+xvelocity),0),511),
yposition+53);
    do
Screen.drawRect(Math.min(Math.max(xposition+46,0),511),
yposition+30, Math.min(Math.max(xposition+(48+xvelocity),0),511),
yposition+55);
    do Screen.drawRect(Math.min(Math.max(xposition+2,0),511),
yposition+56, Math.min(Math.max(xposition+(4+xvelocity),0),511),
255);
    do
Screen.drawRect(Math.min(Math.max(xposition+44,0),511),
yposition+56, Math.min(Math.max(xposition+(46+xvelocity),0),511),
255);

    do Screen.drawRect(Math.min(Math.max(xposition,0),511),

```

```

yposition+(-(fixedgap+23)), Math.min(Math.max(xposition+1,0),511),
yposition+(-(fixedgap+1)));
    do
Screen.drawRect(Math.min(Math.max(xposition+46,0),511),
yposition+(-(fixedgap+25)), Math.min(Math.max(xposition+47,0),511),
1+(yposition+(-fixedgap)));
    do Screen.drawRect(Math.min(Math.max(xposition+2,0),511),
0, Math.min(Math.max(xposition+3,0),511), yposition+(-
(fixedgap+26)));
    do
Screen.drawRect(Math.min(Math.max(xposition+44,0),511), 0,
Math.min(Math.max(xposition+45,0),511), yposition+(-(fixedgap+25)));

return;
}

```

```

method void updpipeline() {
    let time = time + 1;
    // skip every 5th time (overall slow down a bit)
    if (time = 5) {
        let time = 0;
    } else {
        let xposition = xposition - xvelocity;
        // out of bounds (left)
        if (xposition < (-50)) {
            let pass = true;
            let xposition = 514;
            let yposition = LCGRandom.randrange(80,194);
        }
        // far out of bounds (right)
        if (xposition > 614) {
            let yposition = LCGRandom.randrange(80,194);
        }
    }
}

```



```

    return;
}

method void update() {
    do updpipeline();
    do draw();
    return;
}

method int x() {
    return xposition;
}

method int y() {
    return yposition;
}

method int gameover(int x, int y) {
    if ((x>(xposition-32))&(x<(xposition+40))) { // inside pipe region
(left-right)
        if ((y>yposition)|(y<(yposition-(fixedgap+6)))) { // inside pipe
boundary (bottom-top)
            return 2; // xpositionlision
        }
        if (pass) {
            let pass = false;
            return 1; // safe-inside-region first time
        }
        else {return 0;} // safe-inside-region other time
    }
    return 0; // not in pipe region
}

method void setpipe(int x, int y){

```

```

    let xposition = x;
    let yposition = y;
    let time = -1;
    let pass = true;
    return;
}

}

class Score {
    field int total;
    field int v1;
    field int v2;
    field int v3;

    constructor Score new() {
        do reset();
        return this;
    }

    method void dispose() {
        do Memory.deAlloc(this);
        return;
    }

    method void render(int x, int v) {
        if (v = 0) {do Flappy.draw_digit0(x+512);} else {
        if (v = 1) {do Flappy.draw_digit1(x+512);} else {
        if (v = 2) {do Flappy.draw_digit2(x+512);} else {
        if (v = 3) {do Flappy.draw_digit3(x+512);} else {
        if (v = 4) {do Flappy.draw_digit4(x+512);} else {
        if (v = 5) {do Flappy.draw_digit5(x+512);} else {
        if (v = 6) {do Flappy.draw_digit6(x+512);} else {
        if (v = 7) {do Flappy.draw_digit7(x+512);} else {
        if (v = 8) {do Flappy.draw_digit8(x+512);} else {

```

```

    if (v = 9) {do Flappy.draw_digit9(x+512);} else {
    } } } } } } } } } }
    return;
}

```

```

method void show() {
    if (total > 99) {
        do render(16,v1);
        do render(14,v2);
        do render(12,v3);
    } else {
        if (total > 9) {
            do render(16,v1);
            do render(14,v2);
        }
        else {
            if (total > 0) {do render(15,v1);}
        }
    }
    return;
}

```

```

// keeps track of individual digits
method void increment() {
    let total = total + 1;
    let v1 = v1 + 1;
    if (v1 > 9) {
        let v1 = 0;
        let v2 = v2 + 1;
        if (v2 > 9) {
            let v2 = 0;
            let v3 = v3 + 1;
        }
    }
}

```

```

        return;
    }

    method int value() {
        return total;
    }

    method void reset() {
        let v1 = 0;
        let v2 = 0;
        let v3 = 0;
        let total = 0;
        return;
    }
}

class Mod {

function int mod(int a, int b) {
    var int qoutient;
    var int reminder;
    let qoutient = (a / b);
    let reminder = a - (b * qoutient);
    return reminder;
}

}

class LCGRandom {
    static int seed;
    static int A;
    static int M;
    static int Q;
    static int R;

```

```

function void setSeed(int newSeed) {
    let seed = newSeed;
    if(seed=0) {
        let seed=1;
    }
    let A=219;
    let M=32749;
    let Q=M/A;
    let R=Mod.mod(M,A);
    return;
}

/* returns a random int in range 0..(M-1) inclusive */
function int rand() {
    var int test;
    let test=(A*(Mod.mod(seed,Q)))-(R*(seed/Q));
    if(test<0) {
        let seed=test+M;
    }
    else {
        let seed=test;
    }
    return seed;
}

/* returns a random int in range low..high inclusive */
function int randRange(int low, int high) {
    var int scale;
    let scale = (M / (high - low + 1));
    return (LCGRandom.rand() / scale) + low;
}

}

class Flappy {
    function void draw_digit0(int location) {

```

```
var int memAddress;  
let memAddress = 16384+location;  
  
do Memory.poke(memAddress+0, -1);  
do Memory.poke(memAddress+32, -1);  
do Memory.poke(memAddress+64, 3);  
do Memory.poke(memAddress+96, 3);  
do Memory.poke(memAddress+128, 3);  
do Memory.poke(memAddress+160, 3);  
do Memory.poke(memAddress+192, 3);  
do Memory.poke(memAddress+224, 3);  
do Memory.poke(memAddress+256, 3);  
do Memory.poke(memAddress+288, 3);  
do Memory.poke(memAddress+320, 15363);  
do Memory.poke(memAddress+352, 15363);  
do Memory.poke(memAddress+384, 15363);  
do Memory.poke(memAddress+416, 15363);  
do Memory.poke(memAddress+448, 15363);  
do Memory.poke(memAddress+480, 15363);  
do Memory.poke(memAddress+512, 15363);  
do Memory.poke(memAddress+544, 15363);  
do Memory.poke(memAddress+576, 15363);  
do Memory.poke(memAddress+608, 15363);  
do Memory.poke(memAddress+640, 15363);  
do Memory.poke(memAddress+672, 15363);  
do Memory.poke(memAddress+704, 15363);  
do Memory.poke(memAddress+736, 15363);  
do Memory.poke(memAddress+768, 15363);  
do Memory.poke(memAddress+800, 15363);  
do Memory.poke(memAddress+832, 3);  
do Memory.poke(memAddress+864, 3);  
do Memory.poke(memAddress+896, 3);  
do Memory.poke(memAddress+928, 3);  
do Memory.poke(memAddress+960, 3);  
do Memory.poke(memAddress+992, 3);
```

```
do Memory.poke(memAddress+1024, 3);
do Memory.poke(memAddress+1056, 3);
do Memory.poke(memAddress+1088, -1);
do Memory.poke(memAddress+1120, -1);
// sprite draw column_1
do Memory.poke(memAddress+1, 255);
do Memory.poke(memAddress+33, 255);
do Memory.poke(memAddress+65, 192);
do Memory.poke(memAddress+97, 192);
do Memory.poke(memAddress+129, 192);
do Memory.poke(memAddress+161, 192);
do Memory.poke(memAddress+193, 192);
do Memory.poke(memAddress+225, 192);
do Memory.poke(memAddress+257, 192);
do Memory.poke(memAddress+289, 192);
do Memory.poke(memAddress+321, 192);
do Memory.poke(memAddress+353, 192);
do Memory.poke(memAddress+385, 192);
do Memory.poke(memAddress+417, 192);
do Memory.poke(memAddress+449, 192);
do Memory.poke(memAddress+481, 192);
do Memory.poke(memAddress+513, 192);
do Memory.poke(memAddress+545, 192);
do Memory.poke(memAddress+577, 192);
do Memory.poke(memAddress+609, 192);
do Memory.poke(memAddress+641, 192);
do Memory.poke(memAddress+673, 192);
do Memory.poke(memAddress+705, 192);
do Memory.poke(memAddress+737, 192);
do Memory.poke(memAddress+769, 192);
do Memory.poke(memAddress+801, 192);
do Memory.poke(memAddress+833, 192);
do Memory.poke(memAddress+865, 192);
do Memory.poke(memAddress+897, 192);
do Memory.poke(memAddress+929, 192);
```

```

do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
do Memory.poke(memAddress+1121, 255);
return;
}

```

```

function void draw_digit1(int location) {
  var int memAddress;
  let memAddress = 16384+location;

  do Memory.poke(memAddress+0, -256);
  do Memory.poke(memAddress+32, -256);
  do Memory.poke(memAddress+64, 768);
  do Memory.poke(memAddress+96, 768);
  do Memory.poke(memAddress+128, 768);
  do Memory.poke(memAddress+160, 768);
  do Memory.poke(memAddress+192, 768);
  do Memory.poke(memAddress+224, 768);
  do Memory.poke(memAddress+256, 768);
  do Memory.poke(memAddress+288, 768);
  do Memory.poke(memAddress+320, 16128);
  do Memory.poke(memAddress+352, 16128);
  do Memory.poke(memAddress+384, 12288);
  do Memory.poke(memAddress+416, 12288);
  do Memory.poke(memAddress+448, 12288);
  do Memory.poke(memAddress+480, 12288);
  do Memory.poke(memAddress+512, 12288);
  do Memory.poke(memAddress+544, 12288);
  do Memory.poke(memAddress+576, 12288);
  do Memory.poke(memAddress+608, 12288);
  do Memory.poke(memAddress+640, 12288);
  do Memory.poke(memAddress+672, 12288);
}

```



```
do Memory.poke(memAddress+704, 12288);
do Memory.poke(memAddress+736, 12288);
do Memory.poke(memAddress+768, 12288);
do Memory.poke(memAddress+800, 12288);
do Memory.poke(memAddress+832, 12288);
do Memory.poke(memAddress+864, 12288);
do Memory.poke(memAddress+896, 12288);
do Memory.poke(memAddress+928, 12288);
do Memory.poke(memAddress+960, 12288);
do Memory.poke(memAddress+992, 12288);
do Memory.poke(memAddress+1024, 12288);
do Memory.poke(memAddress+1056, 12288);
do Memory.poke(memAddress+1088, -4096);
do Memory.poke(memAddress+1120, -4096);
```

```
do Memory.poke(memAddress+1, 255);
do Memory.poke(memAddress+33, 255);
do Memory.poke(memAddress+65, 192);
do Memory.poke(memAddress+97, 192);
do Memory.poke(memAddress+129, 192);
do Memory.poke(memAddress+161, 192);
do Memory.poke(memAddress+193, 192);
do Memory.poke(memAddress+225, 192);
do Memory.poke(memAddress+257, 192);
do Memory.poke(memAddress+289, 192);
do Memory.poke(memAddress+321, 192);
do Memory.poke(memAddress+353, 192);
do Memory.poke(memAddress+385, 192);
do Memory.poke(memAddress+417, 192);
do Memory.poke(memAddress+449, 192);
do Memory.poke(memAddress+481, 192);
do Memory.poke(memAddress+513, 192);
do Memory.poke(memAddress+545, 192);
do Memory.poke(memAddress+577, 192);
do Memory.poke(memAddress+609, 192);
```

```

do Memory.poke(memAddress+641, 192);
do Memory.poke(memAddress+673, 192);
do Memory.poke(memAddress+705, 192);
do Memory.poke(memAddress+737, 192);
do Memory.poke(memAddress+769, 192);
do Memory.poke(memAddress+801, 192);
do Memory.poke(memAddress+833, 192);
do Memory.poke(memAddress+865, 192);
do Memory.poke(memAddress+897, 192);
do Memory.poke(memAddress+929, 192);
do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
do Memory.poke(memAddress+1121, 255);
return;
}

```

```

function void draw_digit2(int location) {
    var int memAddress;
    let memAddress = 16384+location;

    do Memory.poke(memAddress+0, -1);
    do Memory.poke(memAddress+32, -1);
    do Memory.poke(memAddress+64, 3);
    do Memory.poke(memAddress+96, 3);
    do Memory.poke(memAddress+128, 3);
    do Memory.poke(memAddress+160, 3);
    do Memory.poke(memAddress+192, 3);
    do Memory.poke(memAddress+224, 3);
    do Memory.poke(memAddress+256, 3);
    do Memory.poke(memAddress+288, 3);
    do Memory.poke(memAddress+320, 16383);
    do Memory.poke(memAddress+352, 16383);
}

```

```
do Memory.poke(memAddress+384, 16383);
do Memory.poke(memAddress+416, 16383);
do Memory.poke(memAddress+448, 3);
do Memory.poke(memAddress+480, 3);
do Memory.poke(memAddress+512, 3);
do Memory.poke(memAddress+544, 3);
do Memory.poke(memAddress+576, 3);
do Memory.poke(memAddress+608, 3);
do Memory.poke(memAddress+640, 3);
do Memory.poke(memAddress+672, 3);
do Memory.poke(memAddress+704, -1021);
do Memory.poke(memAddress+736, -1021);
do Memory.poke(memAddress+768, -1021);
do Memory.poke(memAddress+800, -1021);
do Memory.poke(memAddress+832, 3);
do Memory.poke(memAddress+864, 3);
do Memory.poke(memAddress+896, 3);
do Memory.poke(memAddress+928, 3);
do Memory.poke(memAddress+960, 3);
do Memory.poke(memAddress+992, 3);
do Memory.poke(memAddress+1024, 3);
do Memory.poke(memAddress+1056, 3);
do Memory.poke(memAddress+1088, -1);
do Memory.poke(memAddress+1120, -1);
```

```
do Memory.poke(memAddress+1, 255);
do Memory.poke(memAddress+33, 255);
do Memory.poke(memAddress+65, 192);
do Memory.poke(memAddress+97, 192);
do Memory.poke(memAddress+129, 192);
do Memory.poke(memAddress+161, 192);
do Memory.poke(memAddress+193, 192);
do Memory.poke(memAddress+225, 192);
do Memory.poke(memAddress+257, 192);
do Memory.poke(memAddress+289, 192);
```

```

do Memory.poke(memAddress+321, 192);
do Memory.poke(memAddress+353, 192);
do Memory.poke(memAddress+385, 192);
do Memory.poke(memAddress+417, 192);
do Memory.poke(memAddress+449, 192);
do Memory.poke(memAddress+481, 192);
do Memory.poke(memAddress+513, 192);
do Memory.poke(memAddress+545, 192);
do Memory.poke(memAddress+577, 192);
do Memory.poke(memAddress+609, 192);
do Memory.poke(memAddress+641, 192);
do Memory.poke(memAddress+673, 192);
do Memory.poke(memAddress+705, 255);
do Memory.poke(memAddress+737, 255);
do Memory.poke(memAddress+769, 255);
do Memory.poke(memAddress+801, 255);
do Memory.poke(memAddress+833, 192);
do Memory.poke(memAddress+865, 192);
do Memory.poke(memAddress+897, 192);
do Memory.poke(memAddress+929, 192);
do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
do Memory.poke(memAddress+1121, 255);
return;
}

```

```

function void draw_digit3(int location) {
    var int memAddress;
    let memAddress = 16384+location;

    do Memory.poke(memAddress+0, -1);
    do Memory.poke(memAddress+32, -1);
}

```

```
do Memory.poke(memAddress+64, 3);
do Memory.poke(memAddress+96, 3);
do Memory.poke(memAddress+128, 3);
do Memory.poke(memAddress+160, 3);
do Memory.poke(memAddress+192, 3);
do Memory.poke(memAddress+224, 3);
do Memory.poke(memAddress+256, 3);
do Memory.poke(memAddress+288, 3);
do Memory.poke(memAddress+320, 16383);
do Memory.poke(memAddress+352, 16383);
do Memory.poke(memAddress+384, 16383);
do Memory.poke(memAddress+416, 16383);
do Memory.poke(memAddress+448, 3);
do Memory.poke(memAddress+480, 3);
do Memory.poke(memAddress+512, 3);
do Memory.poke(memAddress+544, 3);
do Memory.poke(memAddress+576, 3);
do Memory.poke(memAddress+608, 3);
do Memory.poke(memAddress+640, 3);
do Memory.poke(memAddress+672, 3);
do Memory.poke(memAddress+704, 16383);
do Memory.poke(memAddress+736, 16383);
do Memory.poke(memAddress+768, 16383);
do Memory.poke(memAddress+800, 16383);
do Memory.poke(memAddress+832, 3);
do Memory.poke(memAddress+864, 3);
do Memory.poke(memAddress+896, 3);
do Memory.poke(memAddress+928, 3);
do Memory.poke(memAddress+960, 3);
do Memory.poke(memAddress+992, 3);
do Memory.poke(memAddress+1024, 3);
do Memory.poke(memAddress+1056, 3);
do Memory.poke(memAddress+1088, -1);
do Memory.poke(memAddress+1120, -1);
// sprite draw column_1
```

```
do Memory.poke(memAddress+1, 255);
do Memory.poke(memAddress+33, 255);
do Memory.poke(memAddress+65, 192);
do Memory.poke(memAddress+97, 192);
do Memory.poke(memAddress+129, 192);
do Memory.poke(memAddress+161, 192);
do Memory.poke(memAddress+193, 192);
do Memory.poke(memAddress+225, 192);
do Memory.poke(memAddress+257, 192);
do Memory.poke(memAddress+289, 192);
do Memory.poke(memAddress+321, 192);
do Memory.poke(memAddress+353, 192);
do Memory.poke(memAddress+385, 192);
do Memory.poke(memAddress+417, 192);
do Memory.poke(memAddress+449, 192);
do Memory.poke(memAddress+481, 192);
do Memory.poke(memAddress+513, 192);
do Memory.poke(memAddress+545, 192);
do Memory.poke(memAddress+577, 192);
do Memory.poke(memAddress+609, 192);
do Memory.poke(memAddress+641, 192);
do Memory.poke(memAddress+673, 192);
do Memory.poke(memAddress+705, 192);
do Memory.poke(memAddress+737, 192);
do Memory.poke(memAddress+769, 192);
do Memory.poke(memAddress+801, 192);
do Memory.poke(memAddress+833, 192);
do Memory.poke(memAddress+865, 192);
do Memory.poke(memAddress+897, 192);
do Memory.poke(memAddress+929, 192);
do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
```

```

    do Memory.poke(memAddress+1121, 255);
    return;
}

function void draw_digit4(int location) {
    var int memAddress;
    let memAddress = 16384+location;

    do Memory.poke(memAddress+0, -1);
    do Memory.poke(memAddress+32, -1);
    do Memory.poke(memAddress+64, 15363);
    do Memory.poke(memAddress+96, 15363);
    do Memory.poke(memAddress+128, 15363);
    do Memory.poke(memAddress+160, 15363);
    do Memory.poke(memAddress+192, 15363);
    do Memory.poke(memAddress+224, 15363);
    do Memory.poke(memAddress+256, 15363);
    do Memory.poke(memAddress+288, 15363);
    do Memory.poke(memAddress+320, 15363);
    do Memory.poke(memAddress+352, 15363);
    do Memory.poke(memAddress+384, 15363);
    do Memory.poke(memAddress+416, 15363);
    do Memory.poke(memAddress+448, 3);
    do Memory.poke(memAddress+480, 3);
    do Memory.poke(memAddress+512, 3);
    do Memory.poke(memAddress+544, 3);
    do Memory.poke(memAddress+576, 3);
    do Memory.poke(memAddress+608, 3);
    do Memory.poke(memAddress+640, 3);
    do Memory.poke(memAddress+672, 3);
    do Memory.poke(memAddress+704, 16383);
    do Memory.poke(memAddress+736, 16383);
    do Memory.poke(memAddress+768, 12288);
    do Memory.poke(memAddress+800, 12288);
    do Memory.poke(memAddress+832, 12288);

```

```
do Memory.poke(memAddress+864, 12288);
do Memory.poke(memAddress+896, 12288);
do Memory.poke(memAddress+928, 12288);
do Memory.poke(memAddress+960, 12288);
do Memory.poke(memAddress+992, 12288);
do Memory.poke(memAddress+1024, 12288);
do Memory.poke(memAddress+1056, 12288);
do Memory.poke(memAddress+1088, -4096);
do Memory.poke(memAddress+1120, -4096);
```

```
do Memory.poke(memAddress+1, 255);
do Memory.poke(memAddress+33, 255);
do Memory.poke(memAddress+65, 192);
do Memory.poke(memAddress+97, 192);
do Memory.poke(memAddress+129, 192);
do Memory.poke(memAddress+161, 192);
do Memory.poke(memAddress+193, 192);
do Memory.poke(memAddress+225, 192);
do Memory.poke(memAddress+257, 192);
do Memory.poke(memAddress+289, 192);
do Memory.poke(memAddress+321, 192);
do Memory.poke(memAddress+353, 192);
do Memory.poke(memAddress+385, 192);
do Memory.poke(memAddress+417, 192);
do Memory.poke(memAddress+449, 192);
do Memory.poke(memAddress+481, 192);
do Memory.poke(memAddress+513, 192);
do Memory.poke(memAddress+545, 192);
do Memory.poke(memAddress+577, 192);
do Memory.poke(memAddress+609, 192);
do Memory.poke(memAddress+641, 192);
do Memory.poke(memAddress+673, 192);
do Memory.poke(memAddress+705, 192);
do Memory.poke(memAddress+737, 192);
do Memory.poke(memAddress+769, 192);
```



```

do Memory.poke(memAddress+801, 192);
do Memory.poke(memAddress+833, 192);
do Memory.poke(memAddress+865, 192);
do Memory.poke(memAddress+897, 192);
do Memory.poke(memAddress+929, 192);
do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
do Memory.poke(memAddress+1121, 255);
return;
}

```

```

function void draw_digit5(int location) {
    var int memAddress;
    let memAddress = 16384+location;

    do Memory.poke(memAddress+0, -1);
    do Memory.poke(memAddress+32, -1);
    do Memory.poke(memAddress+64, 3);
    do Memory.poke(memAddress+96, 3);
    do Memory.poke(memAddress+128, 3);
    do Memory.poke(memAddress+160, 3);
    do Memory.poke(memAddress+192, 3);
    do Memory.poke(memAddress+224, 3);
    do Memory.poke(memAddress+256, 3);
    do Memory.poke(memAddress+288, 3);
    do Memory.poke(memAddress+320, -1021);
    do Memory.poke(memAddress+352, -1021);
    do Memory.poke(memAddress+384, -1021);
    do Memory.poke(memAddress+416, -1021);
    do Memory.poke(memAddress+448, 3);
    do Memory.poke(memAddress+480, 3);
    do Memory.poke(memAddress+512, 3);
}

```

```

do Memory.poke(memAddress+544, 3);
do Memory.poke(memAddress+576, 3);
do Memory.poke(memAddress+608, 3);
do Memory.poke(memAddress+640, 3);
do Memory.poke(memAddress+672, 3);
do Memory.poke(memAddress+704, 16383);
do Memory.poke(memAddress+736, 16383);
do Memory.poke(memAddress+768, 16383);
do Memory.poke(memAddress+800, 16383);
do Memory.poke(memAddress+832, 3);
do Memory.poke(memAddress+864, 3);
do Memory.poke(memAddress+896, 3);
do Memory.poke(memAddress+928, 3);
do Memory.poke(memAddress+960, 3);
do Memory.poke(memAddress+992, 3);
do Memory.poke(memAddress+1024, 3);
do Memory.poke(memAddress+1056, 3);
do Memory.poke(memAddress+1088, -1);
do Memory.poke(memAddress+1120, -1);
// sprite draw column_1
do Memory.poke(memAddress+1, 255);
do Memory.poke(memAddress+33, 255);
do Memory.poke(memAddress+65, 192);
do Memory.poke(memAddress+97, 192);
do Memory.poke(memAddress+129, 192);
do Memory.poke(memAddress+161, 192);
do Memory.poke(memAddress+193, 192);
do Memory.poke(memAddress+225, 192);
do Memory.poke(memAddress+257, 192);
do Memory.poke(memAddress+289, 192);
do Memory.poke(memAddress+321, 255);
do Memory.poke(memAddress+353, 255);
do Memory.poke(memAddress+385, 255);
do Memory.poke(memAddress+417, 255);
do Memory.poke(memAddress+449, 192);

```

```

do Memory.poke(memAddress+481, 192);
do Memory.poke(memAddress+513, 192);
do Memory.poke(memAddress+545, 192);
do Memory.poke(memAddress+577, 192);
do Memory.poke(memAddress+609, 192);
do Memory.poke(memAddress+641, 192);
do Memory.poke(memAddress+673, 192);
do Memory.poke(memAddress+705, 192);
do Memory.poke(memAddress+737, 192);
do Memory.poke(memAddress+769, 192);
do Memory.poke(memAddress+801, 192);
do Memory.poke(memAddress+833, 192);
do Memory.poke(memAddress+865, 192);
do Memory.poke(memAddress+897, 192);
do Memory.poke(memAddress+929, 192);
do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
do Memory.poke(memAddress+1121, 255);
return;
}

```

```

function void draw_digit6(int location) {
  var int memAddress;
  let memAddress = 16384+location;

  do Memory.poke(memAddress+0, -1);
  do Memory.poke(memAddress+32, -1);
  do Memory.poke(memAddress+64, 3);
  do Memory.poke(memAddress+96, 3);
  do Memory.poke(memAddress+128, 3);
  do Memory.poke(memAddress+160, 3);
  do Memory.poke(memAddress+192, 3);
}

```

```
do Memory.poke(memAddress+224, 3);
do Memory.poke(memAddress+256, 3);
do Memory.poke(memAddress+288, 3);
do Memory.poke(memAddress+320, -1021);
do Memory.poke(memAddress+352, -1021);
do Memory.poke(memAddress+384, -1021);
do Memory.poke(memAddress+416, -1021);
do Memory.poke(memAddress+448, 3);
do Memory.poke(memAddress+480, 3);
do Memory.poke(memAddress+512, 3);
do Memory.poke(memAddress+544, 3);
do Memory.poke(memAddress+576, 3);
do Memory.poke(memAddress+608, 3);
do Memory.poke(memAddress+640, 3);
do Memory.poke(memAddress+672, 3);
do Memory.poke(memAddress+704, 15363);
do Memory.poke(memAddress+736, 15363);
do Memory.poke(memAddress+768, 15363);
do Memory.poke(memAddress+800, 15363);
do Memory.poke(memAddress+832, 3);
do Memory.poke(memAddress+864, 3);
do Memory.poke(memAddress+896, 3);
do Memory.poke(memAddress+928, 3);
do Memory.poke(memAddress+960, 3);
do Memory.poke(memAddress+992, 3);
do Memory.poke(memAddress+1024, 3);
do Memory.poke(memAddress+1056, 3);
do Memory.poke(memAddress+1088, -1);
do Memory.poke(memAddress+1120, -1);
```

```
do Memory.poke(memAddress+1, 255);
do Memory.poke(memAddress+33, 255);
do Memory.poke(memAddress+65, 192);
do Memory.poke(memAddress+97, 192);
do Memory.poke(memAddress+129, 192);
```

```

do Memory.poke(memAddress+161, 192);
do Memory.poke(memAddress+193, 192);
do Memory.poke(memAddress+225, 192);
do Memory.poke(memAddress+257, 192);
do Memory.poke(memAddress+289, 192);
do Memory.poke(memAddress+321, 255);
do Memory.poke(memAddress+353, 255);
do Memory.poke(memAddress+385, 255);
do Memory.poke(memAddress+417, 255);
do Memory.poke(memAddress+449, 192);
do Memory.poke(memAddress+481, 192);
do Memory.poke(memAddress+513, 192);
do Memory.poke(memAddress+545, 192);
do Memory.poke(memAddress+577, 192);
do Memory.poke(memAddress+609, 192);
do Memory.poke(memAddress+641, 192);
do Memory.poke(memAddress+673, 192);
do Memory.poke(memAddress+705, 192);
do Memory.poke(memAddress+737, 192);
do Memory.poke(memAddress+769, 192);
do Memory.poke(memAddress+801, 192);
do Memory.poke(memAddress+833, 192);
do Memory.poke(memAddress+865, 192);
do Memory.poke(memAddress+897, 192);
do Memory.poke(memAddress+929, 192);
do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
do Memory.poke(memAddress+1121, 255);
return;
}

```

```

function void draw_digit7(int location) {

```

```
var int memAddress;  
let memAddress = 16384+location;  
  
do Memory.poke(memAddress+0, -1);  
do Memory.poke(memAddress+32, -1);  
do Memory.poke(memAddress+64, 3);  
do Memory.poke(memAddress+96, 3);  
do Memory.poke(memAddress+128, 3);  
do Memory.poke(memAddress+160, 3);  
do Memory.poke(memAddress+192, 3);  
do Memory.poke(memAddress+224, 3);  
do Memory.poke(memAddress+256, 3);  
do Memory.poke(memAddress+288, 3);  
do Memory.poke(memAddress+320, 16383);  
do Memory.poke(memAddress+352, 16383);  
do Memory.poke(memAddress+384, 12288);  
do Memory.poke(memAddress+416, 12288);  
do Memory.poke(memAddress+448, 12288);  
do Memory.poke(memAddress+480, 12288);  
do Memory.poke(memAddress+512, 12288);  
do Memory.poke(memAddress+544, 12288);  
do Memory.poke(memAddress+576, 12288);  
do Memory.poke(memAddress+608, 12288);  
do Memory.poke(memAddress+640, 12288);  
do Memory.poke(memAddress+672, 12288);  
do Memory.poke(memAddress+704, 12288);  
do Memory.poke(memAddress+736, 12288);  
do Memory.poke(memAddress+768, 12288);  
do Memory.poke(memAddress+800, 12288);  
do Memory.poke(memAddress+832, 12288);  
do Memory.poke(memAddress+864, 12288);  
do Memory.poke(memAddress+896, 12288);  
do Memory.poke(memAddress+928, 12288);  
do Memory.poke(memAddress+960, 12288);  
do Memory.poke(memAddress+992, 12288);
```

```
do Memory.poke(memAddress+1024, 12288);  
do Memory.poke(memAddress+1056, 12288);  
do Memory.poke(memAddress+1088, -4096);  
do Memory.poke(memAddress+1120, -4096);
```

```
do Memory.poke(memAddress+1, 255);  
do Memory.poke(memAddress+33, 255);  
do Memory.poke(memAddress+65, 192);  
do Memory.poke(memAddress+97, 192);  
do Memory.poke(memAddress+129, 192);  
do Memory.poke(memAddress+161, 192);  
do Memory.poke(memAddress+193, 192);  
do Memory.poke(memAddress+225, 192);  
do Memory.poke(memAddress+257, 192);  
do Memory.poke(memAddress+289, 192);  
do Memory.poke(memAddress+321, 192);  
do Memory.poke(memAddress+353, 192);  
do Memory.poke(memAddress+385, 192);  
do Memory.poke(memAddress+417, 192);  
do Memory.poke(memAddress+449, 192);  
do Memory.poke(memAddress+481, 192);  
do Memory.poke(memAddress+513, 192);  
do Memory.poke(memAddress+545, 192);  
do Memory.poke(memAddress+577, 192);  
do Memory.poke(memAddress+609, 192);  
do Memory.poke(memAddress+641, 192);  
do Memory.poke(memAddress+673, 192);  
do Memory.poke(memAddress+705, 192);  
do Memory.poke(memAddress+737, 192);  
do Memory.poke(memAddress+769, 192);  
do Memory.poke(memAddress+801, 192);  
do Memory.poke(memAddress+833, 192);  
do Memory.poke(memAddress+865, 192);  
do Memory.poke(memAddress+897, 192);  
do Memory.poke(memAddress+929, 192);
```

```

do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
do Memory.poke(memAddress+1121, 255);
return;
}

function void draw_digit8(int location) {
    var int memAddress;
    let memAddress = 16384+location;

    do Memory.poke(memAddress+0, -1);
    do Memory.poke(memAddress+32, -1);
    do Memory.poke(memAddress+64, 3);
    do Memory.poke(memAddress+96, 3);
    do Memory.poke(memAddress+128, 3);
    do Memory.poke(memAddress+160, 3);
    do Memory.poke(memAddress+192, 3);
    do Memory.poke(memAddress+224, 3);
    do Memory.poke(memAddress+256, 3);
    do Memory.poke(memAddress+288, 3);
    do Memory.poke(memAddress+320, 15363);
    do Memory.poke(memAddress+352, 15363);
    do Memory.poke(memAddress+384, 15363);
    do Memory.poke(memAddress+416, 15363);
    do Memory.poke(memAddress+448, 3);
    do Memory.poke(memAddress+480, 3);
    do Memory.poke(memAddress+512, 3);
    do Memory.poke(memAddress+544, 3);
    do Memory.poke(memAddress+576, 3);
    do Memory.poke(memAddress+608, 3);
    do Memory.poke(memAddress+640, 3);
    do Memory.poke(memAddress+672, 3);

```



```
do Memory.poke(memAddress+704, 15363);
do Memory.poke(memAddress+736, 15363);
do Memory.poke(memAddress+768, 15363);
do Memory.poke(memAddress+800, 15363);
do Memory.poke(memAddress+832, 3);
do Memory.poke(memAddress+864, 3);
do Memory.poke(memAddress+896, 3);
do Memory.poke(memAddress+928, 3);
do Memory.poke(memAddress+960, 3);
do Memory.poke(memAddress+992, 3);
do Memory.poke(memAddress+1024, 3);
do Memory.poke(memAddress+1056, 3);
do Memory.poke(memAddress+1088, -1);
do Memory.poke(memAddress+1120, -1);
```

```
do Memory.poke(memAddress+1, 255);
do Memory.poke(memAddress+33, 255);
do Memory.poke(memAddress+65, 192);
do Memory.poke(memAddress+97, 192);
do Memory.poke(memAddress+129, 192);
do Memory.poke(memAddress+161, 192);
do Memory.poke(memAddress+193, 192);
do Memory.poke(memAddress+225, 192);
do Memory.poke(memAddress+257, 192);
do Memory.poke(memAddress+289, 192);
do Memory.poke(memAddress+321, 192);
do Memory.poke(memAddress+353, 192);
do Memory.poke(memAddress+385, 192);
do Memory.poke(memAddress+417, 192);
do Memory.poke(memAddress+449, 192);
do Memory.poke(memAddress+481, 192);
do Memory.poke(memAddress+513, 192);
do Memory.poke(memAddress+545, 192);
do Memory.poke(memAddress+577, 192);
do Memory.poke(memAddress+609, 192);
```

```

do Memory.poke(memAddress+641, 192);
do Memory.poke(memAddress+673, 192);
do Memory.poke(memAddress+705, 192);
do Memory.poke(memAddress+737, 192);
do Memory.poke(memAddress+769, 192);
do Memory.poke(memAddress+801, 192);
do Memory.poke(memAddress+833, 192);
do Memory.poke(memAddress+865, 192);
do Memory.poke(memAddress+897, 192);
do Memory.poke(memAddress+929, 192);
do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
do Memory.poke(memAddress+1121, 255);
return;
}

function void draw_digit9(int location) {
    var int memAddress;
    let memAddress = 16384+location;

    do Memory.poke(memAddress+0, -1);
    do Memory.poke(memAddress+32, -1);
    do Memory.poke(memAddress+64, 3);
    do Memory.poke(memAddress+96, 3);
    do Memory.poke(memAddress+128, 3);
    do Memory.poke(memAddress+160, 3);
    do Memory.poke(memAddress+192, 3);
    do Memory.poke(memAddress+224, 3);
    do Memory.poke(memAddress+256, 3);
    do Memory.poke(memAddress+288, 3);
    do Memory.poke(memAddress+320, 15363);
    do Memory.poke(memAddress+352, 15363);

```

```
do Memory.poke(memAddress+384, 15363);
do Memory.poke(memAddress+416, 15363);
do Memory.poke(memAddress+448, 3);
do Memory.poke(memAddress+480, 3);
do Memory.poke(memAddress+512, 3);
do Memory.poke(memAddress+544, 3);
do Memory.poke(memAddress+576, 3);
do Memory.poke(memAddress+608, 3);
do Memory.poke(memAddress+640, 3);
do Memory.poke(memAddress+672, 3);
do Memory.poke(memAddress+704, 16383);
do Memory.poke(memAddress+736, 16383);
do Memory.poke(memAddress+768, 16383);
do Memory.poke(memAddress+800, 16383);
do Memory.poke(memAddress+832, 3);
do Memory.poke(memAddress+864, 3);
do Memory.poke(memAddress+896, 3);
do Memory.poke(memAddress+928, 3);
do Memory.poke(memAddress+960, 3);
do Memory.poke(memAddress+992, 3);
do Memory.poke(memAddress+1024, 3);
do Memory.poke(memAddress+1056, 3);
do Memory.poke(memAddress+1088, -1);
do Memory.poke(memAddress+1120, -1);
```

```
do Memory.poke(memAddress+1, 255);
do Memory.poke(memAddress+33, 255);
do Memory.poke(memAddress+65, 192);
do Memory.poke(memAddress+97, 192);
do Memory.poke(memAddress+129, 192);
do Memory.poke(memAddress+161, 192);
do Memory.poke(memAddress+193, 192);
do Memory.poke(memAddress+225, 192);
do Memory.poke(memAddress+257, 192);
do Memory.poke(memAddress+289, 192);
```

```

do Memory.poke(memAddress+321, 192);
do Memory.poke(memAddress+353, 192);
do Memory.poke(memAddress+385, 192);
do Memory.poke(memAddress+417, 192);
do Memory.poke(memAddress+449, 192);
do Memory.poke(memAddress+481, 192);
do Memory.poke(memAddress+513, 192);
do Memory.poke(memAddress+545, 192);
do Memory.poke(memAddress+577, 192);
do Memory.poke(memAddress+609, 192);
do Memory.poke(memAddress+641, 192);
do Memory.poke(memAddress+673, 192);
do Memory.poke(memAddress+705, 192);
do Memory.poke(memAddress+737, 192);
do Memory.poke(memAddress+769, 192);
do Memory.poke(memAddress+801, 192);
do Memory.poke(memAddress+833, 192);
do Memory.poke(memAddress+865, 192);
do Memory.poke(memAddress+897, 192);
do Memory.poke(memAddress+929, 192);
do Memory.poke(memAddress+961, 192);
do Memory.poke(memAddress+993, 192);
do Memory.poke(memAddress+1025, 192);
do Memory.poke(memAddress+1057, 192);
do Memory.poke(memAddress+1089, 255);
do Memory.poke(memAddress+1121, 255);
return;
}

```

```

function void flappy1(int location) {
    var int memAddress;
    let memAddress = 16384+location;

    do Memory.poke(memAddress+0, 0);
    do Memory.poke(memAddress+32, 0);
}

```

```
do Memory.poke(memAddress+64, 0);
do Memory.poke(memAddress+96, 0);
do Memory.poke(memAddress+128, 0);
do Memory.poke(memAddress+160, 0);
do Memory.poke(memAddress+192, -4096);
do Memory.poke(memAddress+224, -4096);
do Memory.poke(memAddress+256, 3840);
do Memory.poke(memAddress+288, 3840);
do Memory.poke(memAddress+320, 192);
do Memory.poke(memAddress+352, 192);
do Memory.poke(memAddress+384, 1020);
do Memory.poke(memAddress+416, 1020);
do Memory.poke(memAddress+448, 3843);
do Memory.poke(memAddress+480, 3843);
do Memory.poke(memAddress+512, 15363);
do Memory.poke(memAddress+544, 15363);
do Memory.poke(memAddress+576, 12291);
do Memory.poke(memAddress+608, 12291);
do Memory.poke(memAddress+640, 3084);
do Memory.poke(memAddress+672, 3084);
do Memory.poke(memAddress+704, 1008);
do Memory.poke(memAddress+736, 1008);
do Memory.poke(memAddress+768, 48);
do Memory.poke(memAddress+800, 48);
do Memory.poke(memAddress+832, 960);
do Memory.poke(memAddress+864, 960);
do Memory.poke(memAddress+896, -1024);
do Memory.poke(memAddress+928, -1024);
do Memory.poke(memAddress+960, 0);
do Memory.poke(memAddress+992, 0);
do Memory.poke(memAddress+1024, 0);
do Memory.poke(memAddress+1056, 0);
do Memory.poke(memAddress+1088, 0);
do Memory.poke(memAddress+1120, 0);
```

```
do Memory.poke(memAddress+1, 0);
do Memory.poke(memAddress+33, 0);
do Memory.poke(memAddress+65, 0);
do Memory.poke(memAddress+97, 0);
do Memory.poke(memAddress+129, 0);
do Memory.poke(memAddress+161, 0);
do Memory.poke(memAddress+193, 255);
do Memory.poke(memAddress+225, 255);
do Memory.poke(memAddress+257, 780);
do Memory.poke(memAddress+289, 780);
do Memory.poke(memAddress+321, 3075);
do Memory.poke(memAddress+353, 3075);
do Memory.poke(memAddress+385, 13059);
do Memory.poke(memAddress+417, 13059);
do Memory.poke(memAddress+449, 13059);
do Memory.poke(memAddress+481, 13059);
do Memory.poke(memAddress+513, 12300);
do Memory.poke(memAddress+545, 12300);
do Memory.poke(memAddress+577, 16368);
do Memory.poke(memAddress+609, 16368);
do Memory.poke(memAddress+641, -16372);
do Memory.poke(memAddress+673, -16372);
do Memory.poke(memAddress+705, -13);
do Memory.poke(memAddress+737, -13);
do Memory.poke(memAddress+769, 12300);
do Memory.poke(memAddress+801, 12300);
do Memory.poke(memAddress+833, 4080);
do Memory.poke(memAddress+865, 4080);
do Memory.poke(memAddress+897, 15);
do Memory.poke(memAddress+929, 15);
do Memory.poke(memAddress+961, 0);
do Memory.poke(memAddress+993, 0);
do Memory.poke(memAddress+1025, 0);
do Memory.poke(memAddress+1057, 0);
do Memory.poke(memAddress+1089, 0);
```

```

    do Memory.poke(memAddress+1121, 0);
    return;
}

function void flappy2(int location) {
    var int memAddress;
    let memAddress = 16384+location;

    do Memory.poke(memAddress+0, 0);
    do Memory.poke(memAddress+32, 0);
    do Memory.poke(memAddress+64, 0);
    do Memory.poke(memAddress+96, 0);
    do Memory.poke(memAddress+128, 0);
    do Memory.poke(memAddress+160, 0);
    do Memory.poke(memAddress+192, -4096);
    do Memory.poke(memAddress+224, -4096);
    do Memory.poke(memAddress+256, 3840);
    do Memory.poke(memAddress+288, 3840);
    do Memory.poke(memAddress+320, 192);
    do Memory.poke(memAddress+352, 192);
    do Memory.poke(memAddress+384, 48);
    do Memory.poke(memAddress+416, 48);
    do Memory.poke(memAddress+448, 12);
    do Memory.poke(memAddress+480, 12);
    do Memory.poke(memAddress+512, 4092);
    do Memory.poke(memAddress+544, 4092);
    do Memory.poke(memAddress+576, 15375);
    do Memory.poke(memAddress+608, 15375);
    do Memory.poke(memAddress+640, 12291);
    do Memory.poke(memAddress+672, 12291);
    do Memory.poke(memAddress+704, 4092);
    do Memory.poke(memAddress+736, 4092);
    do Memory.poke(memAddress+768, 48);
    do Memory.poke(memAddress+800, 48);
    do Memory.poke(memAddress+832, 960);

```

```
do Memory.poke(memAddress+864, 960);
do Memory.poke(memAddress+896, -1024);
do Memory.poke(memAddress+928, -1024);
do Memory.poke(memAddress+960, 0);
do Memory.poke(memAddress+992, 0);
do Memory.poke(memAddress+1024, 0);
do Memory.poke(memAddress+1056, 0);
do Memory.poke(memAddress+1088, 0);
do Memory.poke(memAddress+1120, 0);
// sprite draw column_1
do Memory.poke(memAddress+1, 0);
do Memory.poke(memAddress+33, 0);
do Memory.poke(memAddress+65, 0);
do Memory.poke(memAddress+97, 0);
do Memory.poke(memAddress+129, 0);
do Memory.poke(memAddress+161, 0);
do Memory.poke(memAddress+193, 255);
do Memory.poke(memAddress+225, 255);
do Memory.poke(memAddress+257, 780);
do Memory.poke(memAddress+289, 780);
do Memory.poke(memAddress+321, 3075);
do Memory.poke(memAddress+353, 3075);
do Memory.poke(memAddress+385, 13059);
do Memory.poke(memAddress+417, 13059);
do Memory.poke(memAddress+449, 13059);
do Memory.poke(memAddress+481, 13059);
do Memory.poke(memAddress+513, 12300);
do Memory.poke(memAddress+545, 12300);
do Memory.poke(memAddress+577, 16368);
do Memory.poke(memAddress+609, 16368);
do Memory.poke(memAddress+641, -16372);
do Memory.poke(memAddress+673, -16372);
do Memory.poke(memAddress+705, -13);
do Memory.poke(memAddress+737, -13);
do Memory.poke(memAddress+769, 12300);
```



```

do Memory.poke(memAddress+801, 12300);
do Memory.poke(memAddress+833, 4080);
do Memory.poke(memAddress+865, 4080);
do Memory.poke(memAddress+897, 15);
do Memory.poke(memAddress+929, 15);
do Memory.poke(memAddress+961, 0);
do Memory.poke(memAddress+993, 0);
do Memory.poke(memAddress+1025, 0);
do Memory.poke(memAddress+1057, 0);
do Memory.poke(memAddress+1089, 0);
do Memory.poke(memAddress+1121, 0);
return;
}

```

```

function void flappy3(int location) {
    var int memAddress;
    let memAddress = 16384+location;

    do Memory.poke(memAddress+0, 0);
    do Memory.poke(memAddress+32, 0);
    do Memory.poke(memAddress+64, 0);
    do Memory.poke(memAddress+96, 0);
    do Memory.poke(memAddress+128, 0);
    do Memory.poke(memAddress+160, 0);
    do Memory.poke(memAddress+192, -4096);
    do Memory.poke(memAddress+224, -4096);
    do Memory.poke(memAddress+256, 3840);
    do Memory.poke(memAddress+288, 3840);
    do Memory.poke(memAddress+320, 192);
    do Memory.poke(memAddress+352, 192);
    do Memory.poke(memAddress+384, 48);
    do Memory.poke(memAddress+416, 48);
    do Memory.poke(memAddress+448, 12);
    do Memory.poke(memAddress+480, 12);
    do Memory.poke(memAddress+512, 12);
}

```

```
do Memory.poke(memAddress+544, 12);
do Memory.poke(memAddress+576, 4092);
do Memory.poke(memAddress+608, 4092);
do Memory.poke(memAddress+640, 12291);
do Memory.poke(memAddress+672, 12291);
do Memory.poke(memAddress+704, 3075);
do Memory.poke(memAddress+736, 3075);
do Memory.poke(memAddress+768, 771);
do Memory.poke(memAddress+800, 771);
do Memory.poke(memAddress+832, 1020);
do Memory.poke(memAddress+864, 1020);
do Memory.poke(memAddress+896, -1024);
do Memory.poke(memAddress+928, -1024);
do Memory.poke(memAddress+960, 0);
do Memory.poke(memAddress+992, 0);
do Memory.poke(memAddress+1024, 0);
do Memory.poke(memAddress+1056, 0);
do Memory.poke(memAddress+1088, 0);
do Memory.poke(memAddress+1120, 0);
```

```
do Memory.poke(memAddress+1, 0);
do Memory.poke(memAddress+33, 0);
do Memory.poke(memAddress+65, 0);
do Memory.poke(memAddress+97, 0);
do Memory.poke(memAddress+129, 0);
do Memory.poke(memAddress+161, 0);
do Memory.poke(memAddress+193, 255);
do Memory.poke(memAddress+225, 255);
do Memory.poke(memAddress+257, 780);
do Memory.poke(memAddress+289, 780);
do Memory.poke(memAddress+321, 3075);
do Memory.poke(memAddress+353, 3075);
do Memory.poke(memAddress+385, 13059);
do Memory.poke(memAddress+417, 13059);
do Memory.poke(memAddress+449, 13059);
```

```

do Memory.poke(memAddress+481, 13059);
do Memory.poke(memAddress+513, 12300);
do Memory.poke(memAddress+545, 12300);
do Memory.poke(memAddress+577, 16368);
do Memory.poke(memAddress+609, 16368);
do Memory.poke(memAddress+641, -16372);
do Memory.poke(memAddress+673, -16372);
do Memory.poke(memAddress+705, -13);
do Memory.poke(memAddress+737, -13);
do Memory.poke(memAddress+769, 12300);
do Memory.poke(memAddress+801, 12300);
do Memory.poke(memAddress+833, 4080);
do Memory.poke(memAddress+865, 4080);
do Memory.poke(memAddress+897, 15);
do Memory.poke(memAddress+929, 15);
do Memory.poke(memAddress+961, 0);
do Memory.poke(memAddress+993, 0);
do Memory.poke(memAddress+1025, 0);
do Memory.poke(memAddress+1057, 0);
do Memory.poke(memAddress+1089, 0);
do Memory.poke(memAddress+1121, 0);
return;
}

}

```

4.4 OUTPUT :

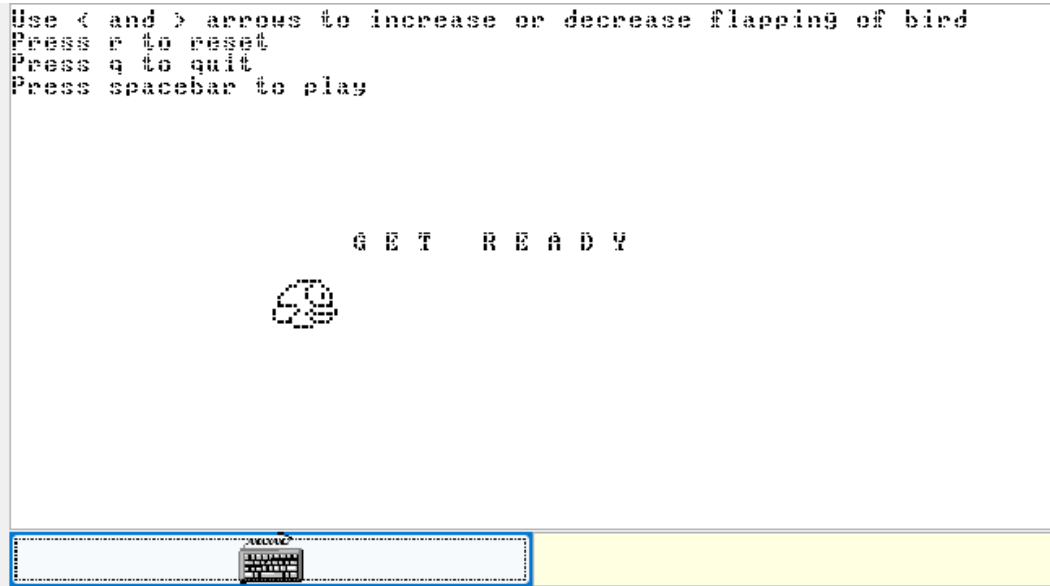
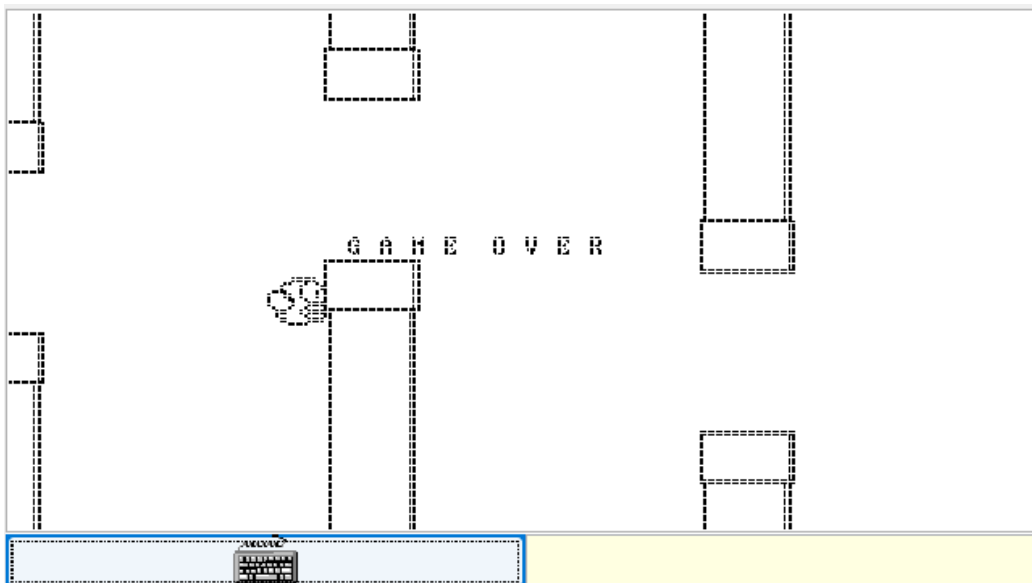


Figure 13 : Flappy bird output



The Get Ready and Game Over messages are displayed in the right circumstances and the score is incremented every time the bird passes a pipe.

5 ROCK, PAPER AND SCISSORS :

5.1 IMPLEMENTATION OF CLASSES:

In this multiplayer game a simple logic is utilized that is rock value is lower than the paper value. Paper value is lower than the scissor value. scissor value is lower than the rock value. Example if player 1 chooses rock and player 2 chooses paper in this case player 2 wins and gets a point. There is a condition if both the players choose same value then match will tie no one will get the point. Like that we can choose how many sets we have to play.at the end which player will have more points then that player will win the series. In the implementation of the code we used three different class names:

- 1) main class
- 2) GUI_R class
- 3) GUI_L class

Firstly we declare variables such as player 1,player 2,sets, pl1score,pl2score,tiescore,a.the code starts with a class called main. Initially the scores of each player is zero and uses a loop to ask the user for the number of sets they want to play. Then the loop starts inside the loop the code clears the screen and displays information about the game and the player choices.it allows the user to enter their choices (0 for rock, 1 for paper, 2 for scissor) for player 1 and player 2. After receiving the input from both players , the code clears the screen and displays the output in a diagram from. For clearing the screen we use a function call “Screen.clearScreen”. Here we used a function called “sys.wait” to give some time for outputs before starting the next set of game. After completing the sets the code clears the screen again and displays the scores summary and announce the winner of the game.

The GUI_R class is for player 2. draw_R named function is there inside the class. We took an integer called player. there are three condition if user selects 0(ROCK) or 1(PAPER) or 2(SCISSOR) using if statements and separate diagram for ROCK,PAPER AND SCISSOR character will be formed for the respective numbers on the screen. Except those three numbers anything else will shows invalid. The “output.printString” commands are used to display text on the screen.

The GUI_L class is for player 1. draw_L named function is there inside the class. We took an integer called player. there are three condition if user selects 0(ROCK) or 1(PAPER) or 2(SCISSOR) using if statements and separate diagram for ROCK, PAPER AND SCISSOR character will be formed for the respective numbers on the screen. Except those three numbers anything else will shows invalid. The “output.printString” commands are used to display text on the screen.

5.2 CODE IN JACK:

```
class Main {

    function void main(){
    var int player1;
    var int player2;
    var int sets;
    var int pl1score;
    var int pl2score;
    var int tiescore;
    var int a;

    let pl1score = 0;
    let pl2score = 0;
    let tiescore = 0;
    do Output.moveCursor(3,12);
    do Output.printString("ROCK-PAPER-SCISSOR");
    do Output.moveCursor(6,22);
    let sets = Keyboard.readInt("How many sets: ");

    let a=1;
    while((sets > a) | (sets = a)){

        do Screen.clearScreen();
        do Output.moveCursor(0,0);

        let a = a+1;
        do Output.moveCursor(0, 15);
        do Output.printString("-- PLAYERS CHOICES -");
        do Output.println();
        do Output.moveCursor(3, 12);
        do Output.printString(" 0 - ROCK || 1 - PAPER || 2-
SCISSOR");
```

```

do Output.println();
do Output.moveCursor(5, 0);
do Output.printString(" GAME SET: ");
do Output.printInt(a - 1);
do Output.println();
do Output.println();
do Output.moveCursor(7, 22);
let player1 = Keyboard.readInt("PLAYER 1 PICK : ");

do Screen.clearScreen();
do Output.moveCursor(0, 15);
do Output.printString("-- PLAYERS CHOICES ---");
do Output.println();
do Output.moveCursor(3, 12);
do Output.printString("0 - ROCK || 1 - PAPER || 2 -
SCISSOR");
do Output.println();
do Output.moveCursor(5, 0);
do Output.printString(" GAME SET: ");
do Output.printInt(a - 1);
do Output.println();
do Output.println();
do Output.moveCursor(7, 22);
let player2 = Keyboard.readInt("PLAYER 2 PICK: ");

do Screen.clearScreen();
do Output.moveCursor(6, 0);
do GUI_L.draw_L(player1);
do Output.moveCursor(6, 32);
do GUI_R.draw_R(player2);
do Output.println();
do Output.moveCursor(18, 20);
if((player1 = 0)&(player2 = 0)){
    do Output.printString("it's a tie");

```



```

    let tiescore = tiescore + 1;
}
if((player1 = 1)&(player2 = 1)){
    do Output.printString("it's a tie");
    let tiescore = tiescore + 1;
}
if((player1 = 2)&(player2 = 2)){
    do Output.printString("it's a tie");
    let tiescore = tiescore + 1;
}

if((player1 = 0)&(player2 = 1)){
    do Output.printString(" player2 wins");
    let pl2score = pl2score + 1;
}

if((player1 = 1)& (player2 = 2)){
    do Output.printString(" player2 wins");
    let pl2score = pl2score + 1;
}

if((player1 = 2)&(player2 = 0)){
    do Output.printString(" player2 wins");
    let pl2score = pl2score + 1;
}

if((player1 = 1)&(player2 = 0)){
    do Output.printString(" player1 wins");
    let pl1score = pl1score + 1;
}
if((player1 = 2)&(player2 = 1)) {
    do Output.printString(" player1 wins");
    let pl1score = pl1score+ 1;
}
if((player1 = 0)&(player2 = 2)){

```

```

        do Output.printString("player1 wins");
        let pl1score = pl1score + 1;
    }
    if(player1>2){
    do Output.printString("invalid");
    }
    if(player2>2){
    do Output.printString("invalid");
    }

    do Sys.wait(5000);
    }
    do Screen.clearScreen();
    if(pl1score > pl2score){
        do Output.println();
        do Output.moveCursor(8, 15);
        do Output.printString("<----- -Score Summary ----- >");
        do Output.println();
        do Output.println();
        do Output.moveCursor(11, 15);
        do Output.printString(" player1 : ");
        do Output.printInt(pl1score);
        do Output.println();
        do Output.println();
        do Output.moveCursor(14, 15); do Output.printString("
player2 : ");
        do Output.printInt(pl2score);
        do Output.println();
        do Output.println();
        do Output.moveCursor(17, 15);
    }

    if(pl2score > pl1score) {
        do Output.println();
        do Output.moveCursor(8, 15);
    }

```

```

do Output.printString(" <--- scroe summary---> ");
do Output.println();
do Output.println();
do Output.moveCursor(11, 15);
do Output.printString(" player1 : ");
do Output.printInt(pl1score);
do Output.println();
do Output.println();
do Output.moveCursor(14, 15);
do Output.printString(" player2 : ");
do Output.printInt(pl2score);
do Output.println();
do Output.println();
do Output.moveCursor(17, 15);
}

if(pl1score = pl2score) {
do Output.println();
do Output.moveCursor(8, 15);
do Output.printString("<---Score Summary--->");
do Output.println();
do Output.println();
do Output.moveCursor(11, 15);
do Output.printString(" player1 : ");
do Output.printInt(pl1score);
do Output.println();
do Output.println();
do Output.moveCursor(14, 15);
do Output.printString(" player2 : ");
do Output.printInt(pl2score );
do Output.println();
do Output.println();
do Output.moveCursor(17, 15);
do Output.printString(" Overall it is a tie.");
}

```

```
    return;  
  }  
}
```

Explanation:

We initially declare the variables player 1, player 2, sets, pl1score, pl2score, tiescore, a. The variables “pl1score”, “pl2score”, “tiescore” are zero initially using let keyword. The program asks the user to input the number of sets they want to play and the value is stored in the variable sets.

The code uses various function calls like “Output.moveCursor”, “Output.printString”,

” Keyboard.readInt”, “Screen.clearScreen”, “GUI_L.draw_L”, and “GUI_R.draw_R”. These functions are likely part of a custom library used for input/output operations and graphical user interface.

The code uses a while loop through the game sets. The loop continues as long as the number of sets is greater than or equal to the variable "a." Inside the loop, the code allows the players to input their choices for “rock” or “paper” and “scissor”.

After receiving the input from both players, the code clears the screen and displays the output in a diagram form. For clearing the screen we use a function call “Screen.clearScreen”.

Here we used a function called “sys.wait” to give some time for outputs before starting the next set of game. After completing the sets the code clears the screen again and displays

Here we use an function “output.movecursor” to specify the position on the screen with the given coordinates. and also it determines where the next output will be printed. Another function is ”Keyboard.readInt” which is used to store the character.

Based on the choices made by the players, the code determines the winner or if it's a tie, After displaying the outcome of each set, the code waits for 5 seconds using “Sys.wait(5000)”.

if the user enters wrong input then the output is invalid. mostly this code is done by if else conditions and while loops.

Once all the sets have been played, the code clears the screen again and displays the score summary based on the values of pl1score and pl2score. The summary indicates which player has the higher score or if it's a tie overall.

Class GUI_R :

```
class GUI_R {  
    function void draw_R(int player){  
        if (player = 0){  
            do Output.printString(" ROCK");  
            do Output.println();  
            do Output.moveCursor(7, 32);  
            do Output.printString(" _____");  
            do Output.println();  
            do Output.moveCursor(8, 32);  
            do Output.printString(" __(| |");  
            do Output.println();  
            do Output.moveCursor(9, 32);  
            do Output.printString("( )");  
            do Output.println();  
            do Output.moveCursor (10, 32);
```

```

do Output.printString("(_____) ");
do Output.println();
do Output.moveCursor(11, 32);
do Output.printString("(_____) ");
do Output.println();
do Output.moveCursor(12, 32);
do Output.printString("(_____)_____ ");
do Output.println();
}

if (player = 1){
do Output.printString(" PAPER");
do Output.println();
do Output.moveCursor(7, 32);
do Output.printString("_____");
do Output.println();
do Output.moveCursor(8, 32);
do Output.printString(" ____(_  _ |_  _ ");
do Output.println();
do Output.moveCursor(9, 32);
do Output.printString(" __(_  _  _  _ ) ");
do Output.println();
do Output.moveCursor(10, 32);
do Output.printString("(_____) ");
do Output.println();
do Output.moveCursor(11, 32);
do Output.printString(" (_____) ");
do Output.println();
do Output.moveCursor(12, 32);
do Output.printString(" (_____)_____ ");
do Output.println();
}

if (player = 2) {
do Output.printString(" SCISSOR");

```

```

do Output.println();
do Output.moveCursor(7, 32);
do Output.printString("      _____");
do Output.moveCursor(8, 32);
do Output.printString(" _____(____ |__");
do Output.println();
do Output.moveCursor (9, 32);
do Output.printString(" (_____) ");
do Output.println();
do Output.moveCursor(10, 32);
do Output.printString("  (_____)");
do Output.println();
do Output.moveCursor(11, 32);
do Output.printString("    (_____)");
do Output.println();
do Output.moveCursor(12, 32);
do Output.printString("      (____)_____");
do Output.println();
}

return ;
}
}

```

Explanation:

This code defines a class called GUI_R which contains a function called draw_R.

The purpose of this function is to draw a visual representation of a game character

The function draw_R takes an integer called player. there are multiple if statements that check the value of the player who is giving the input.

If player is equal to 0, it means the character is "ROCK". The code then uses the Output object to print a visual representation of the character It includes printing the text "ROCK" and drawing various lines and symbols to depict the character.

If player is equal to 1, it means the character is "PAPER". Similar to the previous case, the code prints the text "PAPER" and draws various lines and symbols to depict the character.

If player is equal to 2, it means the character is "SCISSOR". Again, the code prints the text "SCISSOR" and draws various lines and symbols to depict the character.

Each visual representation is created by a series of Output.printString and Output.println commands, which print different parts of the character at specific positions on the screen.

Class GUI_L:

```
class GUI_L {  
function void draw_L(int player){  
    if (player = 0) {  
  
        do Output.printString(" ROCK");  
        do Output.println();  
        do Output.printString(" _____");  
        do Output.println();  
        do Output.printString("    |_____)__");  
        do Output.println();  
        do Output.printString("    (_____)");  
        do Output.println();  
    }  
}
```



```

do Output.printString("      (_____)");
do Output.println();
do Output.printString("      (_____)");
do Output.println();
do Output.printString("  _____(_____)");
do Output.println();
}
if (player = 1){

do Output.printString(" PAPER");
do Output.println();
do Output.printString("  _____");
do Output.println();
do Output.printString("___|  _____");
do Output.println();
do Output.printString("      (_____)");
do Output.println();
do Output.printString("      (_____)");
do Output.println();
do Output.printString("      (_____)");
do Output.println();
do Output.printString("_____,_____");
do Output.println();
}

if (player = 2){
do Output.printString(" SCISSOR");
do Output.println();
do Output.printString("  _____");
do Output.println();
do Output.printString("___|  _____");
do Output.println();
do Output.printString("      (_____)");
do Output.println();
do Output.printString("      (_____)");

```

```

do Output.println();
do Output.printString("    (____)");
do Output.println();
do Output.printString("____(____)");
do Output.println();
}

return;
}
}

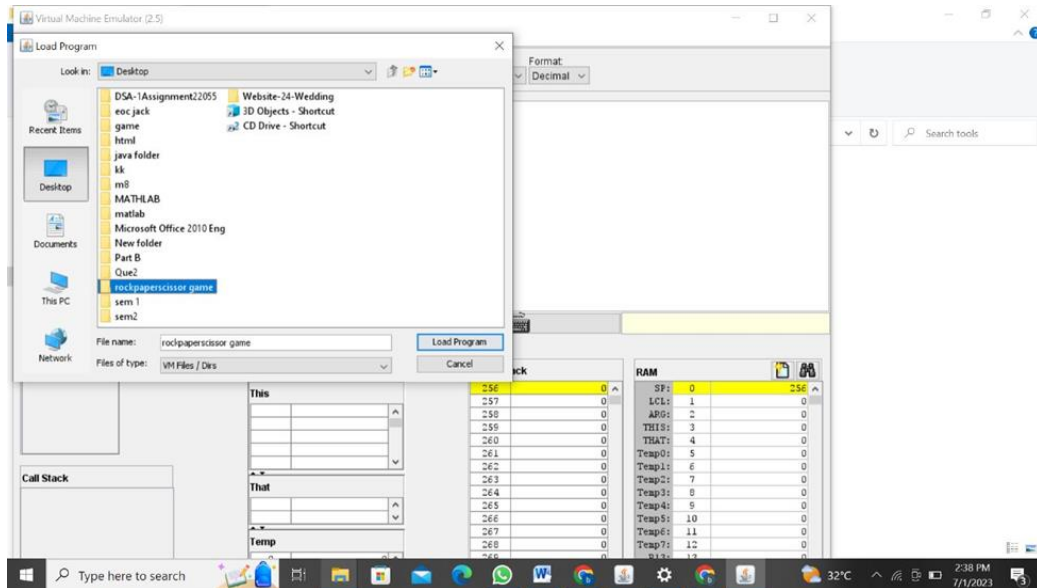
```

This code defines a class called GUI_L which contains a function called draw_L. The purpose of this function is to draw a visual representation of a game character. This code is similar to the above code .all the functions, strings, keywords that are used in the above code will also be used here in the same manner. The output diagram will be displayed in the left side of the display.

5.3 OUTPUT :

Code running –

Loading the folder contain all three class.



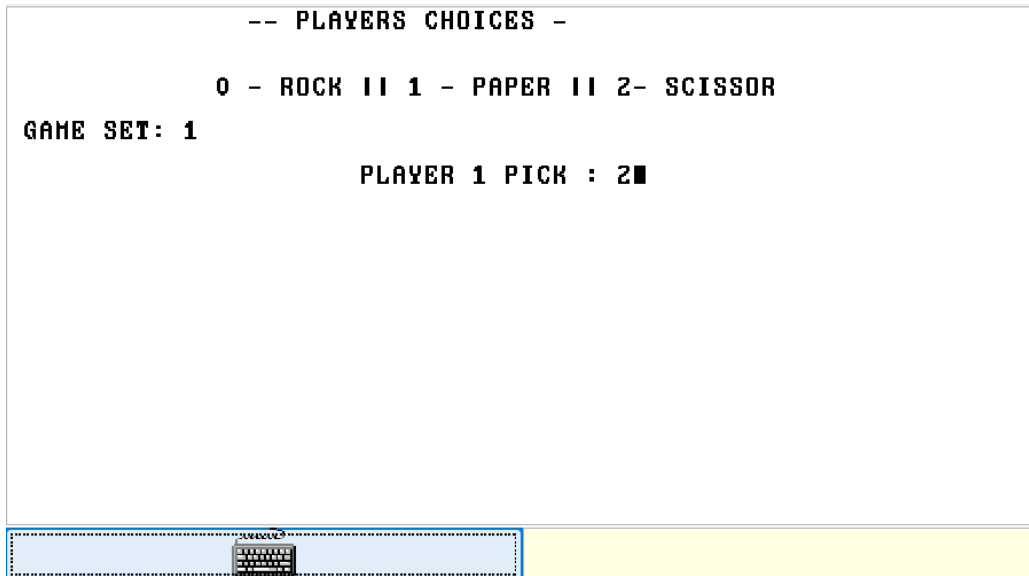
Execution- choose to play no.of sets.

ROCK-PAPER-SCISSOR

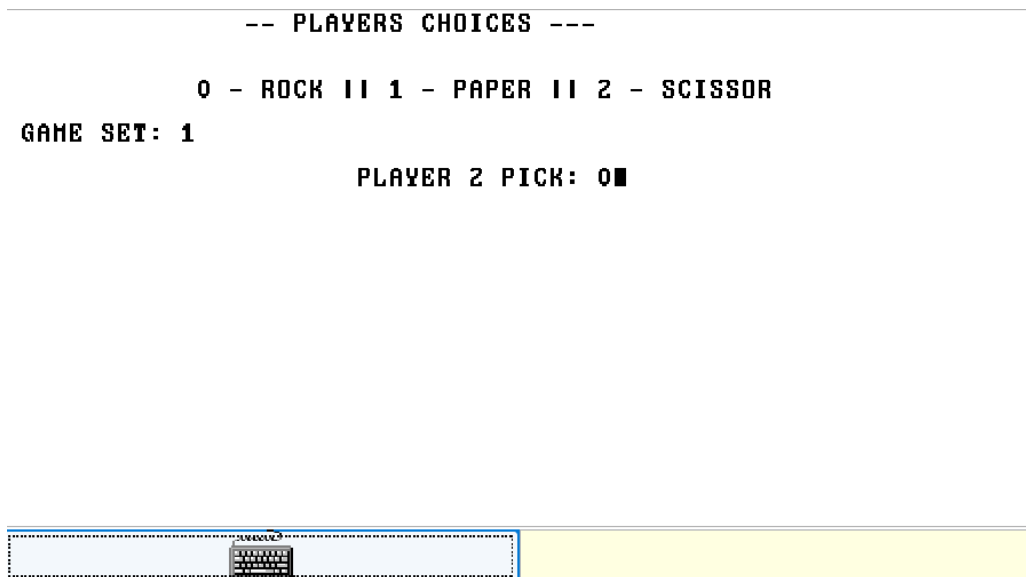
How many sets: 1



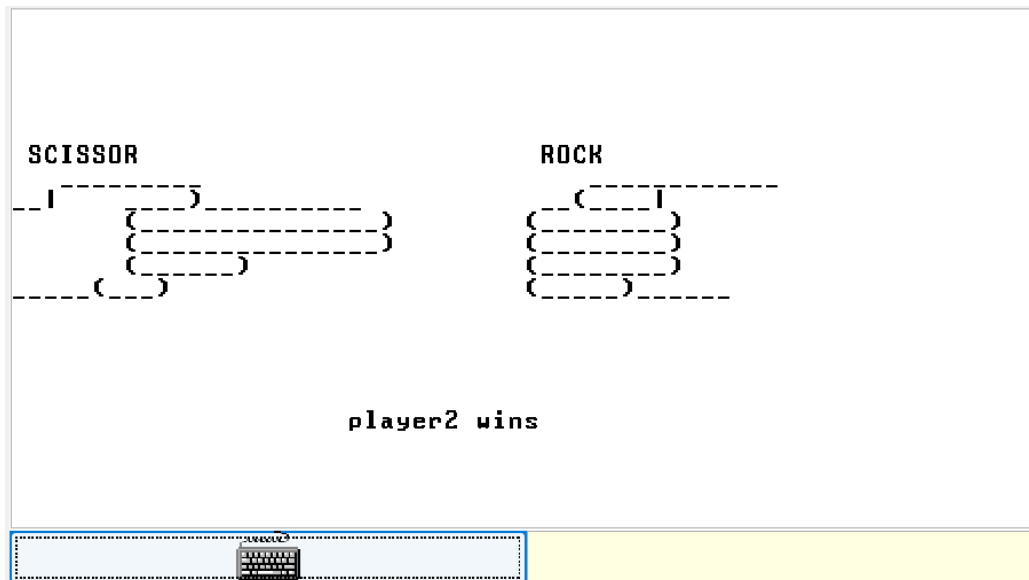
Player 1 choice-



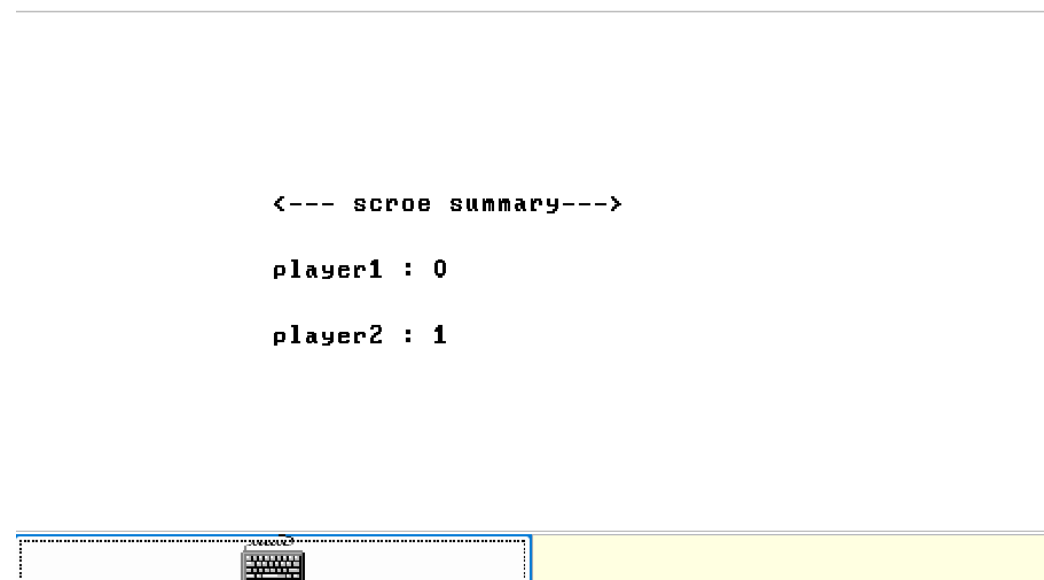
Player 2 choice-



Result-



Summary-



Invalid condition

Player 1 choice-

```
-- PLAYERS CHOICES -  
  
0 - ROCK || 1 - PAPER || 2- SCISSOR  
GAME SET: 1  
  
PLAYER 1 PICK : 2■
```

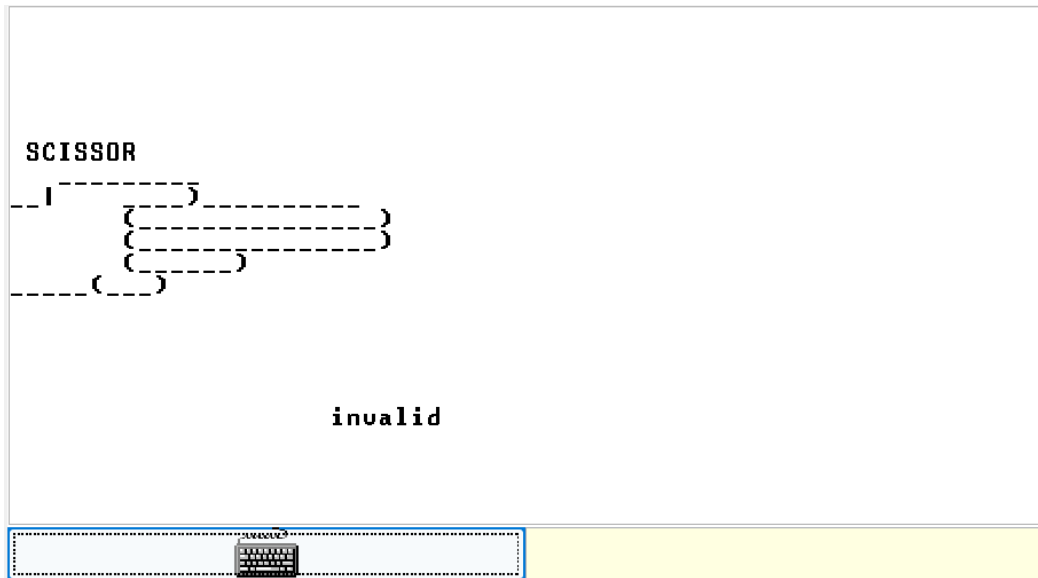


Player 2 choice-

```
-- PLAYERS CHOICES ---  
  
0 - ROCK || 1 - PAPER || 2 - SCISSOR  
GAME SET: 1  
  
PLAYER 2 PICK: 5■
```



Result -



6 CONCLUSION :

In this paper we have discussed building the Computer and the working of it. The codes have been written and implemented in HACK platform. The components of Computer such as ALU, CPU , Memory Unit have been discussed. Then, a Tic-Tac-Toe Game has been implemented using backtracking algorithm. When run in VM Emulator, output results allowing the player to play the game with the computer as well as another player. In the final part, a Flappy Bird Game is implemented by building a bird, pipes and other objects of the game.

7 REFERENCES :

1. <https://github.com/ronnieqt/FlappyBird>
2. <https://www.youtube.com/watch?v=jlLNXmi4Nmw>
3. <https://drive.google.com/file/d/1CJ1ymH6xdC5Z-Da8G0tgowaoOXq1cdbU/view>
4. https://teams.microsoft.com/_#/school/FileBrowserTabApp/General?threadId=19:KTp10l0-9-458EYLSGNpg8TIqV2LqfHHgt1_swqwL8Q1@thread.tacv2&ctx=channel
5. <https://www.youtube.com/watch?v=zg5v2rlV1tM>
6. <https://www.youtube.com/watch?v=nC1rbW2YSz0>

8 LIST OF FIGURES :

Figure No.	Figure name	Page No.
1	Hack Computer	9
2	Von Neumann architecture	12
3	Hack instruction (Part 1)	14
4	Hack instruction (Part 2)	15
5	Jump Instructions	16
6	Hack CPU	17
7	Hack character set	21
8	Computer output	32
9	Tic Tac Toe	33
10	Location values for insertion	35
11	Bitmap editor	53
12	Sprite and numbers	107
13	Flappy bird output	153
14	Loading execution no of sets	168
15	Player 1 choice	168
16	Player 2 choice	169
17	Result	169
18	Player 1 choice	170
19	Player 2 choice``	171
20	Result	171