**MAZE GAME FINAL PROJECT REPORT**

**IJAZ ULLAH CSC-18F-011 (4B)**

**ASSEMBLY LANGUAGE FINAL PROJECT REPORT**

**MISS SEHER**

**Abstract**

This document describes the design and implementation of an electronic tilting maze game in Verilog with an accelerometer-driven controller. The game renders a virtual tilting board with a maze of walls and holes. A player can tilt the controller to change the acceleration on a virtual ball. The goal of the game is to navigate the virtual ball around the maze, avoiding obstacles while guiding the ball to a target.

**Overview**

Our project was inspired by Labyrinth, a children’s game consisting of a tilting wooden maze board with holes and a metal ball. A Labyrinth game board has a configuration of obstacles such as walls and trap holes, along with a target hole. By appropriately tilting the board, the player controls how gravity affects the ball and can control the ball’s motion. The objective of Labyrinth is to navigate the ball to the target hole without falling through any trap holes. We designed and implemented an electronic version of Labyrinth using the 6.111 labkit and a prototype accelerometer-driven user interface. Our version of Labyrinth consists of a virtual ball on a virtual tilting maze board. The user controls the ball’s movement by interacting with the accelerometer driven interface.

The accelerometer-driven user interface device detects how the user wishes to tilt the virtual board, changing the effect of gravity on the virtual ball. The device consists of a two-axes accelerometer attached to a breadboard, complete with circuitry allowing the labkit to detect the current tilt of the breadboard. A state machine in the FPGA tracks the physical state of the game, updating the game state based on input from the user interface device. The game state consists of the ball’s velocity and position along with miscellaneous information such as the current level. The 6.111 labkit renders the current state of the maze board, obstacles and ball via an XVGA display.

**Design**

The design of the tilting maze game can be divided into three major components: the sensor input, the game state FSM, and the display. The sensor input modules transform input information into useful data for the game state module. The game state module uses the input information to change the current state of the game and output display information to the display modules. The display modules render the graphics for the game on the screen. The sensor input modules consist of the SPI synchronizer, physics, and debouncer modules. The game state FSM modules consist of the game state, collision detection, collision detection lookup, collision detection mask, memory interface, and memory modules. The display modules consist of the level mask, the draw circle unit, the draw unit, and the XVGA modules.

The block diagram in Figure 1 shows all of the major modules and interconnections between these modules.

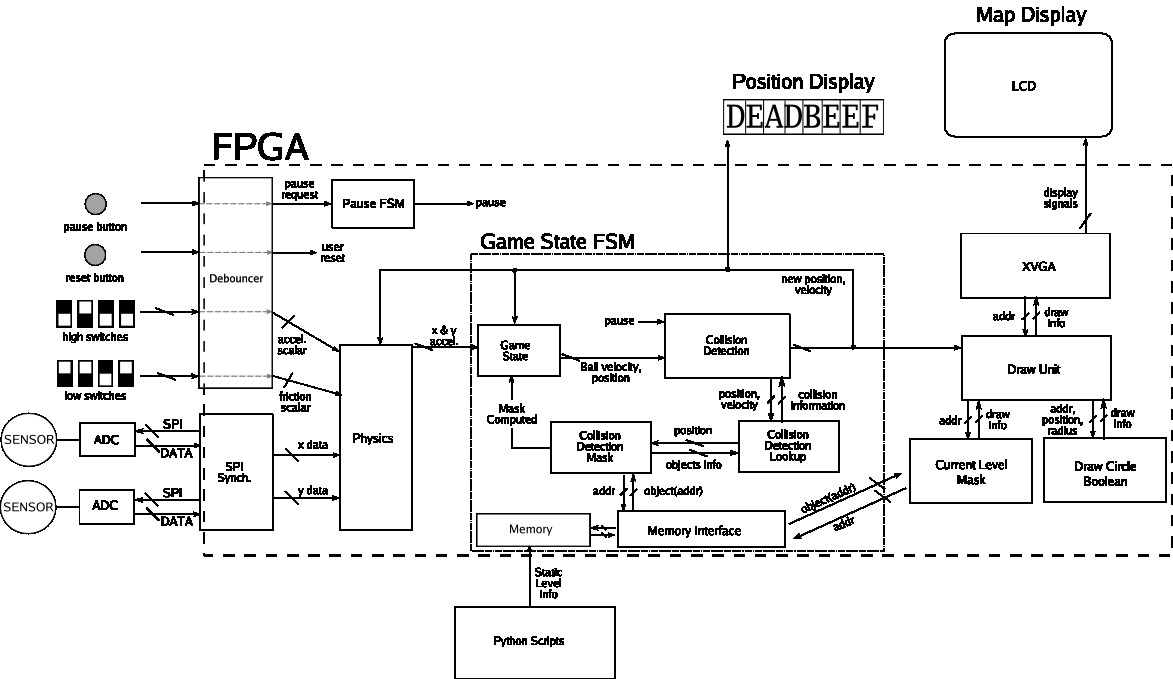


Figure 1: Block Diagram

**Sensor Input**

The sensor input consists of accelerometers and ADCs outputting data to the labkit on the order of 200 Hz. We chose this sampling frequency because the signal to noise ratio would suffer if the inputs were clocked faster. However, if the inputs were clock more slowly, then the sensors would not update fast enough to make the game playable.

**Game State FSM**

The game state FSM contains the logic to drive the movement and behavior of objects in the game. Actions in the game occur in discrete time frames; each time frame represents a fraction of a second in real time. Each loop through the game state FSM modules runs through a number of computations and collision detection steps to update the state of the game for the next time frame.

The game state FSM updates the internal game state at 8192 Hz. This frequency was chosen because the ball will never move more than one pixel given one Earth *g* of acceleration during a clock cycle with the clock running at 8192 Hz. Running this update rate any faster will increase drift error from acceleration, but running this update rate slower would cause animation quality to suffer.

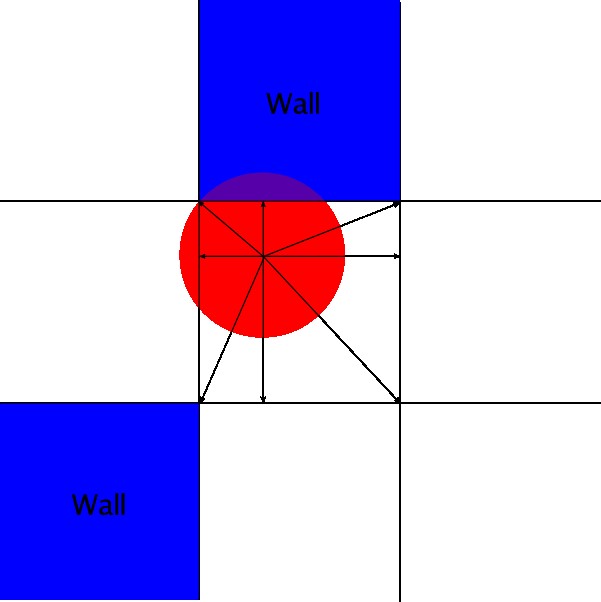


Figure 2: Collision detection diagram.

Collision Detection

The virtual game board is divided into discrete 16 pixel by 16 pixel blocks, so a level is 64 blocks wide and 48 blocks tall at 1024 by 768 resolution. Each block can be filled by a wall, a trap hole or a target hole.

Our collision detection scheme is performed on a block by block basis, as suggested by Professor Terman. On each collision detection cycle, we first determine the block that the center of the ball currently occupies. We then look up the contents of the eight blocks that immediately surround the center block as shown in Figure 2. If any part of the ball overlaps with a block that has a wall in it, then a collision has occurred with that block. We determine if the ball overlaps with one of the adjacent eight blocks by computing the distance from the center of the ball to the four corners and the sides of the center block and checking if the ball’s radius is greater than any of these distances.

One advantage of this collision detection system is that it scales well with ball size. If the size of the ball needs to be changed, a one parameter can be changed rather than reimplementing an entire module.

# 3 Module Specifications

Our system design is composed of thirteen major modules:

* Clock dividers
* Accelerometer user interface device
* SPI synchronizer
* Physics engine
* Game state module
* Collision detection module
* Collision lookup module
* Collision detection mask
* Memory Interface
* Memory
* XVGA unit
* Draw unit
* Display level mask

**SPI Synchronizer by Matt**

The SPI synchronizer module acts as an interface for the external user components, sending the latest data from the UI device to the physics unit. The SPI synchronizer module uses the five-wire SPI protocol to communicate with the ADCs, and send the latest 16-bit word output from the ADC to the physics unit.

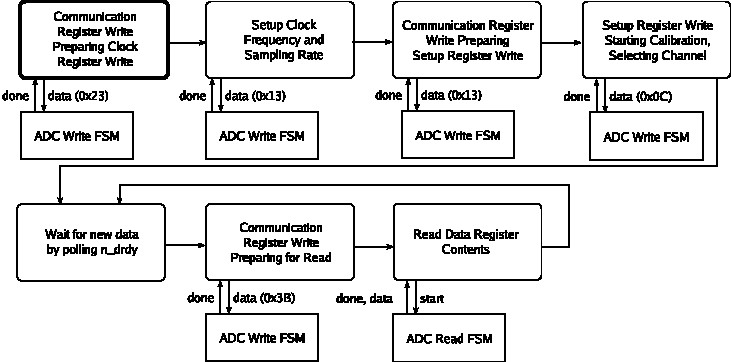


Figure 3: The SPI FSM. The FSM consists of an initialization routine followed by sampling.

**Collision Lookup Unit by Hongyi**

The collision lookup unit takes the ball’s position from the collision detection module and queries the collision detection mask to determine what collisions, if any, have occurred. The collision lookup unit contains a ten state FSM as shown in Figure 4.

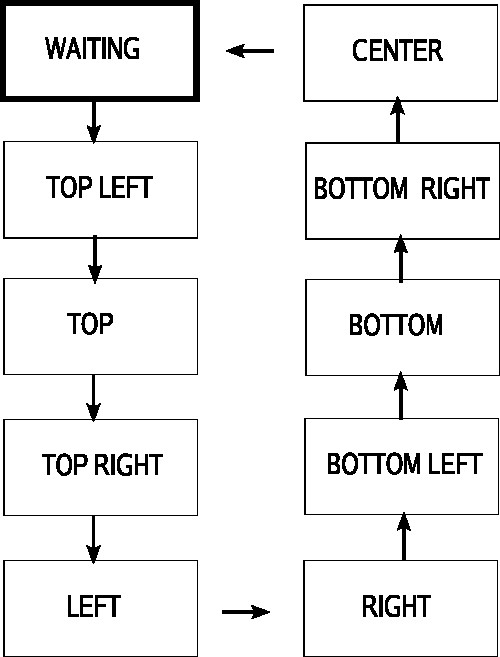


Figure 4: The Collision Lookup FSM. The FSM consists of a WAITING state followed by a series of memory lookup states.

Initially, the module is in the WAITING state, where it is waiting for new data from the collision detection unit. Once the collision detection unit has provided the ball’s updated position, the lookup unit loops through a series of nine states to query the collision detection mask for the contents of the block that holds the center of the ball and the eight blocks that surround that center block. At the end of this cycle, the module provides the results of the collision detection lookups and returns to the WAITING state.

**Collision Detection Mask by Hongyi**

The collision detection mask acts as a level cache for the main game memory. Both the collision detection logic and the display logic need to access the data in memory to perform their functions, so we implemented two masks, a collision detection mask and a level mask, to avoid interleaving memory lookups.

The collision detection mask stores the locations of all the walls and holes for one level. It contains a three-bit wide memory module with 3072 locations. Each location is associated with a specific block on the screen and contains a code indicating whether that block contains a wall, a trap hole, a target hole or nothing. The mask takes in the level number from the game state module so that when the level number changes, the mask holds mask computed low while it queries the game’s main memory to cache all the data for the next level. Once this process is complete, mask computed is held high.

**Memory Interface by Hongyi**

The memory interface takes access requests from the collision detection mask and the level mask during level transitions. The level mask has priority over the collision detection mask, so all of the level mask’s read requests are processed before the collision detection mask’s read requests.

**Memory by Matt**

The memory unit stores the locations of obstacles such as walls and holes for all levels in the game. The memory is composed of eight levels, with each level having 4096 fields. Each field is three bits, and represents a 16 by 16 block of pixels in a particular level. The three bit field corresponds to an object on the map - the codes for each map object can be found in Appendix A. In total, the memory is three bits wide with 215 locations.

**Level Generation**

A python script generated a .coe file which contained level information to be loaded into memory. The input to the python script was a string of ASCII characters corresponding to walls, trap holes or target holes.

**XVGA Unit from Lab 4**

The XVGA unit is responsible for communication between the ADV7125 and the draw unit. The XVGA unit was identical to the unit from lab four, but relies on the draw unit instead of the pong game unit.

**Draw Unit by Matt**

The draw unit is responsible for creating the pixel output the XVGA unit displays on the screen. The draw unit takes the x and y coordinates of the current ball position from the game state FSM and communicates with the current level mask to determine if any object is present at the current location. If a hole or ball exists at the current location, the draw unit communicates with the circle draw unit to determine what pixels are active to draw a circle. Based on whether any objects are present at the current location and when these objects should be drawn, the draw unit outputs what pixels should be colored and what colors these pixels should be.

**Circle Draw Unit by Matt**

The circle draw unit takes a the center and radius of a circle and returns whether or not the pixel currently being drawn is inside the circle. The circle draw unit takes at most one clock cycle to compute this information.

**Level Mask by Matt**

The level mask caches information about the topological layout of the current level. The level mask acts as a buffer for the memory to avoid draw unit latency and potential memory access conflicts with the FSM logic. Whenever the ball ends up in the target hole and moves onto a new level, the level mask communicates with the main memory to load the next level. Until the new level information is loaded from the main memory, the mask computed signal is set low so that the game state FSM suspends gameplay. The level mask takes the current pixel being drawn from the draw unit and returns whether or not an object exists at that location, along with the object type if the object does exist.

# 4 Testing and Debugging

We generally tested our code by using individual test bench waveforms to test the functionality of a single module. After each module had been tested we would integrate our code on the labkit to test the functionality of our combined components. When a bug was encountered at this stage we used the logic analyzer to track down and fix the bug.

**Frame Synchronization Bug**

One of the more difficult bugs to track down was a frame synchronization bug with the ADCs. The ADCs use the n cs signal to synchronize register writes and reads. It was incorrectly assumed that this signal’s falling edge was required to start a write to the ADC, while in reality the ADC will continue to write bits to a register as long as the n cs signal is low. The logic analyzer was of great use in tracking down this bug.

**Collision Detection Timing Issue**

Initially a simpler collision detection scheme was being used which checked only the four adjacent walls. A timing issue in the collision detection code caused the center of the ball to be checked instead of the left and right sides of the ball. This timing issue, coupled with a display bug causing the ball’s center to be offset by a pixel, was difficult to track down due to the seemingly random nature of when collisions were being output. The timing bug was finally discovered when the waveforms for the collision detection unit were drawn with a closer look at the memory access latency.

**Inferred Bit Flips**

During the upgrade of the collision detection algorithm the ball would often stop seven or eight pixels from the wall. The error was caused by the synthesis tools behaving differently than expected. We compute a five-bit value by taking the bitwise complement of a four-bit value and adding one. What happened, however, was that the most significant bit of the five-bit value was always flipped to one, which caused an error in our collision detection calculations. We hypothesize that this was caused by Verilog casting the four-bit value to five-bit value before doing the bitwise complement. This bug was discovered after we ran the signals through the logic analyzer and saw some erroneous values.

# 5 Conclusion

In summary, we built an electronic tilting maze game based on the children’s game Labyrinth. We used an accelerometer control device to feed input into the labkit via an SPI interface. The labkit maintained the state of the game and drove the movement and behavior of the game. The labkit also rendered the graphics using an XVGA interface. A Python script was used to generate levels for the game.

Our original design goals were to avoid complexity, to make our design scalable and to make our design easily mutable. Since we were able to add extensions easily to our project, we believe that we were fairly successful in meeting these objectives. For example, we implemented a collision detection scheme that was cleaner and far more efficient than our original solution. Due to the flexible modularization of our design, changes were required in only one module to upgrade the game from the old collision detection method to the new, superior one. We were also able to add pause functionality with few changes to existing code. Finally, because we reserved a number of unused memory codes, we could easily add new objects to the game. This can be accomplished with small changes to the collision detection module and the display logic.

We had hoped to eventually implement this design on a smaller FPGA, creating a unit that could simply be attached to an LCD, but after finishing the bulk of the project, we focused on implemented digital design changes and additions.

**Acknowledgments**

We would like to thank Professor Terman, Professor Chuang, and our TA Jae Lee for their expertise and help with our project. We would also like to thank Analog Devices for their generous donation of equipment for our project.

# A Memory Field Codes

|  |  |
| --- | --- |
| **Code** | **Object** |
| 000 | Nothing |
| 001 | Target Hole |
| 010 | Trap Hole |
| 011 | Wall |
| 100 – 111 | Reserved for Future Use |

**Code**

**title project(EXE)**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MACRO TO SCROLL SPECIFIC ROWS to SPECIFIC \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**SetResolution macro srow,scol,erow,ecol,attrib**

**mov ah,06h**

**mov al,0h**

**mov bh,attrib**

**mov ch,srow**

**mov cl,scol**

**mov dh,erow**

**mov dl,ecol**

**int 10h**

**endm**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MACRO TO Copy two Elements into another \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**copy macro row1,oldrow,col1,oldcol**

**mov bl,row1**

**mov oldrow,bl**

**mov bl,col1**

**mov oldcol,bl**

**endm**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MACRO TO Print a String \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**print macro string**

**mov ah,09h**

**mov dx,offset string**

**int 21h**

**endm**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MACRO TO write specific element with specific attributes\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**write macro elem,attrib**

**mov ah,09h**

**mov bh,0h**

**mov cx,01h**

**mov bl,attrib**

**mov al,elem**

**int 10h**

**endm**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MACRO TO SET THE CURSOR POSITION at SPECIFIC Row and column \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**setCursor macro row,col**

**mov ah,02**

**mov bh,0h**

**mov dh,row**

**mov dl,col**

**int 10h**

**endm**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**.model small ;define the model of the programe to small**

**.stack 64h ;reserve 64 h places reserve in Stack for variables**

**.data**

**row1 db 23 ;row of player**

**col1 db 37 ;column of player**

**oldrow db 23 ;old row of player**

**oldcol db 37 ;old column of player**

**youwin db 'Well done man you have won the game $' ;string to print the indicator of win**

**intro db ' Welcome to the MAZE Game $' ;string to print the starting text**

**spoint db 'Pink point Indicate the starting point $' ;string to print the starting text**

**epoint db 'Red Point indicate the Ending Point $' ;string to print the starting text**

**press db 'Press Any Key to Continue $' ;string to print the starting text**

**ulabel db '^ $'**

**maze db ' Maze Game $'**

**lrlabel db 'Controls are : < + > $'**

**dlabel db 'v $'**

**llabel db 'OOps You Hit the ememy $'**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**;\*\*\*\*\*\*\*\*\*\*\*\*\* rows and columns of enemy 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**e1col db 49**

**e1row db 3**

**e1oldcol db 49**

**e1oldrow db 3**

**;\*\*\*\*\*\*\*\*\*\*\*\*\* rows and columns of enemy 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**e2row db 15**

**e2col db 25**

**e2oldrow db 15**

**e2oldcol db 25**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**.code**

**main proc far**

**jmp gamestart**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* up down left and right labels for player \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**go\_up:**

**dec row1**

**jmp back**

**go\_down:**

**inc row1**

**jmp back**

**go\_left:**

**dec col1**

**jmp back**

**go\_right:**

**inc col1**

**jmp back**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**gamestart: ;label indicating that the game is start**

**mov ax,@data**

**mov ds,ax**

**mov ch , 32 ;hides the cursor**

**mov ah , 1**

**int 10h**

**call IniFrame**

**call GameFrame**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**;\*\*\*\*\*\*\*\*\*\*\*\*\*write the @ at starting point**

**setCursor 23,37**

**write '@',0A0h**

**alive:**

**call ENEMY ;call the enemy for the movements of ememies**

**copy row1,oldrow,col1,oldcol**

**mov ah,08h ;read the input from the user**

**int 21h**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**;\*\*\*\*\*\*\*\*\*\*\*\*\* comparisn for the player movement with arrow keys ascii\*\*\***

**cmp al,75 ;left**

**je go\_left**

**cmp al,72 ;up**

**je go\_up**

**cmp al, 77 ;right**

**je go\_right**

**cmp al,80 ;down**

**je go\_down**

**CMP al,27**

**je exit**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**back: ;label after setting up the player row and column**

**CALL GameFrame ;draw the game frame**

**setCursor row1,col1 ;set cursor at row1 and column1**

**call GetAttribute ;get the attribute and store it in ah**

**cmp ah,0CCh ; compare the ah with the red colour which indicate the winning point**

**je win**

**cmp ah,0E0h ;compare the ah with the 0E0h which indicate the enemy**

**je lose**

**cmp ah,099h ;compare the ah with 099h which indicate the colour of the walls**

**jne move\_cursor ;jump to move if the colour is not blue**

**; cmp ah,099h**

**je go\_back ;else jump to go back to same position**

**compare:**

**cmp al,'+' ;compare the + the hotkey to stop the loop**

**jne alive**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* label indicatiing the winning goal\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**win:**

**SetResolution 0,0,24,79,07h**

**setCursor 10,23**

**print youwin**

**mov ah,01h**

**int 21h**

**jmp exit**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*label indicatiing the loosing goal\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**lose:**

**SetResolution 0,0,24,79,07h**

**setCursor 10,23**

**print llabel**

**mov ah,01h**

**int 21h**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*label indicatiing the exit point\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**exit:**

**SetResolution 0,0,24,79,07h**

**mov ax,4c00h**

**int 21h**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**move\_cursor: ;label to write @ at the cursor positin**

**write '@',0A0h**

**jmp compare**

**go\_back: ;if there is a wal then go back to the previus position**

**mov bl,oldrow**

**mov row1,bl**

**mov bl,oldcol**

**mov col1,bl**

**setCursor row1,col1**

**jmp move\_cursor**

**main endp**

**IniFrame proc near ;procedure to set the front page of the screen**

**SetResolution 0,0,24,79,55h ;to clear the screen**

**SetResolution 7,19,15,60,0eh**

**setCursor 8,21**

**print intro**

**setCursor 9,21**

**print spoint**

**setCursor 10,21**

**print epoint**

**setCursor 11,46**

**print ulabel**

**setCursor 12,21**

**print lrlabel**

**setCursor 13,46**

**print dlabel**

**setCursor 14,25**

**print press**

**mov ah,01h**

**int 21h**

**ret**

**IniFrame endp**

**GameFrame proc near ;procedure to set the frame of the game**

**SetResolution 0,0,24,79,77h ;to clear the screen**

**SetResolution 0,0,1,79,99h**

**SetResolution 0,0,23,2,99h**

**SetResolution 23,0,24,79,99h**

**SetResolution 1,77,24,79,99h**

**SetResolution 23,36,23,38,0DDh**

**SetResolution 10,29,16,50,99h**

**SetResolution 13,43,16,44,77h**

**SetResolution 13,33,13,44,77h**

**SetResolution 14,33,14,34,077h**

**SetResolution 15,33,15,34,0CCh**

**SetResolution 17,39,22,42,099h**

**SetResolution 19,9,20,39,099h**

**SetResolution 4,6,20,9,099h**

**SetResolution 4,14,5,40,099h**

**SetResolution 2,33,5,36,099h**

**SetResolution 8,14,16,16,099h**

**SetResolution 15,14,16,24,099h**

**SetResolution 4,40,7,43,099h**

**SetResolution 6,21,14,24,099h**

**SetResolution 4,48,9,50,099h**

**SetResolution 6,51,7,63,099h**

**SetResolution 10,55,15,57,099h**

**SetResolution 6,68,20,71,099h**

**SetResolution 18,45,20,71,099h**

**SetResolution 19,45,20,71,099h**

**SetResolution 20,45,23,47,099h**

**SetResolution 12,57,13,68,099h**

**SetResolution 11,32,11,48,03eh**

**setCursor 11,33**

**print maze**

**setCursor e1row,e1col**

**write 'E',0E0h**

**setCursor e2row,e2col**

**write 'E',0E0h**

**ret**

**GameFrame endp**

**GetAttribute proc near ;procedure to get the attribute of the cursor position**

**mov ah,08h**

**mov bh,0h**

**int 10h**

**ret**

**GetAttribute endP**

**random proc near ;procedure to get a random number**

**MOV AH, 00h ; interrupts to get system time**

**INT 1AH ; CX:DX now hold number of clock ticks since midnight**

**mov ax, dx**

**xor dx, dx**

**mov cx, 4**

**div cx ; here dx contains the remainder of the division - from 0 to 9**

**add dl, '0' ; to ascii from '0' to '9'**

**ret**

**random endP**

**ENEMY proc near ;procedure to control the movements of the ememies**

**jmp start1 ;jump to go to the starting instruction of the procedure**

**;labels for enemy 1 movements**

**e1\_up:**

**dec e1row**

**setCursor e1row,e1col**

**jmp e1done**

**e1\_down:**

**inc e1row**

**setCursor e1row,e1col**

**jmp e1done**

**e1move:**

**jmp e1moved**

**start1:**

**copy e1row,e1oldrow,e1col,e1oldcol**

**call random ;call the procedure random**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*set the position acccording to the random number\*\*\*\*\*\*;**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;**

**cmp dl,'0'**

**je e1\_up**

**cmp dl,'1'**

**je e1\_down**

**cmp dl,'2'**

**je e1\_right**

**cmp dl,'3'**

**je e1\_left**

**e1done:**

**call GetAttribute**

**cmp ah,0B0h**

**je e1\_goback**

**cmp ah,099h**

**jne e1move**

**cmp ah,099h**

**je e1\_goback**

**e1moved:**

**jmp enemy1Ind**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* up down left and right labels for ememy 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**e1\_left:**

**dec e1col**

**setCursor e1row,e1col**

**jmp e1done**

**e1\_right:**

**inc e1row**

**setCursor e1row,e1col**

**jmp e1done**

**e1\_goback:**

**copy e1oldrow,e1row,e1oldcol,e1col**

**setCursor e1row,e1col**

**jmp e1move**

**copy e2row,e2oldrow,e2col,e2oldcol**

**enemy1Ind:**

**call random**

**cmp dl,'0'**

**je e2\_up**

**cmp dl,'1'**

**je e2\_down**

**cmp dl,'2'**

**je e2\_right**

**cmp dl,'3'**

**je e2\_left**

**e2done:**

**call GetAttribute**

**cmp ah,099h**

**jne e2move**

**cmp ah,099h**

**je e2\_goback**

**cmp ah,0B0h**

**je e2\_goback**

**e2moved:**

**ret**

**;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* up down left and right labels for enemy 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**e2\_up:**

**dec e2row**

**setCursor e2row,e2col**

**jmp e2done**

**e2\_down:**

**inc e2row**

**setCursor e2row,e2col**

**jmp e2done**

**e2\_left:**

**dec e2col**

**setCursor e2row,e2col**

**jmp e2done**

**e2\_right:**

**inc e2row**

**setCursor e2row,e2col**

**jmp e2done**

**e2\_goback:**

**copy e2oldrow,e2row,e2oldcol,e2col ;use the macro copy to copy the values**

**setCursor e2row,e2col**

**jmp e2move**

**jmp e2done**

**e2move:**

**;write 'E',0E0h**

**jmp e2moved**

**ENEMY endp**

**end main**