
Implementation of the Snake Game in C using RIPES

Practice 2

enrique.rios@iteso.mx / Ex: 750259,
yael.morales@iteso.mx / Ex: 751669

2024-12-01

Contents

1	Introduction	1
1.1	Purpouse	1
1.2	Github repository	1
2	Development of the snake game in ripes	2
2.1	Flowchart	3
2.2	C code	4
3	Cache Testing	8
3.1	Direct mapping with 4 lines	8
3.2	Associative with 2 lines and 2 ways	10
3.3	Fully associative with 4 ways	12
3.4	Direct mapping with 16 lines	14
3.5	Associative with 4 lines and 4 ways	16
3.6	Fully associative with 16 ways	18
4	CONCLUSION	20

1 Introduction

1.1 Purpose

In the realm of computer architecture, system performance is influenced by cache memory configuration and efficiency. This practice aims to analyze the impact that different cache configurations have on the performance of a system. To do this, the classic Snake Game will be implemented in C language, using the RIPES simulator as a development and testing environment.

The game display will be carried out through a matrix of LEDs, while the player's movement control will be managed through a D-Pad. Through this implementation, we seek to measure and compare performance under different cache memory configurations, identifying how these affect key aspects such as response time and general efficiency of the program.

This activity combines programming fundamentals, hardware simulation, and memory systems analysis, offering a comprehensive experience to understand the impact of memory design on practical applications.

1.2 Github repository

[Click here] https://github.com/Hoverpi/C_Snake

2 Development of the snake game in ripes

2.1 Flowchart

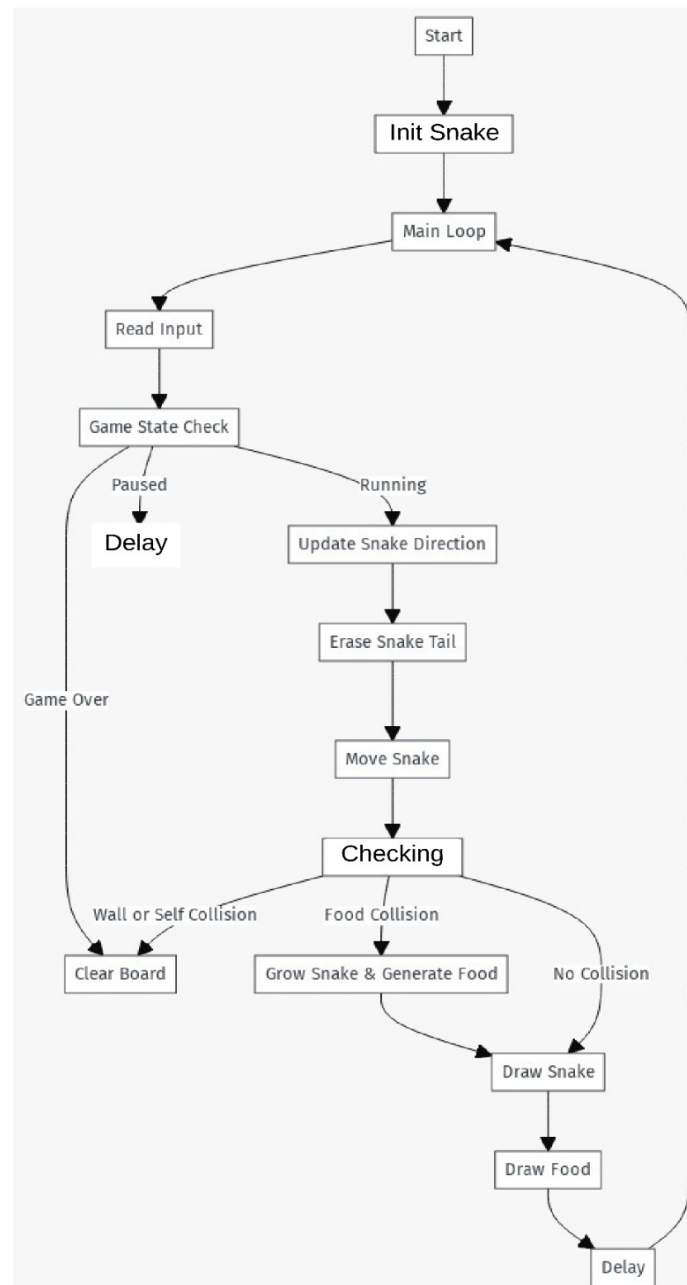


Figure 2.1: Flowchart

2.2 C code

```
#include "ripes_system.h"
#include <stdlib.h>

// Ripes I/O Macros
volatile unsigned int *led_base = (volatile unsigned int *)LED_MATRIX_0_BASE;
volatile unsigned int *switch_base = (volatile unsigned int *)SWITCHES_0_BASE;
volatile unsigned int *d_pad_up = (volatile unsigned int *)D_PAD_0_UP;
volatile unsigned int *d_pad_down = (volatile unsigned int *)D_PAD_0_DOWN;
volatile unsigned int *d_pad_left = (volatile unsigned int *)D_PAD_0_LEFT;
volatile unsigned int *d_pad_right = (volatile unsigned int *)D_PAD_0_RIGHT;

// Colors configuration
#define SNAKE_COLOR 0xFF0000 // Green
#define FOOD_COLOR 0x00FF00 // Red
#define BACKGROUND_COLOR 0xFFFFFF // White
#define BORDER_COLOR 0x000000 // Black

// Game configuration
#define MAX_SNAKE_LENGTH 50
#define PIXEL_SIZE 2
#define WAIT_DELAY 25
#define START_X 10
#define START_Y 10
```

- **Hardware Addresses:** Defines pointers (led_base, switch_base, etc.) to interact with the LED matrix, switches, and directional buttons. These pointers map to memory-mapped I/O for the embedded system.
- **Colors:** Specifies RGB values for snake, food, background, and border.
- **Game Configurations:** Sets parameters like the maximum snake length, size of each “pixel” in the LED grid, delay between movements, and the snake’s starting position.

```
typedef enum { GAME_OVER, RUNNING, PAUSED } GameState;
typedef enum { UP, DOWN, LEFT, RIGHT } Direction;

typedef struct {
    unsigned int x, y;
} Position;

typedef struct {
    Position segments[MAX_SNAKE_LENGTH];
    int length;
    Direction direction;
} Snake;

typedef struct {
    Position position;
} Food;
```

- Enums:
 - **GameState** tracks the current state of the game: GAME_OVER, RUNNING, or PAUSED.
 - **Direction** defines possible movement directions: UP, DOWN, LEFT, RIGHT.
- Structs:
 - **Position:** Stores the x and y coordinates in the grid.
 - **Snake:** Contains the snake's segments (positions), length, and current direction.
 - **Food:** Stores the position of the food.

```
void initSnake(Snake *snake);
void drawSnake(const Snake *snake);
void eraseSnakeTail(const Snake *snake);
void moveSnake(Snake *snake);
void generateFood(Snake *snake, Food *food);
void drawFood(const Food *food);
int checkWallCollision(const Snake *snake);
int checkSelfCollision(const Snake *snake);
int checkFoodCollision(const Snake *snake, const Food *food);
void clearBoard();
void delay(int milliseconds);
```

- These function declarations provide the core functionality:
- **Initialization:** initSnake sets up the snake at the start of the game.
- **Drawing and Erasing:** drawSnake, eraseSnakeTail, drawFood, and clearBoard handle visuals.
- **Movement:** moveSnake updates the snake's position.
- **Collision Checks:** Functions detect collisions with walls, itself, or food.
- **Delays:** delay pauses execution to control the game's speed.

```
void main() {
    Snake snake;
    Food food;
    GameState game_state = GAME_OVER;
    int score = 0;
    int previous_switch_state = *switch_base & 0x01;

    initSnake(&snake);
    generateFood(&snake, &food);
    clearBoard();
    drawSnake(&snake);
    drawFood(&food);
```

Initialization: Creates snake and food objects, sets the initial game state to GAME_OVER, and resets the score. **Game Setup:** Calls initialization functions to prepare the snake, food, and LED grid.

```
while (1) {
    int current_switch_state = *switch_base & 0x03;

    if (current_switch_state == 0x01 && game_state == GAME_OVER) {
        game_state = RUNNING;
        score = 0;
        initSnake(&snake);
        generateFood(&snake, &food);
        clearBoard();
        drawSnake(&snake);
        drawFood(&food);
    }
    else if (current_switch_state == 0x02) {
        game_state = GAME_OVER;
        clearBoard();
        initSnake(&snake);
        generateFood(&snake, &food);
        drawSnake(&snake);
        drawFood(&food);
        score = 0;
        delay(200);
    }
}
```

- Switch Handling:

- Starts a **new game** when switch 0 is pressed (current_switch_state == 0x01).
- **Ends** the game and **resets** it when switch 1 is pressed (current_switch_state == 0x02).

```
if (game_state == GAME_OVER || game_state == PAUSED) {
    delay(100);
    continue;
}

if (*d_pad_up && snake.direction != DOWN) {
    snake.direction = UP;
} else if (*d_pad_down && snake.direction != UP) {
    snake.direction = DOWN;
} else if (*d_pad_left && snake.direction != RIGHT) {
    snake.direction = LEFT;
} else if (*d_pad_right && snake.direction != LEFT) {
    snake.direction = RIGHT;
}
```

Game State Check: Skips further logic if the game is over or paused. **Direction Input:** Updates the snake's direction based on the directional pad, avoiding moves that reverse the current direction.

```
eraseSnakeTail(&snake);
moveSnake(&snake);
```



```
    if (checkWallCollision(&snake) || checkSelfCollision(&snake)) {
        game_state = GAME_OVER;
        continue;
    }

    if (checkFoodCollision(&snake, &food)) {
        if (snake.length < MAX_SNAKE_LENGTH) {
            snake.length++;
        }
        score++;
        generateFood(&snake, &food);
    }

    drawSnake(&snake);
    drawFood(&food);

    delay(WAIT_DELAY);
}
```



- **Movement and Erasing:** Moves the snake forward and erases its tail to simulate movement.
- **Collision Checks:** Ends the game if the snake hits a wall or itself.
- **Food Handling:** Increases the snake's length and generates new food when eaten.
- **Redraw and Delay:** Updates the LED grid and pauses briefly to control the game's speed.

The remaining functions (initSnake, drawSnake, eraseSnakeTail, moveSnake, generateFood, checkWallCollision, checkSelfCollision, checkFoodCollision, clearBoard, delay) implement the logic described in the explanation above.

3 Cache Testing

3.1 Direct mapping with 4 lines

Cache configuration:

Preset:  

2 ^N Lines:	<input type="text" value="2"/>	Repl. policy:	<input type="text" value="LRU"/>
2 ^N Ways:	<input type="text" value="0"/>	Wr. hit:	<input type="text" value="Write-back"/>
2 ^N Words/Line:	<input type="text" value="2"/>	Wr. miss:	<input type="text" value="Write allocate"/>

Plot configuration:


Numerator

Denominator

☒ Ratio

☒ Moving avg.

Statistics:

Size (bits): 

Hit rate: Writebacks:

Hits: Misses:

Figure 3.1: Direct mapping with 4 lines config

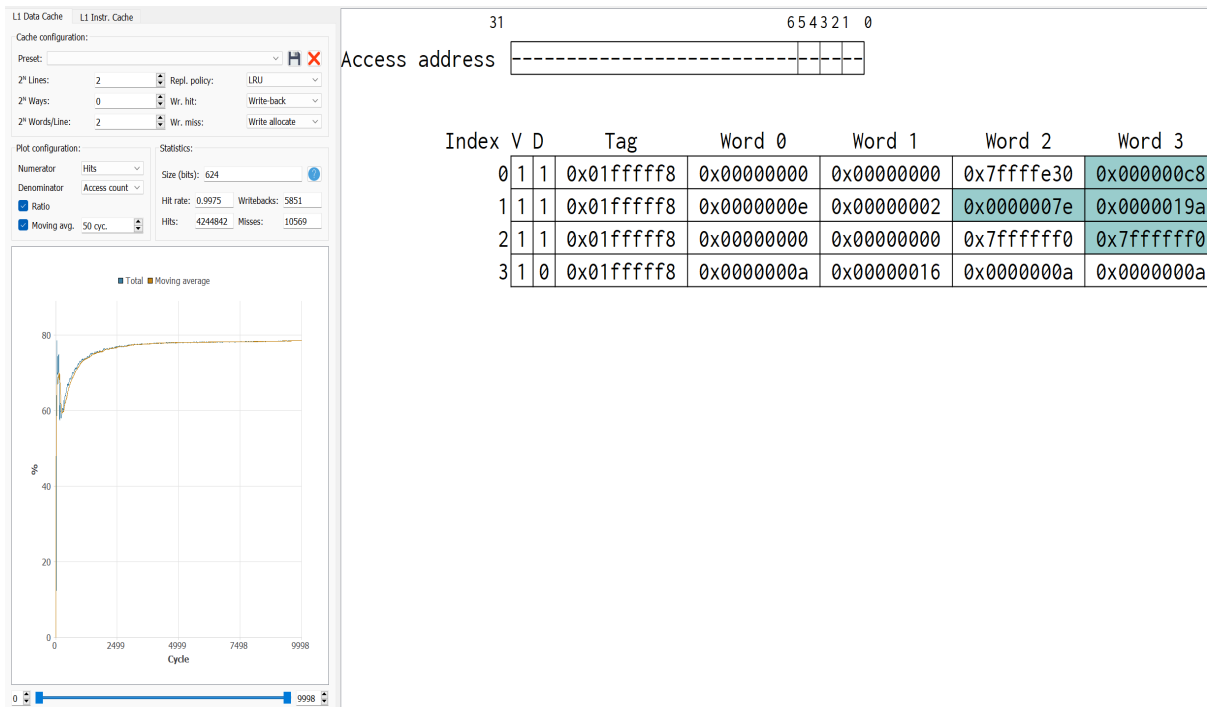






Figure 3.2: Direct mapping with 4 lines graph



HIT RATE: 0.9975



3.2 Associative with 2 lines and 2 ways

Cache configuration:

Preset:  

2^N Lines:   Repl. policy:

2^N Ways:   Wr. hit:



2^N Words/Line:   Wr. miss:

Plot configuration:


Numerator

Denominator

☒ Ratio

☒ Moving avg.  

Statistics:

Size (bits): 

Hit rate: Writebacks:



Hits: Misses:



Figure 3.3: Associative with 2 lines and 2 ways config







3.3 Fully associative with 4 ways

Cache configuration:

Preset:  

2^N Lines:   Repl. policy:

2^N Ways:   Wr. hit:



2^N Words/Line:   Wr. miss:

Plot configuration:


Numerator

Denominator

☒ Ratio

☒ Moving avg.  

Statistics:

Size (bits): 

Hit rate: Writebacks:



Hits: Misses:



Figure 3.5: Fully associative with 4 ways config



**HIT RATE: 0.9982**



3.4 Direct mapping with 16 lines

Cache configuration:

Preset:  

2^N Lines:   Repl. policy:

2^N Ways:   Wr. hit:



2^N Words/Line:   Wr. miss:

Plot configuration:


Numerator

Denominator

☒ Ratio

☒ Moving avg.  

Statistics:

Size (bits): 

Hit rate: Writebacks:

Hits: Misses:

Figure 3.7: Direct mapping with 16 lines config

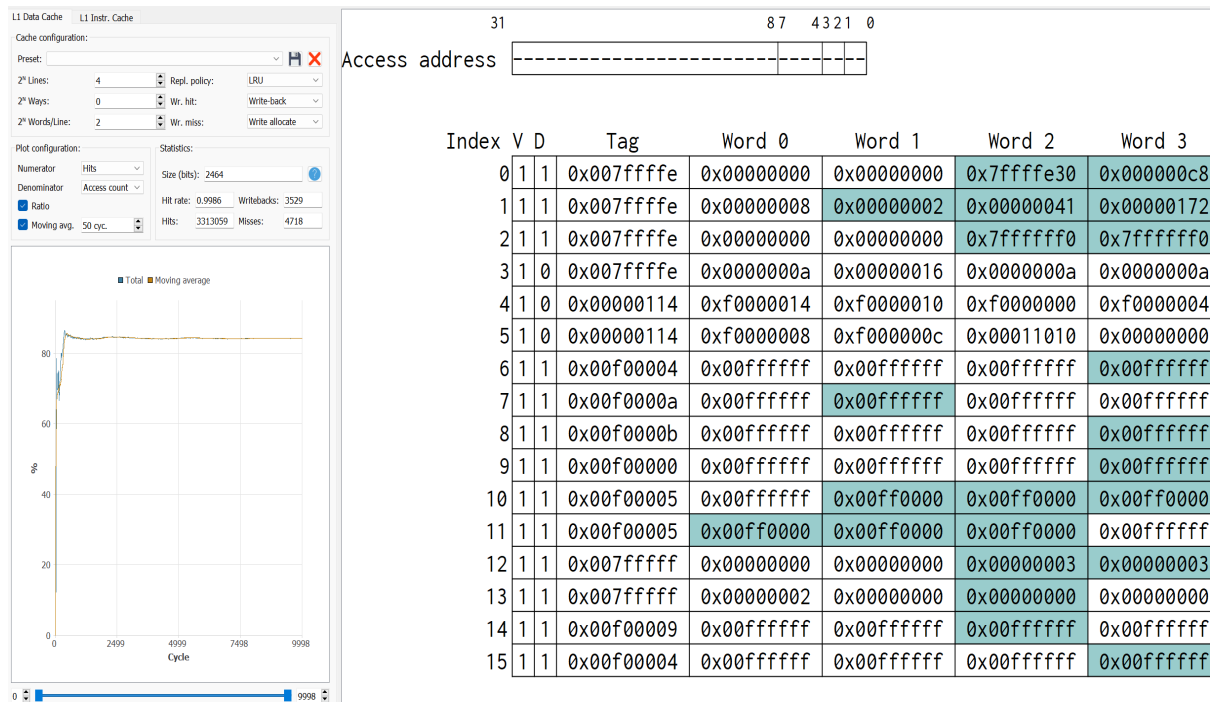






Figure 3.8: Direct mapping with 16 lines graph



HIT RATE: 0.9986



3.5 Associative with 4 lines and 4 ways

Cache configuration:

Preset:  

2^N Lines:   Repl. policy:

2^N Ways:   Wr. hit:



2^N Words/Line:   Wr. miss:

Plot configuration:


Numerator

Denominator

☒ Ratio

☒ Moving avg.  

Statistics:

Size (bits): 

Hit rate: Writebacks:

Hits: Misses:



Figure 3.9: Associative with 4 lines and 4 ways config



Figure 3.10: Associative with 4 lines and 4 ways



HIT RATE: 0.999



3.6 Fully associative with 16 ways

Cache configuration:

Preset:  

2^N Lines:   Repl. policy:

2^N Ways:   Wr. hit:



2^N Words/Line:   Wr. miss:

Plot configuration:


Numerator

Denominator

☒ Ratio

☒ Moving avg.  

Statistics:

Size (bits): 

Hit rate: Writebacks:

Hits: Misses:

Figure 3.11: Fully associative with 16 ways config

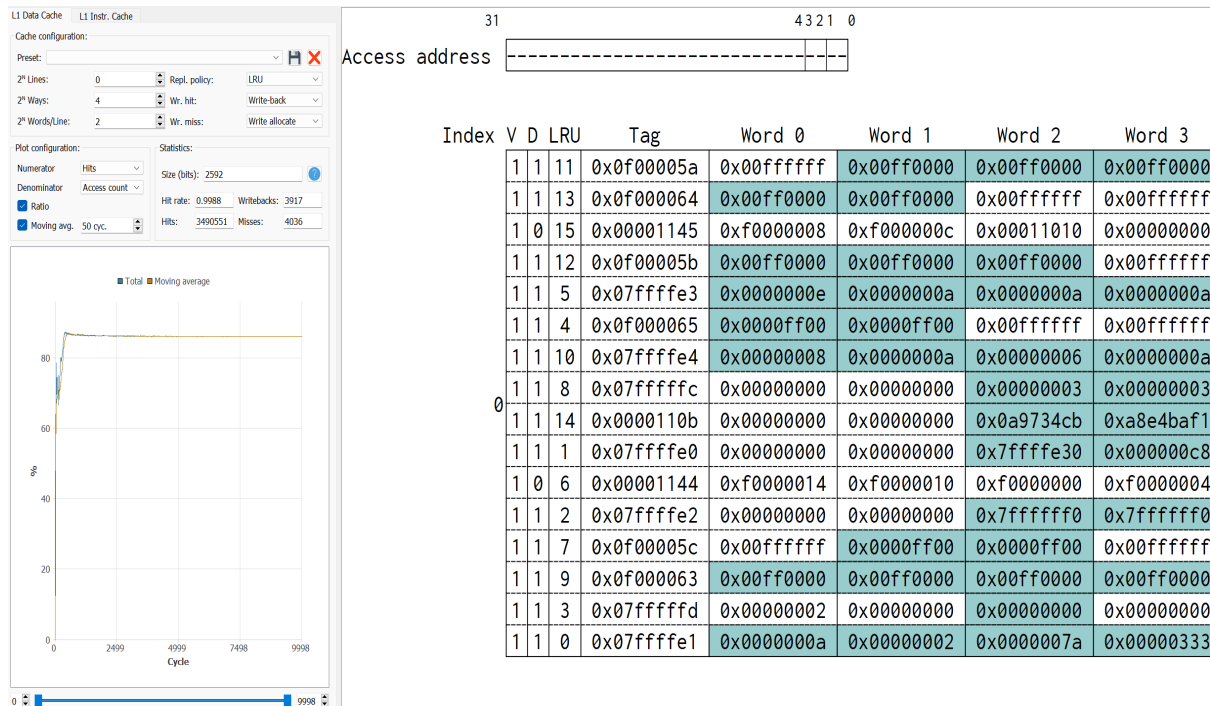


Figure 3.12: Fully associative with 16 ways graph

HIT RATE: 0.9988**BEST HIT RATE:** 0.999

4 CONCLUSION

José Enrique Rios Gómez: This code was built as a playground to dive into different cache mapping techniques and test how they work in practice. It simulates scenarios like direct mapping with varying line counts, associative caches with specific sets and ways, and even fully associative designs. The goal was to understand how these configurations behave and how the snake game's performance interacts with each setup.

It's like giving the same puzzle to different kinds of storage and seeing how each one solves it. Whether it's a simple 4-line direct map or a beefy 16-way fully associative setup, this code helped bring the theory to life. Honestly, it was pretty cool to see how the different mappings influenced the behavior of something as simple as moving pixels on a grid. This hands-on approach made those dry cache mapping concepts way more relatable.

Yael Salvador Morales Renteria: By working with this code, I have learned about C programming with hardware interaction. First, I understood how to use pointers to manipulate hardware registers. This was crucial for controlling the system's LEDs and buttons, which helped me directly connect the program logic to the physical outputs. Another thing I learned was looking at data in an LED array and the importance of considering performance and hardware limits such as LED array size and maximum snake length.