MINISTRY OF SCIENCE AND HIGHER EDUCATION OF THE RUSSIAN FEDERATION
FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION
"NOVOSIBIRSK NATIONAL RESEARCH UNIVERSITY
STATE UNIVERSITY"
(NOVOSIBIRSK STATE UNIVERSITY, NSU)

15.03.06 - Mechatronics and Robotics
Focus (profile): Artificial Intelligence

EXPLANATORY NOTE

Job topic: **‘THE FALL’**

Roman Tsaregorodtsev, 23930
Aleksey Spirkin, 23930

Novosibirsk
2024

TABLE OF CONTENTS

1 TERMS AND ABBREVIATIONS

CdM-8 Mark 5 Processor	Coco-de-Mer 8 Processor, Mark 5
GSC	Game state controller
Harvard architecture	System architecture with instructions and data memory sharing the different address spaces
Hitbox	An invisible shape commonly used in video games for real-time collision detection
I/O	Input/output
KIM	Keyboard integration module

2 INTRODUCTION

This document (hereinafter referred to as the Explanatory Note) is designed to describe developed technical solutions for the program product ‘The Fall’.

Explanatory Note as the part of documentation and the content of Explanatory Note meet the requirements to the Digital Platforms project activity. All the presented work and its further result are presented as the project work for the Digital Platforms subject.

3 PURPOSE AND AREA OF APPLICATION

The program product ‘The Fall’ (hereinafter referred to as The Fall) is the quick-time action game aimed to interactively increase the reaction of a player. In addition, The Fall is meant to also increase the player’s decision-making speed.

The gameplay of The Fall is described in Section 4.1 of Specification.

The Fall is open to free use and further distribution within the limits of Digital Platforms education program.

4 TECHNICAL CHARACTERISTICS

The Fall is designed as the independent computing system with pre-loaded software instructions using circuit logic for structuring.

4.1 Technology stack

The Fall is developed using next technology stack:

- Logisim for designing the circuit logic and implementing processor into them;
- CocoIDE for writing software instructions, compiling them and creating memory images.

4.2 Hardware modules

The whole circuit logic can be divided into several modules:

- Displays:
    a) Gameplay display;
    b) Game state display;
    c) Score display;
- Game state controller;
- Platform controller;
- Coin line controller;
- Character controller;
- Movement speed controller;
- Collision detector;
- Score counter;
- Keyboard integration module;
- Leaderboard module.

Leaderboard module has the integration with CdM-8 Mark 5 Processor (hereinafter referred to as Processor) for functioning.

There are 3 main states of the game that significantly changes the hardware processes:

– ‘Start’ – all the processes related to gameplay stay active, the interaction is available;
– ‘Pause’ – all the processes related to gameplay stop and wait until repeated call for ‘Pause’ state;
– ‘End’ – all the processes related to gameplay stop and need full restart.

The Fall’s single game activity diagram is presented in the Appendix A.

4.2.1 The outer design

The outer design is presented the way all the displays are visible at the same time. The outer design is presented on the figure below (Figure 1).

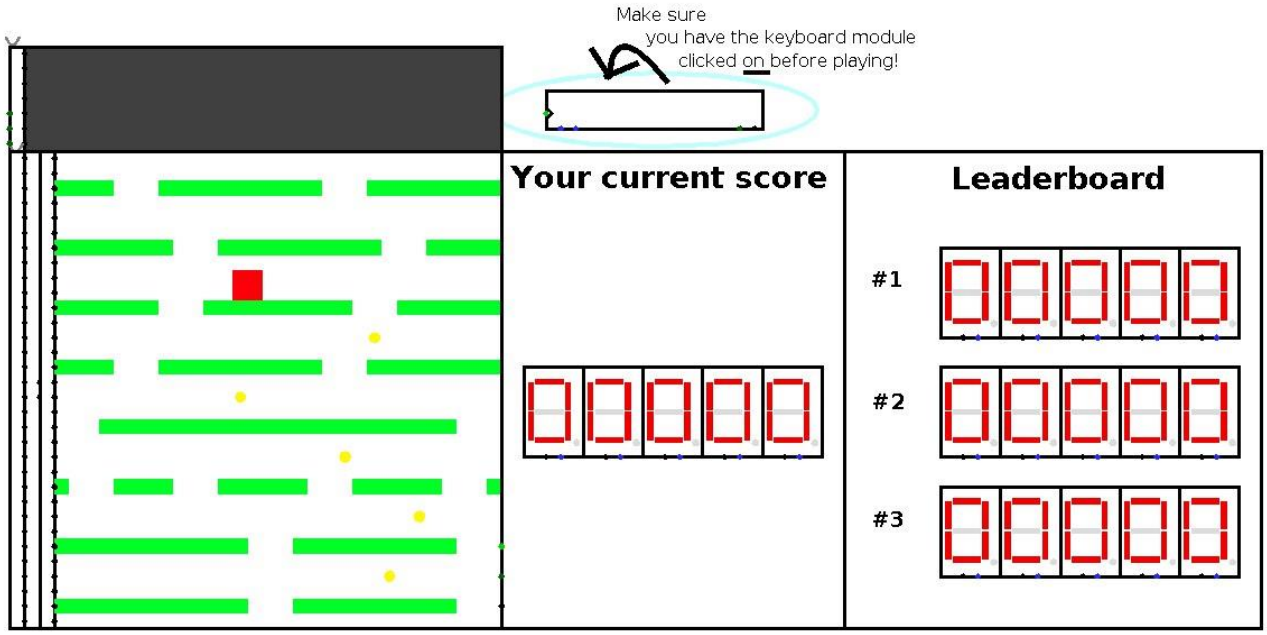


Figure 1 – The Fall’s outer design

4.2.1.1 Gameplay display

The gameplay display is a set of matrices layered upon each other to concatenate values from main controller connected to its left side. This display provides the game surroundings output for a player.

The design of gameplay display is presented on a figure below (Figure 2).

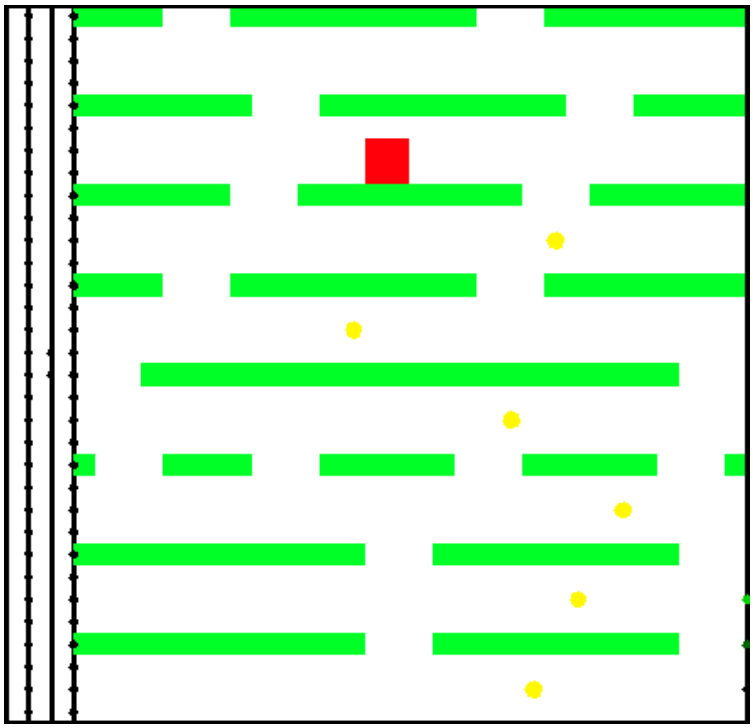


Figure 2 – The gameplay display design

4.2.1.2 Game state display

The game state display is a LED matrix that outputs values sent by game state controller. The design of game state display output variations is presented on a figure below (Figure 3).

Figure 3 – Game state output variations design

4.2.1.3 Score display

The score display outputs the score sent by main controller in real-time. The score display is a part of leaderboard panel circuit logic (see the Section 4.2.5.2).

The design of score display is presented on a figure below (Figure 4).

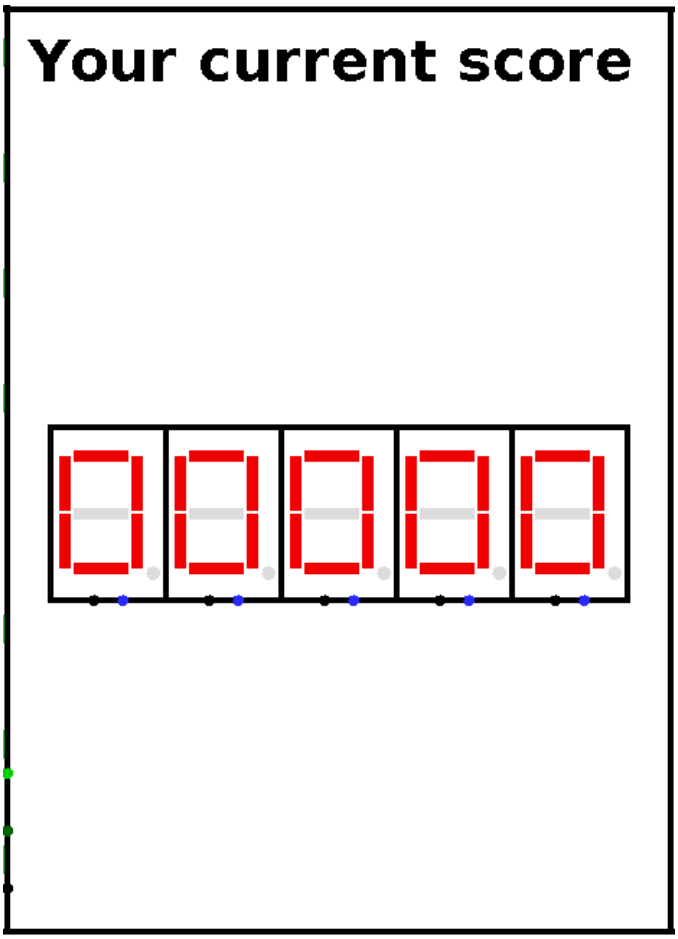


Figure 4 – The score display design

4.2.2 Game state controller (GSC)

GSC works with the game state output. When the game changes its state, GSC gets the specific signal and outputs the state on the game state display according to the signal.

The actions of GSC during game states are listed in the table below (Table 1).

Table 1 – The actions of GSC during game states

State of the game	GSC's actions during the state	Ways of calling the state
Start	Game state display shows nothing. A signal is sent to main controller to initialize all the processes on main display.	Manually by player
End	Game state display shows "DEFEAT". All main controller processes are stopped and require full restart.	Automatically by collision detection
	All main controller processes are fully restarted.	Manually by player
Pause	Game state display shows "PAUSE". All main controller processes are stopped until Pause button is pressed once again.	Manually by player

The design of game state controller is presented on a figure below (Figure 5).

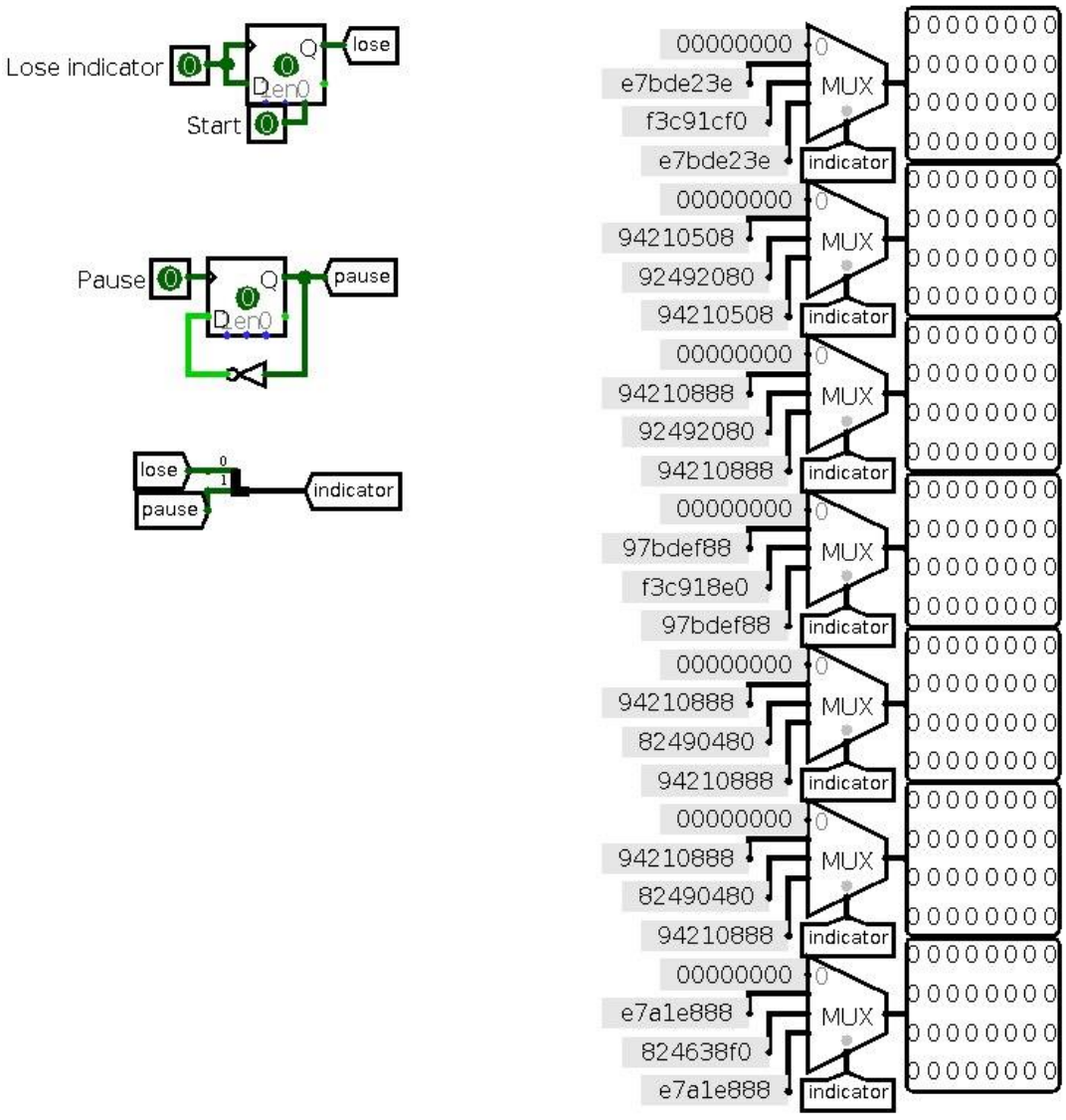


Figure 5 – Game state controller design

4.2.3 Main controller

The main controller is directly connected to the gameplay display and completely controls its interface. It is responsible for the next actions:

- Creating the platforms and coin lines;
- Moving the platforms and coin lines;
- Showing the character;
- Giving commands to move the character;
- Changing the speed of platforms, coin lines and characters;
- Detecting the collisions and sending signal according to them;
- Counting the score.

4.2.3.1 Platform controller

Before the game starts, the controller generates platforms in the distance of the display.

During the game process, the platform controller repeatedly generates a platform, randomly choosing one value from 32 hard-coded values. There are intervals between each generation, so the character can fit between platforms and move along them. Every platform has one or several gaps for the character to go down through.

The platforms are generated at the bottom of the display and moving upwards.

The actions of platform controller during game states are listed in the table below (Table 2).

Table 2 – The actions of platform controller during game states

State of the game	Platform controller's actions during the state	Ways of calling the state
Start	Platforms are generated and start moving.	Manually by player
End	Platforms stop their motion and generate in new order. If the game is lost by collision detection, the new order of platforms won't be shown on the screen until the 'Start' state is on.	Manually by player; Automatically by collision detection
Pause	Platforms stop their motion until another press of 'Pause' button.	Manually by player

The design of platform controller is presented on a figure below (Figure 6).

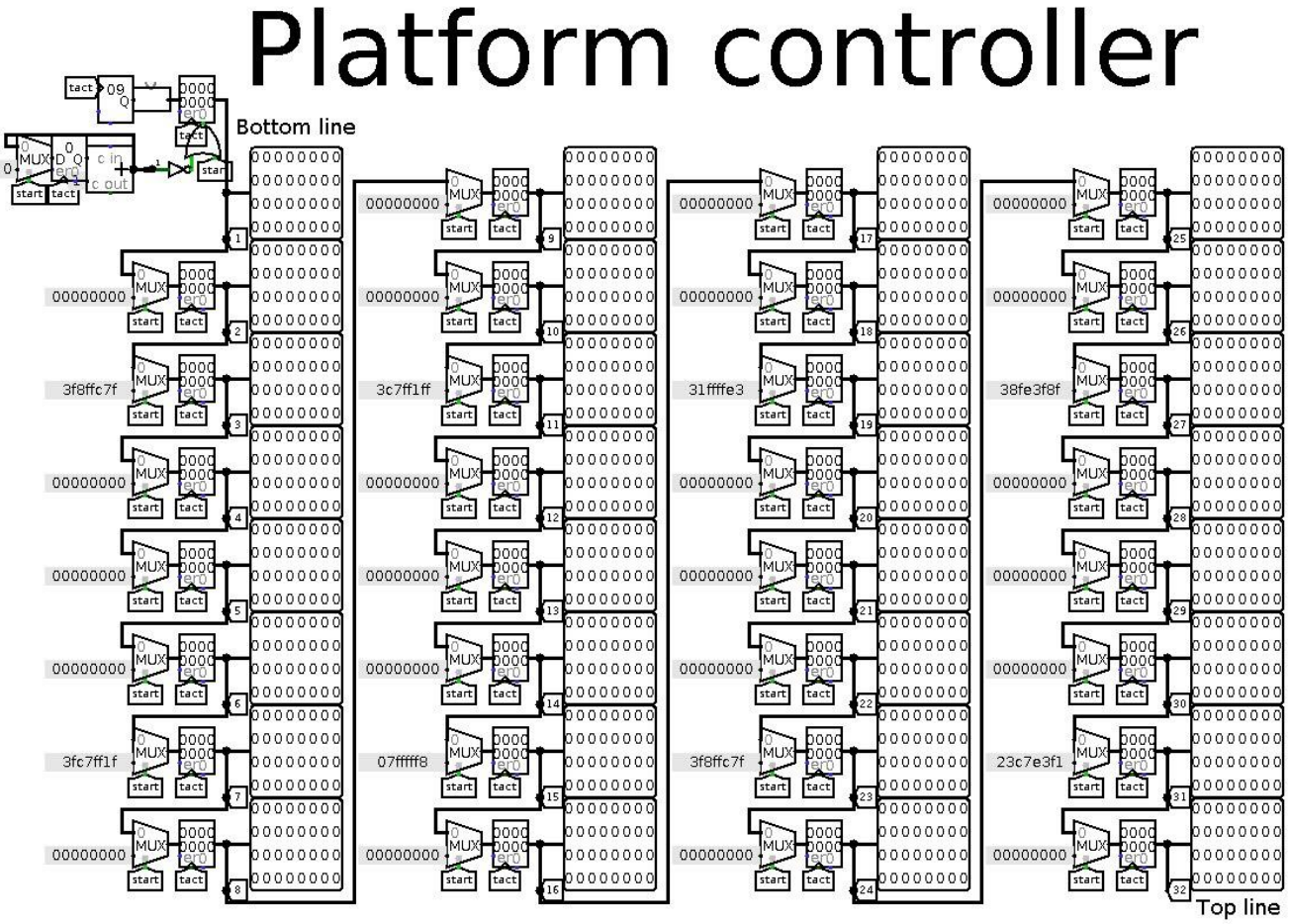


Figure 6 – Platform controller design

4.2.3.2 Coin line controller

During the game process, the coin line controller repeatedly generates a line of coins. Each coin is randomly set on the line. There are intervals correlating with platform generation the way the platforms and coin lines are moving consecutively at the same speed. Coin lines are also moving in the distance of character's hitbox, so the character can collide with coin. After the collision, controller removes the coin and sends the signal to score counter for improving the score.

The coin lines are generated at the bottom of the display and moving upwards.

The actions of coin line controller during game states are listed in the table below (Table 3).

Table 3 – The actions of coin line controller during game states

State of the game	Coin line controller's actions during the state	Ways of calling the state
Start	Coin lines are generated and start moving. The collision removing option is on.	Manually by player
End	Coin lines stop their motion and generate in new order. If the game is lost by collision detection, the new order of coin lines won't be shown on the screen until the 'Start' state is on.	Manually by player; Automatically by collision detection
Pause	Coin lines stop their motion until another press of 'Pause' button.	Manually by player

The design of coin line controller is presented on a figure below (Figure 7).

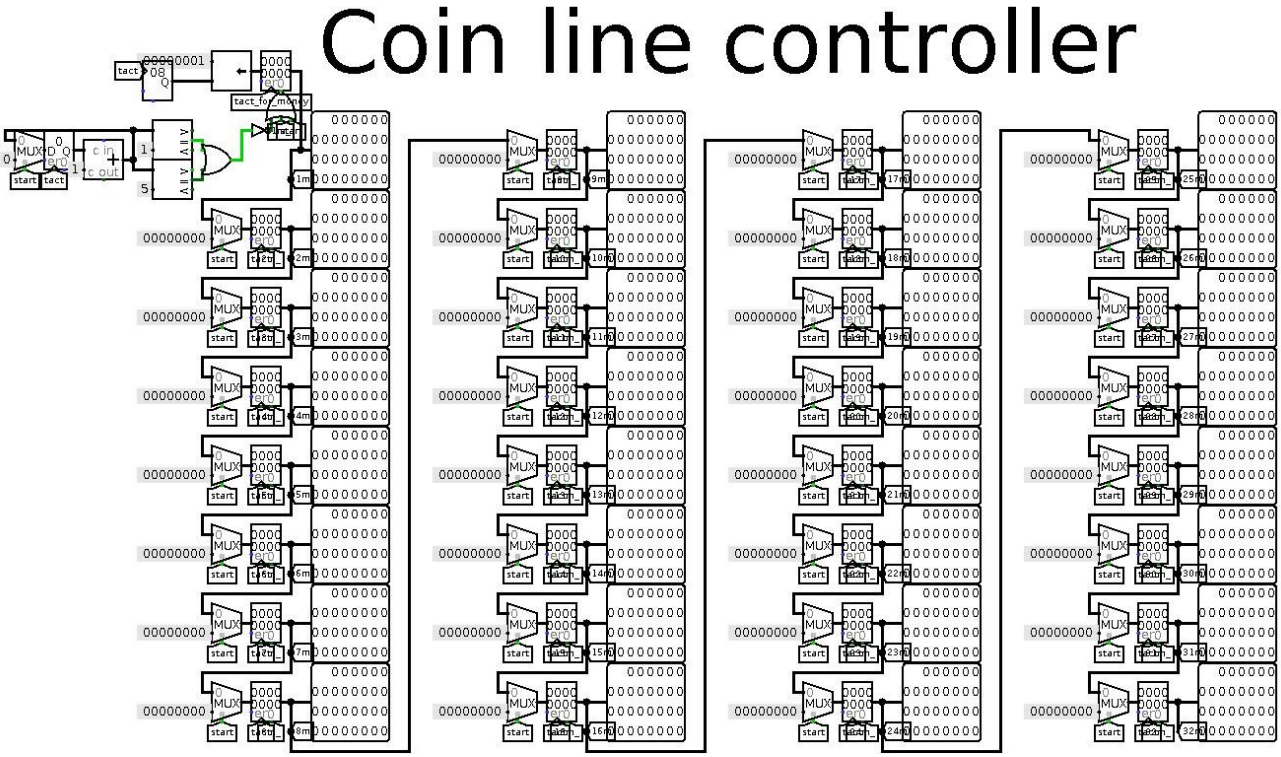


Figure 7 – Coin line controller design

4.2.3.3 Character controller

Before the game is started, character controller puts the character in upper part of the display, so the player can decide the route. During the game process, the controller checks for commands to move the character left or right and changes the horizontal position of character on the display. If there is no platform under character, the controller moves the character down until it stays on the platform. While the character is standing on the platform, its vertical position changes in the same speed as platform one.

The character can move only within the limits of the display. The character's horizontal movement speed and falling speed are slightly higher than the speed of platforms and coin lines.

The actions of character controller during game states are listed in the table below (Table 4).

Table 4 – The actions of character controller during game states

State of the game	Character controller's actions during the state	Ways of calling the state
Start	Character is placed on its starting position. Character is allowed to be moved on the display.	Manually by player
End	Character is stopped and cannot be moved. If the game is lost by collision detection, the character won't be shown on starting position until the 'Start' state is on.	Manually by player; Automatically by collision detection
Pause	Character is stopped and cannot be moved until another press of 'Pause' button.	Manually by player

The design of character controller is presented on a figure below (Figure 8).

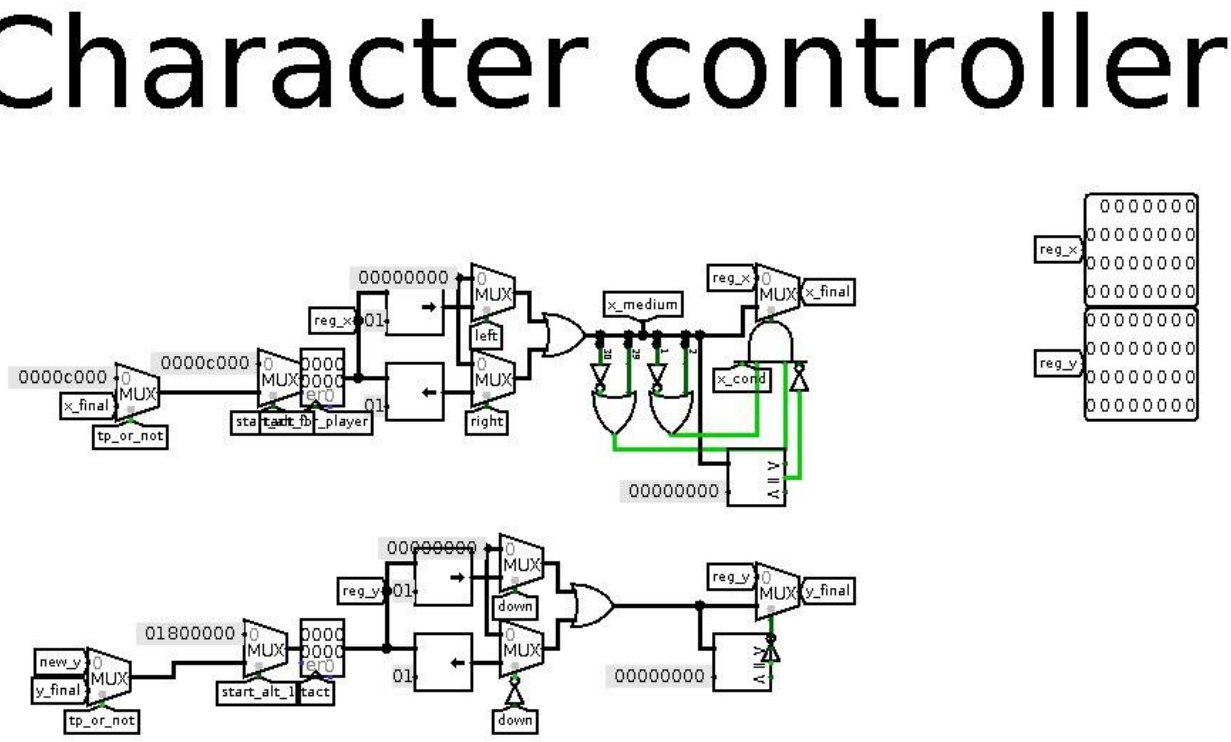


Figure 8 – Character controller design

4.2.3.4 Movement speed controller

Movement speed controller defines the movement speed of platforms, coin lines and the character. The character movement speed is designed to be slightly higher than the others. This controller also has to check for a game score defined by the score counter – by the time the score is 20, the overall movement speed gets two times faster. It repeats on the score of 60.

The actions of movement speed controller during game states are listed in the table below (Table 4).

Table 5 – The actions of movement speed controller during game states

State of the game	Movement speed controller's actions during the state	Ways of calling the state
Start	The overall speed is defining depending on score.	Manually by player
End	No movement is given for the gameplay display elements.	Manually by player; Automatically by collision detection
Pause	No movement is given for the gameplay display elements until another press of 'Pause' button.	Manually by player

The design of movement speed controller is presented on a figure below (Figure 9).

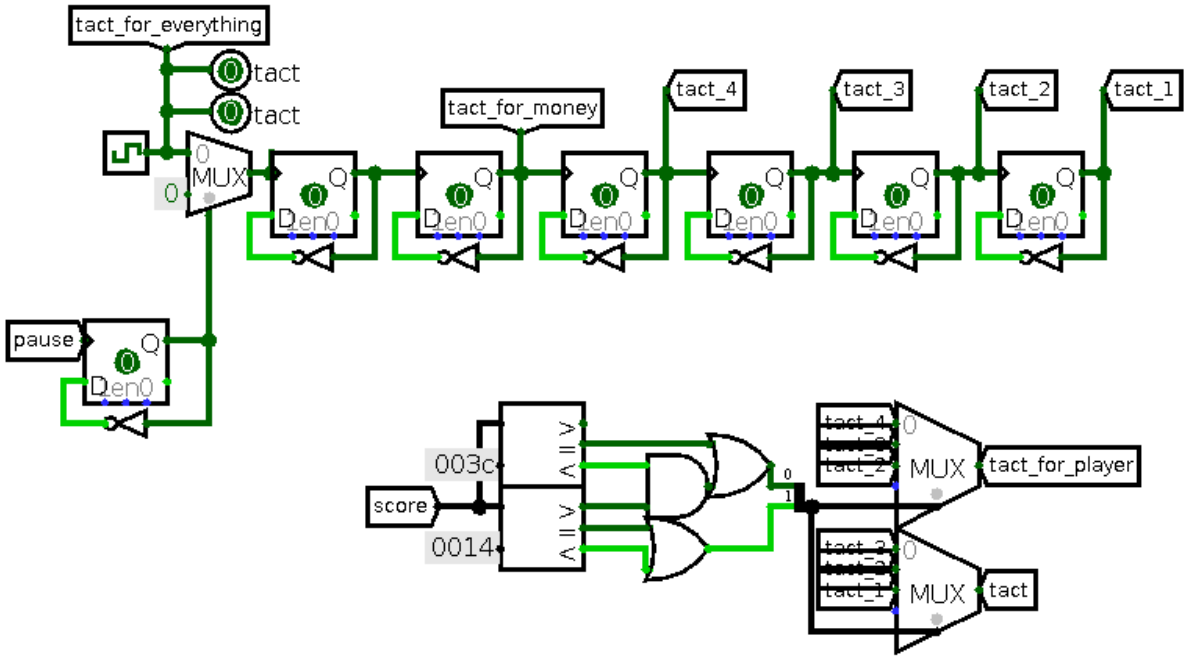


Figure 9 – Movement speed controller design

4.2.3.5 Collision detector

Collision detector checks if the character collides with upper border of the main display, which counts as the option of ending the game. If collision happens, it will send the signal to the game state controller to change the state to 'End'.

Other than that, collision detector is responsible for preventing the character from going through the platform, which results in character standing on the platform and falling down in case of no platform right below.

The actions of collision detector during game states are listed in the table below (Table 6).

Table 6 – The actions of collision detector during game states

State of the game	Collision detector's actions during the state	Ways of calling the state
Start	Collision detector starts checking for any collision cases.	Manually by player
End	Collision detector stops checking.	Manually by player; Automatically by collision detection
Pause	Collision detector stops checking until another press of 'Pause' button.	Manually by player

The design of upper border collision detection logic is presented on a figure below (Figure 10).

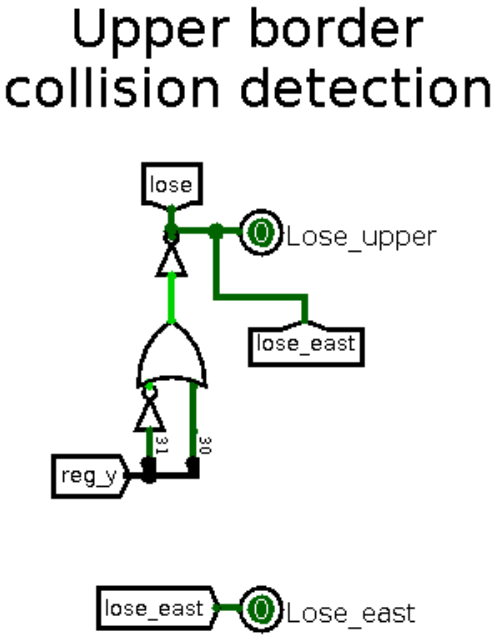


Figure 10 – Upper border collision detection logic design

The design of character-platform collision detection logic is presented on a figure below (Figure 11).

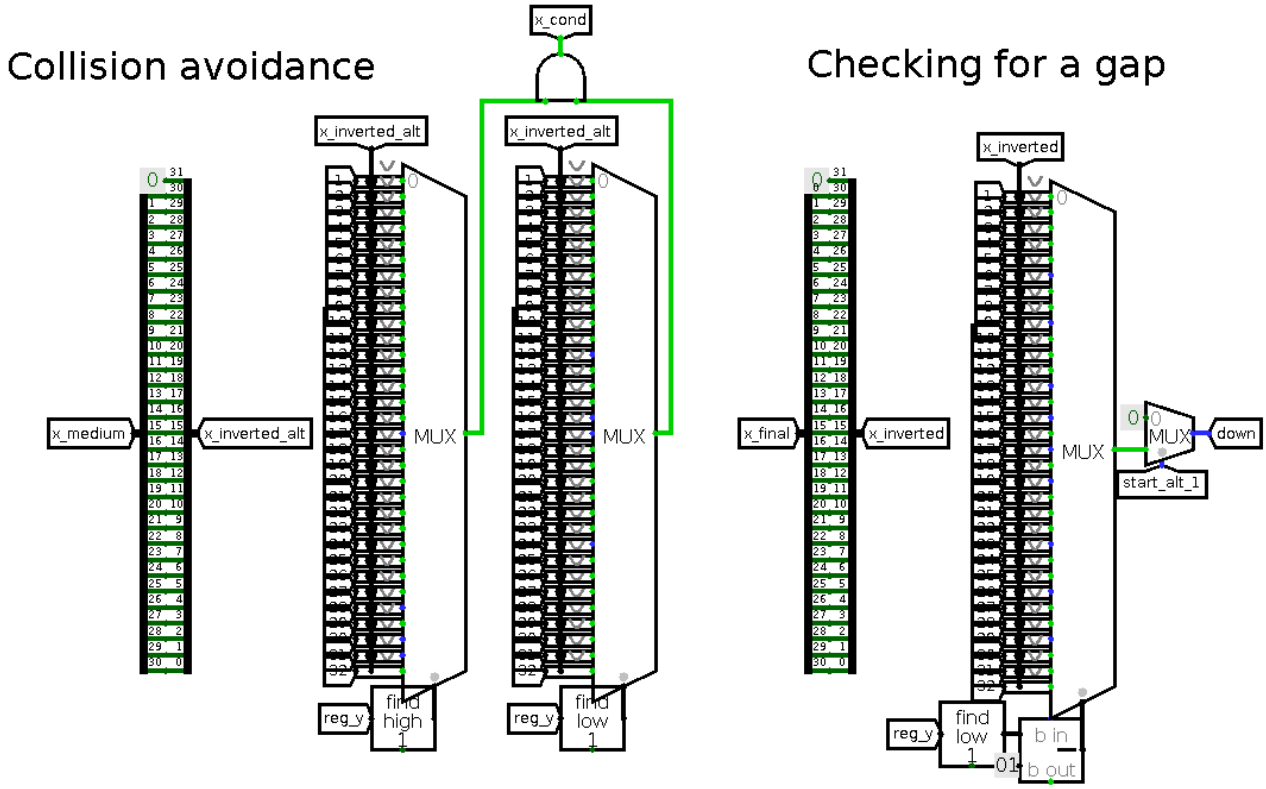


Figure 11 – Character-platform collision detection logic design

4.2.3.6 Score counter

Score counter checks for any collision between character and coin. If the collision happens, score counter improves the score by 1.

Apart from that, the score counter is responsible for detecting a collision between character and the lower border of the gameplay display. If the collision happens, score counter improves the score by 5 and teleports the character back to the middle of the gameplay display on the available platform surface.

Besides that, score counter waits for a signal about ‘Lose’ state to send the recent score to the Leaderboard.

The maximum score the player can make is 16383 (maximum 14-bit value).

The actions of score counter during game states are listed in the table below (Table 7).

Table 7 – The actions of score counter during game states

State of the game	Score counter’s actions during the state	Ways of calling the state
Start	The score is set to 0, score counter waits for collision signals.	Manually by player
End	The score counter stop working.	Manually by player; Automatically by collision detection
Pause	The score counter stop working until another press of ‘Pause’ button.	Manually by player

The design of character-coin collision detection logic is presented on a figure below (Figure 12).

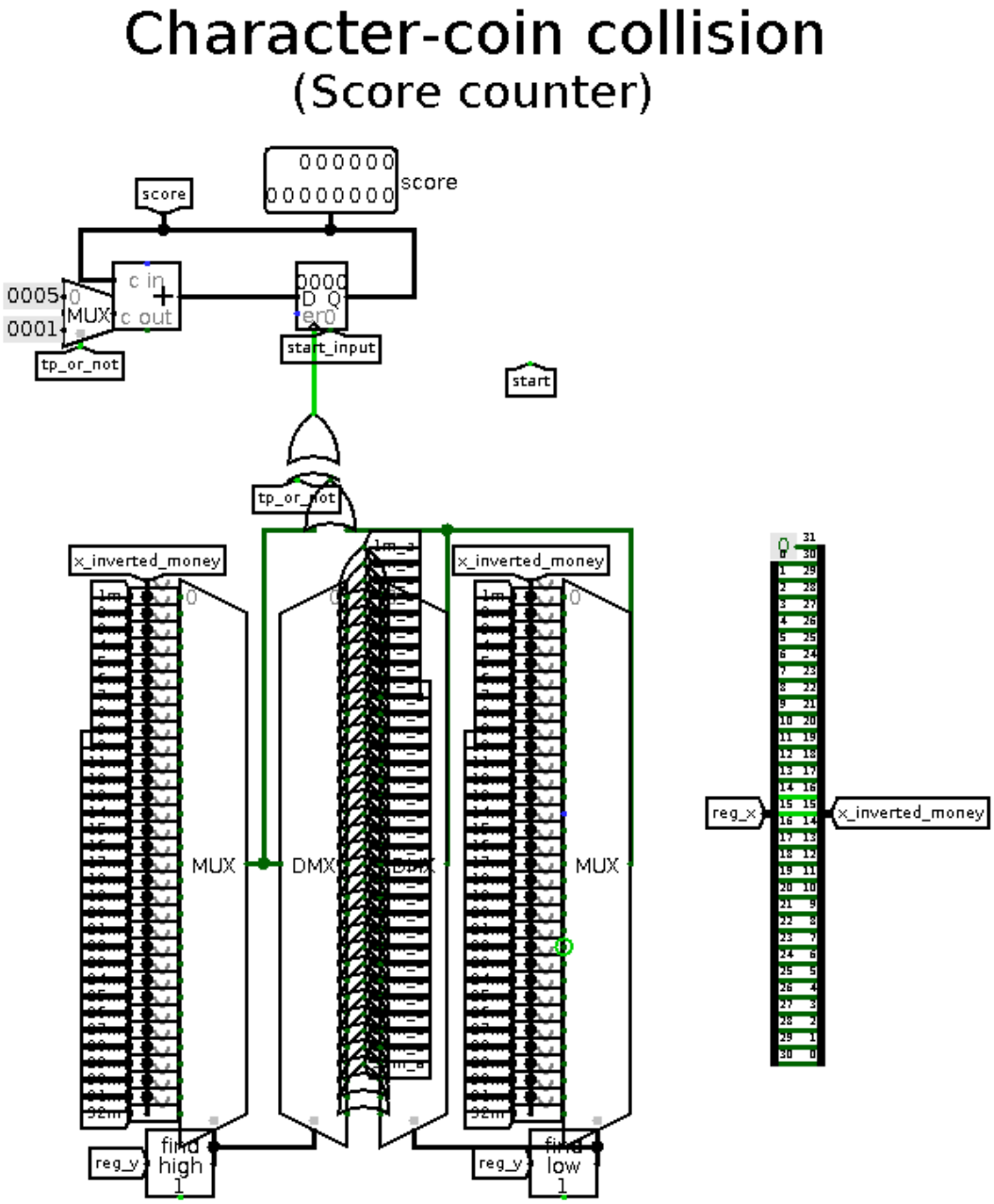


Figure 12 – Character-coin collision detection logic design

The design of lower border collision detection logic is presented on a figure below (Figure 13).

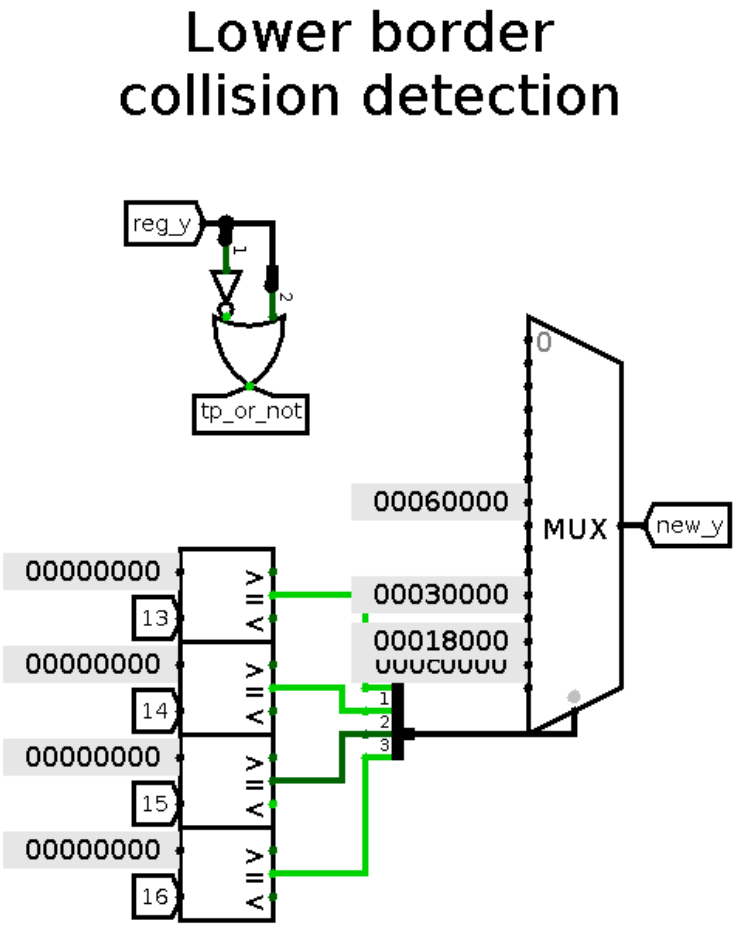


Figure 13 – Lower border collision detection logic design

4.2.4 Keyboard integration module (KIM)

The KIM allows to connect the keyboard input to the circuit logic. The setting of the keys is hardcoded.

The Fall's control keys set for the KIM is listed in the table below (Table 8).

Table 8 – The Fall's control keys

Key	Action
Enter	Start the game; End the game manually
A	Move left
D	Move right
Space	Pause the game

The design of KIM is presented on a figure below (Figure 14).

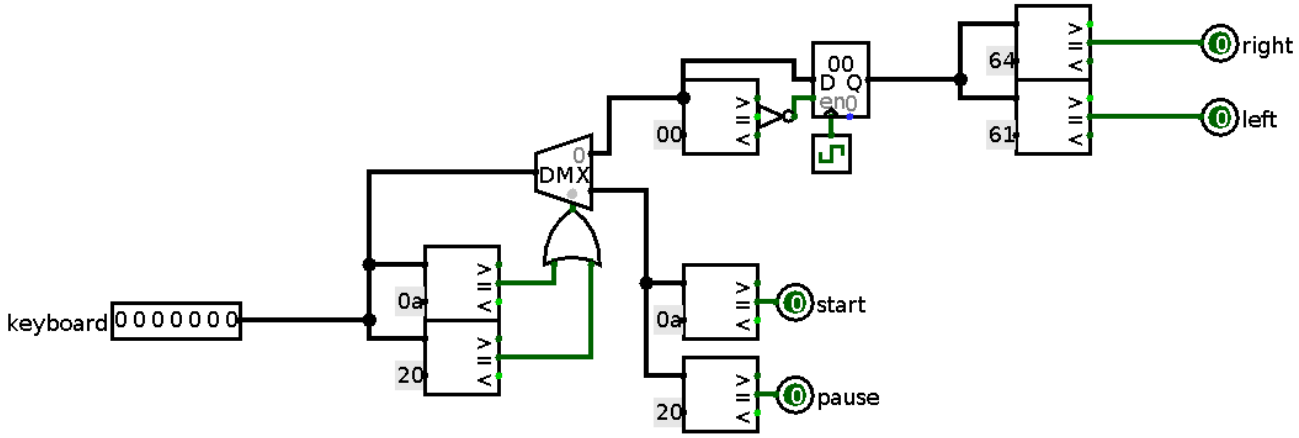


Figure 14 – Keyboard integration module design

4.2.5 Leaderboard module

Leaderboard module saves and shows the best 3 in-game results at the right side of the main display. As the recent score comes in, leaderboard module sends it to the Processor for further comparison and leaderboard update.

To print the best scores, the module converts the hexadecimal value of the score counter to the decimal output.

The outer design of leaderboard module is presented on a figure below (Figure 15).

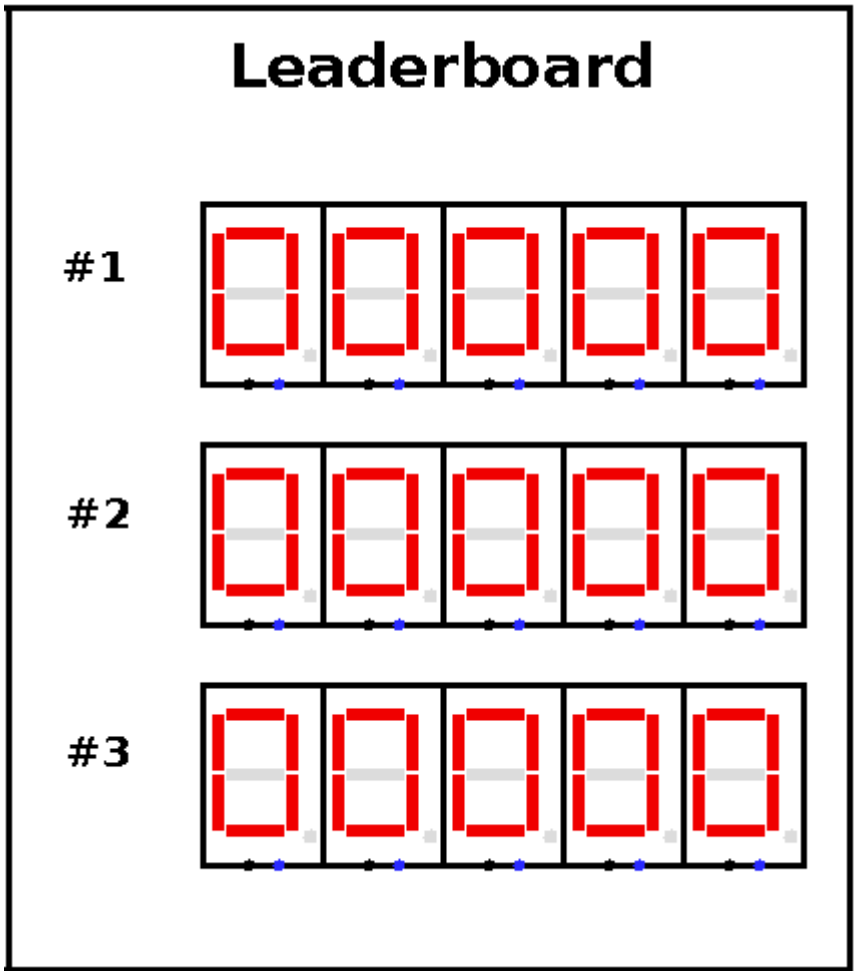


Figure 15 – Leaderboard's outer design

4.2.5.1 CdM-8 Mark 5 Processor integration

Processor is storing the memory and has the instruction to repetitively checks for the recent score and compare it with best three results. If the score is big enough to be mentioned on the leaderboard, Processor sorts these values in ascending order and sends the first three values to the leaderboard. Processor works in the background of the in-game processes.

Processor is connected with other modules by I/O bus, allowing to send the values in both directions. By using the Harvard architecture, Processor is allowed to store the leaderboard values apart from loaded instructions.

Each score is connected to a specific address of memory. Considering that score value is hexadecimal, every score requires two bytes of memory. Using the value of the address, leaderboard can define, which score should be updated.

The instruction reading is looped, so to increase the overall performance, the instruction reading will be happening until the specific command number, ending in a switch off. To switch the reading back on, the signal about the game being over should come. As long as this signal is on, the reading loop will be performing.

The design of I/O bus and integration processes is presented on a figure below (Figure 16).

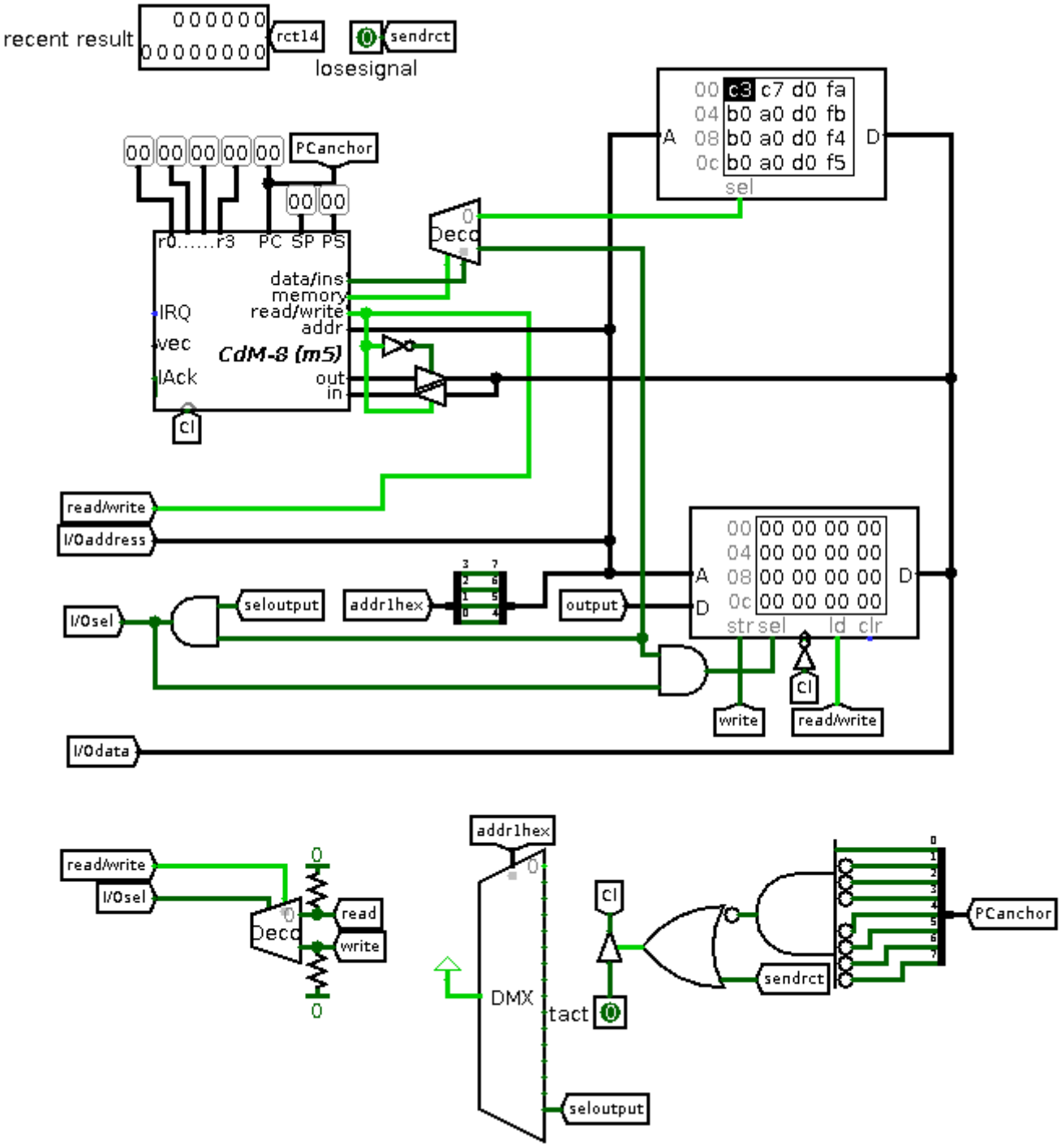


Figure 16 – I/O bus and Processor integration design

4.2.5.2 Leaderboard panel

During instruction execution, Processor sends the signals to read or to write the values. If the signal to read comes, leaderboard panel intercepts it. Each interception helps to update the score values on the leaderboard and in memory.

The updating of leaderboard happens depending on what address has been defined. If the address matches, leaderboard executes it.

Before sending the 16-bit value to memory, leaderboard panel splits it into two %0xxxxxxx-type bytes of memory, which means that a possible 14-bit score value is split in half and each half is sent independently with one additional 7th zero bit.

The design of leaderboard panel is presented on a figure below (Figure 17).

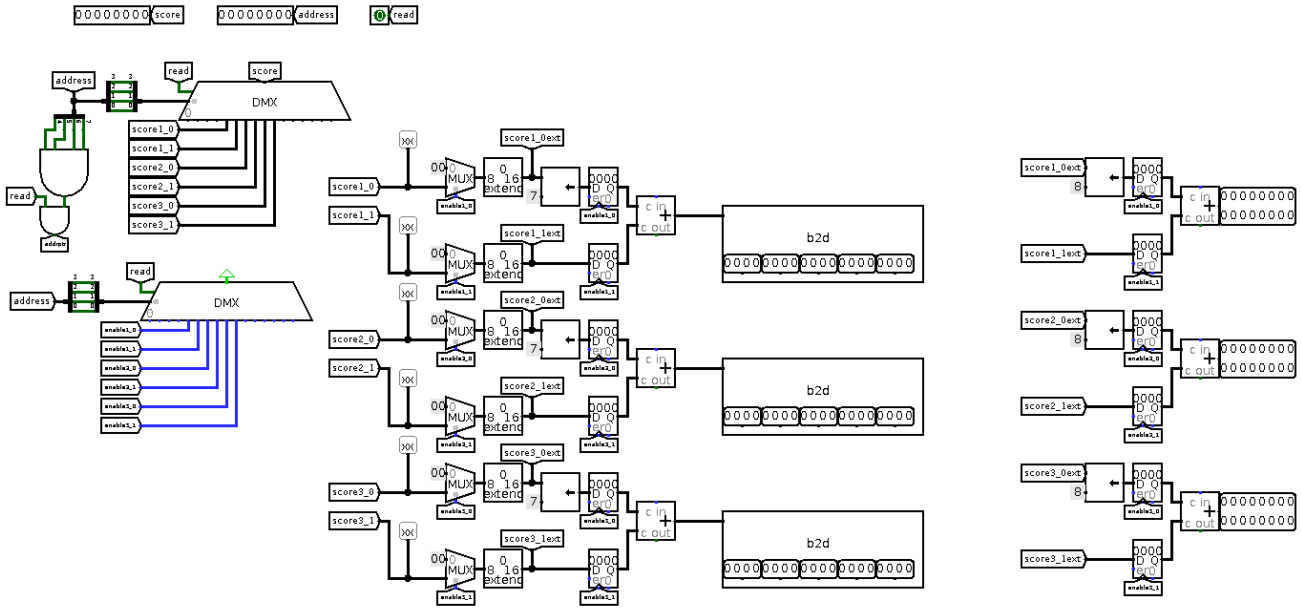


Figure 17 – Leaderboard panel design

Along with it, the score display is also connected to reading and writing processes. It converts the 14-bit value from the score counter to 16-bit value and outputs it.

The design of score display is presented on a figure below (Figure 18Figure 17).

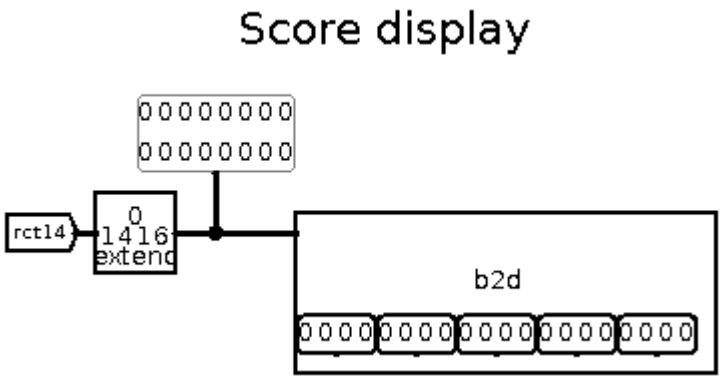


Figure 18 – Score display design

4.2.5.3 Leaderboard writing module

When the Processor's signal to write comes, the leaderboard writing module executes the process. It gets the recent score's 14-bit value from the score counter and converts it to 16-bit the same way leaderboard panel does it. Depending on what address has been defined by Processor, writing module chooses what byte should be sent and sends it.

The design of leaderboard writing module is presented on a figure below (Figure 19Figure 17).

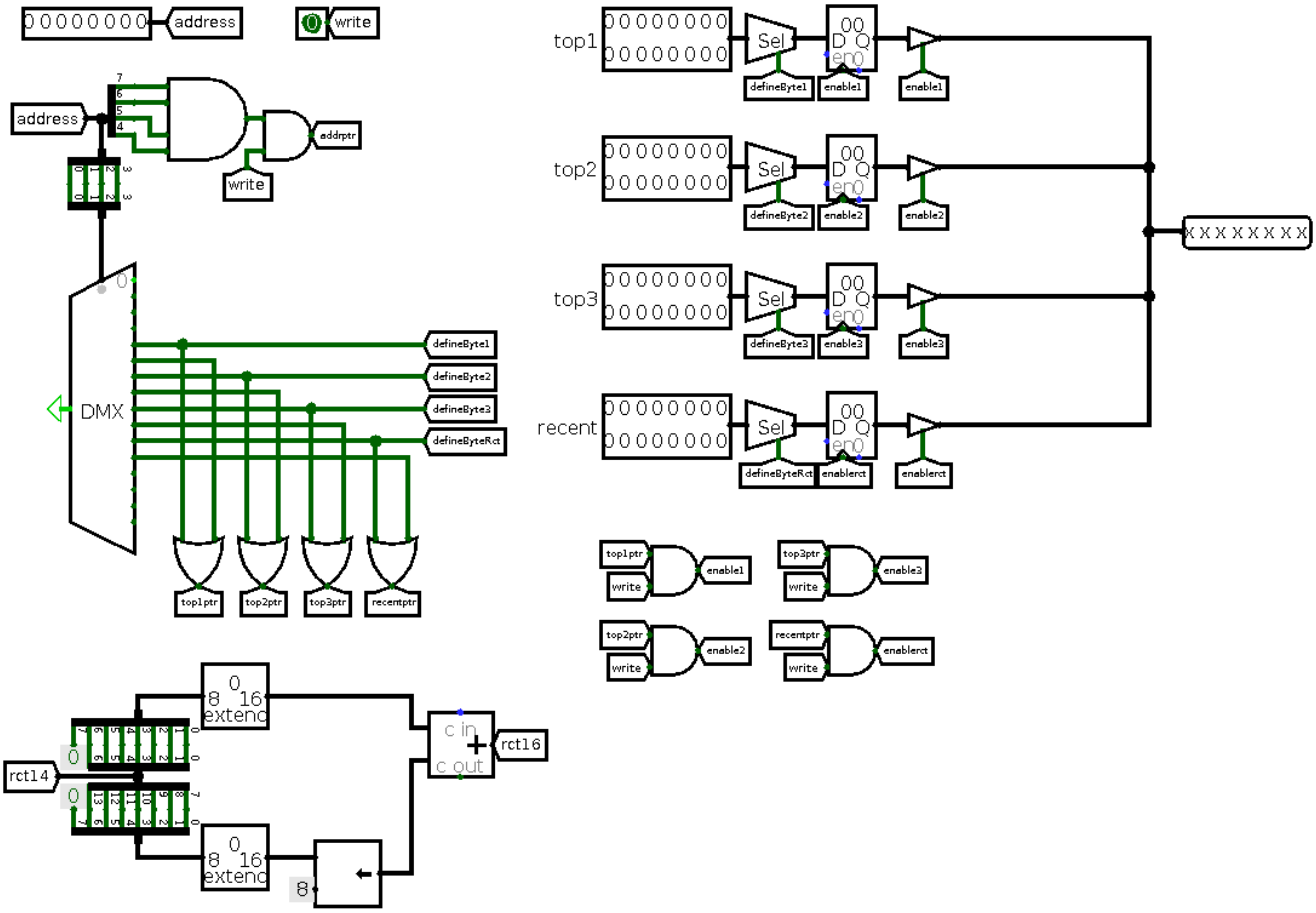


Figure 19 – Leaderboard writing module design

4.3 Potential interactions with another programs

There are no assumed interactions with another programs. Every possible interaction will require a development of specific integration bus.

5 FUNCTIONAL CHARACTERISTICS

The Processor is fully controlled by the software instructions. The uploaded memory image to instruction memory allows the Processor to go through its integration processes (see the Section 4.2.5.1).

The complete code of instructions is presented in the Appendix B.

6 CONCLUSION

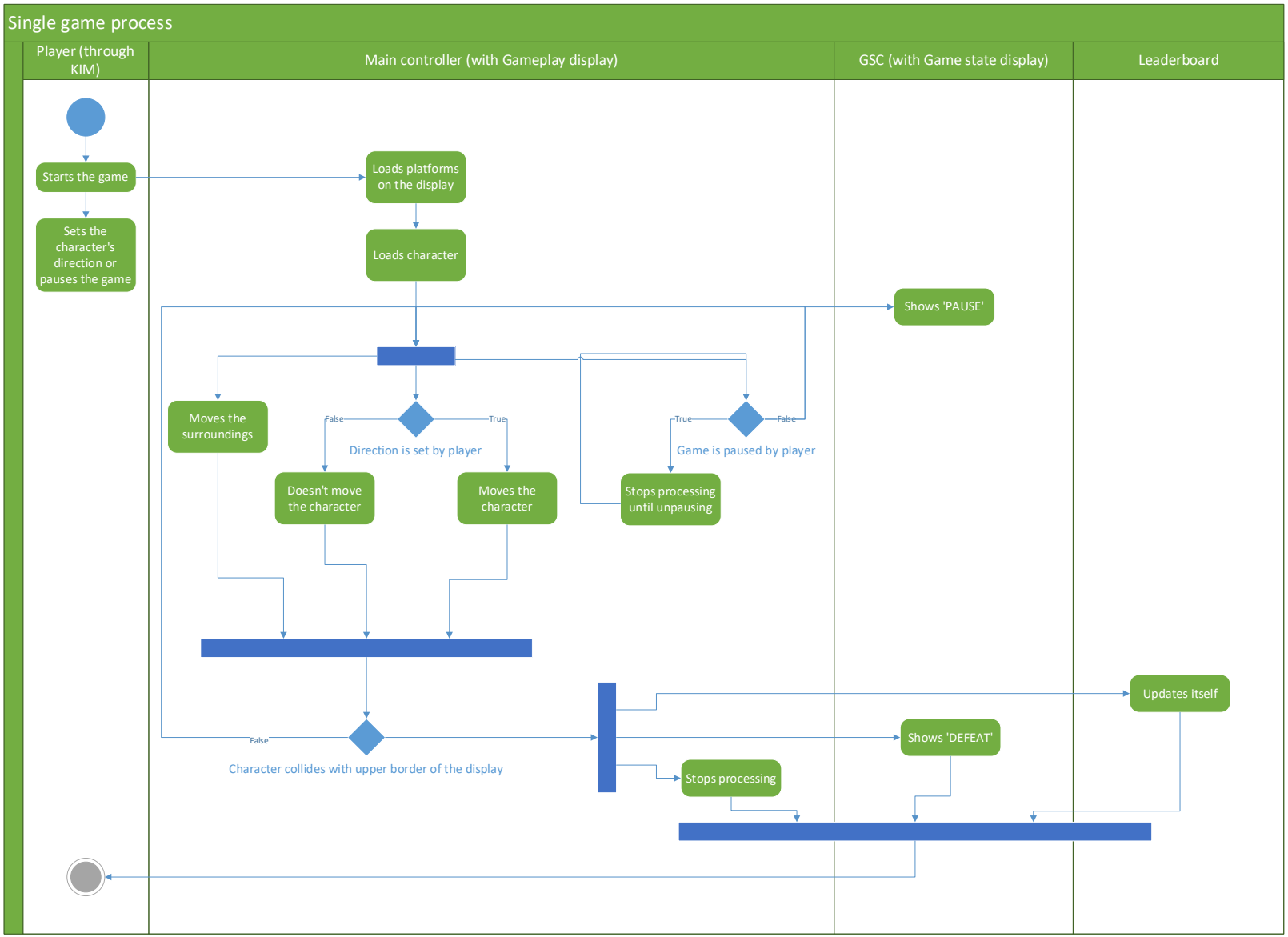
The Fall is completely finished according to Specification requirements. The quality assurance procedures have shown the good system response, which proves the comfortable level of usage. This response has been reached by adding the hardcoded solutions and the absence of any peripheral modules between displays and controllers. In addition, setting the limits for the Processor performance has also helped to increase the response speed.

Despite this, there are still possible ways to improve The Fall by varying the gameplay or using tools to raise the frequency.

7 SOURCES USED IN DEVELOPING

- Shafarenko, A. and Hunt, S.P. (2015). Computing platforms. School of Computer Science, University of Hertfordshire.
- Burch, C. (2005). Documentation. [online] Logisim. Available at: http://www.cburch.com/logisim/docs.html

APPENDIX A
The Fall's single game activity diagram



APPENDIX B
Software instructions source code

```
asect 0x00
push r3

# Updating each byte of the leaderboard.
# Getting the values from the leaderboard.
main:
    pop r3
    ldi r0, bitRct_0
    ld r0, r0
    st r0, r0

    ldi r0, bitRct_1
    ld r0, r0
    st r0, r0

    ldi r0, bitTop1_0
    ld r0, r0
    st r0, r0

    ldi r0, bitTop1_1
    ld r0, r0
    st r0, r0

    ldi r0, bitTop2_0
    ld r0, r0
    st r0, r0

    ldi r0, bitTop2_1
    ld r0, r0
    st r0, r0

    ldi r0, bitTop3_0
    ld r0, r0
    st r0, r0

    ldi r0, bitTop3_1
    ld r0, r0
    st r0, r0

    jsr cmpTop1
```

```
# Comparing the recent score with top-1 value.
# If both high-order bytes are equal,
# then the low-order bytes comparison starts.
# If the scores are equal by both bytes, rewriting doesn't happen
cmpTop1:
    pop r3
    ldi r0, bitTop1_0
    ld r0, r0

    ldi r2, bitTop1_1
    ld r2, r2

    ldi r1, bitRct_0
    ld r1, r1


    #if1----------------------------------------
    if
        cmp r0, r1
    is mi
        jsr replaceTop1
    else
        #if2----------------------------------------
        if
            cmp r0, r1
        is z
            ldi r1, bitRct_1
            ld r1, r1

            #if3----------------------------------
            if
                cmp r2, r1
            is mi
                jsr replaceTop1
            else
                #if4----------------------------
                if
                    cmp r2, r1
                is z
                    jsr main
                else
                    jsr cmpTop2
                fi
                #if4----------------------------
            fi
            #if3----------------------------------
        else
            jsr cmpTop2
        fi
        #if2----------------------------------------
    fi
    #if1----------------------------------------
```

```
# Comparing the recent score with top-2 value.
# If both high-order bytes are equal,
# then the low-order bytes comparison starts.
# If the scores are equal by both bytes, rewriting doesn't happen.

cmpTop2:
    pop r3
    ldi r0, bitTop2_0
    ld r0, r0

    ldi r2, bitTop2_1
    ld r2, r2

    ldi r1, bitRct_0
    ld r1, r1

    #if1-----------------------------------------
    if
        cmp r0, r1
    is mi
        jsr replaceTop2
    else
        #if2-------------------------------------
        if
            cmp r0, r1
        is z
            ldi r1, bitRct_1
            ld r1, r1

            #if3---------------------------------
            if
                cmp r2, r1
            is mi
                jsr replaceTop2
            else
                #if4-------------------------------
                if
                    cmp r2, r1
                is z
                    jsr main
                else
                    jsr cmpTop3
                fi
                #if4-------------------------------
            fi
            #if3---------------------------------
        else
            jsr cmpTop3
        fi
        #if2-------------------------------------
    fi
    #if1-----------------------------------------
```

```
# Comparing the recent score with top-3 value.
# If both high-order bytes are equal,
# then the low-order bytes comparison starts.
# If the scores are equal by both bytes, rewriting doesn't happen.
cmpTop3:
    pop r3
    ldi r0, bitTop3_0
    ld r0, r0

    ldi r2, bitTop3_1
    ld r2, r2

    ldi r1, bitRct_0
    ld r1, r1

    #if1-----------------------------------------
    if
        cmp r0, r1
    is mi
        jsr replaceTop3
    else
        #if2-------------------------------------
        if
            cmp r0, r1
        is z
            ldi r1, bitRct_1
            ld r1, r1

            #if3---------------------------------
            if
                cmp r2, r1
            is mi
                jsr replaceTop3
            else
                #if4-------------------------------
                if
                    cmp r2, r1
                is z
                    jsr main
                fi
                #if4-------------------------------
            fi
            #if3---------------------------------
        else
            jsr main
        fi
        #if2-------------------------------------
    fi
    #if1-----------------------------------------
```

```
# Rewriting the top-1 value with recent score.
# All the other scores are being shifted down.
replaceTop1:
    pop r3

    ldi r0, bitTop2_0
    ld r0, r0
    ldi r1, bitTop3_0
    st r1, r0

    ldi r0, bitTop2_1
    ld r0, r0
    ldi r1, bitTop3_1
    st r1, r0

    ldi r0, bitTop1_0
    ld r0, r0
    ldi r1, bitTop2_0
    st r1, r0

    ldi r0, bitTop1_1
    ld r0, r0
    ldi r1, bitTop2_1
    st r1, r0

    ldi r0, bitRct_0
    ld r0, r0
    ldi r1, bitTop1_0
    st r1, r0

    ldi r0, bitRct_1
    ld r0, r0
    ldi r1, bitTop1_1
    st r1, r0

    jsr main
```

```
# Rewriting the top-2 value with recent score.
# All the other scores lower are being shifted down.
replaceTop2:
    pop r3

    ldi r0, bitTop2_0
    ld r0, r0
    ldi r1, bitTop3_0
    st r1, r0

    ldi r0, bitTop2_1
    ld r0, r0
    ldi r1, bitTop3_1
    st r1, r0

    ldi r0, bitRct_0
    ld r0, r0
    ldi r1, bitTop2_0
    st r1, r0

    ldi r0, bitRct_1
    ld r0, r0
    ldi r1, bitTop2_1
    st r1, r0

    jsr main
```

```
# Rewriting the top-3 value with recent score.
# The previous top-3 score gets deleted.
replaceTop3:
    pop r3

    ldi r0, bitRct_0
    ld r0, r0
    ldi r1, bitTop3_0
    st r1, r0

    ldi r0, bitRct_1
    ld r0, r0
    ldi r1, bitTop3_1
    st r1, r0

    jsr main

jsr main
```

```
asect 0xF4
bitTop1_0: ds 1
bitTop1_1: ds 1

bitTop2_0: ds 1
bitTop2_1: ds 1

bitTop3_0: ds 1
bitTop3_1: ds 1

bitRct_0: ds 1
bitRct_1: ds 1

end
```