COMP2611: Computer Organization
Fall 2025
Programming Project: Super Mario Bros.
(Deadline 11:55pm, Dec 1th, Sunday via Canvas)

1 Introduction

Super Mario Bros. is a groundbreaking side-scrolling platform game that revolutionized the video game industry and became a global cultural icon.

In the game, players control Mario, an Italian plumber, or his brother Luigi. Their mission is to rescue Princess Peach from the evil Bowser, who has taken over the Mushroom Kingdom. You need to guide them to face enemies like Goombas and Koopa Troopas, cross complex platforms, and defeat Bowser's minions.

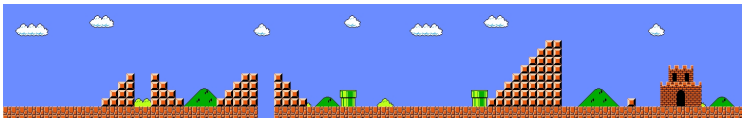
In this project, you will first implement the core features of Super Mario Bros., focusing on mastering precise jump mechanics, enemy collision detection, and the classic level progression system. Beyond the core features, you can also use your creativity to design and create a unique custom map—such as building floating cloud platforms and setting hidden power-up locations—to reconstruct a corner of the Mushroom Kingdom with your imagination.

A demo video is included in the same package for your reference. This project is a streamlined version of the original, and the detailed rules and mechanics will be introduced in the following sections.

The game runs on a modified version of MARS. To get started on your exciting journey as a game developer, download Modified MARS and the skeleton code provided. We will also regularly update an FAQ file on the course website, so be sure to check it frequently for any questions or clarifications.

System Requirement: This modified MAR implementation Mars.jar requires `JDK 8` or above. Please check your Java version prior to execution.

File Path Configuration: To ensure all game images (e.g., Mario sprites, obstacles, backgrounds) load correctly, please download the entire project package first. After extracting, place the `img/` folder (containing all game images) in the *same directory* as the `MARS-main` folder (the root directory of the modified MARS). Incorrect file placement will result in missing images and abnormal game operation.



(a) Game

(b) Coordinate System

Fig. 1: Game Image

2 Game Objects and Coordinate System

The game window, as illustrated in Fig. 1, is organized as a grid, with the x-axis aligned horizontally and the y-axis aligned vertically. The origin (0,0) is located at the top left corner of the game window. As you move away from the origin, the x-coordinate increases to the right, while the y-coordinate increases downward. The dimensions of the canvas are 1439 pixels in width and 220 pixels in height.

Table 1 outlines the various objects that will be utilized in the game. These objects will be rendered within the game window and will each serve different functionalities. Detailed descriptions of these objects will be provided in the following sections.

2.1 Game Control

In the Super Mario Bros., certain physical principles are simulated, we will introduce them one by one.

Image	Game Object	Width × Height (pixels)
	Mario	20 × 24
	Goomba	16 × 16
	Piranha Plant	16 × 16
	Brick Block	16 × 16
	Coin	16 × 16
	Star	16 × 16
	Flag	16 × 16

Table 1: Game Objects

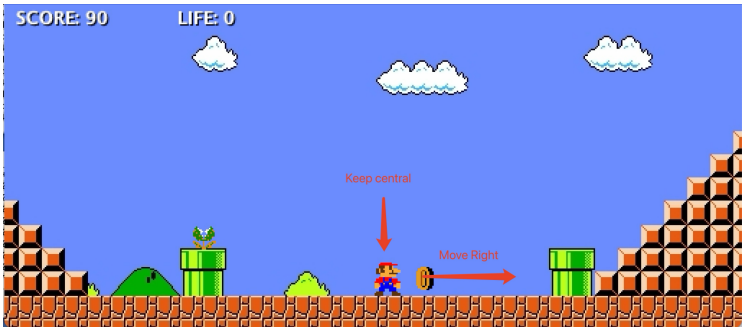
In the provided video, you can observe how keyboards control your own character movement.

2.2 Background

The game features a horizontal-only camera system that is tied to Mario's (or Luigi's) movement, rather than a fixed background. This camera behavior is shown in Fig. 2. The camera only moves rightward—specifically, it shifts when the character moves past the horizontal center of the screen, ensuring Mario (or Luigi) never goes beyond this center point. It does not move leftward at all, even if the character walks to the left side of the current view. Players can focus entirely on controlling their character and navigating obstacles, without needing to manually adjust the camera position.

The game design includes a total of two maps. Your in-game goal is to earn enough points and successfully clear the entire game.

The total points earned and the remaining player lives will be displayed in the top-left corner of the game interface. Please note that when the number of lives drops to zero, the entire game will restart.



(a) Camera Movement

(b) Score and Life

Fig. 2: Game Interface

2.3 Objects

The key part of this project is to implement the logic of the character movement and interaction with the environment. Collision detection is the core component.

The key aspect of this project is to implement the logic for character movement and interaction with the environment, with collision detection being the core component. In the game, each interactive object is created independently using MIPS. The coordinates of each object and its collision detection boundaries are included in the skeleton.

Tab. 1 has listed all objects used in the game and the function description of each object is included.

2.4 Character control

You're responsible for controlling one character in the game: Mario.

The game's character movement uses specific keybindings: Mario is controlled via arrow keys for basic movement—up arrow (jump), left arrow (move left), and right arrow (move right).

Your task is to implement smooth left and right movement mechanics for Mario, ensuring his movement feels responsive. Additionally, you'll need to develop the jumping and falling mechanisms, which follow physics-based rules—falling triggers automatically when Mario is no longer supported by the ground.

The movement system comprises four directional capabilities: left, right, jumping, and falling. While the first three are player-initiated through keyboard inputs, falling is gravity-driven and occurs passively. This system allows for smooth navigation across platforms and obstacles in the level. Additionally, pressing the X key will instantly return Mario to the level's starting position, useful for resetting after mistakes.

Make sure to carefully review the .data section in Skeleton—you can modify parameters there for debugging purposes, such as increasing Mario's movement speed, setting lives to 99, or adjusting his jump height to test different scenarios.

2.5 Collision Detection

There are multiple types of collision in the game, with each object type distinguished by a unique ID. You need to implement a loop to detect the IDs of colliding objects and partially realize the functionality triggered after collision with different objects:

- Collision between Mario and ground/walls (ID: 1) – Prevents Mario from passing through, serving as solid obstacles to keep him within the playable area.
- Collision between Mario and Brick (ID: 2) – Triggers interactions like breaking the brick (if Mario has a power-up) or bouncing off it; contact from the side will block Mario's movement.
- Collision between Mario and Star (ID: 3) – Grants Mario temporary invincibility, allowing him to defeat enemies on contact without taking damage.
- Collision between Mario and Coin (ID: 4) – Increases the total score by a fixed value and makes the collected coin disappear from the level.
- Collision between Mario and Piranha Plant (ID: 5) – Triggers damage to Mario (reducing lives or reverting to small Mario) if contacted; Mario can avoid damage by jumping on the plant when it's fully extended.

- Collision between Mario and Goomba (ID: 6) – Defeats the Goomba if Mario jumps on its top (the Goomba will flatten and disappear); causes damage to Mario if contacted from the side or below.
- Collision between Mario and Flagpole (ID: 7) – Triggers level completion, initiating the end-of-level sequence (e.g., Mario sliding down the pole and moving to the next area).

The pseudo code for iterating through object IDs and basic collision response logic is provided in the MIPS file. Ensure to reference it when implementing the loop for ID detection and the corresponding post-collision behaviors.

2.6 Game Score and Game End

In the game, your goal is to guide Mario safely through the level while collecting as many coins as possible to accumulate points, and ultimately complete the level by reaching the exit.

The level contains several environmental hazards and scoring elements with the following rules:

- Collecting one Coin (ID: 4) grants 10 points, contributing to the total score.
- Defeating a Goomba (ID: 6) by jumping on its top grants 30 points; the Goomba will flatten and disappear after being defeated.
- The total maximum score for the first level is 100 points, calculated based on all collectible coins and defeatable Goombas in the level.
- Mario will be eliminated if he falls out of the map (when his Y-coordinate exceeds 250).
- Mario will be eliminated if he collides with a Goomba (ID: 6) horizontally (from the side or below) instead of jumping on its top.
- Mario will be eliminated if he collides with a Piranha Plant (ID: 5), regardless of the contact direction.

Game Progression and End Conditions:

- To complete the level, Mario must first lower the Flagpole (ID: 7) to its lowest position, then reach the exit door.
- If Mario is eliminated due to falling out of the map, colliding with a Goomba (horizontally), or touching a Piranha Plant, the current level will restart immediately.
- You can press "X" to return Mario to the level's starting position at any time (for quick reset during debugging or after mistakes).

3 Game Implementation

Implementing Super Mario bros. in MIPS assembly can be challenging, but you won't have to start everything from scratch. The modified MARS (copyright: COMP2611 teaching team) takes care of the fancy user interface, while the MIPS code primarily focuses on the game logic. You will build upon the provided skeleton code, which is well-organized with subtasks assigned to different MIPS procedures. Although it is recommended to read the skeleton code and grasp the details, you should still be able to complete the project by understanding the overall code structure and focusing on individual tasks assigned to different procedures with well-defined interfaces.

3.1 Data Structure

All objects are surrounded by a rectangle. This table contains four elements you need to consider during maneuvering any objects.

Rectangle attribute	**Description**
x	The x coordinate of the top left corner
y	The y coordinate of the top left corner
width	The width of the rectangle
height	The height of the rectangle
ID	The ID of the item

Table 2: Data Structure of a **Rectangle**

All data variables used in the game are declared as static and are defined at the beginning of the skeleton code.

It's understandable if the list of variables feels overwhelming, and some of them may not make sense to you at this moment. However, once you begin working on the project skeleton and actively engage in implementing the tasks, these variables will become much more comprehensible and meaningful to you. Don't worry, as you delve into the project and gain hands-on experience, their purpose and relevance will become clearer.

You can also add more content in data segment. If you have any new ideas, you can post them in piazza. More information is shown in the code.

3.2 Game Loop

1. **loop**: This is the main loop where all game logic for Mario is executed, including input handling, physics, collision, and rendering.
2. **Initialization (before loop)**: Before entering the *loop*, the system initializes critical components:
   - Loads constants (e.g., `CENTER_X`, `GROUND_Y`, movement speeds like `vx_walk_c`) into registers for quick access.
   - Sets initial states: Mario's starting position (`X=32`, `Y=GROUND_Y - MARIO_H`), scroll position (`scrollX=0`), vertical velocity (`vy=0`), and facing direction (`1=right`).
   - Initializes game status: Resets score to 0, sets initial lives to 1, marks Mario as "on ground" (`onGround=1`), and clears the previous frame's key input.
   - Renders basic visuals: Loads Mario's sprite via `syscall 5104`, displays the background via `syscall 5100`, and initializes all in-game objects (from `obj_arr`) via `syscall 5102`.
   - Starts background music (BGM) via `syscall 5120`.
3. **Key Input Reading**: Execute `syscall 5103` to read the current keyboard input, storing the result in register `$t7` for subsequent parsing.
4. **Reset / Death Check**: Trigger reset logic based on specific conditions:
   - If the `X` key is pressed (checked via mask `0x0020`), jump to `do_reset` to reset Mario's state.
   - If Mario's Y-coordinate exceeds 250 (falls out of the map), jump to `fall_die` to decrease lives by 1. If lives drop below 0, call `full_reset` for a complete game reset; otherwise, trigger `do_reset`.
5. **Reset Execution (do_reset)**: Reset core game states to their initial values:
   - Set `scrollX=0` and restore Mario's starting position (`X=32`, `Y=GROUND_Y - MARIO_H`).
   - Reset vertical velocity (`vy=0`), facing direction (`1=right`), and ground state (`onGround=1`).
   - Clear the previous frame's key input and refresh the window via `syscall 5101` to update visuals.
6. **Input Parsing (if no reset)**: Analyze the current key input to determine movement intent:
   - Parse `LEFT (0x0001)`, `RIGHT (0x0002)`, `UP (0x0004)` for jump, and `Z (0x0010)` for running.
   - Select horizontal speed (`vx`): Use `vx_run_c` if `Z` is pressed; otherwise, use `vx_walk_c`. Adjust sign based on direction (negative for left, positive for right).
7. **Horizontal Collision & Position Update**: Call `resolve_horizontal` with Mario's current `X/Y` position and target `vx`. Update Mario's X position (`$s4`) with the returned value and adjust his facing direction based on the final horizontal velocity.

8. **Jump Handling**: Check for the rising edge of the `UP` key (to prevent holding jump). If Mario is on the ground (`onGround=1`), set his vertical velocity to `jump_imp` (jump impulse) to trigger a jump.
9. **Physics Update (Gravity & Vertical Movement)**:
   - Apply gravity by adding `gravity_c` to Mario's vertical velocity (`vy = vy + gravity_c`).
   - Call `resolve_vertical` with Mario's current `X/Y` position and `vy` to handle vertical collisions. Update Mario's Y position (`$s5`) and `vy` with the returned values.
10. **Object Collision Detection**: Call `check_obj_collision` to detect interactions between Mario and in-game objects (e.g., coins, stars, or enemies like Goomba and Piranha Plant). If a fatal collision occurs (e.g., touching an enemy), trigger `do_reset`.
11. **Enemy Movement Update**: Call `update_enemy_positions` to update the positions of all enemies (e.g., Goomba) based on their AI logic.
12. **Camera & Boundary Management**:
    - Call `update_camera` to adjust `scrollX`, making the camera follow Mario horizontally while keeping him within view bounds.
    - Clamp Mario's X position to stay within the camera view (left bound = `scrollX`, right bound = `scrollX + VIEW_W - MARIO_W`).
13. **Visual Refresh**: Execute `syscall 5101` to refresh the window, updating Mario's sprite position, facing direction, and movement state (running or jumping) to reflect all logic changes.
14. **Frame Timing**: Enter `delay_loop` to apply a frame delay (based on `fps_delay`) ensuring consistent game speed regardless of system performance.
15. **Repeat**: Return to the start of the *loop* to repeat the cycle until a game-over condition (`lives < 0`) is met.

4 Tasks

Read the skeleton code carefully. Do not trap yourself in understanding every single line of the code.

Top Priorities:

1. Understand the data structures used in the skeleton;
2. Trace the game loop, understand how it works;
3. Figure out the functionality of each procedure.

Once you build up a big picture of the project, zoom in to the programming tasks, and examine the comments and example codes carefully.

Caution: It is important to note that you should refrain from modifying the skeleton code directly. Instead, you should add your own code within the designated procedures while adhering to the structure and organization of the existing code. This approach ensures that you can integrate your implementation seamlessly without disrupting the functionality of the skeleton code.

Task 1-5: Coding Tasks

Tab. 3 outlines all the programming tasks you need to implement, with each task corresponding to a specific procedure. You may find it convenient to call other procedures within these tasks to streamline your implementation. It is essential to carefully read the step-by-step instructions provided in the skeleton code for each task, as the descriptions in this document are brief. Ensure that your implementation adheres strictly to the instructions outlined in the project skeleton, without overlooking any operations on the required variables, and avoid performing any actions outside of what is specified. Additionally, remember to preserve any `s` registers used within any procedure by saving them onto the stack, as this is crucial for maintaining the integrity of your program's state across procedure calls.

Try to follow MIPS conventions and good programming practices, e.g., comment your code properly and use registers wisely.

5 Syscall Services

The MIPS game runs on a modified MARS, which provides additional set of syscall services related to the game, e.g., draw the game screen, obtain mouse action etc.

Tasks	Procedure	Descriptions
Task 1	`Mario_move`	Move Mario leftwards and rightwards based on keyboard input.
Task 2	`check_objects_collision`	Loop collision detection with different objects.
Task 3	`Wall_collision`	Collision detection with solid objects which will stop the mario.
Task 4	`Move_the_Goomba`	Just Make Goomba move horizontally!!!
Task 5	`Drop_the_flagpole`	Continuous collision detection with the flagpole to drop the flag.
Task 6*	`Add_your_own_item`	Add two custom object types in Map 2 and implement basic interaction logic with them.
Task 7*	`Custom_Movable_Objects`	Create two types of objects that can move within a certain range (horizontally or vertically) and implement basic interaction logic with them.

Table 3: Programming Tasks (Tasks 6* and 7* are optional extensions)

If you have problems running the modified MARS, that is most likely due to the outdated Java Running Environment (JRE). Follow the instructions from your MARS labs, and update your Java Development Kit (JDK) version if necessary.

6 Submission

You are required to submit a zip archive **named with your student ID** (e.g., `12345678.zip`, where `12345678` is your student ID). This archive must include the MIPS assembly file named `2611S2025_yourStudentID.s` and all resource files associated with your project. It is critical that the TA can execute this skeleton code without additional configuration; failure to meet this requirement may result in a loss of points.

Submission is via Canvas. The deadline is a hard deadline. Try not to upload at the last minute. If you upload multiple times, we will grade the latest version by default.

7 Grading

The programming project is an individual assessment. You can discuss with your friends if you have difficulty in understanding the tasks. But every single line of your code should be your own work (not something copied from your friends). Do **NOT** share your code with anyone else.

Points: Task 1: 15 pts, Task 2: 20 pts, Task 3: 25 pts, Task 4: 20 pts, Task 5: 20 pts. The total score for the main tasks is **100 points**.

In addition, **Task 6*** and **Task 7*** are optional bonus tasks that can grant extra credit:

- **Task 6***: +5 bonus points
- **Task 7***: +10 bonus points

Including these bonus tasks, you can earn up to a maximum of **115 points** in total.

Important: If you implement **Task 6*** or **Task 7***, you must include detailed comments **in the code** explaining the additional features you added, how they are integrated into the existing framework, and the interaction logic of these new elements. In addition to the `2611F2025_yoursStudentID.s` assembly file, write a Word document to explain your work. Zip them into a single `2611F2025_yoursStudentID.zip` for submission.

You are allowed to use any pseudo-instruction supported by the modified MARS in your implementation. Generally, your source code will not be reviewed in detail; however, be prepared to explain specific segments of your MIPS code if we have any suspicions of plagiarism.

Services	Code	Parameters	Result
Mario Background & Window	5100	`$a0`: Path of background image (char*)	`$v0` = 0 (success); -1 (image not found/unparseable); -2 (UI update failed); -3 (path read error)
Mario State Update	5101	`$a0`: scrollX (camera position); `$a1`: mx (Mario X); `$a2`: my (Mario Y); `$a3`: Mario state (running/jumping); `$v1`: facing direction (1=right, -1=left)	`$v0` = 0 (success, updates camera and Mario display)
Obstacle Initialization	5102	`$a0`: base_addr (array base); `$a1`: N (count); `$a2`: baseImgTbl (image path table address); `$a3`: m (table length)	`$v0` = 0 (success); -3 (memory read error)
Key Input Reading	5103	None	`$v0` = bitmask (bit0=LEFT, bit1=RIGHT, bit2=UP, bit3=DOWN, bit4=Z, bit5=X, bit6=ENTER, bit7=SHIFT)
Mario Sprite Setting	5104	`$a0`: Path of right-facing sprite; `$a1`: Path of left-facing sprite (optional, auto-flip if null)	`$v0` = 0 (success); -1 (load error); -2 (UI update error); -3 (path read error)
Score Setting	5107	`$a0`: new score value	`$v0` = 0 (success, updates on-screen score)
Life Setting	5108	`$a0`: new life count	`$v0` = 0 (success, updates life display)
All Obstacles/Items Clear	5109	`$a0`: item_arr base (address); `$a1`: item_n address (count variable)	`$v0` = 0 (success, clears memory and UI); -1 (execution failed)
BGM Play	5120	`$a0`: Path of BGM file; `$a1`: loopFlag (0=once, 1=loop); `$a2`: volume_millibel (volume level)	`$v0` = 0 (success); -1 (file/play error); -2 (audio error); -3 (path read error)

Table 4: Mario Custom Syscall Services