Mario - Assembler MIPS

Matías Berardo, Alexis Byrne, Agustina Caffaratti, Lucas Ruberto, Franco Rubín

Noviembre, 2025

Abstract

Este informe presenta la documentación del proyecto *Super Mario Bros.*, un videojuego de plataformas 2D desarrollado íntegramente en lenguaje Assembler MIPS para el simulador MARS 4.5. El proyecto demuestra la implementación de físicas simples, renderizado por píxeles, detección de colisiones, desplazamiento horizontal de cámara y animaciones básicas dentro de un entorno de bajo nivel. Se describe la arquitectura del sistema, el modelo de memoria, el motor de física, el sistema de entrada y las optimizaciones aplicadas. Finalmente, se realiza un análisis de los resultados obtenidos y las dificultades encontradas durante el desarrollo.

1. Introducción

El desarrollo de este proyecto consistió en implementar una versión funcional de *Super Mario Bros* en lenguaje ensamblador MIPS. El objetivo fue comprender en profundidad el funcionamiento de la arquitectura del procesador, el manejo de memoria, los registros y la interacción con un entorno gráfico simulado, construyendo desde cero cada aspecto del motor del juego: físicas, renderizado, entrada, colisiones y entidades.

Más allá del resultado visual, este trabajo permitió afianzar el pensamiento algorítmico de bajo nivel, la optimización de código y la capacidad de estructurar un programa complejo en un lenguaje sin abstracciones de alto nivel.

2. Arquitectura del Sistema

El juego se compone de distintos subsistemas:

- **Bucle principal de juego** (en `main`): gestiona la secuencia lógica de ejecución.
- **Entrada del usuario**: detecta las teclas presionadas y modifica la velocidad o acciones de Mario.
- **Motor de física**: aplica gravedad, saltos y movimiento horizontal.
- **Sistema de colisiones**: evita que Mario atraviese el suelo, plataformas, tuberías y enemigos.

- **Renderizado**: dibuja los elementos visuales en la memoria gráfica simulada.
- **Gestión de entidades**: maneja enemigos (Goombas), monedas y power-ups.

3. Modelo de Memoria

La memoria del programa se divide en dos secciones principales:

- **.data**: contiene las constantes del juego (dimensiones, colores, posiciones iniciales, sprites).
- **.text**: almacena el código ejecutable. Cada rutina se encarga de una parte del motor.

Las posiciones de pantalla se almacenan en palabras de 32 bits y se accede a la memoria de video mediante dirección base `0x10008000`, calculando el desplazamiento a partir de las coordenadas X e Y.

4. Motor de Física

El motor de física aplica gravedad y controla la posición de Mario. La función principal es `update_mario_physics`, que actualiza las coordenadas según la velocidad actual y limita la velocidad de caída.

```
update_mario_physics:
    addi $sp, $sp, -8
    sw $ra, 0($sp)
    sw $s0, 4($sp)

    lw $s0, mario_y

    lw $t0, mario_vy
    lw $t1, GRAVITY
    add $t0, $t0, $t1

    lw $t1, MAX_FALL_SPEED
    blt $t0, $t1, vy_ok
    move $t0, $t1
vy_ok:
    sw $t0, mario_vy

    lw $t1, mario_y
    add $t1, $t1, $t0
    sw $t1, mario_y
```

Rutina de física de Mario

Esta rutina:

- Incrementa la velocidad vertical de Mario aplicando gravedad.
- Limita la velocidad máxima de caída.
- Actualiza la posición en el eje Y y X.
- Llama a otras rutinas que verifican colisiones con el suelo, tuberías y plataformas.

5. Sistema de Renderizado

El renderizado se realiza escribiendo directamente en la memoria de video. Cada píxel se representa por un color en formato hexadecimal RGB. La rutina `draw_pixel` es la base de todas las funciones gráficas.

```
draw_pixel:
    bltz $a0, pixel_skip
    bltz $a1, pixel_skip
    li $t0, 128
    bge $a0, $t0, pixel_skip
    li $t0, 64
    bge $a1, $t0, pixel_skip

    li $t0, 0x10008000
    sll $t1, $a1, 7
    add $t1, $t1, $a0
    sll $t1, $t1, 2
    add $t0, $t0, $t1
    sw $a2, 0($t0)
pixel_skip:
    jr $ra
```

Rutina base de dibujo de píxeles

Esta rutina:

- Verifica que las coordenadas estén dentro de los límites de la pantalla.
- Calcula la dirección de memoria correspondiente al píxel.
- Escribe el color en esa posición.

6. Sistema de Entrada

El módulo `process_input` gestiona la lectura del teclado mapeado en memoria y modifica la velocidad o acciones de Mario. Permite movimiento lateral y salto con teclas `A/D/W` o `Espacio`.

```
process_input:
    li $t0, 0xffff0000
    lw $t1, 0($t0)
    andi $t1, $t1, 1
    beqz $t1, apply_friction
    ...
```

Procesamiento de entrada del usuario

El manejo de entrada se basa en:

- Leer el bit de disponibilidad de teclado.
- Analizar el código ASCII de la tecla presionada.
- Ajustar la velocidad horizontal o vertical de Mario.
- Aplicar fricción para desacelerar al no presionar teclas.

7. Gestión de Entidades

Las entidades principales son:

- **Goombas**: enemigos con desplazamiento lateral.
- **Monedas**: se recolectan al contacto.
- **Power-ups**: aparecen de bloques misteriosos y aplican efectos.

El sistema de enemigos se actualiza con `update_goombas`, que gestiona el movimiento y dirección de cada Goomba, mientras que `check_goomba_collisions` detecta colisiones con Mario, aplicando daño o puntos según la situación.

8. Sistema de Colisiones

Las colisiones se manejan mediante comparaciones de coordenadas entre los bordes de Mario y los objetos del entorno (suelo, plataformas, tuberías). Este sistema evita que el personaje atraviese elementos sólidos y permite rebotes o muertes de enemigos.

9. Gestión de Power-ups

Los bloques misteriosos pueden generar power-ups mediante la rutina `spawn_powerup`, que busca un espacio libre en el arreglo y los inserta con una velocidad vertical inicial, simulando su salida desde el bloque.

10. Flujo de Ejecución

El flujo principal del juego se encuentra en la rutina `main`, que controla el ciclo de ejecución:

```
main:
    jal clear_screen
wait_for_start:
    li $t0, 0xffff0000
    lw $t1, 0($t0)
    andi $t1, $t1, 1
    beqz $t1, wait_for_start
    ...
game_loop:
    jal process_input
    jal update_mario_physics
    jal update_goombas
    jal check_goomba_collisions
    jal check_coin_collisions
    jal render_frame
    j game_loop
```

Bucle principal de ejecución

El flujo general es:

1. Limpiar pantalla e iniciar variables.
2. Esperar la entrada del jugador.
3. Ejecutar el bucle del juego (entrada → física → colisiones → renderizado).
4. Mostrar pantallas de victoria o derrota.

11. Análisis de Complejidad

Aunque el lenguaje ensamblador MIPS no ofrece estructuras de alto nivel que faciliten el análisis formal, es posible estimar la complejidad del sistema observando su comportamiento general. Las actualizaciones que se realizan en cada cuadro del juego presentan

una complejidad de orden O(n), siendo n la cantidad de entidades activas en pantalla, como enemigos, monedas o power-ups. Cada una de estas entidades requiere ser procesada individualmente en cada iteración del bucle principal.

Las rutinas de colisión también tienen un costo lineal respecto al número de objetos, ya que se basan en comparaciones simples entre coordenadas y dimensiones para determinar contactos o superposiciones. Este enfoque resulta eficiente al evitar estructuras de datos complejas y mantener un flujo de ejecución directo.

Finalmente, el sistema de renderizado sigue una complejidad similar, dado que dibuja en memoria cada entidad visible por cuadro. Aunque el costo total crece linealmente con el número de objetos en pantalla, el desempeño se mantiene estable gracias a la naturaleza secuencial y optimizada del código ensamblador.

12. Problemas Conocidos y Soluciones

Durante el desarrollo se identificaron diversas dificultades que requirieron ajustes técnicos. Uno de los problemas iniciales fue el desborde de pantalla, ocasionado por la falta de verificación de los límites de coordenadas. Esto se resolvió mediante la utilización de instrucciones condicionales como `bltz` y `bge`, que impiden que los valores de posición superen el rango válido del entorno gráfico.

Otro inconveniente importante fue la detección inexacta de colisiones, especialmente en los saltos y las plataformas flotantes. Se corrigió ajustando las tolerancias verticales y mejorando las condiciones de comparación entre las posiciones de Mario y los objetos del entorno.

Por último, se abordó la lentitud general del sistema, especialmente en fases con muchas entidades simultáneas. La solución consistió en reducir el retardo dentro de la rutina `delay`, optimizando así la frecuencia de actualización del juego y obteniendo un mejor equilibrio entre rendimiento y estabilidad visual.

13. Optimizaciones Implementadas

Con el fin de mejorar la eficiencia del programa, se aplicaron diversas optimizaciones a nivel de código. Se priorizó el uso de registros temporales para minimizar los accesos a memoria, reduciendo el tiempo de ejecución de las operaciones más frecuentes. Además, se implementó un control riguroso del stack en cada subrutina, garantizando la integridad del flujo de ejecución y evitando conflictos entre llamadas anidadas.

Finalmente, se promovió la reutilización del código mediante saltos y subrutinas compartidas, lo que permitió mantener la estructura del programa más compacta, legible y modular, sin sacrificar el rendimiento general del sistema.

14. Conclusiones

El desarrollo de este *Super Mario Bros.* en Assembler MIPS permitió aplicar de manera práctica los conceptos fundamentales de la arquitectura del procesador, la gestión de memoria y la programación de bajo nivel. Durante el proyecto se enfrentaron desafíos relacionados con la sincronización, la precisión de las colisiones y el rendimiento del renderizado, los cuales fueron resueltos a través de análisis, depuración y optimización del código.

Más allá del objetivo técnico, el trabajo permitió comprender el funcionamiento interno de un videojuego y el valor del control total sobre cada componente del sistema. Programar en MIPS, sin herramientas de alto nivel, obliga a razonar sobre cada instrucción y a optimizar cada ciclo, reforzando la lógica, la organización y la capacidad de resolución de problemas.

En conclusión, este proyecto no solo representa una implementación funcional de un juego clásico, sino también un ejercicio integral de pensamiento computacional, trabajo en equipo y aplicación de conocimientos teóricos en un contexto realista.