# Documentación Técnica Integral de Juego Pac-Man en MIPS Assembly

Dana Cejas, Tobias Leanza, Anna Smitarello, Santiago Pricco, Ignacio Giovanetti

5 de noviembre de 2025

## Índice

# 1. Estructura de Archivos y Registros Globales

El juego está desarrollado bajo una arquitectura modular en **MIPS Assembly**, con un sistema de coordenadas basado en píxeles y una pantalla de $\approx 64 \times 80$ unidades.

## 1.1. Estructura de Datos Central (`Posicao`)

La variable `Posicao` es el vector central para el estado del juego.

| Offset (Bytes) | Personaje | Propósito | Detalle |
|---|---|---|---|
| 0 – 7 | Pac-Man | Coordenadas (X, Y) | `Posicao+0` (X), `Posicao+4` (Y) |
| 8 – 15 | Fantasma 1 | Coordenadas (X, Y) | |
| 16 – 23 | Fantasma 2 | Coordenadas (X, Y) | |
| 24 – 31 | Fantasma 3 | Coordenadas (X, Y) | |
| 32 – 39 | Fantasma 4 | Coordenadas (X, Y) | |
| 40 – 43 | Pac-Man | Último Comando | |
| 44 – 59 | Fantasmas 1-4 | Último Comando | |
| 60 – 63 | **Flag Power-Up** | Estado de Pac-Man | **0** = Normal, $\neq$ **0** = Potenciado ($**t8** en `comer`) |
| 64 – 79 | Fantasmas 1-4 | Contador de Reaparición | Tiempo de muerte (umbral: **15**) |

## 1.2. Convenciones de Registros Esenciales

- `$s0`**:** Utilizado para el **Color** (RGB Hexadecimal) en rutinas de dibujo (`draw_P`).
- `$s1`**,** `$s2`**:** Coordenadas **X** e **Y** del personaje o píxel actual.
- `$s3`**: Dirección/Comando** de Movimiento (e.g., ASCII de 'w', 's', 'a', 'd').
- `$s4`**:** Registro de Retorno para análisis (`AnalisisMapa`), ej. 1 para pared, 0 para libre; o valor de punto/fruta.
- `$s7`**:** Dirección base de `Posicao`.
- `$t8`**:** Copia del **Flag Power-Up** de Pac-Man (`Posicao+60`), crucial en la lógica de IA y colisiones.

# 2. `main.asm` – Control de Flujo del Juego

## 2.1. `.text` – Bucle de Juego (`Pac_Man` a `fimmov`)

El bucle principal gestiona un *frame* del juego.

1. **Movimiento de Pac-Man:**
   - Se carga el comando y coordenadas de Pac-Man.
   - Se llama a `mov_Pac_Man()` para mover, comer puntos/frutas y actualizar las coordenadas.
2. **Lógica de Colisión (Pac-Man $\leftrightarrow$ Fantasmas):**
   - Se llama a `comer()`. Esta macro utiliza el flag `$t8` para decidir si Pac-Man muere (si $**t8** $= 0$) o si come al fantasma (si $**t8** $\neq 0$).
3. **Verificación de Fin de Juego:**
   - Se llama a `victoria()`, que chequea si se han recolectado **218** puntos y reinicia el juego si es verdadero.
4. **Secuencia de Fantasmas (`Fantasma_1` a `Fantasma_4`):**
   - Se cargan las coordenadas y el *offset* del comando anterior (`$s6`).
   - Se llama a `reaparecer()` (verifica si el contador $\geq 15$).
   - Si reaparece, se llama a `decidir_el_color()`, `decidir_Ghost()`, y `mov_Ghost()`.

- Finalmente, se llama a `comer()` de nuevo para chequear colisiones después de mover al fantasma.

5. **Sincronización (`fimmov`):**
   - `syscall` con código **32** y valor **75** (**ms** de pausa) para controlar la velocidad del juego.

# 3. `pintar.asm` y `draw.asm` – Gráficos de Bajo Nivel

## 3.1. `pintar.asm` – Macro `draw_P`

Macro fundamental para dibujar un píxel individual en la memoria mapeada.

- **Registros de Entrada:** `$s0` (Color), `$s1` (X), `$s2` (Y).
- **Dirección de Memoria (`$t0`):** El cálculo es:

$$\texttt{BaseAddress} + (\mathbf{X} \times 4) + (\mathbf{Y} \times 256)$$

- **Lógica:** La coordenada X se multiplica por 4 (*word size*), y la coordenada Y se multiplica por 256 (el número de bytes en una fila) mediante un bucle (`loop_y:`). Finalmente, el color (`$s0`) se almacena en la dirección calculada (`sw $s0,($t0)`).

## 3.2. `draw.asm` – Dibujo de Entidades

Utiliza `draw_P` repetidamente para formar el patrón del personaje.

- `draw_Pac_Man`**:** Dibuja Pac-Man como un patrón de $\approx$ 6 píxeles. La posición de la "boca abierta"se ajusta dinámicamente en función de la dirección de movimiento actual (`$s3`) para simular **animación** (ej. la boca se omite en la dirección 'd' (100) si se mueve hacia la derecha).
- `draw_Ghost`**:** Dibuja el patrón simple de un fantasma (generalmente un cuadrado 2x2, compuesto por 4 píxeles).

# 4. `AnalisisMapa.asm` – Lógica de Colisiones

Este módulo provee las herramientas para la interacción de los personajes con la geometría del laberinto.

- `carga` **(`$s4` ← Color):** Macro auxiliar que implementa la misma lógica de cálculo de dirección que `draw_P` pero en reversa, cargando el valor del color del píxel en (`$s1`, `$s2`) en el registro de retorno `$s4` (`lw $s4,($t0)`).
- `next_Block` **(`$s4` ← 0 o 1):**
  - Calcula la posición del píxel **2** unidades adelante en la dirección `$s3`.
  - Llama a `carga()`.
  - Compara el color cargado (`$s4`) con el color de la pared (`16711680` o `0xFF0000` - Rojo).
  - Retorna **1** si es pared, **0** si es camino libre.
- `block`**:** Lógica de **detección y gestión de colisiones de pared**.
  1. Llama a `next_Block()` para la dirección actual.
  2. Si detecta pared (`$s4`=1), compara el comando actual (`$s3`) con el comando anterior (`Posicao+40`).
  3. Si el comando actual es igual al anterior, se asume que está atascado, y se establece `$s3` en **0** (**parada forzada**).
  4. Si es diferente, se revierte al **comando anterior** y se verifica si ese movimiento es válido. Si no lo es, también se establece $\$\mathbf{s3} = 0$.
- `bifurcacion` **(`$s4` ← 0 o 1):**

- Verifica si la posición actual (`$s1`, `$s2`) tiene **3** o **4** caminos disponibles (no pared) en N, S, E, O.
- El número de caminos disponibles se acumula en `$t4`.
- Retorna **1** si `$t4` $\geq 3$, indicando que el fantasma debe tomar una decisión de IA.

# 5. `mov.asm` – Lógica de Movimiento y Teletransporte

## 5.1. `mov_Ghost` y `mov_Pac_Man`

Ambas macros comparten la lógica de movimiento en pasos de $\pm$**3** unidades por ciclo.

- **Lógica de Teletransporte:** Se verifica si el personaje está en el túnel lateral (X=61 o X=1) y la dirección (`$s3`) corresponde a la salida.
  - **Derecha:** Si `$s1`=61 y `$s3`=100 ('d') → Salida por el túnel izquierdo (`$s1`=1, `$s2`=40).
  - **Izquierda:** Si `$s1`=1 y `$s3`=97 ('a') → Salida por el túnel derecho (`$s1`=61, `$s2`=40).
- **Lógica de `mov_Pac_Man` (Comida):**
  - **Puntos:** Llama a `next_Block_punto()`. El valor del punto (`$s4`) se suma a `pontos`.
  - **Frutas/Power Pill:** Llama a `next_Block_fruta()`. Si `$s4`=1 (fruta encontrada), establece el flag de Power-Up en `Posicao+60` a **0** (`sw $zero, 60($s7)`).

# 6. `desicion_Ghost.asm` – Inteligencia Artificial (IA)

El módulo de IA utiliza la aleatoriedad para simular un comportamiento fantasmal impredecible, siempre condicionado a la existencia de una `bifurcacion()`.

## 6.1. `decidir_Ghost`

Macro de selección de modo:

- Si `$t8` (Flag Power-Up) = **1** (o $\neq 0$), llama a `decision_huir()`.
- Si `$t8` = **0**, llama a `decision_Persigue()`.

## 6.2. `decision_Persigue` (Modo Persecución)

1. **Eje de Movimiento:** Se genera un número aleatorio entre 0 y 1. 0 → Eje X, 1 → Eje Y.
2. **Lógica 4/5 (Perseguir):** Se genera un número aleatorio entre 0 y 4 (`li $a1, 5, syscall 42`).
   - Si el resultado es $0, 1, 2, 3$ (**80** % de probabilidad), el fantasma elige la dirección que **acerca** su coordenada (`$s1` o `$s2`) a la de Pac-Man (`$t1` o `$t2`).
   - Si el resultado es 4 (**20** % de probabilidad), el fantasma elige la dirección **opuesta** (error intencional para evitar IA perfecta).
3. Se repite la lógica si el movimiento elegido (`$s3`) resulta ser un muro (`next_Block()`).

## 6.3. `decision_huir` (Modo Huida)

Es la lógica inversa de `decision_Persigue`. Utiliza la misma aleatoriedad, pero la probabilidad 4/5 se utiliza para elegir la dirección que **aleja** al fantasma de Pac-Man.

## 6.4. `esquina`

Macro de **corrección de dirección**. Si el fantasma se mueve en un pasillo y choca con una pared, esta macro lo fuerza a cambiar su movimiento al eje **perpendicular** (ej. si choca en X, intenta moverse en Y), garantizando que el fantasma no se quede atascado repitiendo la colisión.

# 7. `comer.asm` – Colisiones y Reaparición

## 7.1. `comer`

Macro que compara la posición de Pac-Man (`$s1`, `$s2`) con las coordenadas de cada fantasma (`$t1`, `$t2`).

- **Pac-Man ∩ Fantasma y $t8 = 0 (Normal):**
  - **Acción:** Llama a `apagar()`, reinicia `pontos` a 0, y salta a `Seta` (reinicio de nivel/vida).
- **Pac-Man ∩ Fantasma y $t8 ≠ 0 (Potenciado):**
  - **Acción:** Se reescribe la posición (X, Y) del fantasma a su punto de **respawn inicial** (e.g., F1 → (4, 4)). El contador de reaparición (*offset* 64-76) se establece en **0** (`sw $zero, 64($s7)`) para iniciar el tiempo de inactividad.

## 7.2. `reaparecer`

Macro que gestiona el temporizador de muerte del fantasma.

- **Lógica:** Carga el contador (`Posicao+64, 68, 72` o `76`).
- Si el contador es $< \mathbf{15}$, lo incrementa y retorna $\mathbf{\$s5 = 0}$ (**Inactivo**).
- Si el contador es $\geq \mathbf{15}$, retorna $\mathbf{\$s5 = 1}$ (**Activo/Reaparece**).

# 8. Módulos Auxiliares: `mapa.asm`, `punto.asm`, `control_puntos.asm`

## 8.1. `mapa.asm`

Contiene la macro `set_mapa` que dibuja la geometría estática del laberinto. Utiliza `draw_P` con el color Azul (muro) para pintar línea por línea todas las paredes del mapa.

## 8.2. `punto.asm`

Contiene la macro `set_puntos` que dibuja la distribución inicial de los puntos pequeños y las frutas (Power Pills) en el laberinto. Utiliza `draw_P` con el color Amarillo/Naranja (punto) para marcarlos en el mapa de puntos en memoria.

## 8.3. `control_puntos.asm`

Define el manejo de un **Mapa de Puntos en Memoria** (`Pontos`) separado del mapa de la pantalla.

- `draw_Puntos`**:** Escribe el color (`$s0`) en el mapa de puntos en memoria (`Pontos`), **no en la pantalla de video**. Esto es crucial para marcar qué puntos han sido consumidos (al escribir color negro).
- `cargar_puntos`**:** Lee el color del píxel en la memoria `Pontos` y lo carga en `$s4`. Las macros `next_Block_punto` y `next_Block_fruta` utilizan esta función para verificar la existencia de un punto o fruta antes de que Pac-Man lo çoma".