Documentación del Proyecto - Arkanoid en MIPS Assembly

Índice

Descripción General

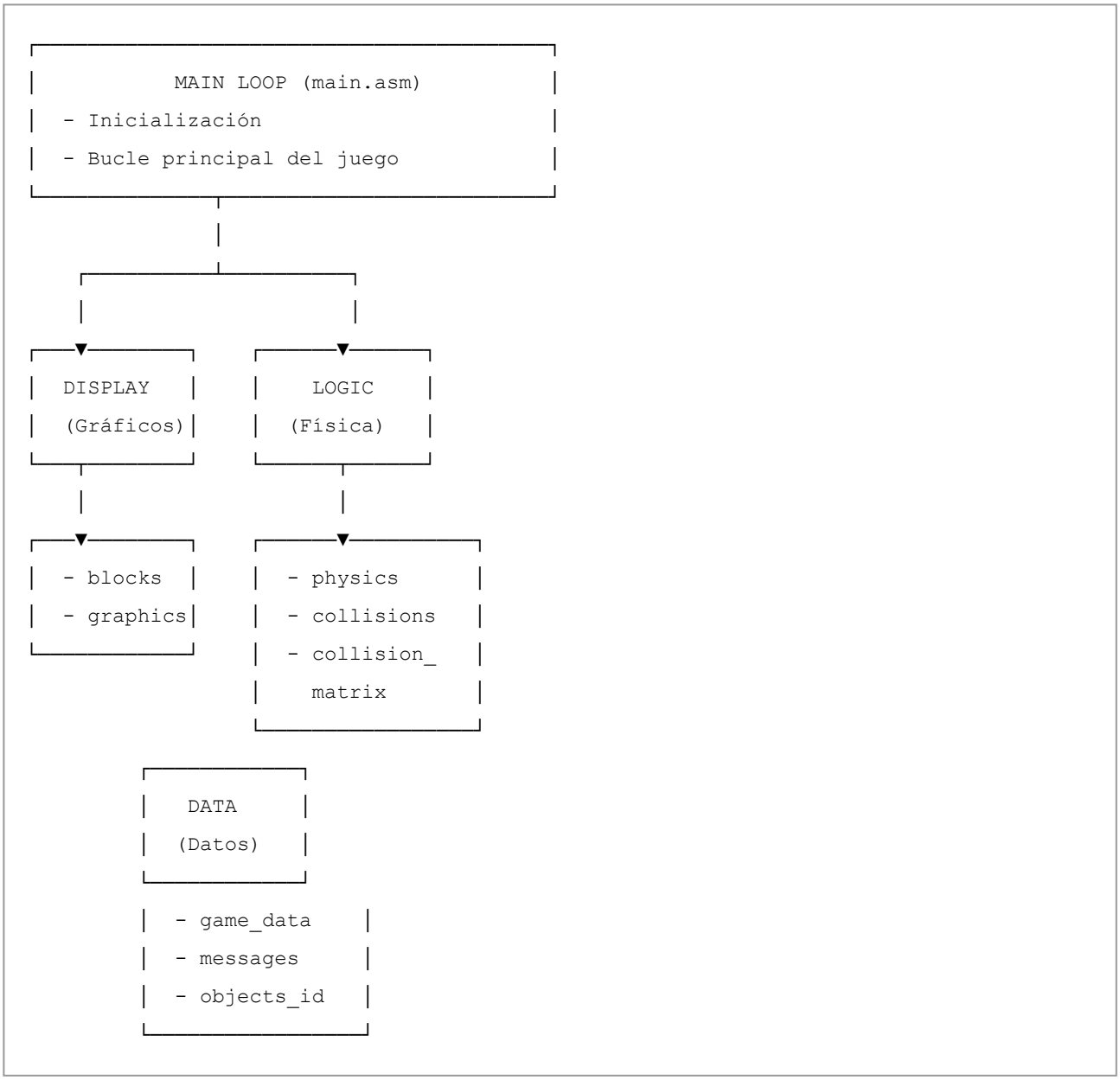
Este proyecto es una implementación del clásico juego **Breakout** (Arkanoid) desarrollado en **MIPS Assembly**. El juego utiliza un sistema de doble buffer para manejar gráficos y colisiones de manera eficiente.

Características Principales

- Resolución: 256x256 píxeles
- 60 bloques destructibles organizados en 6 filas
- Sistema de vidas (3 vidas)
- Sistema de puntuación
- Física de rebote avanzada con ángulos variables
- Detección de colisiones mediante matriz dedicada

Arquitectura de Software

El proyecto sigue una arquitectura modular dividida en capas funcionales:



Módulos del Sistema

1. **Data (Datos)**

Contiene todas las constantes, variables y configuraciones del juego.

2. **Display (Visualización)**

Maneja todo lo relacionado con el renderizado gráfico en el buffer de video.

3. **Input (Entrada)**

Procesa las entradas del teclado del usuario.

4. **Logic (Lógica)**

Implementa la física del juego, detección de colisiones y matriz de colisiones.

Sistema de Buffers

El proyecto utiliza **dos buffers independientes** para optimizar el rendimiento y separar responsabilidades:

Buffer 1: Buffer de Video (Display Address)

- **Dirección**: `0x10010000`
- **Tamaño**: 256×256 píxeles × 4 bytes = 262,144 bytes
- **Propósito**: Renderizado gráfico visible para el usuario

Características:

- Almacena colores en formato RGB (4 bytes por píxel)
- Actualizado constantemente para mostrar el estado visual del juego
- Contiene: paleta, pelota, bloques, fondo

Cálculo de offset:

```
offset = (y * 256 + x) * 4 + displayAddress
```

Buffer 2: Buffer de Colisiones (Collision Matrix)

- **Dirección**: `0x10050000`
- **Tamaño**: 256×256 bytes = 65,536 bytes
- **Propósito**: Detección de colisiones mediante IDs de objetos

Características:

- Cada píxel almacena un byte con el ID del objeto presente
- No se renderiza visualmente
- Permite detección O(1) de colisiones
- IDs disponibles: 0-255

Cálculo de offset:

```
offset = (y * 256 + x) + collisionMatrix
```

Ventajas del Sistema de Doble Buffer

Aspecto	Ventaja
Rendimiento	Detección de colisiones en tiempo constante O(1)
Separación de Responsabilidades	Gráficos y lógica independientes
Escalabilidad	Fácil agregar nuevos tipos de objetos

Aspecto	Ventaja
Debugging	Problemas gráficos no afectan la lógica de colisiones

---

Módulos y Funciones

1. Data Module (`data/`)

`game_data.asm`

Define todas las variables y constantes del juego.

Variables Principales:

```
displayAddress:     .word 0x10010000  # Dirección del buffer de video
collisionMatrix:    .word 0x10050000  # Dirección del buffer de colisiones
screenWidth:        .word 256          # Ancho de pantalla
screenHeight:       .word 256          # Alto de pantalla
```

Colores:

- `bgColor`: Negro (0x00000000)
- `paddleColor`: Blanco (0x00FFFFFF)
- `ballColor`: Rojo (0x00FF0000)
- `blockColor1-4`: Magenta, Cyan, Amarillo, Verde

Configuración de la Paleta:

- Posición inicial: (110, 240)
- Tamaño: 35×8 píxeles
- Velocidad: 4 píxeles/frame

Configuración de la Pelota:

- Posición inicial: (128, 128)
- Tamaño: 6×6 píxeles
- Velocidad inicial: (1, -1)

Configuración de Bloques:

- Tamaño: 20×8 píxeles
- Matriz: 10 columnas × 6 filas = 60 bloques
- Posición inicial: (28, 30)

`messages.asm`

Contiene los mensajes de texto del juego.

Mensajes:

- `msgGameOver`: "=== GAME OVER ==="
- `msgWin`: "=== VICTORIA ==="
- `msgScore`: "Puntuacion final: "
- `msgLives`: "Vidas restantes: "

objects_id.asm

Define los identificadores únicos para cada tipo de objeto en la matriz de colisiones.

IDs de Objetos:

```
OBJ_EMPTY:              0   # Espacio vacío
OBJ_BLOCK_RED:          1   # Bloque rojo normal
OBJ_BLOCK_YELLOW:       2   # Bloque amarillo normal
OBJ_BLOCK_BLUE:         3   # Bloque azul normal
OBJ_BLOCK_GREEN:        4   # Bloque verde normal
OBJ_BLOCK_MAGENTA:      5   # Bloque magenta normal
OBJ_BLOCK_WHITE:        6   # Bloque blanco normal
OBJ_BALL:              14   # Pelota
OBJ_PADDLE:            15   # Paleta
OBJ_WALL:              20   # Paredes
```

---

2. Display Module (`display/`)

graphics.asm

Funciones de renderizado rápido en el buffer de video.

`drawPaddleFast`

Propósito: Dibuja la paleta en el buffer de video sin usar la pila.

Algoritmo:

1. Carga posición (paddleX, paddleY) y dimensiones (paddleWidth, paddleHeight)
2. Itera sobre cada píxel del rectángulo
3. Calcula offset: `(y * 256 + x) * 4`
4. Escribe el color en el buffer de video

Complejidad: O(width × height)

Registros utilizados:

- `$t0-$t9, $s0-$s1`: Temporales para cálculos y loops

`clearPaddleFast`

Propósito: Borra la paleta del buffer de video pintándola con el color de fondo.

Funcionamiento: Idéntico a `drawPaddleFast` pero usando `bgColor` en lugar de `paddleColor`.

`drawBallFast`

Propósito: Dibuja la pelota (6×6 píxeles) en el buffer de video.

Características especiales:

- Dibuja un cuadrado sólido de 6×6
- Utiliza loops anidados para recorrer área
- No guarda `$ra` (función hoja optimizada)

`clearBallFast`

Propósito: Borra la pelota del buffer de video.

Funcionamiento: Similar a `drawBallFast` pero con `bgColor`.

`blocks.asm`

Funciones especializadas para el manejo de bloques.

`drawAllBlocks`

Propósito: Dibuja todos los bloques activos al inicio del juego.

Algoritmo:

```
Para cada fila (0 a 5):
    Para cada columna (0 a 9):
        Si blocks[fila][columna] == 1:
            drawBlock(columna, fila)
```

Parámetros: Ninguno

Complejidad: O(filas × columnas)

`drawBlock`

Propósito: Dibuja un bloque individual en su posición correspondiente.

Parámetros:

- `$a0`: Columna (0-9)
- `$a1`: Fila (0-5)

Algoritmo:

1. Calcula posición X: `blockStartX + (columna * 20)`
2. Calcula posición Y: `blockStartY + (fila * 8)`
3. Selecciona color según fila (4 colores alternados)
4. Dibuja rectángulo de 20×8 píxeles

Colores por fila:

- Fila 0: Magenta (`blockColor1`)
- Fila 1: Cyan (`blockColor2`)
- Fila 2: Amarillo (`blockColor3`)
- Fila 3: Verde (`blockColor4`)
- Fila 4: Magenta (`blockColor1`)
- Fila 5: Cyan (`blockColor2`)

`eraseBlock`

Propósito: Borra un bloque específico del buffer de video.

Parámetros:

- `$a0`: Columna del bloque
- `$a1`: Fila del bloque

Funcionamiento: Igual que `drawBlock` pero pinta con `bgColor`.

---

3. Input Module (`input/`)

`keyboard.asm`

`checkInput`

Propósito: Lee la entrada del teclado y mueve la paleta horizontalmente.

Teclas soportadas:

- `'a'` o `'A'`: Mover izquierda
- `'d'` o `'D'`: Mover derecha

Algoritmo:

1. Verifica si hay tecla presionada (MMIO 0xFFFF0000)
2. Lee el código ASCII de la tecla (MMIO 0xFFFF0004)
3. Compara con 'a', 'A', 'd', 'D'
4. Actualiza `paddleX` según la tecla
5. Aplica límites para evitar salirse de pantalla

Límites de movimiento:

- Mínimo X: 0
- Máximo X: `256 - paddleWidth = 221`

Velocidad de movimiento: `paddleSpeed = 4` píxeles por frame

---

4. Logic Module (`logic/`)

`collision_matrix.asm`

Gestiona la matriz de colisiones en memoria.

`initCollisionMatrix`

Propósito: Inicializa el buffer de colisiones limpiándolo y registrando objetos estáticos.

Algoritmo:

1. **Limpia toda la matriz** (65,536 bytes a 0)
2. **Dibuja paredes laterales** (ID 20):
    - Columna izquierda (x=0)
    - Columna derecha (x=255)
3. **Dibuja techo** (ID 20):
    - Fila superior (y=0)
4. **Registra bloques** (IDs 1-60):
    - Llama a `fillBlocksInMatrix`

Complejidad: O(n) donde n = 65,536

`fillBlocksInMatrix`

Propósito: Registra todos los bloques en la matriz de colisiones.

Algoritmo:

```
Para cada fila (0 a 5):
    Para cada columna (0 a 9):
        Si blocks[fila][columna] == 1:
            ID = fila * 10 + columna + 1
            fillRectInMatrix(x, y, 20, 8, ID)
```

IDs asignados: 1 a 60 (único por bloque)

`fillRectInMatrix`

Propósito: Rellena un rectángulo en la matriz de colisiones con un ID específico.

Parámetros:

- `$a0`: X inicial
- `$a1`: Y inicial
- `$a2`: Ancho
- `$a3`: Alto
- `$t0`: ID del objeto

Algoritmo:

1. Itera sobre cada píxel del rectángulo
2. Calcula offset: `(y * 256 + x) + collisionMatrix`
3. Escribe el byte del ID en esa posición

Uso: Registrar objetos sólidos en el buffer de colisiones

`updatePaddleInMatrix`

Propósito: Actualiza la posición de la paleta en la matriz de colisiones.

Algoritmo:

1. **Limpia posición anterior**: Llama a `clearPreviousPaddlePosition`
2. **Registra nueva posición**: Llama a `fillRectInMatrix` con ID 15

Optimización: Solo actualiza si la paleta se movió

`clearPreviousPaddlePosition`

Propósito: Borra la posición anterior de la paleta calculando dónde estaba antes.

Cálculo de posición anterior:

```
Si se movió a la derecha:
    X_anterior = X_actual - paddleSpeed
Si se movió a la izquierda:
    X_anterior = X_actual + paddleSpeed
```

Algoritmo:

1. Determina dirección del último movimiento
2. Calcula posición anterior
3. Llama a `fillRectInMatrix` con ID 0 (vacío)

collisions.asm

Detecta colisiones consultando la matriz.

`checkCollision`

Propósito: Verifica si hay colisión en una posición específica.

Parámetros:

- `$a0`: Coordenada X
- `$a1`: Coordenada Y

Retorno:

- `$v0`: 1 si hay colisión, 0 si está vacío
- `$v1`: ID del objeto colisionado

Algoritmo:

1. Calcula offset: `(y * 256 + x) + collisionMatrix`
2. Lee el byte en esa posición
3. Si byte != 0: colisión detectada
4. Retorna ID del objeto

Complejidad: O(1) - acceso directo a memoria

physics.asm

Implementa el movimiento y física de la pelota.

`moveBall`

Propósito: Mueve la pelota y gestiona todas las colisiones.

Algoritmo principal:

```
1. Calcular nueva posición (ballX + ballVelX, ballY + ballVelY)
2. Verificar colisiones en 4 esquinas de la pelota (6x6 píxeles)
3. Detectar tipo de colisión:
   - Piso (Y >= 255): Perder vida
   - Paredes/Techo (ID 20): Rebotar
   - Paleta (ID 15): Rebotar con ángulo
   - Bloques (ID 1-60): Rebotar y destruir
4. Aplicar rebote modificando velocidad
5. Mover pelota a nueva posición
6. Actualizar matriz de colisiones
```

Puntos de colisión verificados:

- Esquina superior izquierda: `(ballX, ballY)`
- Esquina superior derecha: `(ballX + 5, ballY)`
- Esquina inferior izquierda: `(ballX, ballY + 5)`
- Esquina inferior derecha: `(ballX + 5, ballY + 5)`

Tipos de rebote:

1. **Rebote horizontal**: Invierte `ballVelX`
2. **Rebote vertical**: Invierte `ballVelY`
3. **Rebote con paleta**: Invierte `ballVelY` + ajusta ángulo

`calculatePaddleBounceAngle`

Propósito: Calcula el ángulo de rebote según dónde golpea la pelota en la paleta.

Sistema de zonas (paleta de 35px):

```
|  Z1  |  Z2  |  Z3  |  Z4  |  Z5  |
|  7px |  7px |  7px |  7px |  7px |
| -2   | -1   |  0   | +1   |  +2  | → ballVelX resultante
```

Algoritmo:

1. Calcula posición relativa: `ballX - paddleX`
2. Determina zona según posición (0-7, 7-14, 14-21, 21-28, 28-35)
3. Asigna velocidad X según zona

Efecto:

- **Zonas extremas**: Rebote más inclinado (±2)
- **Zona central**: Rebote recto (0)

`updateBallInMatrix`

Propósito: Actualiza la posición de la pelota en la matriz de colisiones.

Algoritmo:

1. Limpia área anterior (6×6): `clearPreviousBallPosition`
2. Registra nueva área (6×6): `fillRectInMatrix` con ID 14

`clearPreviousBallPosition`

Propósito: Borra el área anterior de la pelota en la matriz.

Cálculo posición anterior:

```
X_anterior = ballX - ballVelX
Y_anterior = ballY - ballVelY
```

Algoritmo:

1. Calcula coordenadas anteriores
2. Limpia área 6×6 con ID 0

`destroyBlockInMatrix`

Propósito: Destruye un bloque cuando la pelota lo golpea.

Parámetros:

- `$a0`: ID del bloque a destruir (1-60)

Algoritmo:

1. **Busca el bloque** en la matriz recorriendo 65,536 bytes
2. **Marca como vacío** (ID 0) todas las apariciones
3. **Actualiza contadores**:
    - `blocksRemaining--`
    - `score += 10`

Complejidad: O(n) donde n = 65,536 (búsqueda lineal)

Optimización posible: Mantener tabla de posiciones de bloques

`lostBall`

Propósito: Gestiona la pérdida de una vida cuando la pelota cae.

Algoritmo:

1. Decrementa vidas: `lives--`
2. Muestra mensaje con vidas restantes
3. Si `lives > 0`:
    - Llama a `respawn` para reiniciar
4. Si `lives == 0`:
    - Muestra "GAME OVER"
    - Muestra puntuación final
    - Termina el programa

`respawn`

Propósito: Reinicia la posición de la pelota después de perder una vida.

Valores de reinicio:

- `ballX = 128` (centro X)
- `ballY = 120` (centro-arriba Y)
- `ballVelX = 1`
- `ballVelY = -1` (hacia arriba)

Pausa: 1 segundo (1000ms) antes de continuar

---

5. Main Module

`main.asm`

Punto de entrada y bucle principal del juego.

`main`

Propósito: Inicializa el juego y entra en el bucle principal.

Secuencia de inicialización:

1. `initCollisionMatrix`: Prepara buffer de colisiones
2. `updatePaddleInMatrix`: Registra paleta inicial
3. `drawAllBlocks`: Dibuja bloques en pantalla
4. Entra en `mainLoop`

`mainLoop`

Propósito: Bucle principal del juego que se ejecuta cada frame.

Secuencia de operaciones (cada frame):

```
1. BORRAR GRÁFICOS ANTERIORES
   - clearPaddleFast
   - clearBallFast

2. PROCESAR ENTRADA
   - checkInput

3. ACTUALIZAR LÓGICA
   - updatePaddleInMatrix
   - moveBall (incluye detección de colisiones)

4. RENDERIZAR NUEVOS GRÁFICOS
   - drawPaddleFast
   - drawBallFast

5. VERIFICAR VICTORIA
   - Si blocksRemaining == 0 → gameWon

6. DELAY
   - Pausa de 5ms para control de frame rate
```

Frame rate aproximado: 200 FPS (5ms por frame)

`gameWon`

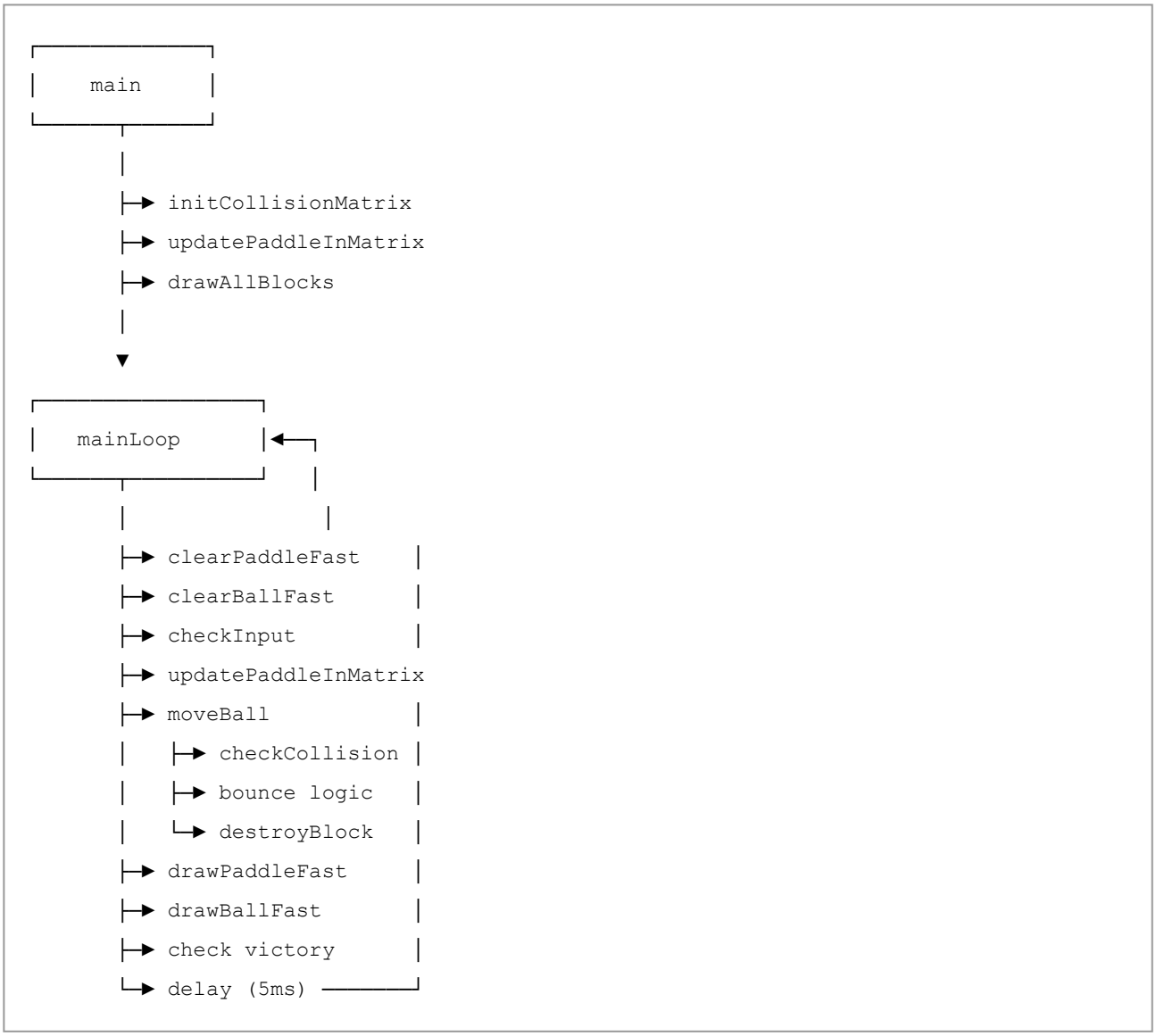
Propósito: Gestiona el final exitoso del juego.

Acciones:

1. Muestra mensaje "=== VICTORIA ==="
2. Termina el programa (syscall 10)

Flujo de Ejecución

Diagrama de Flujo Principal



Flujo de Detección de Colisiones

```
moveBall
  │
  ├─► Calcular nueva posición
  │
  ├─► checkCollision (4 esquinas)
  │     │
  │     ├─► Leer matriz en (x, y)
  │     └─► Retornar ID del objeto
  │
  ├─► Analizar ID colisionado:
  │     ├─► ID 0 (vacío) → No hay colisión
  │     ├─► ID 20 (pared/techo) → Rebotar
  │     ├─► ID 15 (paleta) → Rebotar con ángulo
  │     ├─► ID 1-60 (bloque) → Destruir + Rebotar
  │     └─► Y >= 255 (piso) → lostBall
  │
  ├─► Aplicar física de rebote
  │
  ├─► Mover pelota
  │
  └─► updateBallInMatrix
```

Estructuras de Datos

1. Matriz de Bloques

```
blocks: .word 1,1,1,1,1,1,1,1,1,1  # Fila 0
        .word 1,1,1,1,1,1,1,1,1,1  # Fila 1
        .word 1,1,1,1,1,1,1,1,1,1  # Fila 2
        .word 1,1,1,1,1,1,1,1,1,1  # Fila 3
        .word 1,1,1,1,1,1,1,1,1,1  # Fila 4
        .word 1,1,1,1,1,1,1,1,1,1  # Fila 5
```

- **Tipo**: Array 2D de words (4 bytes)
- **Dimensiones**: 6 filas × 10 columnas = 60 elementos
- **Valores**: 1 = activo, 0 = destruido
- **Acceso**: `blocks[fila * 10 + columna]`

2. Buffer de Video (Display)

```
Dirección: 0x10010000
Tamaño: 256 × 256 × 4 bytes = 262,144 bytes
Formato: [RGBA] por píxel (4 bytes)
```

Mapa de memoria:

```
0x10010000 ┌──────────────────┐
           │  Píxel (0,0)     │ ← 4 bytes
           │  Píxel (1,0)     │
           │  ...             │
           │  Píxel (255,0)   │
           │  Píxel (0,1)     │
           │  ...             │
0x1004FFFC │  Píxel (255,255) │
           └──────────────────┘
```

3. Buffer de Colisiones (Collision Matrix)

```
Dirección: 0x10050000
Tamaño: 256 × 256 × 1 byte = 65,536 bytes
Formato: 1 byte por píxel (ID del objeto)
```

Mapa de memoria:

```
0x10050000 ┌──────────────────┐
           │  ID (0,0)        │ ← 1 byte
           │  ID (1,0)        │
           │  ...             │
           │  ID (255,0)      │
           │  ID (0,1)        │
           │  ...             │
0x1005FFFF │  ID (255,255)    │
           └──────────────────┘
```

Distribución de IDs en la matriz:

```
┌──────────────────────────────────┐
│ 20 20 20 20 20 ... 20 20 20 20│ ← Techo (y=0)
│ 20  0  0  0  0 ...  0  0  0 20│ ← Paredes laterales
│ 20  0  1  1  1 ...  5  5  5 20│ ← Bloques (IDs 1-60)
│ 20  0 11 11 11 ... 15 15 15 20│
│ 20  0 21 21 21 ... 25 25 25 20│
│ 20  0  0  0  0 ...  0  0  0 20│
│ 20  0  0  0  0 ... 14 14 14 20│ ← Pelota (ID 14)
│ 20  0  0  0  0 ...  0  0  0 20│
│ 20  0 15 15 15 ... 15 15 15 20│ ← Paleta (ID 15)
│ 20  0  0  0  0 ...  0  0  0 20│ ← Zona inferior (sin piso)
└──────────────────────────────────┘
```

Optimizaciones Implementadas

1. **Funciones "Fast" sin Stack**

Las funciones de dibujo (`drawPaddleFast`, `clearPaddleFast`, etc.) no usan la pila, lo que reduce overhead.

2. **Detección de Colisiones O(1)**

Gracias a la matriz de colisiones, verificar colisión es una simple lectura de memoria.

3. **Actualización Diferencial**

Solo se borran y redibujan los elementos que se mueven (paleta y pelota), no toda la pantalla.

4. **Delay Mínimo**

Frame rate alto (5ms de delay) para movimiento fluido.

Limitaciones y Posibles Mejoras

Limitaciones Actuales

1. Solo un tipo de bloque (1 golpe)
2. Una sola pelota
3. Sin power-ups implementados
4. Sin niveles adicionales
5. Búsqueda lineal para destruir bloques (O(n))

Mejoras Propuestas

1. **Tabla de hash de bloques**: Mejorar `destroyBlockInMatrix` a O(1)
2. **Bloques con más resistencia**: Usar IDs 7-13 (ya definidos)
3. **Power-ups**: Implementar IDs 16-18
4. **Múltiples pelotas**: Gestionar array de pelotas
5. **Sonido**: Agregar syscalls de audio en colisiones
6. **Niveles**: Cargar diferentes configuraciones de `blocks[]`

Conclusión

Este proyecto demuestra una implementación eficiente de un juego clásico en ensamblador MIPS, utilizando técnicas avanzadas como:

- **Separación de buffers** para optimizar gráficos y lógica
- **Programación modular** para facilitar mantenimiento
- **Algoritmos eficientes** para detección de colisiones
- **Física realista** con sistema de rebote variable

El sistema de doble buffer (video + colisiones) es la clave de la arquitectura, permitiendo un rendimiento excelente mientras mantiene el código organizado y extensible.