15-1-2025

Trabajo Final: Videojuego a la Old-School

Grado de Ingeniería Informática e Inteligencia Artificial

JAVIER REVUELTA FERNÁNDEZ
FRANCISCO JAVIER LIARTE NASARRE

Arquitectura de Ordenadores – 2ºA

~ ÍNDICE ~

Resumen:

En este proyecto se desarrolla un videojuego implementado completamente en ensamblador RISC-V, utilizando el simulador RARS. El objetivo principal es reforzar los conocimientos sobre el uso de subrutinas, la gestión eficiente de la pila, y la interacción con dispositivos de entrada y salida, elementos clave en la programación a bajo nivel.

Para alcanzar este objetivo, los estudiantes han diseñado un juego funcional, implementando su lógica mediante subrutinas modulares y optimizando el uso de registros y memoria. La interacción con el usuario se logra a través de entradas simuladas por teclado y mensajes claros en pantalla.

El desarrollo de este trabajo permitió consolidar el entendimiento sobre el funcionamiento interno de los procesadores y la importancia de un diseño estructurado, sentando las bases para futuras aplicaciones en el ámbito de la arquitectura de ordenadores.

Introducción:

El juego elegido por los estudiantes es una modificación del famoso juego retro “Bomberman”. El juego consiste en alcanzar un punto, moviendo al personaje en un pequeño tablero, esquivando las bombas ya plantadas en este mismo.

La memoria técnica se organiza en las siguientes secciones:

1. Decisiones de diseño: Se presentan las estrategias empleadas en la implementación, como la modularización del código, la optimización de recursos y las técnicas para garantizar un flujo lógico claro.

2. Resultados: Se detallan los logros alcanzados, incluyendo pruebas realizadas y una evaluación de la funcionalidad del videojuego. Todo ello acorde a las especificaciones establecidas en las decisiones de diseño.

3. Conclusión: Se expone una visión global del proyecto, destacando los aprendizajes obtenidos y los retos superados durante el desarrollo.

4. Referencias: Se observa los citados externos al material de la asignatura.

Este trabajo combina aspectos fundamentales de la programación en bajo nivel y diseño lógico, ofreciendo una experiencia práctica que refuerza la comprensión de cómo interactúan software y hardware en un entorno controlado.

Siguiendo esta estructura, se espera ofrecer un documento técnico completo y detallado que muestre de forma transparente todo el proceso de los alumnos durante el transcurso y desarrollo de la práctica planteada.

Decisiones de Diseño:

En esta sección se expondrá el proceso de desarrollo por el que han pasado los alumnos en la elaboración de la práctica, así como el porqué de las decisiones tomadas garantizando de esta forma su justificación y fundamentación.

Este proyecto se ha basado en el famoso videojuego arcade “Bomberman”, pero de una forma más innovadora. Se llegó a la conclusión que para que el proyecto fuese más entretenido y tuviese una finalidad real, como un juego, no era necesario el posicionamiento de una bomba en el tablero y que funcionase como un elemento independiente. Es por eso por lo que se tomó la decisión de que las bombas estuviesen ya plantadas aleatoriamente en el tablero al comenzar, simulando que son minas, y que el jugador tuviese que esquivarlas para llegar a la casilla final para ganar.

Código en C:

Antes de comenzar a desarrollar el videojuego en ensamblador RISC-V, se creó una primera versión de este en C debido a la mayor familiaridad de los alumnos con este lenguaje. Esto permitió visualizar de manera más clara la lógica del juego, establecer un flujo estructurado y plantear las funciones principales del código antes de trasladarlo al nivel de ensamblador, que se traducirán posteriormente a las subrutinas correspondientes.

Como ya se explicó más arriba, el juego desarrollado se centra en un tablero de 8x8 en el que el jugador debe desplazarse desde una posición inicial (esquina superior izquierda del tablero) hasta una celda de victoria representada con una “W” (esquina inferior derecha del tablero), evitando bombas ('*'). El diseño del juego incluye los siguientes elementos y funciones principales

La función init_board se encarga de construir el tablero de juego delimitando los bordes del área jugable con “#”, rellenando el interior del mismo con puntos “.” para representar las casillas libres de bombas, y colocando al jugador “P” en la posición inicial acordada.

A través de la función add_bombs, el código se encarga de definir una celda de victoria en la posición fija (6,6) acordada, y distribuir 5 bombas a lo largo del tablero asegurándose que estas no coinciden con una posición ya ocupada. Es decir, se asegura de que ninguna bomba coincida con la posición de ninguna otra bomba, o con la posición de inicio y final del jugador .

La función display_board imprime el estado actual del tablero, permitiendo al jugador observar su posición y planificar sus movimientos.

Una de las funciones principales que da vida al videojuego es la función process_move, encargada de gestionar la interacción del jugador. Indagando en ella, encontramos que es esta función la que detecta la entrada del usuario (w,a,s,d) para mover al personaje por el tablero en la dirección correspondiente; verifica colisiones con paredes, bombas y la meta, actualizando el estado del juego en consecuencia; y permite al jugador continuar si se mueve hacia una celda vacía.

Por último, en el main se ejecuta un bucle principal muy sencillo que mantiene el juego en marcha hasta que se cumpla una condición de fin: alcanzar la celda de victoria, colisionar con una bomba, o, por otro lado, el jugador decida voluntariamente terminar la partida presionando la tecla “q” (quit).

Código en Ensamblador RISC-V:

Prosiguiendo con la traducción del código en C recientemente explicado a lenguaje ensamblador RISC-V, destacamos algunos puntos clave que de forma conexa han dado lugar al programa resultante. A continuación, se detallan los aspectos más relevantes del diseño implementado:

1. Gestión de Registros y Memoria:

En primer lugar, de cara a la minimización del uso de registros y memoria, todo ello acorde a la convención de uso de registros; decidimos utilizar los registros s0-s4 para almacenar valores esenciales en segmentos específicos del programa, o que necesitan persistir durante varias llamadas a funciones. Además, de acuerdo con la convención de llamadas, es responsabilidad de las funciones que los usen restaurar su contenido original antes de retornarlos. Algunos ejemplos de estos son la dirección base del tablero (s0) o las coordenadas del jugador (s2, s3).

Los registros temporales (t0 - t6) fueron exclusivamente empleados para cálculos intermedios u otras operaciones dentro de bucles, lo que redujo la carga en los registros de propósito general.

Por último, todas las subrutinas ajustaron adecuadamente el marco de pila, almacenando los registros necesarios al inicio y restaurándolos antes de retornar, asegurando así un flujo limpio y modular, evitando conflictos entre subrutinas.

2. Modularización del Código:

El diseño del programa se basó en una estructura modular, dividiendo el flujo lógico en subrutinas específicas acordes a las funciones anteriormente diseñadas en el programa en C, permitiendo una implementación clara y reutilizable.

En la subrutina init_board se crea un bucle sobre la matriz del tablero en el que dependiendo de dónde se encuentre, se le asigna un carácter u otro llamando a la función secundaria “place_wall” siempre y cuando el contador indique que nos encontramos en los bordes del tablero. Si no, se rellenará con espacios vacíos indicados por puntos. Para identificar las posiciones del tablero se hace uso de operaciones como div (división) y rem (resto).

La subrutina add_bombs, como su propio nombre indica, implementó la colocación aleatoria de bombas por el tablero y de la celda de victoria. Se generaron posiciones válidas utilizando llamadas al sistema (ecall) para números aleatorios, asegurándose de que las celdas seleccionadas estuvieran vacías antes de colocar una bomba.

Display_board es la subrutina encargada de imprimir el tablero por pantalla de forma que los caracteres almacenados en memoria adopten la forma del tablero cuadrado de 8x8.

Por último, process_move maneja la lógica de movimiento del jugador, interpretando las entradas (w,a,s,d) a través de la subrutina get_input. Esta última función lee la entrada del usuario y la almacena en un registro temporal para su posterior procesamiento. Una vez leída la entrada del usuario, process_move valida las nuevas posiciones mediante la subrutina validate_move, comprobando choques con paredes, bombas o la meta, y actualizando el estado del juego en función de ello. Si el jugador intenta moverse hacia una pared, el movimiento no se efectuará, quedando en el mismo sitio; si el movimiento es válido, la posición anterior del jugador se vaciará mostrando un punto y la nueva posición se marcará con el carácter “P” representando al jugador; por último, si el jugador se encuentra con una bomba o con la casilla de meta, se terminará el juego mostrando un mensaje de derrota o victoria respectivamente.

Todas estas funciones se efectúan durante la ejecución del bucle de juego game_loop que encontramos en el main hasta que se alcance una condición de fin.

Todas las subrutinas siguen fielmente la convención de llamadas de RISC-V. De acuerdo con esta convención es la función llamadora la encargada de almacenar los registros temporales y cargar los parámetros en los registros a0-a7 para pasárselos a la subrutina que será llamada a través de la instrucción JAL. A continuación, la función llamada, asigna la memoria de pila y guarda los registros que vaya a emplear en la pila para, después, proceder con la ejecución de su código y devolver los resultados cargados en los registros a0 y a1.

Resultados:

Comenzando con la sección de inicialización de variables previas a la ejecución principal, correspondiente al ".data", podemos apreciar como son creados tanto el array donde se almacenarán los caracteres que forman el tablero, como los mensajes que se imprimirán durante la ejecución del programa en función del estado del juego, con el objetivo de guiar al usuario y facilitar su experiencia.

```
.data
    board:      .space 64    # tablero 8x8
    player_pos: .word 1, 1   # posicion x,y del jugador
    newline:    .string "\n"
    prompt:     .string "Move (wasd, q to quit):\n"
    win_msg:    .string "Ganaste!\n"
    lose_msg:   .string "Game Over! Te topaste con una bomba!\n"
    game_over:  .word 0      # 0 = en juego, 1 = game over
```

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	0	0	0	0	0	0	0	0
268501024	0	0	0	0	0	0	0	0
268501056	1	1	1867317258	673211766	1685283191	544284716	1897951092	695495029
268501088	1191184954	1935765089	169960820	1835091712	1984897125	539062885	1948280148	1935765615
268501120	1663067508	1965059695	1646289262	1633840495	2593	0	0	0

Tal y como se observa en el Data Segment, en las dos primeras direcciones de memoria comprobamos que el array que almacena los caracteres del tablero ha sido correctamente inicializado con ceros para cada posición del mismo. Además, justo debajo de este, se aprecian tanto la posición inicial del jugador (x, y), como los diferentes mensajes anteriormente nombrados. Para poder visualizarlos mejor, aplicaremos la opción ASCII de Rars que nos permite traducir los números anteriores a sus letras o símbolos correspondientes.

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
268501024	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
268501056	\0 \0 \0 .	\0 \0 \0 .	o M \0 \n	(e v	d s a w	q ,	q o t	) t i u
268501088	G \0 \n :	s a n a	\n ! e t	m a G \0	v O e	! r e	t e T	s a p c
268501120	c e t	u n o	b a n	a b m o	\0 \0 \n !	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

Es ahora cuando los mensajes son algo más fácilmente legibles. Si nos damos cuenta, leyendo al revés los registros de derecha a izquierda y de arriba abajo, podremos identificar cada uno de los mensajes en su orden correspondiente.

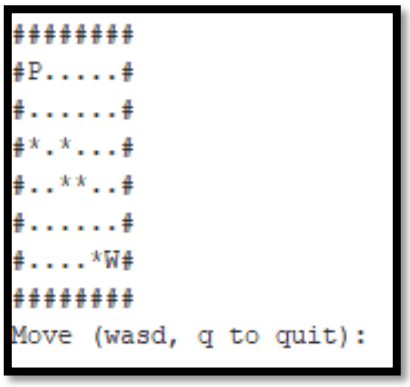
Además, una vez ejecutado el programa y asignados todos los caracteres del tablero a sus posiciones correspondientes, si volvemos a chequear las dos primeras direcciones de memoria podremos apreciar estos nuevos caracteres.

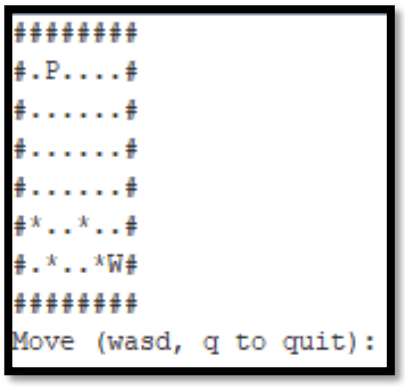
Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	# # # #	# # # #	. . P #	# . . .	. . . #	# . . .	* . * #	# . . .
268501024	* . . #	# . . *	. . . #	# . . .	. . . #	# W * .	# # # #	# # # #

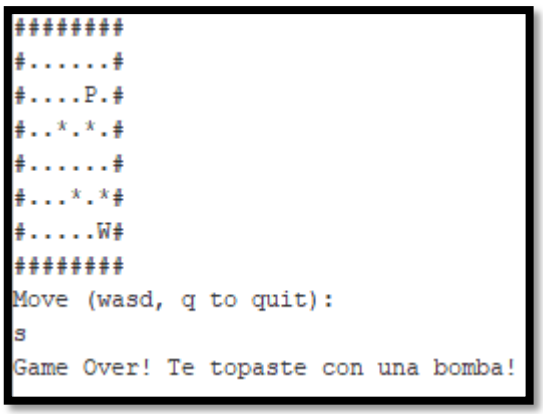
Por último, comprobaremos el correcto funcionamiento del programa en sí, aportando imágenes de las 4 posibles situaciones de movimiento: hacia una

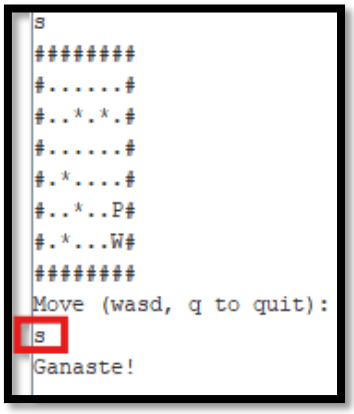
casilla vacía, hacia una pared, hacia una casilla de bomba (derrota) y hacia la meta (victoria).





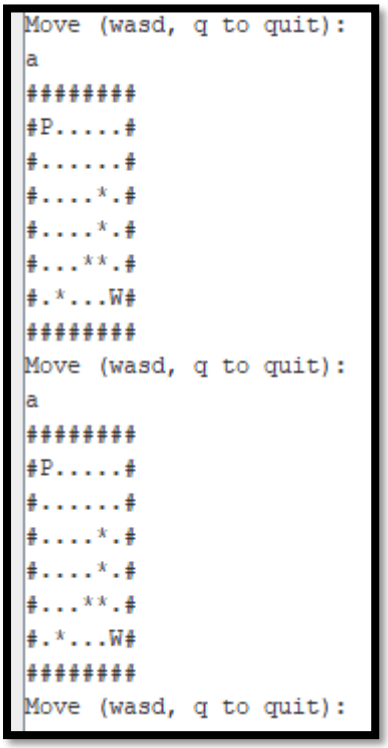
En estas primeras imágenes podemos comprobar cómo, a través de la subrutina display_board anteriormente comentada, el tablero se muestra con su forma correspondiente dando lugar a la situación inicial de juego (imagen izquierda). Después se comprueba el correcto avance del personaje hacia una casilla libre y cómo la casilla en la que se encontraba anteriormente el personaje se restaura con un punto "." Indicando que esta se vuelve a encontrar vacía (imagen derecha).





A continuación, se observa cómo actúa el programa ante las condiciones de finalización de partida ya sea por victoria o derrota. En la imagen superior izquierda podemos ver cómo se muestra el mensaje de derrota y el programa termina antes de imprimir el siguiente tablero con el personaje en la posición de la bomba. Por otro lado, en la imagen superior derecha se aprecia cómo se muestra el mensaje de victoria y el programa termina antes de imprimir el siguiente tablero con el personaje en la posición de la victoria.

Para finalizar con la sección de resultados, se comprueba el último caso posible en el que el jugador trata de moverse hacia una casilla que forma parte del límite del tablero y, por lo tanto, este movimiento no se efectúa ya que no es válido.



Conclusión:

Problemas:

En el siguiente apartado se plantean los problemas que han surgido previamente a la conclusión que muestra la visión global con los problemas ya resueltos.

No se han encontrado problemas graves que impidan la realización de la práctica. Pero se han encontrado algunos que han cambiado el rumbo de su ejecución esperada. Igualmente, esos problemas han sido gestionados para solucionarse o en su defecto puedan ser resueltos de formas innovadoras a las pautadas.

En esta práctica se presenta una subida notable de nivel. Tanto para el manejo como para el entendimiento de RISC-V. El alumno debe haber realizado y entendido correctamente la segunda practica si quiere intentar realizar esta.

El problema principal radica en el manejo del personaje y la constante actualización de este mismo en el tablero. En ensamblador, manejar al personaje y actualizar su posición en el tablero es mucho más complicado que en C debido a la falta de abstracción y las operaciones manuales necesarias. En C, estructuras como arrays y funciones para manejar la entrada y salida simplifican la lógica del juego. En cambio, en ensamblador, se debe calcular manualmente la posición en memoria del tablero, convertir coordenadas bidimensionales a lineales y gestionar cada operación a nivel de registros. Además, las validaciones, como evitar movimientos hacia paredes o bombas, requieren múltiples pasos explícitos, incluyendo saltos condicionales y comparaciones. En C, esto se realiza con condicionales más legibles y concisos.

Otro desafío es que ensamblador no ofrece protección contra errores de memoria, lo que puede causar sobreescrituras accidentales fuera del tablero si los cálculos fallan. Tareas simples, como inicializar un tablero, requieren bucles y comparaciones explícitas, que en C son mucho más directas.

Un problema importante radica en el uso de la pila. Su uso es muy abstracto y ha requerido una cantidad elevada de horas invertidas para poder llevar a cabo su uso en la práctica. Sin duda en esta práctica el entendimiento de la pila es crucial, ya que su uso para optimizar este código es clave. El problema ha podido ser resuelto tanto con las clases del docente de la asignatura como con videos externos y el manual pertinente.

Otro problema para destacar ha sido la impresión por pantalla. La complejidad que trae este problema en su realización en ensamblador frente la simplicidad que tiene en la realización en C requiere un salto grande de comprensión de RISC-V. Este problema ha sido resuelto gracias a la pequeña guía ofrecida por el docente, pero sobre todo mirando la documentación pertinente y realizando pruebas, errando y volviéndolo a intentar hasta conseguirlo.

Las convenciones de registro de RISC-V, han sido manejadas de forma óptima. En esta práctica, el uso de registros temporales no ha supuesto ningún problema, debido a que los alumnos han aprendido tras la realización de varias practicas a su uso. Si bien es cierto mencionar que para evitar que los registros temporales no interfiriesen de forma que no borrasen información necesaria en pasos posteriores también fue un problema que requiso más tiempo del esperado, pero resolverlo mejoró el código con creces. La ejecución y el uso de memoria es prácticamente perfecto.

Se recomienda encarecidamente comprender a la perfección las practicas anteriores ya que el problema de la impresión ha requerido más tiempo del esperado.

Conclusión final:

Tras completar esta práctica, se concluye que los alumnos han adquirido un entendimiento sólido sobre los principios fundamentales de la programación en ensamblador, específicamente en el diseño e implementación de un sistema interactivo basado en RISC-V. El desarrollo de un juego ensamblador como ejercicio práctico ha permitido reforzar conceptos clave como el uso eficiente de registros, la manipulación de la pila y la implementación de estructuras lógicas a bajo nivel, que son esenciales para el control de flujo y la interacción con el hardware.

El planteamiento de los alumnos ha consistido en trasladar la lógica de un sistema de juego desde un enfoque de alto nivel hacia el nivel más básico, abordando tareas como la inicialización de un tablero en memoria, la generación de elementos aleatorios (bombas) y la gestión de entradas del usuario para mover al jugador. Son claves las decisiones sobre el manejo de la memoria, la pila y los registros impactan directamente en el rendimiento y la funcionalidad del programa.

Además, los desafíos técnicos, como la validación de movimientos, la detección de colisiones con bombas y la impresión dinámica del estado del juego, han desarrollado su habilidad para depurar y optimizar programas de bajo nivel.

Los alumnos han afianzado habilidades prácticas y teóricas esenciales, sino que también ha preparado a los estudiantes para enfrentar problemas de mayor complejidad en entornos donde los recursos computacionales son limitados.

Referencias:

1. *Instrucciones de ensamblador:*
   RISC-V Assembler Cheat Sheet - Project F

2. *Comprensión del lenguaje ensamblador:*
   - Simulador RARS, para procesadores RISC V - myTeachingURJC/2018-19-LAB-AO GitHub Wiki (github-wiki-see.page)
   - DSD S5 RISC-V 19 Ejecución de factorial recursivo en ensamblador