PRÁCTICA DE LABORATORIO 3: INFORME

Luis Romero V26.729.561

Julio 2025

1 ¿Cómo se organiza la memoria cuando un sistema utiliza memory-mapped I/O? ¿En qué región de memoria se suelen mapear los dispositivos? ¿Qué implicaciones tiene para las instrucciones `lw` y `sw`?

Cuando un sistema emplea *memory-mapped I/O*, parte del espacio de direcciones de memoria se reserva para comunicarse con los periféricos. Cada dispositivo se ve como si fuera una dirección de RAM: leer o escribir en esa dirección en realidad acciona un registro del hardware. Normalmente se elige una región alta de la memoria (por ejemplo, a partir de 0xFFFF0000 en muchos diseños MIPS) para no interferir con código y datos convencionales. Para las instrucciones `lw` y `sw` esto no cambia nada sintácticamente: se usan exactamente igual que con direcciones de RAM. Sin embargo, semánticamente la CPU sabe—gracias al decoder del bus—que esa dirección no es memoria DRAM sino el registro de un periférico, y en lugar de activar la memoria activa la lógica de E/S. De esa forma, `lw` desde 0xFFFF0004 podría devolver el estado de un sensor, y `sw` a 0xFFFF0000 podría iniciar una medición.

2 ¿Cuál es la principal diferencia entre memory-mapped I/O y la entrada/salida por puertos? ¿Qué ventajas y desventajas tiene cada enfoque? ¿Por qué MIPS32 utiliza principalmente memory-mapped I/O?

La diferencia clave es que en *port-mapped I/O* (o *isolated I/O*) existe un espacio de direcciones separado y se utilizan instrucciones específicas (`in`, `out`) para acceder a puertos. En cambio, *memory-mapped I/O* integra todo en un único espacio y reutiliza `lw`/`sw`.

Ventajas de memory-mapped I/O

- Simplifica el conjunto de instrucciones: no se necesitan opcodes extra, basta con `lw`/`sw`.
- Permite usar punteros y estructuras en C para acceder a dispositivos, lo cual es muy cómodo en sistemas embebidos.

Desventajas

- Se "dispone" de parte del espacio de direcciones para E/S, reduciendo la memoria de usuario.
- Hay que vigilar el mapeo para no solapar dispositivos accidentalmente.

MIPS32 elige mayormente memory-mapped I/O porque su filosofía RISC busca un ISA minimalista y consistente: agrega complejidad mínima a la CPU dejando que el diseño de placa/SoC gestione el mapeo.

3 En un sistema con memory-mapped I/O: ¿Qué problemas pueden surgir si dos dispositivos usan direcciones solapadas? ¿Cómo se evita este conflicto?

El solapamiento de direcciones significa que dos periféricos responderían a la misma dirección, provocando lecturas/escrituras ambiguas: el hardware no sabría a qué dispositivo dirigir la transacción. Esto puede corromper datos o disparar comportamientos erráticos.
Para evitarlo, durante el diseño de la placa (o del mapa de memoria del SoC) se asignan rangos disjuntos a cada periférico y se usan decodificadores de dirección en el bus de sistema. A nivel de firmware o sistema operativo, se documentan las bases y longitudes de cada región para que el software no interfiera. Además, los manuales de la arquitectura MIPS32 especifican cuidadosamente el rango reservado para E/S mappeada.

4 ¿Por qué se considera que el memory-mapped I/O simplifica el diseño del conjunto de instrucciones de un procesador? ¿Qué tipo de instrucciones adicionales serían necesarias si se usara E/S por puertos?

Con memory-mapped I/O, el procesador implementa solo `lw`, `sw` y no distingue si la dirección es RAM o periférico. El circuito de bus y lógica de decodificación se encargan de enrutar la transacción.

Si se usara E/S por puertos, habría que añadir al ISA instrucciones dedicadas, por ejemplo `in port, reg` y `out reg, port`. Cada una implicaría opcodes nuevos, decodificación especial y lógica extra en la CPU para gestionar ese espacio aparte. Eso va en contra del principio RISC de mantener el set de instrucciones tan pequeño y uniforme como sea posible.

5 ¿Qué ocurre a nivel del bus de datos y direcciones cuando el procesador accede a una dirección de memoria que corresponde a un dispositivo? ¿Cómo sabe el hardware que debe acceder a un periférico en lugar de la RAM?

Al ejecutar `lw $t0, 0xFFFF0004`, el procesador coloca en el bus de direcciones la constante `0xFFFF0004` y activa la señal de lectura. El controlador del bus incluye un decodificador de rangos: si la dirección cae en la región mapeada a E/S, la señal de chip-select de la RAM permanece inactiva y se activa el chip-select del periférico. El periférico entonces coloca su dato en el bus de datos, que la CPU lee en la fase MEM de la instrucción. El mismo mecanismo funciona al revés para `sw`, con señales de escritura.

6 ¿Es posible que un programa normal (sin privilegios) acceda a un dispositivo mapeado en memoria? ¿Qué mecanismos de protección existen para evitar accesos no autorizados?

En un sistema con paginación o protección de memoria, el hardware de MMU define permisos en cada página: lectura, escritura, ejecución y nivel de privilegio. Los dispositivos mapeados suelen asignarse a páginas que solo el kernel (modo supervisor) puede usar. Si un proceso de usuario intenta `lw`/`sw` a esa dirección, la MMU provoca una excepción de protección (page fault o bus error), saltando al kernel. Así se evita que aplicaciones sin privilegio interfieran con hardware crítico.

7 ¿Qué técnicas se pueden emplear para evitar esperas activas innecesarias al interactuar con dispositivos?

Las *esperas activas* (*busy-wait*) consumen ciclo de CPU sin hacer trabajo útil. Para aliviarlas, se usan:

- **Interrupciones:** El periférico genera una `IRQ` cuando cambia de estado (por ejemplo, medición lista). La CPU libera la ejecución de otras tareas y atiende la interrupción al ocurrir el evento.
- **DMA (Direct Memory Access):** El DMA transfiere datos periférico–memoria sin intervención de la CPU, liberándola para otras tareas.
- **Sleep y wake-up:** El software puede entrar en modo bajo consumo (`wait-for-interrupt`) en lugar de hacer `while(estado==0)`.

8 Análisis y Discusión de los Resultados

En esta práctica he profundizado en cómo el processor y el bus cooperan para comunicarse con periféricos usando un único espacio de direcciones. El enfoque de *memory-mapped I/O* encaja con la filosofía RISC de MIPS32, que busca simplicidad del ISA delegando la complejidad en el diseño del bus y la memoria. Al implementar en ensamblador las rutinas de sensor y tensión arterial, vi de primera mano cómo `lw`/`sw` pueden servir para datos o E/S según la dirección apuntada.
La comparación con port-mapped I/O me hizo apreciar aún más la elegancia de un espacio unificado: sin necesidad de instrucciones especiales, podemos invocar dispositivos con el mismo conjunto de operaciones que usamos para datos normales. Sin embargo, esto también requiere disciplina en el mapeo de direcciones y en la protección de memoria para evitar accesos indebidos.
Finalmente, la discusión de técnicas como interrupciones o DMA muestra que, más allá de la simple rutina de sondeo, los sistemas reales combinan varios mecanismos para maximizar la eficiencia y minimizar el uso de ciclos de CPU en espera activa. Esta experiencia refuerza la relevancia de entender tanto el ISA como la arquitectura de sistema en aplicaciones embebidas y de sistemas operativos.