Práctica de Laboratorio 3: Memory-Mapped I/O en MIPS32

Orlando Bonilla 30869545 Enyerber Flores 27157754

Julio 2025

1. Código MIPS32

1.1. Ejercicio 1: Sensor de Temperatura

```
.data
    temp_init: .asciiz "Inicializando sensor...\n"
    temp_ready: .asciiz "Sensor listo. Temperatura: "
    temp_error: .asciiz "Error en sensor\n"
    SensorControl: .word 0xFFFF0010
    SensorEstado: .word 0xFFFF0014
    SensorDatos: .word 0xFFFF0018

.text
main:

    la $a0, temp_init
    li $v0, 4
    syscall


    jal InicializarSensor
    jal LeerTemperatura

    beq $v1, -1, error
    j mostrar_temp

error:
    la $a0, temp_error
    li $v0, 4
    syscall
    j end

mostrar_temp:
```

```
    move $t2, $v0


    la $a0, temp_ready
    li $v0, 4
    syscall


    move $a0, $t2
    li $v0, 1
    syscall

end:
    li $v0, 10
    syscall

InicializarSensor:
    lw $t0, SensorControl
    li $t1, 0x2
    sw $t1, 0($t0)
    jr $ra

LeerTemperatura:
    lw $t0, SensorEstado
    lw $t1, 0($t0)
    beq $t1, -1, error_sensor
    beq $t1, 0, no_listo

    lw $t0, SensorDatos
    lw $v0, 0($t0)
    li $v1, 0
    jr $ra

no_listo:
    li $v0, 0
    li $v1, -1
    jr $ra

error_sensor:
    li $v0, 0
    li $v1, -1
    jr $ra
```

1.1.1. Ejercicio 2: Módulo de Tensión Arterial

```
.data
    msg_inicio:    .asciiz "Iniciando medición de tensión arterial...\n"
    msg_espera:    .asciiz "Esperando resultados...\n"
    msg_resultado: .asciiz "Resultados:\nSistólica: "
    msg_barra:     .asciiz " / "
    msg_diastolica:.asciiz "\nDiastólica: "
    msg_error:     .asciiz "Error en la medición\n"

    # Direcciones MMIO estándar en MARS
    TensionControl: .word 0xFFFF0020
    TensionEstado:  .word 0xFFFF0024
    TensionSistol:  .word 0xFFFF0028
    TensionDiastol: .word 0xFFFF002C

.text
main:
    # Mensaje de inicio
    la $a0, msg_inicio
    li $v0, 4
    syscall

    jal controlador_tension

    # Verificar si hubo error (en MARS siempre retorna valores)
    bltz $v0, error_medicion

    # Mostrar resultados
    la $a0, msg_resultado
    li $v0, 4
    syscall

    move $a0, $v0   # Valor sistólico
    li $v0, 1
    syscall

    la $a0, msg_barra
    li $v0, 4
    syscall

    move $a0, $v1   # Valor diastólico
    li $v0, 1
    syscall

    j exit
```

```
error_medicion:
    la $a0, msg_error
    li $v0, 4
    syscall

exit:
    li $v0, 10
    syscall

controlador_tension:
    # Mensaje de espera
    la $a0, msg_espera
    li $v0, 4
    syscall

    # Iniciar medición
    lw $t0, TensionControl
    li $t1, 1
    sw $t1, 0($t0)

    # Esperar a que esté listo (polling)
    espera:
        lw $t0, TensionEstado
        lw $t1, 0($t0)
        beq $t1, 0, espera

    # Leer resultados
    lw $t0, TensionSistol
    lw $v0, 0($t0)

    lw $t0, TensionDiastol
    lw $v1, 0($t0)

    jr $ra
```

2. Respuestas a Preguntas

2.1. 1. Organización de memoria con memory-mapped I/O

En sistemas con memory-mapped I/O, los dispositivos periféricos se mapean en el espacio de direcciones de memoria principal. Los registros de control, estado y datos de los dispositivos aparecen como posiciones de memoria específicas.

Los dispositivos suelen mapearse en regiones reservadas de la memoria, típicamente en las zonas altas del espacio de direcciones. Por ejemplo, en MIPS32

es común encontrar dispositivos mapeados a partir de la dirección 0xFFFF0000. Esta organización permite que las instrucciones de carga (lw) y almacenamiento (sw) puedan acceder tanto a la memoria principal como a los registros de los dispositivos periféricos de manera uniforme.

2.2. 2. Comparación entre memory-mapped I/O y E/S por puertos

La principal diferencia radica en cómo se direccionan los dispositivos. En memory-mapped I/O, los dispositivos comparten el espacio de direcciones con la memoria, mientras que en E/S por puertos existe un espacio de direcciones separado exclusivo para dispositivos.

Memory-mapped I/O ofrece la ventaja de utilizar el mismo conjunto de instrucciones para acceder tanto a memoria como a dispositivos, simplificando el diseño del procesador. Sin embargo, consume espacio de direcciones que podría usarse para memoria. La E/S por puertos requiere instrucciones especiales pero mantiene separados los espacios de memoria y dispositivos.

MIPS32 utiliza principalmente memory-mapped I/O porque se alinea con su filosofía de diseño RISC, que favorece un conjunto reducido de instrucciones versátiles en lugar de operaciones especializadas.

2.3. 3. Conflictos por direcciones solapadas

Cuando dos dispositivos comparten la misma dirección de memoria mapeada, se producen conflictos que pueden generar comportamientos impredecibles. Esto puede manifestarse como escrituras en el dispositivo incorrecto, lecturas corruptas o incluso fallos del sistema.

Para evitar estos conflictos, los diseñadores de hardware implementan esquemas de asignación de direcciones únicas para cada dispositivo. Durante la fase de diseño del sistema, se define cuidadosamente el mapa de memoria, asignando rangos no superpuestos a cada periférico. Adicionalmente, se utilizan circuitos decodificadores que garantizan que solo un dispositivo responda a cada dirección específica.

2.4. 4. Simplificación del conjunto de instrucciones

El memory-mapped I/O simplifica significativamente el conjunto de instrucciones del procesador al eliminar la necesidad de operaciones especializadas para entrada/salida. En lugar de requerir instrucciones específicas como IN y OUT, el sistema puede utilizar las mismas instrucciones de carga y almacenamiento que ya existen para acceder a la memoria.

Si MIPS32 utilizara E/S por puertos, necesitaría incorporar nuevas instrucciones en su ISA, lo que aumentaría la complejidad del diseño. Además, requeriría buses y lógica de control adicionales para manejar el espacio de direcciones separado que este enfoque requiere.

2.5. 5. Operación a nivel del bus

Cuando el procesador accede a una dirección mapeada a un dispositivo, el hardware realiza un proceso específico. El bus de direcciones lleva la solicitud a todos los dispositivos conectados, pero solo el dispositivo cuya dirección coincide responde. Un decodificador de direcciones determina si la dirección corresponde a memoria RAM o a un periférico.

El hardware sabe que debe acceder a un periférico en lugar de la RAM gracias a circuitos combinacionales que comparan la dirección con rangos predefinidos. Cuando la dirección cae dentro del rango asignado a dispositivos, se activan señales de control específicas que habilitan la comunicación con el periférico correspondiente.

2.6. 6. Protección de accesos

En la mayoría de los sistemas modernos, los programas normales ejecutados en modo usuario no pueden acceder directamente a dispositivos mapeados en memoria. Los mecanismos de protección incluyen bits de privilegio en las tablas de paginación, registros de control del MMU y supervisión del sistema operativo.

Estos mecanismos evitan accesos no autorizados asignando permisos específicos a cada página de memoria. Las áreas mapeadas a dispositivos suelen marcarse como accesibles solo en modo kernel, generando una excepción si un programa en modo usuario intenta acceder a ellas.

2.7. 7. Alternativas a esperas activas

Las esperas activas (polling) consumen recursos del procesador innecesariamente. Entre las técnicas más eficientes se encuentran:

Las interrupciones permiten que el dispositivo notifique al procesador cuando está listo, liberando la CPU para otras tareas durante la espera. El DMA (Acceso Directo a Memoria) permite transferencias de datos entre dispositivos y memoria sin intervención del procesador. Otra alternativa es el polling programado, donde se verifican los dispositivos a intervalos regulares en lugar de continuamente.

3. 8.Análisis y Discusión de Resultados

Los ejercicios implementados demuestran los principios fundamentales del memory-mapped I/O en MIPS32. El código del sensor de temperatura muestra un patrón típico de inicialización y lectura de dispositivos, donde es esencial verificar los registros de estado antes de operar con los datos.

En el caso del controlador de tensión arterial, se evidencia el uso de espera activa, que aunque simple, no es la solución más eficiente. En sistemas reales, este enfoque podría mejorarse con interrupciones para optimizar el uso de la CPU.

Los resultados confirman que el memory-mapped I/O proporciona un modelo de programación consistente para interactuar con dispositivos, aunque requiere

un cuidadoso manejo de los registros de control y estado. La principal limitación observada es la necesidad de verificar manualmente los estados de los dispositivos, lo que puede llevar a código redundante.

Estos ejercicios subrayan la importancia de comprender los mapas de memoria y los protocolos de comunicación con dispositivos en sistemas embebidos. También destacan cómo el diseño de MIPS32, al favorecer la simplicidad, delega en el programador ciertas responsabilidades que en otras arquitecturas podrían manejarse automáticamente.