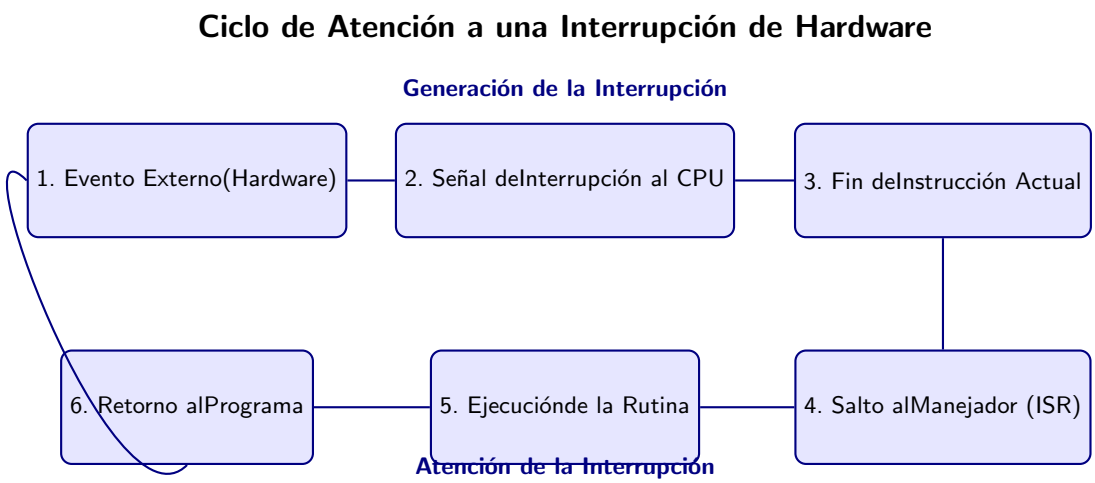
practica4

Abraham Quintero
Jesus Contreras

July 2025

1.



Explicación detallada del ciclo:

1. Ocurre un evento físico (dispositivo, timer, etc.)
2. El hardware genera señal de interrupción (IRQ)
3. CPU termina instrucción actual y guarda estado (PC, registros)
4. Salto a ISR mediante vector de interrupciones
5. Ejecución del manejador (atención al dispositivo)
6. Restauración del estado y retorno al programa original

Figura 1: Diagrama del ciclo completo de atención a interrupciones hardware

2. Comparación entre Gestión de E/S por Sondeo e Interrupciones

2.1. Método de Sondeo (Polling)

Funcionamiento: El CPU verifica constantemente el estado de los dispositivos mediante un bucle activo. En cada iteración, comprueba si el dispositivo necesita atención, consumiendo ciclos de procesador continuamente.

Ventajas:

- Implementación simple y directa
- Comportamiento determinista (ideal para sistemas de tiempo real críticos)
- No requiere hardware adicional
- Fácil de depurar y monitorear

Desventajas:

- Ineficiencia energética (CPU siempre activo)
- Desperdicio significativo de ciclos de procesamiento
- Latencia dependiente de la frecuencia de verificación
- Escalabilidad limitada con múltiples dispositivos

Ejemplo típico en código: El CPU ejecuta un bucle que verifica un registro de estado del dispositivo, continuando con el procesamiento solo cuando detecta que los datos están listos.

2.2. Método por Interrupciones

Funcionamiento: Los dispositivos generan señales de interrupción cuando requieren atención. El CPU suspende su actividad actual, salva el contexto, ejecuta una rutina de servicio (ISR) específica para atender el dispositivo, y luego restaura el contexto para continuar.

Ventajas:

- Máxima eficiencia del CPU (activo solo cuando es necesario)
- Bajo consumo energético
- Respuesta inmediata a eventos
- Escalabilidad óptima para múltiples dispositivos
- Priorización flexible de eventos

Desventajas:

- Mayor complejidad de implementación
- Overhead por gestión de contexto (salvar/restaurar registros)
- Requiere hardware especializado (controlador de interrupciones)
- Comportamiento menos predecible en términos temporales

Ejemplo típico: Cuando un dispositivo como un teclado tiene datos disponibles, interrumpe al CPU, que ejecuta una rutina específica para leer el buffer de entrada antes de retomar la tarea previa.

2.3. Diferencias Clave

Uso del CPU: El sondeo mantiene al CPU ocupado verificando estados incluso cuando no hay trabajo real que hacer, mientras que las interrupciones permiten que el procesador se dedique a otras tareas hasta que sea realmente necesario atender un dispositivo.

Latencia: En el sondeo, el tiempo de respuesta depende directamente de la frecuencia con la que se verifica el dispositivo. Con interrupciones, la latencia está determinada principalmente por el hardware y es generalmente más consistente.

Implementación: El sondeo puede implementarse con código mínimo y sin soporte hardware especial. Las interrupciones requieren mecanismos para salvar/restaurar contexto, vectores de interrupción y rutinas de servicio bien definidas.

Escalabilidad: Mientras que agregar más dispositivos en un sistema por sondeo ralentiza proporcionalmente el rendimiento, las interrupciones permiten manejar múltiples dispositivos eficientemente mediante prioridades y manejo adecuado de colas.

2.4. Recomendaciones de Uso

Use sondeo cuando:

- Los tiempos de respuesta del dispositivo son extremadamente rápidos
- Se requiere comportamiento completamente predecible
- Los recursos hardware son muy limitados
- La simplicidad de implementación es prioritaria

Prefiera interrupciones cuando:

- Los dispositivos tienen tiempos de respuesta variables o largos
- La eficiencia energética es importante
- El sistema debe manejar múltiples dispositivos concurrentemente
- Se necesita máxima utilización del CPU para otras tareas

3. Ventajas del Uso de Interrupciones

Las interrupciones ofrecen importantes beneficios en la gestión eficiente del procesador:

- **Maximización del tiempo útil:**
  - El procesador puede ejecutar otras tareas mientras espera eventos de E/S
  - No consume ciclos verificando dispositivos innecesariamente
  - Aprovechamiento óptimo de la capacidad de cálculo
- **Reducción del consumo energético:**
  - Permite poner el procesador en estados de bajo consumo durante esperas
  - Elimina el gasto energético de bucles de sondeo activos
  - Ideal para sistemas con restricciones de energía (móviles, embebidos)
- **Respuesta inmediata a eventos:**
  - Atención en tiempo real a dispositivos cuando lo requieren
  - Latencia predecible determinada por el hardware

- No depende de frecuencias de muestreo como en el sondeo

- **Escalabilidad con múltiples dispositivos:**
  - Manejo eficiente de numerosos periféricos simultáneamente
  - Priorización inteligente mediante controlador de interrupciones
  - El costo computacional no aumenta linealmente con más dispositivos
- **Arquitectura más limpia:**
  - Separación clara entre código principal y rutinas de servicio
  - Mejor organización del flujo del programa
  - Más fácil de mantener y ampliar funcionalidades

3.1. Impacto en el Rendimiento

El uso de interrupciones mejora significativamente el rendimiento del sistema porque:

- Elimina el *overhead* por verificación constante de estados
- Reduce los conflictos por acceso a buses de E/S
- Permite paralelismo real entre cómputo y operaciones de E/S
- Facilita la implementación de sistemas multitarea

4. Registros para Gestión de Interrupciones en MIPS32

4.1. Registros del Coprocesador 0 (CP0)

El conjunto de instrucciones MIPS32 utiliza registros especiales del Coprocesador 0 para gestionar interrupciones. Los principales son:

4.2. Status Register (SR - $12)

Registro de estado con los siguientes campos relevantes:

- IE (bit 0): Interrupt Enable - Habilita globalmente las interrupciones cuando está a 1
- EXL (bit 1): Exception Level - Indica que se está ejecutando una excepción cuando está a 1
- IM (bits 15-8): Interrupt Mask - Habilita interrupciones específicas (1=habilitada)
- KSU (bits 4-3): Kernel/Supervisor/User Mode - Indica el nivel de privilegio actual

4.3. Cause Register ($13)

Registro de causa con los siguientes campos:

- IP (bits 15-8): Interrupt Pending - Muestra qué interrupciones están pendientes
- ExcCode (bits 6-2): Exception Code - Identifica el tipo de excepción (6 para interrupción hardware)

4.4. EPC Register ($14)

Registro que almacena el Program Counter (PC) en el momento de ocurrir la interrupción, permitiendo retornar a la instrucción correcta después de atenderla.

4.5. Config Register ($16)

Registro de configuración del sistema que puede afectar el manejo de interrupciones, aunque su uso principal es para configuración general del procesador.

4.6. Ejemplo de Configuración

Para habilitar interrupciones en MIPS32 se debe:
1. Configurar los bits IM en el registro Status para habilitar las interrupciones deseadas 2. Poner a 1 el bit IE en el registro Status para habilitar globalmente las interrupciones 3. Configurar el vector de interrupciones en memoria 4. Implementar las rutinas de servicio de interrupción (ISR)

4.7. Flujo de Atención de Interrupciones

Cuando ocurre una interrupción:
1. El procesador completa la instrucción actual en ejecución 2. Guarda el PC en el registro EPC 3. Pone a 1 el bit EXL en el registro Status 4. Salta a la dirección del vector de interrupciones 5. La rutina ISR debe: - Identificar la fuente de interrupción (leyendo Cause) - Atender el dispositivo que generó la interrupción - Limpiar la condición de interrupción - Usar la instrucción ERET para retornar
La instrucción ERET restaura el estado anterior poniendo EXL a 0 y retomando la ejecución en la dirección almacenada en EPC.

5. Preservación del Contexto en Rutinas de Interrupción

5.1. Necesidad de Guardar el Contexto

Al entrar en una Rutina de Servicio de Interrupción (ISR), es **imprescindible** guardar el contexto del programa interrumpido por las siguientes razones técnicas:

5.2. Integridad del Estado del Procesador

- Los registros del CPU contienen valores críticos del programa interrumpido
- Cualquier modificación durante la ISR corrompería el estado original
- Sin preservación, al retornar el programa se comportaría erráticamente

5.3. Transparencia de la Interrupción

- Las interrupciones deben ser transparentes al programa principal
- El programa no debe detectar que fue interrumpido (excepto por el retardo)
- Requiere que todos los registros mantengan sus valores originales

5.4. Registros Críticos a Preservar

- **Registros de propósito general** ($t0-$t9, $a0-$a3, etc.)
- **Registro de estado** (SR) - Contiene flags importantes
- **Contador de programa** (PC) - Guardado automáticamente en EPC
- **Registros de coprocesador** - Si son utilizados por el programa

5.5. Mecanismos de Preservación

5.6. Salvado Manual

En arquitecturas como MIPS32, el programador debe guardar explícitamente:

- Los registros que la ISR modificará
- El registro $ra si usa llamadas a subrutinas
- Flags importantes del registro de estado

5.7. Ejemplo en MIPS32

```
ISR:
    # Guardar contexto
    sw $t0, save_t0
    sw $t1, save_t1
    mfc0 $k0, $12       # Leer Status
    sw $k0, save_status

    # Cuerpo de la ISR
    ...

    # Restaurar contexto
    lw $k0, save_status
    mtc0 $k0, $12       # Escribir Status
    lw $t0, save_t0
    lw $t1, save_t1
    eret
```

5.8. Consecuencias de No Preservar

- **Corrupción de datos**: Valores de registros se pierden
- **Fallos aleatorios**: Comportamiento no determinista
- **Problemas de reinicio**: El programa no puede continuar correctamente
- **Dificultad de depuración**: Errores difíciles de rastrear

5.9. Consideraciones de Rendimiento

- El salvado/restauración añade overhead (15-30 ciclos típicos)
- Se optimiza guardando solo los registros necesarios
- En sistemas críticos se usan registros dedicados para ISRs
- Arquitecturas modernas implementan salvado automático parcial

6. Excepciones en Arquitectura MIPS32

6.1. Situaciones Generadoras de Excepciones

En MIPS32, las excepciones pueden generarse en las siguientes situaciones:

1. **Desbordamiento aritmético (Overflow)**:

- Ocurre en operaciones como `add`, `sub` cuando el resultado excede el rango representable
- Ejemplo: $2^{31} + 2^{31}$ en enteros de 32 bits con signo

2. **Fallo de dirección (Address Error)**:
   - Acceso a direcciones no alineadas (ej. `lw` en dirección no múltiplo de 4)
   - Acceso a memoria fuera del espacio direccionable
3. **Instrucción no implementada (Reserved Instruction)**:
   - Intento de ejecutar un código de operación no definido
   - Ejecución de instrucción privilegiada en modo usuario
4. **Interrupción externa (Hardware Interrupt)**:
   - Señal de dispositivos periféricos (timer, UART, etc.)
   - Requiere configuración previa en registros `Status` y `Cause`
5. **Fallo de página TLB (TLB Miss)**:
   - Cuando la traducción de dirección virtual a física falla
   - Ocurre tanto en carga como en almacenamiento
6. **Breakpoint/Syscall**:
   - Generadas deliberadamente por instrucciones `syscall` y `break`
   - Mecanismo para llamadas al sistema y depuración

6.2. Etapas del Pipeline y Excepciones

El pipeline de 5 etapas en MIPS32 puede generar excepciones en diferentes fases:

1. **IF (Instruction Fetch)**:
   - `TLB Miss` en acceso a memoria de instrucciones
   - `Bus Error` por fallo en lectura
   - `Address Error` por dirección no alineada
2. **ID (Instruction Decode)**:
   - `Reserved Instruction` al decodificar opcode inválido
   - `Coprocessor Unavailable` para instrucciones de CP no habilitado
3. **EX (Execute)**:
   - `Overflow` en operaciones aritméticas
   - `Integer Divide by Zero` en división
   - `Trap` en comparaciones condicionales
4. **MEM (Memory Access)**:
   - `TLB Miss` en acceso a datos
   - `Bus Error` en escritura/lectura
   - `Address Error` por desalineamiento
5. **WB (Write Back)**:
   - Normalmente no genera excepciones
   - Puede detectar `TLB Miss` retardado en algunas implementaciones

6.3. Jerarquía de Manejo

Cuando múltiples excepciones ocurren simultáneamente en diferentes etapas del pipeline, MIPS32 prioriza según:

1. Excepciones de etapas tempranas (IF ¿ID ¿EX ¿MEM)
2. Dentro de la misma etapa, prioridad fija por tipo
3. El bit `EXL` en `Status` evite anidamiento no deseado

6.4. Diferencias entre Interrupciones y Excepciones

Interrupciones:
- **Asíncronas**: Ocurren independientemente del flujo de ejecución del programa
- Generadas por eventos externos al procesador (periféricos, timer, etc.)
- Pueden habilitarse/deshabilitarse mediante el registro `Status`
- Ejemplo: Llegada de datos por UART, fin de temporización

Excepciones:
- **Síncronas**: Relacionadas directamente con la ejecución de instrucciones
- Generadas por condiciones especiales durante la ejecución
- No pueden deshabilitarse (siempre activas)
- Ejemplo: Overflow, acceso inválido a memoria, instrucción no definida

7. Estrategias de Tratamiento de Excepciones e Interrupciones en MIPS32

7.1. Estrategias para Tratar Excepciones en MIPS32

7.2. 1. Manejo mediante Vectores Fijos

- Todas las excepciones saltan a una dirección fija: `0x80000180` (kseg1)
- La rutina principal lee el registro `Cause` para determinar la causa exacta
- Ventaja: Simple y predecible
- Desventaja: Requiere análisis secuencial de causas

Flujo:

1. El hardware guarda automáticamente el PC en `EPC`
2. Salta a la dirección del vector fijo
3. Software consulta `Cause` y `BadVAddr` si aplica
4. Ejecuta la rutina específica
5. Retorna con `eret` (restaura PC desde `EPC`)

7.3. 2. Manejo mediante Tabla de Vectores

- Implementación común en versiones posteriores (MIPS64)
- Diferentes causas saltan a direcciones distintas
- Ventaja: Menor latencia en el manejo
- Desventaja: Mayor complejidad hardware

Flujo:

1. Hardware determina el vector según `ExcCode`
2. Salta a la dirección correspondiente en la tabla
3. No necesita análisis software inicial
4. Retorno igual con `eret`

7.4. Función del Registro EPC

El **Exception Program Counter (EPC)** cumple roles críticos:

- Almacena la dirección de la instrucción que causó la excepción
- En interrupciones, guarda la dirección de la **siguiente** instrucción a ejecutar
- Permite reanudar la ejecución normal después del manejo
- En excepciones recuperables (ej. TLB Miss), permite reintentar la instrucción
- Su valor se restaura automáticamente con `eret`

7.5. Ejemplo de Código

```
.ktext 0x80000180    # Vector fijo
    mfc0 $k0, $13    # Leer Cause
    srl $k0, 2       # Aislar ExcCode
    andi $k0, 0x1f
    # Saltar a rutina según valor
    eret             # Retornar
```

8. Habilitación de Interrupciones en Sistema MIPS32

8.1. Habilitación de Interrupciones por Dispositivo

8.2. Teclado

- Se configura mediante el registro de control del controlador de teclado (ej. PS/2 o USB)
- Habilitación típica:
  1. Escribir '1' en el bit de habilitación de interrupciones del registro de control del teclado
  2. Configurar el vector de interrupción correspondiente en el controlador de interrupciones
  3. Establecer prioridad si el controlador lo soporta
- Ejemplo código:

```
li $t0, 0xFFFF0000   # Dirección base teclado
li $t1, 0x00000002   # Bit 1 = Habilitar interrupción
sw $t1, 0($t0)       # Escribir en registro de control
```

8.3. Pantalla

- Depende del controlador gráfico (ej. framebuffer)
- Pasos típicos:
  1. Habilitar interrupción de refresco vertical (VSync)
  2. Configurar dirección ISR en el controlador
  3. Establecer máscara de interrupción
- Ejemplo:

```
li $t0, 0xFFFF8000   # Dirección controlador video
li $t1, 0x00000001   # Habilitar VSync interrupt
sw $t1, 4($t0)       # Escribir en registro de control
```

8.4. Habilitación en el Procesador

En MIPS32 se configuran los registros del Coprocesador 0:

- **Status Register ($12)**:
  - Bit 0 (IE): Interrupt Enable global (1=habilitado)
  - Bits 15-8 (IM): Máscara de interrupciones hardware (1=habilitar)
  - Bit 1 (EXL): Debe ser 0 para permitir interrupciones
- **Cause Register ($13)**:
  - Bits 15-8 (IP): Muestra interrupciones pendientes
- **Ejemplo de habilitación**:

```
mfc0 $t0, $12        # Leer Status
ori $t0, 0x00000101   # IE=1, IM[0]=1 (habilitar int. 0)
mtc0 $t0, $12        # Escribir Status
```

8.5. Consecuencias de Habilitar Solo en Dispositivos

Si los dispositivos generan interrupciones pero el procesador no está configurado para atenderlas:

- **Señales perdidas**: Las interrupciones quedan pendientes en el registro `IP` (Cause)
- **Comportamiento indefinido**: Algunos controladores pueden saturar el bus
- **Contaminación de estado**: Los bits pendientes pueden afectar futuras interrupciones
- **Posible deadlock**: Dispositivos que esperan atención pueden bloquearse

Solución: Siempre seguir la secuencia:

1. Configurar manejadores (ISR)
2. Habilitar interrupciones en dispositivos
3. Finalmente habilitar globalmente en el procesador

9. Procesamiento de Interrupciones en MIPS32

9.1. Proceso de Atención a Interrupción de Reloj

9.2. Paso a Paso

1. **Generación de la interrupción**:
   - El temporizador hardware alcanza el valor programado
   - El dispositivo de reloj activa la línea IRQ correspondiente
2. **Reconocimiento por el procesador**:
   - El CPU completa la instrucción actual en ejecución
   - Establece el bit `IP[7]` en el registro `Cause`
   - Verifica que `IE=1` y `EXL=0` en `Status`
3. **Acciones del hardware**:
   - Guarda automáticamente el PC en `EPC`
   - Establece `EXL=1` (nivel excepción)
   - Salta a la dirección del vector de interrupciones (`0x80000180` en MIPS32)
4. **Rutina de servicio (software)**:
   - Guarda manualmente los registros que usará:

     ```
     sw $at, 0($sp)
     sw $v0, 4($sp)
     sw $a0, 8($sp)
     # ... etc
     ```

   - Lee el registro `Cause` para confirmar la fuente
   - Atiende el dispositivo (reinicia el temporizador)
   - Restaura los registros guardados
   - Ejecuta `eret` (instrucción de retorno de excepción)
5. **Retorno (hardware)**:
   - Restaura `PC` desde `EPC`
   - Limpia `EXL` (`EXL=0`)
   - Reanuda ejecución del programa interrumpido

9.3. Registros Involucrados

- **Guardados automáticamente**:
  - `EPC`: Contador de programa
  - `SR`: Registro de estado (solo `EXL` se modifica)
- **Guardados manualmente**:
  - Registros de propósito general (`$at, $v0, $a0, etc.`)
  - Registros del coprocesador si se usan
  - Registro `$ra` si hay llamadas anidadas
- **Restaurados automáticamente**:
  - `PC` desde `EPC` (vía `eret`)
  - `SR.EXL` se limpia

9.4. Importancia de Guardar el Contexto

El salvado de contexto es crítico porque:

- **Transparencia**: El programa interrumpido no debe percibir cambios en sus registros
- **Consistencia**: Previene corrupción de datos entre rutinas
- **Estado preciso**: Permite reanudación exacta de la ejecución
- **Anidamiento**: Habilita el manejo de interrupciones durante otras interrupciones

Consecuencias de no guardar contexto:

- Corrupción de registros usados por el programa principal
- Comportamiento errático del sistema
- Pérdida de datos críticos
- Fallos difíciles de depurar

9.5. Ejemplo Completo

```
.ktext 0x80000180
    # 1. Guardar contexto
    sw $at, 0($sp)
    sw $v0, 4($sp)
    # ... (todos los registros a usar)

    # 2. Atender interrupción
    mfc0 $k0, $13       # Leer Cause
    andi $k0, 0x8000    # Verificar bit 15 (Timer)
    beqz $k0, not_timer

    # 3. Reiniciar timer
    li $t0, 0xFFFF0010
    lw $t1, 0($t0)      # Leer contador
    addi $t1, 1000      # Nuevo valor
    sw $t1, 0($t0)      # Escribir

not_timer:
    # 4. Restaurar contexto
    lw $at, 0($sp)
    lw $v0, 4($sp)
    # ...

    # 5. Retornar
    eret
```

9.6. Interrupciones de Reloj y Control de Ejecución en MIPS32

10. Uso de Interrupciones de Reloj

10.1. Prevención de Bucles Infinitos

El sistema operativo puede configurar el temporizador hardware para generar interrupciones periódicas:

- **Configuración inicial**:

```
# Configurar intervalo del timer (ej. 10ms)
li $t0, 0xFFFF0010   # Dirección del timer
li $t1, 10000        # Valor inicial
sw $t1, 0($t0)       # Escribir valor
```

- **Mecanismo de protección**:

  1. El SO mantiene un contador por proceso
  2. En cada interrupción de reloj:

     ```
     subi $s0, $s0, 1     # Decrementar contador
     blez $s0, kill_proc  # Si <=0, terminar proceso
     ```

  3. Si el contador llega a cero, se fuerza la terminación

10.2. Limitación de Tiempo de Ejecución

Para programas con tiempo máximo definido:

- **Implementación típica**:

  1. Asignar quantum de tiempo al cargar el proceso
  2. Programar el timer con el quantum
  3. En la ISR:

     ```
     # Verificar si es el proceso actual
     lw $t2, current_pid
     bne $t2, $s1, skip
     # Forzar cambio de contexto
     jal scheduler
     skip:
     ```

10.3. Manejo de Finalización Anticipada

Cuando un programa termina antes del intervalo del timer:

1. **Liberación de recursos**:
   - El SO debe desprogramar el temporizador:

     ```
     sw $zero, 0($t0)    # Detener timer
     ```

   - Liberar el contador asociado al proceso
2. **Optimizaciones**:
   - Reutilizar el quantum restante para otros procesos
   - Actualizar estadísticas de uso de CPU
   - Notificar al planificador para inmediata reasignación
3. **Caso especial - Syscall exit**:

```
# En handler de syscall:
li $v0, 10         # Código de exit
# ... limpieza ...
sw $zero, 0($t0)   # Desactivar timer
jr $ra             # Retornar al scheduler
```

10.4. Consideraciones de Implementación

- **Sincronización**: El SO debe proteger estructuras de datos compartidas con el ISR
- **Overhead**: Balancear frecuencia de interrupciones vs. responsividad
- **Prioridades**: Asignar adecuadamente la prioridad del timer
- **Recuperación**: Manejar adecuadamente el caso donde el ISR es interrumpido