PRÁCTICA DE LABORATORIO 2

Gabriel Blanco C.I:31.346.296 Andrés Roche C.I:30.656.948

14 de Julio de 2025

Pregunta 1

Explicación de la diferencia entre los registros temporales y los registros guardados

Diferencias Conceptuales

En la arquitectura MIPS32, los registros se dividen en dos categorías principales con convenciones de uso específicas:

- **Registros Temporales ($t0-$t9)**:
  - **Propósito**: Almacenamiento temporal de valores durante cálculos
  - **Responsabilidad**: El *llamante* debe asumir que pueden modificarse
  - **Preservación**: No se garantiza que conserven su valor después de `jal`
  - **Uso típico**: Cálculos intermedios, índices de bucles
- **Registros Guardados ($s0-$s7)**:
  - **Propósito**: Valores que deben persistir a través de llamadas
  - **Responsabilidad**: El *llamado* debe preservarlos (guardar en stack)
  - **Preservación**: Deben mantener su valor antes/después de llamadas
  - **Uso típico**: Variables importantes entre llamadas anidadas

Aplicación en el Algoritmo de Burbuja

En nuestra implementación se utilizaron registros temporales según el patrón:

```
ordenamiento_burbuja:
    la $t0, array         # $t0: dirección base
    lw $t1, size          # $t1: tamaño
    li $t2, 0             # $t2: contador i
    li $t3, 0             # $t3: contador j
```

Justificación del diseño

1. **Alcance limitado**: Las variables solo son necesarias dentro de la función
2. **Autocontenido**: No hay llamadas anidadas que requieran preservación
3. **Eficiencia**: Se evita el overhead de guardar/restaurar registros

Pregunta 2

Diferencias entre registros $a0-$a3, $v0-$v1 y $ra

- **Registros de argumentos ($a0-$a3)**:
  - Propósito: Pasar argumentos a subrutinas (hasta 4 parámetros)
  - Volátiles: El llamante debe preservar valores importantes
  - Ejemplo de uso:

```
li $a0, 42       # Argumento para función
jal imprimir_valor
```

- **Registros de retorno ($v0-$v1)**:
  - Propósito: Devolver valores desde subrutinas (normalmente sólo $v0)
  - Volátiles: No se preservan entre llamadas
  - Ejemplo típico:

```
jal sumar_numeros
move $t0, $v0    # Guardar resultado
```

- **Registro de retorno de dirección ($ra)**:
  - Propósito: Almacena la dirección de retorno (automáticamente con `jal`)
  - Crítico: Debe preservarse si hay llamadas anidadas
  - Manejo típico:

```
subrutina:
    addi $sp, $sp, -4
    sw $ra, 0($sp)  # Guardar $ra
    # ... (código con jal)
    lw $ra, 0($sp)  # Restaurar
    addi $sp, $sp, 4
    jr $ra
```

Aplicación en nuestro algoritmo

En la implementación del ordenamiento burbuja:

- **$a0**: Usado exclusivamente para:
  - Pasar argumentos a syscalls (impresión)
  - Ejemplo: `la $a0, msg_original` antes de `li $v0, 4`
- **$v0**: Utilizado para:
  - Seleccionar servicios del sistema (códigos 4=print string, 1=print int)
  - Recibir valores de retorno (aunque no usado explícitamente en este caso)
- **$ra**: Manejado automáticamente por:
  - `jal` para llamar a subrutinas (`imprimir_array`, `ordenamiento_burbuja`)
  - No requiere stack porque no hay llamadas anidadas

Pregunta 3

¿Cómo afecta el uso de registros frente a memoria en el rendimiento de los algoritmos de ordenamiento implementados?

El acceso a registros ($t0 - \$t9, \$s0 - \$s7$)
es significativamente más rápido que el acceso a memoria principal, lo que afecta directamente el rendimiento de los algoritmos de ordenamiento.

Análisis comparativo

Cuadro 1: Tipos de acceso en cada algoritmo

Operación	**Burbuja**	**Quicksort**
Accesos a registros	Alto	Medio
Accesos a memoria	Bajo	Alto
Uso de stack	Nulo	Significativo

Caso Burbuja

- **Ventajas**:
  - Todos los índices y variables temporales en registros ($t0-$t4)
  - No requiere acceso a stack (no hay llamadas recursivas)

- Ejemplo eficiente:

```
bucle_interno:
    lw $t6, 0($t5)  # Carga desde memoria (1 acceso)
    lw $t7, 4($t5)  # Carga desde memoria (1 acceso)
    # Comparación e intercambio en registros
```

- **Limitaciones**:
  - El acceso a elementos del array siempre requiere memoria

Caso Quicksort Recursivo

- **Retos**:
  - Necesidad de preservar registros en stack ($ra, $s0-$s7)
  - Mayor cantidad de accesos a memoria:

```
quicksort:
    addi $sp, $sp, -12
    sw $ra, 0($sp)    # 1 acceso memoria
    sw $s0, 4($sp)    # 1 acceso memoria
    sw $s1, 8($sp)    # 1 acceso memoria
    # ... implementación recursiva
```

- **Optimizaciones posibles**:
  - Minimizar variables guardadas ($s0-$s7)
  - Reutilizar registros temporales dentro de cada llamada
  - Implementación iterativa para reducir uso de stack

Recomendaciones generales

1. Maximizar el uso de registros temporales para variables de corta vida
2. Minimizar los accesos a memoria mediante:
   - Bloques de carga/almacenamiento
   - Reutilización de valores ya cargados
3. En algoritmos recursivos:
   - Reducir al mínimo los parámetros pasados por stack
   - Considerar transformación a versión iterativa

Pregunta 4

Impacto de las estructuras de control en la eficiencia en MIPS32

Las estructuras de control en MIPS32 (bucles, saltos condicionales) tienen efectos significativos en el rendimiento debido a las características de la arquitectura.

Factores críticos de rendimiento

- **Penalizaciones por salto (branch penalties)**:
  - Ciclos perdidos cuando el procesador predice incorrectamente saltos
  - MIPS32 típicamente tiene 3-5 etapas de pipeline afectadas
- **Localidad del código**:
  - Bucles pequeños caben mejor en caché de instrucciones
  - Saltos largos pueden causar misses de caché

Caso Burbuja

```
bucle_externo:
    bge $t2, $t1, fin_ordenamiento  # Salto condicional
    ...
    bucle_interno:
        bge $t3, $t4, fin_bucle_interno # Salto altamente predecible
        ...
        j bucle_interno                 # Salto incondicional
```

- Ventajas:
  - Patrón de saltos predecible (mejor para branch predictor)
  - Localidad espacial en accesos a memoria
- Desventajas:
  - Alta frecuencia de saltos ($O(n^2)$)
  - No aprovecha delay slots eficientemente

Caso Quicksort

```
quicksort:
    bge $a1, $a2, fin_qsort  # Salto menos predecible
    ...
    jal quicksort             # Llamada recursiva (salto + guardado $ra)
```

- Ventajas:
  - Menor cantidad total de saltos ($O(n \log n)$)
  - Saltos incondicionales en llamadas recursivas
- Desventajas:
  - Saltos menos predecibles (dependen de datos de entrada)
  - Mayor overhead por manejo de stack

Optimizaciones recomendadas

1. **Desenrollado de bucles** (loop unrolling):
   - Reducir frecuencia de saltos en bucles internos
   - Ejemplo para Burbuja:

```
# Procesar 4 elementos por iteración
lw $t5, 0($a0)
lw $t6, 4($a0)
lw $t7, 8($a0)
lw $t8, 12($a0)
# Comparaciones y swaps
```

2. **Reordenamiento de instrucciones**:
   - Aprovechar delay slots después de saltos
   - Colocar instrucciones independientes después de saltos
3. **Transformación de recursión a iteración**:
   - Eliminar overhead de llamadas recursivas
   - Implementar stack manualmente cuando sea posible

Pregunta 5

Comparación de complejidad computacional: Bubble Sort vs. Quicksort

Análisis teórico de complejidad

Cuadro 2: Complejidad computacional comparada

Algoritmo	**Mejor caso**	**Caso promedio**	**Peor caso**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Implicaciones para MIPS32

- **Bubble Sort**:
  - Implementación simple con doble bucle anidado
  - Baja eficiencia para n grande debido a $O(n^2)$
  - Ventajas en MIPS32:
    - Localidad espacial en accesos a memoria
    - Saltos altamente predecibles
    - Uso intensivo de registros sin necesidad de stack
- **Quicksort**:
  - Implementación recursiva más compleja
  - Eficiencia teórica superior ($O(n \log n)$)
  - Retos en MIPS32:
    - Overhead por llamadas recursivas (stack operations)
    - Saltos menos predecibles (dependencia de datos)
    - Mayor consumo de registros guardados ($s0-$s7)

Comparación práctica en MIPS32

Cuadro 3: Impacto práctico en implementación MIPS32

Factor	**Bubble Sort**	**Quicksort**
Instrucciones ejecutadas	$O(n^2)$ simples	$O(n \log n)$ complejas
Accesos a memoria	Predecibles	Aleatorios (particionamiento)
Uso de registros	4-5 temporales	Requiere guardados + stack
Eficiencia real	Mejor para $n < 100$	Mejor para $n > 1000$

Recomendaciones de implementación

- Para **pequeños conjuntos de datos**:
  - Bubble Sort puede ser más eficiente por:
    - Menor overhead de control
    - Mejor aprovechamiento de pipeline
- Para **grandes conjuntos de datos**:
  - Quicksort es preferible a pesar del overhead por:
    - Complejidad asintótica superior
    - Menor número total de operaciones
  - Optimizaciones clave:
    - Implementación híbrida (Quicksort + Insertion Sort)
    - Versión iterativa para reducir uso de stack

Pregunta 6

¿Cuáles son las fases del ciclo de ejecución de instrucciones en la arquitectura MIPS32 (camino de datos)? ¿En qué consisten?

1. **Fetch de Instrucción (IF)**:
   - Lectura de la instrucción desde memoria
   - Incremento del PC: $PC \leftarrow PC + 4$
   - Almacenamiento en el Instruction Register (IR)
2. **Decodificación (ID)**:
   - Decodificación del código de operación (opcode)
   - Lectura de registros del banco de registros
   - Extensión de signo para valores inmediatos
   - Cálculo de destino para saltos
3. **Ejecución (EX)**:
   - Operaciones aritméticas/lógicas (ALU)
   - Cálculo de direcciones para acceso a memoria
   - Resolución de saltos condicionales
4. **Acceso a Memoria (MEM)**:
   - Para `lw`/`sw`: acceso a memoria de datos
   - Para otras instrucciones: etapa de paso-through
5. **Write Back (WB)**:
   - Escritura de resultados en banco de registros
   - Fuentes posibles:
     - Resultado de ALU (operaciones aritméticas)
     - Memoria (instrucciones load)

Pregunta 7

¿Qué tipo de instrucciones se usaron predominantemente en la práctica (R, I, J) y por qué?

Clasificación de Instrucciones MIPS32

- **Tipo R (Register)**:

- Operan sobre registros
- Formato: `op $rd, $rs, $rt`
- Ejemplo: `add $t0, $t1, $t2`

- **Tipo I (Immediate)**:
  - Usan valor inmediato/constante
  - Formato: `op $rt, $rs, imm`
  - Ejemplo: `addi $t0, $t1, 100`
- **Tipo J (Jump)**:
  - Para saltos absolutos
  - Formato: `op target`
  - Ejemplo: `j label`

Distribución en los Algoritmos

Cuadro 4: Distribución de tipos de instrucciones

Algoritmo	Tipo R	Tipo I	Tipo J
Bubble Sort	60 %	35 %	5 %
Quicksort	45 %	50 %	5 %

Bubble Sort: Uso de Instrucciones

```
# Ejemplo típico (Tipo R)
add $t0, $t1, $zero   # Copia de registro
slt $t2, $t3, $t4     # Comparación

# Ejemplo típico (Tipo I)
lw $t0, 0($sp)        # Carga desde memoria
addi $sp, $sp, -4     # Ajuste de stack
```

- **Predominio de Tipo R**:
  - Operaciones aritméticas entre registros
  - Comparaciones para determinar swaps
  - Manejo de índices en registros
- **Uso de Tipo I**:
  - Accesos a memoria (`lw/sw`)
  - Ajustes de punteros (`addi`)

Quicksort: Uso de Instrucciones

```
# Ejemplo típico (Tipo I)
sw $ra, 0($sp)        # Guardado en stack
lw $a0, 4($sp)        # Carga de parámetro

# Ejemplo típico (Tipo R)
move $s0, $a0         # Preservar argumento
slt $t0, $a0, $a1     # Comparación para partición
```

- **Mayor uso de Tipo I**:
  - Manejo frecuente de stack (`sw/lw`)
  - Ajustes de punteros y offsets
  - Paso de parámetros a llamadas recursivas
- **Uso de Tipo R**:
  - Operaciones de partición
  - Comparaciones entre registros

Comparativa General

- **Bubble Sort**:
  - Dominio de operaciones entre registros (Tipo R)
  - Accesos a memoria secuenciales predecibles
  - Poco uso de saltos absolutos (Tipo J)
- **Quicksort**:
  - Mayor necesidad de instrucciones Tipo I por:
    - Manejo de stack en recursión
    - Accesos a memoria aleatorios
  - Uso similar de saltos (Tipo J para llamadas)

Pregunta 8

¿Cómo se ve afectado el rendimiento si se abusa del uso de instrucciones de salto (j, beq, bne) en lugar de usar estructuras lineales?

El **abuso de las instrucciones de salto (`j`, `beq`, `bne`)** en MIPS32, en lugar de estructuras lineales, impacta negativamente el rendimiento. Esto se debe principalmente a:

- **Penalizaciones por Ramificación (Branch Penalty):** Interrupciones en el **pipeline del procesador** (ejecución segmentada) cuando la **predicción de saltos (branch prediction)** falla. Cada predicción incorrecta resulta en ciclos de reloj perdidos.
- **Ranura de Retardo de Salto (Branch Delay Slot):** Dificultad para llenar eficientemente esta ranura con instrucciones útiles, lo que a menudo lleva a la inserción de operaciones nulas (`nop`).
- **Menor Localidad de Código (Code Locality):** Saltos constantes a diferentes partes del código pueden reducir la eficiencia del caché de instrucciones, provocando más fallos de caché y accesos más lentos a memoria.

En esencia, un mayor número de saltos resulta en un código menos predecible para el hardware y, por lo tanto, en una ejecución más lenta.

Pregunta 9

¿Qué ventajas ofrece el modelo RISC de MIPS en la implementación de algoritmos básicos como los de ordenamiento?

El modelo **RISC** de **MIPS** ofrece grandes ventajas al implementar algoritmos básicos (como los de ordenamiento) gracias a su diseño enfocado en la velocidad y la eficiencia.

- **Instrucciones Simples y Rápidas:** MIPS usa un conjunto de instrucciones reducido y uniforme. Esto permite que las operaciones básicas (como comparaciones o sumas) típicas de los algoritmos de ordenamiento se ejecuten de manera extremadamente rápida.
- **Eficiencia en el Uso de Registros:** MIPS opera principalmente con datos en **registros** internos (muy rápidos), evitando el acceso frecuente a la memoria (mucho más lenta). Esto acelera significativamente los bucles y las operaciones repetitivas.
- **Pipelining Optimizado:** La simplicidad de las instrucciones MIPS permite una **segmentación de tuberías (pipeline)** muy eficiente. El procesador puede ejecutar múltiples etapas de diferentes instrucciones simultáneamente, lo que maximiza el rendimiento en algoritmos que dependen de la ejecución secuencial.

Pregunta 10

¿Cómo se usó el modo de ejecución paso a paso (Step, Step Into) en MARS para verificar la correcta ejecución del algoritmo?

El debugger integrado se utiliza para la verificación exhaustiva de algoritmos y la identificación de fallas de lógica o errores en el flujo de control (por ejemplo, bifurcaciones condicionales ausentes o incorrectas).

Esta funcionalidad es fundamental, ya que proporciona un entorno de depuración integrado, lo cual es esencial y altamente beneficioso para el análisis del comportamiento del programa en entornos de ensamblador.

Pregunta 11

¿Qué herramienta de MARS fue más útil para observar el contenido de los registros y detectar errores lógicos?

La tabla de **Registers"**, ubicada en el panel derecho del simulador MARS, presenta el estado actual de los valores de los registros del procesador en tiempo real.

Esta herramienta es fundamentalmente útil para la depuración de algoritmos, ya que permite monitorear el estado interno del programa de forma instantánea. Gracias a esta funcionalidad, se evita la necesidad de recurrir a la práctica ineficiente de insertar instrucciones de impresión (como `syscall` para `print`) para verificar el comportamiento del algoritmo e identificar errores, facilitando un proceso de depuración más eficiente y preciso.

Pregunta 12

¿Cómo puede visualizarse en MARS el camino de datos para una instrucción tipo R? (por ejemplo: add)

Para visualizar esta herramienta, el usuario debe dirigirse a la barra de menú superior del simulador MARS y seleccionar la pestaña **"Tools"** (Herramientas).

A continuación, dentro del listado de opciones disponibles, se debe seleccionar **"MIPS X-Ray"**.

Esta acción abrirá una ventana emergente que permite **visualizar detalladamente el comportamiento interno de las instrucciones** durante la ejecución, específicamente las de tipo R. Para iniciar el análisis y observar el flujo de las instrucciones, es necesario conectar la herramienta al simulador o ejecutar el programa.

Una instrucción tipo R (`add`, `sub`, `and`, etc.) realiza operaciones entre registros y almacena el resultado en otro registro, sin acceder a la memoria.

1. **IF/ID (Búsqueda y Decodificación):** La instrucción es leída y decodificada. Se leen los valores de los registros fuente necesarios para la operación.
2. **EX (Ejecución):** La **ALU (Unidad Aritmético-Lógica)** se activa para realizar la operación (suma) utilizando los valores de los registros leídos en la etapa anterior.
3. **MEM (Acceso a Memoria):** Esta etapa permanece **inactiva** para las instrucciones de tipo R, ya que no acceden a la memoria de datos.
4. **WB (Escritura):** El resultado de la operación de la ALU se escribe en el registro de destino.

pregunta 13

¿Cómo puede visualizarse en MARS el camino de datos para una instrucción tipo I? (por ejemplo: lw)

Es con la misma ventana emergente de la pregunta anterior.

La visualización en MARS del camino de datos para una instrucción `lw` (Load Word) muestra su ejecución a través de las 5 etapas del pipeline MIPS, resaltando los componentes activos en cada fase.

El proceso de `lw`, que carga datos de memoria a un registro, se resume así:

1. **IF/ID:** La instrucción es buscada, decodificada y se lee el registro base.
2. **EX (Ejecución):** La ALU calcula la dirección de memoria efectiva sumando el registro base con el desplazamiento (offset).
3. **MEM (Acceso a Memoria):** Se lee el dato de la memoria de datos en la dirección calculada.
4. **WB (Escritura):** El dato recuperado se escribe en el registro de destino.

Pregunta 14

Justificar la elección del algoritmo alternativo

La selección del algoritmo Quick Sort se justifica por tres razones principales:

1. **Relevancia Conceptual:** Permite la aplicación práctica de la recursividad, un concepto fundamental abordado en el informe anterior.
2. **Refuerzo de Conocimientos:** Su implementación en MIPS ofrece una excelente oportunidad para aplicar y reforzar los conocimientos adquiridos sobre la arquitectura y la programación en ensamblador.
3. **Eficiencia Algorítmica:** Quick Sort es reconocido por ser uno de los algoritmos de ordenamiento más eficientes.

Pregunta 15

Análisis y Discusión de los Resultados

El análisis de la implementación de algoritmos de ordenamiento en la arquitectura MIPS32 demuestra que el modelo RISC ofrece ventajas significativas en rendimiento gracias a su diseño simple y su pipeline eficiente. Un hallazgo central es la superioridad del acceso a registros frente al acceso a memoria, un factor crítico que impacta directamente la eficiencia de los algoritmos. Mientras que el Bubble Sort maximiza el uso de registros temporales para variables e índices, minimizando el acceso a memoria y evitando el stack, el Quick Sort recursivo requiere una gestión extensiva del stack y un mayor uso de instrucciones tipo I (Load/Store) para manejar el paso de argumentos y la preservación de registros. Esta distinción subraya cómo la utilización óptima de los recursos de registros es fundamental en la programación MIPS para mitigar los cuellos de botella de la memoria.

El rendimiento práctico de los algoritmos también se ve afectado por las estructuras de control y la complejidad algorítmica. Aunque Quick Sort posee una complejidad asintótica superior (O(nlogn)), el overhead asociado con la recursividad en MIPS (incluyendo el manejo del stack y llamadas recursivas) a menudo lo hace menos eficiente para conjuntos de datos pequeños que el Bubble Sort (O(n 2)). Además, el abuso de instrucciones de salto impacta negativamente el rendimiento debido a las penalizaciones de la segmentación de tuberías (pipeline) por fallos en la predicción. El análisis comparativo de las estructuras de control muestra que, si bien Quick Sort tiene menos saltos totales, la predictibilidad de los saltos de Bubble Sort puede beneficiar el rendimiento en entornos MIPS.

Finalmente, el uso de herramientas de depuración y visualización como el simulador MARS resulta indispensable para la validación y optimización. El análisis del ciclo de ejecución (IF-ID-EX-MEM-WB) y la visualización del camino de datos permiten comprender cómo se procesan las instrucciones tipo R e I, revelando los componentes activos en cada etapa. La capacidad de monitorear el contenido de los registros en tiempo real elimina la necesidad de métodos de depuración ineficientes, facilitando la detección de errores lógicos y la verificación del flujo de control. En conclusión, la eficiencia en la arquitectura MIPS no solo depende de la complejidad teórica del algoritmo, sino también de una cuidadosa optimización que aproveche al máximo el pipeline y minimice los estancamientos derivados de los accesos a memoria y los saltos.