Análisis de Implementación de Algoritmos de Ordenamiento en MIPS32

Arquitectura de Computadores

23 de febrero de 2026

Resumen

Este informe analiza la implementación práctica de algoritmos de ordenamiento (Bubble Sort, Merge Sort y Quicksort) en ensamblador MIPS32. Se examinan aspectos fundamentales como el uso de registros, estructuras de control, complejidad computacional, y herramientas de depuración en MARS. El objetivo es comprender las particularidades de la programación en ensamblador RISC y su impacto en el rendimiento de algoritmos clásicos.

1. Registros Temporales vs Registros Guardados

1.1. Diferencias Conceptuales

Característica	**Registros Temporales ($t0-$t9)**	**Registros Guardados ($s0-$s7)**
Convención	No preservados entre llamadas	Preservados entre llamadas
Persistencia	Pueden ser sobrescritos por funciones llamadas	Deben ser restaurados al retornar
Uso típico	Valores temporales, cálculos intermedios	Variables que deben mantenerse
Responsabilidad	Caller debe guardarlos si los necesita	Callee debe guardarlos si los usa

Cuadro 1: Diferencias entre registros temporales y guardados

1.2. Aplicación Práctica en Bubble Sort

Listing 1: Uso de registros en Bubble Sort

```
bubble_sort:
    # Uso de registros guardados (deben preservarse)
    move $s0, $a0               # $s0 = direccion base (debe
        ↪ mantenerse)
    move $s1, $a1               # $s1 = n (debe mantenerse)

    # Uso de registros temporales (pueden sobrescribirse)
    li $t0, 0                    # $t0 = i (temporal)
    lw $t1, 0($s0)              # $t1 = array[i] (temporal)
    addi $t2, $t1, 1            # $t2 = calculo temporal
```

En la práctica: - **$s0-$s7**: Se usaron para la dirección base del array y el tamaño, valores que deben persistir - **$t0-$t9**: Se usaron para índices de bucle, comparaciones y valores temporales - Al llamar funciones, se guardaron $s0-$s7 en pila pero no $t0-$t9

2. Registros de Propósito Específico

2.1. Funciones de $a0-$a3, $v0-$v1 y $ra

Registro	Propósito	Aplicación práctica
$a0-$a3	Argumentos para funciones	Dirección del array, low, high
$v0-$v1	Valores de retorno	Índice del pivote, resultados
$ra	Dirección de retorno	Guardado en pila para recursión

Cuadro 2: Registros de propósito específico

Listing 2: Ejemplo de uso en Quicksort

```
quicksort:
    # $a0 = direccion array
    # $a1 = low
    # $a2 = high

    jal partition               # Llamada a funcion
    move $s3, $v0               # Guardar resultado ($v0)

    # Llamada recursiva
    jal quicksort               # $ra se sobrescribe
        ↪ automaticamente
```

3. Registros vs Memoria: Impacto en Rendimiento

3.1. Comparativa de Accesos

Operación	Ciclos	Energía	Uso en algoritmo
Acceso a registro	1 ciclo	Baja	Constante
Acceso a memoria (lw/sw)	2-100+ ciclos	Alta	Para array y pila

Cuadro 3: Comparativa de costos de acceso

3.2. Impacto en Algoritmos

Para Bubble Sort con n=1000:

- **Usando registros**: Todas las comparaciones entre registros ≈ 500,000 ciclos
- **Usando memoria**: Cada comparación requeriría accesos a memoria ≈ 50,000,000 ciclos
- **Diferencia**: 100x más lento usando memoria

4. Estructuras de Control y Eficiencia

4.1. Impacto de Bucles Anidados

Listing 3: Bucles anidados en Bubble Sort

```
bucle_externo:
    # 1: Control de bucle externo
    blez $s2, fin_bubble

bucle_interno:
    # 2: Control de bucle interno
    bge $s3, $s2, siguiente_pasada

    # 3: Operaciones de comparacion
    lw $t1, 0($t0)            # Acceso a memoria
    lw $t3, 0($t2)            # Acceso a memoria
    ble $t1, $t3, no_swap      # Salto condicional

    # 4: Intercambio (2 stores)
```

```
15      sw $t3, 0($t0)
16      sw $t1, 0($t2)
```

4.2. Costo de Saltos

Cada instrucción de salto (beq, bne, j) causa:

- Penalización por salto: 1-3 ciclos (pipelines)
- Posible vaciado de pipeline si se predice incorrectamente
- En algoritmos de ordenamiento: $O(n^2)$ saltos condicionales

5. Análisis de Complejidad: Quicksort vs Bubble Sort

5.1. Comparativa Teórica

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

Cuadro 4: Comparativa de complejidades

5.2. Implicaciones en MIPS32

Para n = 1000:

- **Bubble Sort optimizado**: ≈ 500,000 comparaciones
- **Quicksort**: ≈ 10,000 comparaciones
- **Diferencia**: 50x más rápido Quicksort

5.3. Impacto de la Recursión

Quicksort requiere:

- Aproximadamente log(n) niveles de recursión
- 8-16 bytes por nivel en pila
- Para n=1000: ≈ 10 niveles, 160 bytes de pila

6. Ciclo de Ejecución en MIPS32

6.1. Las 5 Fases del Pipeline

IF	ID	EX	MEM	WB
Fetch	Decode	Execute	Memory	Write Back

Figura 1: Fases del pipeline MIPS32

6.2. Descripción de cada fase

1. **IF (Instruction Fetch)**: Obtiene instrucción de memoria
2. **ID (Instruction Decode)**: Decodifica y lee registros
3. **EX (Execute)**: Ejecuta operación (ALU)
4. **MEM (Memory Access)**: Accede a memoria (lw/sw)
5. **WB (Write Back)**: Escribe resultado en registro

6.3. Ejemplo con instrucción de ordenamiento

```
lw $t1, 0($t0)    # IF->ID->EX->MEM->WB
add $t2, $t1, $s0 # IF->ID->EX->WB
sw $t2, 4($t0)    # IF->ID->EX->MEM->WB
```

7. Tipos de Instrucciones Utilizadas

7.1. Distribución en Bubble Sort

Tipo	Formato	Ejemplos	Porcentaje
Tipo R	op(6) rs(5) rt(5) rd(5) shamt(5) funct(6)	add, sub, sll	30 %
Tipo I	op(6) rs(5) rt(5) immediate(16)	lw, sw, beq, addi	60 %
Tipo J	op(6) address(26)	j, jal	10 %

Cuadro 5: Distribución de tipos de instrucciones

7.2. Justificación

- **Tipo I predominante**: Acceso a array (lw/sw) y comparaciones (beq)

- **Tipo R**: Operaciones aritméticas y de desplazamiento
- **Tipo J**: Llamadas recursivas y saltos largos

8. Impacto de Instrucciones de Salto

8.1. Penalizaciones

- **Saltos condicionales**: 1-3 ciclos de penalización
- **Saltos incondicionales**: 1-2 ciclos
- **Estructuras lineales**: 0 penalización

8.2. Ejemplo comparativo

```
# Version con saltos (menos eficiente)
loop:
    beq $t0, $t1, end
    add $t2, $t2, 1
    j loop

# Version lineal (mas eficiente pero no siempre posible)
    add $t2, $t2, 100
```

9. Ventajas del Modelo RISC en MIPS

9.1. Beneficios para Algoritmos de Ordenamiento

1. **Instrucciones simples**: Cada instrucción hace una operación básica
2. **Formato fijo**: Facilita decodificación y pipeline
3. **32 registros**: Suficientes para variables frecuentes
4. **Modos de direccionamiento simples**: Acceso eficiente a arrays
5. **Pipeline optimizado**: 1 instrucción por ciclo ideal

9.2. Comparativa CISC vs RISC

- **RISC (MIPS)**: 4-5 instrucciones para intercambio
- **CISC (x86)**: 1 instrucción compleja (xchg)
- Pero RISC permite mejor pipeline y frecuencia más alta

10. Depuración Paso a Paso en MARS

10.1. Modos de Ejecución

Comando	**Función**	**Uso**
Step (F7)	Ejecuta 1 instrucción	Verificar instrucción específica
Step Over (F8)	Ejecuta hasta retorno	Evitar entrar en funciones
Run (F5)	Ejecución continua	Probar funcionalidad completa
Reset	Reinicia programa	Reiniciar prueba

Cuadro 6: Modos de ejecución en MARS

10.2. Verificación de Algoritmos

Pasos seguidos:

1. Breakpoints en puntos clave (partición, intercambio)
2. Step into para funciones recursivas
3. Observar cambios en array después de cada pasada
4. Verificar condición de término de bucles

11. Herramientas de Depuración en MARS

11.1. Ventanas más Útiles

1. **Register Window**: Muestra valores de todos los registros
   - Crítico para ver $a0-$a3 en llamadas recursivas
   - Verificar $sp (pila) en cada nivel
2. **Data Segment Window**: Visualiza el array

- Cambios en tiempo real durante ordenamiento
- Verificar que los elementos se ordenan correctamente

3. **Text Segment**: Código con breakpoints
   - Resaltado de instrucción actual
   - Facilita seguimiento de flujo

11.2. Detección de Errores Comunes

- **Registros no inicializados**: Valores basura en ventana de registros
- **Desbordamiento de pila**: $sp fuera de rango
- **Bucles infinitos**: PC no avanza o se repite

12. Visualización del Camino de Datos en MARS

12.1. Para instrucción Tipo R (add)

Listing 4: Ejemplo: add $t1$

```
1 add $t1, $t2, $t3
```

12.2. Componentes involucrados

1. **IF**: PC → Instruction Memory → Instrucción
2. **ID**: Decodifica: op=0, rs=$t2, rt$ =t3, rd=$t1$, $funct = 32$
2. **EX**: ALU recibe valores de registros, funct=32 (add)
3. **MEM**: No hay acceso a memoria (bypass)
4. **WB**: Resultado ALU → Register File ($t1$)

12.3. Visualización en MARS

- **Menú**: Tools → Data Path Viewer
- Muestra el flujo de datos en cada ciclo
  item Útil para entender pipeline y forwarding

13. Conclusiones

13.1. Resumen de Hallazgos

1. La correcta gestión de registros es crucial en MIPS32
2. Bubble Sort es simple pero ineficiente para grandes datasets
3. Quicksort ofrece mejor rendimiento pero requiere manejo cuidadoso de pila
4. Las herramientas de MARS (Step, Register Window) son indispensables para depuración
5. El modelo RISC facilita la comprensión del hardware subyacente

13.2. Recomendaciones

- Usar registros $s para valores persistentes
- Minimizar accesos a memoria manteniendo datos en registros
- Para n ¿100, preferir Quicksort sobre Bubble Sort
- Utilizar breakpoints estratégicos en lugar de paso a paso continuo