Universidad de Carabobo

Facultad de Ciencias y Tecnología
Departamento de Computación
Arquitectura del Computador 2025-II

Integrantes:
Simón Tovar V31.678.578
Gabriel Becerra V31.654.243

Febrero 2026

Informe: Prácticas de Laboratorio 1 y 2

Índice

1. Práctica 1: Paridad

1.1. Preguntas:

1.1.1. ¿Cómo se implementa la recursividad en MIPS32? ¿Qué papel cumple la pila ($sp)?

La recursividad se implementa usualmente mediante instrucciones de salto y enlace (jal). Al ejecutar jal el procesador guarda en $ra la dirección de la siguiente instrucción y luego salta a la etiqueta especificada.

En cada llamada recursiva se debe sobrescribir el $ra, para que la dirección de retorno original no se pierda y el programa pueda retornar al punto de ejecución antes de la llamada recursiva, se utiliza la pila $sp.

Antes de realizar la llamada recursiva se debe crear espacio restando el tamaño en bytes de los datos que se van a guardar en la pila, ya sea el $ra o cualquier otro registro (comúnmente los registros $s0-$s7).

Al llegar al caso base el algoritmo debe comenzar a desapilar los registros guardados y sumar la memoria desocupada de nuevo a $sp.

1.1.2. ¿Qué riesgos de desbordamiento existen? ¿Cómo mitigarlos?

El riesgo de desbordamiento mas critico en algoritmos recursivos es el desbordamiento de pila (Stack Overflow), si el numero n es muy grande, o hay un error en el algoritmo y nunca llega al caso base, la pila puede crecer hasta agotar la memoria, o como la pila crece "hacia abajo"podría invadir el segmento de datos causando un error de ejecución o corrupción de datos.

Para mitigarlos la mejor opción es usar la versión iterativa, ya que esta usa un bucle y registros constantes, manteniendo el puntero de pila estático.

También es importante asegurar que el caso base sea alcanzable. Si se llega a ingresar un numero negativo por ejemplo, se produciría una recursión infinita causando desbordamiento.

1.1.3. ¿Qué diferencias encontraste entre una implementación iterativa y una recursiva en cuanto al uso de memoria y registros?

La diferencia mas notable se encuentra en el uso de memoria, en la versión recursiva cada vez que el algoritmo se llama a si mismo se reservan 8 bytes en la pila, por lo que la memoria consumida crecerá linealmente $(8 \times n)$ bytes, mientras que la versión iterativa no utiliza la pila.

En cuanto a la gestión de registros, en la versión recursiva ademas de guardar en la pila la información del registro $ra, también se guarda el argumento del registro $a0, para poder recuperarlo después del retorno y poder usarlo para el calculo de la función. Mientras que en la versión iterativa el algoritmo se apoya en los registros temporales.

1.1.4. ¿Qué diferencias encontraste entre los ejemplos académicos del libro y un ejercicio completo y operativo en MIPS32?

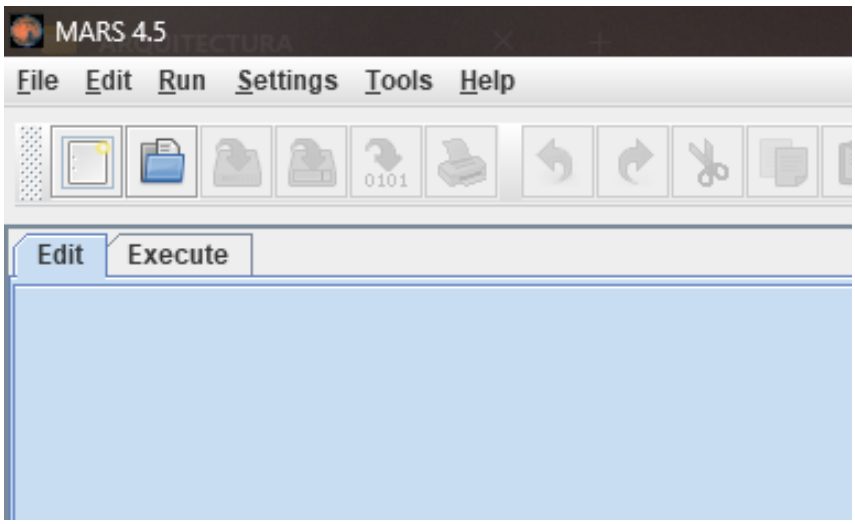
En los ejemplos del libro se prioriza la parte mas teórica del ensamblador MIPS32, muchas veces asumiendo que los datos ya se encuentran en los registros, y muchas veces

los fragmentos de código terminan repentinamente, sin mostrar la estructura completa. Mientras que al implementarlo en el simulador mars de forma operativa, es obligatorio implementar la interfaz con el usuario mediante llamadas al sistema operativo usando syscall, esto para la salida y también para la entrada de datos algunas veces.

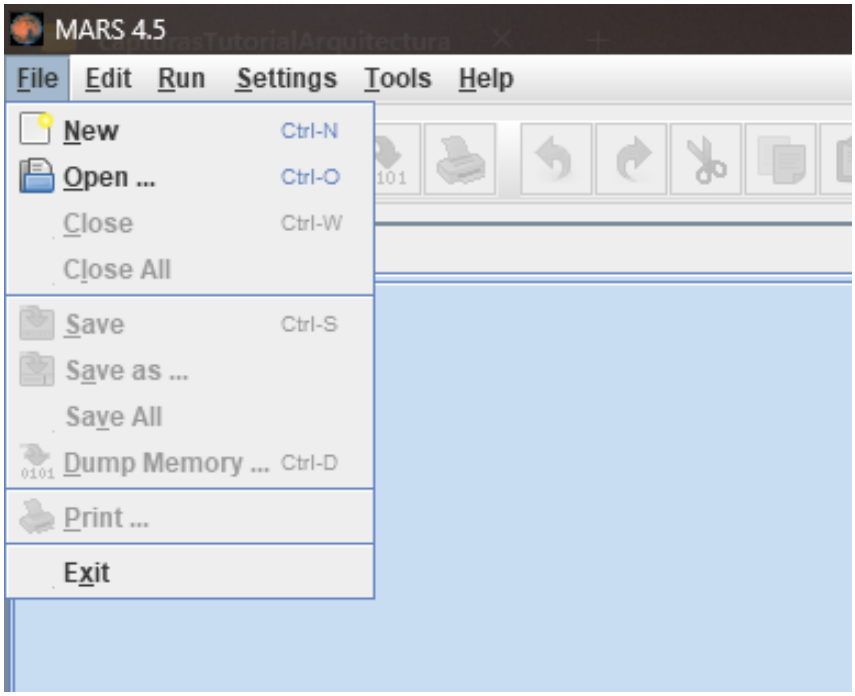
Al implementarlo de forma operativa se deben tener los respectivos campos de datos (.data), donde se pueda simular la memoria para que el algoritmo pueda tomar los datos de ahí y cargarlos en los registros, y al terminar el programa se deben hacer las respectivas llamadas a sistema operativo.

1.1.5. Tutorial de ejecución paso a paso en el simulador mars:

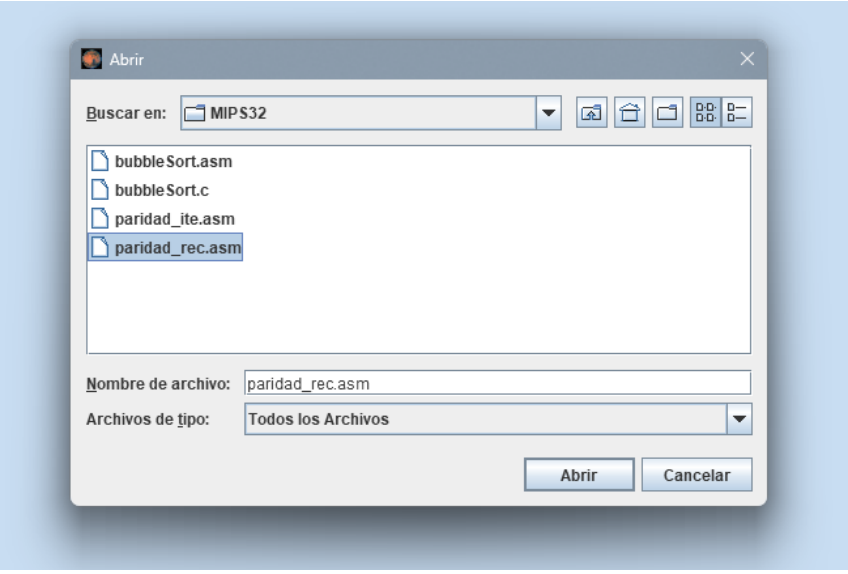
1. **Abrir el archivo .asm:**
   Hacer click al botón file en la esquina superior izquierda del programa.



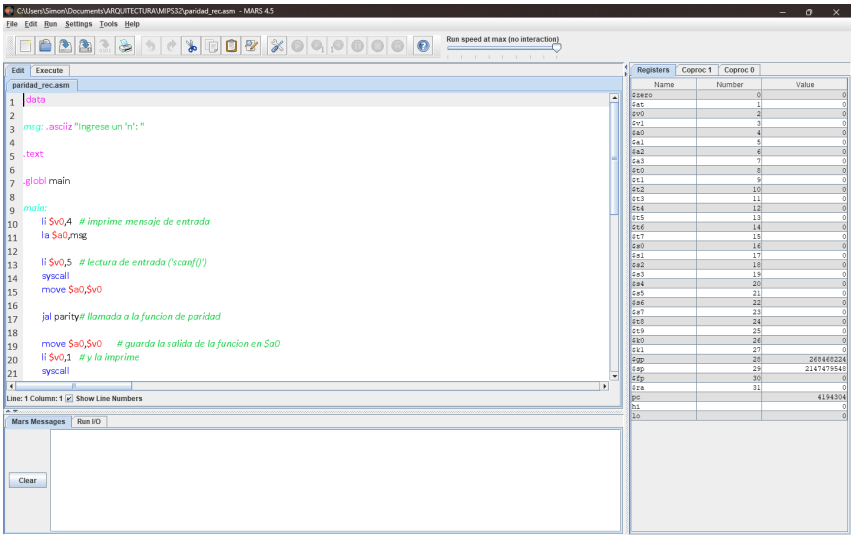
Luego hacer click en el botón open del menú.



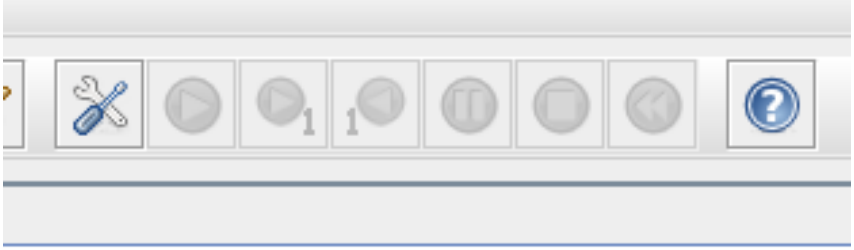
Se abrira una ventana donde debe seleccionar el archivo .asm del algoritmo, al seleccionarlo debe hacer click en abrir.



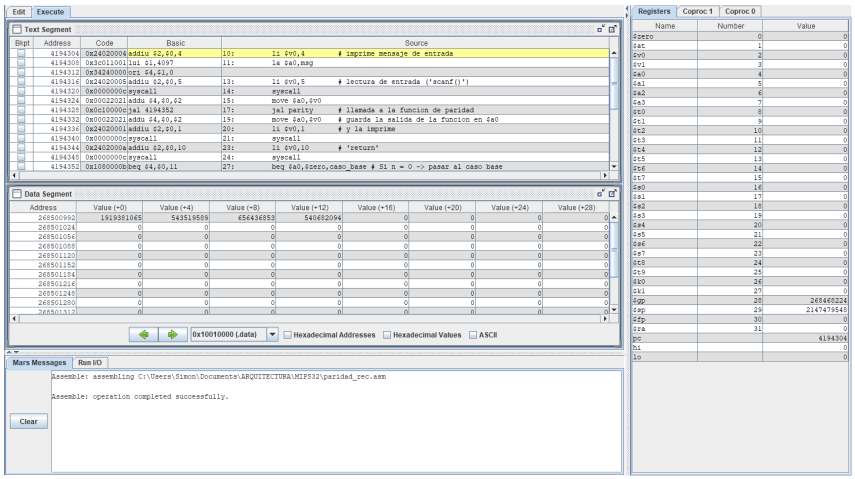
Luego de esto se abrirá el archivo y podrá ver su contenido. En la barra superior hay distintas opciones.



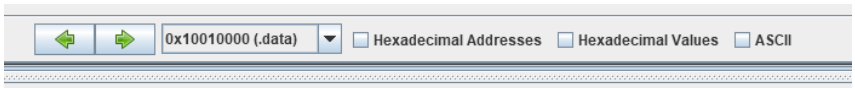
2. **Esamblar y ejecutar el programa:**
   En la barra superior podra encontrar las siguientes opciones, primero seleccione el botón con una llave inglesa y un destornillador, este botón ensamblara el código.



Esto automáticamente lo llevara a la ventana de ejecución, y activara las opciones correspondientes en la barra superior.



En la parte de abajo del cuadro de ejecución podemos encontrar las siguientes opciones:



Se recomienda desmarcarlas, para así poder leer los valores en números decimales y mejorar la legibilidad.

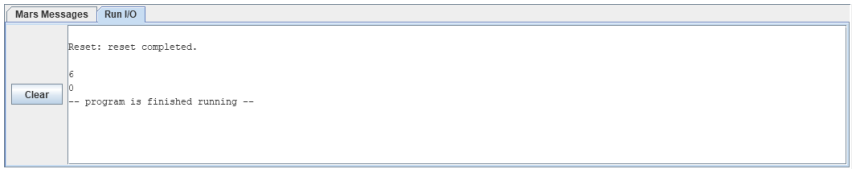
En la barra de la parte superior encontraremos opciones para ejecutar el programa completamente, ejecutar linea por linea, volver atrás una linea y resetear el programa completamente.

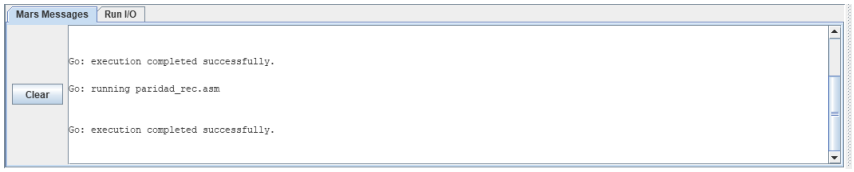
En la parte derecha podremos ver los valores de los registros y como cambian durante la ejecución.

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	268500992
$v0	2	3
$v1	3	0
$a0	4	3
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	4194336
pc		4194352
hi		0
lo		0

En la parte de abajo podemos ver una consola con dos pestañas, en la pestaña Run I/O podemos ver los mensajes de entrada y salida del programa, y en la pestaña de Mars Messages podemos ver los mensajes de ejecución del simulador.





1.1.6. Justificar la elección del enfoque (iterativo o recursivo) según eficiencia y claridad en MIPS.

El enfoque iterativo es significativamente mas eficiente que la versión recursiva, principalmente en consumo de memoria, ya que la versión iterativa es $O(1)$, porque opera solo con los registros del procesador mientras que la que la versión recursiva es $O(n)$ porque en cada llamada reserva 8 bytes de la pila.

En cuanto a claridad, desde un punto de vista matemático la versión recursiva es mas clara, ya que refleja directamente la definición inductiva del problema, sin embargo en lenguaje ensamblador esta claridad se pierde debido a la gestión manual de la pila que debe realizar el programador.

1.1.7. Análisis y Discusión de los Resultados

Tras implementar las variantes recursivas y iterativas del algoritmo de paridad podemos llegar a las siguientes conclusiones:

Ambos algoritmos reciben la misma entrada y terminan con la misma salida, verificando que ambos cumplen con la ecuación matemática presentada y que fue correctamente traducida en MIPS32.

La versión iterativa del algoritmo es mas eficiente en cuanto a consumo de memoria ya que es orden lineal, mientras que la versión recursiva es orden n.

La versión iterativa también es superior en términos de velocidad, ya que con entradas muy grandes la versión recursiva tiene que regresar luego de expandirse, mientras que la iterativa salta directamente al cumplir la condición del ciclo.

También se realizo una prueba con el máximo n permitido por la arquitectura en simple precisión (2.147.483.647), ambas versiones tardaron significativamente, pero mientras la versión iterativa logro completar el ciclo y terminar correctamente, la versión recursiva llego a un stack overflow que termino abruptamente con el programa.

Esto demuestra que la decisión entre el enfoque iterativo y recursivo no es solo una cuestión de preferencia, es algo que realmente puede cambiar si tu programa funciona o no.

2. Práctica 2: Algoritmos de Ordenamiento

2.1. Preguntas:

2.1.1. ¿Qué diferencias existen entre registros temporales ($t0–$t9) y registros guardados ($s0– $s7) y cómo se aplicó esta distinción en la práctica?

Los registros $t0-$t9 son registros temporales, por convención si una función o procedimiento quiere usarlos no tiene la obligación de resguardar los datos anteriores que estén en estos registros, previo a la ejecución de la misma. Mientras que los registros $s0-$s7 son registros guardados, es decir no volátiles, por lo que si una función o procedimiento quiere usarlos tiene la obligación de guardar su valor original en la pila y restaurarlo al finalizar su ejecución.

En el quicksort implementado, esto se puede ver en la gestion manual de la pila. Debido a que se utilizaron registros de argumento y temporales para controlar los limites

de la partición, y dado que estos no se preservan en llamadas recursivas, se guardaron los registros $ra, $a0, $a1 y $a2 en la pila.

En el bubble sort se utilizaron principalmente registros temporales y los registros $a0 y $a1 para los parámetros, y al no tener una función anidada y ser iterativo, no fue necesario respaldar valores de registros en la pila.

2.1.2. ¿Qué diferencias existen entre los registros $a0–$a3, $v0–$v1, $ra y cómo se aplicó esta distinción en la práctica?

En MIPS32 estos registros tienen sus funciones determinadas dentro de la convenciones de llamada, los registros $a0-$a3 son los argumentos, se utilizan para pasar a la función llamada.

Los registros $v0-$v1 son los valores de retorno, se usan para devolver los resultados desde la función llamada.

El registro $ra es la dirección de retorno, almacena la dirección de la instrucción a la que el programa debe volver una vez finalizada la función.

En el buble sort se utilizo $a0 para recibir la dirección base del arreglo y $a1 para recibir el tamaño del mismo. En el quicksort se utilizo $a0 para el arreglo, $a1 para el indice superior y $a2 para el indice inferior, también se utilizo el $v0 en la función partition para devolver el pivote.

Al ser recursivo en el bubble sort el $ra se mantiene intacto y solo se usa al final para regresar al main. Mientras que el quicksort al ser recursivo, el $ra se guarda en la pila en cada llamada para luego ser restaurado al regresar.

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

El uso de registros favorece la velocidad de ejecución al minimizar la latencia. En bubble sort, el rendimiento es alto en ciclos por instrucción ya que casi toda la lógica es registro a registro. En quicksort, aunque el algoritmo es mas rapido descartando las comparaciones innecesarias, sufre una penalización de rendimiento por tener que mantener un trafico constante entre la CPU y la RAM para el uso de la pila, esto demuestra que un mayor uso de registros en lugar de RAM favorece a la eficiencia en ejecución en sistemas RISC. Aun asi, si el arreglo fuera muy grande el quicksort ganaría por su lógica superior.

2.1.4. ¿Qué impacto tiene el uso de estructuras de control (bucles anidados, saltos) en la eficiencia de los algoritmos en MIPS32?

Las estructuras de control son necesarias para la lógica de ordenamiento, pero tienen un costo. Mientras que los bucles anidados de Bubble Sort son fáciles de seguir pero ineficientes por la repetición masiva de saltos, la estructura de saltos recursivos de Quicksort es más compleja pero reduce el volumen de instrucciones ejecutadas al dividir el problema.

2.1.5. ¿Cuáles son las diferencias de complejidad computacional entre el algoritmo Quicksort y el algoritmo alternativo? ¿Qué implicaciones tiene esto para la implementación en un entorno MIPS32?

Bubble sort tiene una complejidad de $O(n^2)$ mientras que quicksort tiene una complejidad $O(n\text{Log } n)$, al implementarlos en MIPS32 se ven los efectos reales en el hardware.

En MIPS32, la complejidad $O(n^2)$ del Bubble Sort se traduce en una ejecución masiva de instrucciones bge, addi y sll dentro de los bucles. Para arreglos grandes, esto satura el tiempo de procesamiento en el simulador mars. La eficiencia del Quicksort tiene un costo en MIPS: el uso de la memoria RAM. Mientras que la complejidad de Bubble Sort en memoria es $O(1)$ (solo usa el arreglo original), Quicksort tiene una complejidad espacial de $O(\log n)$ debido a la recursión. Cada nivel de recursión consume 20 bytes en la pila para guardar registros como $ra, $a0 y $a1.

En MIPS32, la pila es limitada. Una implementación de Quicksort con un pivote mal elegido en un arreglo muy grande podría causar un Stack Overflow al profundizar demasiado en las llamadas recursivas. El Bubble Sort, aunque más lento, es "más seguro.en este entorno ya que nunca desbordará la memoria de la pila.

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

1. Búsqueda de la instrucción:
   El procesador obtiene la instrucción desde la memoria del programa utilizando la dirección almacenada en el registro PC (Program Counter).

2. Decodificación y lectura de registros:
   El procesador interpreta la instrucción (determina si es de tipo R, I o J) y accede a los registros necesarios en el Banco de Registros.

3. Ejecución y Cálculo de Direcciones:
   La ALU (Unidad Aritmético Lógica) realiza la operación solicitada (suma, resta, comparaciones lógicas) o calcula la dirección efectiva de memoria para las cargas y almacenes.

4. Acceso a Memoria:
   Si la instrucción es de carga (lw) o almacenamiento (sw), el procesador accede a la memoria de datos. Si no es una instrucción de memoria, esta etapa es ignorada por la instrucción actual.

5. Escritura en el Registro:
   El resultado de la operación (ya sea de la ALU o de la memoria) se escribe de vuelta en el registro destino del Banco de Registros.

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

Predominaron las instrucciones de Tipo I, ya que los algoritmos de ordenamiento requieren un manejo constante de la memoria (acceso al arreglo y la pila) y de saltos condicionales para las comparaciones de datos.

2.1.8. ¿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 rendimiento baja porque los saltos (j, beq) obligan al procesador a desperdiciar ciclos de reloj esperando a conocer el destino del salto. Mientras que el código lineal fluye sin interrupciones por la línea de ejecución, el abuso de estructuras de control en

Bubble Sort y Quicksort genera esperas innecesarias que ralentizan el tiempo total de procesamiento por cada instrucción.

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

Al tener un conjunto de instrucciones simplificado y de tamaño fijo (32 bits), el procesador puede ejecutar casi todas las instrucciones en un solo ciclo de reloj. Esto permite que los bucles de comparación en el Bubble Sort tengan un tiempo de ejecución predecible y constante. RISC favorece los algoritmos de ordenamiento al minimizar el acceso a la memoria principal y maximizar el uso de registros de alta velocidad.

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

El modo de ejecución paso a paso es muy útil para validar la lógica de los ciclos y los intercambios, permitiendo hacer pruebas rápidas con entradas pequeñas para verificar que la lógica va por el camino correcto y pulir errores.

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

Conectando con la respuesta anterior considero que el modo paso a paso es la mayor ventaja, ya que en caso de un error logico o desbordamiento permite ver exactamente a partir de que linea empieza a fallar en la ejecución, también destaca la opción de retroceder en la ejecución.

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

Puede hacerse con la herramienta MIPS X-Ray, esta muestra un diagrama del camino de datos en tiempo real de la ejecucion del programa y las instrucciones, a través de esta animación, se comprobó visualmente que los operandos viajan desde el banco de registros hacia la ALU, y que el resultado retorna directamente a su registro de destino, puenteando por completo el bloque funcional de la memoria RAM.

2.1.13. Análisis y discusión de los resultados

Aunque el QuickSort es algoritmicamente superior $O(n \log n)$, su implementación en MIPS32 es mas compleja debido a la gestión de la pila. Mientras que el Bubble sort a pesar de su complejidad $O(n^2)$, tiene un código mas lineal y fácil de seguir gracias a su naturaleza iterativa.

Utilizando la herramienta de ejecución paso a paso se observo como el Quicksort consume memoria dinámica de forma recursiva, reservando 20 bytes por cada llamada en la pila, mientras que el bubble sort se mantiene con memoria lineal operando el arreglo y los registros temporales.

3. Anexos:

3.0.1. Algoritmo Paridad Recursivo:

```
parity:
        beq $a0,$zero,caso_base # Si n = 0 -> pasar al caso base

        addi $sp,$sp,-8         # reserva de espacio en la pila
        sw $ra,4($sp)
        sw $a0,0($sp)

        addi $a0,$a0,-1         # n - 1

        jal parity              # llamada recursiva

        lw $a0,0($sp)           # recuperamos el n original
        lw $ra,4($sp)           # y el valor de retorno
        addi $sp,$sp,8

        li $t0,1
        sub $v0,$t0,$v0         # 1 - n - 1

        jr $ra                  # retornar valor


        caso_base:
                li $v0,0        # retornar 0
                jr $ra
```

3.1. Algoritmo Paridad Iterativo

```
parity:
        li $v0,0        # resultado inicial
        li $t0,1        # constante = 1

        loop:
                beq $a0,$zero,fin_loop  # Si n = 0 parar el ciclo
                sub $v0,$t0,$v0         # res = 1 - res
                addi $a0,$a0,-1         # decrementar el contador
                    (n = n - 1)

                j loop                  # iterar
        fin_loop:
        jr $ra                          # retornar valor
```

3.1.1. Algoritmo QuickSort

```
partition:
```

```
                                # primero se calcula el valor y la
                                    direccion del pivote
    sll  $t0, $a2, 2            # offset high
    add  $t0, $a0, $t0          # $t0 = &arr[high]
    lw   $t9, 0($t0)            # $t9 = valor del pivote

    addi $t1, $a1, -1           # i = low - 1
    move $t2, $a1               # j = low

    loop:
        bge  $t2, $a2, endloop

        # cargar arr[j]
        sll  $t3, $t2, 2
        add  $t3, $a0, $t3      # $t3 = &arr[j]
        lw   $t4, 0($t3)        # $t4 = arr[j]

        bge  $t4, $t9, next_it

        # swap arr[i] y arr[j]
        addi $t1, $t1, 1        # i++

        sll  $t5, $t1, 2
        add  $t5, $a0, $t5      # $t5 = &arr[i]
        lw   $t6, 0($t5)        # $t6 = arr[i]

        sw   $t4, 0($t5)        # arr[i] = arr[j]
        sw   $t6, 0($t3)        # arr[j] = temp

    next_it:
        addi $t2, $t2, 1        # j++
        j    loop

    endloop:
        # colocamos el pivote en su lugar
        addi $t1, $t1, 1        # i + 1

        sll  $t5, $t1, 2        # offset (i+1)

        add  $t5, $a0, $t5      # sumamos a $a0

        lw   $t6, 0($t5)        # cargar quien estaba en i+1

        sw   $t9, 0($t5)        # arr[i+1] = pivote
        sw   $t6, 0($t0)        # arr[high] = el que estaba en i+1

        move $v0, $t1           # retornamos  ndice  i+1
        jr   $ra

quicksort:

```

```
52      # if (low >= high) return
53      bge  $a1, $a2, return_qs
54
55      # stack frame 20 bytes (ra, a0, a1, a2, y un espacio para
            pivote)
56      addi $sp, $sp, -20
57      sw   $ra, 16($sp)
58      sw   $a0, 12($sp)
59      sw   $a1, 8($sp)
60      sw   $a2, 4($sp)
61
62      jal  partition
63
64      sw   $v0, 0($sp)          # guardar pivote en la pila
65
66      # llamada por la izquierda
67      lw   $a1, 8($sp)          # restaurar low
68      lw   $v0, 0($sp)          # leer pivote
69      addi $a2, $v0, -1         # high = p - 1
70      jal  quicksort
71
72      # llamada por la derecha
73      lw   $a0, 12($sp)         # restaurar arr
74      lw   $a2, 4($sp)          # restaurar high original
75      lw   $v0, 0($sp)          # leer pivote de nuevo
76      addi $a1, $v0, 1          # low = p + 1
77      jal  quicksort
78
79      # retornamos stack y salida
80      lw   $ra, 16($sp)
81      addi $sp, $sp, 20
82      return_qs:
83      jr   $ra
```

3.1.2. Algoritmo Bubble Sort

```
1   bubbleSort:
2
3           #$a0= array, $a1= n
4           #$t0=i, $t1=j, $t3= intercambio
5
6           li $t0, 0         #i=0
7           addi $t4, $a1, -1         #$t4=n-1
8
9           forExt:
10          bge $t0, $t4, endforExt
11
12          li $t3, 0          #intercambio=0
13          li $t1, 0         #j=0
14          sub $t5, $a1, $t0
15          addi $t5, $t5, -1         #$t5= n-i-1
```

```
16
17                  forIn:
18                          bge $t1, $t5, endforIn
19
20                          #acceder a array[j]
21                          sll $t6, $t1, 2          #j*4
22                          add $t6, $a0, $t6        # $t6 apunta a
                               array[j]
23                          lw $t7, 0($t6)  #$t7=array[j]
24
25                          #acceder a array[j+1]
26                          lw $t9, 4($t6)  #$t9= array[j+1]
27
28                          ble $t7, $t9, endif
29
30                          if:
31                                  sw $t7, 4($t6)  #el valor
                                       guardado en array[j ] se guarda
                                        en array[j+1]
32                                  sw $t9, 0($t6)  #el valor
                                       guardado en array[j+1] se
                                       guarda en array[j]
33                                  li $t3, 1                       #
                                       intercambio=1
34                          endif:
35
36                          addi $t1, $t1, 1
37                          j forIn
38                  endforIn:
39                  beq $t3, $zero, endBlubbleSort
40                  addi $t0, $t0, 1
41                  j forExt
42                  endforExt:
43  endBlubbleSort:
44          jr $ra
```