UNIVERSIDAD DE SANTIAGO DE
CHILE FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Organización de Computadores

Laboratorio 2: MIPS

Nombre: Aylin Castillo

Sección: L-1

Profesor: Viktor Tapia

Ayudante: Nicolas Henríquez

Introducción

En el presente informe se documentará los distintos resultados obtenidos al realizar las actividades del laboratorio 2 de la asignatura Organización de Computadores, el cual tiene como objetivos realizar la verificación de números pares e impares, y las operaciones de multiplicación, factorial y división haciendo solo uso de operaciones aritméticas básicas, saltos y subrutinas en lenguaje MIPS y ejecutadas en el simulador MARS, teniendo en cuenta el uso de positivos, negativos y 2 decimales en el caso de la división.

Lo mencionado anteriormente se divide en 2 partes, la primera es sobre el uso de syscall haciendo verificación de números pares e impares y la segunda es de la aplicación de la multiplicación, factorial y división con números enteros.

Marco Teórico

En el informe se hará uso de distintos conceptos los cuales serán descritos a continuación:

- **Arquitectura MIPS:** sus siglas traducidas al español significan “microprocesador sin bloqueos en las etapas de segmentación (De Nihil, 2019).
- **Lenguaje ensamblador:** formato legible por humanos de instrucciones (Tapia, 2024).
- **Lenguaje de máquina:** formato legible por ordenador (1 y 0) (Tapia, 2024).
- **MARS:** programa hecho con Java que puede ensamblar y simular la ejecución de un programa realizado en el lenguaje ensamblador MIPS (*MARS MIPS Simulator - Missouri State University*, s. f.).
- **Registros:** pequeños montones de memoria que se encuentran en la CPU que son de acceso muy rápido (De Nihil, 2019).
- **Subrutina:** “Las subrutinas o subprogramas son fragmentos de código diseñados para realizar determinadas tareas y que pueden ser invocados desde diferentes puntos del programa principal o desde otras subrutinas”(Tema12_Ensamblador_MIPS, s. f.).
- **Syscall:** “sirve para pedirle alguna operación o servicio al sistema operativo. Antes de usar syscall tenemos que cargar un código numérico en el registro $v0 y en algunos casos poner un valor en $a0 como argumento” (Trini, 2020).
- **Stack:** Es una pila de subrutinas que se va llenando a medida que se van llamando en el programa. Tienen una capacidad limitada, ya que dependen de la memoria disponible (¿Qué Es un Stack Overflow (No el Sitio)?, 2015).
- **Multiplicación:** “es una suma abreviada de sumandos iguales, que pueden repetirse muchas veces” (Admin_Copesa, 2023).
- **Factorial:** se multiplican todos los números enteros y positivos que hay entre el número n (n!) y el número 1. En el caso de ser 0!, el resultado será 1 (Logo, s. f.).
- **División:** permite averiguar cuantas veces el dividendo está contenido en el divisor (Admin_Copesa, 2023).
- **Instrucciones de salto:** Son todas aquellas instrucciones que permiten

realizar distintos saltos en la ejecución de instrucciones de un programa, alteran el flujo en el cual se ejecuta el programa (Tapia, 2024).

- **Instrucciones aritméticas básicas:** Son todas aquellas instrucciones que permiten realizar operaciones matemáticas, tales como: sumar, multiplicar, restar, etc. (Tapia, 2024).

Desarrollo de la Solución

Para darle inicio al desarrollo de la solución del laboratorio se hizo uso de lo aprendido en el curso sobre MIPS, lo aplicado en el laboratorio 1 y las definiciones entregadas en el marco teórico, en específico sobre subrutinas, instrucciones de salto, uso de syscall y el uso de operaciones aritméticas básicas, como lo son la suma y resta para realizar cálculos más complejos, como lo son el de la multiplicación, factorial y división con números enteros.

Resultados

Después de terminado el desarrollo se obtuvieron los siguientes resultados:

- **Parte 1: Uso de syscall.**
  Para el desarrollo de la primera parte, se tuvo que investigar cómo obtener entradas del usuario y poder manipularlas dentro del programa como argumentos para las subrutinas que estarían encargadas de indicar el número máximo entre los números ingresados.

  La subrutina “maximoInt2” está encargada de hacer el cambio de argumentos en caso de ser el segundo número ingresado el máximo, esto para tener el correcto funcionamiento de las subrutinas posteriores.

```
calculoMaximo:
        slt $t5, $a1, $a2 #si $a1 < $a2? 1 $t5 = 1 si, = 0 no
        beq $t5, 1, maximoInt2
        jr $ra #vuelve al inicio


maximoInt2:
        move $v0, $a2
        move $a2, $a1
        move $a1, $v0
        jr $ra
```

Figura 1: Subrutinas cálculo máximo.

Después de identificar al máximo, el programa iba a hacer uso de una instrucción de “jr $ra” para volver al main y así realizar la verificación de si la diferencia entre los números enteros ingresados era par o impar. Cabe decir que esta diferencia siempre va a ser positiva debido a que el máximo se pone en el primer registro y el mínimo en el segundo registro de la operación add.

```
diferencia:
#aqui adentro se asume que $a1 siempre será el mayor
    sub $t2, $a1, $a2 # Calcular la diferencia (a0 - a1) y guardarla en $t2
    move $v0, $t2      # Guardar la diferencia en $v0 para devolverla
    jr $ra             # Retornar al llamador
```

Figura 2: Subrutina de diferencia.

Realizada la anterior subrutina, se vuelve a main para verificar la paridad, ejecutando las siguientes líneas de código.

```
# $v0 ahora tiene la diferencia
move $t3, $v0 # Guardar la diferencia en $t2

# Determinar si la diferencia es par o impar
andi $t3, $t2, 1 # Revisar si el número es par, 0 es par, 1 impar
beq $t3, 1, impar # Si $t3 es 1 (impar), salta a impar
j par # Si no es 1 (par), salta a par
```

Figura 3: Verificación paridad en main.

Dependiendo del valor obtenido en el registro $t3, se ejecutaban las siguientes subrutinas:

```
impar:
    # Imprimir el mensaje de que la diferencia es
    li $v0, 4
    la $a0, msjDiferencia
    syscall
    # Imprimir la diferencia
    li $v0, 1
    move $a0, $t2
    syscall
    # Imprimir que es impar
    li $v0, 4
    la $a0, msjImpar
    syscall
    # Imprimir un salto de linea
    li $v0, 4
    la $a0, salto
    syscall
    #$a3 pasa a ser el valor de la diferencia
    move $a3, $t2
    # Llamar a la subrutina de suma final
    jal sumaEntero
    # $v0 ahora tiene la suma
    move $t4, $v0 #guarda el resultado de la suma en $t4
    # Imprimir el mensaje de que la suma
    li $v0, 4
    la $a0, msjSumaImpar
    syscall
   #imprimir la suma
    li $v0, 1
    move $a0, $t4
    syscall
    j exit
```

```
par:
    # Imprimir el mensaje de que la diferencia es
    li $v0, 4
    la $a0, msjDiferencia
    syscall
    # Imprimir la diferencia
    li $v0, 1
    move $a0, $t2
    syscall
    # Imprimir que es par
    li $v0, 4
    la $a0, msjPar
    syscall
    #$a3 pasa a ser el valor de la diferencia
    move $a3, $t2
    #$a1 ya es el valor del segundo entero, pero estamos usando $a2 en la subrutina
    move $a2, $a1
    # Llamar a la subrutina diferencia
    jal sumaEntero
    # $v0 ahora tiene la suma
    move $t4, $v0 #guarda el resultado de la suma en $t4
    # Imprimir el mensaje de que la suma
    li $v0, 4
    la $a0, msjSumaPar
    syscall
   #imprimir la suma
    li $v0, 1
    move $a0, $t4
    syscall
    j exit
```

Figura 4: Subrutinas de paridad.

Como se puede observar en la figura anterior, las subrutinas se diferencian en solo el mensaje que imprimen en la consola y hacia qué tipo de suma realizan, ya que esta suma depende del tipo de paridad de la diferencia.

Se obtuvieron los siguientes resultados para las distintas entradas ingresadas para probar el funcionamiento del código:

Entradas	Paridad diferencia	Suma final
a: 4 b: 2	2 (par)	a = 4 + 2 = 6
a: 7 b: 2	5 (impar)	b = 5 + 2 = 7
a: -7 b: -2	5 (impar)	a = 5 - 7 = -2

Tabla 1: Distintos valores obtenidos en la parte 1.

- **Parte 2: Subrutinas para multiplicación y división de números.**
  Para el desarrollo de la segunda parte, se hizo uso de una subrutina por cada posible caso de multiplicación, los cuales son: ambos números positivos, ambos números negativos, el primer número negativo y el otro positivo y viceversa. Estos casos fueron controlados por las siguientes líneas de código, encargadas de hacer salto a distintas subrutinas dependiendo del resultado obtenido en los registros de los branch.

```
#Numeros enteros de prueba en $t0 y $t1
addi $t0, $zero, -10
addi $t1, $zero, 1

#caso base alguno sea 0
beqz $t0, esCero
beqz $t1, esCero

#en $t2 y $t3 se guardan "signos"
slt $t2, $t0, $zero      # si $t0 < 0, $t2 da 1
slt $t3, $t1, $zero #si $t1 < 0, $t3 da 1

#Casos negativos
beq $t2, $t3, argumentoAmbosNegativos
bne $t2, $zero, argumentoIzq
bne $t3, $zero, argumentoDer
#y si no cumple con ningun caso negativo:
j argumentoPositivos
```

Figura 5: Verificación de casos de multiplicación.

Cada caso lleva a las subrutinas que le asignan los registros “$a” para poder hacer uso de las subrutinas de multiplicación a futuro.

```
#caso negativo izquierda
argumentoIzq:
        move $a1, $t0
        move $a2, $t0
        j multiplicacionNegativaIzq

argumentoPositivos:
        move $a1, $t0
        move $a2, $t0
        j multiplicacion

#caso negativo derecha
argumentoDer:
        move $a1, $t1
        move $a2, $t1
        j multiplicacionNegativaDer

argumentoAmbosNegativos:
        slt $t2, $t0, $zero     # si $t0 < 0, $t2 da 1
        beq $t2, $zero, argumentoPositivos
        #convierte a positivos
        sub $t6, $zero, $t0       # Hacer $t0 positivo
        sub $t7, $zero, $t1     #hace t1 positivo
        move $a1, $t6
        move $a2, $t6
        j multiplicacionNegativaAmbos
```

Figura 6: Paso de argumentos por cada caso.

Las distintas multiplicaciones realizadas contienen el uso de una subrutina llamada “suma” que realiza el ciclo de sumas necesarias para obtener el resultado de la multiplicación y hace uso de los argumentos vistos anteriormente. Lo que cambia en la implementación de las multiplicaciones por caso es el orden de los argumentos y lo que se imprime por consola posteriormente, ya que en el caso de los números ingresados tienen distinto signo el resultado es negativo y si son del mismo su resultado es positivo.

```
multiplicacion:
        beq $t1, 1, exit
        jal suma
        #a2 es el acumulador
        move $a2, $v0
        sub $t1, $t1, 1
        j multiplicacion

multiplicacionNegativaIzq:
        beq $t1, 1, exit
        #argumentos pasados al inicio
        jal suma
        move $a2, $v0
        sub $t1, $t1, 1
        j multiplicacionNegativaIzq

multiplicacionNegativaDer:
        beq $t0, 1, exit
        #argumentos pasados al inicio
        jal suma
        move $a2, $v0
        sub $t0, $t0, 1
        j multiplicacionNegativaDer

multiplicacionNegativaAmbos:
        beq $t7, 1, exit
        jal suma
        move $a2, $v0
        sub $t7, $t7, 1
        j multiplicacionNegativaAmbos
```

```
suma:
# ingresa el acumulador y el valor de $t0
        add $t2, $a1, $a2
        move $v0, $t2
        jr $ra
```

Figura 7: Multiplicaciones y subrutina suma.

Se obtuvieron los siguientes resultados en la implementación de la multiplicación:

Entradas	Resultado
a: 10 b: 5	50
a: -10 b: -5	50
a: 10 b: -5	-50
a: -10 b: 5	-50
a: 10 b: 0	0

Tabla 2: Distintos valores obtenidos en la multiplicación.

Posteriormente se realizó la implementación del factorial, el cual consiste en multiplicaciones sucesivas desde el número 1 en adelante según se indique. Para esto se hizo uso de la subrutina de multiplicación entre dos números positivos implementada anteriormente, la cual estará en un ciclo que controlará el factorial. En esta parte no se logró hacer funcionar correctamente el código, ya que se va a un ciclo infinito que no se pudo solucionar antes de la entrega.

```
factorial:
    addi $a2, $zero, 0
    move $a1, $t8
    move $a2, $t9
    jal multiplicacion
    #acumulador factorial con resultado multiplicacion
    move $t9, $a2
    beq $t8, $t0, exit
    #contador
    addi $t8, $t8, 1
    j factorial

multiplicacion:
        beq $t1, 1, terminoMul
        jal suma
        #a2 es el acumulador que va guardando la multiplicacion
        move $a2, $v0
        sub $t1, $t1, 1
        j multiplicacion

terminoMul:
        jr $ra #vuelve a donde fue llamado multiplicacion


suma:
        add $t2, $a1, $a2
        move $v0, $t2
        jr $ra
```

Figura 8: Factorial y multiplicación de positivos.

La última parte consistía en implementar la división con enteros incluyendo 2 decimales en el caso de la división inexacta. Para esto al igual que en la multiplicación, se hicieron subrutinas para cada caso.

```
#Casos negativos:
slt $t3, $t0, $zero
slt $t4, $t1, $zero

#caso donde ambos son negativos o positivos
beq $t4, $t3, argumentoAmbosNegativos
#caso donde alguno de los dos sean negativos
bne $t3, $zero, divisionDividendoNegativo
bne $t4, $zero, divisionDivisorNegativo
```

```
argumentoPositivos:
        move $a1, $t0
        move $a2, $t1
        j divisionPositivos

argumentoAmbosNegativos:
        slt $t3, $t0, $zero
        beq $t3, $zero, argumentoPositivos
        #convierte a positivos
        sub $t4, $zero, $t0
        sub $t7, $zero, $t1
        j divisionAmbosNegativos
```

```
divisionPositivos:
        beqz $a1, exit
        addi $t2, $t2, 1
        sub $a1, $a1, $a2
        j divisionPositivos
divisionAmbosNegativos:
        beqz $t4, exit
        slt $t3, $t4, $t7
        bne $t3, $zero, primerDecimal
        addi $t2, $t2, 1
        sub $t4, $t4, $t7
        j divisionAmbosNegativos

divisionDividendoNegativo:
        bne $t3, $zero, convertirPositivoDividendo
        beqz $t0, exitNegativo
        sub $t0, $t0, $t1
        addi $t2, $t2, 1
        j divisionDividendoNegativo

divisionDivisorNegativo:
        bne $t4, $zero, convertirPositivoDivisor
        beqz $t0, exitNegativo
        sub $t0, $t0, $t1
        addi $t2, $t2, 1
        j divisionDivisorNegativo

convertirPositivoDivisor:
        sub $t1, $zero, $t1
        sub $t4, $t4, 1
        j divisionDivisorNegativo

convertirPositivoDividendo:
        sub $t0, $zero, $t0
        sub $t3, $t3, 1
        j divisionDividendoNegativo
```

Figura 9: Subrutinas división.

Para la implementación del cálculo de los dos decimales, se usó la forma básica para calcular el resto, la cual consiste en que cada vez que se agrega un decimal al resultado, se va multiplicando por 10 al resto de la operación.

```
primerDecimal:
        #para la suma
        move $a1, $t4
        move $a2, $t4
        addi $t5, $zero, 10 #multiplica x 10

        jal multiplicacion
        move $s2, $a2 #resultado multiplicacion = primer decimal
        sub $t4, $t4, $t7
        addi $t2, $t2, 1
        slt $t3, $t4, $t7
        beqz $t4, exit #si $t4 queda en 0
        bne $t3, $zero, segundoDecimal #si t6 < t7

segundoDecimal:
        move $a1, $t4
        move $a2, $t4
        addi $t5, $zero, 10 #multiplica x 10
        jal multiplicacion
        move $s1, $a2 #resultado multiplicacion = segundo decimal
        slt $t3, $t4, $t7
        bne $t3, $zero, exit #si t6 < t7
        j exit
```

Figura 10: Subrutinas decimales.

Las subrutinas de los dos decimales pedidos hasta el final del plazo para la entrega del laboratorio no funcionaron de manera correcta al probarlo con números cuya división es inexacta, pero las divisiones enteras funcionan bien independiente del signo de sus componentes, obteniendo así los siguientes resultados:

Entradas	Salida
a: 8 b: 4	2.00
a: -8 b: -4	2.00
a: -8 b: 4	-2
a: 8 b: -4	-2

Tabla 3: Distintos valores obtenidos en la división.

Conclusiones

Para finalizar, teniendo en cuenta los objetivos planteados al inicio del presente informe, se lograron cumplir en su mayoría, ya que se logra la verificación de paridad y la multiplicación en su totalidad y la división de manera parcial, pero hubieron problemas con el cálculo del factorial porque se va a un ciclo infinito y con el cálculo de los decimales en la división, estos no pudieron ser corregidos antes de la fecha de entrega del laboratorio.

La principal dificultad que se tuvo fue el cómo manejar las subrutinas y registros de manera efectiva, porque a veces ocurrían confusiones con el orden de estos y las instrucciones de salto para volver al código principal.

Se espera en proyectos futuros evitar que ocurran estos errores reforzando la teoría de MIPS para así implementar de mejor manera lo solicitado.

Referencias

Tapia, V. (2024). Lenguaje Ensamblador y Máquina. Organización de Computadores – Primer Semestre 2024.

De Nihil, V. T. L. E. (2019, 9 septiembre). Introducción a la Arquitectura del MIPS. La Cripta del Hacker. https://lacriptadelhacker.wordpress.com/2019/09/09/introduccion-a-la-arquitectura-del-mips/

MARS MIPS simulator - Missouri State University. (s. f.). https://courses.missouristate.edu/kenvollmar/mars/

tema12_ensamblador_MIPS. (s. f.). 12. Programación En Ensamblador MIPS. Recuperado 30 de mayo de 2024, de https://www.cartagena99.com/recursos/electronica/apuntes/tema12_ensamblador_MIPS.pdf

Trini, S. (2020, 13 agosto). Introducción a MIPS. la35.net. https://la35.net/orga/mips-intro.html#system-calls

¿Qué es un stack overflow (no el sitio)? (2015, 21 septiembre). El Blog de Make It Real. https://blog.makeitreal.camp/que-es-un-stack-overflow-desbordamiento-de-pila/#:~:text=El%20stack%20funciona%20como%20una,libro%20encima%20de%20la%20pila.

Admin_Copesa. (2023, 24 enero). Multiplicación y división de números naturales. Icarito. https://www.icarito.cl/2010/03/103-8690-9-5-numeros-naturales-conjunto-n.shtml/

Logo. (s. f.). Qué es la función factorial y cómo usarla - Paso a paso. Factorial HR. https://factorialhr.cl/numero-funcion-factorial