Grado en ingeniería informática

Estructura de Computadores 25/26

Grupo 84

Práctica 1

«Memoria de la Práctica 1»

Lucas Sotomayor Barrios

100538813@alumnos.uc3m.es

Mario Torrente Bucero

100429022@alumnos.uc3m.es

Profesor

Jose Manuel Perez Lobato

Tabla de Contenidos

1. Introducción

Esta práctica se centra en el desarrollo de funciones en ensamblador RISC-V utilizando el simulador CREATOR. El trabajo realizado abarca tres ejercicios principales. El primer ejercicio requiere la implementación del binomio de Newton para exponentes enteros, desarrollando funciones para el cálculo de coeficientes binomiales y la aplicación de la fórmula matemática correspondiente. El segundo ejercicio se aplica el teorema generalizado del binomio, lo que implica la implementación de aproximaciones de series de Taylor para funciones exponenciales y logarítmicas. Finalmente, el tercer ejercicio integra las funciones previas para realizar operaciones sobre matrices.

2. Ejercicio 1

2.1. Razonamiento

El objetivo de este ejercicio es calcular la potencia de un binomio $(a+b)^n$ aplicando el teorema del binomio de Newton. Este método evita calcular directamente $(a+b)^n$ mediante multiplicaciones sucesivas, descomponiendo en su lugar la operación en una suma de términos que involucran potencias individuales de a y b, ponderadas por coeficientes binomiales. La fórmula del binomio de Newton establece que:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

Para implementar esta fórmula, es necesario calcular el coeficiente binomial $\binom{n}{k}$ para cada término de la sumatoria. Este coeficiente representa el número de combinaciones de n elementos tomados de k en k, y se define mediante la siguiente expresión que utiliza factoriales:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

2.2. Implementación

2.2.1. Pseudocódigo

Para afrontar el ejercicio, utilizamos el lenguaje de alto nivel Python para poder estructurar la solución y poder trasladarla a ensamblador RISC-V de una manera más sencilla.

Primero, crearemos la función del factorial, que es necesaria para poder hacer los cálculos de las siguientes dos funciones.

```
def factorial(n):
  resultado = 1
  for i in range(1, n + 1):
    resultado *= i
  return resultado
```

Los parámetros se han guardado en pila para poder utilizarse en las siguientes funciones.

Esta es la función que nos permite poder hacer el el coeficiente binomial para poder implementarlo en la función del binomio de Newton.

```
def binomial_coef(n, k):
  n_factorial = factorial(n)
  k_factorial = factorial(k)
  diff_factorial = factorial(n - k)
  return n_factorial // (k_factorial * diff_factorial)
```

Finalmente, necesitamos implementar la función para poder hallar el binomio de Newton.

```
def newton_int(n, a, b)
    resultado = 0.0
    for k in range(n + 1):
        # Término k-ésimo: C(n,k) * a^(n-k) * b^k
        coef = binomial_coef(n, k)
        termino = coef * (a ** (n - k)) * (b ** k)
        resultado += termino
    return resultado
```

2.2.2. RISC-V

Tras haber presentado la implementación en Python, procedemos ahora a desarrollar la misma funcionalidad en lenguaje ensamblador RISC-V. Esta implementación mantiene la misma lógica algorítmica pero requiere una gestión explícita de recursos de bajo nivel.

2.2.2.1. Función Binomial_coef

La función calcula tres factoriales separadamente. Para cada factorial, utiliza un bucle que multiplica secuencialmente desde 1 hasta el valor correspondiente. De forma similar se calcula $k!$ y $(n-k)!$.

```
li t0, 1                # t0 = resultado = 1
li t1, 1                # t1 = contador = 1
ble s0, zero, factorial_n_end
factorial_n_loop:
    bgt t1, s0, factorial_n_end
    mul t0, t0, t1
    addi t1, t1, 1
    j factorial_n_loop
factorial_n_end:
    mv s2, t0           # s2 = n!
```

Finalmente, calcula el coeficiente binomial:

```
mul t0, s3, s4          # t0 = k! * (n-k)!
div a0, s2, t0          # a0 = n! / (k! * (n-k)!)
```

2.2.2.2. Función Newton_int

Implementa un bucle que itera desde $k = 0$ hasta $k = n$. En cada iteración llama a `Binomial_coef` para calcular el coeficiente:

```
newton_loop:
    bgt s1, s0, newton_end  # Si k > n, terminar

    mv a0, s0               # a0 = n
    mv a1, s1               # a1 = k
    jal ra, Binomial_coef   # Llamar función
    mv s2, a0               # s2 = C(n,k)
```

Utiliza la función `pow` para calcular a^{n-k} y b^k:

```
fmv.s fa0, fs0          # fa0 = a
sub a0, s0, s1          # a0 = n - k
jal ra, pow             # Llamar función pow
fmv.s fs3, fa0          # fs3 = a^(n-k)

fmv.s fa0, fs1          # fa0 = b
mv a0, s1               # a0 = k
jal ra, pow             # Llamar función pow
fmv.s fs4, fa0          # fs4 = b^k
```

Calcula el término completo y lo suma al acumulador:

```
fcvt.s.w ft0, s2        # ft2 = C(n,k) como float
fmul.s ft1, ft0, fs3    # ft3 = C(n,k) * a^(n-k)
fmul.s ft1, ft1, fs4    # ft3 = C(n,k) * a^(n-k) * b^k
fadd.s fs2, fs2, ft1    # resultado += término

addi s1, s1, 1
j newton_loop
```

2.2.3. Casos de Prueba

Datos a introducir:	Descripción de la prueba:	Resultado esperado:	Resultado Obtenido:
3, 4, 4	b y n iguales	2401	2401
1, 1, 6	a y b iguales	64	64
1, 0, 10	valor de b=0	1	1

3. Ejercicio 2

3.1. Razonamiento

Para este ejercicio, el objetivo es desarrollar una función para calcular el valor de $(a + b)^n$ siendo a, b y n tres números reales aplicando el teorema generalizado del binomio de Newton:

$$(x+y)^r = \sum_{k=0}^{\infty} \binom{r}{k} x^{r-k} y^k$$

Para este ejercicio, aproximaremos la serie utilizando los primeros 20 términos, lo que nos proporciona una precisión suficiente cuando a y b están próximos en valor. Se debe hacer uso de funciones como las siguientes para poder construir el `powf` que permite calcular $a * b$ con a y b números reales:

$$\binom{r}{k} = \frac{1}{k!} \prod_{n=0}^{k-1} (r-n) = \frac{r(r-1)(r-2)\cdots(r-k+1)}{k!}$$

$$a^b = e^{b \cdot \ln(a)}$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

3.2. Implementación

3.2.1. Pseudocódigo

Empleamos para el pseudocodigo el lenguaje de Python para ver de forma más clara y sencilla el planteamiento del Ejercicio 2. Para ello comenzamos por las primeras funciones que necesitamos para poder construir la función powf que usaremos después. Dichas funciones son:

La funcion arcotangente.

```
def arctan(x):
    term = x
    sum = x
    signo = -1

    para k = 1 hasta 20:
        term = term * x * x
        divisor = 2*k + 1
        sum = sum + signo * (term / divisor)
        signo = -signo
    retornar sum
```

Y la función $\ln()$ que serviran para resolver el $a^b = e^{b \cdot \ln(a)}$.

```
def Ln(x):
    z = (x - 1) / (x + 1)
    z_cuadrado = z × z
    resultado = 0
    pot_z = z

    para i desde 0 hasta 19:
        denominador = 2 × i + 1
        resultado += pot_z / denominador
        pot_z = pot_z × z_cuadrado
    return 2 × resultado
```

A continuación vemos como es la función de e^x, concluyendo así con las funciones que sirven para que el powf funcione correctamente.

```
def E(x):
    resultado = 1.0
    termino = 1.0

    para i desde 1 hasta 19:
        termino = termino × x / i
        resultado += termino
    retornar resultado
```

La función powf acepta a y b en coma flotante en simple precisión devolviendo a^b (a y b son números reales).

```
def powf(a,b):
    x = a
    y = b

    ln_a = Ln(x)
    prod = y * ln_a
    resultado = E(prod)  # calcula e^(b * ln(a))
```

Y finalmente la funcion Newton que aproxima $(a+b)^n$ mediante la serie binomial generalizada:

```
Newton_real(a, b, n):
  if n < 0:
      return NaN
  term = powf(a, n)    # a^n
  sum = term
  for k = 1 hasta 19 hacer:
      f = (n - (k - 1)) / k
      term = term * f * (b / a)
      sum = sum + term
  return sum
```

3.2.2. RISC-V

Para pasar de pseudocodigo a ensamblador los parametros deben de guardarse en pila para poder utilizarse en las funciones siguientes:

3.2.3. Arctanh

```
    li t2, 20
    bge t1, t2, Arctanh_final

    # term = term * y * y
    fmul.s fs1, fs1, fs0
    fmul.s fs1, fs1, fs0
    # divisor = 2*k + 1
    add  t3, t1, t1   # t3 = t1 + t1 = 2*k
    addi t3, t3, 1    # t3 = 2*k + 1
    fcvt.s.w ft0, t3

    # sum += term / divisor
    fdiv.s ft1, fs1, ft0
    fadd.s fs2, fs2, ft1

    addi t1, t1, 1
    j Arctanh_bucle
```

3.2.3.1. ln()

```
    fmv.s fs0, fa0
              # comprobacion de NAN (si x es negativo)
    fmv.s ft0, fs0
    fsub.s ft0, ft0, ft0
    fle.s t0, fs0, ft0      # t0 = 1 si x <= 0
    bne t0, zero, Ln_nan    # saltar si x <= 0

    li t0, 1
    fcvt.s.w ft0, t0
    fsub.s ft1, fs0, ft0  # x - 1
    fadd.s ft2, fs0, ft0  # x + 1
    fdiv.s fs1, ft1, ft2  # y = (x-1)/(x+1)

    fmv.s fa0, fs1
    jal ra, Arctanh       # llama Arctanh
    li t1, 2
    fcvt.s.w ft0, t1
    fmul.s fa0, fa0, ft0  # ln(x) = 2 * artanh(y)
    j Ln_fin              # evita ejecutar Ln_nan cuando x > 0
```

3.2.3.2. e^x (Taylor)

```
    li t2, 20           # limite de la serie
    bge t1, t2, E_fin   # si k >= 20 salir

    # term = term * x / k
    fcvt.s.w ft0, t1
    fmul.s fs2, fs2, fs0 # term *= x
    fdiv.s fs2, fs2, ft0 # term = term / k
    fadd.s fs1, fs1, fs2 # sum += term
    addi t1, t1, 1       # k++
```

3.2.3.3. Powf

```
    fmv.s fs0, fa0      # e^(b * ln(a))
    fmv.s fs1, fa1
    fmv.s fa0, fs0
    jal  ra, Ln       # fa0 = ln(a)
    fmul.s fa0, fa0, fs1  # fa0 = b * ln(a)
    jal  ra, E        # fa0 = e^(b*ln(a)) = a^b
```

3.2.3.4. Newton_bucle

```
    li   t2, 20
    bge  t1, t2, Newton_fin   # salir en t1 >= 20

    fcvt.s.w ft0, t1
    addi t3, t1, -1
    fcvt.s.w ft1, t3    # float(k - 1)
    fsub.s ft2, fs2, ft1
    fdiv.s ft3, ft2, ft0  # n - (k - 1)
    fdiv.s ft4, fs1, fs0  # (n - (k - 1)) / k
    fmul.s fs3, fs3, ft3  # b / a
    fmul.s fs3, fs3, ft4
    fadd.s fs4, fs4, fs3
    addi t1, t1, 1
```

Como resultado concluyente `fa0` contiene la suma truncada de la serie binomial: $\sum\{k = 0..19\}\binom{n}{k}a^{\{n-k\}}b^k$, es decir una aproximación de $(a + b)^n$ truncada a los primeros 20 términos.

3.2.4. Casos de Prueba

Datos a introducir:	Descripción de la prueba:	Resultado esperado:	Resultado Obtenido:
2, 1, 4	a, b y n positivos	81	81
3,6,4	a, b y n positivos	6561	6561.005859375
2,7,-5	valor de n negativo	NaN	NaN

4. Ejercicio 3

4.1. Razonamiento

La función `Compute_pows` constituye una implementación que permite implementar una matriz cuadrada $N \times N$ llamando repetidas veces a la función `Newton_real`. Tenemos que recorrer cada posición de la matriz utilizando dos bucles anidados (filas y columnas), donde para cada elemento en la posición `[i][j]` se calcula su valor usando `Newton_real` $(a+i, b+j, c)$. La implementación en ensamblador requiere considerar que la reserva de memoria para la matriz se realiza en el segmento .data utilizando la directiva `.zero` con un tamaño de $M \times M \times 4$ bytes.

4.2. Implementación

4.2.1. Pseudocódigo

```
def Compute_pows(A, N, a, b, c):
    for i in range(N):
        for j in range(N):
            arg1 = a + i
            arg2 = b + j
            arg3 = c
            resultado = Newton_real(arg1, arg2, arg3)
            A[i][j] = resultado
```

4.2.2. RISC-V

Para la creación de la matriz, procedemos a iterar un bucle para las filas y para las columnas. Por ejemplo, en el siguiente código esta la iteración de las columnas de la matriz. Además, necesitamos llamar a la función de `Newton_real`del ejercicio 2 para poder hacer los cálculos respectivos.

```
bucle_j:
    bge  s3, s1, cambio_fila # si j >= N, siguiente fila
    fcvt.s.w ft0, s2        # convertir i a float
    fadd.s fa0, fs0, ft0    # fa0 = a + i
    fcvt.s.w ft0, s3        # convertir j a float
    fadd.s fa1, fs1, ft0    # fa1 = b + j
    fmv.s fa2, fs2 # c
    jal  ra, Newton_real
```

Después de tener los resultados de las filas y columnas las guardamos en la matriz.

```
# Guardar resultado en matriz
    mul  t0, s2, s1         # t0 = i * N
    add  t0, t0, s3         # t0 = i * N + j
    li   t1, 4
    mul  t0, t0, t1         # t0 = (i * N + j) * 4
    add  t0, s0, t0         # ubicar el elemento en matriz

    fsw  fa0, 0(t0)

    addi s3, s3, 1          # j++
    j    bucle_j
```

4.2.3. Casos de Prueba

Datos a introducir:	Descripción de la prueba:	Resultado esperado:	Resultado Obtenido:
2, 0.5, 1.5, 2	con valores decimales	4, 9, 9, 16	4, 9, 9, 16
2,0,1,4	a igual a cero	1,16, 16, 81	1, 16, 16, 81
2, 2, 2, 2	todos los números iguales	16, 25, 25, 36	16, 25, 25, 36

5. Conclusiones

En conclusión, la realización de esta práctica ha permitido poner en práctica los conocimientos sobre la arquitectura RISC-V. A través de los tres ejercicios desarrollados, hemos implementado funciones matemáticas complejas que van desde el cálculo del binomio de Newton hasta la aproximación de series de Taylor mediante ensamblador. Sin embargo, durante el proceso nos encontramos con diversos problemas, especialmente en la gestión de la pila, el manejo de registros flotantes y el entendimiento de errores. La estimación del tiempo empleado en esta práctica fue de aproximadamente 2 semanas, donde tuvimos que repasar los conceptos teóricos y aplicarlos a los ejercicios propuestos.

La experiencia adquirida durante esta práctica nos ha hecho comprender como varios lenguajes de programación como Python, de alto nivel, nos ahorran mucho tiempo de programación. En esta práctica hemos tenido que manejar mucho con la memoria y los registros que cuando hemos trabajando con otros lenguajes de programación, ni lo mencionábamos. Además, hemos aprendido la diferencia de como manejar números flotantes y manejar enteros para poder hacer cálculos. Finalmente, hemos aprendido que en futuros trabajos podamos implementar algoritmos aprendidos en asignaturas pasadas para poder optimizar nuestro código.