**Computer Architecture CSF342**

**Lab sheet 3**

**Topic**: Understand IEEE floating-point representation, perform floating-point calculations, and implement numerical algorithms using Coprocessor 1.

#### **1. Floating-Point Representation**

**Single-Precision (32-bit)**:

* **Sign (1 bit)**: 0 = positive, 1 = negative
* **Exponent (8 bits)**: Bias = 127
* **Mantissa (23 bits)**: Leading 1 implied
* **Range**: ±1.18×10⁻³⁸ to ±3.4×10³⁸

**Double-Precision (64-bit)**:

* **Sign (1 bit)**
* **Exponent (11 bits)**: Bias = 1023
* **Mantissa (52 bits)**
* **Range**: ±2.23×10⁻³⁰⁸ to ±1.80×10³⁰⁸

**Example**: 0.75 in single-precision

* Binary: 0.11₂ = 1.1₂ × 2⁻¹
* Sign: 0
* Exponent: 126 (127 - 1 = 126 → 01111110)
* Mantissa: 10000000000000000000000
* Final: 0 01111110 10000000000000000000000

#### **2. Floating-Point I/O**

**Reading Floats**:

li $v0, 6 # Read single-precision float → $f0

syscall

li $v0, 7 # Read double-precision float → $f0

syscall



**Printing Floats**:

mov.s $f12, $f0 # Load single-precision value

li $v0, 2 # Print single-precision

syscall

mov.d $f12, $f0 # Load double-precision value

li $v0, 3 # Print double-precision

syscall



#### **3. Coprocessor 1 Operations**

**Conversion Instructions**:

cvt.s.w $f2, $f4 # Convert word to single-precision

cvt.d.w $f2, $f4 # Convert word to double-precision

cvt.w.s $f2, $f4 # Convert single to word (truncates)

cvt.w.d $f2, $f4 # Convert double to word



**Arithmetic Operations**:

add.s $f2, $f4, $f6 # Single: $f2 = $f4 + $f6

sub.d $f2, $f4, $f6 # Double: $f2 = $f4 - $f6

mul.s $f2, $f4, $f6 # Multiply single

div.d $f2, $f4, $f6 # Divide double



**Floating-Point Comparison**

In MIPS, floating-point comparisons are handled differently than integer comparisons. Use the following instructions:

1. **Set comparison flag**:

c.eq.d $f2, $f4 # Set flag if $f2 == $f4

c.lt.d $f2, $f4 # Set flag if $f2 < $f4

c.le.d $f2, $f4 # Set flag if $f2 <= $f4



1. **Branch based on flag**:

bc1t label # Branch if comparison flag is true

bc1f label # Branch if comparison flag is false

### Example: Conditional Branch with Floating-Point

# Check if $f2 > 0.0

mtc1 $zero, $f4 # Load 0.0 into $f4

c.lt.d $f4, $f2 # Set flag if 0.0 < $f2

bc1t positive\_value # Branch if true

# Handle non-positive case

j end\_check

positive\_value:

# Handle positive case

end\_check:

### Pro Tips

1. Always initialize floating-point registers before using them
2. Use double-precision (.d) instructions for better precision
3. Be cautious of precision limitations when comparing floating-point numbers
4. Use the appropriate conversion instructions (cvt.s.w, cvt.d.w, etc.) when mixing integers and floating-point numbers
5. Remember that floating-point operations are slower than integer operations, so minimize them when possible

#### **4. Example: Pi Approximation**

**Single-Precision**:

.data

seven: .float 7.0

twentytwo: .float 22.0

.text

main:

lwc1 $f2, twentytwo

lwc1 $f4, seven

div.s $f12, $f2, $f4 # 22/7

li $v0, 2

syscall



**Double-Precision**:

.data

seven\_d: .double 7.0

twentytwo\_d: .double 22.0

.text

main:

ldc1 $f2, twentytwo\_d

ldc1 $f4, seven\_d

div.d $f12, $f2, $f4 # 22/7

li $v0, 3

syscall



#### **5. Newton-Raphson Square Root**

**Formula**: xₙ₊₁ = (xₙ + n/xₙ)/2

**C Implementation**:

float sqrt(float n) {

float x = n;

for (int i = 0; i < 10; i++) // 10 iterations

x = (x + n/x) / 2.0;

return x;

}



**MIPS Implementation**:

.data

n: .float 25.0 # Input number

two: .float 2.0

epsilon: .float 0.0001 # Precision threshold

.text

main:

lwc1 $f0, n # $f0 = n

mov.s $f1, $f0 # x = n (initial guess)

lwc1 $f2, two

li $t0, 0 # i = 0

sqrt\_loop:

div.s $f3, $f0, $f1 # n/x

add.s $f3, $f1, $f3 # x + n/x

div.s $f1, $f3, $f2 # (x + n/x)/2 → new x

addi $t0, $t0, 1

li $t1, 10

blt $t0, $t1, sqrt\_loop

mov.s $f12, $f1 # Print result

li $v0, 2

syscall



#### **6. Student Task: Euclidean Distance**

**Pseudocode**:

vector1 = [1.0, 2.0, 3.0]

vector2 = [4.0, 5.0, 6.0]

sum = 0.0

for i in range(3):

diff = vector1[i] - vector2[i]

sum += diff \* diff

distance = sqrt(sum)



**Requirements**:

* Hardcode two 3D vectors
* Calculate squared differences
* Use your Newton-Raphson implementation for square root
* Print the final distance

**Sample Output**:

Euclidean distance: 5.196



#### **7. Calculating sin(θ) using Maclaurin Series**

### Maclaurin Series Overview

The Maclaurin series is a special case of the Taylor series expansion around zero. For sin(x), the Maclaurin series is:

sin(x) = x - x³/3! + x⁵/5! - x⁷/7! + x⁹/9! - ...

We can approximate sin(x) by taking the first few terms of this series. For this lab, we'll use the first 5 terms.

### C Code Implementation

#include <stdio.h>

double sin(double x) {

double result = x;

double term = x;

double x\_squared = x \* x;

// Calculate terms 2-5

term = term \* x\_squared / (2\*3);

result -= term;

term = term \* x\_squared / (4\*5);

result += term;

term = term \* x\_squared / (6\*7);

result -= term;

term = term \* x\_squared / (8\*9);

result += term;

return result;

}

int main() {

double angle = 0.785398; // π/4 radians (45 degrees)

printf("sin(%f) = %f\n", angle, sin(angle));

return 0;

}

### MIPS Assembly Implementation (using loop unrolling)

.data

angle: .double 0.7853981633974483 # pi/4 radians (45 degrees)

x\_squared: .double 0.0

term: .double 0.0

result: .double 0.0

const\_2\_3: .double 6.0 # 2\*3

const\_4\_5: .double 20.0 # 4\*5

const\_6\_7: .double 42.0 # 6\*7

const\_8\_9: .double 72.0 # 8\*9

output\_msg: .asciiz "sin(theta) = "

.text

main:

# Load angle into $f0

ldc1 $f0, angle

# Initialize result with x (first term)

mov.d $f2, $f0 # result = x

mov.d $f4, $f0 # term = x

# Calculate x²

mul.d $f6, $f0, $f0 # $f6 = x\_squared = x \* x

# Calculate second term: -x³/3! = -x\*x²/(2\*3)

mul.d $f4, $f4, $f6 # term = term \* x\_squared

ldc1 $f8, const\_2\_3

div.d $f4, $f4, $f8 # term = term / (2\*3)

sub.d $f2, $f2, $f4 # result -= term

# Calculate third term: +x⁵/5! = +term\*x²/(4\*5)

mul.d $f4, $f4, $f6 # term = term \* x\_squared

ldc1 $f8, const\_4\_5

div.d $f4, $f4, $f8 # term = term / (4\*5)

add.d $f2, $f2, $f4 # result += term

# Calculate fourth term: -x⁷/7! = -term\*x²/(6\*7)

mul.d $f4, $f4, $f6 # term = term \* x\_squared

ldc1 $f8, const\_6\_7

div.d $f4, $f4, $f8 # term = term / (6\*7)

sub.d $f2, $f2, $f4 # result -= term

# Calculate fifth term: +x⁹/9! = +term\*x²/(8\*9)

mul.d $f4, $f4, $f6 # term = term \* x\_squared

ldc1 $f8, const\_8\_9

div.d $f4, $f4, $f8 # term = term / (8\*9)

add.d $f2, $f2, $f4 # result += term

# Print result

li $v0, 4

la $a0, output\_msg

syscall

mov.d $f12, $f2

li $v0, 3

syscall

# Exit

li $v0, 10

syscall

#### **8. Student Task: Trigonometric Functions Calculator**

### Task Description

Write a MIPS program that:

1. Takes a floating-point angle in degrees as user input
2. Converts the angle to radians (radians = degrees × π/180)
3. Calculates sin(θ), cos(θ), and tan(θ) using the Maclaurin series approximation
4. Prints all three results

Note: cos(θ) = sin(π/2 - θ), so you can reuse your sin function to calculate cosine.

### Sample Input/Output

Enter angle in degrees: 45.0

sin(45.0) = 0.7071

cos(45.0) = 0.7071

tan(45.0) = 1.0000

### Implementation Guidance

1. Use the syscall service 7 to read a double-precision floating-point number
2. Convert degrees to radians: multiply by π (≈3.141592653589793) and divide by 180
3. Implement the sin function as shown in section 7
4. For cos(θ), calculate sin(π/2 - θ)
5. For tan(θ), divide sin(θ) by cos(θ)
6. Print the results using syscall service 3 for double-precision numbers

### Challenge

Extend your program to handle angles greater than 90 degrees by using trigonometric identities to reduce them to the first quadrant.

## Appendix: Floating-Point Tips for MIPS

### Important Note

The Maclaurin series approximation works best for small angles (close to zero). For larger angles, you should reduce the angle to the range [-π, π] using angle reduction techniques for more accurate results.

### **Appendix: Floating-Point Tips**

1. **Comparison Instructions**:

c.eq.s $f2, $f4 # Set flag if $f2 == $f4

c.lt.s $f2, $f4 # Set flag if $f2 < $f4

bc1t label # Branch if flag true

bc1f label # Branch if flag false



1. **Precision Warning**:
   * Single-precision has ~7 decimal digits precision
   * Double-precision has ~16 decimal digits precision
2. **NaN and Infinity**:
   * Check for NaN: Compare value with itself (NaN != NaN)
   * Detect infinity: Check for exponent all 1s, mantissa 0
3. **Performance Tip**:
   * Use double-precision only when necessary
   * Minimize conversions between formats

**Example Comparison**:

# Check if $f0 >= 0.0

mtc1 $zero, $f2 # $f2 = 0.0

c.lt.s $f0, $f2 # Set flag if $f0 < 0.0

bc1f non\_negative # Branch if false (>= 0.0)

# Handle negative case

non\_negative:



**Remember**: Floating-point comparisons use special flag registers, not standard condition codes!

### **Understanding Floating-Point Representation**

* **Why "Floating" Point?**
  + The term "floating point" refers to the fact that the **decimal (or binary) point can "float"** to any position relative to the significant digits of the number.
  + This is achieved by using an **exponent**. The exponent allows the representation of numbers with vastly different magnitudes (like the mass of an electron and the mass of the sun) using a fixed number of bits.
  + This contrasts with **fixed-point** representation, where the decimal point's position is predetermined and fixed, limiting the range of numbers that can be represented.
* **The Three Components: Sign, Exponent, and Mantissa** Any floating-point number is built from these three parts:
  + **Sign (1 bit):** A single bit that indicates whether the number is positive (0) or negative (1).
  + **Exponent (e bits):** This is the "floating" part. It represents a power (or exponent) that the base (2, in binary) is raised to. This scales the number up or down by powers of two, allowing for a huge range of values.
    - **Bias:** To represent both positive and negative exponents, a fixed value called a *bias* is subtracted from the stored exponent value. For an e-bit exponent, the bias is typically 2^(e-1) - 1.
  + **Mantissa (m bits) (or Significand):** This represents the significant digits (or precision) of the number. It's the part of the number that comes after the binary point. In normalized form, it's assumed to be 1.mantissa (the leading 1 is implicit to save a bit).
* **Understanding Range with an Imaginary 8-Bit Format** Let's invent a simple 8-bit floating-point format to see how the range is calculated:
  + **Format:** 1 bit (sign) + 4 bits (exponent) + 3 bits (mantissa)
  + **Exponent Bias:** For a 4-bit exponent, the bias is 2^(4-1)-1 = 7.
  + **Exponent Values:**
    - The 4-bit exponent can store values from 0 (binary 0000) to 15 (binary 1111).
    - The *actual* exponent used is stored\_value - 7. This gives a range of actual exponents from -7 (0 - 7) to 8 (15 - 7).
  + **Mantissa Values:** With 3 bits, the fractional part can be from .000 to .111 (binary). Because of the implied leading 1, the actual significand values range from 1.000 (=1.0) to 1.111 (≈1.875).
  + **Range Calculation:**
    - **Smallest Positive Number:** This uses the smallest exponent and the smallest mantissa.
      * Sign = 0 (positive)
      * Exponent = 0000 (actual exponent = 0 - 7 = -7)
      * Mantissa = 000 (value = 1.0)
      * Value = +1.0 \* 2^(-7) = 2^(-7) ≈ 0.0078125
    - **Largest Positive Number:** This uses the largest exponent and the largest mantissa.
      * Sign = 0
      * Exponent = 1111 (actual exponent = 15 - 7 = 8)
      * Mantissa = 111 (value ≈ 1.875)
      * Value = +1.875 \* 2^(8) = 1.875 \* 256 = 480
  + **Conclusion:** Our tiny **8-bit** format can represent numbers from about **0.008 to 480**. A signed 8-bit integer, by comparison, can only represent numbers from -128 to 127. This demonstrates the floating-point format's incredible advantage in representing a wide range of values.
* **Example: Representing 0.75 in our 8-bit format**
  1. **Convert to Binary:** 0.75 in decimal is 0.11 in binary.
  2. **Normalize:** 0.11 (binary) is 1.1 \* 2^(-1).
  3. **Identify Parts:**
     + **Sign:** 0 (positive)
     + **Exponent:** The actual exponent is -1. We need to store -1 + bias = -1 + 7 = 6. 6 in 4-bit binary is 0110.
     + **Mantissa:** The part after the decimal point in 1.1 is 1. We have 3 bits for the mantissa, so we store 100 (we add trailing zeros to fill the space).
  4. **Final Bit Pattern:** 0 0110 100