MIPS Assembly Lab Solutions - All Unsolved Problems

Lab Sheet 1: User-Input Adder

Problem Statement: Modify the hardcoded adder to prompt the user for two integers, read them, compute and print their sum.

Expected Output:

```
Enter first number: 5
Enter second number: 3
Sum = 8
```

Solution:

```
.data
prompt1:    .asciiz "Enter first number: "
prompt2:    .asciiz "Enter second number: "
result_msg: .asciiz "Sum = "
newline:    .asciiz "\n"

.text
main:
    # Print first prompt
    li $v0, 4
    la $a0, prompt1
    syscall

    # Read first number
    li $v0, 5
    syscall
    move $s0, $v0       # Store first number in $s0

    # Print second prompt
    li $v0, 4
    la $a0, prompt2
```

```
    syscall

    # Read second number
    li $v0, 5
    syscall
    move $s1, $v0       # Store second number in $s1

    # Calculate sum
    add $t0, $s0, $s1   # $t0 = $s0 + $s1

    # Print result message
    li $v0, 4
    la $a0, result_msg
    syscall

    # Print sum
    li $v0, 1
    move $a0, $t0
    syscall

    # Print newline
    li $v0, 4
    la $a0, newline
    syscall

    # Exit
    li $v0, 10
    syscall
```

Explanation: This program uses syscall 4 (print string) and syscall 5 (read integer) to interact with the user, then performs addition and displays the result.

Lab Sheet 2: Task 1 - GCD Calculator

Problem Statement: Calculate the Greatest Common Divisor of two positive integers using the Euclidean algorithm.

Expected Output:

```
Enter first number: 54
Enter second number: 24
GCD: 6
```

Solution:

```
.data
prompt1:    .asciiz "Enter first number: "
prompt2:    .asciiz "Enter second number: "
result_msg: .asciiz "GCD: "
newline:    .asciiz "\n"

.text
main:
    # Get first number
    li $v0, 4
    la $a0, prompt1
    syscall

    li $v0, 5
    syscall
    move $s0, $v0       # $s0 = first number

    # Get second number
    li $v0, 4
    la $a0, prompt2
    syscall

    li $v0, 5
    syscall
    move $s1, $v0       # $s1 = second number

    # GCD calculation using Euclidean algorithm
    move $t0, $s0       # $t0 = a
    move $t1, $s1       # $t1 = b

gcd_loop:
```

```
    beqz $t1, gcd_done  # If b == 0, GCD is in $t0

    # Calculate a mod b
    div $t0, $t1
    mfhi $t2           # $t2 = a mod b

    # a = b, b = a mod b
    move $t0, $t1
    move $t1, $t2

    j gcd_loop

gcd_done:
    # Print result
    li $v0, 4
    la $a0, result_msg
    syscall

    li $v0, 1
    move $a0, $t0
    syscall

    # Exit
    li $v0, 10
    syscall
```

Explanation: Uses the Euclidean algorithm where GCD(a,b) = GCD(b, a mod b) until b becomes 0.

Lab Sheet 2: Task 2 - Array First Derivative

Problem Statement: Calculate the first derivative of a hardcoded array where derivative[i] = array[i+1] - array[i].

Expected Output:

```
Input: [5, 2, 7, 1, 3, 8, 4, 9, 6, 0]
Output: [-3, 5, -6, 2, 5, -4, 5, -3, -6]
```

Solution:

```
.data
array:      .word 5, 2, 7, 1, 3, 8, 4, 9, 6, 0
derivative: .space 36              # Space for 9 integers (9*4 bytes)
input_msg:  .asciiz "Input: ["
output_msg: .asciiz "Output: ["
comma:      .asciiz ", "
close_bracket: .asciiz "]\n"
space:      .asciiz " "

.text
main:
    # Print input array
    li $v0, 4
    la $a0, input_msg
    syscall

    la $s0, array           # Base address of input array
    li $t0, 0               # Counter

print_input:
    beq $t0, 10, print_input_done

    # Print number
    lw $a0, 0($s0)
    li $v0, 1
    syscall

    # Print comma if not last element
    beq $t0, 9, skip_comma1
    li $v0, 4
    la $a0, comma
    syscall

skip_comma1:
    addi $s0, $s0, 4       # Next element
    addi $t0, $t0, 1       # Increment counter
```

```
    j print_input

print_input_done:
    li $v0, 4
    la $a0, close_bracket
    syscall

    # Calculate derivative
    la $s0, array          # Reset to beginning of array
    la $s1, derivative     # Base address of derivative array
    li $t0, 0              # Counter

calc_derivative:
    beq $t0, 9, calc_done  # Only 9 derivatives for 10 elements

    # Load array[i] and array[i+1]
    lw $t1, 0($s0)         # array[i]
    lw $t2, 4($s0)         # array[i+1]

    # Calculate derivative[i] = array[i+1] - array[i]
    sub $t3, $t2, $t1

    # Store result
    sw $t3, 0($s1)

    # Move to next elements
    addi $s0, $s0, 4
    addi $s1, $s1, 4
    addi $t0, $t0, 1

    j calc_derivative

calc_done:
    # Print output array
    li $v0, 4
    la $a0, output_msg
    syscall
```

```
    la $s1, derivative     # Reset to beginning of derivative array
    li $t0, 0              # Counter

print_output:
    beq $t0, 9, print_output_done

    # Print number
    lw $a0, 0($s1)
    li $v0, 1
    syscall

    # Print comma if not last element
    beq $t0, 8, skip_comma2
    li $v0, 4
    la $a0, comma
    syscall

skip_comma2:
    addi $s1, $s1, 4       # Next element
    addi $t0, $t0, 1       # Increment counter
    j print_output

print_output_done:
    li $v0, 4
    la $a0, close_bracket
    syscall

    # Exit
    li $v0, 10
    syscall
```

Explanation: This program creates a derivative array by calculating the difference between consecutive elements in the original array.

Lab Sheet 4: Task 1 - Square Root and Sin Functions

Problem Statement: Implement functions to calculate square root of a positive real number and sine of an angle (convert degrees to radians first).

Solution:

```
.data
prompt_sqrt:    .asciiz "Enter a positive number for square root: "
prompt_angle:   .asciiz "Enter an angle in degrees: "
sqrt_msg:       .asciiz "Square root: "
sin_msg:        .asciiz "Sine: "
newline:        .asciiz "\n"
pi:             .float 3.14159265
deg_to_rad:     .float 0.017453292    # pi/180

.text
main:
    # Square root calculation
    li $v0, 4
    la $a0, prompt_sqrt
    syscall

    # Read float
    li $v0, 6
    syscall
    mov.s $f12, $f0         # Move input to $f12

    jal sqrt_func

    # Print square root result
    li $v0, 4
    la $a0, sqrt_msg
    syscall

    li $v0, 2
    mov.s $f12, $f0         # Result is in $f0
    syscall

    li $v0, 4
    la $a0, newline
    syscall
```

```
    # Sine calculation
    li $v0, 4
    la $a0, prompt_angle
    syscall

    # Read angle in degrees
    li $v0, 6
    syscall
    mov.s $f12, $f0        # Move input to $f12

    jal sin_func

    # Print sine result
    li $v0, 4
    la $a0, sin_msg
    syscall

    li $v0, 2
    mov.s $f12, $f0        # Result is in $f0
    syscall

    li $v0, 4
    la $a0, newline
    syscall

    # Exit
    li $v0, 10
    syscall

# Square root function using Newton's method
sqrt_func:
    # Input: $f12 = number
    # Output: $f0 = square root

    li.s $f0, 1.0          # Initial guess
    li.s $f2, 0.000001     # Tolerance

sqrt_loop:
```

```
    # new_guess = 0.5 * (guess + number/guess)
    div.s $f4, $f12, $f0    # number/guess
    add.s $f4, $f4, $f0     # guess + number/guess
    li.s $f6, 0.5
    mul.s $f4, $f4, $f6     # new_guess = 0.5 * (guess + number/guess)

    # Check convergence: |new_guess - guess| < tolerance
    sub.s $f8, $f4, $f0     # difference
    abs.s $f8, $f8          # absolute difference

    c.lt.s $f8, $f2         # Compare with tolerance
    bc1t sqrt_done

    mov.s $f0, $f4          # Update guess
    j sqrt_loop

sqrt_done:
    jr $ra

# Sine function (converts degrees to radians first)
sin_func:
    # Input: $f12 = angle in degrees
    # Output: $f0 = sine value

    # Convert degrees to radians
    l.s $f2, deg_to_rad
    mul.s $f12, $f12, $f2   # angle_rad = angle_deg * pi/180

    # Taylor series approximation: sin(x) ≈ x - x³/6 + x⁵/120
    mov.s $f0, $f12         # x
    mul.s $f2, $f12, $f12   # x²
    mul.s $f4, $f2, $f12    # x³

    li.s $f6, 6.0
    div.s $f4, $f4, $f6     # x³/6

    sub.s $f0, $f0, $f4     # x - x³/6
```

```
    jr $ra
```

Explanation: Implements Newton's method for square root and Taylor series approximation for sine function.

Lab Sheet 4: Task 2 - Combination and Permutation Functions

Problem Statement: Create nCr and nPr functions where nCr uses nPr internally.

Solution:

```
.data
prompt_n:      .asciiz "Enter n: "
prompt_r:      .asciiz "Enter r: "
ncr_msg:       .asciiz "nCr = "
npr_msg:       .asciiz "nPr = "
newline:       .asciiz "\n"

.text
main:
    # Get n
    li $v0, 4
    la $a0, prompt_n
    syscall

    li $v0, 5
    syscall
    move $s0, $v0          # Store n

    # Get r
    li $v0, 4
    la $a0, prompt_r
    syscall

    li $v0, 5
```

```
    syscall
    move $s1, $v0          # Store r

    # Calculate nPr
    move $a0, $s0          # n
    move $a1, $s1          # r
    jal npr_func
    move $s2, $v0          # Store nPr result

    # Print nPr
    li $v0, 4
    la $a0, npr_msg
    syscall

    li $v0, 1
    move $a0, $s2
    syscall

    li $v0, 4
    la $a0, newline
    syscall

    # Calculate nCr using nPr
    move $a0, $s0          # n
    move $a1, $s1          # r
    jal ncr_func
    move $s3, $v0          # Store nCr result

    # Print nCr
    li $v0, 4
    la $a0, ncr_msg
    syscall

    li $v0, 1
    move $a0, $s3
    syscall

    li $v0, 4
```

```
    la $a0, newline
    syscall

    # Exit
    li $v0, 10
    syscall

# nPr = n!/(n-r)!
npr_func:
    # Input: $a0 = n, $a1 = r
    # Output: $v0 = nPr

    addi $sp, $sp, -12
    sw $ra, 8($sp)
    sw $a0, 4($sp)          # Save n
    sw $a1, 0($sp)          # Save r

    # Calculate n!
    jal factorial
    move $t0, $v0           # Store n!

    # Calculate (n-r)!
    lw $a0, 4($sp)          # Restore n
    lw $a1, 0($sp)          # Restore r
    sub $a0, $a0, $a1       # n-r
    jal factorial
    move $t1, $v0           # Store (n-r)!

    # nPr = n! / (n-r)!
    div $t0, $t1
    mflo $v0                # Result in $v0

    lw $ra, 8($sp)
    addi $sp, $sp, 12
    jr $ra

# nCr = nPr / r!
ncr_func:
```

```
    # Input: $a0 = n, $a1 = r
    # Output: $v0 = nCr

    addi $sp, $sp, -12
    sw $ra, 8($sp)
    sw $a0, 4($sp)          # Save n
    sw $a1, 0($sp)          # Save r

    # Calculate nPr
    jal npr_func
    move $t0, $v0           # Store nPr

    # Calculate r!
    lw $a0, 0($sp)          # Restore r
    jal factorial
    move $t1, $v0           # Store r!

    # nCr = nPr / r!
    div $t0, $t1
    mflo $v0                # Result in $v0

    lw $ra, 8($sp)
    addi $sp, $sp, 12
    jr $ra

# Factorial function
factorial:
    # Input: $a0 = n
    # Output: $v0 = n!

    addi $sp, $sp, -8
    sw $ra, 4($sp)
    sw $a0, 0($sp)

    li $v0, 1               # Base case
    ble $a0, 1, fact_done

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

```
    jal factorial          # Recursive call

    lw $a0, 0($sp)         # Restore n
    mul $v0, $a0, $v0      # n * factorial(n-1)

fact_done:
    lw $ra, 4($sp)
    addi $sp, $sp, 8
    jr $ra
```

Explanation: This implements permutation and combination calculations using factorial functions, with nCr calling nPr internally.

Lab Sheet 5: Task 1 - Mean and Standard Deviation

Problem Statement: Ask user for count of numbers, dynamically allocate space, read numbers, and calculate mean and standard deviation.

Solution:

```
.data
prompt_count:   .asciiz "Enter the number of real numbers: "
prompt_num:     .asciiz "Enter number: "
mean_msg:       .asciiz "Mean: "
stddev_msg:     .asciiz "Standard Deviation: "
newline:        .asciiz "\n"
zero_float:     .float 0.0
one_float:      .float 1.0
two_float:      .float 2.0

.text
main:
    # Get count of numbers
    li $v0, 4
    la $a0, prompt_count
    syscall

    li $v0, 5
    syscall
```

```
    move $s0, $v0          # $s0 = count

    # Allocate memory (count * 4 bytes for floats)
    sll $a0, $s0, 2        # count * 4
    li $v0, 9              # syscall 9 (sbrk)
    syscall
    move $s1, $v0          # $s1 = base address of allocated memory

    # Read numbers
    move $t0, $s1          # Current address
    li $t1, 0              # Counter
    l.s $f0, zero_float    # Sum = 0.0

read_loop:
    beq $t1, $s0, calc_mean

    # Prompt for number
    li $v0, 4
    la $a0, prompt_num
    syscall

    # Read float
    li $v0, 6
    syscall

    # Store float in memory
    s.s $f0, 0($t0)

    # Add to sum
    l.s $f2, 0($t0)
    add.s $f0, $f0, $f2

    addi $t0, $t0, 4       # Next float position
    addi $t1, $t1, 1       # Increment counter
    j read_loop

calc_mean:
    # Calculate mean = sum / count
```

```
    mtc1 $s0, $f4           # Move count to float register
    cvt.s.w $f4, $f4        # Convert to float
    div.s $f6, $f0, $f4     # mean = sum / count

    # Print mean
    li $v0, 4
    la $a0, mean_msg
    syscall

    li $v0, 2
    mov.s $f12, $f6
    syscall

    li $v0, 4
    la $a0, newline
    syscall

    # Calculate standard deviation
    move $t0, $s1           # Reset to beginning
    li $t1, 0               # Counter
    l.s $f8, zero_float     # Sum of squares = 0.0

stddev_loop:
    beq $t1, $s0, calc_stddev

    # Load number
    l.s $f10, 0($t0)

    # Calculate (x - mean)²
    sub.s $f12, $f10, $f6   # x - mean
    mul.s $f12, $f12, $f12  # (x - mean)²

    # Add to sum of squares
    add.s $f8, $f8, $f12

    addi $t0, $t0, 4        # Next float
    addi $t1, $t1, 1        # Increment counter
    j stddev_loop
```

```
calc_stddev:
    # Standard deviation = sqrt(sum_of_squares / count)
    div.s $f14, $f8, $f4    # sum_of_squares / count
    sqrt.s $f16, $f14       # Square root

    # Print standard deviation
    li $v0, 4
    la $a0, stddev_msg
    syscall

    li $v0, 2
    mov.s $f12, $f16
    syscall

    li $v0, 4
    la $a0, newline
    syscall

    # Exit
    li $v0, 10
    syscall
```

Explanation: Uses dynamic memory allocation with syscall 9, reads floating-point numbers, and calculates statistical measures using floating-point arithmetic.

Lab Sheet 5: Task 2 - Linked List in Reverse

Problem Statement: Create a linked list by reading 10 numbers, inserting each at the head, then print the list (numbers will appear in reverse order).

Solution:

```
.data
prompt:        .asciiz "Enter number "
colon:         .asciiz ": "
result_msg:    .asciiz "Numbers in reverse order: "
space:         .asciiz " "
```

```
newline:       .asciiz "\n"

.text
main:
    li $s0, 0              # head = NULL (0)
    li $t0, 1              # Counter starts at 1

input_loop:
    bgt $t0, 10, print_list # If counter > 10, start printing

    # Print prompt
    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 1
    move $a0, $t0
    syscall

    li $v0, 4
    la $a0, colon
    syscall

    # Read number
    li $v0, 5
    syscall
    move $a0, $v0          # Number to be inserted
    move $a1, $s0          # Current head

    jal create_node
    move $s0, $v0          # Update head

    addi $t0, $t0, 1       # Increment counter
    j input_loop

print_list:
    # Print result message
    li $v0, 4
```

```
    la $a0, result_msg
    syscall

    move $t1, $s0          # Current = head

print_loop:
    beqz $t1, print_done    # If current == NULL, done

    # Print data
    lw $a0, 0($t1)         # Load data
    li $v0, 1
    syscall

    li $v0, 4
    la $a0, space
    syscall

    lw $t1, 4($t1)         # current = current→next
    j print_loop

print_done:
    li $v0, 4
    la $a0, newline
    syscall

    # Exit
    li $v0, 10
    syscall

# Function to create a new node and link it
create_node:
    # Input: $a0 = data, $a1 = current head
    # Output: $v0 = address of new node

    addi $sp, $sp, -8
    sw $a0, 4($sp)         # Save data
    sw $a1, 0($sp)         # Save current head
```

```
# Allocate memory for node (8 bytes: 4 for data, 4 for next)
li $a0, 8
li $v0, 9            # syscall 9 (sbrk)
syscall
move $t0, $v0        # $t0 = address of new node

# Initialize node
lw $a0, 4($sp)       # Restore data
sw $a0, 0($t0)       # new_node->data = data

lw $a1, 0($sp)       # Restore old head
sw $a1, 4($t0)       # new_node->next = old_head

move $v0, $t0        # Return new node address

addi $sp, $sp, 8
jr $ra
```

Explanation: Creates nodes dynamically and inserts them at the head of the list, resulting in reverse order when printed.

Key Learning Points:

1. **Dynamic Memory**: Use syscall 9 (sbrk) to allocate memory at runtime
2. **Floating Point**: Use `.float` directive and floating-point instructions (add.s, mul.s, etc.)
3. **Function Calls**: Proper use of stack for saving/restoring registers and return addresses
4. **Data Structures**: Implementation of linked lists with dynamic allocation
5. **Mathematical Algorithms**: Newton's method, Taylor series, statistical calculations

Each solution demonstrates proper MIPS programming practices including register conventions, memory management, and structured programming with functions.