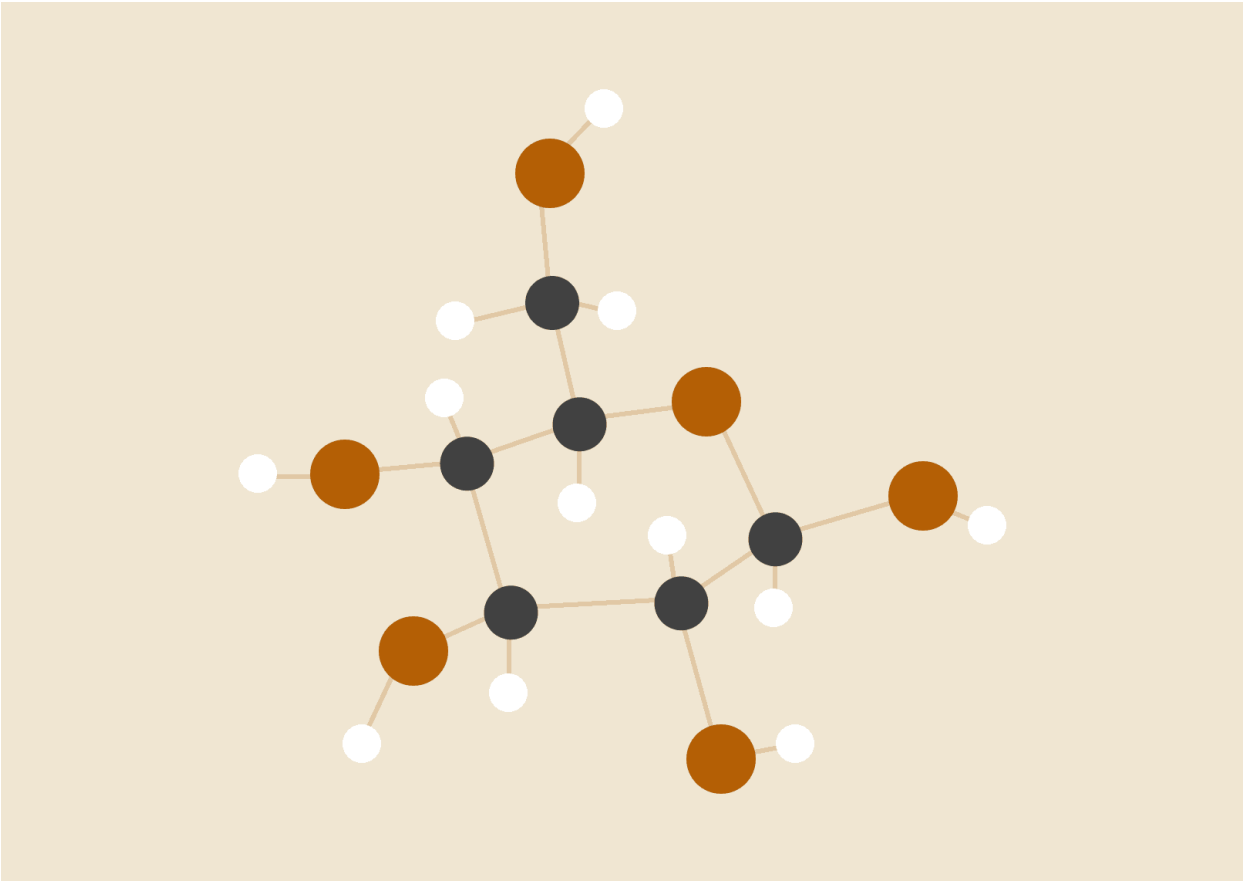
ASSIGNMENT 4

COMPUTER ARCHITECTURE



Yogita Mundankar

B22CS068

TASK 1: Catalan Number

Objective:

The objective of this MIPS assembly program is to compute the **nth Catalan number** using dynamic programming with a tabulation approach. Catalan numbers appear in combinatorics and are used to count various structures, such as the number of binary trees with n nodes, valid sequences of parentheses, and paths in grids. The program takes an integer input n from the user and computes the nth Catalan number, displaying the result.

Explanation of the Program:

1. Input Handling:

The program starts by displaying a prompt asking the user to input the value of n, which is the index of the Catalan number to compute. The user's input is then stored in a register. MIPS system calls are used for input-output operations:

- **Syscall 4**: Used to display a string to the user, such as the prompt message.
- **Syscall 5**: Used to read an integer input from the user, which represents n.

2. Memory Allocation for Dynamic Programming:

To compute the Catalan numbers efficiently, the program uses a **memoization array**. This array will store previously computed Catalan numbers to avoid redundant calculations, reducing the time complexity from exponential to quadratic.

The array is dynamically allocated based on the value of n. The size of the array is n+1, and each entry in the array is 8 bytes (since we use double-word storage to hold the Catalan numbers). The `sbrk` system call is used to allocate memory for the array.

The array is initialized as follows:

- **C[0]** is set to 1, as the 0th Catalan number is 1.
- **C[1]** to **C[n]** are initialized to 0, preparing the array for further computations.

3. Computing Catalan Numbers:

The program then computes the Catalan numbers using a **dynamic programming approach**. The formula for the nth Catalan number is:

$$C_n = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{i=0}^{n-1} C_i \times C_{n-1-i} & \text{if } n > 0 \end{cases}$$

The computation is done using two nested loops:

- **Outer loop**: Iterates through each Catalan number from C1 to Cn, updating the array as each Catalan number is computed.
- **Inner loop**: For each Ci, it iterates through all possible pairs of indices (j,i−j−1) to compute Cj×Ci−j−1 , adding this value to the current Ci.

Each entry is computed as the sum of products of previously computed Catalan numbers, leveraging the memoization array to store intermediate results.

4. Challenges in Computation:

One of the key challenges is correctly managing the memory offsets for accessing the memoization array, since each Catalan number entry is 8 bytes. The program uses bitwise operations (specifically, left shifting) to calculate the correct memory addresses for each entry in the memo array.

Another challenge is ensuring that the base cases, particularly C0, are handled correctly, as this forms the foundation for all subsequent computations.

5. Result Output:

Once the nth Catalan number is computed, the result is displayed to the user. The program retrieves the computed value from the memoization array and prints it using the appropriate MIPS system calls:

- **Syscall 1**: Used to print an integer value (the nth Catalan number).
- **Syscall 4**: Used to print a string before and after the result (i.e., the result message and a new line).

6. Program Termination:

After printing the result, the program exits cleanly using **Syscall 10** to terminate execution.

Challenges Faced:

1. **Dynamic Memory Allocation**: Managing dynamic memory allocation using the `sbrk` syscall in MIPS can be tricky. Ensuring that the memoization array is correctly sized and accessed through memory offsets required careful attention.
2. **Efficient Loop Handling**: The inner and outer loops for calculating the Catalan number needed to be structured to avoid redundant calculations. This was achieved by using a double-nested loop and ensuring the indices were properly calculated for Cj×Ci−j−1.
3. **Factorial and Recursive Dependencies**: Catalan numbers depend on previously computed values, so maintaining an efficient memoization scheme was key to avoiding a recursive computation that could lead to exponential time complexity.

Summary of the Output:

The program successfully outputs the nth Catalan number, given the user's input. For instance, if the user inputs n=4, the program will compute and display C4=14, which is the fourth Catalan number. The program achieves this using dynamic programming, which ensures that intermediate results are reused efficiently.

Reflection:

Through this assignment, I learned how to implement dynamic programming techniques in MIPS, specifically using memoization to optimize recursive algorithms. Understanding memory allocation and how to manage arrays in assembly language was a key takeaway from this task. It was also a good practice in managing input/output in MIPS, which involves manually handling system calls and registers.

Relevance of MIPS Programming:

MIPS programming helps in understanding low-level operations, memory management, and how high-level concepts like recursion and dynamic programming can be implemented in an assembly language. In real-world scenarios, MIPS programming is relevant for system programming, embedded systems, and hardware-level optimizations,

where understanding how to efficiently use resources like memory and CPU cycles is critical.

TASK 2: Taylor Series

Objective:

This MIPS program approximates e^x using the Taylor series expansion up to n terms, where e^x is defined as:

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

The program takes a floating-point number x and an integer n as input from the user. It then calculates the sum of the first n terms of the series to approximate e^x, outputting the result as a floating-point number.

Program Breakdown:

1. **Input Handling**:
    - The program begins by prompting the user for two inputs:
        - A floating-point number x, representing the exponent.
        - An integer n, representing the number of terms in the Taylor series expansion.
    - To handle these inputs:
        - The floating-point value x is read using the `syscall` code 6 (for floating-point input) and stored in the floating-point register `$f12`.
        - The integer n is read using the `syscall` code 5 (for integer input) and stored in the register `$t0`.
2. **Initialization**:
    - Before calculating the terms of the Taylor series, the program initializes the sum with the value 1.0. This is because the first term of the Taylor series, e^x, is always equal to 1.
    - The variable k (stored in `$t1`) is initialized to 1, which is used to track the

current term being calculated in the loop.
    - The sum of the series is stored in the floating-point register `$f2`, which starts at 1.0.
3. **Taylor Series Calculation Loop**:
    - The program enters a loop that iterates k from 1 to n. For each iteration:
        - The program calls a subroutine `compute_term` that calculates the k-th term of the Taylor series $\frac{x^k}{k!}$.
        - The result of the subroutine is added to the sum stored in `$f2`.
    - The loop continues until k exceeds n.
4. **Subroutine (compute_term)**: The subroutine `compute_term` is responsible for calculating the k-th term of the Taylor series. The term is calculated as $\frac{x^k}{k!}$, and the steps are as follows:
    - **Power Calculation (x^k)**:
        - The subroutine first calculates **x^k** by multiplying x with itself k times using a loop. This value is stored in `$f6`, which initially holds 1.0 (for the base case of x0x^0x0).
        - The loop decreases k (stored in `$t3`) until it reaches zero, each time multiplying the result by x.
    - **Factorial Calculation (k!k!k!)**:
        - After computing the power, the subroutine calculates k!(the factorial of k). The factorial is computed by multiplying all integers from 1 to k.
        - The loop for calculating the factorial uses the integer k (stored in `$t3`) and multiplies the running factorial (stored in `$f4`).
    - **Final Term Calculation**:
        - Once both x^k,and k! are calculated, the subroutine divides the power by the factorial to compute the final term $\frac{x^k}{k!}$.
        - The result is stored in `$f0` and returned to the main program, where it is added to the running sum.
5. **Output**:
    - After the loop finishes and all terms up to n have been calculated, the program outputs the final sum, which is the approximation of e^x.
    - The output is printed using the `syscall` for printing floating-point numbers (syscall code 2), with the result stored in `$f12`.

6. **Exit**:
    - Finally, the program terminates by invoking the exit syscall (code 10).

Challenges Faced:

1. **Floating-Point Arithmetic**:
    - MIPS assembly requires explicit handling of floating-point numbers, which is different from handling integers. In particular, the need to convert between integers and floating-point values was a challenge. To address this, I used the `cvt.s.w` instruction to convert integers to floating-point values, ensuring proper computation during power and factorial calculations.
2. **Handling Loops for Power and Factorial**:
    - Proper loop control was essential to ensure that both power and factorial calculations were correctly computed without errors. Mistakes in loop boundaries or in register usage could lead to infinite loops or incorrect results. Careful debugging and attention to register management solved these issues.
3. **Subroutine Handling**:
    - Passing parameters (like k and x) between the main program and subroutine required careful handling. Registers had to be properly saved and restored to avoid overwriting important data.

Program Output:

For an input x=2.0x = 2.0x=2.0 and n=3n = 3n=3, the program outputs an approximation of e^x as:

e2≈3.6402082

This output is consistent with the expected result of calculating the Taylor series up to 3 terms for e^2.

Learning and Reflection:

Working on this assignment significantly improved my understanding of MIPS assembly, especially in relation to:

1. **Floating-Point Operations**: I learned how MIPS handles floating-point numbers separately from integers and how to use specific instructions like `mul.s`, `add.s`,

and `div.s` for floating-point arithmetic. This distinction between integer and floating-point operations is crucial in low-level programming.

2. **Subroutine Design**: This assignment reinforced the importance of modular design in assembly programming. By creating a subroutine to compute each term, the code became more organized and easier to debug. Subroutines also improve code reusability and readability, which are important principles in both high-level and low-level programming.
3. **Real-World Relevance of MIPS**: While MIPS assembly is primarily used for educational purposes and in specific embedded systems, understanding how to write efficient assembly code is valuable for optimizing performance in resource-constrained environments. For example, knowing how to minimize memory usage and reduce instruction cycles can be critical in embedded systems and real-time applications. Additionally, learning MIPS helps bridge the gap between high-level programming languages and the underlying hardware, providing deeper insights into how processors execute instructions.