Computer Architecture Project

MIPS Assembly

Factorial, A^B, Exponent check

Snehal Sharma IMT2023572
Mithilesh IMT2023507
DT Mohan Krishna IMT2023609

1. Factorial

The provided MIPS assembly code iteratively calculates the factorial of a number. The code initializes registers, stores and loads values in memory, and employs a loop to perform multiplication and decrement operations until reaching the specified 'end' label. This assembly program effectively computes the factorial of the initial value stored in $s0, demonstrating a straightforward iterative algorithm for factorial calculation.

Code

C++

```cpp
#include <iostream>
using namespace std;

int main() {
    int n = 5;
    int ans = 1;
    while(n != 1){
        ans = ans*n;
        n--;
    }
    cout<<ans<<endl;
}
```

MIPS ASSEMBLY

```
#$s1 has result
addi $t4,$0, 5 #a
addi $t6,$0, 0
sw $t4, 0($t6)
lw $s0, 0($t6)
addi $s1, $0, 1
beq $s0, 0, end
loop1: mul $s1, $s0, $s1
addi $s0, $s0, -1
bne $s0, 0, loop1
end:
```

RESULT

Instruction Memory:-

```
['00100000', '00001100', '00000000', '00000101', '00100000', '00001110',
'00000000', '00000000', '10101101', '11001100', '00000000', '00000000',
'10001101', '11010000', '00000000', '00000000', '00100000', '00010001',
'00000000', '00000001', '00100000', '00000001', '00000000', '00000000',
'00010000', '00110000', '00000000', '00000100', '01110010', '00010001',
'10001000', '00000010', '00100010', '00010000', '11111111', '11111111',
'00100000', '00000001', '00000000', '00000000', '00010100', '00110000',
'11111111', '11111100']
```

Register file:-

```
{'00000': '00000000000000000000000000000000',
'00001': '00000000000000000000000000000000',
'00010': '00000000000000000000000000000000',
'00011': '00000000000000000000000000000000',
'00100': '00000000000000000000000000000000',
'00101': '00000000000000000000000000000000',
'00110': '00000000000000000000000000000000',
'00111': '00000000000000000000000000000000',
'01000': '00000000000000000000000000000000',
'01001': '00000000000000000000000000000000',
'01010': '00000000000000000000000000000000',
'01011': '00000000000000000000000000000000',
'01100': '00000000000000000000000000000101',
'01101': '00000000000000000000000000000000',
'01110': '00000000000000000000000000000000',
'01111': '00000000000000000000000000000000',
'10000': '00000000000000000000000000000000',
'10001': '00000000000000000000000001111000',
'10010': '00000000000000000000000000000000',
'10011': '00000000000000000000000000000000',
'10100': '00000000000000000000000000000000',
'10101': '00000000000000000000000000000000',
'10110': '00000000000000000000000000000000',
'10111': '00000000000000000000000000000000',
'11000': '00000000000000000000000000000000',
'11001': '00000000000000000000000000000000',
'11010': '00000000000000000000000000000000',
'11011': '00000000000000000000000000000000',
```

'11100': '00000000000000000000000000000000',
'11101': '00000000000000000000000000000000',
'11110': '00000000000000000000000000000000',
'11111': '00000000000000000000000000000000',
'hi': '00000000000000000000000000000000',
'lo': '00000000000000000000000001111000'}

Output:-

120

2. A ^ B

The provided C code recursively calculates the power of a number using a basic exponentiation approach. The power function takes two parameters, base (N) and exponent (P), and computes N^P. The code demonstrates a simple recursive algorithm for exponentiation in MIPS.

CODE

C++

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 0;
    int b = 3;
    int ans = 1;

    while(b >= 1){
        ans = ans*a;
        b--;
    }
    cout<<ans<<endl;
}
```

MIPS ASSEMBLY

```
# a^b: $s0 has a, $s1 has b, $s2 has result

addi $t4,$0, 4 #a
addi $t5,$0, 2 #b
addi $t6,$0, 0
sw $t4, 0($t6)
lw $s0, 0($t6)
sw $t5, 0($t6)
lw $s1, 0($t6)
addi $s2, $0, 1
beq $s1, 0, end
loop1:
mul $s2, $s0, $s2
addi $s1, $s1, -1
bne $s1, 0, loop1
end:
```

RESULT

Instruction Memory:-

```
['00100000', '00001100', '00000000', '00000100', '00100000', '00001101',
'00000000', '00000010', '00100000', '00001110', '00000000', '00000000',
'10101101', '11001100', '00000000', '00000000', '10001101', '11010000',
'00000000', '00000000', '10101101', '11001101', '00000000', '00000000',
'10001101', '11010001', '00000000', '00000000', '00100000', '00010010',
'00000000', '00000001', '00100000', '00000001', '00000000', '00000000',
'00010000', '00110001', '00000000', '00000100', '01110010', '00010010',
'10010000', '00000010', '00100010', '00110001', '11111111', '11111111',
'00100000', '00000001', '00000000', '00000000', '00010100', '00110001',
'11111111', '11111100']
```

Register file:-

```
{'00000': '00000000000000000000000000000000',
'00001': '00000000000000000000000000000000',
'00010': '00000000000000000000000000000000',
'00011': '00000000000000000000000000000000',
'00100': '00000000000000000000000000000000',
'00101': '00000000000000000000000000000000',
'00110': '00000000000000000000000000000000',
'00111': '00000000000000000000000000000000',
'01000': '00000000000000000000000000000000',
'01001': '00000000000000000000000000000000',
'01010': '00000000000000000000000000000000',
'01011': '00000000000000000000000000000000',
'01100': '00000000000000000000000000000100',
'01101': '00000000000000000000000000000010',
'01110': '00000000000000000000000000000000',
'01111': '00000000000000000000000000000000',
'10000': '00000000000000000000000000000100',
'10001': '00000000000000000000000000000000',
'10010': '00000000000000000000000000010000',
'10011': '00000000000000000000000000000000',
'10100': '00000000000000000000000000000000',
'10101': '00000000000000000000000000000000',
'10110': '00000000000000000000000000000000',
'10111': '00000000000000000000000000000000',
'11000': '00000000000000000000000000000000',
'11001': '00000000000000000000000000000000',
'11010': '00000000000000000000000000000000',
'11011': '00000000000000000000000000000000',
'11100': '00000000000000000000000000000000',
'11101': '00000000000000000000000000000000',
```

```
'11110': '00000000000000000000000000000000',
'11111': '00000000000000000000000000000000',
'hi': '00000000000000000000000000000000',
'lo': '00000000000000000000000000010000'}
```

Output:-

16

3.If A is Power of B

This assembly program effectively calculates the result of raising B to the power of X using an iterative approach. It initializes registers, performs memory operations, and utilizes a loop to repeatedly multiply a variable until a specified condition is met. The final result is stored in $s7.

CODE

C++

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 1;
    int b = 3;
    int c = 1;
    int ans = 0;
    while(c < a){
        c = c*b;
    }
    if(c == a || a == 1){
        ans++;
    }
    cout<<ans<<endl;
}
```

MIPS ASSEMBLY

```
#$s7 has result
addi $t4,$0 81 #a
addi $t5,$0, 3 #b
addi $t6,$0, 0
sw $t4, 0($t6)
lw $t0, 0($t6)
sw $t5, 0($t6)
lw $t1, 0($t6)
addi $t2,$0, 1
beq $t0, $t2, B1
START:mul $t2, $t2, $t1
bgt  $t2, $t0, B2
beq $t2, $t0, B1
j START
B1: addi $t3,$0,1
j END
B2:addi $t3,$0, 0
j END
END: add  $s7,$0, $t3
```

<u>RESULT</u>

Instruction Memory:-

```
['00100000', '00001100', '00000000', '01010001', '00100000', '00001101',
'00000000', '00000011', '00100000', '00001110', '00000000', '00000000',
'10101101', '11001100', '00000000', '00000000', '10001101', '11001000',
'00000000', '00000000', '10101101', '11001101', '00000000', '00000000',
'10001101', '11001001', '00000000', '00000000', '00100000', '00001010',
'00000000', '00000001', '00010001', '00001010', '00000000', '00000101',
'01110001', '01001001', '01010000', '00000010', '00000001', '00001010',
'00001000', '00101010', '00010100', '00100000', '00000000', '00000100',
'00010001', '01001000', '00000000', '00000001', '00001000', '00000000',
'00001100', '00001001', '00100000', '00001011', '00000000', '00000001',
'00001000', '00000000', '00001100', '00010010', '00100000', '00001011',
'00000000', '00000000', '00001000', '00000000', '00001100', '00010010',
'00000000', '00001011', '10111000', '00100000']
```

Register file:-

```
{'00000': '00000000000000000000000000000000',
'00001': '00000000000000000000000000000000',
'00010': '00000000000000000000000000000000',
'00011': '00000000000000000000000000000000',
'00100': '00000000000000000000000000000000',
'00101': '00000000000000000000000000000000',
'00110': '00000000000000000000000000000000',
'00111': '00000000000000000000000000000000',
'01000': '00000000000000000000000001010001',
'01001': '00000000000000000000000000000011',
'01010': '00000000000000000000000001010001',
'01011': '00000000000000000000000000000001',
'01100': '00000000000000000000000001010001',
'01101': '00000000000000000000000000000011',
'01110': '00000000000000000000000000000000',
'01111': '00000000000000000000000000000000',
'10000': '00000000000000000000000000000000',
'10001': '00000000000000000000000000000000',
'10010': '00000000000000000000000000000000',
'10011': '00000000000000000000000000000000',
'10100': '00000000000000000000000000000000',
'10101': '00000000000000000000000000000000',
'10110': '00000000000000000000000000000000',
'10111': '00000000000000000000000000000001',
'11000': '00000000000000000000000000000000',
'11001': '00000000000000000000000000000000',
```

```
'11010': '00000000000000000000000000000000',
'11011': '00000000000000000000000000000000',
'11100': '00000000000000000000000000000000',
'11101': '00000000000000000000000000000000',
'11110': '00000000000000000000000000000000',
'11111': '00000000000000000000000000000000',
'hi': '00000000000000000000000000000000',
'lo': '00000000000000000000000001010001'}
Output:-
1
```

Processor:

1. Memory Initialization:
   - `DataMem` is initialized as a byte addressable list representing the data memory of the MIPS processor.

2. Reading Binary File:
   - Reads a binary file containing machine code instructions and converts it into a string representation (`textString`).
   - Divides the binary string into 8-bit segments, creating the byte addressable instruction memory (`InstMem`).

3. 2's Complement Conversion Functions:
   - `int_`: Converts a binary string to a signed integer.
   - `bin_`: Converts an integer to a binary string of specified length.

4. Register Memory Class (`Reg`):
   - Represents the register file with methods for reading, writing, and printing registers.
   - `Read1` and `Read2`: Read values from specified registers.
   - `write`: Writes data to a register.
   - `print`: Displays the content of registers.

5. Register Dictionary (`regs`):
   - Contains initial values for MIPS registers and special registers like `hi` and `lo`.

6. Sign Extend Function:
   - Extends a 16-bit immediate value to 32 bits.

7. Control Signal Function (`control_signal`):
   - Determines control signals based on the opcode and function code of the instruction.

8. ALU (Arithmetic Logic Unit) Function (`ALU`):
   - Performs ALU operations based on the ALU control signals and function code.
   - Contains a sub-function `ALUCont` for ALU control signal generation.

9. Main Execution Loop:
   - The processor executes instructions in a loop until the program counter reaches the end of InstMem.
   - The stages include instruction fetch, decode, execute, memory access, and write-back.
   - The program counter is updated based on normal execution, branch, or jump conditions.

10. Printing Results:
    - Prints the instruction memory (`InstMem`) and the content of registers after execution.
    - Depending on the input file, it prints the result of specific computations (`ifaispowb`, `Factorial`, or `Power`).