Enhanced Processor Architecture for Efficient Computation of EXPONENTIATED SERIES

Computer Architecture Assignment-1: GROUP-71
Course Name: EG 212, Computer Architecture

ROLL NO: SIDDHARTH ANIL, IMT2023503
KRISH PATEL, IMT2023134
KRISH KATHIRIA, IMT2023045

26 January 2023

1 ABSTRACT

Our Python based IAS simulator, presents a customized processor design equipped with an integrated assembler to efficiently compute the sum of exponentiated series, specifically the expression $1^1 + 2^2 + 3^3 + ... + 18^18$ (according to the file we submitted). The processor's ability to handle intricate mathematical operations showcases its potential for applications in scientific computing, cryptography, and other domains that demand high-performance computation. In summary, this project unfolds a tailored processor architecture seamlessly integrated with an assembler, specializing in the efficient computation of exponentiated series.

2 INTRODUCTION

The IAS Simulation, developed in Python, comprises three essential components: Assembly Language, which furnishes instructions to the processor for the efficient computation of Exponentiated Series; Assembler, responsible for converting the assembly language into machine-level language (in binary form); and Processor, which dynamically alters the values stored in various registers based on the provided instructions.

2.1 ASSEMBLY LANGUAGE:

We have employed a set of 11 instructions to meet our goals, incorporating two recently created commands and making certain adjustments to the initial instructions.

Individual Functions of these instructions are:
1. LOAD M(X): Loads contents at address X to Accumulator(AC)
2. STOR M(X): Stores contents in AC at address X
3. ADD M(X): Adds contents at address X to the contents in AC and puts the result in AC
4. SUB M(X): Subtracts contents at address X from contents in AC and puts the result in AC
5. MUL M(X): Multiplies the contents at address X with contents in AC and puts the result in MQ
6. JUMP+ M(X,0:19): If number in AC is non-negative, take next instruction from left half of M(X)
7. LOADMQ M(X): Transfers contents stored at memory location X to MQ
8. LOADMQAC: Transfers contents stored in MQ to AC
9. NOP: These instructions are used to fill space in a program or code segment. This can be done to align instructions or data properly.
10. END: This instruction is used to end the program.

The two new instructions introduced are:

11. CMP M(X): This instruction first transfers the contents at address X to MBR and then **"COMPARES"** it with the data in AC. If value in MBR is greater than value in AC , then value in AC will be changed to -1.
12. RST M(X): This newly introduced instruction **"RESETS"** the value at memory location X to 1.

In addition to this we also introduced NONE:
It is used where no explicit memory location is needed, **"NONE"** can be used as a placeholder.

Note: The file which contains assembly language is **instructions.txt.**

2.2 ASSEMBLER:

The assembly language undergoes a transformation into machine-level language through the assembler's process. This involves the allocation of distinct opcodes to individual instructions, with these instructions being stored in diverse memory locations. Concurrently, diverse data utilized for the summation of n^n, where n spans from 1 to 18, finds its place in designated memory locations. Subsequently, both the opcodes and memory addresses undergo a conversion into binary form.
It's noteworthy that the opcodes employed in this process are of an 8-bit size, while the memory addresses and numerical values consist of 12 bits.

OPCODE MAPPING:-

OPCODE	FUNCTION
00100101	LOAD
00100110	STOR
00100111	ADD
00101000	SUB
00101001	MUL
00101010	CMP
00101011	RST
00101100	JUMP+
00101101	LOADMQ
00101110	LOADMQAC
11111111	END
11111110	NOP

Note: The file which acts as an assembler is **assembler.py**. This file reads from file **instruction.txt** and converts everything in binary form and prints the result in file **binary_code.txt**. .

2.3 PROCESSOR:

The IAS Computer employs a modular design with distinct classes representing various components. The central processing unit (CPU) orchestrates the execution of instructions through the interaction of key components such as:

- Program Counter(PC): Holds the next instruction's address.
- Memory Address Register (MAR): Stores 12-bit addresses.
- Instruction Register (IR): Stores the 8-bit opcode of the instruction to be executed.
- Memory Buffer Register (MBR): Contains a word to be read/stored in memory.
- Instruction Buffer Register (IBR): Holds the RHS instruction temporarily.
- Arithmetic Logic Unit (ALU): Performs arithmetic operations such as addition, subtraction, etc.
- Accumulator (AC): Temporarily stores a result of an ALU operation.
- Multiplier/Quotient (MQ): This register is used when ALU performs division or multiplication operation.

Note: The file which performs as processor is **processor.py**. It reads from file **binary_code.txt** and performs accordingly.

3 Exponentiated Series in IAS Assembly:

3.1 Original C Code:-

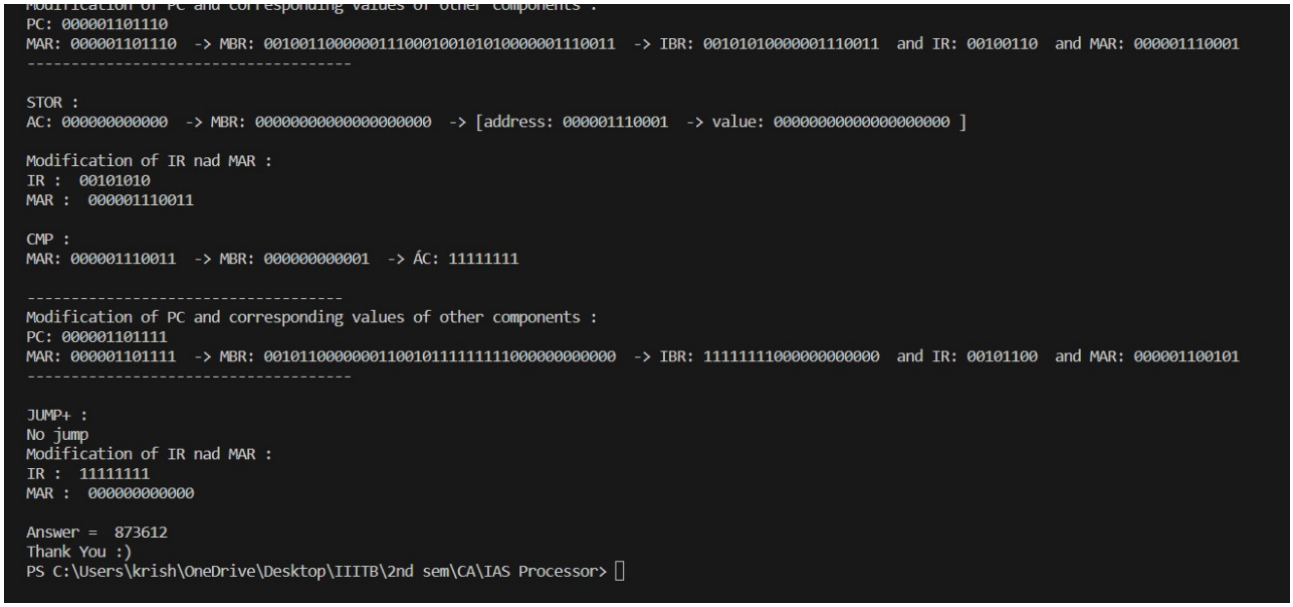
```c
#include<stdio.h>

int n_power_n(int n)
{
        int final=1;
        for(int i=0;i<n;i++)
        {
                final=final*n;
        }
        return final;
}

void main()
{
        printf("This a code for calculating summation of n^n \n");
        int input;
        printf("Enter the value of n: ");
        scanf("%d", &input);
        int res=0;

        for(int i=1;i<=input;i++)
        {
                res=res+ n_power_n(i);
        }
        printf("Summation of n^n is %d.\n", res);

}
```

3.2 To implement a simple C Exponentiated Series code in IAS assembly, we'll leverage the provided initialization and create an assembly code that corresponds to the C code. The following assembly code demonstrates the translation:

```
101 LOADMQ M(116) MUL M(112)
102 LOADMQAC NONE NOP NONE
103 STOR M(116) LOAD M(113)
104 SUB M(114) STOR M(113)
105 LOAD M(113) CMP M(115)
106 JUMP+ M(101,0:19) LOAD M(117)
107 ADD M(116) STOR M(117)
108 RST M(116) LOAD M(112)
109 SUB M(114) STOR M(112)
110 STOR M(113) CMP M(115)
111 JUMP+ M(101,0:19) END NONE
112 3
113 3
114 1
115 1
116 1
117 0
```

4 EXECUTION RESULTS:

The final output of our project (if n=7) is:


Modification of PC and corresponding values of other components :
PC: 000001101110
MAR: 000001101110 -> MBR: 0010011000000111000100101010000001110011 -> IBR: 00101010000001110011 and IR: 00100110 and MAR: 000001110001
STOR :
AC: 000000000000 -> MBR: 0000000000000000000 -> [address: 000001110001 -> value: 0000000000000000000]
Modification of IR nad MAR :
IR : 00101010
MAR : 000001110011
CMP :
MAR: 000001110011 -> MBR: 000000000001 -> AC: 11111111
Modification of PC and corresponding values of other components :
PC: 000001101111
MAR: 000001101111 -> MBR: 0010110000000110010111111111000000000000 -> IBR: 11111111000000000000 and IR: 00101100 and MAR: 000001100101
JUMP+ :
No jump
Modification of IR nad MAR :
IR : 11111111
MAR : 000000000000
Answer = 873612
Thank You :)
PS C:\Users\krish\OneDrive\Desktop\IIITB\2nd sem\CA\IAS Processor>