Project Report

Topic: **IAS processor design**
EG 212 Computer Architecture - Processor design

Members:
M S Dheeraj Murthy (IMT2023552)
Mathew Joseph (IMT2023008)
Priyanshu Pattnaik (IMT2023046)

Index

Assembler

The assembler is written in C++ language. It converts the inputted assembly language to binary output. It takes in the input of ten instructions of the IAS ISA along with some custom instructions like more, les and eq. It converts the assembly code to binary code by string comparison and outputs the binary as the return value of the main function.

The assembler and processor support 15 instructions. The ISA and their corresponding opcodes are as follows:

```
00000000: NOP              00000001: LOAD   ###
00000010: LOAD MQ###       00000011: STOR   ###
00000100: ADD    ###       00000101: SUB    ###
00000110: MUL    ###       00000111: DIV    ###
00001000: JUMP   ###       00001001: CJUMP  ###
00001010: EQUA   ###       00001011: LESS   ###
00001100: MORE   ###       00001101: DISP
00001110: END
```

The Assembler takes in code in lines that are 20 characters wide and contain 2 instructions each. The first 7 characters are reserved for the instruction name and the last 3 characters are for the value. A Sample instruction would be:

```
LOAD   051STOR   060
END       NOP
```

And this piece of code when converted to binary by our assembler will look like:

```
0000000100000011001100000011000000111100
0000111000000000000000000000000000000000
```

Processor:

The processor follows a classic IAS architecture. It starts with initialising the values of the Program Counter (PC), Accumulator (AC) and Multiplier-Quotient (MQ) to 0. Next, it takes in the byte stream of instructions as input and stores them in the main memory from index 0. It then proceeds to store some default values that are required for calculations by the system. These are:

Location 050: $(0000000000000000000000000000000000000000)_2 = (0)_{10}$

Location 051: $(0000000000000000000000000000000000000001)_2 = (1)_{10}$

Location 052: $(0000000000000000000000000000000000000010)_2 = (2)_{10}$

Location 053: $(0000000000000000000000000000000000000011)_2 = (3)_{10}$

Location 054: $(0000000000000000000000000000000000000100)_2 = (4)_{10}$

Location 055: $(0000000000000000000000000000000000000101)_2 = (5)_{10}$

Location 056: $(0000000000000000000000000000000000000110)_2 = (6)_{10}$

Location 057: $(0000000000000000000000000000000000000111)_2 = (7)_{10}$

Location 058: $(0000000000000000000000000000000000001000)_2 = (8)_{10}$

Location 059: $(0000000000000000000000000000000000001001)_2 = (9)_{10}$

Now, it enters a loop which runs till the value of PC does not match the size of the Main Memory. The Memory Buffer Register (MBR) of the system takes the form of a vector of integers and it is loaded with PC^{th} line from the Main Memory. Then, the Instruction Buffer Register (IBR) is used to store the right instruction from the MBR, while the opcode of the left instruction is loaded into the Instruction Register (IR) and the memory address attributed to the same is loaded into the Memory Address Register (MAR). Next, the opcode in the IR is decoded and the corresponding function is performed using data from the MAR and the Main Memory. After the left instruction has been executed, the IR and MAR are cleared and the right instruction stored in the IBR is split into opcode and memory address and sent to the IR and MAR, respectively. Now, the value of PC is incremented by one. Further, the new opcode is decoded and implemented as needed. This marks the end of the execution cycle, and the control of the code returns to the fetch cycle to continue traversing through the rest of the byte code.

Implemented Functions:

A brief description of the instructions we have implemented, and their functionalities are as follows:

1. `NOP`

   A function meant to be the code equivalent of 'pass'. It does not perform any operation.

2. `LOAD ###`

   The standard function which updates the value of the AC to match that of the memory address inputted.

3. `LOAD MQ###`

   The standard function which updates the value of the MQ to match that of the memory address inputted.

4. `STOR ###`

   The standard function which stores the value of the AC at the memory address inputted.

5. `ADD ###`

   The standard function which adds the value stored at the memory address inputted to the that in AC and stores it in the AC.

6. `SUB ###`

The standard function which subtracts the value stored at the memory address inputted from the that in AC and stores the remainder in the AC.

7. `MUL ###`

The standard function which updates the value of the AC to the product of the value in the MQ and that in the memory address inputted.

8. `DIV ###`

The standard function which divides the value in the Ac with that in the memory address inputted and stores the quotient in the MQ and the remainder in the AC.

9. `JUMP ###`

The standard function which changes the value of the PC to the memory address inputted.

10. `CJUMP ###`

A function which changes the value of the PC to the memory address inputted if the value in the AC is positive. Else, it performs no operation.

11. `EQUA ###`

A function which, if the value in the AC is equal to that in the memory address inputted is equal, stores 1 in the AC. Else, it stores 0 in the AC.

12. `LESS ###`

A function which, if the value in the AC is less than or equal to that in the memory address inputted is equal, stores 1 in the AC. Else, it stores 0 in the AC.

13. `MORE ###`

A function which, if the value in the AC is greater than equal to that in the memory address inputted is equal, stores 1 in the AC. Else, it stores 0 in the AC.

14. `DISP`

A function that prints out the value in the AC.

15. `END`

A function that immediately terminates the code by changing the value of the PC to 999.

Primality Check Program

Our team has used the created assembler and processor to implement a typical primality check function.

In terms of C++, the assembly code works like this:

```
bool isPrime(int n)
{
    if (n == 2 || n == 3)
        return true;

    if (n <= 1 || n % 2 == 0 || n % 3 == 0)
        return false;

    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0)
            return false;
    }

    return true;
}
```

Using our Code:

To use our program, the assembler and processor need not be run separately. Input, in the form of assembly code, needs to be fed into the processor only.

The last line of the inputted assembly code must be the number for which primality is to be checked (in decimal notation).

Given below is a screenshot of the code along with the input and output: