COMPUTER ARCHITECTURE AND ORGANISATION

PROJECT REPORT

Kartikeya Agarwal
2019UCO1692

Tismeet Singh
2019UCO1693

Nikhil Gupta
2019UCO1719

Daksh Gupta
2019UCO1669

COE-3

Prologue

This project is an insight in the intricate workings of a full-fledged working CPU. The whole project has mainly two parts, CPU design and explanation followed by Assembler's code for virtual fully functioning CPU. The project enabled us to work-up our theory knowledge into something of practical essence and of essential pragmatic importance.

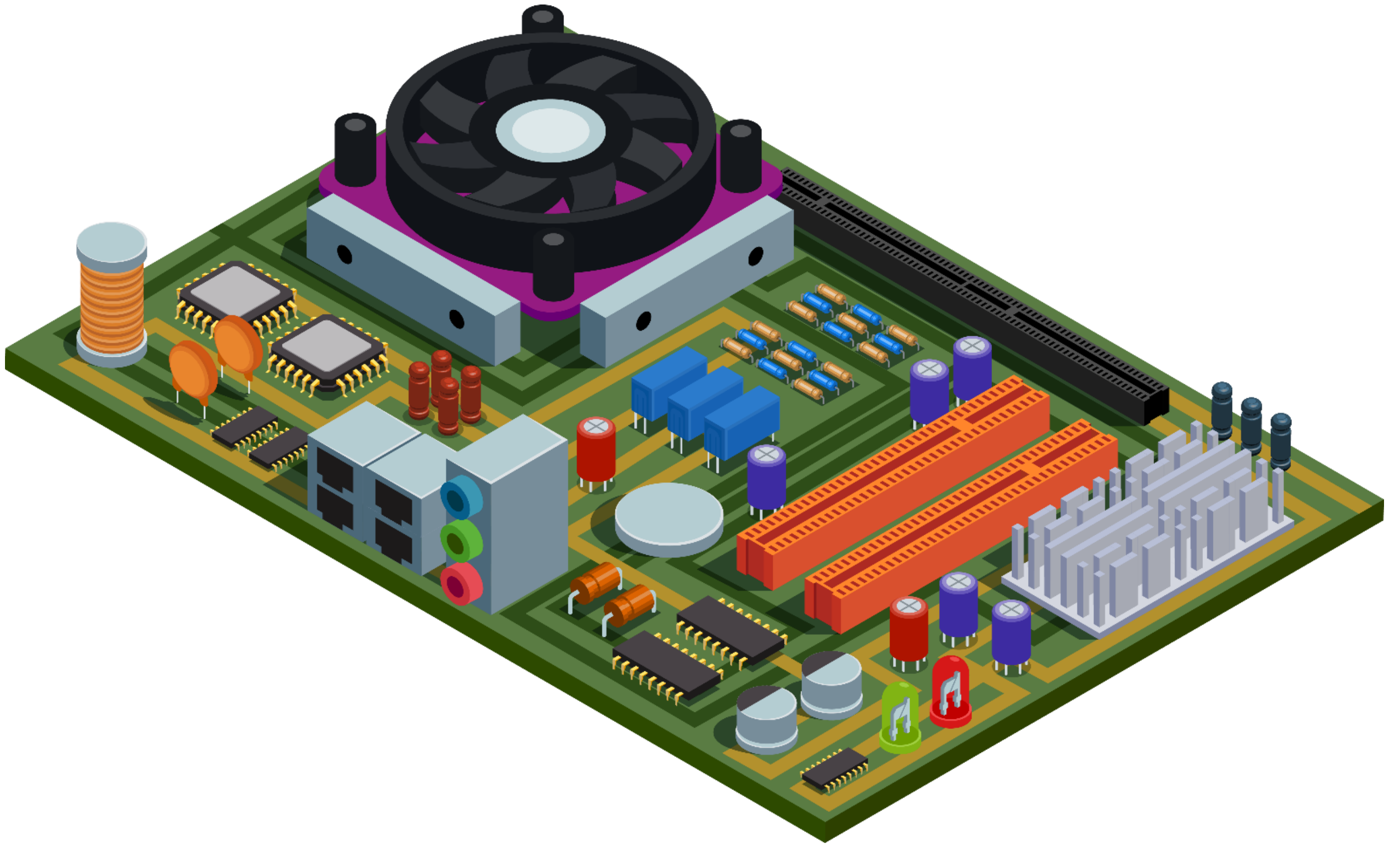
The CPU designed here is based on an 8bit 8086 architecture along with certain optimisations which we have mentioned in the coming sections. The assembler for the aforementioned CPU was written and executed using Python3. The assembler has 16 default instructions in the instruction set with all the possible errors handled. All the diagrams have been digitized from their rough versions, along with elaborate definitions and explanations on the concept of the complete design.

The working of the system was tested using a few input scenarios and the outputs were recorded and were found to match the expected outputs.

INTRODUCTION

What is a CPU?

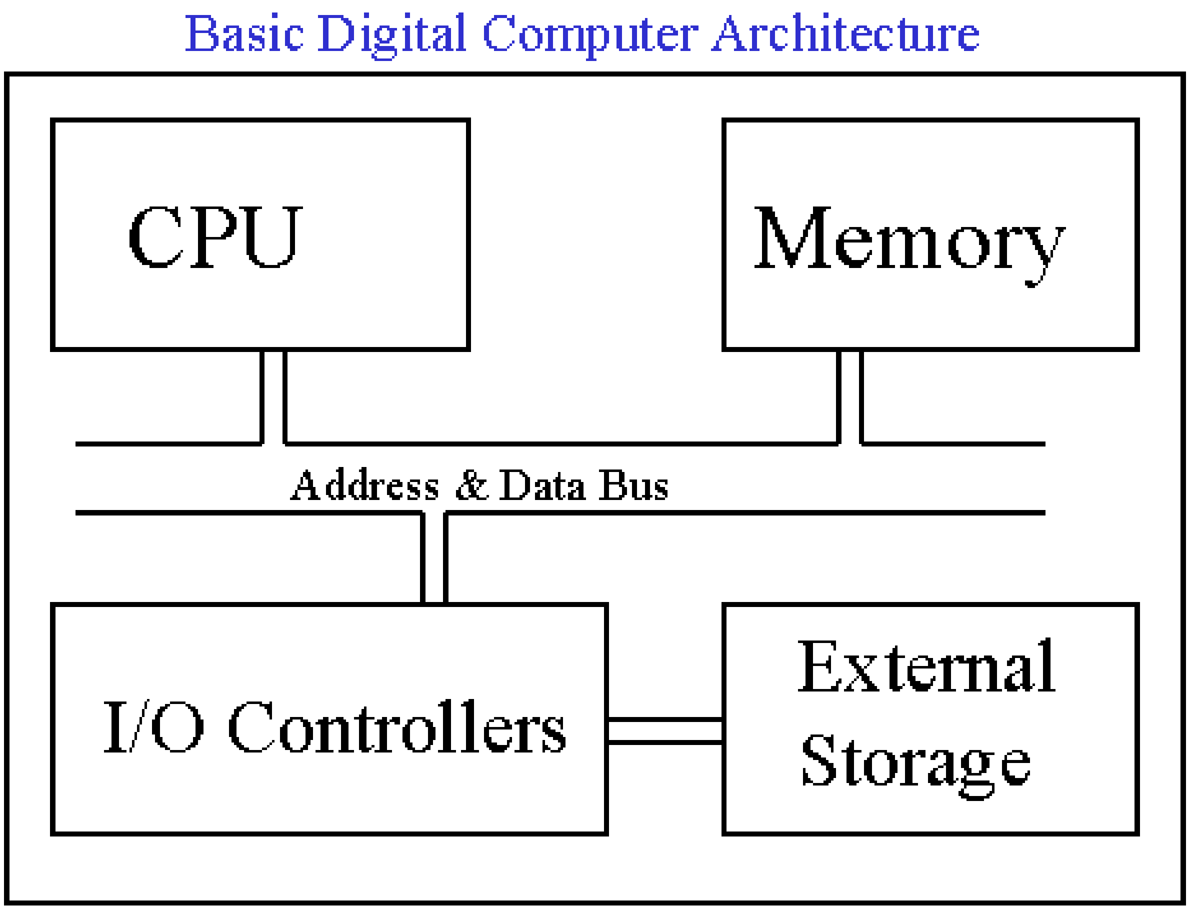
A central processing unit (CPU), also called a central processor, main processor or just processor, is the electronic circuitry within a computer that executes instructions that make up a computer program.



The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program. This contrasts with external components such as main memory and I/O circuitry, and specialized processors such as graphics processing units (GPUs).

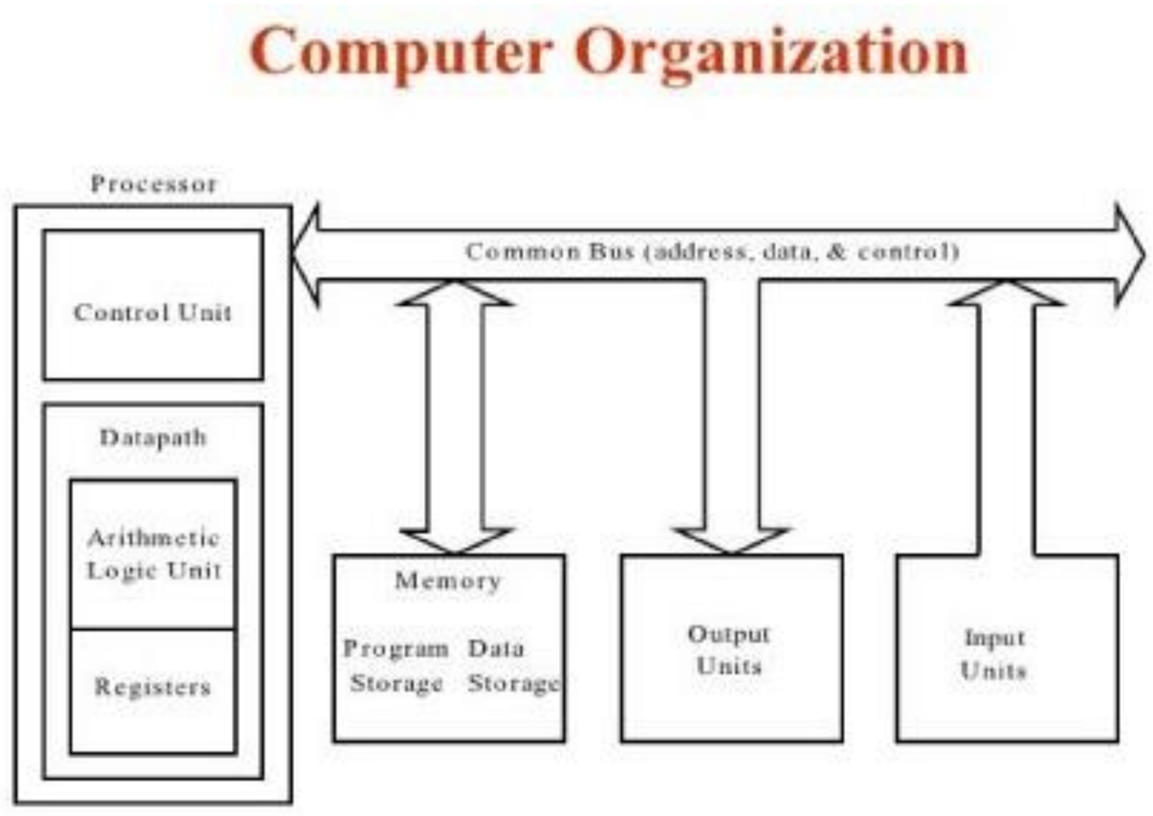
What is Computer Architecture?

Computer architecture is sometimes defined as the computer structure and behaviour as seen by the programmer that uses machine language instructions. This includes the instruction formats, addressing modes, the instruction set, and the general organization of the CPU registers.



What is Computer Organisation?

Organization of a computer system defines the way system is structured so that all those catalogued tools can be used. The significant components of Computer organization are ALU, CPU, memory and memory organization.



Components of a CPU

A typical CPU has three major components: (1) register set, (2) arithmetic logic unit (ALU), and (3) control unit (CU). The register set differs from one computer architecture to another. It is usually a combination of general-purpose and special purpose registers. General-purpose registers are used for any purpose, hence the name general purpose. Special purpose registers have specific functions within the CPU.

The CPU can be divided into a data section and a control section. The data section, which is also called the data path, contains the registers (known as the register file) and the ALU. The control section is basically the control unit, which issues control signals to the data path. The control unit of a computer is responsible for executing the program instructions, which are stored in the main memory.

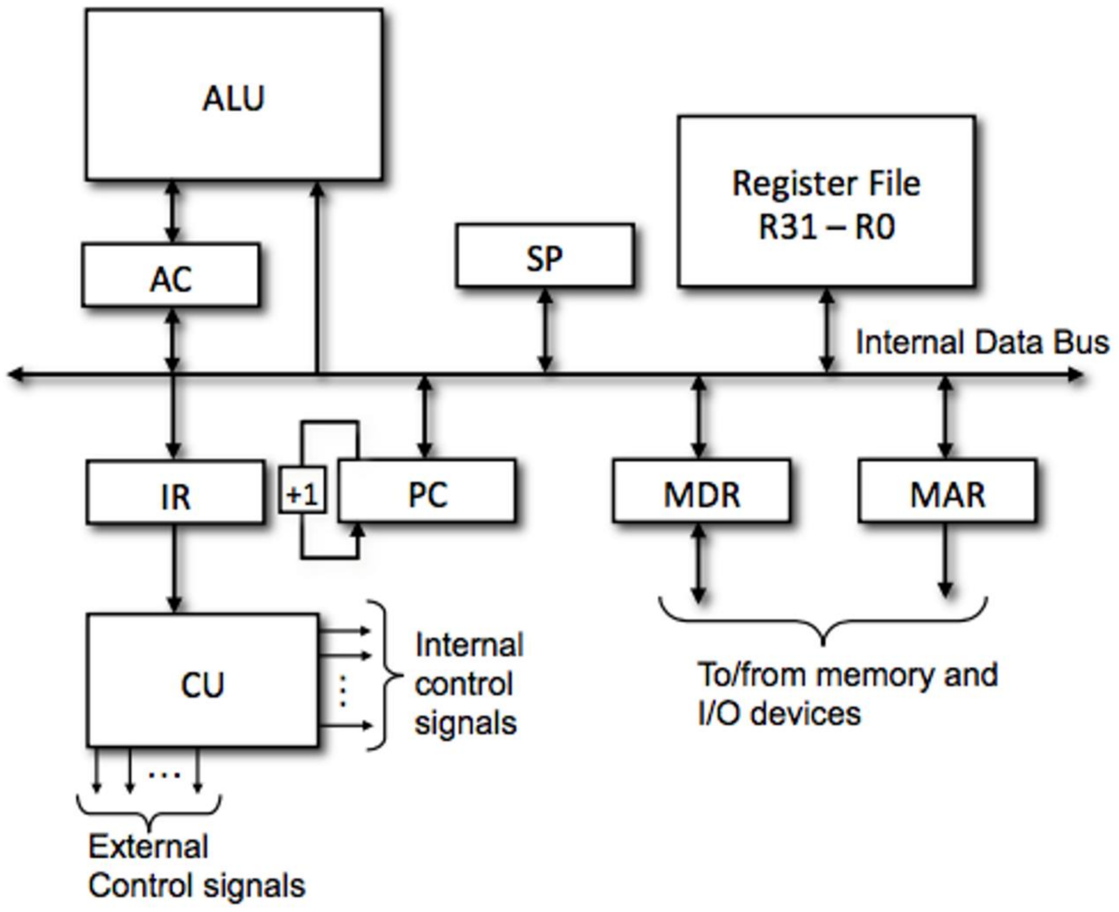
Different Registers

Accumulator:
This is the most frequently used register used to store data taken from memory. It is in different numbers in different microprocessors.

Memory Address Registers (MAR):
It holds the address of the location to be accessed from memory. MAR and MDR (Memory Data Register) together facilitate the communication of the CPU and the main memory.

Memory Data Registers (MDR):
It contains data to be written into or to be read out from the addressed location.

General Purpose Registers:
These are numbered as R0, R1, R2....Rn-1, and used to store temporary data during any ongoing operation. Its content can be accessed by assembly programming. Modern CPU architectures tends to use more GPR so that register-to-register addressing can be used more, which is comparatively faster than other addressing modes.



Program Counter (PC):
Program Counter (PC) is used to keep the track of execution of the program. It contains the memory address of the next instruction to be fetched. PC points to the address of the next instruction to be fetched from the main memory when the previous instruction has been successfully completed. Program Counter (PC) also functions to count the number of instructions. The incrementation of PC depends on the type of architecture being used. If we are using 32-bit architecture, the PC gets incremented by 4 every time to fetch the next instruction.

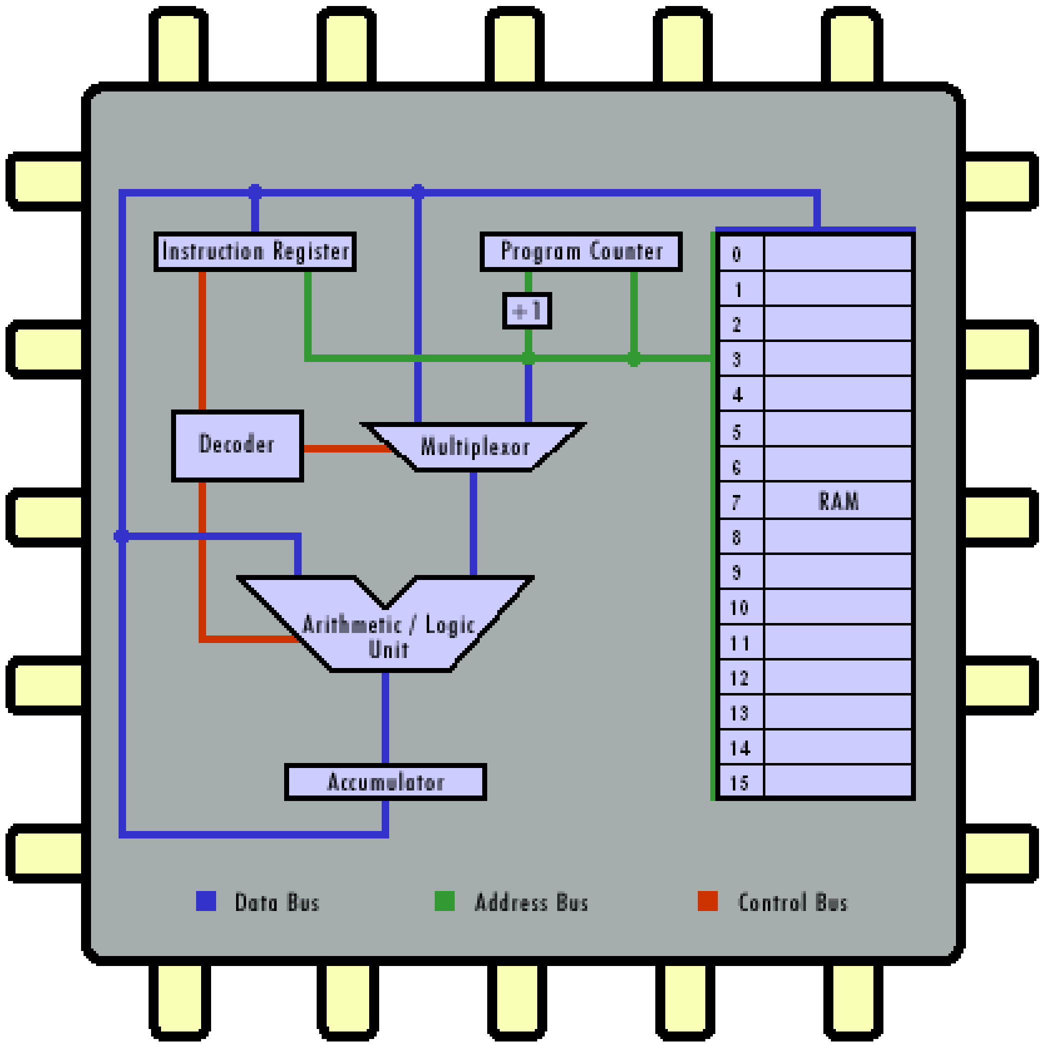
Instruction Register (IR):
The IR holds the instruction which is just about to be executed. The instruction from PC is fetched and stored in IR. As soon as the instruction in placed in IR, the CPU starts executing the instruction and the PC points to the next instruction to be executed.

Arithematic Logic Unit

The arithmetical logical unit is the combinational digital electronic circuit. It can perform arithmetic and bitwise operations on integer binary numbers. The ALU is the fundamental building block of many types of computing circuits. The ALU (Arithmetic Logical Unit) has the status of inputs, outputs, or both which convey the information about the previous operation or the current operation.

Control Unit

The control unit is the component of the central processing unit in the computer system. It is used to control the operation of the processor. The control unit tells to computer memory that how to respond to instructions which are sent to the processor.



A system on a Chip (SoC)

Memory Management Hardware

The memory management unit is the computer hardware unit having the reference of all memories passed through it. This memory unit is used to the translation of virtual memory address to a physical address. The memory management units allowed to managing the multiple programs in single physical memory with its own address space. It is used to provide virtual addressing.

Instruction Set Architecture

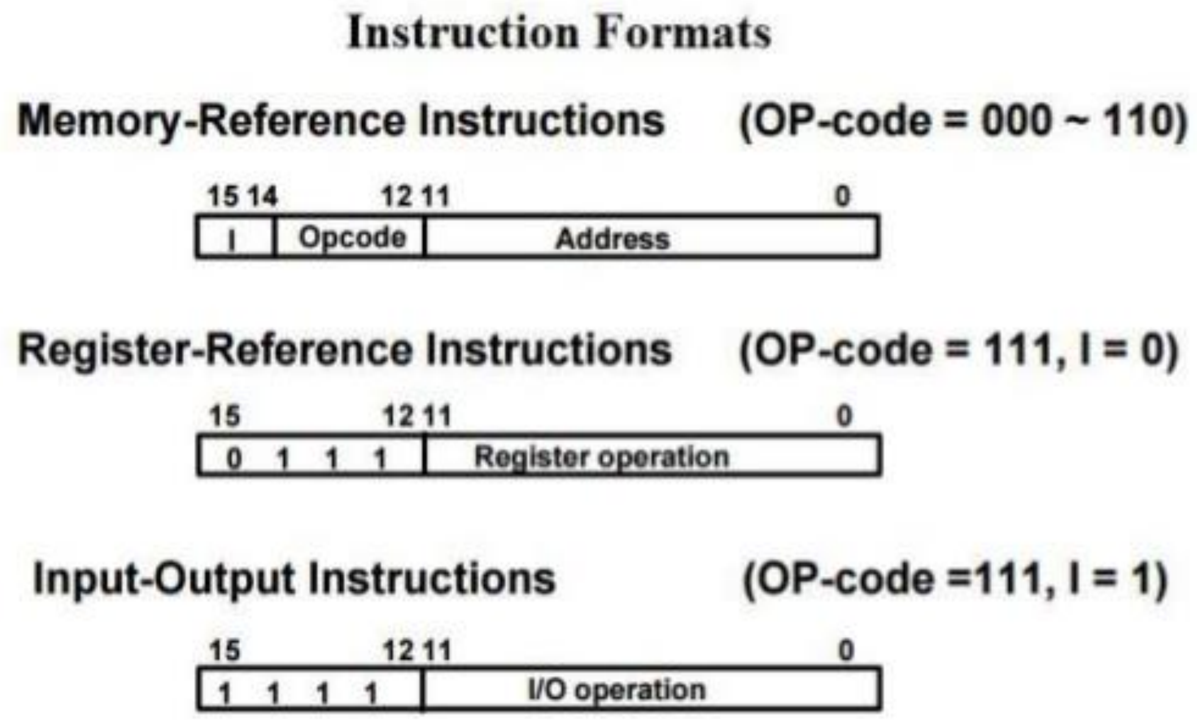
In computer science, an instruction set architecture (ISA) is an abstract model of a computer. It is also referred to as architecture or computer architecture. A realization of an ISA, such as a central processing unit (CPU), is called an implementation.

The basic computer has 16-bit instruction register (IR) which can denote either memory reference or register reference or input-output instruction.

Memory Reference – These instructions refer to memory address as an operand. The other operand is always accumulator. Specifies 12-bit address, 3-bit opcode (other than 111) and 1-bit addressing mode for direct and indirect addressing.

Register Reference – These instructions perform operations on registers rather than memory addresses. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 0 (differentiates it from input/output instructions). The rest 12 bits specify register operation.

Input/Output – These instructions are for communication between computer and outside environment. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 1 (differentiates it from register reference instructions). The rest 12 bits specify I/O operation.



Interrupt

Interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high priority process requiring interruption of the current working process. In I/O devices one of the bus control lines is dedicated for this purpose and is called the Interrupt Service Routine (ISR).

Hardware Interrupts:

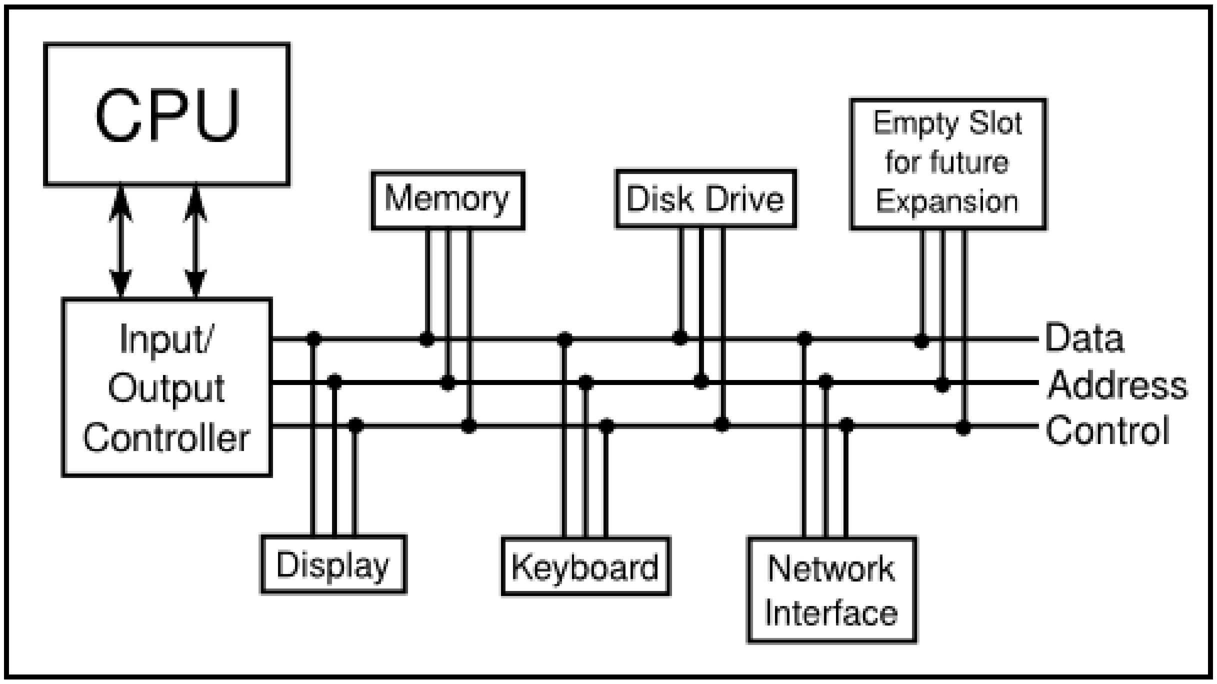
In a hardware interrupt, all the devices are connected to the Interrupt Request Line. A single request line is used for all the n devices. To request an interrupt, a device closes its associated switch. When a device requests an interrupts, the value of INTR is the logical OR of the requests from individual devices.

Internal Interrupt:

A software interrupt is generally raised when there is some kind of an error raised by the system or failing in the execution of a process.

Software Interrupt:

Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.



Language Processor

Assembler

The Assembler is used to translate the program written in Assembly language into machine code. The source program is a input of assembler that contains assembly language instructions. The output generated by assembler is the object code or machine code understandable by the computer.

Interpreter

The translation of single statement of source program into machine code is done by language processor and executes it immediately before moving on to the next line is called an interpreter.

Compiler

The language processor that reads the complete source program written in high level language as a whole in one go and translates it into an equivalent program in machine language is called as a Compiler.

Assembly Language or ASM

Assembly Language or ASM is a low level language better understood by a CPU. Programs written in assembly languages are compiled by an assembler. Every assembler has its own assembly language, which is designed for one specific computer architecture.
Machine language is a series of numbers, which is not easy for humans to read. Using ASM, programmers can write human-readable programs that correspond almost exactly to machine language.

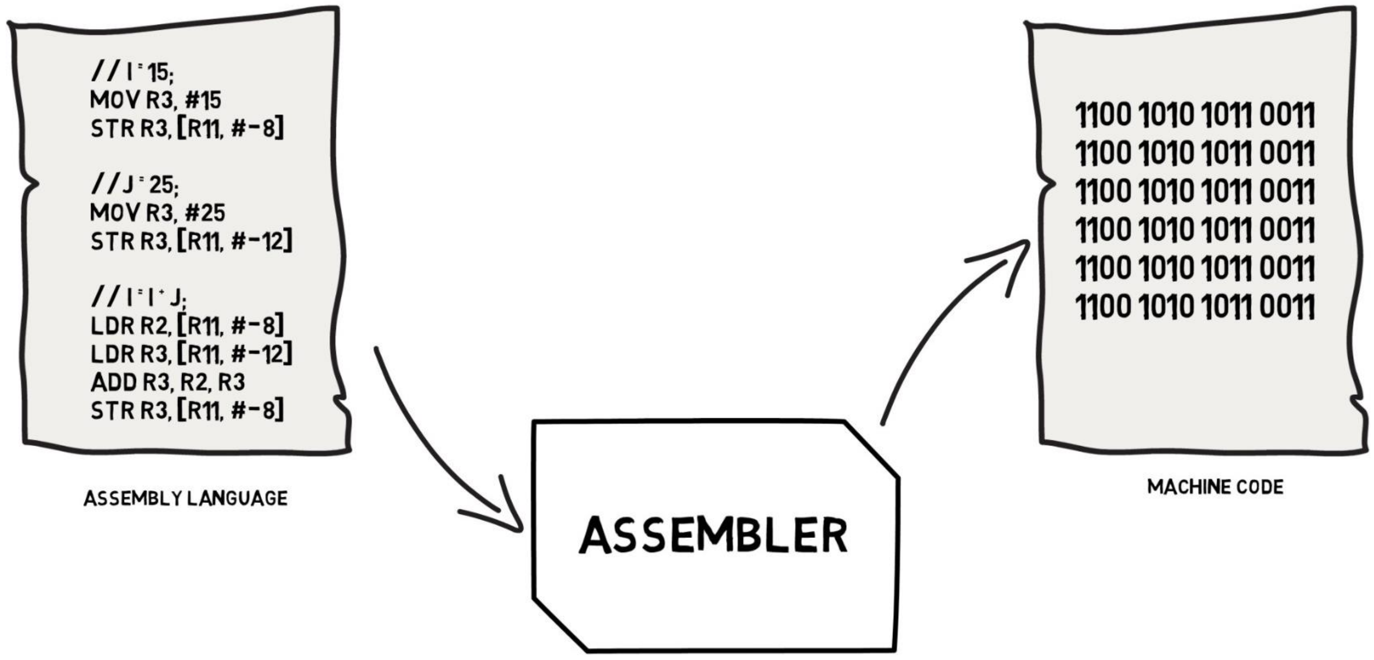
The compilation of Assembly language can be divided into to parts:-

Pass-1:

1. Define symbols and literals and remember them in symbol table and literal table respectively.
2. Keep track of location counter
3. Process pseudo-operations

Pass-2:

1. Generate object code by converting symbolic op-code into respective numeric op-code
2. Generate data for literals and look for values of symbols



```
// I = 15;
MOV R3, #15
STR R3, [R11, #-8]

// J = 25;
MOV R3, #25
STR R3, [R11, #-12]

// I = I + J;
LDR R2, [R11, #-8]
LDR R3, [R11, #-12]
ADD R3, R2, R3
STR R3, [R11, #-8]
```

```
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
```

CPU DESIGN

Section 1 - Introduction

The complete design of our CPU has been splitted in 4 parts, and in the coming sections, we have explained these 4 parts individually and in detail in the coming sections. The 4 divisions are as follows:

1. **Input/Output Organization (Section 3)**
    1. Deals with the input and output handling of our CPU and the general overall design and construction of pathways to internal processors and registers.
    2. Deals with components including Peripheral Devices, Interface Modules, IOPs etc.
2. **Memory Organization (Section 4)**
    a. The components of memory taken for accentuating the efficacy and smoothening the access to memory words, data, pointers, addresses and variables residing in the memory.
    b. Memory Access modes and Address Mapping tables.
3. **Bus System, Register Organization and Processing of Instructions (Section 5)**
    a. Bus system organization, with efficient and balanced arrangement of registers inside the processor

b. The process of fetching, decoding, executing and writing the result back ; explained through the working of the Instruction register along with Program Counter and Accumulator.

Final section of the document briefly describes the final integration of the components, and introduces the *Total Integrated CPU (TIC)*, and lays out the final, complete and concise designs of the entire CPU model.

Section 2 CPU Specifications

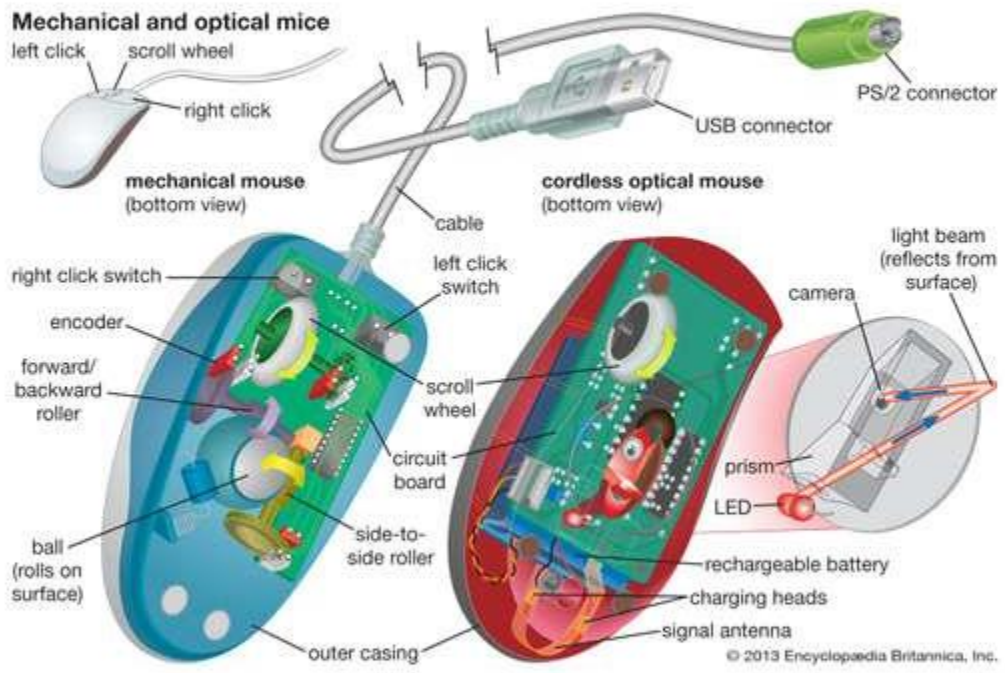
```
Architecture:                  Self-made
CPU op-mode(s):                16-bit, 32-bit
Byte Order:                    Little Endian
Address sizes:                 39 bits
physical, 48 bits virtual
Thread(s) per core:            2
Core(s) per socket:            6
Socket(s):                     1
CPU MHz:                       4300.003
CPU max MHz:                   4600.0000
CPU min MHz:                   800.0000
L1 cache:                      64 B
Main Memory size:              256 B
```

Section 3 Input/Output Organization

Section 3.1 General Introduction

Peripherals

Peripheral device, also known as peripheral, computer peripheral, input-output device, or input/output device, any of various devices (including sensors) used to enter information and instructions into a computer for storage or processing and to deliver the processed data to a human operator or, in some cases, a machine controlled by the computer. Such devices make up the peripheral equipment of modern digital computer systems



<u>Interfaces</u>

Interface is a shared boundary between two separate components of the computer system which can be used to attach two or more components to the system for communication purposes.

We will be specifically dealing with I/O Interface:

The method that is used to transfer information between internal storage and external I/O devices is known as I/O interface. The CPU is interfaced using special communication links by the peripherals connected to any computer system. These communication links are used to resolve the differences between CPU and peripheral. There exists special hardware components between CPU and peripherals to supervise and synchronize all the input and output transfers that are called interface units.

<u>Modes of Transfer</u>

There are 3 basic and absolute modes of transfer using I/O interface module:

1. *Programmed I/O.*
2. *Interrupt- initiated I/O.*
3. *Direct memory access (DMA).*

However we will be focussing on **Direct Memory Access** because as we will see that we have implemented/ used

IOPs(*Section2.3.3*) which are based on DMAs, and thus we will be discussing few concepts related to DMA in order to better understand the ***implementation of our input/output design.***

Section 3.2 Introduction to DMA (Direct Memory Access)

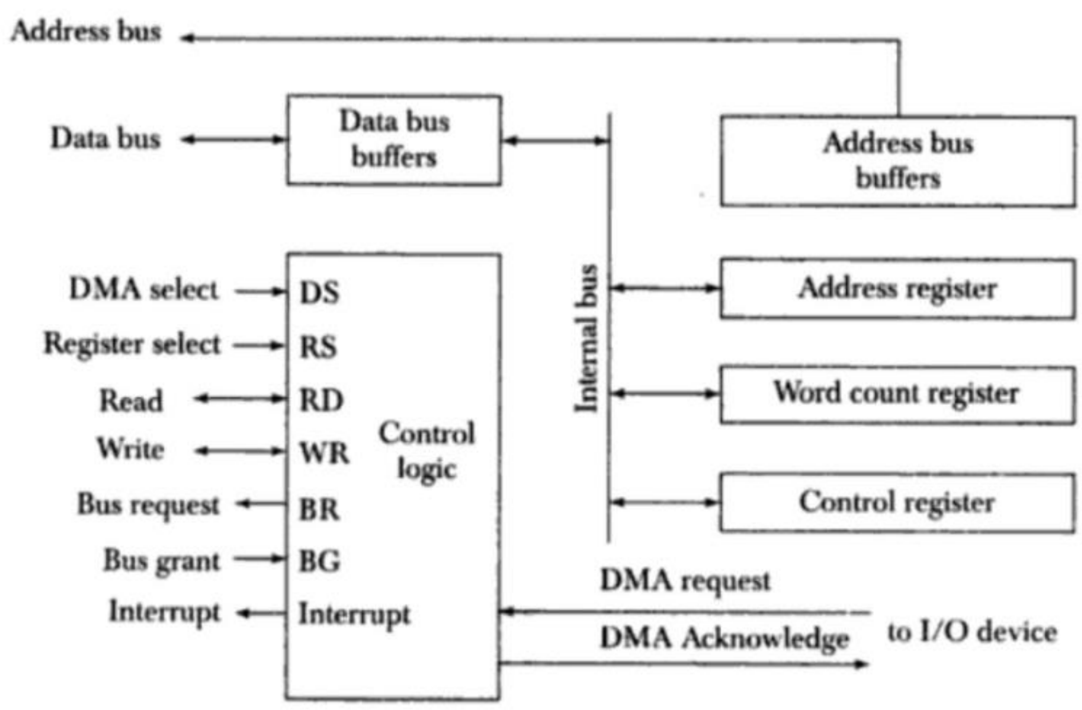
The data transfer between a fast storage media such as a magnetic disk and memory unit is limited by the speed of the CPU. Thus we can allow the peripherals to directly communicate with each other using the memory buses, removing the intervention of the CPU. This type of data transfer technique is known as DMA or direct memory access. During DMA the CPU is idle and it has no control over the memory buses. The DMA controller takes over the buses to manage the transfer directly between the I/O devices and the memory unit.

Bus Request

It is used by the DMA controller to request the CPU to relinquish the control of the buses.

Bus Grant

It is activated by the CPU to Inform the external DMA controller that the buses are in high impedance state and the requesting DMA can take control of the buses. Once the DMA has taken control of the buses it transfers the data. This transfer can take place in many ways.



This diagram shows a general DMA controller working along with the main processor.

The Working

The working of DMA controllers starts with the concept of Bus Request and Bus Grant. Whenever a DMA request to the control unit of the DMA, a Bus request is generated which is sent to the processor, asking the processor to put its *Data Bus, Address Bus and its read and write lines* in *high impedance state*, so that DMA can take control of these lines, and directly access the memory space without the processor interfering with the process. After the task is completed, DMA transfers the control back to CPU, and again waits to send another Bus request whenever required by the external peripheral devices.

The given registers can be accessed by the processor in case it wants to check anything at the end of DMA's work cycle.

Section 3.3 Implementation of 8089 as IOP

Input-Output Processor

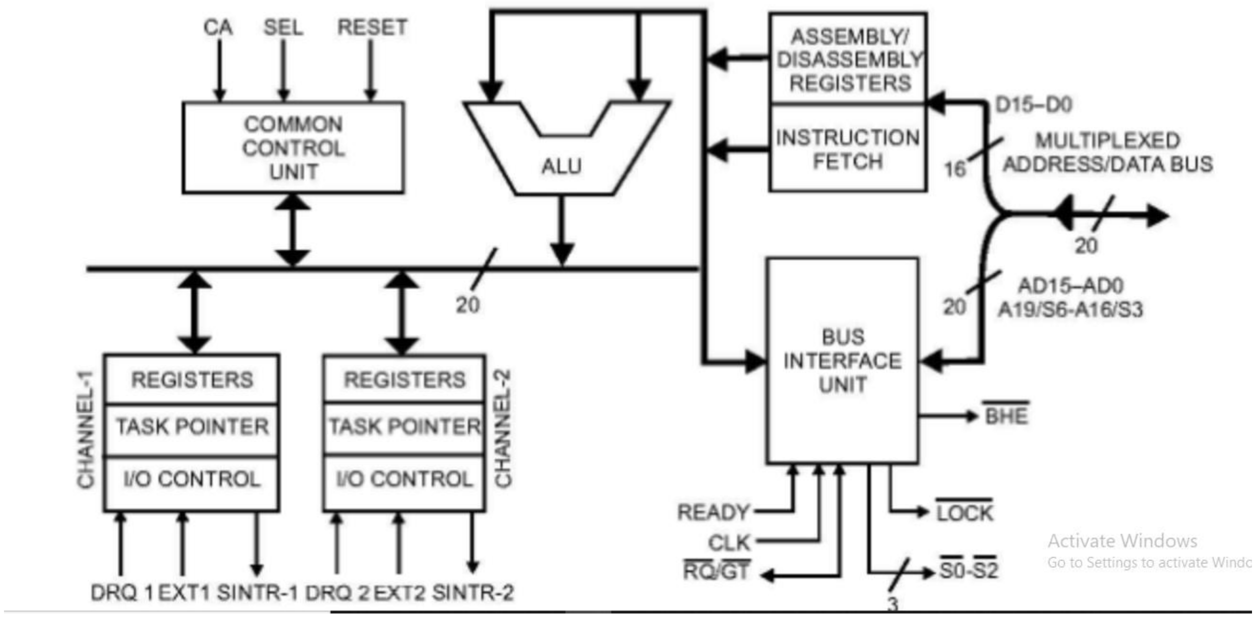
The Input Output Processor (IOP) is just like a CPU that handles the details of I/O operations. It is more equipped with facilities than those available in typical DMA controllers. The IOP can fetch and execute its own instructions that are specifically designed to characterize I/O transfers. In addition to the I/O – related tasks, it can perform other processing tasks like arithmetic, logic, branching and code translation. The main memory unit takes the pivotal role. It communicates with the processor by means of DMA.

8089 Features

Features:

1. 8089 has very high speed DMA capability.
2. It has 1 MB address capability.
3. It is compatible with iAPX 86, 88.
4. It supports local mode and remote mode I/O processing.
5. 8089 allows a mixed interface of 8-and 16-bit peripherals, to 8-and 16-bit processor buses.
6. It supports two I/O channels.
7. Multibus compatible system interface.
8. Memory based communications with CPU.

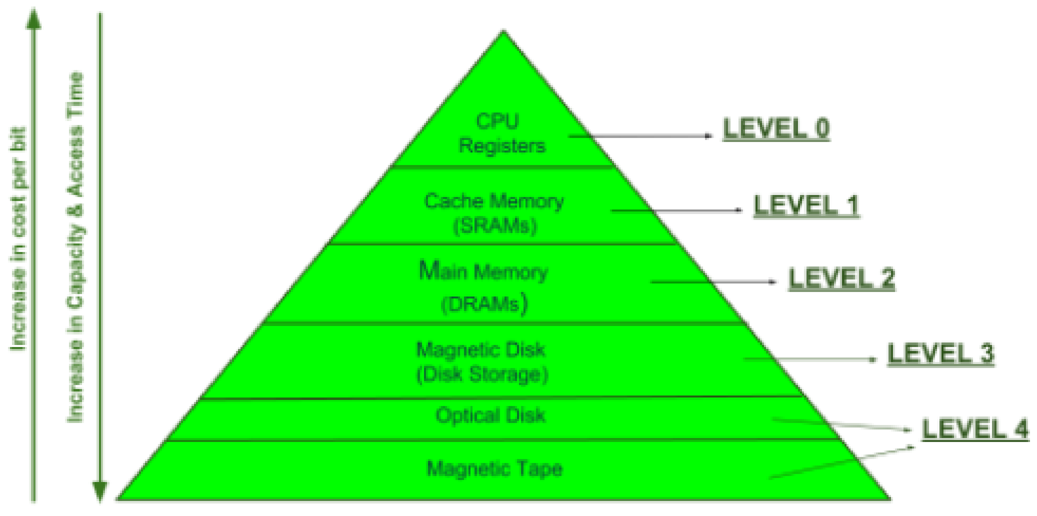
8089 Implementation In Our Design



Section 4 Memory Organization

Section 4.1 Memory Hierarchy

In the Computer System Design, Memory Hierarchy is an enhancement to organize the memory such that it can minimize the access time. The Memory Hierarchy was developed based on a program behaviour known as locality of references.



MEMORY HIERARCHY DESIGN

Locality of Reference

Locality of Reference, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time. There are two basic types of reference locality – temporal and spatial locality. Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality (also termed *data locality*) refers to the use of data elements within relatively close storage locations.

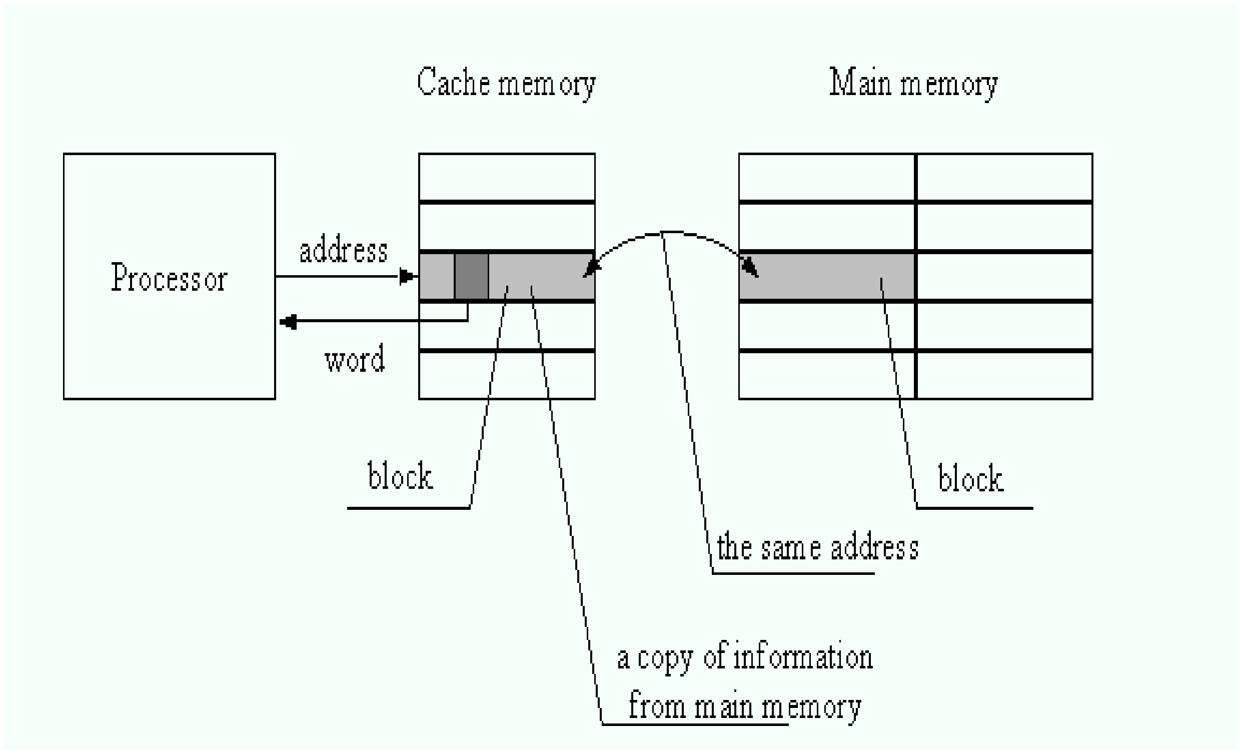
Section 4.2 Memory Distribution in our CPU

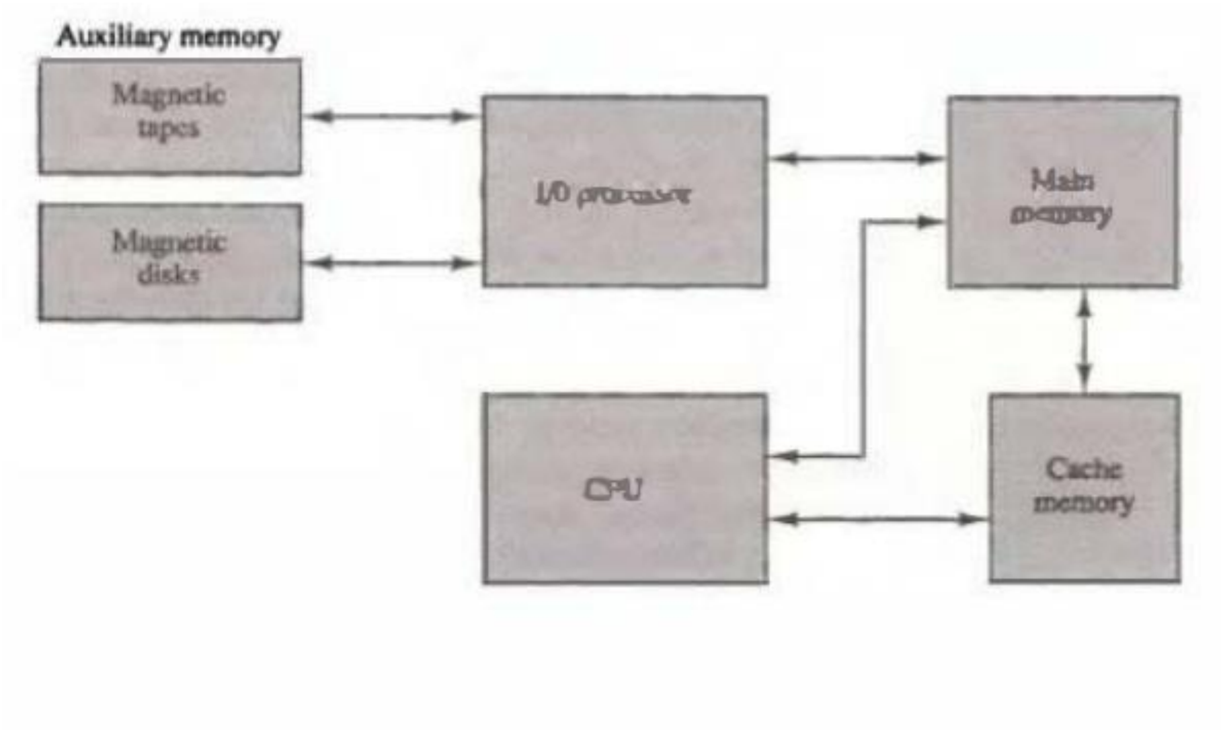
The memory has been divided into 3 parts as, Cache memory(the fastest memory there is), Main memory and Auxiliary memory. The specifications have been given earlier.

Cache memory

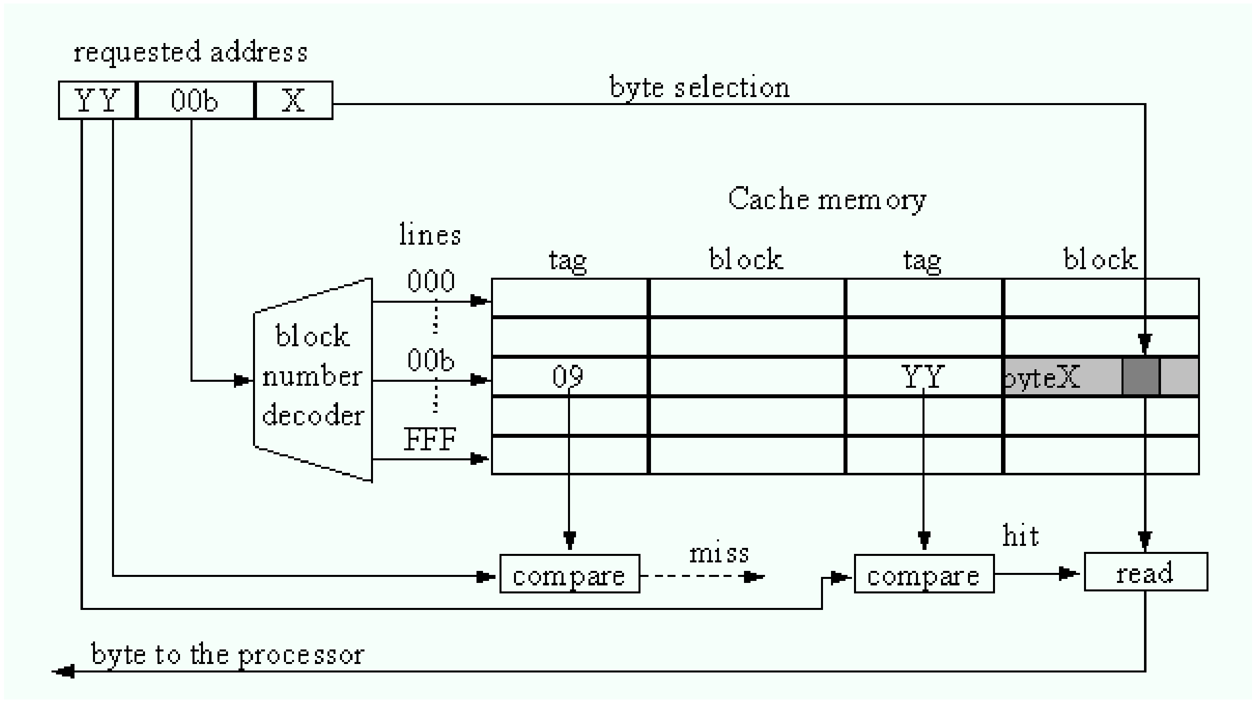
Cache Memory is a special very high-speed memory. It is used to speed up and synchronize with a high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

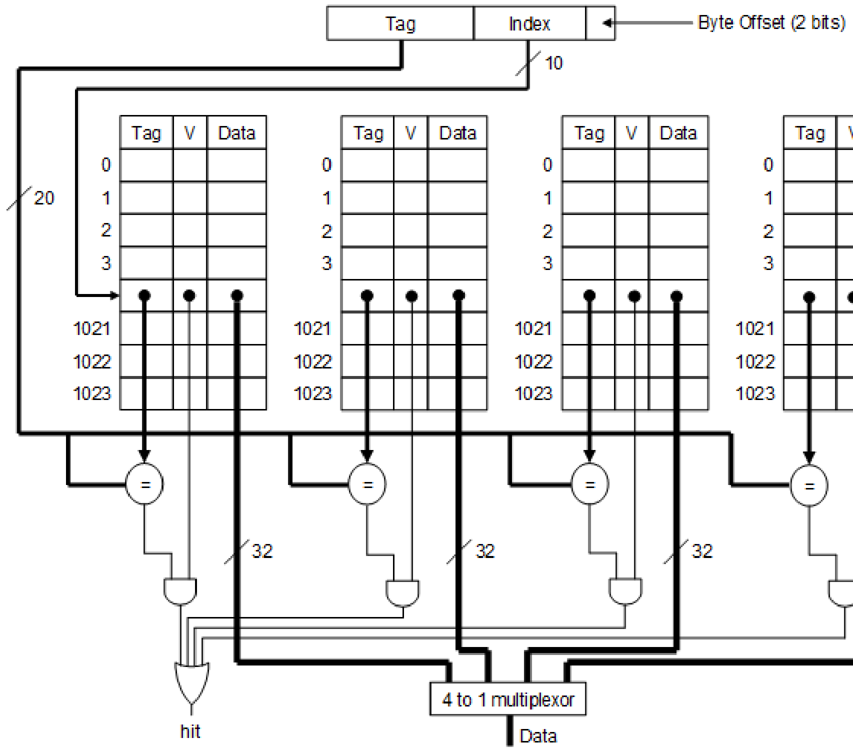
There are several ways to use cache memory, we will be using Set Associative mapping.





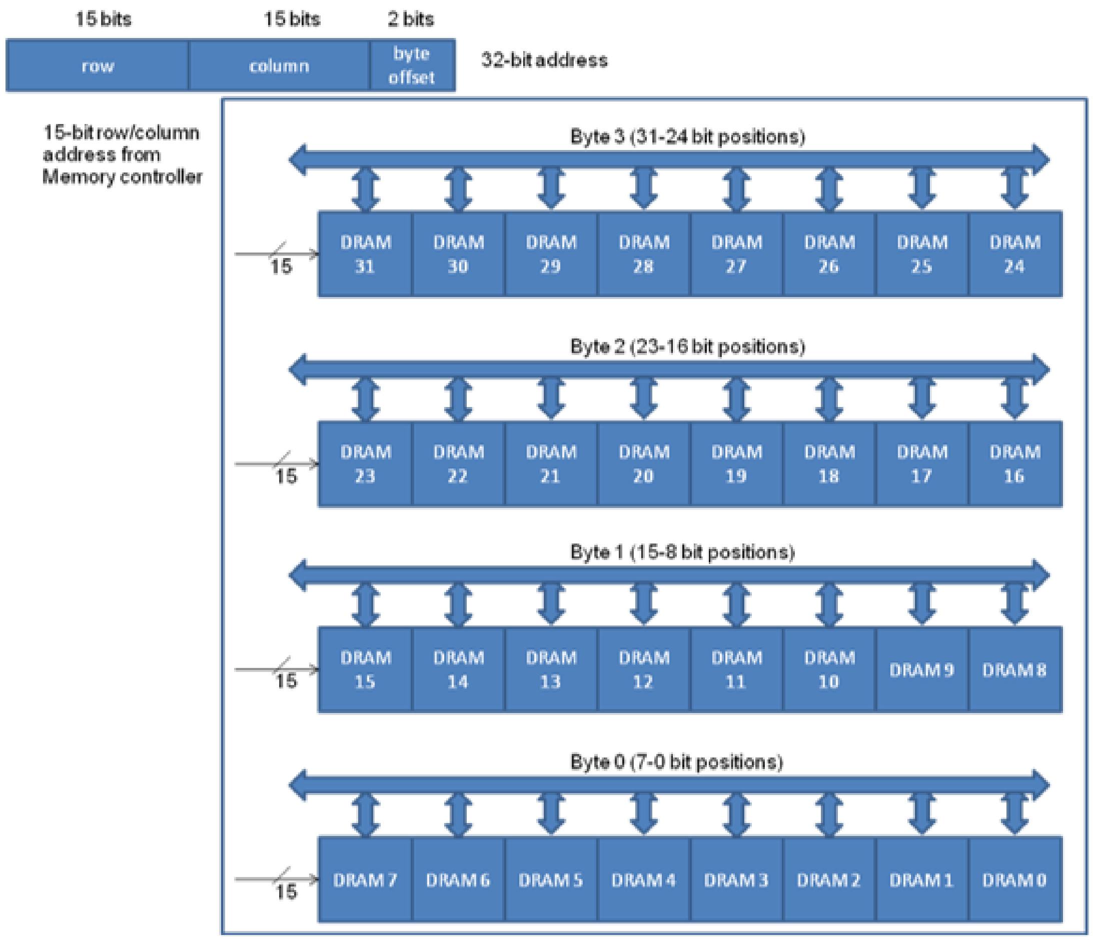
For Set Associative Mapping:





Main Memory

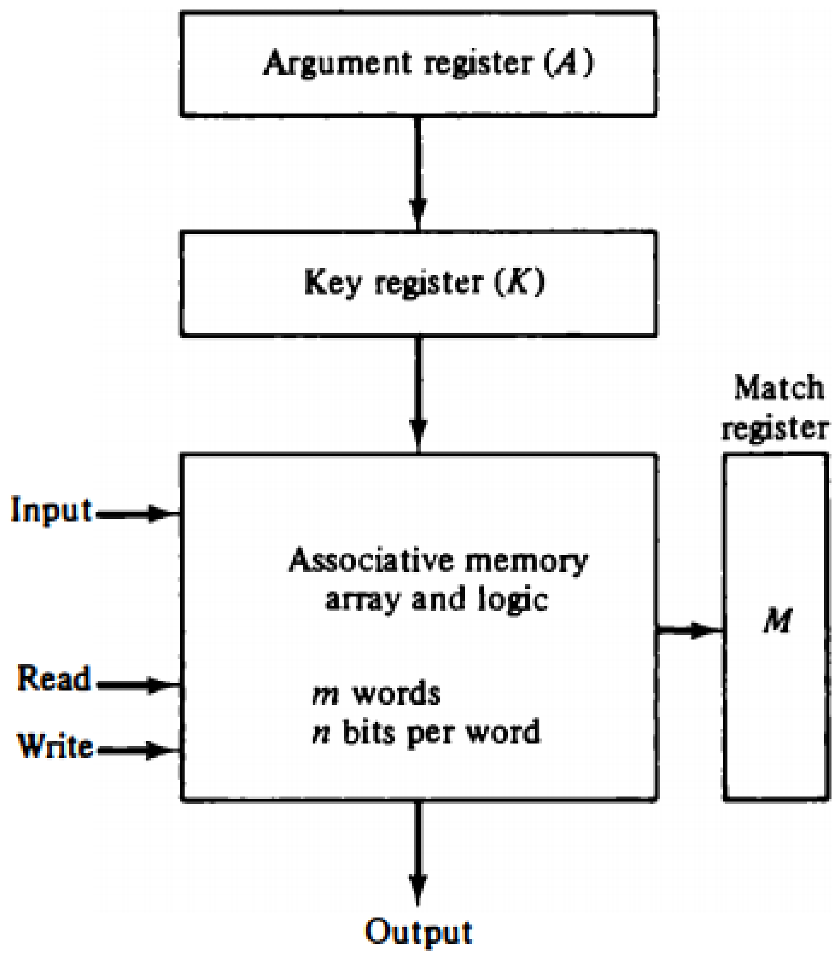
A 2D arrangement of DRAMs is shown which forms the complete memory set of the main memory. The memory control word is tried on the main memory if there is a “miss” in the cache, then the same sized control word is tried in main memory and the data of that address is then , fetched to the processor, and also it is put inside the cache in hope that it will used again or in the near future.



Auxiliary Memory

An Auxiliary memory is known as the lowest-cost, highest-capacity and slowest-access storage in a computer system. It is where programs and data are kept for long-term storage or when not in immediate use. The most common examples of auxiliary memories are magnetic tapes and magnetic disks.

We will be using Associative Memory for our Auxiliary memory:



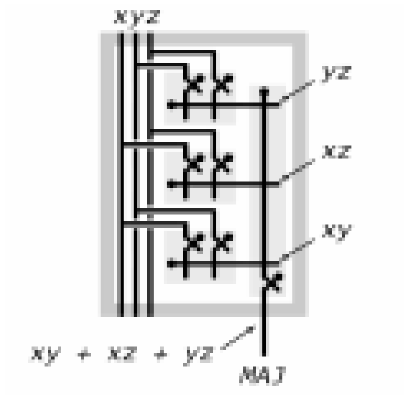
Associative memory Mapping for Auxiliary memory

Section 5 Bus System, Register Organization and Processing of Instructions

<u>Understanding the Basic Symbols</u>

Combinational circuits.:

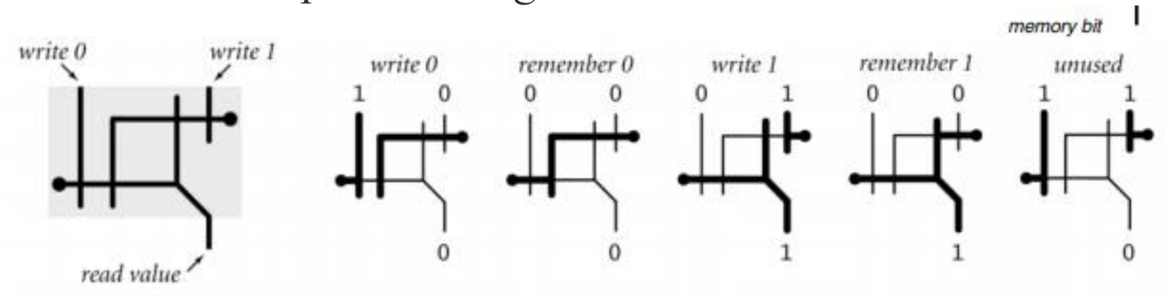
• Output determined solely by inputs.



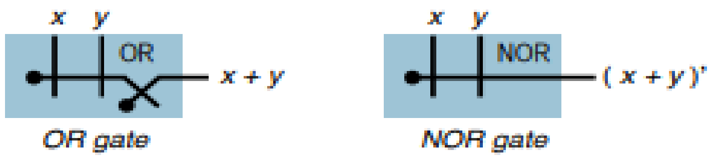
• Ex: majority, adder, decoder, MUX, ALU

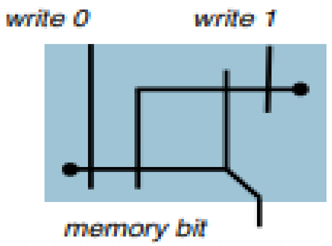
SR Flip-flop.

• Two cross-coupled NOR gates



• A way to control the feedback loop.
• Abstraction that "remembers" one bit





. • Basic building block for memory and registers

Starting with Basic Registers(Part of the Processor)

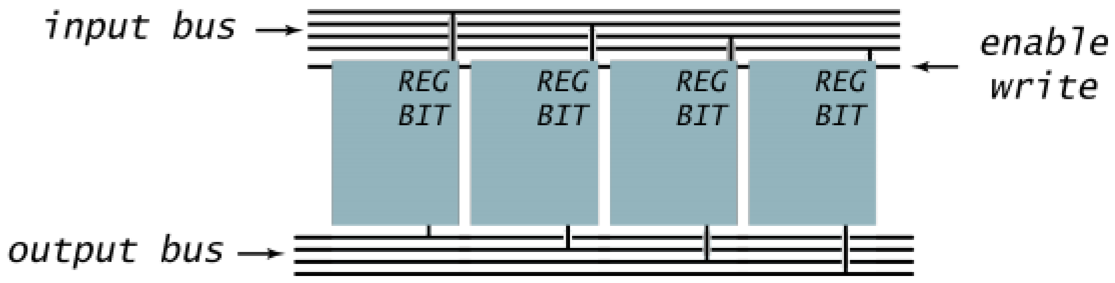
Processor register :

• Stores k bits.

• Register contents always available on the output bus.

• If enabled write is asserted, k input bits get copied into the register.

Program Counter(PC) holds an 8 bit address and instruction register (IR) holds 16-bit current instruction

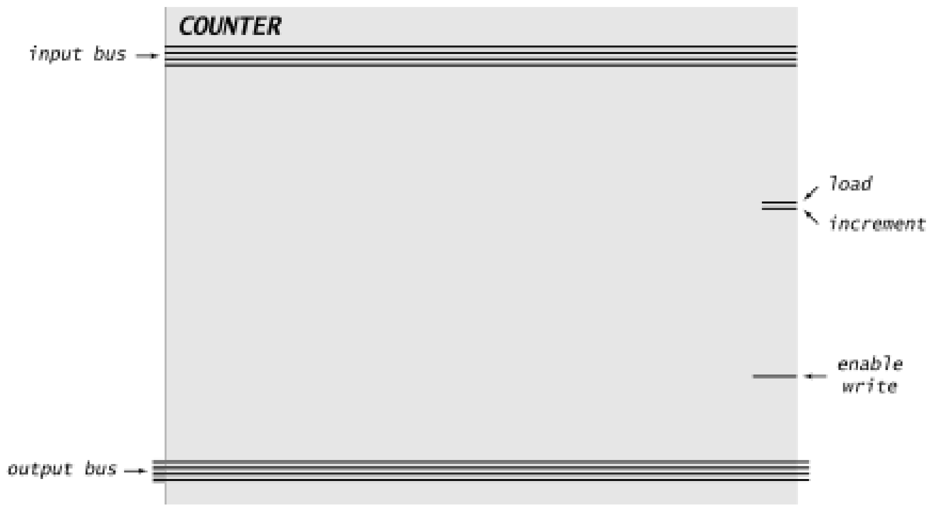


Program Counter:

A program counter is a register in a computer processor that

contains the address (location) of the instruction

- Holds value that represents a binary number.

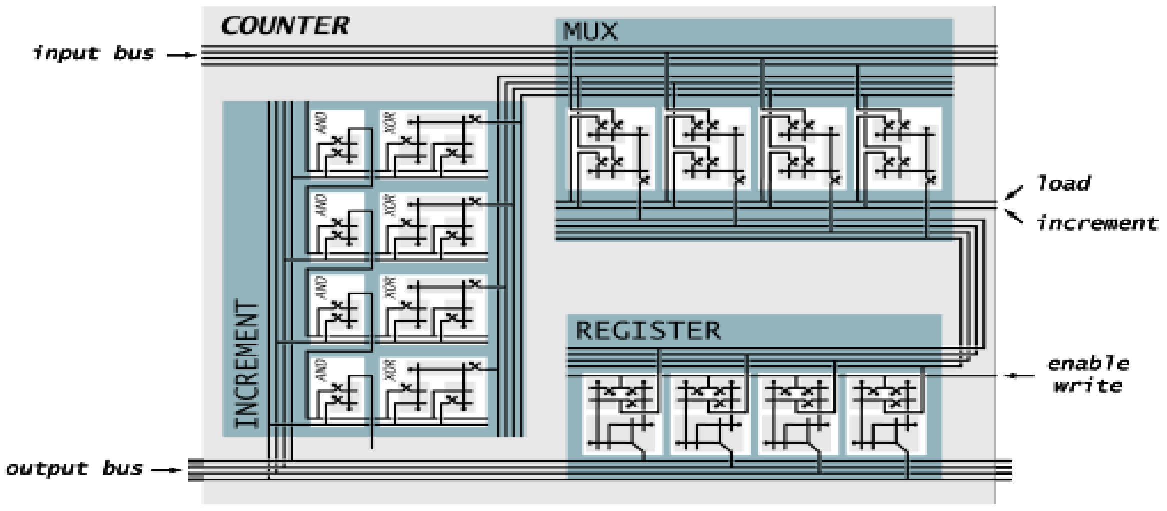


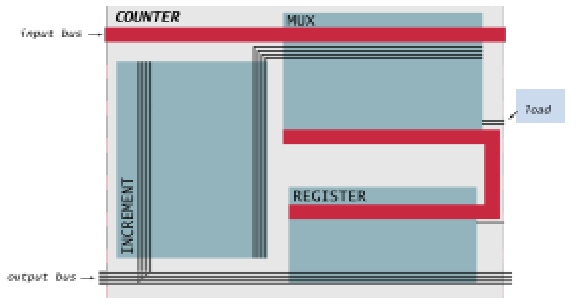
- Load: set value from input bus.
- Increment: add one to value.
- Enable Write: make value available on the output bus.

PC's Datapath:

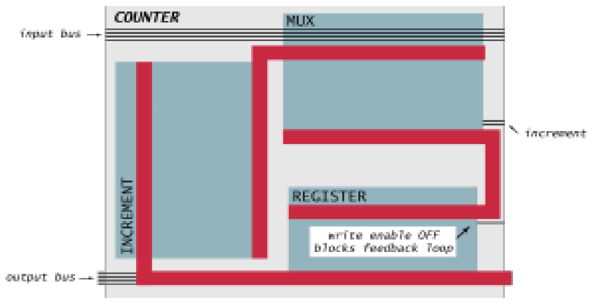
• Layout and interconnection of components.

• Connect input and output buses.

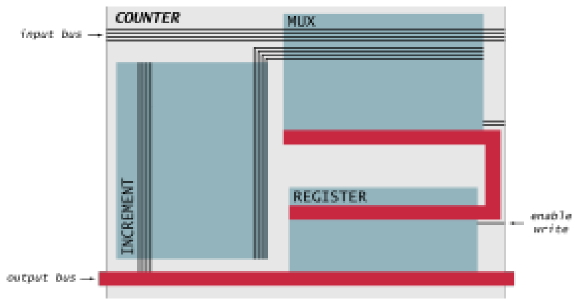




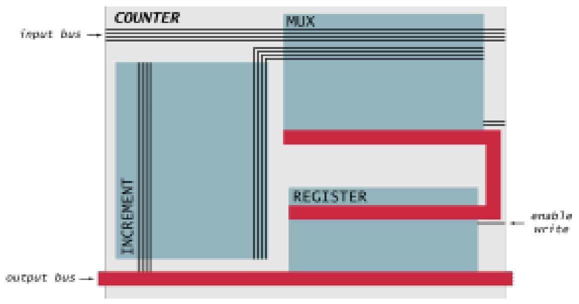
1. load:
copy input to register



3. increment:
output plus 1 available in MUX
copy to register

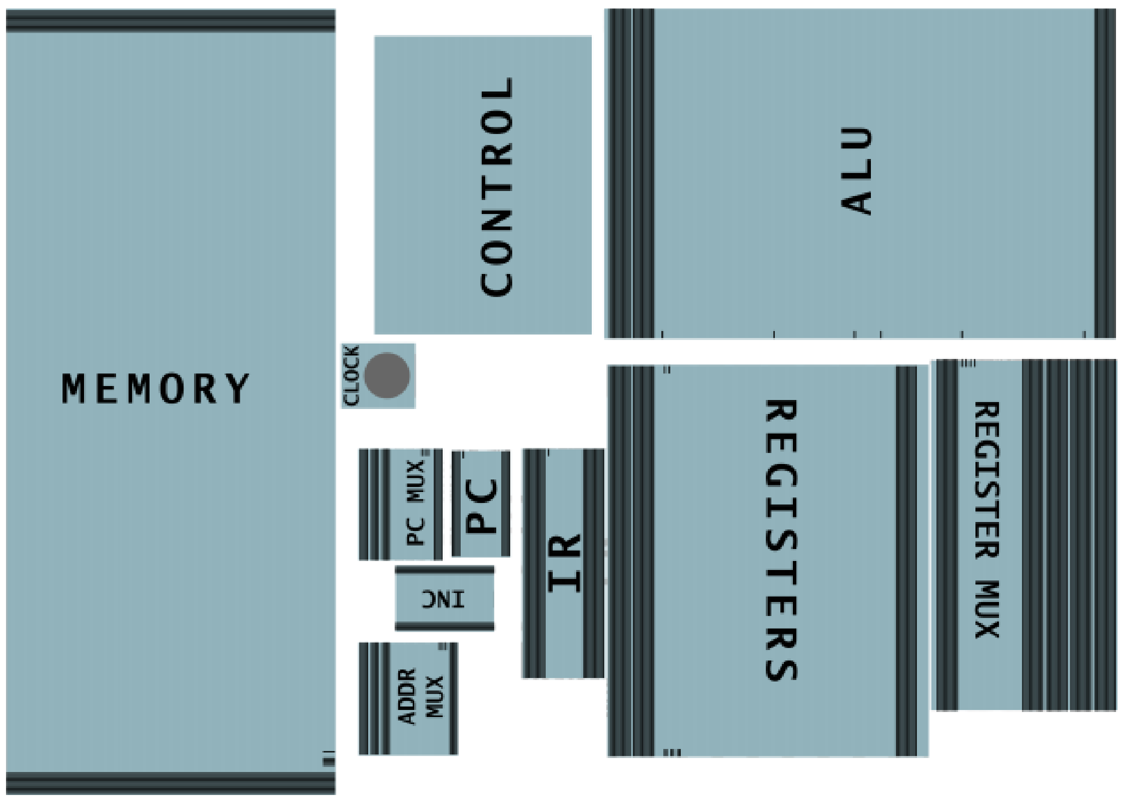


2. enable write:
register contents available on output

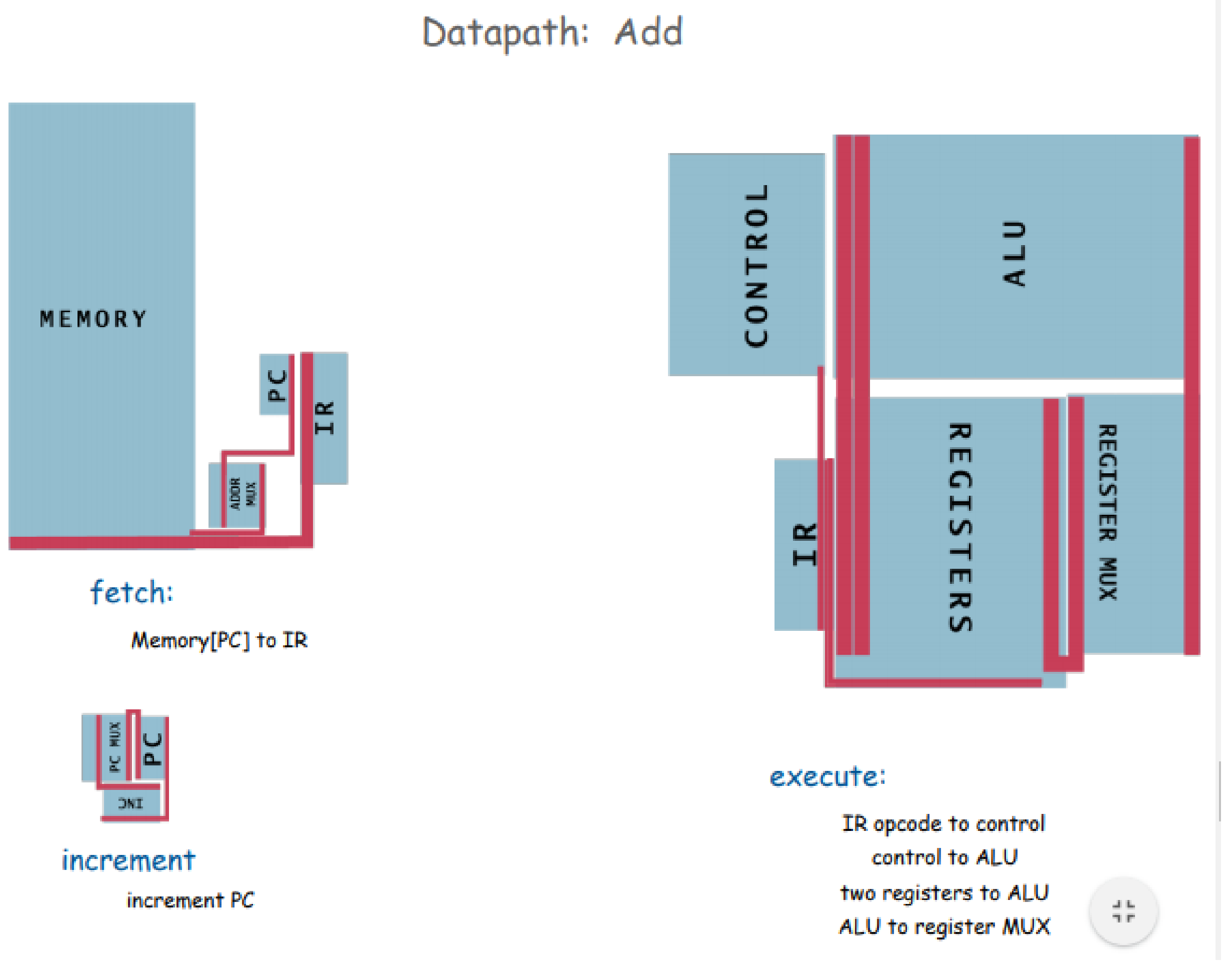


4. enable write:
register contents available on output

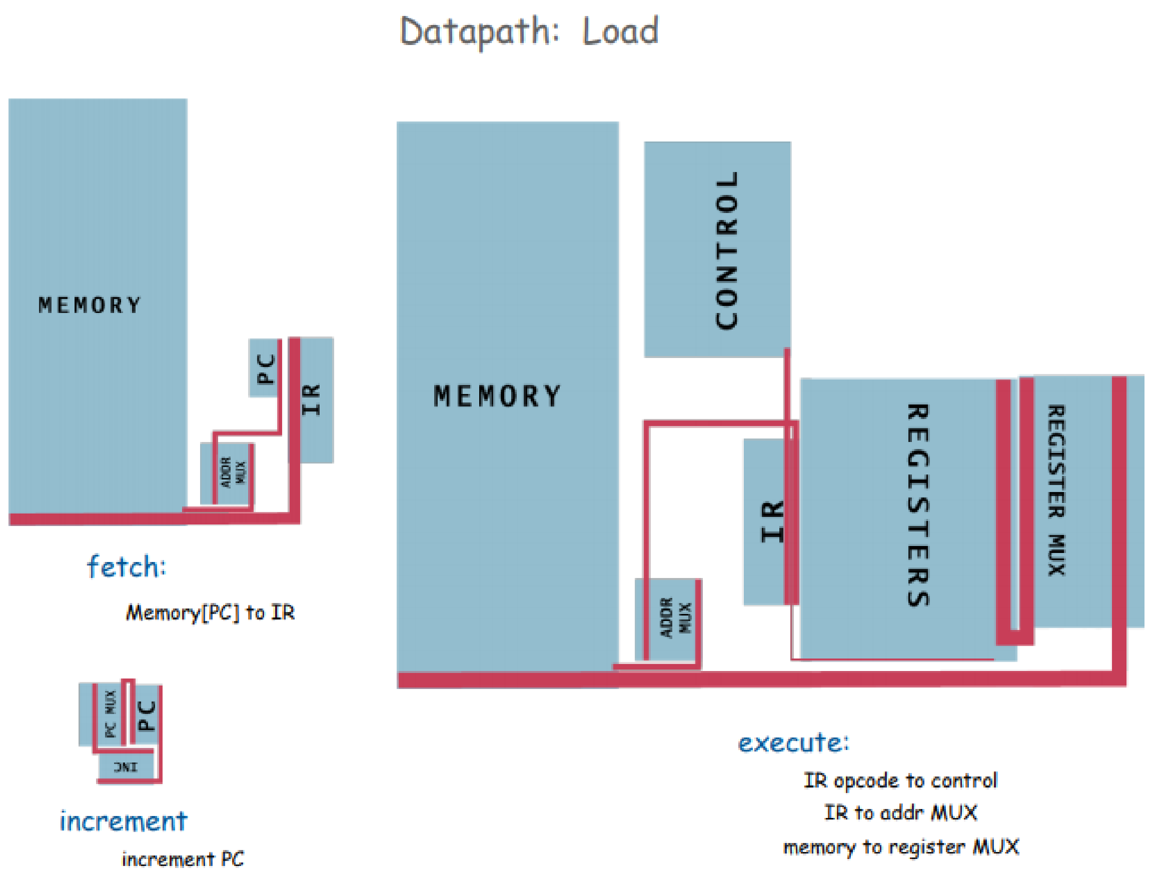
Understanding the working of Instruction Register, Program Counter and Accumulator



General Configuration of our CPU

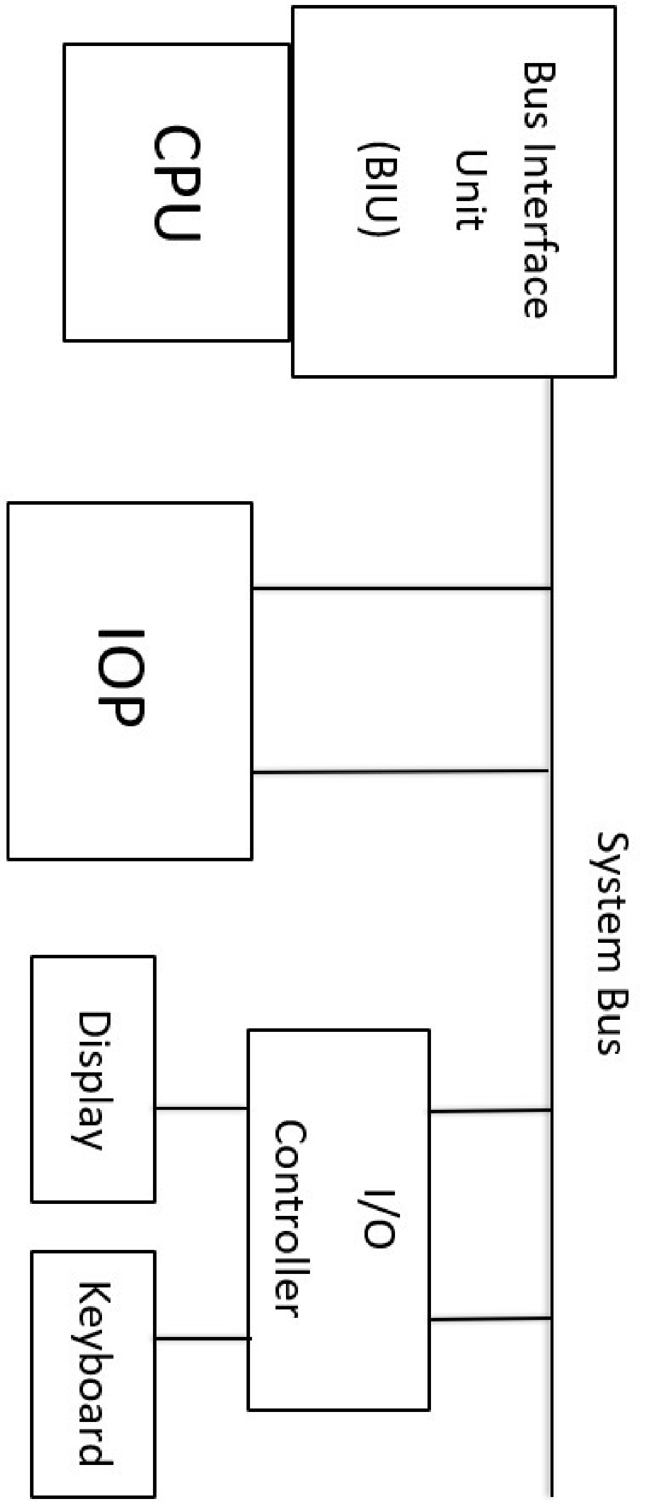


The instruction cycles/steps to add

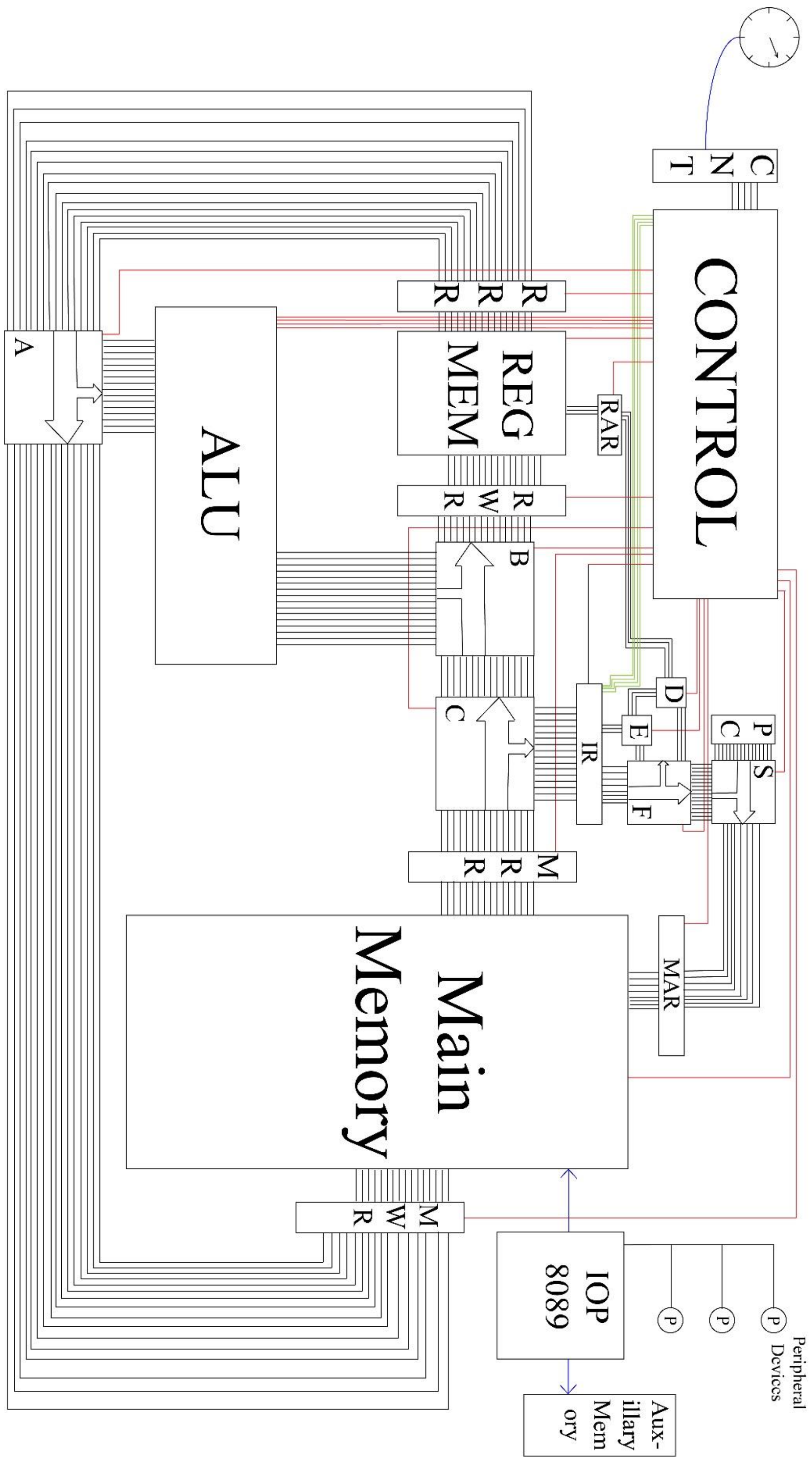


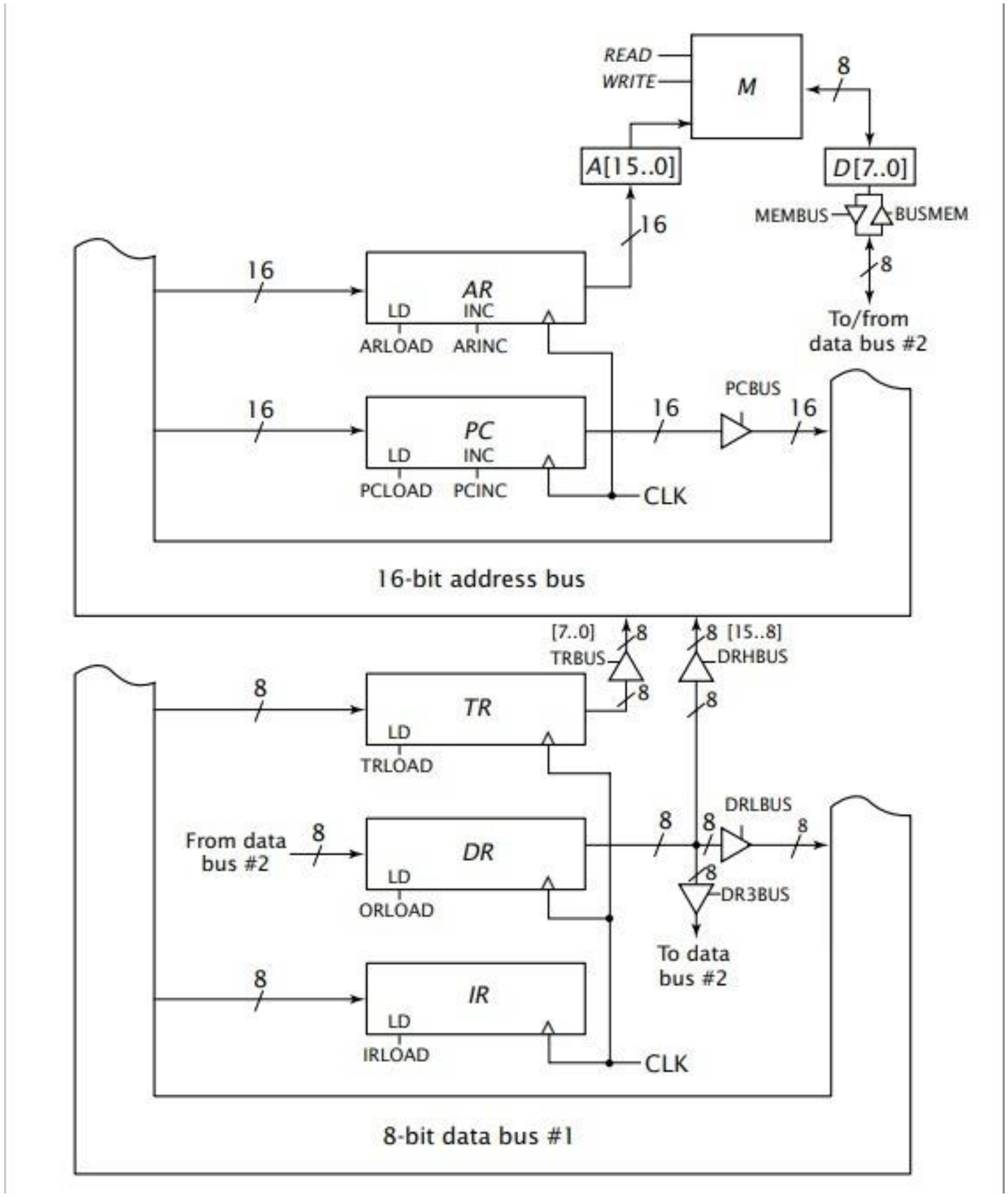
The instructions/steps to load from register MUX

Flowchart

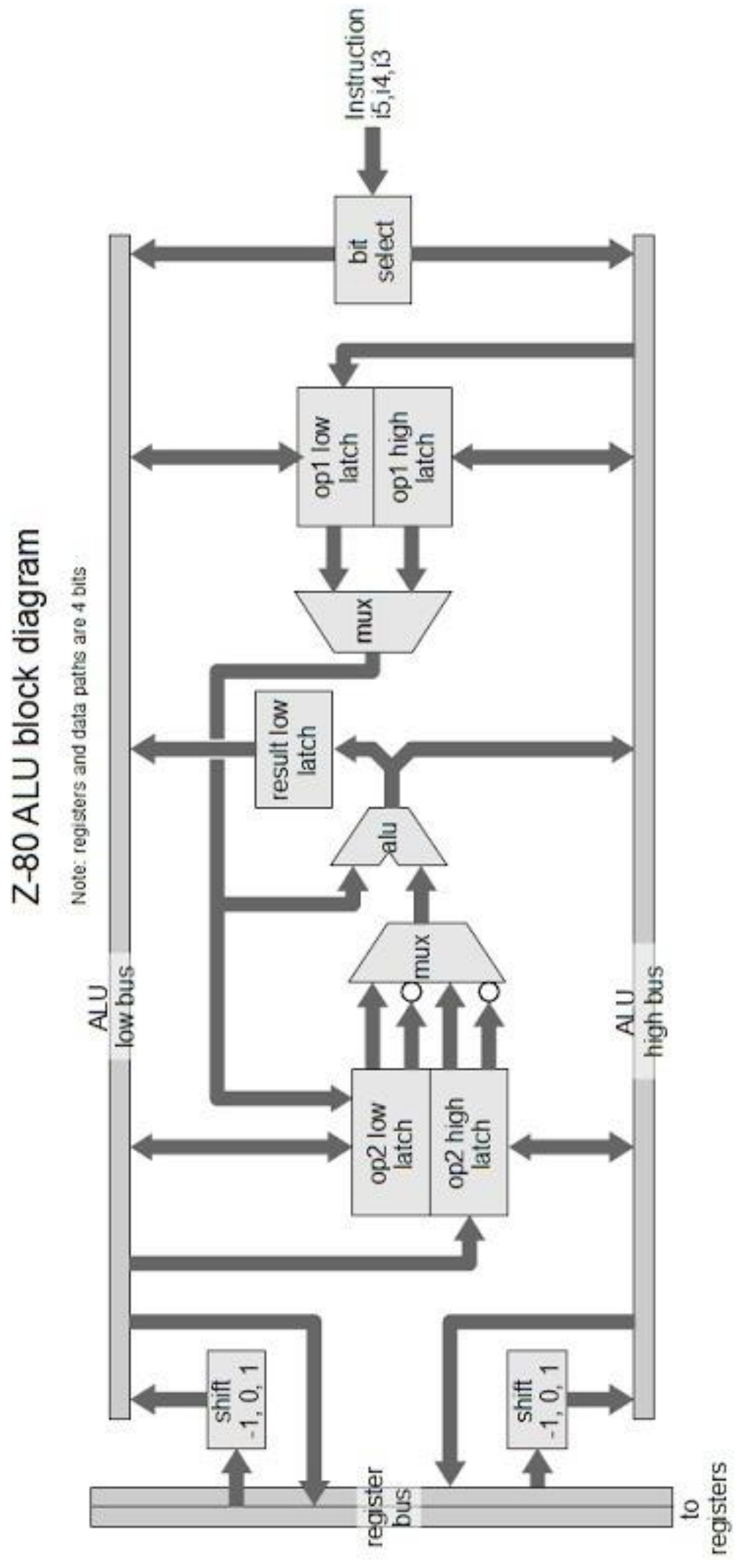

CPU
Bus Interface Unit (BIU)
IOP
System Bus
Display
Keyboard
I/O Controller

Section 5 Total Integrated CPU




READ
WRITE
M
8
A[15..0]
D[7..0]
MEMBUS
BUSMEM
16
8
To/from data bus #2
16
AR
LD
INC
ARLOAD
ARINC
PCBUS
16
16
16
PC
LD
INC
PCLOAD
PCINC
CLK
16-bit address bus
[7..0]
8
8
[15..8]
TRBUS
DRHBUS
8
TR
LD
TRLOAD
8
8
DRLBUS
8
8
8
From data bus #2
DR
LD
ORLOAD
8
DR3BUS
To data bus #2
8
IR
LD
IRLOAD
CLK
8-bit data bus #1

BUS System



ALU Taken from Z-80

Elucidation

Two Pass Assembler

Assumptions

Opcode and Operands

1. CLA and STP take no operands/arguments
2. LAC, SAC, INP, DSP take 1 operand which is an address
3. ADD, SUB, MUL, DIV take 1 operand which can be a literal or an address
4. BRZ, BRN, BRP take 1 operand which should be a valid label
5. If DIV is called, the quotient and remainder get stored in Variables R1 and R2 which can be used in the code later. If DIV is called again, then the value in R1 and R2 will get overwritten.
6. SAC and INP can take undefined address as an argument. Rest of the opcodes cannot. (If no valid value is present in an address, it is treated as an undefined address).
7. RET will return back to the original calling function from where the branch had begun.
8. ISZ is Skip if Zero otherwise increments the program counter.

General

1. Clear accumulator (CLA) resets the accumulator. No address or value is present in the accumulator.
2. Execution stops after the end statement. The code written after the end statement is not executed.
3. START loads the instructions from the specified address. If no address is specified, instruction set is loaded normally.
4. Label name **cannot** be an opcode name and cannot have a Macro name anywhere in it.
   For example if ADDTWO is a macro, then: "SUB", "ADD", "ADDTWOXYZ" are invalid label names. While this can be handled, it is not considered to be a good programming practice.
5. Comments can be added with the help of **; or /**.
6. Literals/Constants can be specified between single quotes "’’". Example ADD '3'
7. Variable names and label names can be alpha-numeric. However, the same variable cannot be used as a label name and vice versa.
8. Literals can be of any value, contiguous memory spaces are allotted accordingly.
9. If the size of the literal uses more than 1 memory space, then only the first is written in the machine code. Exact memory mapping can be seen from the Literal Table.

10. The maximum memory size is 256 words, which can be stored using 8 bit addresses. If the program size is greater than this, the assembler will throw an error and terminate.
11. Total instruction length is 12 bit: 8 bit instruction address + 4 bit opcode

Macros

1. Macros can be handled.
2. Macro is defined at the top of the file before being called.
3. Macro calls cannot be made inside other macro definitions.
4. There can be more than one macro
5. Macros can have labels.

BluePrint

Data Table

Contains all the (numeric) addresses defined in the code

1. **Value:** Integer address value
2. **Defined/Undefined:** Defined if the address is taking in a value using INP. Example: INP 157, else undefined Example:

```
INP 157
LAC 157
DIV 157
```

Label Table

Contains all the Labels defined in the main code and the modified labels created after expanding macro calls.

1. **Label name:** Name of the label as defined in the code
2. **Virtual Address:** Virtual address allotted to the label during the first pass
3. **Code:** refers to the name of the macro or Main code
4. **Physical Address:** Physical address allotted to the label during the second pass

Symbol Table

Contains all the variables defined in the main code

1. **Name:** Variable name defined in the main code
2. **Physical Address:** Physical address allotted to the variable during the second pass

Literal Table

Contains all the constants defined in the code

1. **Name:** String value of the constant eg '1'
2. **Value:** Integer value of the literal
3. **Size:** Word size of the literal considering 2s complement for storing data

4. **Physical address:** Physical address allotted to the Literal during the second pass

Macro Table

Contains all the macros and their definitions

1. **Name:** Name of the macro
2. **Macro Parameters:** Parameters used with the macro. Eg ADDTWO MACRO A,B,C -> parameters=A,B,C
3. **Instruction Table:** Instructions present in the macro without modifications
4. **Labels:** Labels defined in the macro without modifications

Instruction Table

Contains all the instructions and expanded Macro calls

1. **Virtual Address:** Generates a new virtual address for every instruction
2. **Instruction:** the complete instruction When a macro call is encountered, the instructions present in the macro table for the respective macro definition is substituted in the this instruction table.

OPCODES LIST

Opcode	Meaning	Assembly Code
0000	Clear Accumulator	CLA
0001	Load into Accumulator from address	LAC
0010	Store Accumulator contents into address	SAC
0011	Add address contents to accumulator contents	ADD
0100	Subtract address contents to accumulator contents	SUB
0101	Branch to address if accumulator contains zero	BRZ
0110	Branch to address if accumulator contains negative value	BRN
0111	Branch to address if accumulator contains positive value	BRP
1000	Read from terminal and put in address	INP
1001	Display value in address on terminal	DSP
1010	Multiply accumulator and address contents	MUL

1011	Divide accumulator contents by address contents. Quotient in R1 and remainder in R2	DIV
1100	Stop Execution	STP

Function Table

Function Name	Input	Output	Exceptions and Errors
removeComments	Single instruction in assembly language as a string	Instruction in assembly language as a string, without comments Parses the instruction passed as input to remove comments, that is, any text written beyond the ';' character.	None
checkMacro	Instruction	Checking for beginning and ending of Macro definition.	None

addMacro	Macro name and parameters.	Adds the macro and it's parameters to the macro table.	Throws MACRO defined more than once exception.
getLabel	Instruction from instruction table	True if label definition is found, else, returns False.	None
addLabel	label name, label declaration address, part of program to which the label belongs (macro body/main).	Adds detected label to the label table	Throws exception if detected label is invalid: Already used as variable, or, contains the name of a macro, or, has been defined more than once,or, has the

			same name as a valid opcode.
addData	Opcode and operands following the opcode for given instruction.	Adds operands to the dataTable/ literalTable/ symbolTable.	Throws Memory Address out of bounds error.
getLiteral	Operand for given instruction.	Literal if found, else, returns False.	None
addLiteral	Detected Literal	Adds newly detected literal to literal table. '''	None
refine	Instruction	Removes comments, splits instruction into opcode and operands.	None
handleMacroCalls	Macro name, macro parameters and	Maps actual and formal parameters and expands the	None

	number of instructions.	macro call in the instruction table.	
getOffset	Number of instructions present in instruction table.	Offset/Starting address for instruction table, to be stored in a contiguous memory space.	Throws "Not enough space" exception if instruction table size is larger than available memory, or if a contiguous memory space cannot be found.
addOffset	Offset calculated for binding of instructions and labels.	Maps the instructions and labels in Instruction Table and Label Table to physical addresses by adding offset	None
getLiteralPool	offset for Instruction	Offset/Starting address for literal	"Not enough

	table and total number of instructions.	pool, to be stored in a contiguous memory space	space" exception if literal pool is larger than available memory, or if a contiguous memory space cannot be found.
assignLiteralPoo l	Starting address for literal pool	Assigns physical addresses for literals for binding.	None
getSymbolPool	offset for Instruction table, literal pool starting and ending addresses, total number of instructions	Offset/Starting address for variables in symbol table, to be stored in a contiguous memory space	Throws "Not enough space" exception if variable pool is larger than available memory, or if a

			contiguous memory space cannot be found
assignSymbolPool	Starting address for variable pool	Assigns physical addresses for variables for binding.	None
removeLabelDefinitions	None	Removes label declarations from the instruction table for conversion to machine language	None
checkOperands	None	Checks validity of operands corresponding to opcodes. ADD, MUL, LAC, SUB: Only have defined variables/addresses or literals. DSP: Only has defined variable/address.	None

		BRN, BRP, BRZ: Only have defined label. SAC, INP: Only have defined/undefined variables/addresses DIV: Only has first operand as defined variable/address or literal, second and third operands as defined/undefined variables/addresses	
convertOpcodes	None	Convert opcodes in instruction table to machine language	None
convertOperands	None	Convert operands to the physical adresses they are bound to	None
writeToFile	None	Write generated machine code to text file named:	None

		<sample_file>_output.txt. Splits machine code into blocks of four bits for readability.	

CODE FOR ASSEMBLER

Code for Opcode Mapping

```
opcodes = {'CLA': '0000',
           'LAC': '0001',
           'SAC': '0010',
           'ADD': '0011',
           'SUB': '0100',
           'BRZ': '0101',
           'BRN': '0110',
           'BRP': '0111',
           'INP': '1000',
           'DSP': '1001',
           'MUL': '1010',
           'DIV': '1011',
           'STP': '1100'}

opcode_arguments = {'CLA': 0,
                    'LAC': 1,
                    'SAC': 1,
                    'ADD': 1,
                    'SUB': 1,
                    'BRZ': 1,
                    'BRN': 1,
                    'BRP': 1,
                    'INP': 1,
                    'DSP': 1,
                    'MUL': 1,
                    'DIV': 1,
                    'STP': 0}
```

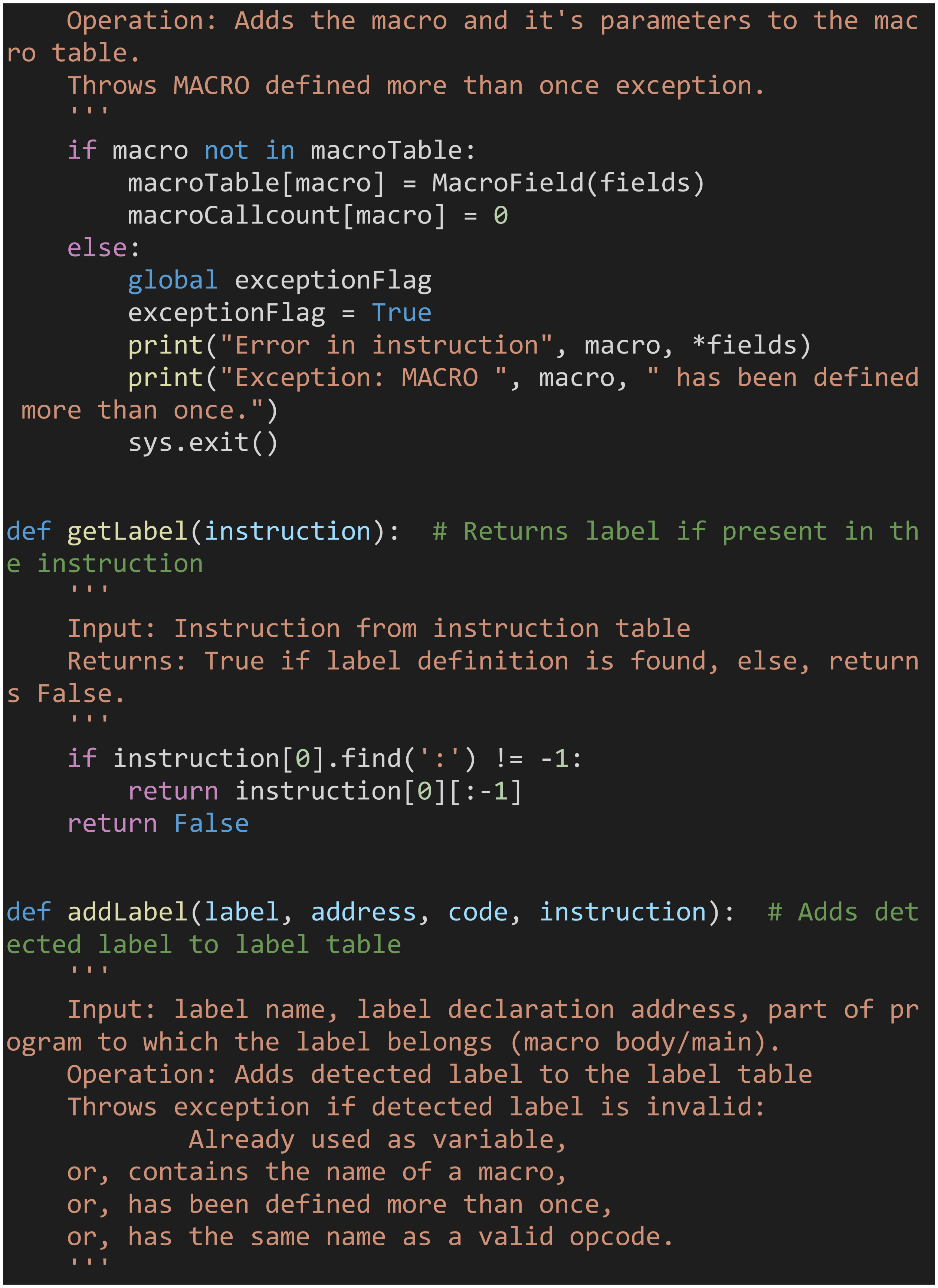
Code for Assembler

```python
from Opcodes import opcodes, opcode_arguments
import copy
import sys

#########classes required#########
exceptionFlag = False


class LiteralField:
    def __init__(self, literal):
        self.value = literal.replace("'", "")
        self.size = 1
        i = 1
        # if the constant value is very large, allocate it m
ore memory spaces
        while(((2**((8*i)-1)//2)-
1) < abs(float(self.value))):
            self.size += 1
            i += 1
        self.physicalAdd = None

    def printThis(self):
        print("Value:", self.value, ", Size:", self.size,
              ", Physical Address:", self.physicalAdd)


class LabelField:
    # code = the function that the label belongs to main or
name of macro
    def __init__(self, virtualAdd, code):
        self.virtualAdd = virtualAdd
        self.physicalAdd = None
        self.code = code

    def printThis(self):
        print("V.Add:", self.virtualAdd, ", P.Add:",
              self.physicalAdd, ", Code:", self.code)
```

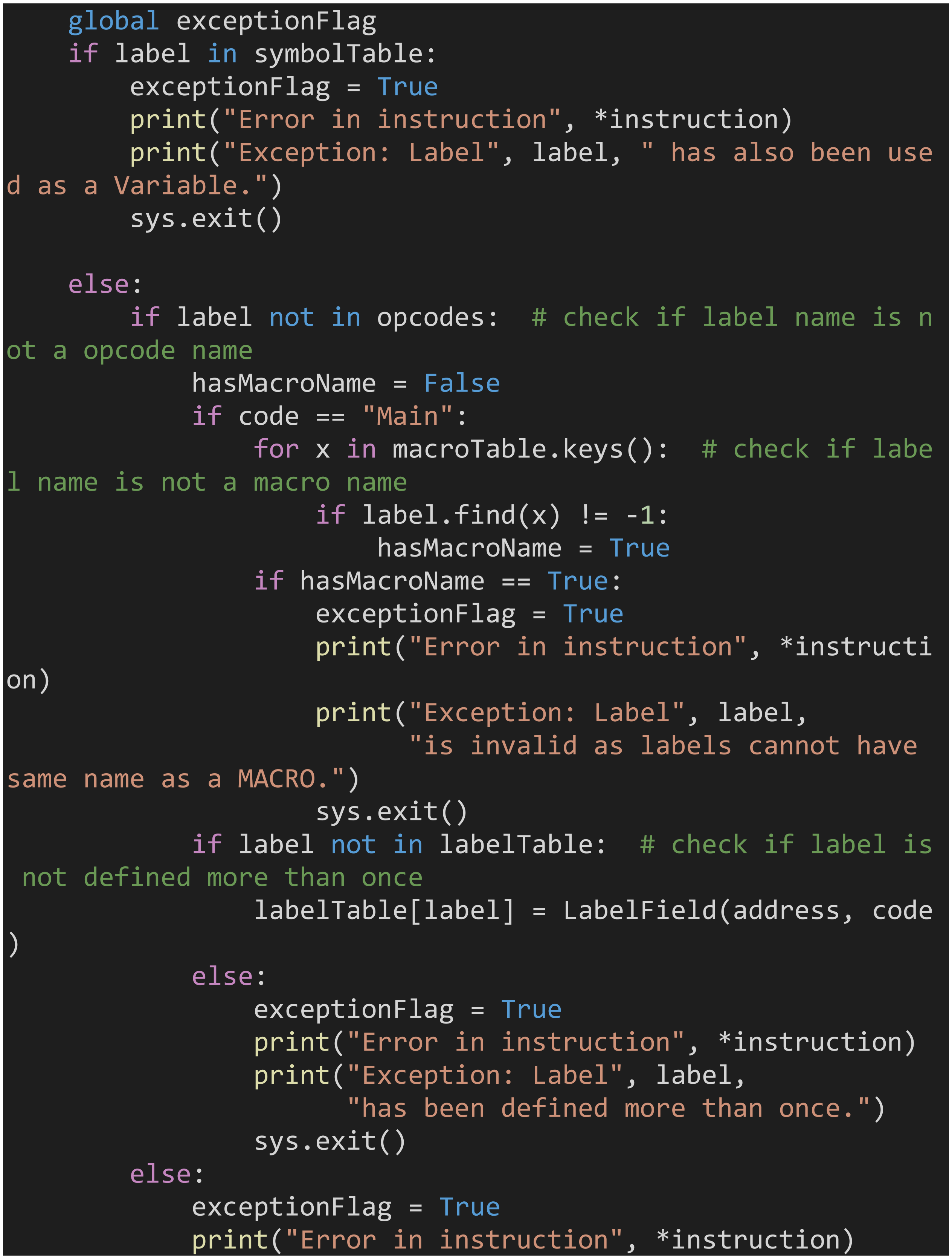
```python
class SymbolField:
    def __init__(self):
        self.physicalAdd = None
        self.status = 'undefined'

    def printThis(self):
        print("P.Add:", self.physicalAdd, ", Status:", str(s
elf.status))


class MacroField:
    def __init__(self, macroparameters):
        self.macroparameters = macroparameters
        self.instructionTable = []
        self.labels = []

    def printThis(self):
        print("Parameters:", str(self.macroparameters),
              ", Labels:", str(self.labels))
        print("Instruction Table:")
        for i in self.instructionTable:
            print(str(i))


#########Initialization#########
dataTable = {}
labelTable = {}
literalTable = {}
macroTable = {}
symbolTable = {}
instructionTable = []
macroCallcount = {}  # stores the number of calls for each m
acro present in the macro table
LoadAddress = False  # stores the physical address to load i
nstructions
instructions = []
num_ins = -1  # counter to count number of instructions
```

```
foundMacroDefinition = False  # flag to check if a macro is
being defined

#########Functions#########


def printTables():      # prints all the tables generated
    print("\nPrinting instruction table")
    printInstructionTable()
    print("\nPrinting macro table")
    printMacroTable()
    print("\nPrinting label table")
    printLabelTable()
    print("\nPrinting symbol table")
    printSymbolTable()
    print("\nPrinting data table")
    printDataTable()
    print("\nPrinting literal table")
    printLiteralTable()


def printMacroTable():
    for i in macroTable:
        print(i)
        macroTable[i].printThis()


def printDataTable():
    print(dataTable)


def printLabelTable():
    for i in labelTable:
        print(i)
        labelTable[i].printThis()


def printSymbolTable():
    for i in symbolTable:
        print(i)
        if(symbolTable[i].physicalAdd != None):
            symbolTable[i].physicalAdd = bin8(symbolTable[i]
.physicalAdd)
        symbolTable[i].printThis()


def printInstructionTable():
    for i in instructionTable:
        print(i)


def printLiteralTable():
    for i in literalTable:
        print(i)
        if(literalTable[i].physicalAdd != None):
            for k in range(0, len(literalTable[i].physicalAd
d)):
                literalTable[i].physicalAdd[k] = bin8(
                    literalTable[i].physicalAdd[k])
        literalTable[i].printThis()


def removeComments(instruction):
    '''
    Input: Single instruction in assembly language as a stri
ng
    Output: Instruction in assembly language as a string, wi
thout comments
    Parses the instruction passed as input to remove comment
s, that is,
    any text written beyond the ';' character.
    '''
    if(instruction.find(";") != (-1)):
        instruction = instruction[0:instruction.find(";")]
    return(instruction)


# returns 8 bit binary address
def bin8(x): return ''.join(reversed([str((x >> i) & 1) for
i in range(8)]))
```

```
def checkMacro(instruction):  # Check if a macro has been de
clared or it has ended
    '''
    Input: Instruction
    Operation: Checking for beginning and ending of Macro de
finition.
    '''
    if len(instruction) >= 2:
        if("MACRO" in instruction[1]):
            return True

    if("MEND" in instruction or "ENDM" in instruction):
        global exceptionFlag
        instruction = refine(instruction)
        labelsPresent = getLabel(instruction)
        if labelsPresent != False:
            if labelsPresent not in macroTable[name].labels:
                macroTable[name].labels.append(labelsPresent
)
                if len(instruction) == 2:
                    return False
            else:
                exceptionFlag = True
                print("Error in instruction", *instruction)
                # if the label is declared multiple times in
 a macro
                print("Exception: Label", labelsPresent,
                      "has been defined multiple times for m
acro", name)
                sys.exit()
        elif len(instruction) == 1:
            return False


def addMacro(macro, fields):  # Add macro to Macro table
    '''
    Input: Macro name and parameters.
```

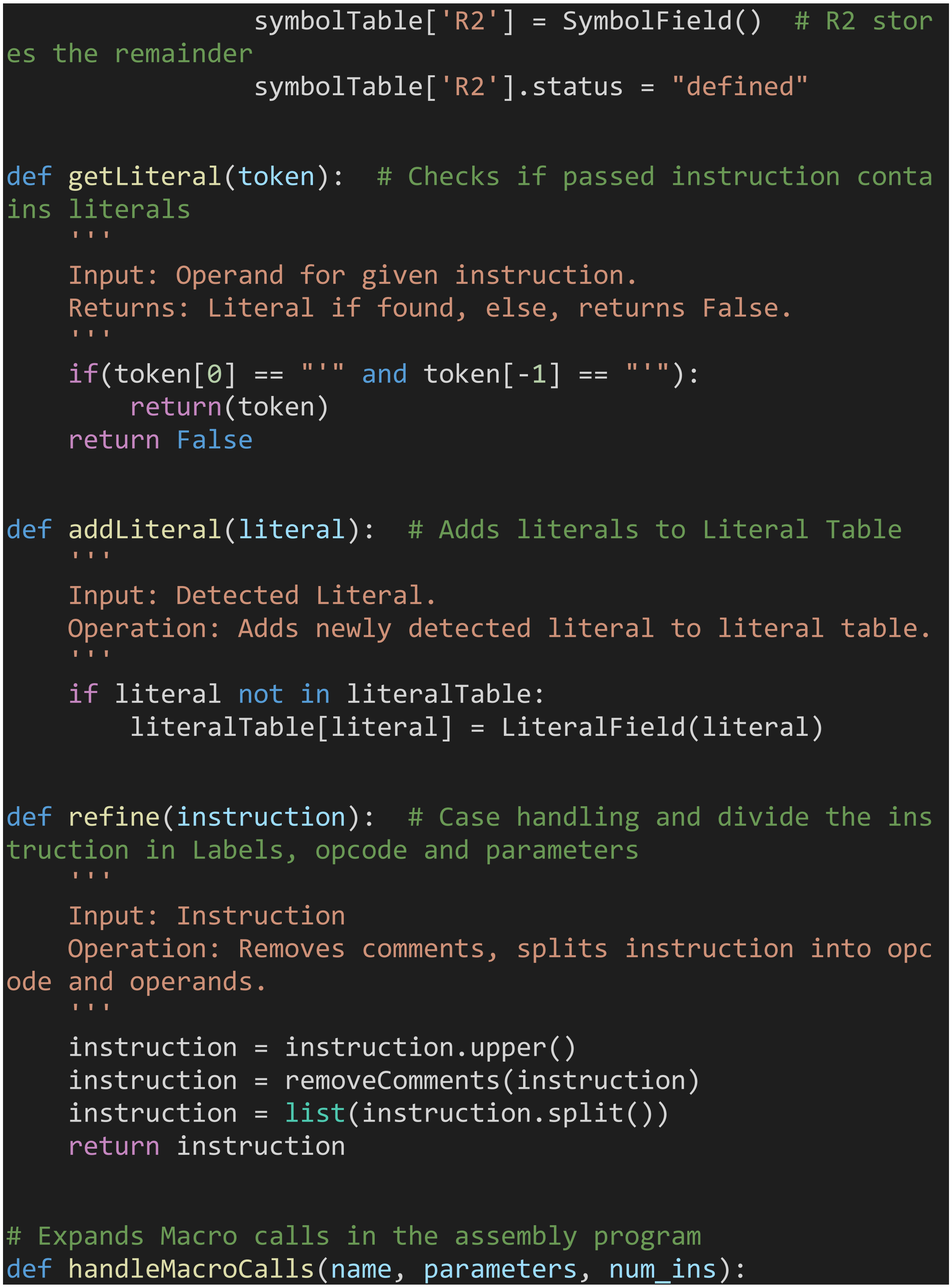


```
    Operation: Adds the macro and it's parameters to the mac
ro table.
    Throws MACRO defined more than once exception.
    '''
    if macro not in macroTable:
        macroTable[macro] = MacroField(fields)
        macroCallcount[macro] = 0
    else:
        global exceptionFlag
        exceptionFlag = True
        print("Error in instruction", macro, *fields)
        print("Exception: MACRO ", macro, " has been defined
 more than once.")
        sys.exit()


def getLabel(instruction):  # Returns label if present in th
e instruction
    '''
    Input: Instruction from instruction table
    Returns: True if label definition is found, else, return
s False.
    '''
    if instruction[0].find(':') != -1:
        return instruction[0][:-1]
    return False


def addLabel(label, address, code, instruction):  # Adds det
ected label to label table
    '''
    Input: label name, label declaration address, part of pr
ogram to which the label belongs (macro body/main).
    Operation: Adds detected label to the label table
    Throws exception if detected label is invalid:
            Already used as variable,
    or, contains the name of a macro,
    or, has been defined more than once,
    or, has the same name as a valid opcode.
    '''
```

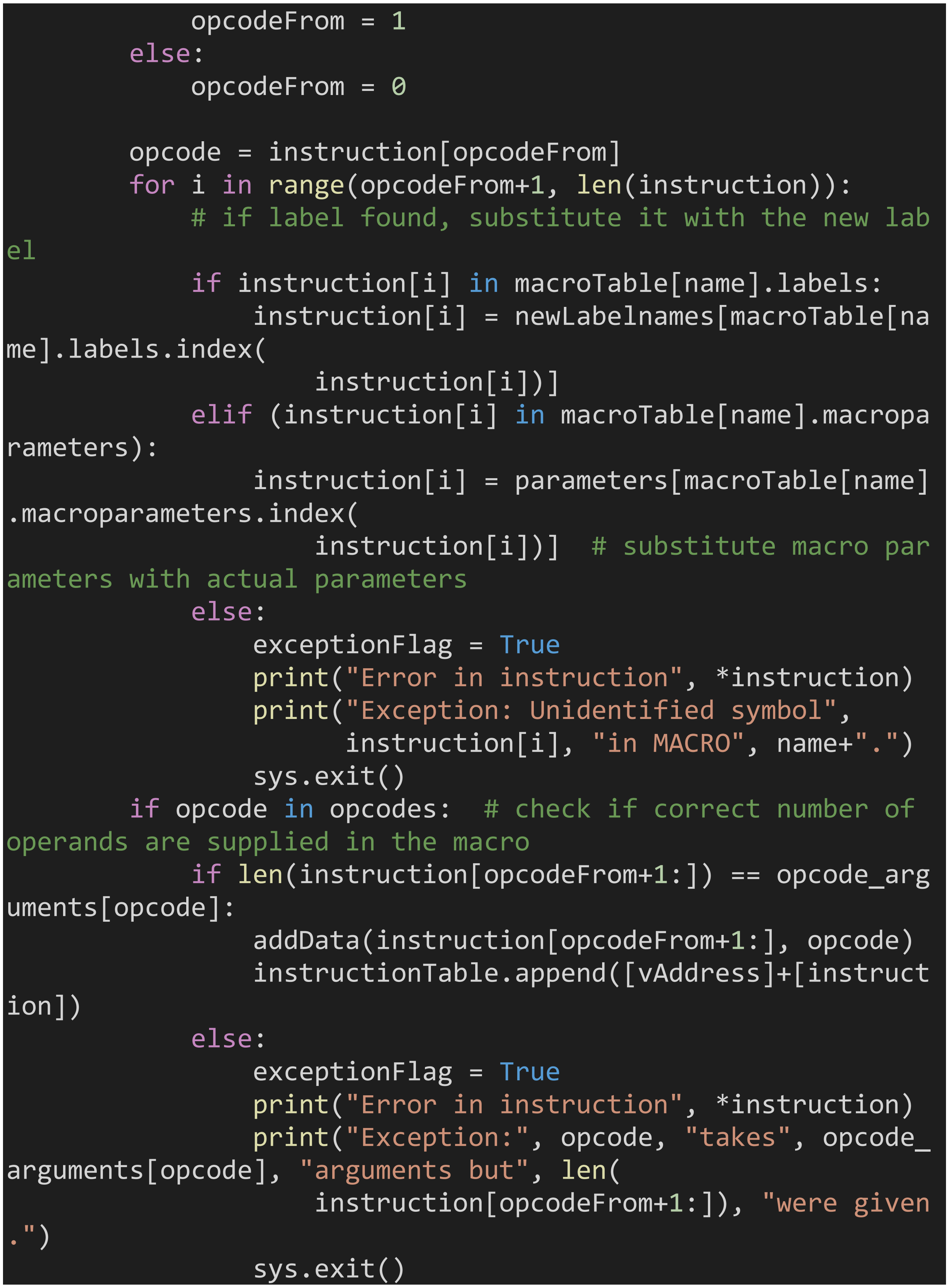


```
    global exceptionFlag
    if label in symbolTable:
        exceptionFlag = True
        print("Error in instruction", *instruction)
        print("Exception: Label", label, " has also been use
d as a Variable.")
        sys.exit()

    else:
        if label not in opcodes:  # check if label name is n
ot a opcode name
            hasMacroName = False
            if code == "Main":
                for x in macroTable.keys():  # check if labe
l name is not a macro name
                    if label.find(x) != -1:
                        hasMacroName = True
                if hasMacroName == True:
                    exceptionFlag = True
                    print("Error in instruction", *instructi
on)
                    print("Exception: Label", label,
                          "is invalid as labels cannot have
same name as a MACRO.")
                    sys.exit()
            if label not in labelTable:  # check if label is
 not defined more than once
                labelTable[label] = LabelField(address, code
)
            else:
                exceptionFlag = True
                print("Error in instruction", *instruction)
                print("Exception: Label", label,
                      "has been defined more than once.")
                sys.exit()
        else:
            exceptionFlag = True
            print("Error in instruction", *instruction)
            print("Exception: Label cannot be an opcode name
.",
                  label, "is an opcode name.")
            sys.exit()


def addData(parameters, opcode):  # Adds the parameters in t
he datatable and literal table
    '''
    Input: Opcode and operands following the opcode for give
n instruction.
    Operation: Adds operands to the dataTable/ literalTable/
 symbolTable.
    Throws : Memory Address out of bounds error.
    '''
    global exceptionFlag
    for i in range(len(parameters)):
        # if literal found, add it to the literal table
        x = getLiteral(parameters[i])
        if x != False:
            addLiteral(x)
        else:
            # as for branch, labels will be supplied which a
re already handled
            if (opcode in ["INP", "ADD", "SUB", "LAC", "SAC"
, "DSP", "MUL", "DIV"]):
                try:
                    parameters[i] = int(parameters[i])
                    if parameters[i] not in dataTable:
                        if -1 < parameters[i] < 256:
                            if opcode == "INP" or opcode ==
"SAC":
                                dataTable[parameters[i]] = "
defined"
                            # if opcode=="SAC" and len(instr
uctionTable)>0:     ##if we consider that cla should result
to 0 value, in which case store 0 would be a defined address
                            #   if instructionTable[-1][-
1]=="CLA":
```

```
                            #       dataTable[i]="defined"
                            else:
                                dataTable[parameters[i]] = "
undefined"
                        else:
                            exceptionFlag = True
                            print("Error in instruction", op
code, *parameters)
                            print(
                                "Exception: Address supplied
 exceeds memory limit. It should be lesser than 8 bits, that
 is 256. Address", i, "is not a valid address.")
                            sys.exit()
                except:
                    if (opcode in ["INP", "ADD", "SUB", "LAC
", "SAC", "DSP", "MUL", "DIV"]):
                        if parameters[i] not in labelTable:
                            if parameters[i] not in symbolTa
ble:
                                symbolTable[parameters[i]] =
 SymbolField()
                                if(opcode in ["INP", "SAC"])
:
                                    symbolTable[parameters[i
]
                                                ].status = "
defined"
                        else:
                            exceptionFlag = True
                            print("Error in instruction", op
code, *parameters)
                            print("Exception:", opcode,
                                  "cannot take labels as par
ameters")
                            sys.exit()
            if(opcode == "DIV"):
                symbolTable['R1'] = SymbolField()  # R1 stor
es the quotient
                symbolTable['R1'].status = "defined"
```

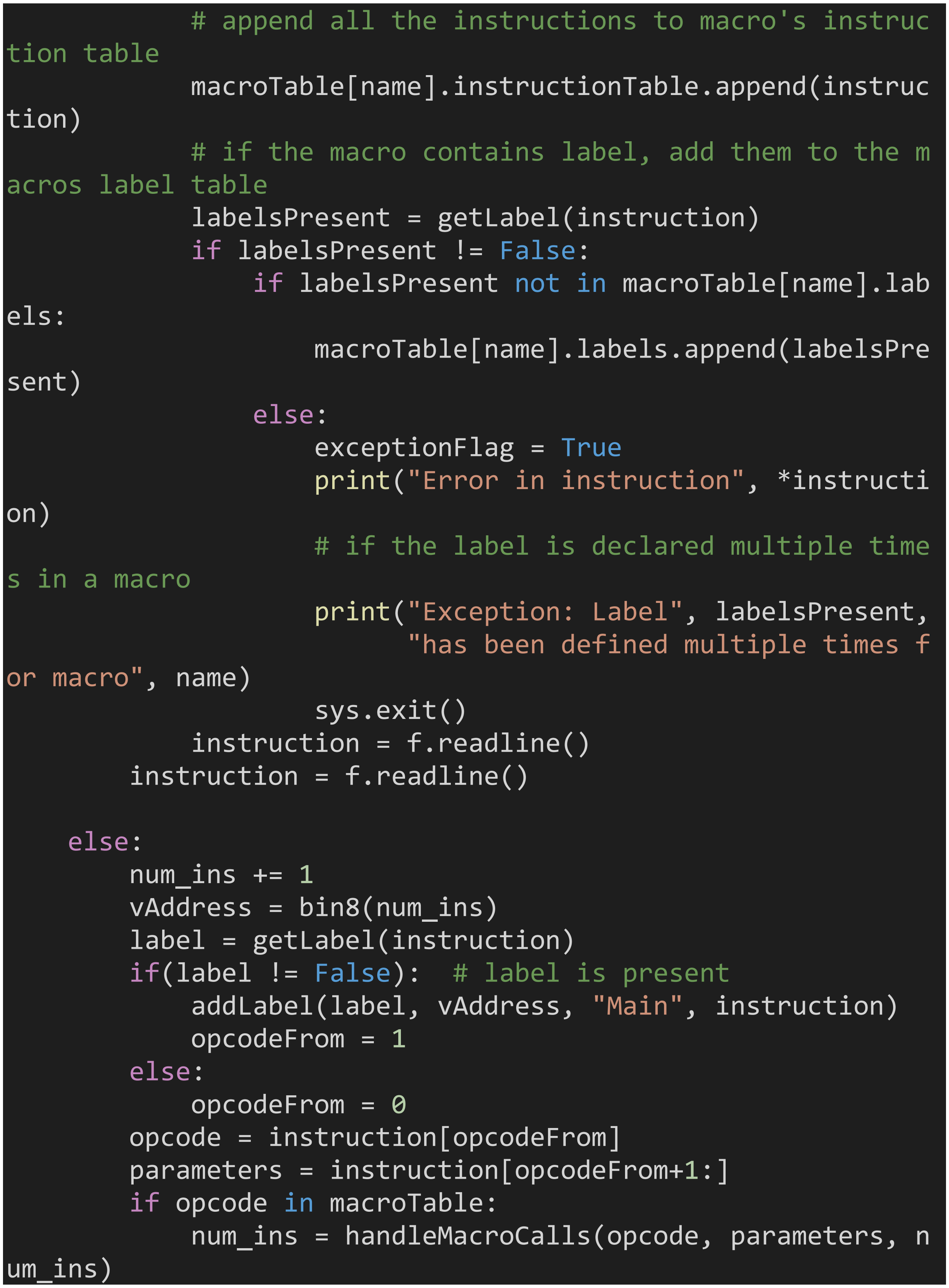


```python
                symbolTable['R2'] = SymbolField()  # R2 stor
es the remainder
                symbolTable['R2'].status = "defined"


def getLiteral(token):  # Checks if passed instruction conta
ins literals
    '''
    Input: Operand for given instruction.
    Returns: Literal if found, else, returns False.
    '''
    if(token[0] == "'" and token[-1] == "'"):
        return(token)
    return False


def addLiteral(literal):  # Adds literals to Literal Table
    '''
    Input: Detected Literal.
    Operation: Adds newly detected literal to literal table.
    '''
    if literal not in literalTable:
        literalTable[literal] = LiteralField(literal)


def refine(instruction):  # Case handling and divide the ins
truction in Labels, opcode and parameters
    '''
    Input: Instruction
    Operation: Removes comments, splits instruction into opc
ode and operands.
    '''
    instruction = instruction.upper()
    instruction = removeComments(instruction)
    instruction = list(instruction.split())
    return instruction


# Expands Macro calls in the assembly program
def handleMacroCalls(name, parameters, num_ins):
    '''
    Input: Macro name, macro parameters and number of instru
ctions.
    Operation: Maps actual and formal parameters and expands
 the macro call in the instruction table.
    '''
    global exceptionFlag
    macroCallcount[name] += 1
    newLabelnames = []
    labelsUsed = []
    # creates new label name set for the macro of the form m
acroName-
    for i in macroTable[name].labels:
        newLabelnames.append(str(name)+str(i)+str(macroCallc
ount[name]))
        labelsUsed.append(False)

    copiedInstructionset = copy.deepcopy(macroTable[name].in
structionTable)
    if len(parameters) != len(macroTable[name].macroparamete
rs):
        exceptionFlag = True
        print("Error in instruction", name, *parameters)
        print("Exception: Macro", name, "takes", len(
            macroTable[name].macroparameters), "parameters b
ut", len(parameters), "were given.")
        sys.exit()
    for instruction in copiedInstructionset:
        vAddress = bin8(num_ins)
        label = getLabel(instruction)
        if label != False:
            instruction[0] = newLabelnames[macroTable[name].
labels.index(
                label)]+":"
            addLabel(newLabelnames[macroTable[name].labels.i
ndex(
                label)], vAddress, name, instruction)
            labelsUsed[macroTable[name].labels.index(label)]
 = True
```

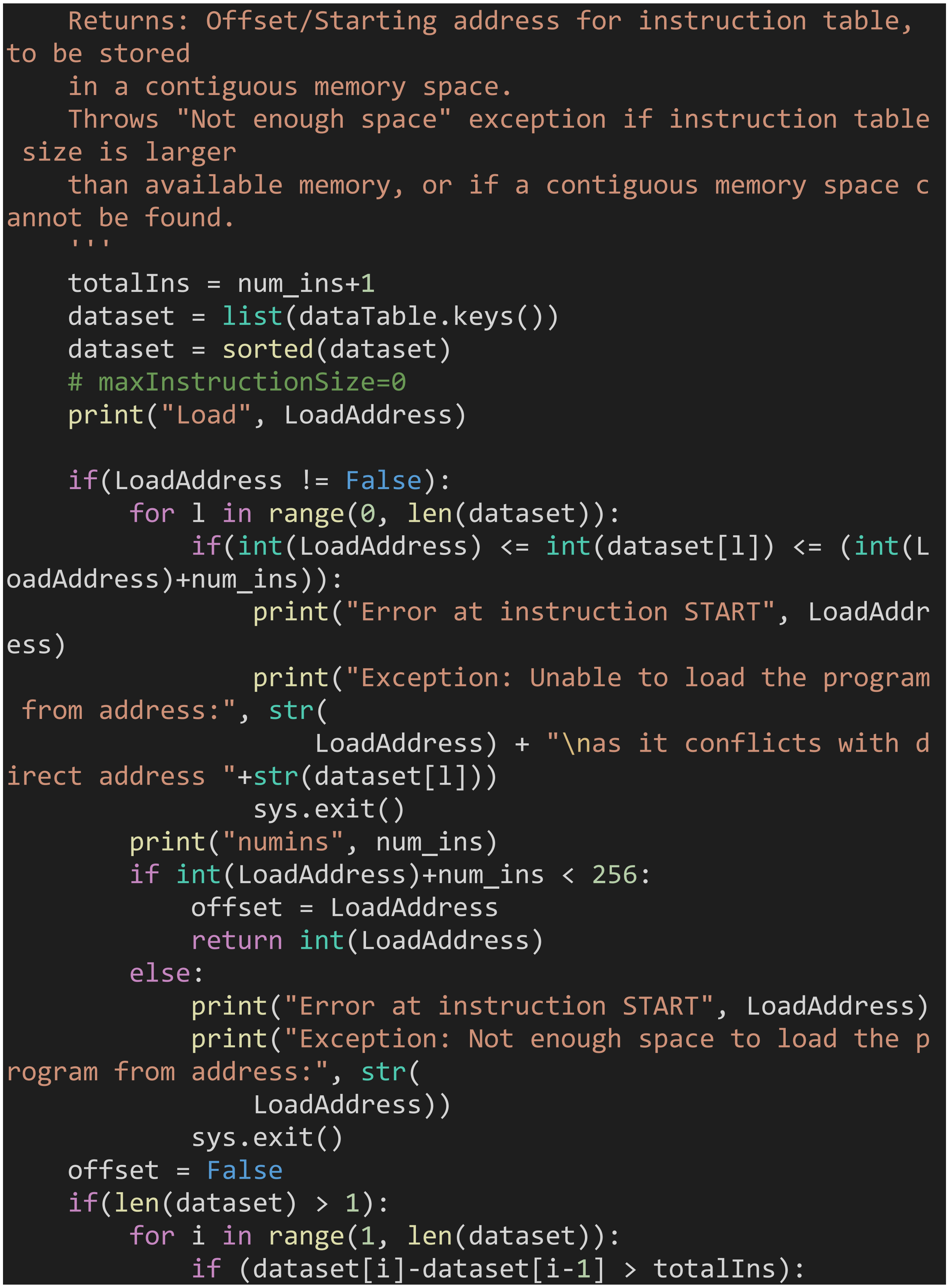


```
            opcodeFrom = 1
        else:
            opcodeFrom = 0

        opcode = instruction[opcodeFrom]
        for i in range(opcodeFrom+1, len(instruction)):
            # if label found, substitute it with the new lab
el
            if instruction[i] in macroTable[name].labels:
                instruction[i] = newLabelnames[macroTable[na
me].labels.index(
                    instruction[i])]
            elif (instruction[i] in macroTable[name].macropa
rameters):
                instruction[i] = parameters[macroTable[name]
.macroparameters.index(
                    instruction[i])]  # substitute macro par
ameters with actual parameters
            else:
                exceptionFlag = True
                print("Error in instruction", *instruction)
                print("Exception: Unidentified symbol",
                      instruction[i], "in MACRO", name+".")
                sys.exit()
        if opcode in opcodes:  # check if correct number of
operands are supplied in the macro
            if len(instruction[opcodeFrom+1:]) == opcode_arg
uments[opcode]:
                addData(instruction[opcodeFrom+1:], opcode)
                instructionTable.append([vAddress]+[instruct
ion])
            else:
                exceptionFlag = True
                print("Error in instruction", *instruction)
                print("Exception:", opcode, "takes", opcode_
arguments[opcode], "arguments but", len(
                    instruction[opcodeFrom+1:]), "were given
.")
                sys.exit()
```

```python
        else:
            exceptionFlag = True
            print("Error in instruction", *instruction)
            print("Exception:", opcode, "is not a valid opco
de name.")
            sys.exit()
        num_ins += 1
    for i in range(len(labelsUsed)):
        if (labelsUsed[i] == False):
            addLabel(newLabelnames[i], bin8(num_ins),
                     name, [newLabelnames[i], 'MEND'])

    return num_ins-1


#########Main code#########
path = input("Enter file path: ")
# Opening file and initializing line, symbol, literal and op
code
path = './CAO_Project/Sample_Inputs/'+path
f = open(path+".txt", 'r')
endEncountered = False
instruction = f.readline()
while instruction:
    if instruction == "END" or instruction == "END\n":  # if
 end is encountered, stop execution
        endEncountered = True
        break
    if(len(instruction) == 1):  # check for empty lines
        instruction = f.readline()
        continue
    instruction = refine(instruction)
    if len(instruction) == 0:  # check if the line is just a
 comment
        instruction = f.readline()
        continue
    if instruction[0] == 'START':
        if(len(instruction) == 2):
            LoadAddress = instruction[1]
    instruction = f.readline()
    continue

  # Add macros to macro table
  # check if instruction is a macro
  foundMacroDefinition = checkMacro(instruction)
  if(foundMacroDefinition):
    s = ''
    name = instruction[0]
    for i in range(2, len(instruction)):  # Find out all
 the parameters of the macro
      s = s+instruction[i]
    s = s.replace(' ', '')
    parameters = list(s.split(','))
    addMacro(instruction[0], parameters)
    instruction = f.readline()
    while(checkMacro(instruction) != False):
      if (not instruction):  # If end of file appears
without getting MEND or END
        # exceptionFlag=True
        print("Exception: MEND/ENDM not specified af
ter Macro definition", name)
        sys.exit()
      if(len(instruction) == 1):  # check for empty li
nes
        instruction = f.readline()
        continue
      # If another macro is declared or end of file ap
pears
      if ("MACRO" in instruction or "END" in instructi
on):
        print("Exception: MEND/ENDM not specified af
ter Macro definition", name)
        sys.exit()
      instruction = refine(instruction)
      if len(instruction) == 0:  # check if the line i
s just a comment
        instruction = f.readline()
        continue
```

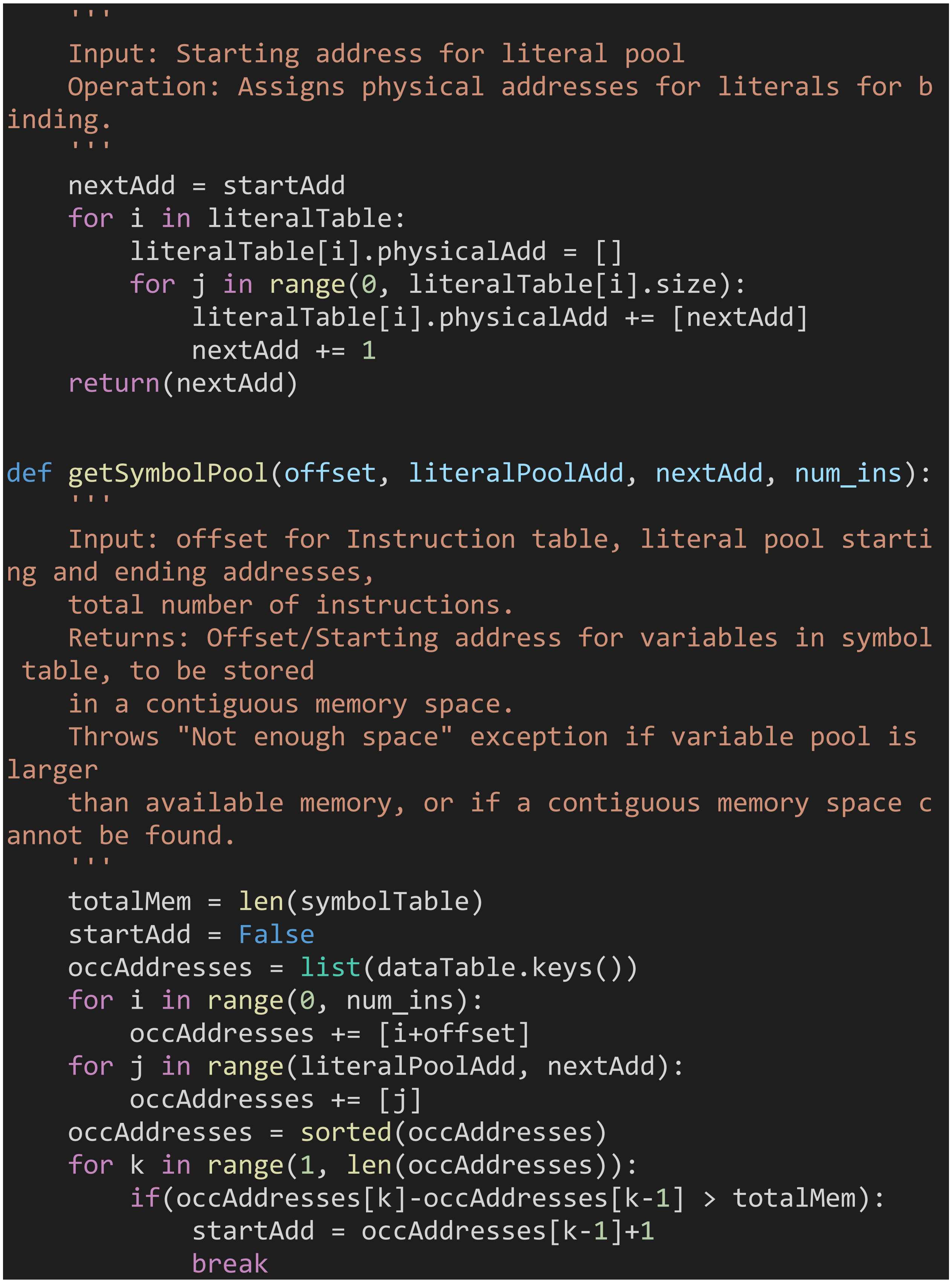

```
            # append all the instructions to macro's instruc
tion table
            macroTable[name].instructionTable.append(instruc
tion)
            # if the macro contains label, add them to the m
acros label table
            labelsPresent = getLabel(instruction)
            if labelsPresent != False:
                if labelsPresent not in macroTable[name].lab
els:
                    macroTable[name].labels.append(labelsPre
sent)
                else:
                    exceptionFlag = True
                    print("Error in instruction", *instructi
on)
                    # if the label is declared multiple time
s in a macro
                    print("Exception: Label", labelsPresent,
                          "has been defined multiple times f
or macro", name)
                    sys.exit()
            instruction = f.readline()
        instruction = f.readline()

    else:
        num_ins += 1
        vAddress = bin8(num_ins)
        label = getLabel(instruction)
        if(label != False):  # label is present
            addLabel(label, vAddress, "Main", instruction)
            opcodeFrom = 1
        else:
            opcodeFrom = 0
        opcode = instruction[opcodeFrom]
        parameters = instruction[opcodeFrom+1:]
        if opcode in macroTable:
            num_ins = handleMacroCalls(opcode, parameters, n
um_ins)
```

```python
        elif opcode in opcodes:
            if len(parameters) == opcode_arguments[opcode]:
                addData(parameters, opcode)
                instructionTable.append([vAddress]+[instruct
ion])
            else:
                exceptionFlag = True
                print("Error in instruction", *instruction)
                print("Exception: Opcode", opcode, "takes",
                      opcode_arguments[opcode], "arguments b
ut", len(parameters), "were given.")
                sys.exit()
        else:
            exceptionFlag = True
            print("Error in instruction", *instruction)
            print("Exception:", opcode, "is not a valid opco
de or a macro name.")
            sys.exit()
        instruction = f.readline()

if endEncountered == False:
    exceptionFlag = True
    print("Exception: END of program not found. Please decla
re 'END' command at the end of the assembly program.")
    sys.exit()
if exceptionFlag == False:
    print('######## SUCCESS: First pass ended successfully #
#######')
    num_ins += 1
printTables()


########################SECOND PASS######################
def getOffset(num_ins):
    '''
    Input parameters: Number of instructions present in inst
ruction table.
```

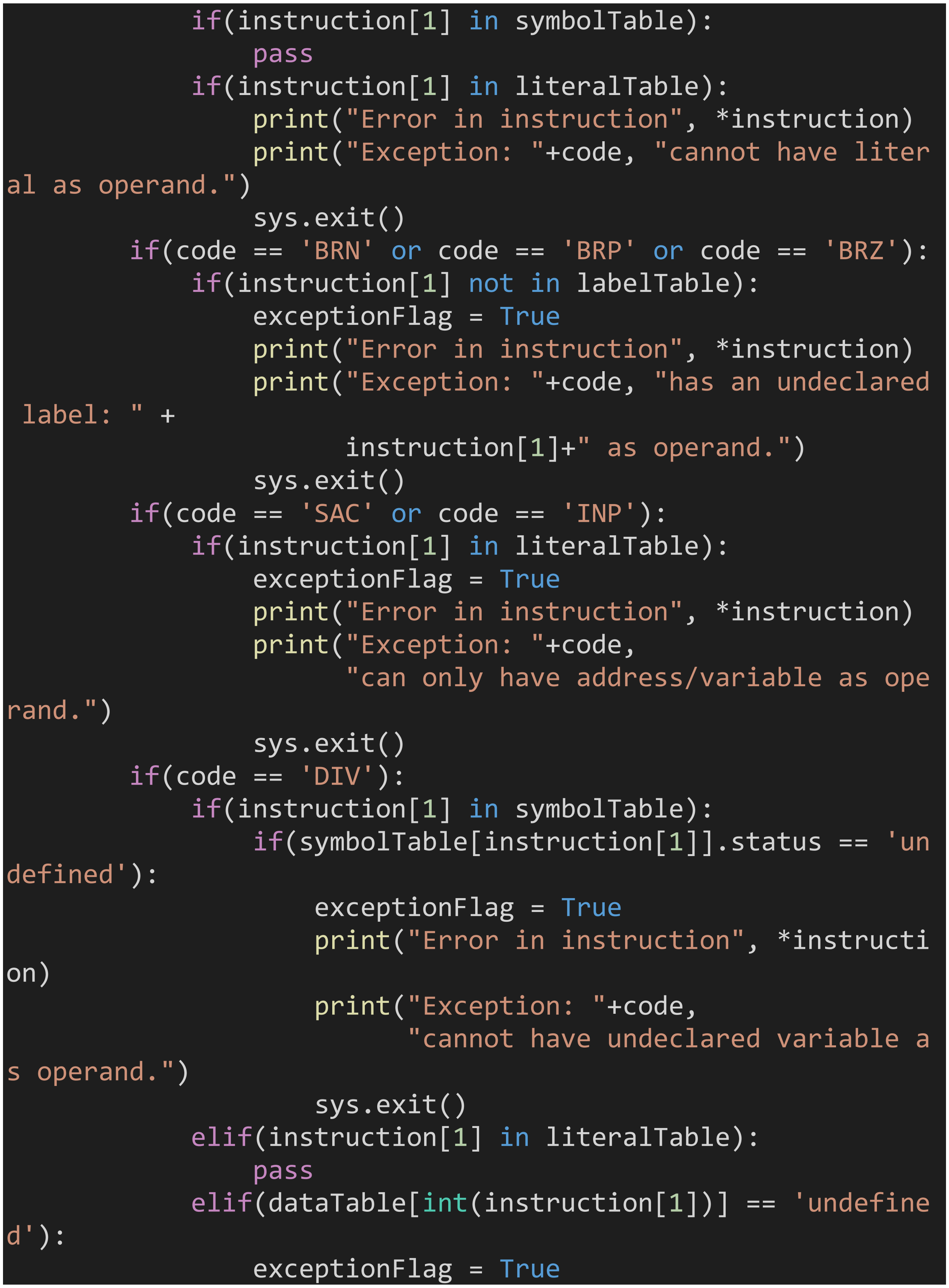


```
    Returns: Offset/Starting address for instruction table,
to be stored
    in a contiguous memory space.
    Throws "Not enough space" exception if instruction table
 size is larger
    than available memory, or if a contiguous memory space c
annot be found.
    '''
    totalIns = num_ins+1
    dataset = list(dataTable.keys())
    dataset = sorted(dataset)
    # maxInstructionSize=0
    print("Load", LoadAddress)

    if(LoadAddress != False):
        for l in range(0, len(dataset)):
            if(int(LoadAddress) <= int(dataset[l]) <= (int(L
oadAddress)+num_ins)):
                print("Error at instruction START", LoadAddr
ess)
                print("Exception: Unable to load the program
 from address:", str(
                    LoadAddress) + "\nas it conflicts with d
irect address "+str(dataset[l]))
                sys.exit()
        print("numins", num_ins)
        if int(LoadAddress)+num_ins < 256:
            offset = LoadAddress
            return int(LoadAddress)
        else:
            print("Error at instruction START", LoadAddress)
            print("Exception: Not enough space to load the p
rogram from address:", str(
                LoadAddress))
            sys.exit()
    offset = False
    if(len(dataset) > 1):
        for i in range(1, len(dataset)):
            if (dataset[i]-dataset[i-1] > totalIns):
                offset = dataset[i-1]+1
                break
    if(len(dataset) == 1):
        if((dataset[-1]+num_ins+1) < 256):
            offset = dataset[-1]+1
    if(len(dataset) == 0):
        offset = 0
        return(offset)
    if (offset == False and len(dataset) != 0):
        if((dataset[-1]+num_ins+1) < 256):
            offset = dataset[-1]+1
    if offset == False:
        global exceptionFlag
        exceptionFlag = True
        print("Exception: Not enough space for complete prog
ram")
        sys.exit()
    else:
        return offset


def addOffset(offset):
    '''
    Input: Offset calculated for binding of instructions and
 labels.
    Operation: Maps the instructions and labels in Instructi
on Table and Label Table to
    physical addresses by adding offset
    '''
    for i in range(0, len(instructionTable)):
        instructionTable[i][0] = bin8(int(instructionTable[i
][0], 2)+offset)
    for j in labelTable:
        labelTable[j].physicalAdd = bin8(
            int(labelTable[j].virtualAdd, 2)+offset)


def getLiteralPool(offset, num_ins):
    '''
```

```python
    Input: offset for Instruction table and total number of
instructions.
    Returns: Offset/Starting address for literal pool, to be
 stored
    in a contiguous memory space.
    Throws "Not enough space" exception if literal pool is l
arger
    than available memory, or if a contiguous memory space c
annot be found.
    '''
    totalMem = 0
    startAdd = False
    for i in literalTable:
        totalMem += literalTable[i].size
    occAddresses = list(dataTable.keys())
    for j in range(0, num_ins):
        occAddresses += [j+offset]
    occAddresses = sorted(occAddresses)
    if(totalMem < occAddresses[0]):
        startAdd = occAddresses[0]-totalMem
        return(startAdd)
    for k in range(1, len(occAddresses)):
        if (occAddresses[k]-occAddresses[k-1] > totalMem):
            startAdd = occAddresses[k-1]+1
            break
    if startAdd == False:
        if((occAddresses[-1]+totalMem+1) < 256):
            startAdd = occAddresses[-1]+1
    if startAdd == False:
        global exceptionFlag
        exceptionFlag = True
        print("Exception: Not enough space for complete prog
ram")
        sys.exit()
    else:
        return startAdd


def assignLiteralPool(startAdd):
```



```
    '''
    Input: Starting address for literal pool
    Operation: Assigns physical addresses for literals for b
inding.
    '''
    nextAdd = startAdd
    for i in literalTable:
        literalTable[i].physicalAdd = []
        for j in range(0, literalTable[i].size):
            literalTable[i].physicalAdd += [nextAdd]
            nextAdd += 1
    return(nextAdd)


def getSymbolPool(offset, literalPoolAdd, nextAdd, num_ins):
    '''
    Input: offset for Instruction table, literal pool starti
ng and ending addresses,
    total number of instructions.
    Returns: Offset/Starting address for variables in symbol
 table, to be stored
    in a contiguous memory space.
    Throws "Not enough space" exception if variable pool is
larger
    than available memory, or if a contiguous memory space c
annot be found.
    '''
    totalMem = len(symbolTable)
    startAdd = False
    occAddresses = list(dataTable.keys())
    for i in range(0, num_ins):
        occAddresses += [i+offset]
    for j in range(literalPoolAdd, nextAdd):
        occAddresses += [j]
    occAddresses = sorted(occAddresses)
    for k in range(1, len(occAddresses)):
        if(occAddresses[k]-occAddresses[k-1] > totalMem):
            startAdd = occAddresses[k-1]+1
            break
    if startAdd == False:
        if((occAddresses[-1]+totalMem+1) < 256):
            startAdd = occAddresses[-1]+1
    if startAdd == False:
        global exceptionFlag
        exceptionFlag = True
        print("Exception: Not enough space for complete prog
ram")
        sys.exit()
    else:
        return(startAdd)


def assignSymbolPool(startAdd):
    '''
    Input: Starting address for variable pool
    Operation: Assigns physical addresses for variables for
binding.
    '''
    nextAdd = startAdd
    for i in symbolTable:
        symbolTable[i].physicalAdd = nextAdd
        nextAdd += 1
    return(nextAdd)


def removeLabelDefinitions():
    '''
    Operation: Removes label declarations from the instructi
on table
    for conversion to machine language.
    '''
    for i in range(0, len(instructionTable)):
        if(instructionTable[i][1][0].find(":") != (-1)):
            del instructionTable[i][1][0]


def checkOperands():
    '''
    Operation: Checks validity of operands corresponding to
opcodes.
    ADD, MUL, LAC, SUB: Only have defined variables/addresse
s or literals.
    DSP: Only has defined variable/address.
    BRN, BRP, BRZ: Only have defined label.
    SAC, INP: Only have defined/undefined variables/addresse
s
    DIV: Only has first operand as defined variable/address
or literal, second and third operands as
    defined/undefined variables/addresses
    '''
    global exceptionFlag
    for i in range(0, len(instructionTable)):
        instruction = instructionTable[i][1]
        code = instructionTable[i][1][0]
        if(code == 'ADD' or code == 'MUL' or code == 'LAC' o
r code == 'DSP' or code == 'SUB'):
            if(instruction[1] in symbolTable):
                if(symbolTable[instruction[1]].status == 'un
defined'):
                    exceptionFlag = True
                    print("Error in instruction", *instructi
on)
                    print("Exception: "+code,
                          "cannot have undeclared variable a
s operand.")
                    sys.exit()
            elif(instruction[1] in literalTable):
                pass
            elif(dataTable[int(instruction[1])] == 'undefine
d'):
                exceptionFlag = True
                print("Error in instruction", *instruction)
                print("Exception: "+code,
                      "cannot have undefined address as oper
and.")
                sys.exit()
        if(code == "DSP"):
```

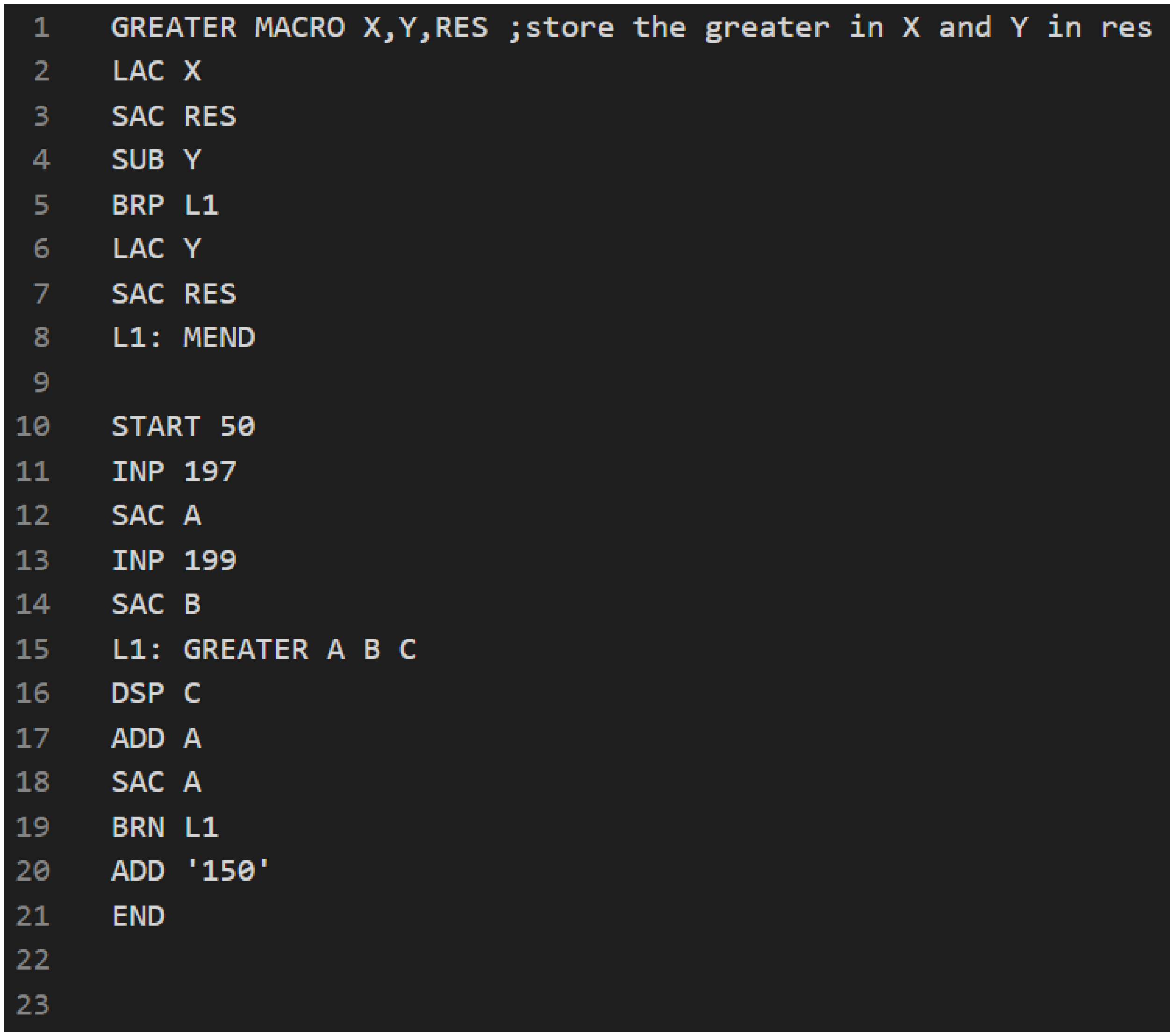


```
            if(instruction[1] in symbolTable):
                pass
            if(instruction[1] in literalTable):
                print("Error in instruction", *instruction)
                print("Exception: "+code, "cannot have liter
al as operand.")
                sys.exit()
        if(code == 'BRN' or code == 'BRP' or code == 'BRZ'):
            if(instruction[1] not in labelTable):
                exceptionFlag = True
                print("Error in instruction", *instruction)
                print("Exception: "+code, "has an undeclared
 label: " +
                      instruction[1]+" as operand.")
                sys.exit()
        if(code == 'SAC' or code == 'INP'):
            if(instruction[1] in literalTable):
                exceptionFlag = True
                print("Error in instruction", *instruction)
                print("Exception: "+code,
                      "can only have address/variable as ope
rand.")
                sys.exit()
        if(code == 'DIV'):
            if(instruction[1] in symbolTable):
                if(symbolTable[instruction[1]].status == 'un
defined'):
                    exceptionFlag = True
                    print("Error in instruction", *instructi
on)
                    print("Exception: "+code,
                          "cannot have undeclared variable a
s operand.")
                    sys.exit()
            elif(instruction[1] in literalTable):
                pass
            elif(dataTable[int(instruction[1])] == 'undefine
d'):
                exceptionFlag = True
                print("Error in instruction", *instruction)
                print("Exception: "+code, "should have first
 operand as address/variable or constant. " +
                      instruction[1]+" is an undefined addre
ss.")
                sys.exit()


def convertOpcodes():
    '''
    Operation: Convert opcodes in instruction table to machi
ne language.
    '''
    for i in range(0, len(instructionTable)):
        instruction = instructionTable[i][1]
        code = instructionTable[i][1][0]
        instructionTable[i][1][0] = opcodes[code]


def convertOperands():
    '''
    Operation: Convert operands to the physical adresses the
y are bound to.
    '''
    for i in range(0, len(instructionTable)):
        instruction = instructionTable[i][1]
        for k in range(1, len(instruction)):
            if(instruction[k] in labelTable):
                instructionTable[i][1][k] = labelTable[instr
uction[k]].physicalAdd
            elif(instruction[k] in literalTable):
                instructionTable[i][1][k] = bin8(
                    literalTable[instruction[k]].physicalAdd
[0])
            elif(instruction[k] in symbolTable):
                instructionTable[i][1][k] = bin8(
                    symbolTable[instruction[k]].physicalAdd)
            elif(int(instruction[k]) in dataTable):
                instructionTable[i][1][k] = bin8(int(instruc
tion[k]))
```

```
def writeToFile():
    '''
    Operation: Write generated machine code to text file nam
ed:
    <sample_file>_output.txt
    Splits machine code into blocks of four bits for readabi
lity.
    '''
    f = open(path+"_output.txt", "w+")
    for i in range(0, len(instructionTable)):
        instruction = instructionTable[i][1]
        s = ''
        s += instructionTable[i][0]
        for k in range(0, len(instruction)):
            s += instruction[k]
        l = (len(s))
        if(len(s) == 12):
            s += '00000000'
        l = len(s)
        block = 0
        machine_ins = ''
        machine_ins += s[block:block+8]+" "
        block += 8
        machine_ins += s[block:block+4]+" "
        block += 4
        machine_ins += s[block:]
        machine_ins += "\n"
        f.write(machine_ins)
        print(machine_ins)
    f.close()


############MAIN CODE##############
literalPoolAdd = 0
variablePoolAdd = 0
nextAdd = 0
offset = getOffset(num_ins)
addOffset(offset)
```

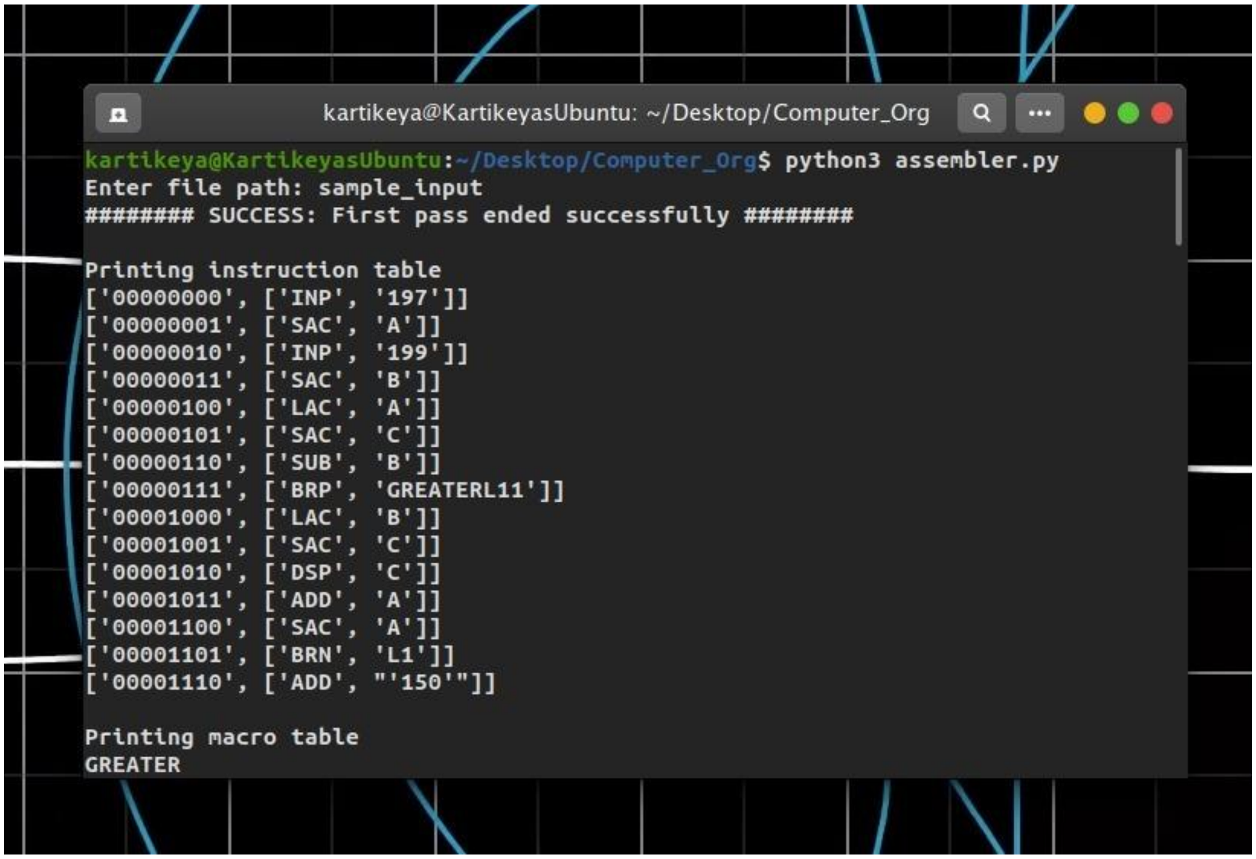
```
if(len(literalTable) != 0):
    literalPoolAdd = getLiteralPool(offset, num_ins)
    nextAdd = assignLiteralPool(literalPoolAdd)
if(len(symbolTable) != 0):
    variablePoolAdd = getSymbolPool(offset, literalPoolAdd,
nextAdd, num_ins)
    assignSymbolPool(variablePoolAdd)
removeLabelDefinitions()
checkOperands()

if exceptionFlag == False:
    print('######## SUCCESS: Second pass ended successfully
########')
    convertOpcodes()
    convertOperands()
    writeToFile()
    printTables()

# print(LoadAddress)
```

Input

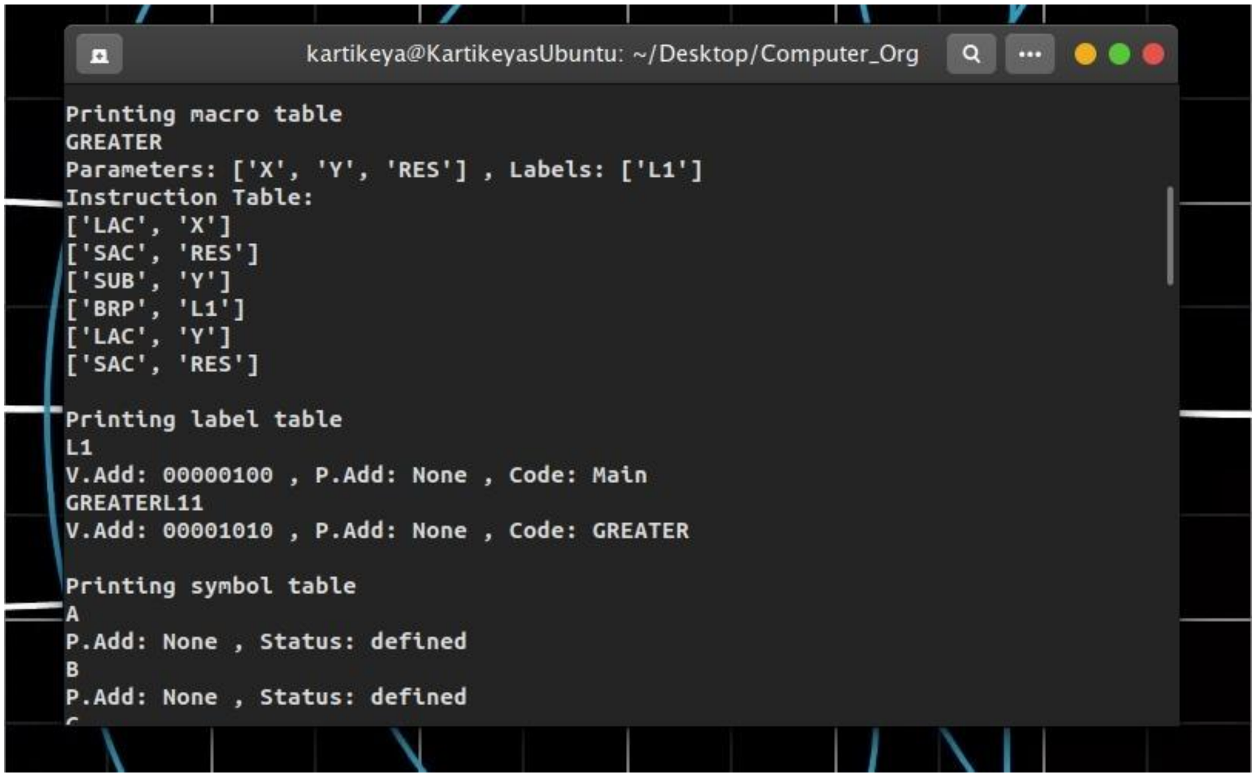


```
GREATER MACRO X,Y,RES ;store the greater in X and Y in res
LAC X
SAC RES
SUB Y
BRP L1
LAC Y
SAC RES
L1: MEND

START 50
INP 197
SAC A
INP 199
SAC B
L1: GREATER A B C
DSP C
ADD A
SAC A
BRN L1
ADD '150'
END
```

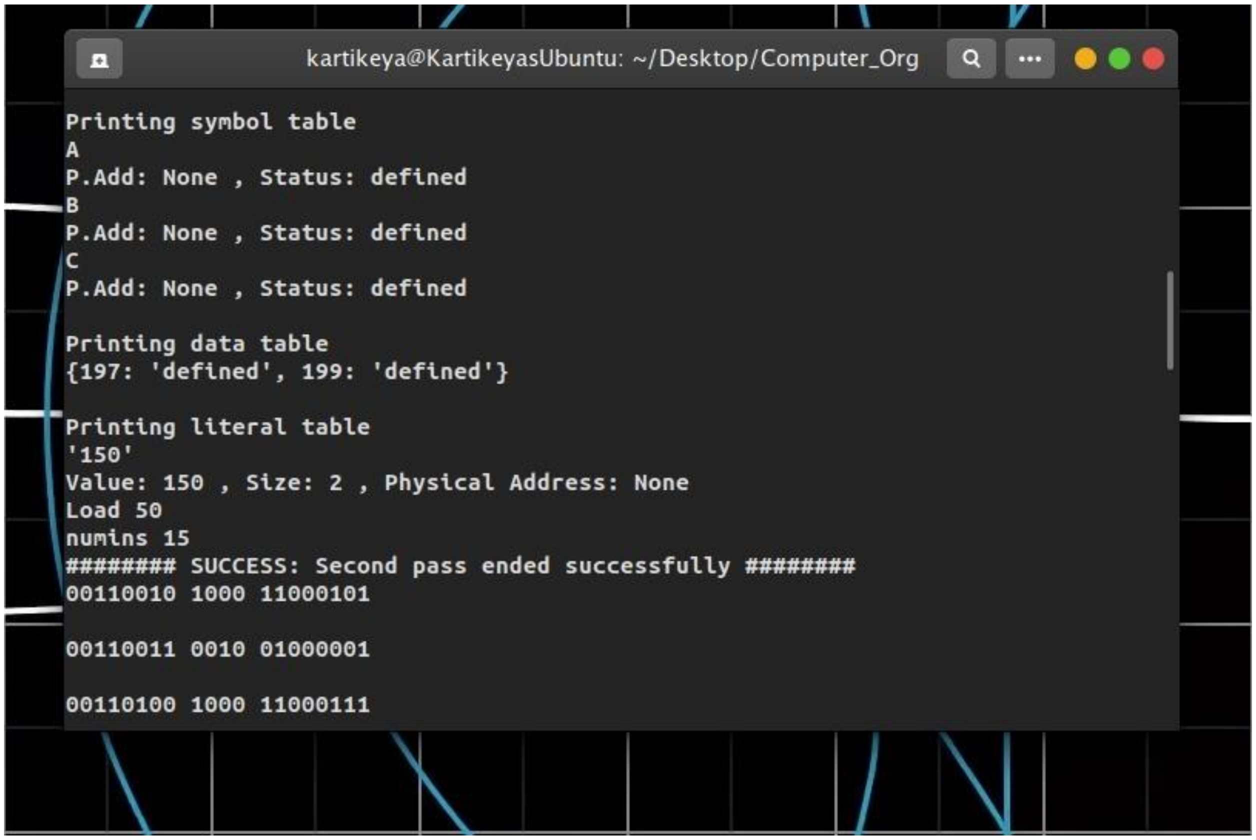
Output



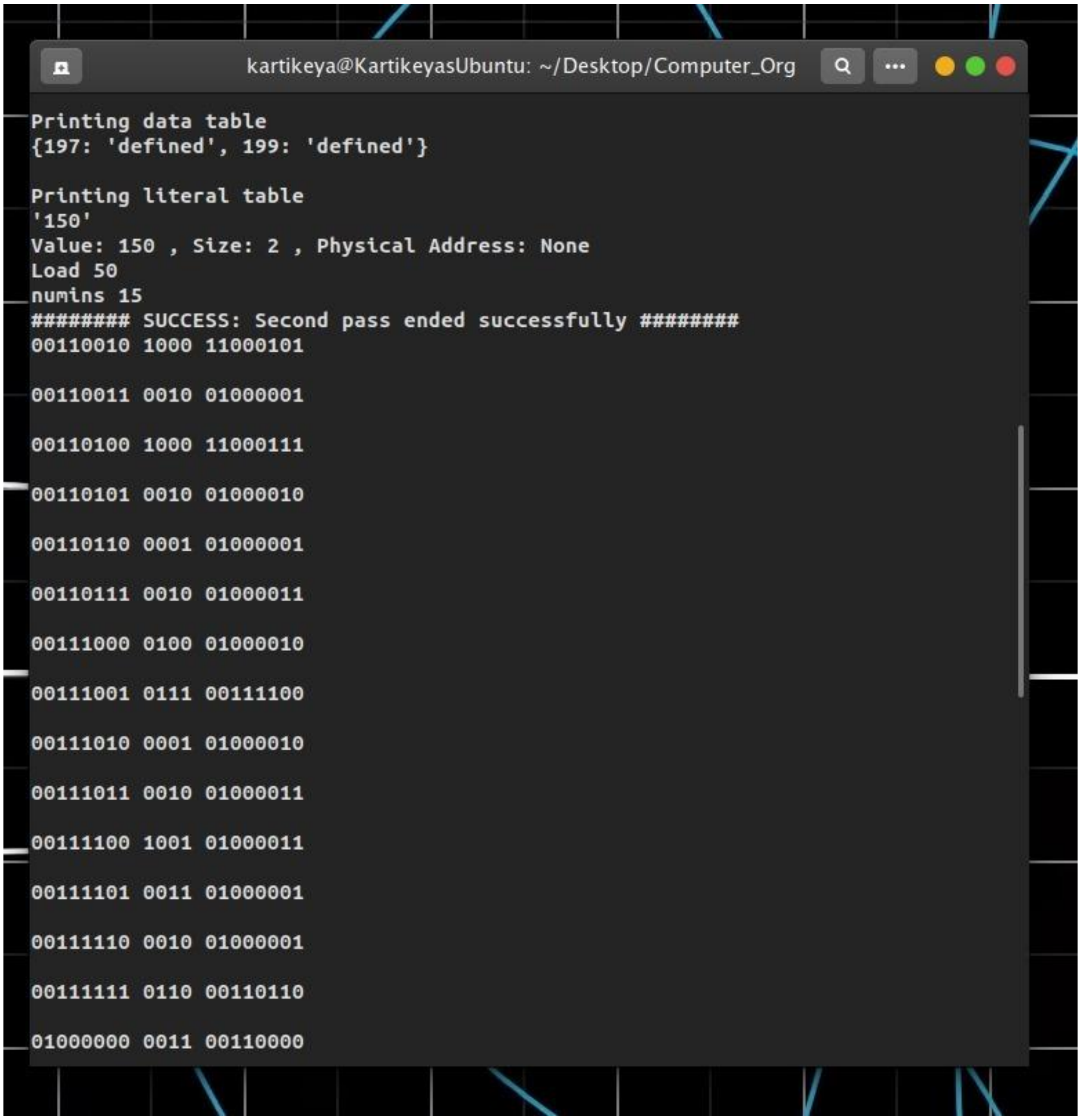
```
kartikeya@KartikeyasUbuntu:~/Desktop/Computer_Org$ python3 assembler.py
Enter file path: sample_input
######## SUCCESS: First pass ended successfully ########

Printing instruction table
['00000000', ['INP', '197']]
['00000001', ['SAC', 'A']]
['00000010', ['INP', '199']]
['00000011', ['SAC', 'B']]
['00000100', ['LAC', 'A']]
['00000101', ['SAC', 'C']]
['00000110', ['SUB', 'B']]
['00000111', ['BRP', 'GREATERL11']]
['00001000', ['LAC', 'B']]
['00001001', ['SAC', 'C']]
['00001010', ['DSP', 'C']]
['00001011', ['ADD', 'A']]
['00001100', ['SAC', 'A']]
['00001101', ['BRN', 'L1']]
['00001110', ['ADD', "'150'"]]

Printing macro table
GREATER
```



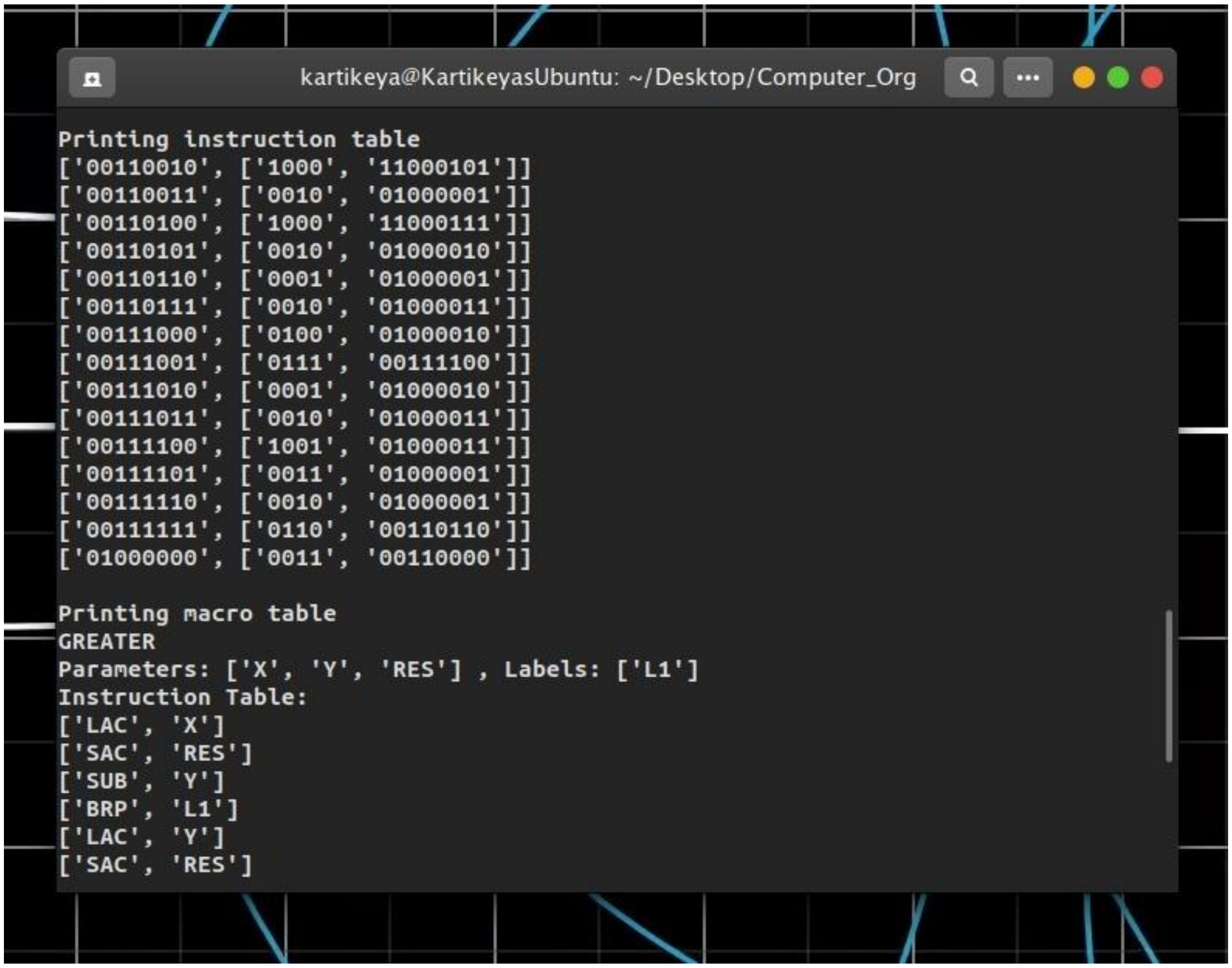
```
Printing macro table
GREATER
Parameters: ['X', 'Y', 'RES'] , Labels: ['L1']
Instruction Table:
['LAC', 'X']
['SAC', 'RES']
['SUB', 'Y']
['BRP', 'L1']
['LAC', 'Y']
['SAC', 'RES']

Printing label table
L1
V.Add: 00000100 , P.Add: None , Code: Main
GREATERL11
V.Add: 00001010 , P.Add: None , Code: GREATER

Printing symbol table
A
P.Add: None , Status: defined
B
P.Add: None , Status: defined
```



```
Printing symbol table
A
P.Add: None , Status: defined
B
P.Add: None , Status: defined
C
P.Add: None , Status: defined

Printing data table
{197: 'defined', 199: 'defined'}

Printing literal table
'150'
Value: 150 , Size: 2 , Physical Address: None
Load 50
numins 15
######## SUCCESS: Second pass ended successfully ########
00110010 1000 11000101

00110011 0010 01000001

00110100 1000 11000111
```

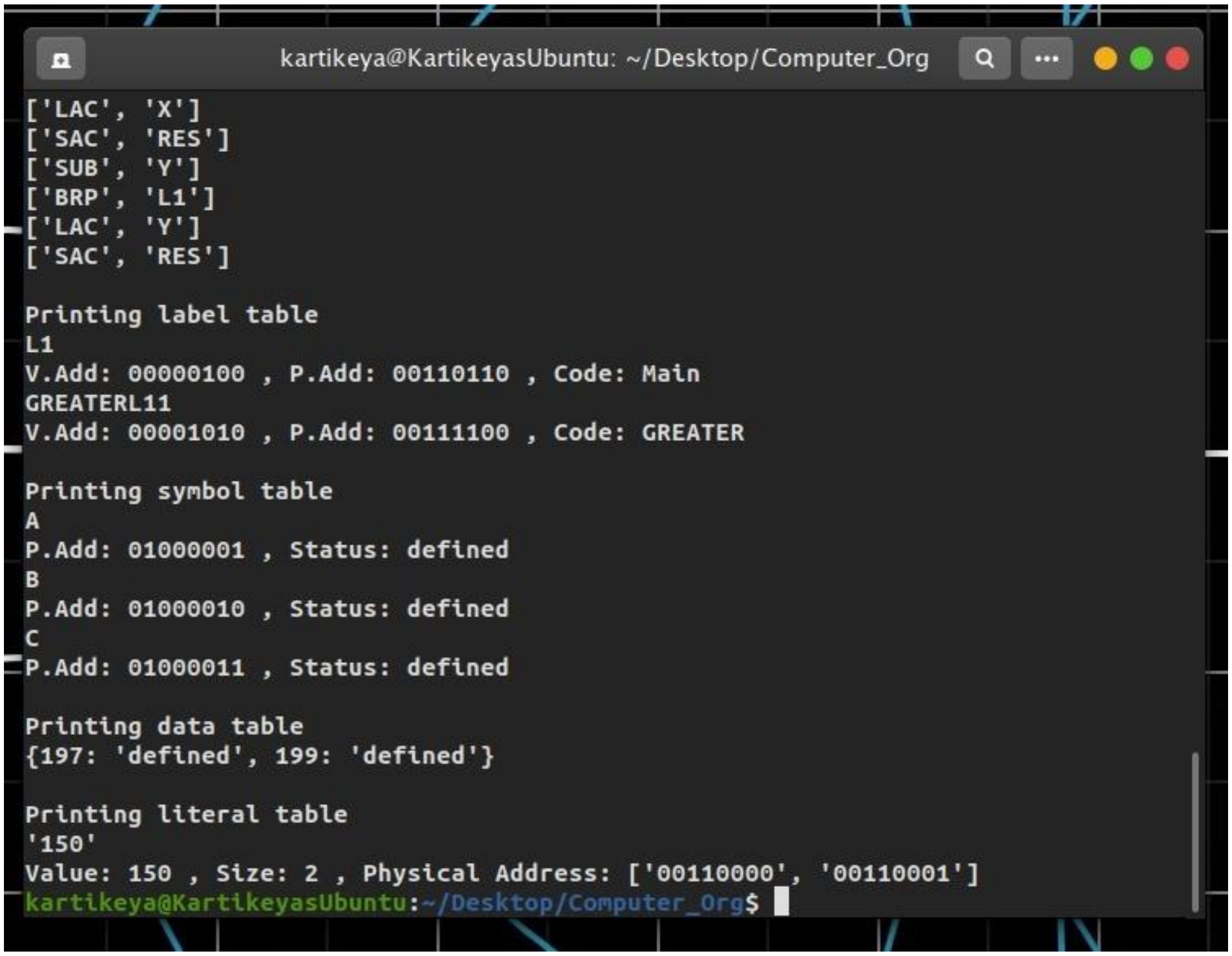


```
Printing data table
{197: 'defined', 199: 'defined'}

Printing literal table
'150'
Value: 150 , Size: 2 , Physical Address: None
Load 50
numins 15
######## SUCCESS: Second pass ended successfully ########
00110010 1000 11000101

00110011 0010 01000001

00110100 1000 11000111

00110101 0010 01000010

00110110 0001 01000001

00110111 0010 01000011

00111000 0100 01000010

00111001 0111 00111100

00111010 0001 01000010

00111011 0010 01000011

00111100 1001 01000011

00111101 0011 01000001

00111110 0010 01000001

00111111 0110 00110110

01000000 0011 00110000
```



kartikeya@KartikeyasUbuntu: ~/Desktop/Computer_Org

```
Printing instruction table
['00110010', ['1000', '11000101']]
['00110011', ['0010', '01000001']]
['00110100', ['1000', '11000111']]
['00110101', ['0010', '01000010']]
['00110110', ['0001', '01000001']]
['00110111', ['0010', '01000011']]
['00111000', ['0100', '01000010']]
['00111001', ['0111', '00111100']]
['00111010', ['0001', '01000010']]
['00111011', ['0010', '01000011']]
['00111100', ['1001', '01000011']]
['00111101', ['0011', '01000001']]
['00111110', ['0010', '01000001']]
['00111111', ['0110', '00110110']]
['01000000', ['0011', '00110000']]

Printing macro table
GREATER
Parameters: ['X', 'Y', 'RES'] , Labels: ['L1']
Instruction Table:
['LAC', 'X']
['SAC', 'RES']
['SUB', 'Y']
['BRP', 'L1']
['LAC', 'Y']
['SAC', 'RES']
```


kartikeya@KartikeyasUbuntu: ~/Desktop/Computer_Org

```
['LAC', 'X']
['SAC', 'RES']
['SUB', 'Y']
['BRP', 'L1']
['LAC', 'Y']
['SAC', 'RES']

Printing label table
L1
V.Add: 00000100 , P.Add: 00110110 , Code: Main
GREATERL11
V.Add: 00001010 , P.Add: 00111100 , Code: GREATER

Printing symbol table
A
P.Add: 01000001 , Status: defined
B
P.Add: 01000010 , Status: defined
C
P.Add: 01000011 , Status: defined

Printing data table
{197: 'defined', 199: 'defined'}

Printing literal table
'150'
Value: 150 , Size: 2 , Physical Address: ['00110000', '00110001']
kartikeya@KartikeyasUbuntu:~/Desktop/Computer_Org$
```

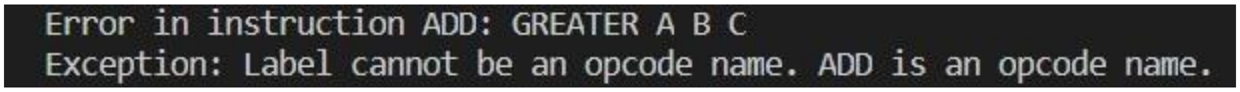
Errors Handling

Errors handled in first pass

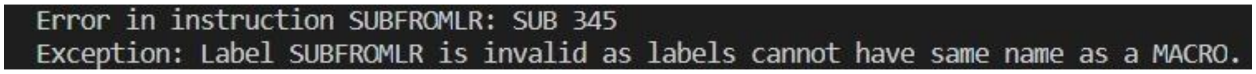
1. Invalid label name

There are two conditions for a label name to be invalid. Label cannot have a macro name in it and it cannot be an opcode name. This error occurs if such a label is detected.

Examples: ADD: GREATER A B C or SUBFROMLR : SUB 355 (assuming a macro named SUBFROMLR is defined in the code)



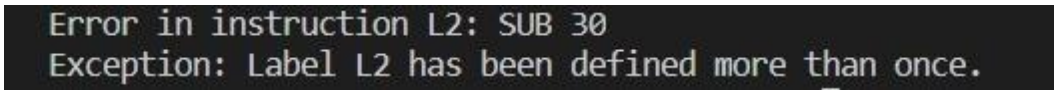
```
Error in instruction ADD: GREATER A B C
Exception: Label cannot be an opcode name. ADD is an opcode name.
```



```
Error in instruction SUBFROMLR: SUB 345
Exception: Label SUBFROMLR is invalid as labels cannot have same name as a MACRO.
```

2. Multiple label definitions

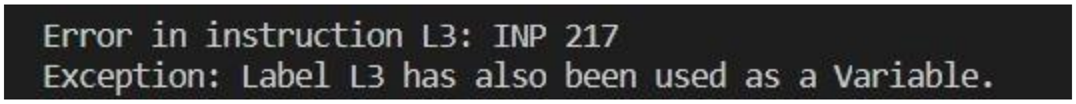
This error occurs when a label is defined/declared multiple times. Label declaration includes statements such as L1: ADD 589



```
Error in instruction L2: SUB 30
Exception: Label L2 has been defined more than once.
```

3. Defined label names cannot be used as variable names and vice versa

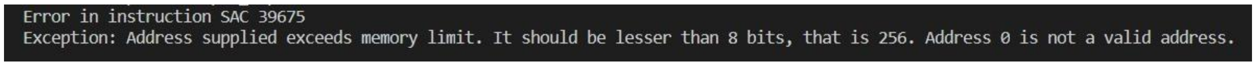
Since both label names and variable names can be alpha-numeric, it is possible for ADD L4 and BRN L4 can both be valid. To avoid this, an error is thrown if the label is declared as a variable before or vice versa.



```
Error in instruction L3: INP 217
Exception: Label L3 has also been used as a Variable.
```

4. Address supplied is out of bounds

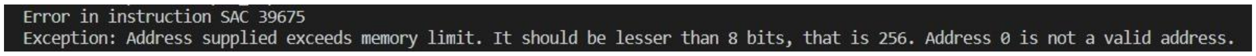
Since the maximum size of the memory is 8 bit, user cannot access memory cells greater than 256. This error is thrown if the user tries to access addresses greater than 4096 such as SAC 12392



```
Error in instruction SAC 39675
Exception: Address supplied exceeds memory limit. It should be lesser than 8 bits, that is 256. Address 0 is not a valid address.
```

5. Incorrect number of operands supplied for an opcode

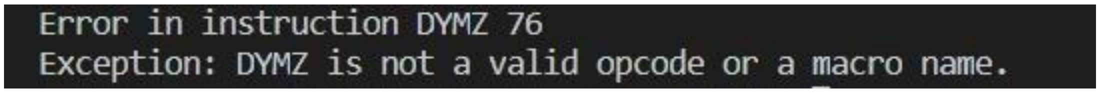
This error will be throws in the parameters/arguments supplied to the argument is greater than or less than the number of arguments required by the opcode. For example CLA 88 or DIV



```
Error in instruction SAC 39675
Exception: Address supplied exceeds memory limit. It should be lesser than 8 bits, that is 256. Address 0 is not a valid address.
```

6. Invalid opcode name/Macro name

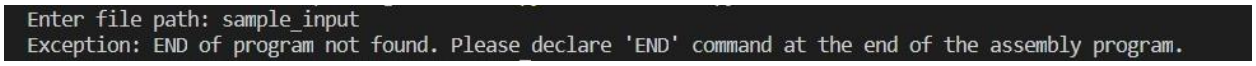
This error is thrown if the opcode supplied is not a macro call and neither is it a part of the available opcodes i.e. ['CLA',LAC','SAC','ADD','SUB','BRZ','BRN','BRP','INP','DSP', 'MUL','DIV','STP'] For example WHDK 123



```
Error in instruction DYMZ 76
Exception: DYMZ is not a valid opcode or a macro name.
```

7. END of program not found

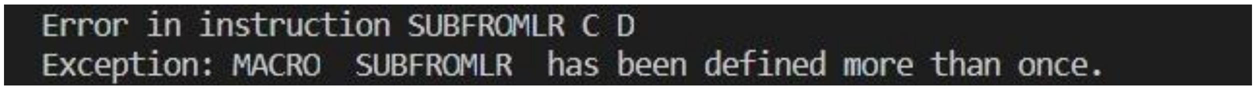
This error is throw if the assembly directive END is not found anywhere in the code.



```
Enter file path: sample_input
Exception: END of program not found. Please declare 'END' command at the end of the assembly program.
```

8. Multiple macro definitions

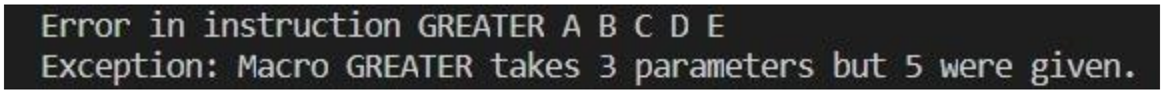
This error occurs when a macro is defined/declared multiple times. Macro declaration includes statements such as ADDTWO MACRO ABC



```
Error in instruction SUBFROMLR C D
Exception: MACRO  SUBFROMLR  has been defined more than once.
```

9. Incorrect number of parameters supplied in a macro

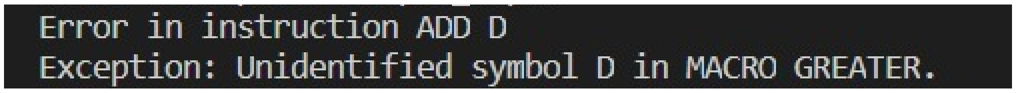
This error occurs if the parameters supplied during the macro call does not match the number of parameters supplied during macro definition. For example: (consider the ADDTWO macro declared above) ADDTWO 157 158 or ADDTWO '5' '6' 192 102



```
Error in instruction GREATER A B C D E
Exception: Macro GREATER takes 3 parameters but 5 were given.
```

10. Unidentified symbol used in a macro

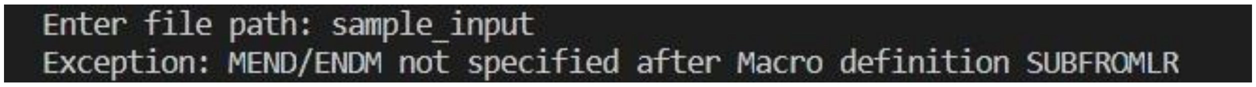
This error occurs if a symbol(variable) is used in a macro which is not a one of the parameters supplied. For example: (consider the ADDTWO macro declared above ADDTWO MACRO A,B,C and Macro call ADDTWO 157, D,E), ADD F will throw this error.



```
Error in instruction ADD D
Exception: Unidentified symbol D in MACRO GREATER.
```

11. MEND.ENDM for macro not found

This error is thrown if the assembly
directive ENDM or MEND to specify the end of a MACRO is not found anywhere in the code.

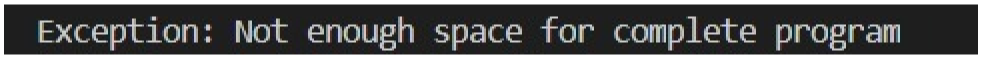


```
Enter file path: sample_input
Exception: MEND/ENDM not specified after Macro definition SUBFROMLR
```

Errors handled in second pass

1. Not enough space for complete program

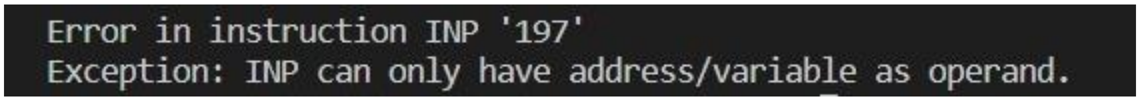
This error is rare and will be thrown if the memory spaces occupied by the direct addresses, literals, symbols and instructions exceeds the total available memory or if there is no contiguous space available to fit all the instructions.



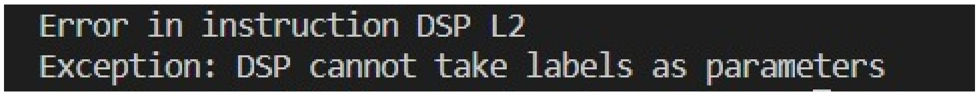
```
Exception: Not enough space for complete program
```

2. Opcode can only have address/variable as operand

Operands supplied for LAC, DSP, INP should be an address/variable only. No constants or labels allowed. For example, consider: INP '110' or DSP L1 (provided, L1 is a defined label).



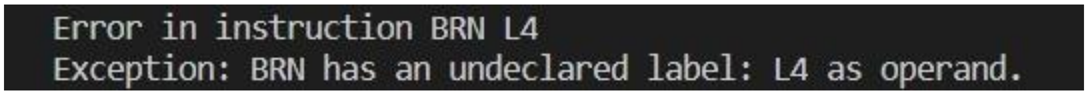
```
Error in instruction INP '197'
Exception: INP can only have address/variable as operand.
```



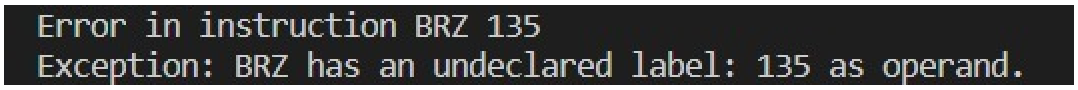
```
Error in instruction DSP L2
Exception: DSP cannot take labels as parameters
```

3. Opcode can only have a valid label as operand

Operands supplied for BRN, BRZ, BRP should have a defined and valid label. This error is thrown if the operand is anything but a defined label. For example, BRZ 120 or BRN L5 (provided, L5 is not a defined label).



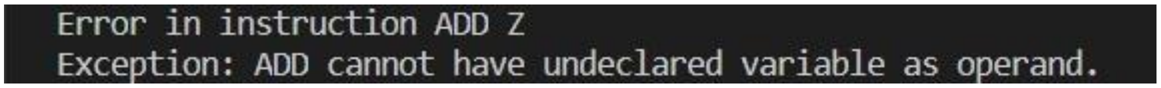
```
Error in instruction BRN L4
Exception: BRN has an undeclared label: L4 as operand.
```



```
Error in instruction BRZ 135
Exception: BRZ has an undeclared label: 135 as operand.
```

4. Opcode can only have address or variable or constant as operand

Operands supplied for ADD, MUL, SUB and DIV should be a defined address or a constant (not undefined address). For example, ADD X will throw an error if the variable X is undefined.



```
Error in instruction ADD Z
Exception: ADD cannot have undeclared variable as operand.
```