CRAZYOS: A REAL MODE OPERATING SYSTEM FOR EDUCATIONAL PURPOSES

MAJOR PROJECT REPORT

Submitted in partial fulfilment of the requirements for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

ELECTRICAL & ELECTRONICS ENGINEERING

by

Praneet Kapoor	Rachit Jain	Shristi Sharma	Sumit Gaur
04115604918	04315604918	05415604918	06115604918

Under the guidance
of
Mr. Md. Saquib Faraz
Assistant Professor

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING
DR. AKHILESH DAS GUPTA INSTITUTE OF TECHNOLOGY AND MANAGEMENT
(AFFLIATED TO GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY, DELHI)
NEW DELHI - 110053

JUNE 2022

CANDIDATES' DECLARATION

It is hereby certified that the work which is being presented in the B.Tech Major Project Report entitled **CrazyOS: A Real Mode Operating System for Educational Purposes** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** and submitted in the **Department of Electrical & Electronics Engineering** of **Dr. Akhilesh Das Gupta Institute of Technology & Management, New Delhi (Affiliated to Guru Gobind Singh Indraprastha University, Delhi)** is an authentic record of our own work carried out during the period from **February 2022 to June 2022** under the guidance of **Mr. Md. Saquib Faraz, Assistant Professor, EEE department**.

The matter presented in the B. Tech Major Project Report has not been submitted by us for the award of any other degree of this or any other Institute.

Praneet Kapoor	**Rachit Jain**	**Shristi Sharma**	**Sumit Gaur**
04115604918	**04315604918**	**05415604918**	**06115604918**

This is to certify that the above statement made by the candidates is correct to the best of our knowledge.

Mr. Md. Saquib Faraz
Assistant Professor

Mr. Ajit Kumar Sharma
H.O.D., EEE Deptt.

The B. Tech. Major Project Viva-Voice Examination of Praneet Kapoor, Rachit Jain, Shristi Sharma, and Sumit Gaur has been held on

Dr. Amruta Pattnaik
Project Coordinator

(Signature of the External Examiner)

ABSTRACT

Technical education of undergraduate students should ensure that they are provided with practical knowledge which makes them fit for the ever changing industry. One of the ways of accomplishing this is by attempting to include project based learning alongside regular coursework. This project is a step towards this direction. We have presented a single-user, single-tasking, real mode operating system. The operating system comes with two applications, a shell and a line editor. A library of subroutines has also been created which includes custom implementation of getchar, putchar, printf, strlen, strcmp, strcpy and atoi in x86 assembly language. With only 1690 lines of code, the source code of the operating system can be easily understood by students and then used as a sandbox for learning about the design of the 8086 microprocessor and the original IBM PC.

ACKNOWLEDGEMENTS

We express our deep gratitude to **Mr. Md. Saquib Faraz, Assistant Professor, Department of Electrical & Electronics Engineering** for his valuable guidance and suggestions throughout our project work.

We would like to extend our sincere thanks to **Mr. Ajit Kumar Sharma, Head of the Department, EEE** for his time to time suggestions to complete our project work. We are also thankful to **Prof. (Dr.) Sanjay Kumar, Director** for providing us the facilities to carry out our project work.

We are thankful to **Dr. Amruta Pattnaik, Project Coordinator** for her valuable guidance.

Praneet Kapoor	**Rachit Jain**	**Shristi Sharma**	**Sumit Gaur**
04115604918	**04315604918**	**05415604918**	**06115604918**

TABLE OF CONTENTS

CANDIDATE'S DECLARATION i

ABSTRACT ii

ACKNOWLEDGEMENTS iii

TABLE OF CONTENTS iv

LIST OF FIGURES vii

LIST OF TABLES viii

LIST OF ABBREVIATIONS ix

Chapter 1: Introduction 1

1.1 Introduction 1

1.2 Literature Survey 3

1.2.1 Algorithms and Programs 3

1.2.2 Operating Systems 7

1.2.3 A Brief History of Computers, Programming Languages, and Operating Systems 9

1.2.4 The 8086 Microprocessor from a Programmer's Point of View 18

1.2.5 Design of the original IBM PC 25

1.2.6 Deficiencies of Model Curriculum for Engineering and Technology Developed by AICTE 30

1.3 Objective 33

1.4 Overview 33

Chapter 2: Implementation of CrazyOS 34

2.1 Introduction 34

2.2 Set-up of the Development Environment 35

2.2.1 Code Editor 35

2.2.2 Assembler 35

2.2.3 Emulator 36

2.2.4 Makefile 37

2.2.5 Git 37

2.3 Structure of the Project Directory 38

2.3.1 Parent Directory . . . 38
2.3.2 Project Directory: ../8086 . . . 39
2.4 Bootloader: ../boot . . . 40
2.4.1 boot.asm . . . 40
2.4.2 boot_util.asm . . . 43
2.5 Kernel Header: ../kernel/kernel.asm . . . 45
2.6 Library Functions: ../include . . . 46
2.6.1 Organization of Library Functions . . . 48
2.6.2 %include Guard . . . 48
2.6.3 Standard I/O subroutines: ../include/ttyio . . . 49
2.6.4 String Handling: ../include/string . . . 53
2.6.5 CMOS and RTC: ..include/cmos . . . 55
2.6.6 Random Number Generator: ../include/random . . . 57
2.6.7 Sound Generator: ../include/sound . . . 58
2.6.8 Disk Read/Write Operations: ..include/disk . . . 61
2.6.9 APM Support: ../include/apm . . . 62
2.7 Applications . . . 64
2.7.1 Shell: ../apps/shell . . . 65
2.7.2 Editor: ../apps/edit . . . 67
2.8 Disk Applications: ../disk-apps . . . 68
2.9 Building Binaries and Disk Images . . . 69
Chapter 3: Results and Discussion **71**
3.1 Introduction . . . 71
3.2 Cloning & Running CrazyOS . . . 71
3.3 Using the Shell . . . 72
3.3.1 Using time, date, srng, and clear . . . 72
3.3.2 Running Disk Applications . . . 73
3.3.3 Power Commands . . . 73
3.4 Using the Editor . . . 74
3.5 Bugs . . . 75
3.6 Discussion . . . 75

Chapter 4: Conclusion & Future Scope **77**

4.1 Conclusion 77

4.2 Future Scope 77

References **79**

Research Paper **83**

Curriculum Vitae **91**

LIST OF FIGURES

1.1 Steps involved in building a C program having multiple source files 6
1.2 Layout of a typical computer system [39] 8
1.3 Pin configuration of the 8086 [3] 19
1.4 Block diagram of the 8086/8088 microprocessor [2] 20
1.5 8086 registers [2] 21
1.6 Ordering of bytes of words in registers and memory 23
1.7 Storing doublewords in memory 23
1.8 Storing pointers on stack during intersegment calls 24
1.9 Cylinder/tracks, sectors and heads on a hard disk platter [28] 26
1.10 Boot sequence of an IBM PC 28
1.11 Typical memory map of an IBM PC after boot [14] 29

2.1 Parent directory tree (depth = 0) 38
2.2 Directory tree for `CrazyOS/8086` (depth = 0) 40
2.3 Directory tree for `CrazyOS/8086/boot` (depth = 0) 40
2.4 Hexdump of the bootloader showing the placement of the magic number 0xaa55 43
2.5 Memory map of CrazyOS 45
2.6 Directory tree for `CrazyOS/8086/include` (depth = 1) 47
2.7 Block diagram of the 8254 PIT chip [4] 58
2.8 Control word format for the PIT chip 59
2.9 Connection of motherboard speaker with timer 2 of the PIT chip 59
2.10 Directory tree for `CrazyOS/8086/apps` (depth = 1) 65

3.1 CrazyOS startup screen 72
3.2 Using `time`, `date`, `srng`, and `clear` commands 73
3.3 Using `disk read` and `run` commands to run a disk application 73
3.4 Using `power` command 74
3.5 Using `edit` to create and edit a file 74
3.6 Bug in line editor causing it include contents of invalid commands 75

LIST OF TABLES

1.1 Generations of Computers on the Basis of their Hardware 10
1.2 Typical Memory Map of an IBM PC after boot-up sequence 30

2.1 Hex-codes for 8-bit colours 42

LIST OF ABBREVIATIONS

ACPI Advanced Configuration and Power Interface

AGC Apollo Guidance Computer

AICTE All India Council for Technical Education

ANSI American National Standards Institute

APM Advanced Power Management

ARM Advanced RISC Machines, originally Acorn RISC Machine

BDA BIOS Data Area

BIOS Basic Input/Output System

BIU Bus Interface Unit

CHS Cylinder-Head-Sector

CP/M Control Program/Monitor, later Control Program for Microcomputers

CPU Central Processing Unit

CTSS Compatible Time Sharing System

DEC Digital Equipment Corporation

DIP Dual Inline Package

DOS Disk Operating System

EBDA Extra BIOS Data Area

EDSAC Electronic Delay Storage Automatic Calculator

EDVAC Electronic Discrete Variable Automatic Computer

ENIAC Electronic Numerical Integrator and Comparator

EU Execution Unit

FAT File Allocation Table

GCC GNU Compiler Collection

GNU GNU is Not UNIX

IBM International Business Machines

IC Integrated Circuit

IT Information Technology

IVT Interrupt Vector Table

LCD Liquid Crystal Display

LMA Low Memory Area

MBR Master Boot Record

MINIX Mini-UNIX

MIT Massachusetts Institute of Technology

MULTICS MULTiplexed Information and Computation Service

NASA National Aeronautics and Space Administration

NASM Netwide ASseMbler

OEM Original Equipment Manufacturer

OS Operating System

PC Personal Computer

PCI Peripheral Component Interconnect

PDP Programmed Data Processor

PIC Programmable Interrupt Controller

PIT Programmable Interval Timer

RAM Random-Access Memory

RISC Reduced Instruction Set Computer

ROM Read-Only Memory

UEFI Unified Extensible Firmware Interface

UMA Upper Memory Area

UNIVAC Universal Automatic Computer

CHAPTER 1: INTRODUCTION

1.1 Introduction

Digital electronics and programming has become the backbone of every industry. To undermine the value of digital computers and programming skills in 21st century would be an indication of the ignorance of the person making this judgement. Therefore, it becomes highly important that all undergrad students of engineering and various other technical discipline have

1. at least a rudimentary knowledge of computer hardware, and
2. basic programming skills in a high level language, preferably Python 3.

The depth of such knowledge would depend upon the profession they wish to undertake.
For an electrical engineer, having an understanding of underlying hardware of a computer is of utmost importance. With the mathematical training they have along with their knowledge of electronics, electrical engineers are the ones who are responsible for designing the next generation computers and programming languages. This has been true in the history of computing. ENIAC, IBM 1401, Apollo Guidance Computer (AGC), Apple II, the original IBM PC, and many other similar revolutionary computers were all designed by electrical engineers. C programming language and UNIX operating system, both of which have been the backbone of the industry for over 50 years, were developed by Ken Thompson and Dennis Ritchie, both of whom were electrical engineers. The development of Linux by Linus Torvalds was also motivated by his curiosity to understand the underlying hardware of his new i386 IBM PC [40]. TempleOS, a 64-bit ring 0 operating system was designed from scratch by Terry Davis, an electrical engineer, in spirit of the Commodore VIC-20 he used in his youth [27]. This makes it clear that electrical engineers also enrich the field of computer architecture and programming by their contributions. A similar case can be made for other fields like mathematics and physics. Therefore, the widely held belief that programming is for those in the field of computer science or information technology (IT) is wide off the mark.
Now, engineering is a practical profession. A good engineer should be able to use fundamental principles to solve problems or derive practical results. The ability to do so requires experience, which is often termed as "*getting hands dirty*". It is of primary importance that an engineering

student should have their hands sufficiently dirty in the profession they wish to pursue. A student interested in pursuing a career in silicon industry should have considerable experience in designing systems on an FPGA board. If he* has only memorized the important principles from some standard book, that student will have no value in the industry. This is similar to studying mathematics: learning theory is not enough, one needs to solve enough examples to get a *feel* for the concept at hand. It is also possible that while getting their hands dirty in a field they disliked at first, they might find it likeable.

It were these opportunities to develop practical experience which were found to be missing from the course on Microprocessers and Microcontrollers. The course, in a span of 4 units and 45 hours, covers 8085 and 8086 microcontrollers, their support components (8259 PIC, 8289 bus arbiter, LCD, stepper motor, etc.), and 8051 microcontroller. Instead of following texts like *The 8086 Family User's Manual* [2], students and teachers alike follow notes and presentations which don't talk about the practical side of various concepts like segmentation, effective address calculation and string operations, to name a few. Lab experiments consist of writing code to multiply and divide two numbers, move strings from one location to another, and control a stepper motor. Rare and costly development kits, and toy emulators like emu8086 are used in favour of professional tools like NASM and QEMU. In the end, students treat it like another boring subject with "*no possible application in the future*".

Disgruntled by the approach taken in teaching one of his favourite subjects, one of the authors of this report (Kapoor), decided to write his own 16-bit operating system from scratch. He considered this to be fun challenge which would allow him to tinker with the hardware and understand the architecture of 8086 microprocessor and the original IBM PC. His objective at first was to develop a UNIX like operating system. However, as UNIX and its derivatives (Linux, MINIX, BSD, etc.) are complex systems and developing them would require ample amount of free time, it was decided that operating system designed would be a simple one consisting of just a primitive shell and line editor. Basic commands can be executed from shell. Students can develop the their own programs using the libraries provided by the author, assemble using NASM, create a virtual disk and run it in real mode on QEMU's emulation of i386 based IBM PC. Various unconventional decisions were made while making this operating system, some of which were:

1. Several standard functions like `printf`, and `strcmp` were written from scratch following some simple non-standard calling conventions.

*"He" should be read as "he or she" throughout the report.

2. Instead of using object files and a linker to produce the final binary of the operating system, source files are combined with each other using `%include` preprocessor directive. Source files are prevented from getting included more than once by using the `%ifndef %define %endif` guard (which is also used in C) [7].

3. Filesystem of the operating system is extremely simple. Files are stored in contiguous sectors and are terminated with a magic number, `0xaa`. If the user has not kept a track of the sectors belonging to a particular file, he will most probably overwrite a file during disk write operation. There is no concept of directories.

Due to such unusual design of the operating system, it was named *CrazyOS* by its creator. CrazyOS is a single-user, single-tasking real mode operating system which, along with its boot-loader, is written in x86 assembly language from scratch. It features a shell and a line editor. Being just 1690 lines of code, students can use it as a sandbox to gain a practical understanding of the 8086 microprocessor and architecture of the IBM PC.

1.2 Literature Survey

The purpose of this literature survey is to provide the reader with a frame of reference from which they can understand the motivation, design choices, and the overall scope of this project. We begin with informal definitions of several key concepts which will be encountered in this report. A brief history of computing systems, and the design of the 8086 microprocessor and the original IBM PC are discussed. We conclude this literature survey with a brief discussion of the deficiencies of the current engineering and technical education in India.

1.2.1 Algorithms and Programs

The definition of computer has gone a several changes throughout history. Before 1950's, the term usually referred to a person who had the job to do calculations or *computations* [21]. Nowadays, it refers to a machine which processes data according to instructions fed into it. Majority of such machines are made out of electronic components and work on the principles of digital logic. Therefore, by the term computer one implicitly refers to a *digital electronic computing system* [11]. One common denominator throughout this transformation of computing systems has been that they need to be told to perform a particular task. The task to be performed is usually divided into smaller, manageable tasks or instructions. A sequence of such tasks is called an *algorithm*.

Each sub-task which makes up an algorithm is called a *step* or an *instruction*. Algorithms are usually written in a human language or mathematical symbols or, in general, both.
A good algorithm has five important features [26]:

1. *Finiteness*: It must stop after a finite number of steps.
2. *Definiteness*: Each step of the algorithm must be precisely defined.
3. *Input*: An algorithm should have zero or more inputs.
4. *Output*: An algorithm should have one or more outputs.
5. *Effectiveness*: An algorithm is said to be effective if, in principle, if each of its steps can be executed manually, using pencil and paper, in a finite length of time.

A simple example at this point would be helpful. Consider a student who is given the task to print out whether a given number is prime or not. He will first begin by defining prime numbers.
Definition: A number is said to be prime if it is only divisible by 1 and the number itself.
With this definition at hand, it becomes obvious that to test whether a number N is prime or not, we have to check whether it is divisible by any other natural number except for 1 and itself. Upon inspection the student might observe that one does not need to test N's divisibility with numbers greater than $N/2$. The final algorithm that is obtained is:

Step 1: $n = 2$, $N = 0$.

Step 2: Input N.

Step 3: If $n > N/2$, then jump to step 6.

Step 4: If N is divisible by n, then jump to step 7.

Step 5: If N is not divisible n, then $n = n + 1$ and jump to step 3.

Step 6: Print `N is a prime number` and end program.

Step 7: Print `N is not a prime number` and end program.

A modern digital computer would not be able to understand these commands. It can only understand and execute instructions native to its architecture. A program consisting of opcodes of instructions and data, all in binary/hexadecimal format, is called *machine code*. A file consisting

of machine code is called a *binary file* or, simply, *binary*. A computer is able to understand only binaries and executes them in one shot (provided they mean something. Not every hexadecimal number can be mapped to an opcode). However, implementing any algorithm in machine code would take a significant amount of time. Debugging the binary would be even more troublesome. The problem can be solved by defining a language which satisfies the following criteria:

1. After sufficient training, a human being should be able to implement and debug algorithms in this language.

2. The algorithm implemented in this language can be easily converted into a binary using some conversion tool.

A language satisfying both of these criteria is called a *programming language*. The act of writing code in a programming language is called *programming* or *coding*. An algorithm implemented in such a language is called a *program* or *code*.
The simplest kind programming language is *assembly language* (One can call machine code as a programming language, however, it is no longer used directly to practically program any computer). In assembly language, every instruction and register, instead of being referred by their opcode, are referred by a mnemonic. Labels are used to mark locations instead of hardcoded addresses. An *assembler* is used to convert these mnemonics and labels into machine code.
While learning assembly language is not impossible, it has, however, a very steep learning curve. This calls the need for *higher level languages* (HLL). These programming languages mask the programmer from the architecture of the machine they are working on. This enables them to direct their energies towards developing and implementing algorithm to get the task done. Every HLLs have its own set of symbols for defining operations and data structures, and semantics and syntax for successfully implementing algorithms [6][17]. C is one such programming language. The algorithm for testing whether a number is prime or not can be implemented in it.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char ** argv)
{
        int n = 2, N = 0;
        fscanf(stdin, "%d", &N);
```

```
        for (; n <= N / 2; ++ n) {
                if (N % n == 0) {
                        fprintf(stdout, "N is not a prime number\n");
                        exit(0);
                }
        }
        fprintf(stdout, "N is a prime number\n");
        return 0;
}
```

A C *compiler* takes this *source file*, checks it for errors, and if none are found, converts it into its equivalent assembly code. The assembler takes this assembly source file and converts it into an *object file*. An object file contains machine code (binaries), but is not yet in machine executable form as addresses of labels and library functions have not yet been included into it. An object file has a code section and data section (besides other sections depending upon the object file format being used). Machine code of instructions corresponding to a single statement of C code are kept in the code section, whereas symbols/variables like `n` and `N` in the above code are placed in the data section. Finally, the object file is passed to another program called *linker* which resolves the addresses of various symbols used, includes the library functions' binaries and produces an executable binary file.

Problems, algorithms and their implementations in a programming language get more interesting and complex than the simple example that has been provided here. As the size of a program increases, it is often wise to split it into several source files, each of which is individually compiled, and the resulting object files are linked to produce a single binary. Figure 1.1 shows the the compiling, assembling, and linking of programs with multiple source files.

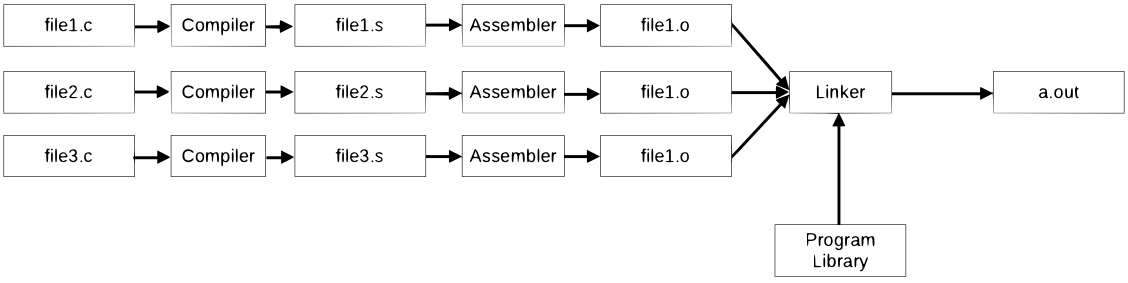


Figure 1.1: Steps involved in building a C program having multiple source files

1.2.2 Operating Systems

What is an Operating System?

Generally a computer is running multiple tasks at a time. This increases the complexity of building binaries of each individual program and running them. The complexity of running multiple tasks at the same time can be understood by two examples.

Example 1: Take the case of a student taking a numerical methods exam. Generally, questions in numerical methods require us to perform certain operations repeatedly till the answer starts converging satisfactorily. The student will be required to perform such computations many times. He must make sure that his scratch-pad is neat so that the results of different iterations and questions do not get mixed up. He must also make sure that the scratch-pad is utilized efficiently, lest he will run out of space to perform more computations.

Example 2: A librarian is tasked with managing a library. This task is composed of several tasks, for example, issuing a book as per member's demand, making sure that all books are in their specific section, etc. These tasks require that the librarian create and maintain a record of every member and every piece of literature which is present in the library.

In each of these examples, a single human computer was given several tasks to perform which can be performed only if every individual task is dealt with without affecting other tasks. In a computer, we run multiple tasks/programs at the same time. An editor allows us to edit source files, a shell allows us to run the compiler and a linker, a document viewer allows us to read manuals, and so on. It has to be made sure that no two programs interfere with each other. As programs often acts on files, a *filesystem* also needs to be in place to make sure that the files are operated on correctly. The program tasked with managing the resources of the computer and providing an environment in which programs can run is the *operating system* [39].

Without an operating system, all sorts of chaos can happen. A program can start interpreting the code section of another as data, and overwrite it completely. Two or more programs accessing the printer without any prioritization and buffering will result in parts of text outputted by each program getting printed. Thus, an operating system can be thought of as the crucial software responsible for maintaining order and preventing possible chaos.

System and Application Software

Certain application-independent programs like shell, editor, compilers, windowing system, etc. are shipped along with the operating system. These are called *system software*. Software like web browser, games, word processors, etc. are called *application software*. System software provides the environment in which application software can run. Continuing with the analogies provided in the previous two examples, if the student is to be considered as an operating system which is managing everything, then the sheet of paper and the pen he uses are the system software. The equations he scrolls are the application software. He could have scrolled them out on the furniture, or on the walls, but he uses the tools he is provided with to get the job done. Similarly, if the librarian is an operating system, then the chairs and lights in the library are analogous to system software. Members can be made to sit on the floor and read; they are simply using the tools they are provided. Clearly, while one can choose between the shell, compiler, or the window manager they wish to use, the user is not free to write their own scheduler [39]. Figure 1.2 shows the arrangement of hardware and software in a typical computer.

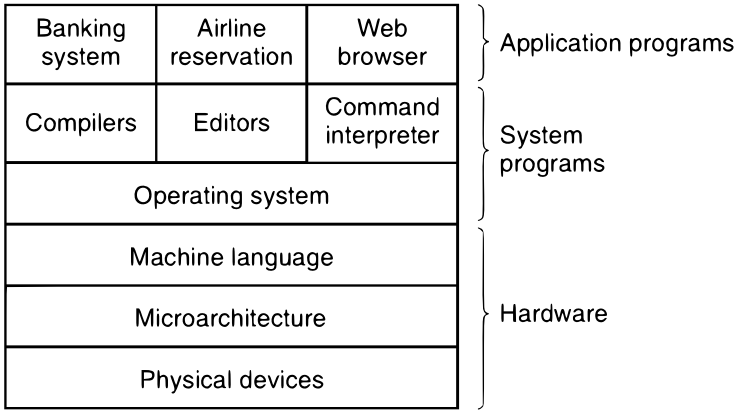


Figure 1.2: Layout of a typical computer system [39]

Functions of an Operating System

We have described what an operating system is with the help of analogies. These analogies showed us that in operating system is the software which is responsible for managing the resources of the system. As a resource manager, the operating system has to make sure that programs do not interfere with each other. As most of the operating system today support multiple users, it becomes important that the resources allocated to each user are managed carefully. This becomes important in large organizations like universities and industries, where multiple com-

puters are connected to a single server. This type of resource management involves sharing of resources or *multiplexing*. There are two types of multiplexing: time multiplexing and space multiplexing. Time multiplexing involves allocating a specific a unit of time for each program to use the CPU. It also involves accessing I/O devices based on priority by various programs. Space multiplexing involves giving each user (or a program) a part of the resource. An example would be the storing of all programs in main memory. This way, they can easily be time multiplexed to use the CPU and memory is not entirely given to one program only. Secondary memory like hard disks are also space multiplexed by giving each user some disk space in which they can store and organize their files.

Operating systems also perform the function of extending the capabilities of the machine so as to make it pleasant to deal with. When a programmer, or an ordinary user use a computer, they should be only considered with getting their job done. They should not have worry implementing a driver for hard disk, monitor, and other peripherals. Operating systems extend the instruction set of the machine by providing utility functions which allow a programmer to easily use the resources of the computer. The most fundamental utility functions are called *system calls*. Examples would be `read` and `write` system calls in UNIX-like operating systems. These system calls are used by programs to access and operate on various files and peripherals like the screen. By providing the programmer with system calls and by setting up an environment in which work can easily be done, operating system extend the capabilities of the machine. One can say that operating systems provide the user with a virtual machines which is easier to deal with [39].

1.2.3 A Brief History of Computers, Programming Languages, and Operating Systems

Knowing about the history of computers is important for several reasons. Tannenbaum [39] explained the importance of learning about old systems with the example of contiguous memory allocation:

> *...history may repeat itself in computer science as new generations of technology occur. Contiguous allocation was actually used on magnetic disk file systems years ago due to its simplicity and high performance (user friendliness did not count for much then). Then the idea was dropped due to the nuisance of having to specify final file size at file creation time. But with the advent of CD-ROMs, DVDs, and other write-once optical media, suddenly contiguous files are a good idea again. For such media, contiguous allocation is feasible and, in fact, widely used. Here*

all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system. It is thus important to study old systems and ideas that were conceptually clean and simple because they may be applicable to future systems in surprising ways.

A lesson in the history also allows a person to understand the reason behind the design of present computers. For example, after reading the design of DEC's PDP-11, it can be seen how it influenced the design of x86 family of processors [38].

History of computers involve studying their evolution. Just like a study of human history consists of the study of various civilizations and generations, a study of computer history involves the study of various generations of computers. Beginning of a new generation are marked with the usage of newer, better, and reliable technology in comparison to the previous generation. The technology most used for this classification is the hardware. By using hardware as the basis for classification, most authors divide computer history into five generations, starting in 1951 with the launch of UNIVAC I [11]. The Table 1.1 is an excerpt from one such detailed table giving the history of computers [22].

Table 1.1: Generations of Computers on the Basis of their Hardware

Generation	Year	Technological Features	Examples
1st Generation	1951-1960	Vacuum tubes	UNIVAC I, IBM 650
2nd Generation	1959-1965	Transistors	IBM 1401, IBM 7094
3rd Generation	1964-present	Integrated circuits	IBM 360, PDP-11 (16-bit minicomputer)
4th Generation	1971-present	VLSI, Microprocessors	Altair 8800, Commodore VIC-20
5th Generation	Present-future	Parallel processing architecture	Computers using Pentium or higher

By using programming languages and operating systems as the basis for classification, we arrive at similar results [39]. This type of classification is more useful in context of the present report and, therefore, it will be discussed from here on.

The Zero Generation Computers (1645–1945)

This generation of computers came before the invention of digital computers. In reality, if one takes into consideration human computers and mechanical calculators, then computers have been

around since eternity. The oldest known mechanical computer would be the Antikythera mechanism from 87 BC which was supposedly used to predict astronomical positions and eclipses [11][20]. More examples of early calculators would be abacuses and slide rules, the latter being used well into the 1970's and is credited for putting the man on the Moon [37].

The first mechanical calculator would be Pascal's calculator (or Pascaline). It was capable of addition and subtraction, and multiplication and division using repeated addition and subtraction, respectively [16].

The most significant mechanical computers were Charles Babbage's difference engine and analytical engine. Construction of the difference engine began in 1819 and a small portion of it was completed and demonstrated in 1822. The difference engine used the method of finite differences to interpolate functions [19]. The analytical engine was a more complex machine which introduced several new concepts like program and memory. Although not completed during Babbage's lifetime, the analytical engine was shown to be programmable by Ada Lovelace [39].

Mechanical differential analyzer's and their analog counterparts used during the two world wars were also programmable. The integrators and differentiators they were made up of were linked together to model systems through their differential equations. The output was a plot of the system's output with respect to input. As signals out of blocks were feeble, they were passed through an amplifier before being passed to the next block. Each of these amplifications were to be taken into account when interpreting the output [33].

For each of these mechanical computers, the concept of a program was plugboard wiring or positions of mechanical linkages. The concept of an operating system, or even programming languages, didn't exist yet.

The First Generation Computers (1945–1955)

Work on developing mechanical digital computers continued until 1945. These early calculating engines were made up of relays, which were later replaced by much faster vacuum tubes. One of the most remarkable person involved with the first generation computers was Konrad Zuse. In 1936, he completed a relay based mechanical computer named *Z1* in his parent's basement. It was a floating point mechanical calculator which could be programmed using perforated 35 mm films. Later, he designed Z2 and Z3, the latter having 2600 relays, 22-bit word length and operated at frequency of 5-10 Hz [30]. While working on Z4, he found that programming on machine code was hard. He then designed the world's first high-level language named *Plankalkül*

[25].

Meanwhile in United States, J. Presper Eckert and John Mauchly invented the ENIAC (Electronic Numerical Integrator and Comparator) at the University of Pennsylvania in 1945. It was the first general-purpose programmable digital computer. Both the hardware and software of the ENIAC was different from modern computers. It was made up different machines each capable of performing a specific arithmetic operation. It was programmed by wiring a plugboard and three portable function tables. Each function table had 1200 ten-way switches which were used to enter a table of numbers. Programming was done only by the members of the ENIAC team after the program has been thoroughly tested [18]. In 1996, on the 50th anniversary of the ENIAC, using CMOS technology, a 7.4 mm x 5.3 mm chip with the same functionality as the ENIAC was successfully built [34].

EDVAC (Electronic Discrete Variable Automatic Computer), ENIAC's successor, was also designed by Eckert and Mauchly before ENIAC got completed. It was put into operation in 1951. Eckert and Mauchly, and the design team of the EDVAC were joined by the famous mathematician John von Neumann who acted as a consultant. Neumann later wrote a 101 page incomplete monograph titled *The First Draft of a Report on the EDVAC* [41]. In it he described a computer architecture in which program and data are stored in the same memory space. This architecture is called the *von Neumann architecture*. A machine using this architecture is called a *stored program* machine. The first computer to implement von Neumann architecture was the EDSAC at Cambridge. A computer architecture having program and memory in different memory spaces is called the *Harvard architecture*.

It was soon realized that programming computers in machine code by interlinking various arithmetic and logic blocks using a plugboard/control panel was inefficient. Computers produced near the end of this generation had, besides a plugboard, a punch card input-output unit. User, using a modified typewriter, punched holes into cards. These cards were fed into a card reader. The card reader read the cards using optical sensors: light shown on the cards passed through the punches and activated the optical sensor below them. This was interpreted as the binary '1'. The rest of the optical sensors registered a binary '0'. The resulting binary was fed into the main memory of the computer, usually a magnetic drum. Results were punched out on the cards using an opposite mechanism.

An example of a machine which was operated this way was the IBM 650. The IBM 650 was a stored program machine. It had console from which the machine could be programmed in

machine code using octal switches and its state during program execution could be seen. Data was written and read from its magnetic drum memory by the IBM 533 input-output unit using punched cards [10]. Donald Knuth dedicated his magnus opus *The Art of Computer Programming* to an IBM 650 which he used to learn programming while an undergraduate at the Case Institute of Technology [26]. It should be noted that besides the complexity of programming these early computers, the task of debugging a program and fixing the machine was also hard. Considerable time was lost in finding the vacuum tube which has burned out.

The Second Generation Computers (1955–1965)

Invention of the transistor in 1947 ushered a new era in the field of electronics and computer science. Vacuum tubes were replaced by transistors to build amplifiers and digital logic circuits. The small size and high power efficiency of transistors meant that computers, instead of requiring a hall to be fit into, could now be fit into a room. As the reliability of transistors was high in comparison to vacuum tubes, it meant that computers made using them could easily be mass produced and sold without the customers worrying about repairs for a long time. Computers of this generation were called *mainframes* [39]. Two important examples of mainframes from this generation would be the IBM 1401 and 7094 computers.

Around this same time, assembly language gained prominence and high-level languages like COBOL and FORTRAN were introduced. Assembly language is still used today to develop firmware (BIOS, embedded system software). Time critical parts of software used in aeroplanes, fighter-jets, rockets, etc. are still written in assembly language. Inline assembly code can be found in the source code of the Linux kernel. FORTRAN is also still used for scientific calculations in experimental physics and other applied technical domains. It has been called the "mother tongue of scientific computing" [24]. COBOL, on the other hand, is a nearly obsolete language: these days COBOL programs can only be found in mainframes in various business firms. They haven't been replaced by programs written in other languages because the cost of replacing a known technology which has been working reliably for the past five decades is very risky [29].

To program these computers, a programmer would first write the program on paper in either assembly language or FORTRAN. Then, he would type it out on a machine like the IBM 029, which would encode each keystroke on the typewriter by punching the cards. In the end, the programmer would be having a deck of cards having his program. This deck was fed into a card

reader connected to the computer. Here, the deck was read and the program was stored in the main memory of the computer. The computer then executed the program and the results were stored in its main memory. To print out the results, a printer would read the memory and print the output on a roll of paper. The next deck of cards containing another program/job was then loaded and executed by the computer.

It should be noted that while the cards were being read and the results were being printed, the computer was sitting idle. That is, it was doing no data processing during the time it was reading decks, copying them to memory, or printing data from memory. To reduce this waste of time, *batch system* was invented. In it, an expensive computer like the 7094 which was good at numerical calculations was were used solely for executing programs. The task of card reading and writing was relegated to computers like the 1401 which were good at reading cards to tape, copying data and reading tape. A collection of decks, each containing different jobs/programs, was called a batch. It was fed into a 1401 which copied the programs in the batch to a magnetic tape. An operator would then take this tape and connect it to the tape drive of the 7094. The 7094 then ran a special program which would read each program on the first tape one-by-one and write their output to a second tape instead of printing them out. This program was a precursor to modern operating systems. When a batch of job had been executed, the second tape was taken by an operator who would connect it to the tape drive of the 1401. Its contents were read and printed on a roll of paper. Meanwhile, another operator would connect a tape containing another batch of job to the tape drive of the 7094 [39].

The Third Generation Computers (1965–1980)

Miniaturization of computers continued with the introduction of integrated circuits (ICs). The first monolithic IC was invented by Robert Noyce in 1959. The first prominent use of integrated circuits was made in the construction of the Apollo Guidance Computer (AGC). Consultants hired by NASA mathematically proved that a computer made using ICs, transistors, and diodes would have a very high probability of failure. Their calculations used the assumed failure rate of these components. Eventually, Eldon C. Hall, leader of the hardware designing team for the AGC at MIT and an advocate for the use of integrated circuits, convinced NASA that such mathematical calculations were bogus as they were not using the actual failure rate of the aforementioned components. Also, measuring the actual failure rate of the components would take years which would put them a step back in the space race. He eventually prevailed and the AGC was built

using the latest semiconductor technology. The computer's main memory was made out of core rope. On it was stored the operating system which controlled various systems of the command and lunar module. It had a scheduler which executed various programs based on their priorities and in case of a failure, it saved the state of the machine, and restarted it. Both the hardware and the software of the computer proved to be highly reliable and had a 100% success rate [23].

Meanwhile, IBM was dealing with the complexity of producing, marketing and supporting two completely different lines of computers: commercial computers, like the 1401, and scientific computers, like the 7094. Both lines had different architecture and instruction set. Customers who wanted to shift from lower end to higher end computers found it challenging to port their entire code base to a newer machine. IBM attempted to reduce this complexity by releasing a single line of computers, the System/360. The machine could be used for both commercial and scientific computing. It was all a matter of the configuration the user wanted. As the architecture and the instruction set of the machine was same for all the lines, users no longer had to worry about porting their code base from one line to another [39].

IBM introduced a new operating system, OS/360, for its System/360 line. It was able to keep the CPU busy 100 percent of the time by assigning it jobs to do while the job it was currently on was waiting for I/O to complete. Each job was stored in its own memory partition and special hardware was used to prevent one job from altering the contents of others. This was called multiprogramming [39].

Though OS/360 introduced several new concepts, it was still fundamentally a batch system. To reduce the time for which programmers had to wait to get their job done, the concept of timesharing was introduced. The most popular timesharing operating system was the Compatible Time Sharing System (CTSS). It was developed at the MIT Computation Center and used a modified IBM 7090 mainframe. It had a kernel, which was the program responsible for scheduling between different users, had system calls which could be called by software interrupts, and implemented memory protection schemes. The scheduler had a quantum time unit of 200 ms, and was responsible for deciding the user who was to be allotted computer time, the amount of time to be allocated, keeping track of activities of all users, and the net charges for the user. The filesystem of CTSS gave each user their own directory. Users would issue commands using a teletype-writer (also called a terminal) like the ASR 33. When the scheduler executed a user's command, the output was displayed on the paper in the typewriter [31][39].

Success of the CTSS made MIT, Bells Labs, and General Electric join hands to develop a *com-*

puter utility. The idea was that just electricity and water are utilities which people access by tapping into some main channel, similarly computers should be a utility. The designers imagined a single powerful computer providing computing power to users in a large area. The computer of choice was a GE-645 mainframe and the operating system designed was called MULTICS (MULTiplexed Information and Computing Service). MULTICS introduced several novel ideas at that time but overall it was a disaster. The enormous size and complexity of the system which was mostly written in PL/I language, compiler for which barely worked, made MULTICS unusable. Ken Thomson referred to MULTICS as "*overdesigned and overbuilt and over everything. It was close to unusable. They [Massachusetts Institute of Technology] still claim it's a monstrous success, but it just clearly wasn't*" [35]. General Electric quitted the computer business and Bell Labs withdrew its efforts into developing it. MIT persisted and developed a functional version of MULTICS by mid 1970's which was installed in over 80 institutions around the world [39].

Around this time, minicomputers were rising in popularity. These computers were smaller and cheaper than a mainframe. Digital Equipment Company (DEC) Programmed Data Processor (PDP) line of minicomputers was extremely popular. Ken Thompson, who had earlier worked on the MULTICS project, started developing an operating system for a PDP-7 so that he could play *Space Wars* on it [40]. He was soon joined by Dennis Ritchie. Their goal was to design an operating system for a single user which had all the good parts of MULTICS and none of its bad parts. The resulting operating system was called UNIX which was a pun on the name MULTICS. On the influence of MULTICS on UNIX, Thompson had the following to say: "*the things that I liked enough (about Multics) to actually take were the hierarchical file system and the shell — a separate process that you can replace with some other process*" [35].

UNIX quickly became popular. It was soon decided that UNIX should be ported to a newer and more powerful PDP-11. However, as most of it was written in assembly language which was tied to the architecture of PDP-7, a need for developing a new systems programming language was felt. This prompted Dennis Ritchie to write a new programming language, C, which was specifically designed to be small, portable, and powerful. Ritchie, along with Brian Kernighan, wrote *The C Programming Language*, 2nd (and last) edition of which is still considered as the *de facto* standard for ANSI C and for writing technical books [32].

The Fourth Generation Computers (1980 - Present)

Development of a complete processor on a square centimetre of silicon was made possible with large scale integration (LSI) technology. These processors on a single integrated circuit were called microprocessors, and the computers using them were called microcomputers. Architecturally, both minicomputers and microcomputers were the same. However, when it came to price, microcomputers were significantly cheaper. Further improvements in miniaturization of technology allowed computer manufacturers to get rid of mechanical components and replace them with electronic ones. An example would be the replacement of a teletype-writer with a keyboard and a monitor. A software called a terminal emulator is used to implement an environment in which commands and their outputs are visible on the screen.

Computers like the Commodore VIC-20 and Apple II started the microcomputer revolution. The competition between different companies meant that affordable computers were available to the home user. As these computers were intended for use by a single user, they were often called personal computers, a term which still persists till this day.

The 8086 microprocessor was released in 1978 and its cheaper version, the 8088, was released in 1979. IBM decided to enter the personal computer market by building a microcomputer around the cheaper 8088. The result was the IBM PC, which debuted on August 12, 1981 [1] and was cloned by competing manufacturers in the following year. The IBM PC soon became the *de facto* standard for PC, a fact which is true for modern PCs as well.

The original IBM PC (the one using 8088 microprocessor) came with two floppy disk drives, a port for attaching cassette tape recorder, and an optional hard disk drive. Software companies started producing disk operating systems (DOS) which came in floppy drives. User had to insert the floppy drive before turning the computer on. A few examples of DOS were CP/M-86 by Digital Research, 86-DOS by Seattle Computer Products, and MS-DOS by Microsoft. MS-DOS was created by the author of 86-DOS, Tim Patterson.

Up till UNIX Version 6, the source code of UNIX was widely available with AT&T license and was frequently studied in universities. Berkley University modified their copy of source code to a great extent to make UNIX suitable for their own requirements. AT&T (Bell Labs were a part of it) realized that UNIX can be commercialized. They issued Version 7 with a license which prohibited universities from studying its source code. These led to a legal dispute between AT&T and Berkley University. Meanwhile, Andrew Tannenbaum wrote his own version of UNIX from scratch for use in the operating systems course he taught at Vrije Universiteit

Amsterdam. He called it MINIX and it, along with his book *Operating Systems: Design and Implementation* soon became popular [39].

By early 1990's, IBM PCs with 80386 (or i386), which was a 32-bit processor with hardware for multitasking, were becoming popular. MINIX was originally designed for the original IBM PC, so it could not use the full potential of the newer, powerful hardware. Seeing these limitations of MINIX, Linus Torvalds, then a student at University of Helsinki, developed an operating system called Linux which was inspired from MINIX. It used MINIX's filesystem but had a monolithic architecture (as opposed to MINIX's microkernel), and was released in August of 1991. Since then, it had become the largest open source project in the world. Linux based distros are used on servers, smartphones, personal computers, etc [40].

1.2.4 The 8086 Microprocessor from a Programmer's Point of View

The 8086 microprocessor was launched on June 8, 1978. It subsequently gave rise to the x86 family of processors which have been the most successful processors till date. Though the original IBM PC had an 8088 instead of an 8086, we will restrict our discussion to the PCs using 8086 as that is more relevant.

Physical Specifications

The 8086 microprocessor was available as a 40 pin DIP IC, often in ceramic casing. Its pin configuration is shown in Figure 1.3.

The 8086 microprocessor can be powered using 5 volt power source. Its standard operating speed is 5 MHz, but versions with clock rate of 8 MHz and 10 MHz are available.

The processor has a multiplexed 20-bit address bus and 16-bit data bus: 16 pins are shared between both the address and data bus. The remaining higher order 4 bits in the address bus come from physical address calculation by the bus interface unit (BIU).

The processor can be operated along with other processors by putting it in maximum mode. This is done by strapping its MN/$\overline{\text{MX}}$ pin to the ground. In this mode, CPU encodes its signal, which are decoded and used by the 8288 bus controller to control the system. To put the processor in minimum mode, MN/$\overline{\text{MX}}$ pin is strapped to +5V.

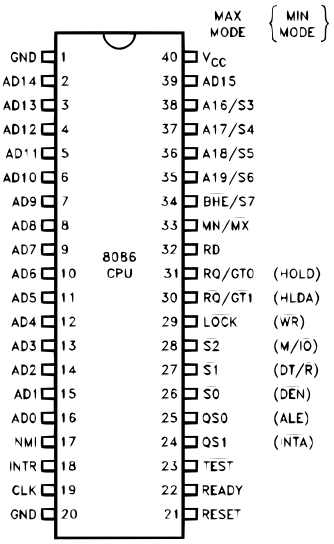


Figure 1.3: Pin configuration of the 8086 [3]

Execution Unit and Bus Interface Unit

In section 2.3.3 and 2.3.4 we discussed how the efficiency of computation by mainframes was maximized by using batch system and multiprogramming. The idea was to minimize the time for which the processor was idle by giving it enough jobs/programs to keep it occupied while the I/O operations were performed by cheaper machines. The concept of batch system and multiprogramming were applied in the design of the 8086 microprocessor. Instead of having a single processing unit which was responsible for fetch and writing data and instructions and execution, the 8086 has two processing units: the execution unit (EU) and the bus interface unit (BIU). The EU is responsible for executing the instructions and the BIU is responsible for fetching instructions, reading operands, and writing data to memory or to the ports. The two processors operate independently of each other thus ensuring that the EU is in operation for most of the time. The BIU has an internal buffer in which it would prefetch and store the instructions so that the EU's time is not wasted by waiting for the next instruction. This buffer is basically a queue. The size of this queue is four bytes for the 8088 and six bytes for the 8086 [3].

Registers

Two sets of registers are provided to both the EU and the BIU. The 16-bit registers provided to the EU are called general purpose registers. The 16-bit registers provided to the BIU are called segment registers. A 16-bit instruction pointer is also provided to the BIU.

General purpose registers are frequently used in arithmetic and logical calculations, and com-

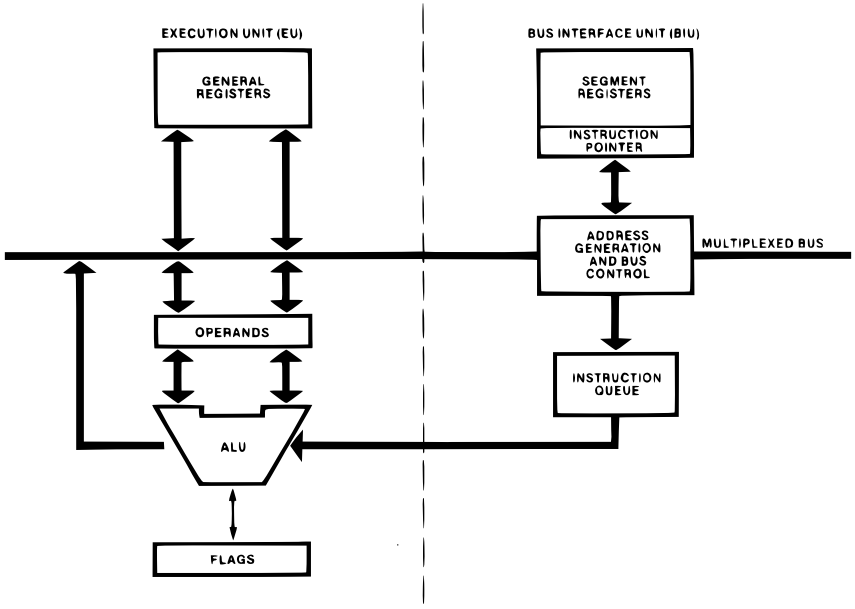


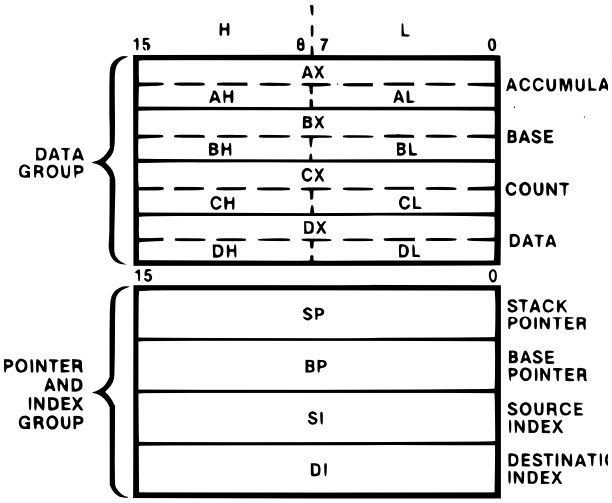
Figure 1.4: Block diagram of the 8086/8088 microprocessor [2]

parisons, and also for addressing data in memory. These are `ax`, `bx`, `cx`, `dx`, `sp`, `bp`, `si`, and `di`*. Each of `ax`, `bx`, `cx`, and `dx` could be split into two 8-bit registers thus providing a total eight 8-bit registers. This is shown in Figure 1.5a. `ax`, `bx`, `cx`, and `dx` are used in computations and comparisons. One can say that there are four 16-bit accumulators in the 8086 processor. The `ax` register along with `dx` is used to store the results of operations. After 16-bit multiplication, the product's low word is stored in `ax` and high word is stored `dx`. After 16-bit division, the quotient is stored in `ax` and the remainder in `dx`. `cx` register is used as a counter in loop operations.

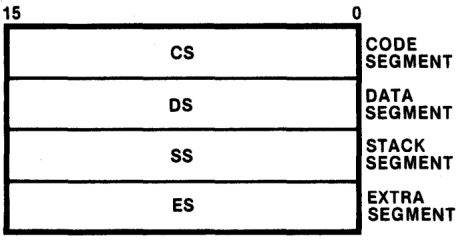
The stack pointer register, `sp`, and the base pointer register, `bp`, are used for setting up stack frames and for accessing parameters passed to the functions. The source index register, `si`, and the destination index register, `di`, are used to provide the index of a single byte (or word) of source and destination strings, respectively.

`bx` and `bp` are used to provide the base address of a data structure or the stack frame. `si` and `di` are used to index through these data structures.

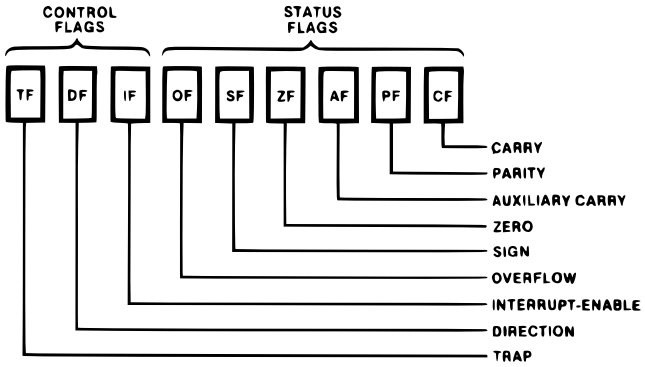
*In older documents and source codes, one would find that instructions, operands, and hexadecimal numbers were exclusively written in uppercase. Nowadays, assembly source codes are written both in lowercase and uppercase literals. Due to the availability of different typesetting software, many newer documents also make use of both the cases for instructions, operands, and hexadecimal numbers. In this report and the source code for this project, instructions, register names, and hexadecimal numbers are exclusively written in lowercase. A breaks in this convention might be observed, especially at places where older documents are referenced, and in figures. In the end, the case which appears to be pleasant to the eyes and is relevant is used.



(a) General purpose registers



(b) Segment registers



(c) Flags

Figure 1.5: 8086 registers [2]

Segment registers are the "hack" which allow programmers to access 1 MiB of memory instead of mere 64 kiB. There are four 16-bit segment registers which are available to the BIU: code segment (`cs`), data segment (`ds`), extra segment (`es`), and stack segment (`ss`) registers (Figure 1.5b). The purpose of segment registers is now presented.

A 16-bit register can address only 64 kiB of memory, with addresses of bytes going from 0x0000 to 0xffff (0 - 65535). If a 16-bit base value is left shifted by 4 bits and added to another 16-bit number, the resulting sum is a 20-bit number which can have any value in the range 0x00000 - 0xfffff (0 - 1048575). A 20-bit variable can address 1 MiB of memory. This is exactly how *physical addresses* are generated from *effective addresses*. In 8086, memory is divided in chunks, with the maximum size of chunks being 64 kiB. These chunks are called segments. The address at which a chunk starts is called segment base address. Segment base addresses are right-shifted

by 4 bits and stored in segment registers. The effective address of a data or instruction byte is the offset at which that byte is located from the segment base address. Effective addresses can be stored in a general purpose register (in the case of data) or in the 16-bit instruction pointer, `ip` (in the case of an instruction). To access the given data/instruction in a block, the BIU left-shifts the value stored in the segment register by 4 bits and to it adds the 16-bit effective address. The resulting 20-bit number is the physical address of the required data or instruction byte. The following relations will clarify the process of physical address generation.

1. physical address of an instruction = (`cs` $<<$ 4) + `ip`
2. a. Byte or word in source string = (`ds` $<<$ 4) + `si`
   b. Byte or word in destination string = (`es` $<<$ 4) + `di`
3. physical address of a data byte = (`ds`/`es` $<<$ 4) + base address (`bx`/`bp`)
   + index (`si`/`di`)
   + diplacement (8-bit or 16-bit)
4. address of the base of a stack frame = (`ss` $<<$ 4) + `bp`
5. address of the top of a stack frame = (`ss` $<<$ 4) + `sp`

As the value of lower nibble of the 20-bit segment base address is always zero, therefore, two segments have at least a gap of 16 bytes between them.
Besides the general purpose registers, the EU is also provided with a flag register having status and control flags (Figure 1.5c). Status flags contain the represent the state of the machine after last operation (overflow, parity, carry bits, etc.). Control flag alter the way programs are executed by (direction of increment of index registers, interrupts, and single-step mode).

Endianness

The order in which a processor stores the high byte and low byte of a word is called *endianness*. There are two types of endianness: little endianness and big endianness. If a processor stores the least significant byte at a lower address and most significant byte at higher address, then it is said to follow little-endian byte ordering. if a processor stores the most significant byte at a lower address and least significant byte at higher address, then it is said to be using big-endian byte ordering. 8086 follows little-endian byte-ordering. It should be noted that little-endian ordering is not applicable for values stored in registers. Therefore, if register `ax` stores

the number 0x1234, then `al` will be storing 0x34 and `ah` will be storing 0x12. On the other hand, if a memory location, say 0x1000 is said to store 0x1234, then 0x34 will stored at the address 0x1000, and 0x12 will be stored at the address 0x1001 (Figure 1.6). Another type of

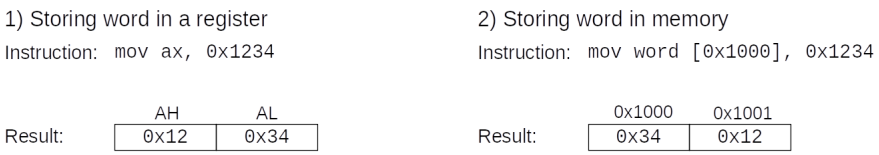


Figure 1.6: Ordering of bytes of words in registers and memory

data variable is a doubleword. A doubleword, as the name suggests, consists of two words and is used to store memory addresses. Therefore, it is often called as a pointer. As has already been discussed in section 2.4.3, physical address is made up of segment base address and an offset value. A doubleword's most significant word contains the segment base address, and its least significant word contains the offset of the data/instruction to be accessed. A doubleword is stored in memory by storing the least significant word (which contains the offset) at the lower address, and then storing the most significant word (which contains the segment base address) (Figure 1.7). This storing convention is most often used in intersegment calls where the calle's

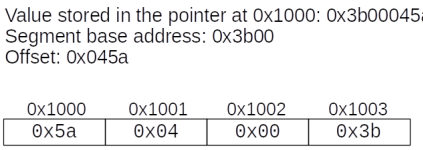


Figure 1.7: Storing doublewords in memory

segment base address are first pushed onto the stack and the offset is pushed afterwards. After this, the far call instruction is executed, which pushes the current `cs`'s and then `ip`'s value onto the stack This is shown in Figure 1.8. From the figure note that the stack grows downwards towards the base address of the segment in which the stack is kept.

Addressing Modes

Programming in 8086 assembly language is easier and enjoyable in comparison to programming in 8085's (or earlier microprocessor's) assembly language. This is because of the availability of powerful arithmetic, loop and string instructions, as well as the presence of different types of addressing modes. The addressing modes of 8086 are given below:

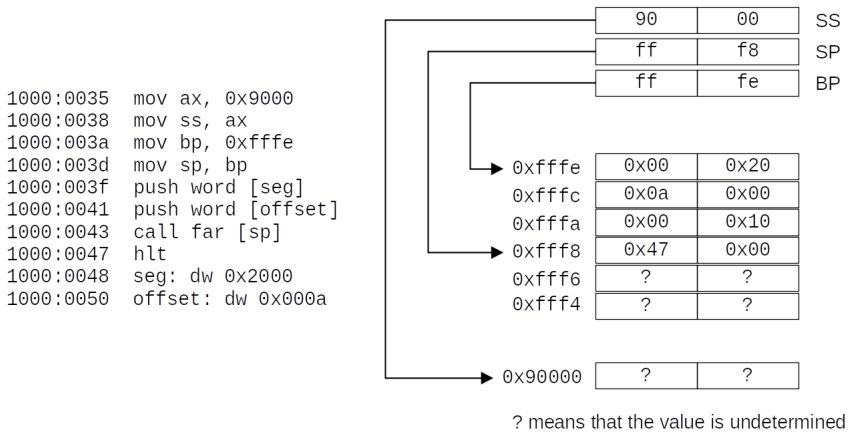


Figure 1.8: Storing pointers on stack during intersegment calls

1. *Immediate Addressing*: Operand is provided in the instruction itself.
   Example: `mov ax, 0x1234`.

2. *Register Addressing*: Both the operands are registers. Example: `xor cx, cx`.

3. *Direct Addressing*: Effective address is provided in the instruction itself.
   Example: `add ax, word [var1]`.

4. *Register Indirect Addressing*: The effective address is stored in base registers (`bx` and `bp`) or index registers (`si` and `di`). Example: `mov al, byte [si]`.

5. *Based Addressing*: The effective address used in the instruction is the sum of the value stored in a base register and a displacement. Using `bp` as the base register allows us to access parameters of a function. Using `bx` as the base register allows us to access data in a structure. Example: `mov ax, word [bp + 2]`.

6. *Indexed Addressing*: Effective address is calculated from the sum of displacement and the value stored in one of the index registers. This is useful for accessing elements of an array. Example: `mov dl, byte [si]`.

7. *Based Indexed Addressing*: The effective address is computed by adding a displacement, the value stored in a base register, and an index register. This addressing mode is useful in accessing elements of a parameter which might be a data structure. Example: `mov dl, byte [bp + si + 4]`.

8. *String Addressing*: String instructions involve the use of string registers (`si` and `di`) and instructions. When string instructions are executed, it is assumed that `si` is a pointer to the first byte or word of the source string, and `di` is a pointer to the first byte or word of the destination string. Either `si`, or `di`, or both are either incremented or decremented depending on whether the direction flag is cleared or set, respectively. Example: `repne lodsb`.

9. *I/O Port Addressing*: Peripherals which have port addresses are accessed using `in` and `out` instructions. In direct port addressing, the 8-bit port address is provided with the instruction itself. The value being read or written is contained in `al`. In indirect port addressing, the 16-bit port address is present in the register `dx`. Example: `out 0x70, al`.

1.2.5 Design of the original IBM PC

The original IBM PC was released on August 12, 1981. Thanks to its cheaper price, easily available hardware and software, and marketing, the IBM PC standardised the personal computer market. Although IBM no longer manufactures PCs, the IBM PC and its successors are still the *de facto* standard for PCs [1].

Physical Specifications

Due its lower price and IBM's familiarity with it, the original IBM PC contained the 8088 microprocessor with a clock rate of 4.77 MHz. Clones of the IBM PC were released less than an year later, and some of them, like the Amstrad PC1512, had the 8086 microprocessor.
Both 8088 and 8086 can access 1 MiB of memory. However, as this memory was fairly large and expensive in the early 1980s, most PCs came with 16 kiB to 640 kiB of memory. Expansion slots were often provided for the user to increase the memory of the system themselves.
The original IBM PC came with upto two 5.25" 160 kiB floppy disk drives. A disk-drive adapter was provided to interface with the disk drives. Cassette tape recorder and hard disks were also supported. Later versions of the IBM PC came with hard disk drives.
Addressing of storage disks was done using cylinder-head-sector (CHS) addressing. A single disk or a platter, in general, has two readable sides. A pair of heads were used to read and write to the disk. A single side of the disk was divided into tracks. A single track is contained within two concentric hollow cylinders and is made up of sectors, with each sector having 512 bytes of memory (Figure 1.9). Numbering of sectors start from 1, but the numbering of cylinders and heads starts from zero. The equation given below gives us the total size of the disk.

$$M = M_s \times N_s \times N_t \times N_h$$

where

$$\begin{aligned} M &= \text{total storage capacity of the disk} \\ M_s &= \text{storage capacity of a single sector} \\ N_s &= \text{total number of sectors per track} \\ N_t &= \text{total number of tracks per head} \\ N_h &= \text{total number of heads per disk} \end{aligned}$$

Consider the case of a standard 3.5” floppy disk. It has the following specifications:

$$\begin{aligned} M_s &= 512 \text{ bytes/sector} \\ N_s &= 18 \text{ sectors/track} \\ N_t &= 80 \text{ tracks/head} \\ N_h &= 2 \text{ heads/disk} \end{aligned}$$

which gives us

$$\begin{aligned} M &= 512 \times 18 \times 80 \times 2 \\ &= 1474560 \text{ bytes} \\ &= 1.44 \text{ MiB} \end{aligned}$$

Modern operating systems frequently use logical block addressing (LBA) instead of CHS.

For input, a standard IBM model F keyboard with 83 keys and five-pin connector was provided.

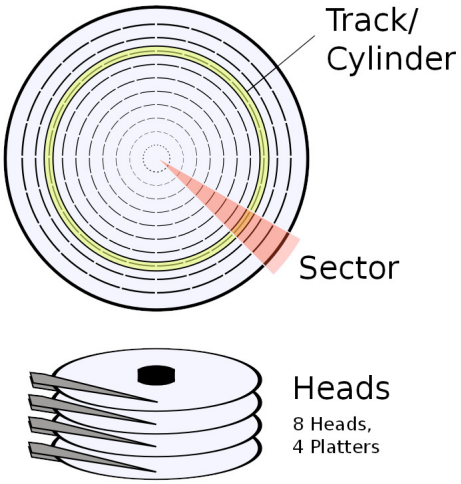


Figure 1.9: Cylinder/tracks, sectors and heads on a hard disk platter [28]

The keys pressed on the keyboard were mapped to the characters present in the ROM of the CGA

card. The characters present in the ROM were printable ASCII characters, diacritics, some Greek letters, and a few graphic characters. This character set present in the ROM was called IBM437 or "code page 437". Later models of IBM PC were shipped with VGA cards which supported a resolution of 640x480 pixels and had a 16-bit color palette.
To generate sounds, an 8254 programmable interval timer was used to generate square waves which were fed into a piezoelectric speaker.

BIOS

The term BIOS, standing for Basic I/O System, was coined by Gary Kildall. It is the firmware which is responsible for loading the bootloader from storage into the main memory. BIOS is stored in ROM and is the first software which is read by the CPU. Besides being responsible for loading the bootloader, the BIOS also provides the bootloader and real-mode OSs with several basic subroutines* which allow them to access the hardware. These subroutines are called BIOS interrupts. BIOS interrupts are called by initializing the registers with appropriate values and issuing a software interrupt using `int` instruction. For example, to print a character on the screen, BIOS interrupt 10,0E* is used. To use this interrupt, 0x0e is stored in `ah` and the hex-code (from code page 437) of the character to be printed is stored in `al`. Then a software interrupt with interrupt number 0x10 is issued. The processor executes the ISR corresponding to interrupt 0x10 with `ah` = 0x0e in the BIOS and prints the character in `al` on the screen.
The BIOS in the original IBM PC was reverse engineered and cloned by other manufacturers so as to make their PCs compatible with the IBM PC. BIOS is slowly fading out of use in favour of UEFI. This is mainly because it is unsecure and is not supported in protected (32-bit) and long (64-bit) mode. *Ralph Brown's Interrupt List* provides hobbyists and professionals with a comprehensive list of BIOS interrupts [15].

Boot Process

When an IBM PC compatible computer is powered on or if the reset button is pressed, the BIOS performs a system check which is called power-on self-test (POST). POST involves initialization the main memory (RAM) and few important peripherals (e.g., keyboard, screen, speakers,

*The words "procedure", "subroutine", and "function" have the same meaning and are used inter-changeably throughout this report.

*BIOS calls are mentioned in this report using INT N,F format. Here, N is the BIOS call number in hex-format, and F is the service which is required. F is usually an 8-bit number stored in `ah` before INT N is executed.

disk drivers). If there is any error during POST, the BIOS informs the user about the same by generating beep codes from the speaker. Once the POST has been successfully completed, INT 19h is used to detect and load the bootloader from a bootable disk. A disk is considered to be bootable if the last word of its very first sector (track = 0 and head = 0) contains the magic number 0xaa55. As this sector also contains the bootloader, it is also known as boot sector or master boot record (MBR). Upon detecting the disk to be bootable, the BIOS loads the content of the MBR at address 0x7c00. The IP jumps to this address and CPU starts executing the instructions which make up the MBR. The bootloader might be simple enough to be contained in 512 bytes, or it might be complicated enough to be contained two sectors. In the latter scenario a two-stage bootloader is used. The boot sector code loads more sectors, and once the bootloader has been loaded, the kernel and then the operating system is loaded into the memory using BIOS calls.

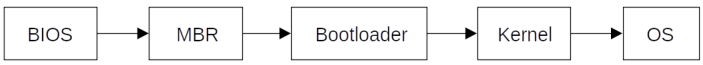


Figure 1.10: Boot sequence of an IBM PC

Memory Map

Memory map is a diagrammatic/tabular method of showing the contents of physical memory of a computer system. In the case of the IBM PC, we are mainly interested in the memory map after booting has been completed. Therefore, by the term memory map we specifically mean the contents of memory after boot-up sequence.
The memory map is shown in Figure 1.11.

As has already been discussed, the maximum size of a segment in 8086 is 64 kiB. Total addressable physical memory is 1 MiB. This memory is divided into continuous 64 kiB segments. Each of these segments is called a block. The blocks are differentiated from each other by the value of the most significant nibble of their segment base address. A block have segment base address equal to 0xn0000 is called block n. For example, the block starting at address 0x50000 is called block 5. Block 0 to block 9 make up 640 kiB of memory and together make up the *low memory area* or *conventional memory area* (LMA). Block A till F make up the *upper memory area* (UMA) which has a size of 384 kiB. The blocks and the purpose they are usually used for is presented in Table 1.2. Note that the blocks extended BIOS data area (EBDA) occupies varies

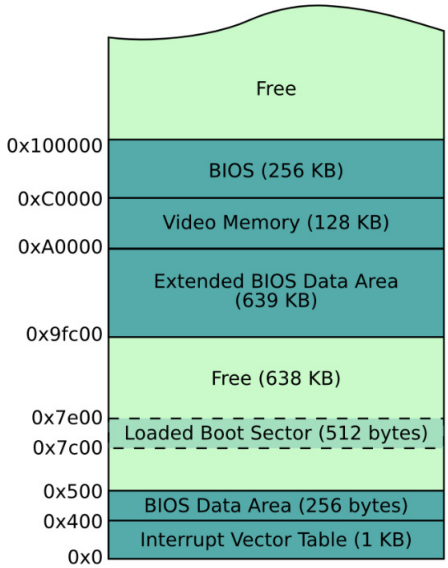


Figure 1.11: Typical memory map of an IBM PC after boot [14]

from system to system and is, therefore, not mentioned in the table. However, EBDA is generally placed before block A.

Table 1.2: Typical Memory Map of an IBM PC after boot-up sequence

Block	Start	End	Description
0	0x000	0x3ff	Interrupt vector table (IVT)
	0x400	0x4ff	BIOS data area (BDA)
	0x500	0x7bff	User memory
	0x7c00	0x7dff	Bootloader
	0x7e00	0xffff	User memory
1	0x10000	0x1ffff	
2	0x20000	0x2ffff	
3	0x30000	0x3ffff	
4	0x40000	0x4ffff	
5	0x50000	0x5ffff	
6	0x60000	0x6ffff	
7	0x70000	0x7ffff	
8	0x80000	0x8ffff	
9	0x90000	0x9ffff	
A	0xa0000	0xaffff	Video memory
B	0xb0000	0xbffff	
C	0xc0000	0xcffff	Mapped to ROM and hardware
D	0xd0000	0xdffff	
E	0xe0000	0xeffff	
F	0xf0000	0xfffff	

1.2.6 Deficiencies of Model Curriculum for Engineering and Technology Developed by AICTE

We discussed in section 1.1 on how engineering is a practical profession which necessitates that engineering students should have practical knowledge of the field they establish their career in. With the aim of getting students employed in an ever-changing industry, it becomes increasingly hard to keep the syllabi appropriately updated. Neither the institutions responsible for maintaining and enforcing the curriculum, nor the students can be entirely blamed for the lack of practical knowledge which can be found among many freshers across the globe. A discussion of

the global technical education scenario would increase the length of the present chapter which is already beyond the usual length. Therefore, we will restrict our discussion to the brief analysis of engineering education in India. This will be appropriate as:

1. All the authors of this report bear the nationality of Indian.

2. It is estimated that India will be the leading producers of engineers by the end of the current and the coming decades [36]. This makes it important that the quality of engineers being produced is great.

It is a well known fact that engineering education in India is not at par with the requirements of the industry. This fact has been pointed out by many people over the years, more recently by Steve Wozniak who pointed out that innovation has taken a backseat in pursuit of employability [8]. Such criticism often results in a blame game. Sometimes counterarguments also blame the British rule. The fact that after more than half a century of independence, we, as a nation, have failed to nurture and promote original thinking and creativity is disregarded. The number of jobs related to computer science and IT are ever increasing. Students in pursuit of getting employed often curb their interests and enrol in courses related to computer science and IT [9]. Innovation and practicality go hand in hand. The saying "*Necessity is the mother of innovation*" is an adage to this. Without getting their hands dirty, a person can never know about the requirements of a particular field. Therefore, it becomes necessary to ensure that students are dispensed practical knowledge in as many areas as possible. By doing so, they can determine their interests and contribute to the fields they like. The best way to dispense practical knowledge is by incorporating project-based learning in the curriculum for technical education.
Though the model curriculum provided by the All India Council for Technical Education (AICTE) is developed with the help of academic and industry experts [12], they have some glaring flaws, three of which are stated below.

1. *Undirected and overloaded course structure*: A single course consists of many theoretical sub-topics without any practical problems for students to work through. A student can solve tons of problems on Kirchhoff's rules however their ability to synthesize a circuit is equally important. As the project presented in this report is related to computer architecture and operating systems, we will take the example the course on Computer Architecture offered to Electrical Engineering undergrads. There are six modules in this course. Module 1 tells about the differences between CISC and RISC, data types, system buses, and

multi-bus organization. Module 2 then jumps to memory organization, followed by Module 3 discussing I/O devices, PCI and PCI Express bus. Module 4 then discusses x86 architecture, and its addressing modes and instruction. Module 5 discusses instruction level pipelining (ILP) and module 6 discusses other architectures like MIPS. The course, and other similar courses, can be considered to be well designed theoretically. However, they are overloaded with theoretical concepts. It would be better to just focus on the architecture of single computer, preferably IBM PC. The course can then describe the 8086 microprocessor and later describe ILP, and paging. Then the architecture of the IBM PC can be dealt with in detail.

2. *Impractical course structure*: Continuing with the example provided previously, most of the courses dealing with microprocessors start of with 8085 microprocessor. Though it is still used in legacy systems in the industry, no new projects are developed using it. Students would be better served by teaching them about ARM or RISC-V or x86 architectures as most of the computing systems today are built around them. Furthermore, it must be noted that the practical assignments given to students are trivial. Examples include writing subroutines to multiply numbers, or to move strings in 8086's assembly language. A more practical exercise would be to write subroutines to print a string on the screen or to separate words out from a string.

3. *Obsolete software tools*: Most of the software tools used in colleges are obsolete. Courses on C programming and data structures still require students to develop and run code in Turbo C environment. This is ridiculous because Turbo C's compiler does not follow the standard implementation of C and C++. In the course on microprocessors, codes are run on expensive trainer kits or on emulators, like emu8086, which is not the industry standard for developing 8086 assembly code.

Remedying the current problems will take years. However, one should not be daunted by this challenge. Baby-steps are what is required. The project presented in this report is one such baby-step. It is hoped that it will assist in nurturing students who are interested in computer architecture and they will not have to face the problems which the author of this project (Kapoor) had to face.

1.3 Objective

The objective of this project was to develop a simple real mode operating system from scratch for educational purposes. This was objective was achieved by completing the following sub-objectives:

1. To develop a bootloader from scratch.
2. To develop a set of libraries containing certain standard functions found in C's `stdio.h` and `string.h` libraries. This would assist the user in building disk applications for the system.
3. To develop a shell which enables the user to issue basic commands to interact with system components like disks, APM, etc.
4. To develop a file system using contiguous sector allocation. The file system should be simple enough to understand, implement and use with programs like the line editor.
5. To develop a build environment which allows the user to write and build their own disk applications, and run them using the operating system.

1.4 Overview

In the present chapter we discussed about our motivation behind developing a simple operating system. In chapter 2 we discuss the implementation of CrazyOS. We have explained the source tree of the parent and the project directory along with the library functions.
In Chapter 3, we have provided the results of this project. Screenshots of the operating system running on an emulator are presented to the reader. The only three known bugs in the operating system are also discussed.
In Chapter 4, we have given a few concluding remarks and have discussed the future scope of this project.
The report ends with the presentation of the references, research paper, and resume of the authors.

CHAPTER 2: IMPLEMENTATION OF CRAZYOS

2.1 Introduction

In the previous chapter we discussed our motivations behind creating a very basic operating system, CrazyOS. We described it as a single-user, single-tasking, real mode operating system. These three terms are described as follows:

1. *Single-user*: Most modern operating systems allow multiple user accounts. Each user has some disk space allocated to them and they can use it as they like. They can keep their personal files and projects separate from others work. Each user account is accessible through a password. CrazyOS does not support multiple users. It does not have the capability to communicate through a network with other computers. It is intended for one user only. No password logins are required to access the services of the system or to access the contents of a file.

2. *Single-tasking*: Modern operating systems allow users to run multiple programs at once. Each program is given its own address space to run without messing with other programs. Various scheduling algorithms, for example round-robin scheduling algorithm, are used to allocate a quantum of CPU's time to programs so that each program gets executed and give the user a feel of simultaneous execution of multiple programs. Threads and cores are also allocated to different batches of programs. CrazyOS is simple: it runs on a single processor and only a single program/process, which is the shell, is running most of the time. Other programs can be run either by passing commands from the shell, or by loading and running them from the disk.

3. *Real mode*: Modern operating systems run on 64-bit CPUs. One such CPU is the x86-64 family of processors produced by both AMD and Intel. These processors use 64-bit registers and address space and use paging (i.e., creation of virtual address spaces) to protect programs from each other. To provide backwards compatibility to programs which would run only on 32-bit and 16-bit machines, x86-64 processors initially run as 16-bit processors when the machine has just been powered on. In this mode, they try to act like an 8086 processor. This mode of operation is called *real mode* or *virtual 8086 mode*. After

going through a complex setup procedure, the user can use the processor in 32-bit mode which is called *protected mode*. After some more setting up, the user can finally run the processor in 64-bit or *long mode*. In this mode, programs can utilize all the capabilities of the processor. Most modern operating systems run in long mode. However, to reduce complexity of its design and to make it relevant for use as a tool in courses on computer architecture and 8086 microprocessor, CrazyOS runs in real mode or virtual 8086 mode only.

The implementation of CrazyOS is presented in this chapter. We will begin by describing the build environment and organization of the source files. Integral components of the operating system are then discussed in detail.

2.2 Set-up of the Development Environment

The development environment of CrazyOS consists of five software tools: code editor, assembler, emulator, build automation software, and source version control software. We will now explain the tools of choice for this project.

2.2.1 Code Editor

Earlier versions of the operating system were written using gVim. As the size of the project grew, the author found it difficult to maintain files using gVim. Therefore, a switch to VScode was made and it has since become the default code editor for this project. *x86 and x86_64 Assembly* syntax highlighting extension was downloaded as it supported NASM's syntax. Github's dark theme is used as the author find it to be pleasing to his eyes.

2.2.2 Assembler

Many assemblers supporting the x86 architecture are available. The author had the choice of using either GNU AS or NASM. GNU AS is a part of GCC and supports multiple architectures. Due to this reason, it is used for developing and maintaining assembly source files in many projects such as the Linux kernel. However, it by default uses AT&T syntax which is different from Intel syntax, the latter being found in datasheets and software developer's manuals provided by both AMD and Intel. Few of the differences between AT&T and Intel syntax are listed below:

1. Intel syntax requires that after writing the instruction, the user has to first mention the destination operand and then the source operand. AT&T syntax requires that the source operand is to be mentioned first. Therefore, `mov eax, ebx`, written in Intel syntax, and `movl %ebx, %eax`, written in AT&T syntax, are both equivalent to each other.

2. Register names are prefixed with `%`, immediate and static operands with `$` and hexadecimal numbers are prefixed with `0x`.

3. Size of the operands is to be mentioned by suffixing a literal(s) with the instruction mnemonic. Three most commonly used literals are `b` for byte operands, `w` for word operands, and `l` for doubleword operands.

Therefore, `mov byte [bx], al`, written in Intel syntax, and `movb %al, (%bx)`, written in AT&T syntax, are equivalent to each other. Being a beginner himself, the author found out that it takes sometime to get used to AT&T syntax. Furthermore, GNU AS partially supports Intel syntax and its documentation was found to be bit terse at places. Therefore, the author decided to use the Netwide Assembler, or NASM as it is usually called, for assembling the source files. NASM supports only x86 and x86-64 architecture and by default uses Intel syntax. It has a rich set of preprocessor directives which make it easier to include files and write macros. It can output assembled files in many different formats, such as ELF, plain binary, COM, etc. An excellent documentation is also provided by the NASM development team [7].

2.2.3 Emulator

The author had a bad experience with emu8086 which is widely used in courses dealing with 8086 microprocessor in Indian colleges. The emulator comes with a free trial period of 14 days after which a license needs to be purchased. The emulator's free trial ended on the day of practical examination, which infuriated all of his colleagues including the author himself. He also found it to be impossible to link his own software with the emulator, which the documentation stated was possible.

During the development of CrazyOS, the author had the choice to either use VMware Workstation Player or QEMU. Both of these software are used by professionals for testing operating systems, with the former being a virtualization software and the latter being an emulator. A virtualization software allows the user to run a guest OS meant for the same architecture on which the host OS is running. Resources of the system are managed by the virtualization software, in

this case VMware Workstation Player. In the case of an emulator, software fills in the role of hardware by emulating hardware devices. An emulator allows the user to run software which is meant for different architectures. QEMU uses kernel-based virtual machine which uses translation of guest machine's instructions to host machines at byte level, which allows it to run guest OS at near native speed [13].

VMware Workstation Player was found to be using too many resources when running Windows XP on the author's laptop. It was also found that it was not able to boot CrazyOS properly. As the author found QEMU to be able to run Windows XP and toy operating systems with ease on his machine, he chose it over VMware Workstation Player and has used it for testing CrazyOS. In particular, QEMU's emulation of an IBM PC using i386 (80386) microprocessor has been used for testing the operating system. This emulator is ran from the terminal using the command `qemu-system-i386`. Emulation of various peripherals and disk drives can be connected to it using by specifying them with typing appropriate commands after typing the name of machine to be emulated. For example, if the user has a binary file, `file.bin` which he wishes to use a floppy disk A for an i386 based IBM PC, he can do so by executing the following command:

```
qemu-system-i386 -fda file.bin
```

2.2.4 Makefile

GNU Make is widely used by developers across the globe for building object files and executables automatically without retyping commands for building each source file. The author has used a single makefile, `CrazyOS/8086/Makefile`, for building the image of the operating system.

2.2.5 Git

Git is the most popular source version control software. It used for maintaining many big open source projects such as the Linux kernel and VScode. A local Git repository can be linked to a remote repository on the internet. Most developers use Github as the platform for hosting their projects and collaboratively developing and maintaining them. The author has used Git v2.34.1 for maintaining this project. The project is hosted at `https://github.com/PraneetKapoor2619/CrazyOS`, with the latest short commit ID as of writing this report being 1a17fa0.

2.3 Structure of the Project Directory

CrazyOS started out as a project to teach its author about the 8086 microprocessor and the IBM PC. The author expects to write similar toy operating systems for computers using i386, x86-64, ARM and RISC-V processors. These ambitions are reflected in the commit history and by the structure of the project directory. We will begin by explaining the structure of the parent directory, `CrazyOS/`. The rest of this chapter and the report will be concerned with the contents of `CrazyOS/8086/` directory.

2.3.1 Parent Directory

Assuming the reader clones the project from Github into a directory named `CrazyOS`, the directory tree that will be obtained is shown in Figure 2.1. *depth = 0* indicates the number of files and subfolders that are being shown in the source tree.

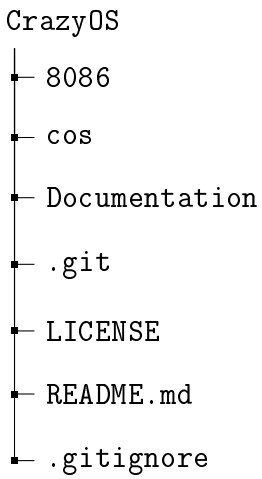


Figure 2.1: Parent directory tree (depth = 0)

The reader will observe that there is a folder named `8086`. It contains the source files for building 16-bit real mode version of CrazyOS which is the subject matter of this report. When in future the author develops CrazyOS for a computer using a processor other than 8086 (or x86-64 in real mode), the source file for that version of CrazyOS will be put in a folder bearing the name of the processor. For example, if the author releases CrazyOS for i386 based computer, its source files will be placed in a folder named `i386`.

Restructured text (`.rst`) files are used to write documentation in favour of markdown files.

These files are kept in `CrazyOS/Documentation/source/` folder. Documentation for different versions of CrazyOS will be kept in folders bearing the name of the processor for which the operating system is designed. Sphinx is the tool which is used for generating documentation using `.rst` files. It can generate documentation in HTML, PDF, LaTex, etc. formats. The configurations for the Python environment using Sphinx is stored in `CrazyOS/cos/` folder. The documentation so generated can be published online using readthedocs.org. This has been done by many project communities, for example, The Linux Kernel documentation and QEMU's documentation have been made and published using the aforementioned toolchain. It should be noted that the documentation for this project is still a work in progress.

Finally, we come to `LICENSE` and `README.md`. `LICENSE` file contains a copy of GNU/GPL Version 3 license under which the project is released. GNU/GPL is a *"free, copyleft license for software and other kinds of works."*. As per this license, the project is distributed as a free software which can be modified by others. `README.md` contains a brief introductory text which tells the reader about the motivation and scope of the project. If the user has enabled the option to view hidden files and folders, he will observe that there is folder named `.git` and a file named `.gitignore` present in the parent directory. The former stores the Git configuration files for this directory/repository. The latter stores the extensions which should not be tracked by Git.

2.3.2 Project Directory: ../8086

As has been mentioned before, the directory `../8086`* contains the source files for the project. The tree of this directory is given in Figure 2.2.

Except for `.vscode` and `README.md`, the files and folders in the source tree shown in Figure 2.2 form the *soul* of this project. They are discussed in following sections in the order in which they are executed when the operating system runs on the emulator. Before ending this section, we explain the purpose of `.vscode` and `README.md`. `.vscode` contains four files, namely `configurationCache.log`, `dryrun.log`, `settings.json`, and `targets.log`. These files setup the VScode environment (theme and syntax highlighting) which has been described in section 3.1.1. `README.md` is a project description file.

*`..` is used to refer to the parent directory, in this case `CrazyOS`. In fact, `CrazyOS` is itself a part of some directory and therefore it can be written as `../CrazyOS`.

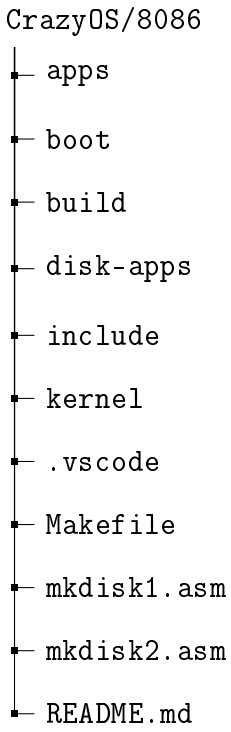


Figure 2.2: Directory tree for `CrazyOS/8086` (depth = 0)

2.4 Bootloader: ../boot

The bootloader was the first program which was developed for this project. All the files which are related to it are kept in `../boot` directory.

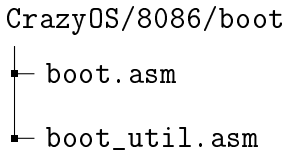


Figure 2.3: Directory tree for `CrazyOS/8086/boot` (depth = 0)

As can be seen from Figure 2.3, there are two files in `../boot`, namely, `boot.asm` and `boot_util.asm`. We will now explain both of these files.

2.4.1 boot.asm

`boot.asm` is the main bootloader source file which is responsible for loading the kernel header from the storage disk. The storage disk in this project is the bootable floppy disk image, `../build/CrazyOS.bin`. Besides loading the kernel header, the bootloader is also reponsible for clearing the screen, setting background and foreground colors, and finally setting the page

number and cursor position.

`boot.asm` begins with the following directives:

```
cpu 386
bits 16
align 16

org 0x7c00
```

`cpu 386` directive tells NASM that the source code in the current file is written for a computer having i386 processor. `bits 16` directive tells the assembler that the instructions used in the source code are 16-bit instructions and not 32-bit instructions. Aligning data and instructions such that they start at even-numbered addresses allows the processor to easily access them which speeds up the execution speed [2]. This requires aligning data and instructions by 16-bits, which is done using `align 16` directive. Finally, the directive `org 0x7c00` tells the assembler that the source file will be loaded at the address 0x7c00. Therefore, addresses of all instructions and data in the given source file should be calculated with respect to this base address.

Using `%include "boot/boot_util.asm"` we include the source code of `boot_util.asm` in `boot.asm` which allows us to use functions present in the included file. To jump around this portion of the code and run the main body of the bootloader, we use `jmp boot_main`.

In the main body of the bootloader, we initialize the stack using the following instructions:

```
boot_main:
        mov bp, 0x7ffe
        mov sp, bp
```

The base point `bp` points at an even-address as this allows the process to perform stack operations faster. As nothing is currently present in the stack, the stack pointer points at the same address as the base pointer.

We now call `clrscr`, `reset_curse`, and `set_pg` subroutines to clear the screen and set its colour, reset cursor position, and set the page number to 0, respectively. Colour of the screen is set by putting 0x1f in register `bh` before executing INT 10,07 interrupt. The higher nibble stores the hex-code for background colour and the lower nibble stores the hex-code for the foreground colour. Different 8-bit colours and their hex-codes are given in Table 2.1.

Table 2.1: Hex-codes for 8-bit colours

Hex-code	**Colour**
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	Grey
8	Dark grey
9	Bright blue
a	Bright green
b	Bright cyan
c	Bright red
d	Bright Magenta
e	Yellow
f	White

After these *beautification* procedures, the operating system is loaded from disk by `read_disk` subroutine. This subroutine uses INT 13,02 to read the disk. As of writing this report, the operating system has a size of 5.9 kiB which makes it necessary that more than one sector are loaded from disk. A total of 32 (0x20) sectors are read. The operating system is loaded at address 1000:0000. The disk operation is considered to be successful if INT 13,01 returns 0x00 in `al`.
To actual run the operating system, we have to perform an intersegment jump to 1000:0000 where the kernel header has been loaded. This is done by pushing 0x1000 and 0x0000 to the stack and executing a far return instruction, `retf`.
As has been mentioned in section 2.5.3, the first stage of the bootloader should be fit into the very first sector of the disk. The disk should also be bootable else the bootloader will never be loaded by the BIOS into the memory. To mark the disk as bootable, the magic number 0xaa55 is placed at the end of the bootsector. We do this by using the following directive:

```
times 510 - ($ - $$) db 0
dw 0xaa55                   ; magic number
```

A padding of zero is performed by `times` directive which repeats a null byte till the size of the bootloader is equal to 510 bytes. The magic number is then placed. This is shown in Figure 2.4 which shows the hexdump of the bootloader. As x86 family uses little-endian byte ordering, therefore, 0x55 is placed first and then 0xaa is placed. There occupy the 510th and 511th bytes of the bootsector, respectively.

```
00000120: be81 7de8 dcfe 8b16 c67d e8f0 fee8 3aff  ..}......}....:.
00000130: be97 7de8 ccfe 8b16 c87d e8e0 fee8 2aff  ..}......}....*.
00000140: c342 4f4f 5449 4e47 202e 2e2e 0044 4953  .BOOTING ....DIS
00000150: 4b20 4552 524f 523a 2000 4449 534b 2052  K ERROR: .DISK R
00000160: 4541 4420 5355 4343 4553 5346 554c 2121  EAD SUCCESSFUL!!
00000170: 204b 4552 4e45 4c20 4c4f 4144 4544 2121   KERNEL LOADED!!
00000180: 004b 4552 4e45 4c20 4241 5345 2041 4444  .KERNEL BASE ADD
00000190: 5245 5353 3a20 004b 4552 4e45 4c20 4f46  RESS: .KERNEL OF
000001a0: 4653 4554 2041 4444 5245 5353 3a20 0046  FSET ADDRESS: .F
000001b0: 4952 5354 2057 4f52 4420 4f46 204b 4552  IRST WORD OF KER
000001c0: 4e45 4c3a 2000 0010 0000 0000 0000 0000  NEL: ..........
000001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
```

Figure 2.4: Hexdump of the bootloader showing the placement of the magic number 0xaa55

2.4.2 boot_util.asm

It is important to print a few messages on the screen when the bootloader is loading the operating system. These messages especially useful in the event disk read operation has failed and the

kernel has not been loaded properly. To print such messages, we make use of INT 10,0E which allows us to print characters to the screen in teletype mode. The subroutines used by the boot-loader to which to print strings, newlines and hexadecimal numbers on the screen are present in `boot_util.asm` file. These were some of the first subroutines which were developed as part of this project. These are:

1. `print`

   Input : `si` = pointer to the first byte of a string

   Output : Nothing*

   Description : Prints a null (0x00) terminated string pointed at by `si`.

2. `printhex`

   Input : `dx` = 16-bit hexadecimal number

   Output : Nothing

   Description : Prints the hexadecimal number stored in `dx` in `0x____` format. Uses right-shift operation `shr` and logical AND, `and`, to decompose the hexadecimal number stored in `dx` into individual nibbles. A string representing the hexadecimal number is then made by adding each individual nibble to the ASCII value of either '0' or 'A' depending upon the whether it is less than 0x0a or not, respectively. The string is then printed using `print` subroutine.

3. `printnl`

   Input : Nothing

   Output : Nothing

   Description : Prints a newline, i.e., positions cursor at the beginning of the next line. This is achieved by printing line-feed character (0x0a) and carriage return (0x0d) using INT 10,0E. This is equivalent to `printf("\n");` used in C programs for positioning the cursor at the beginning of the next line.

*In context of a subroutine, the term "output" means the value that is being returned by it.

2.5 Kernel Header: ../kernel/kernel.asm

If one were to draw a memory map for CrazyOS (Figure 2.5), they will find that the subroutine which is placed at the lowest memory address is the `kernel_entry` subroutine. The operating system can be said to begin with this subroutine. If `kernel.asm` forms the head of the

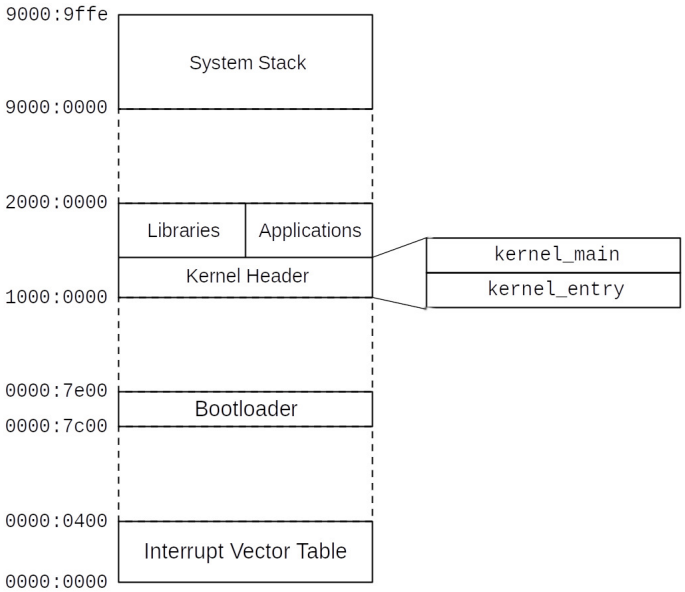


Figure 2.5: Memory map of CrazyOS

operating system, then `kerne_entry` subroutine would be the scalp and `kernel_main` would be the cranium of the operating system. The bootloader performs an intersegment call to *kernel_entry* subroutine which has been loaded, along with the rest of the operating system, in block 1. `kernel_entry` subroutine is responsible for initializing `cs`, `ds`, `es`, and `ss`.

```
kernel_entry:
        mov ax, cs
        mov ds, ax
        mov es, ax
        mov ax, 0x9000
        mov ss, ax
        mov bp, 0x9ffe
        mov sp, bp
```

After an intersegment call has been performed by the bootloader, `cs` has been loaded with

0x1000. `ds` and `es` should also use the same base address as the data for the operating system has also been loaded in block 1 itself. As we cannot assign a value to segment registers directly, therefore, we store the value in `cs` in `ax`. Then the value stored in `ax` is assigned to `ds` and `es`. Furthermore, we must make sure that our stack is not in the same segment as the code and data segments, otherwise problems will occur if `sp` and `bp` start pointing at data and instruction words. That is why the stack for the operating system is formed in block 9 which is far away from block 1. `ss` is initialized with a value of 0x9000, and `bp` and `sp` with a value of 0x9ffe.

`kernel_main` subroutine begins just after `kernel_entry` subroutine. It is analogous to the `while(1) {` loop which is an infinite loop which runs a sequence of tasks again and again. `kernel_main` clears the screen, prints a test message, date and time, and then call the `main` subroutine present in `../apps/shell.asm`.

2.6 Library Functions: ../include

Every operating system comes with a bunch of subroutines which are used by applications to interact with the hardware. Subroutines using which other subroutines can be built are called system calls. System calls can be issued by either calling the subroutine, or by issuing a software interrupt in the same manner a BIOS interrupt call is used to invoke the facilities provided by the BIOS. System calls and other such fundamental functions are kept in a separate directory, usually protected from user's tampering. These functions are often called library functions. In CrazyOS, these library functions are placed in `../8086/include` folder. Source tree of this directory is shown in Figure 2.6. As these functions use BIOS calls to access hardware facilities, the reader can think of BIOS interrupts as system calls. The subroutines present in the library are analogous to functions present in the standard C library such as `printf`, `scanf`, `atoi`, etc., which a programmer often uses in their application to get the job done.

Some of the library functions present in this directory were inspired by the standard C library functions. The rest were created as and when the need for accessing a specific piece of hardware was required. We will describe each these functions beginning with the ones which took inspiration from their standard C library counterparts. Before explaining these library functions, we will explain the organization of library files, and the `include` guard.

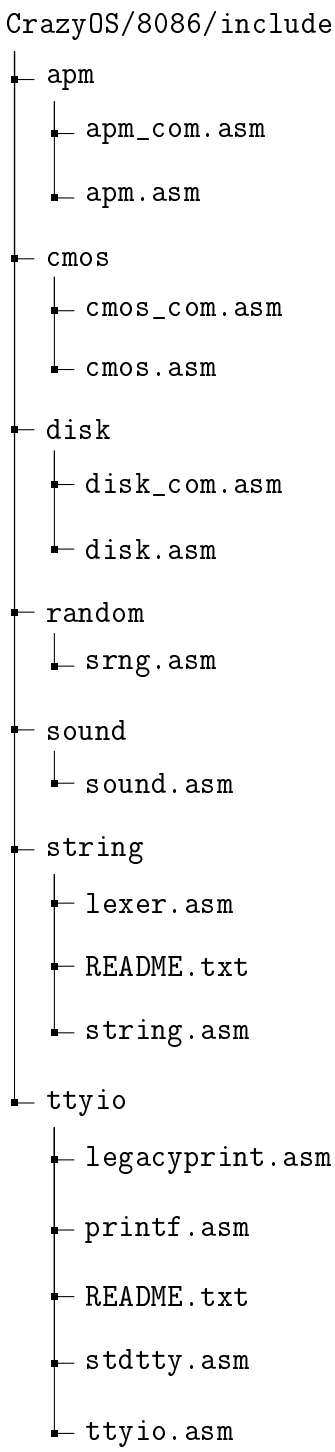


Figure 2.6: Directory tree for `CrazyOS/8086/include` (depth = 1)

2.6.1 Organization of Library Functions

The first few subroutines in the library used to deal with setting up a terminal environment and handling strings. Later on, the author started dealing with device specific codes and also wrote subroutines which were executed when a command was entered in the shell. This prompted the author to organize the `../include` folder. Each subfolder in `../include` deals with a particular component of the computer or some process. The name of the subfolder is same as the name of the device it deals with. Baring a few exceptions, subfolders present in the library contain two types of files:

1. *Fundamental files*: As the name speaks for itself, these files contain core subroutines which deal with devices making up the system. These subroutines might be present in different files, but are always clubbed together into a single file having the same name as that of the folder it is present in. For example, the subroutines dealing with setting up a terminal environment are present in different source files in `../include/ttyio` subdirectory, but are clubbed together into a single file named `ttyio.asm`.

2. *Command processing files*: These files contain subroutines which are executed in response to commands that has been entered in the shell. These subroutines often make use of subroutines present in fundamental files present in the same or other directories. These files also bear the same name as the folder they are present in, but have the suffix `_com` added to them. For example, the fundamental procedures dealing with disk operations are kept in `../include/disk/disk.asm` and those procedures which are executed in response to entering of disk read/write commands in the shell are kept in `../include/disk/disk_com.asm` file.

2.6.2 %include Guard

In C language, the C preprocessor processes statements which begin with #. These statements are called preprocessor directives. Examples are `# include` and `# define` directives. The `# include` directive is followed by the name of a file whose functions are being used in the current file. The preprocessor expands the contents of the file in place of the directive. As the size of a project grows, so does the number of source files. Including the same file more than once can cause problems during the linking stage as the linker will get confused by the multiple occurrences of the same label in different object files. To prevent such problems, The file to

be included has its contents placed in between `# ifndef`, `# define`, and `# endif` directives. These are called `include` guard, as they ensure that if a macro has already been declared, then there is no need to include the contents of the header file again in the final binary. For example, the first two statements of `stdio.h` file are

```
#ifndef _STDIO_H
#define _STDIO_H                1
```

and its last statement is

```
#endif                          /* <stdio.h> included.  */
```

The contents of `stdio.h` are included between these statements. This way a programmer can include the files they think are necessary for a project and not worry about repeated inclusion of files.

NASM provides us with `%ifndef`, `%define`, and `%endif` directives which work exactly like their C counterparts. Library subroutines often use this guard to ensure that no assembly source file is included more than once in the final binary of the project. As an example, the first few lines of `../include/ttyio/ttyio.asm` are

```
bits 16
align 16
%ifndef TTYIO
%define TTYIO
%include "include/ttyio/legacyprint.asm"
%include "include/ttyio/printf.asm"
%include "include/ttyio/stdtty.asm"
```

and its closing line is

```
%endif
```

2.6.3 Standard I/O subroutines: ../include/ttyio

A programmer just needs to include `ttyio.asm` file in his source file to get access to all the functions which are available for printing on the screen and for taking input from the keyboard. The file `ttyio.asm` contains a single subroutine, `fstringdata`, with the rest coming from inclusion of `legacyprint.asm`, `printf.asm`, and `stdtty.asm`. `fstringdata` can be described as follows:

`fstringdata`

Input : `si` = pointer to a null terminated string,
`ax` = data to be entered,
`cx` = position at which data is to be entered

Output : Nothing

Description : Used to enter data next to a null-terminated formatted string.

`legacyprint.asm` was the first library file which was written for this project. Two functions, namely, `print`, and `printnl`, have been directly copied from `../boot/boot_util.asm`. `printhex` subroutine present in this file bears the same name as its counterpart which was used in the boot-loader to print hexadecimal numbers. However, this version of `printhex` uses right rotation instruction, `ror`, to decompose the 16-bit hexadecimal number stored in `ax` into nibbles. Other functions which have been included in this file are:

1. `print8bitpackedBCD`

   Input : `ah` = 0x00,
   `al` = 8-bit packed BCD

   Output : Nothing

   Description : Prints an 8-bit packed BCD stored in `al`.

2. `printdec`

   Input : `ax` = 16-bit number (0 - 65535)

   Output : Nothing

   Description : Prints a 16-bit number stored in `ax` in decimal format. A blank space is printed if the number is equal to 0.

`stdtty.asm` defines several subroutines which enable the user to print characters on the screen as well as take input from the keyboard. The functions defined in this source file are:

1. `clear`

   Input : Nothing

   Output : Nothing

Description : Similar to the working of `clrscr` subroutine in `../boot/boot.asm`, this subroutine clears the screen, sets page-number to 0, resets background and foreground colour to their default values, and positions the cursor at row 0 and column 0. Executed when `clear` command is entered in the shell.

2. `findcursorposition`

   Input : Nothing

   Output : `dh` = row at which the cursor is positioned (0 - 24),
   `dl` = column at which the cursor is positioned (0 - 79)

   Description : Used to find the position of the cursor.

3. `scrollup`

   Input : Nothing

   Output : Nothing

   Description : Scrolls the screen up by a single line only

4. `conditional_scrollup`

   Input : Nothing

   Output : Nothing

   Description : Finds the position of the cursor and scrolls the screen up by a single line if the cursor is found to be positioned at the 24th row.

5. `putchar`

   Input : `al` = code of the character to be printed from code page 437

   Output : Nothing

   Description : Prints a single character onto the screen. Calls `conditional_scrollup` and `printnl` to scroll the screen up by one line in case the cursor is at 24th row.

6. `getchar`

Input : Nothing

Output : `ah` = keyscan code of the key combination that has been pressed, `al` = code of the character corresponding to the key combination that has been pressed.

Description : Reads a key from the keyboard and returns the corresponding key scan codes and character code in `ah` and `al`, respectively.

7. `getline`

Input : Nothing

Output : `si` = pointer to the 80-byte wide buffer, `GETLINE_BUFFER`

Description : Allows the user to input a single line, having a maximum length of 80 characters, through the keyboard. Printable ASCII characters are printed on the screen. Rest of the characters perform control sequence such as deleting a character, positioning the cursor, etc. Not all cursor control sequences are supported. `si` is returned as a pointer to the `GETLINE_BUFFER`.

8. `flush`

Input : Nothing

Output : Nothing

Description : Flushes the `GETLINE_BUFFER` by setting all 80 bytes it is made up of equal to 0x00.

Finally, we come to `printf.asm` which has the source code for a bare-minimum implementation of `printf` found in C.

`printf`

Input : `si` = pointer to a null-terminated string followed by optional data to be printed.

Output : Nothing

Description : A bare-minimum implementation of `printf`. Data which is to be printed should have word size and should be entered after terminating null word

of the string using `fstringdata` subroutine. Supported format specifiers are:

`%c`: Single character

`%d`: 16-bit number in decimal format

`%x`: 16-bit number in hexadecimal format

Supported escape sequences are:

`\n`: Newline

`\t`: Tab

2.6.4 String Handling: ../include/string

String handling is an important part of any language. The programmer who is writing applications for CrazyOS has to include `../include/string/string.asm` in his application program to allow him to use various subroutines to handle strings. These subroutines have been inspired from their counterparts found in `string.h` in C. These subroutines have been described below:

1. `strlen`

   Input : `si` = pointer to a null-terminated string

   Output : `ax` = length of the string pointed at by `si`

   Description : Used to compute the length of the string.

2. `strcmp`

   Input : `si` = pointer to the first null-terminated string,
   `di` = pointer to the second null-terminated string

   Output : `ax` = 0x00 if the the strings are equal, or
   `ax` = 0x01 if the strings are not equal

   Description : Compares two strings which are being pointed at by `si` and `di`. It first compares their length using `strlen` subroutine. If the lengths are found to be equal, it then compares them character by character. If both the strings are found to be equal to each other, then `ax` is returned with 0x00, else `ax` is set to 0x01

3. `strcpy`

   Input : `si` = pointer to the null-terminated source string,
   `di` = pointer to the destination string

   Output : Nothing

   Description : Copies the source string pointed at by `si` to the destination string pointed at by `di`. `rep movsb` instruction is used to perform this copying operation.

4. `atoi`

   Input : `si` = pointer to the string representation of a 16-bit positive number.

   Output : `ax` = number which was stored in the string pointed at by `si`

   Description : Converts the string representation of a number to a 16-bit number, stores it in `ax`, and returns to the caller.

5. `atoh`

   Input : `si` = pointer to the string representation of a hexadecimal number

   Output : `ax` = hexadecimal number which was stored in string format

   Description : Converts the string representation of a hexadecimal number, stores it in `ax`, and returns to the caller. Although the prefix "0x" is used when printing hexadecimal numbers, it should be made sure that the string being passed to atoh does not have this prefix.

`..include/string` directory has another assembly source file besides `string.asm`. It is `lexer.asm` which is responsible for breaking a string of characters into separate "words" using some separator. The function used for performing this primitive lexical-analysis is `cmd_lexer`. Its description is given below:

`cmd_lexer`

Input : `al` = ASCII code of the character to be used as a separator,
`si` = pointer to the 80-byte wide string

Output : Nothing

Description : *Tokenizes* the string that is being pointed at by `si` using the character stored in `al` as a separator. It generates its own personal copy of the string which is stored in local label `.COPY`. This subroutine is capable of generating a maximum of 5 tokens: one command token and four argument tokens. Command token is stored in global label `com` and argument tokens are stored in `arg1`, `arg2`, `arg3`, and `4` which are also global labels.

2.6.5 CMOS and RTC: ..include/cmos

Real time clock, or RTC for short, is responsible for keeping track of date and time even when the computer has been turned off. This is achieved by independently powering the RTC using a CR2032 coin cell. As the power consumption of RTC is very small, one coin cell lasts for many years. The RTC chip also has very low power static CMOS memory. Although the term CMOS specifically refers to a type of semiconductor technology, when used in context with personal computers, the term refers to the static memory on the RTC chip. This memory has several registers which store setup information for the BIOS even when the computer is turned off. The first 14 CMOS registers store the time and date and control the operation of the RTC. CMOS registers can be accessed using `in` and `out` instructions. CMOS's input is mapped to port 0x70 and its output is mapped to 0x71. Before writing anything to these ports the interrupts must be disabled using `cli`. They can later be enabled using `sti`. To read the value of a CMOS register, we store the register number in `al` and then execute the following instructions:

```
cli
out 0x70, al
nop
in al, 0x71
sti
```

The value of the register is returned in `al`. To write a value to a CMOS register, we first select the register by passing its number to port 0x70. Then write the value to be written at port 0x71. The code for the same is shown below:

```
cli
out 0x70, al
nop
```

```
mov al, bl
out 0x71, al
sti
```

Simple subroutines to read and write CMOS registers are kept in `cmos.asm` file. These subroutines are:

1. `read_cmos`

   Input : `al` = index of the register to be accessed

   Output : `al` = value of the register that was accessed

   Description : Used to read the value of a register. Interrupts are disabled before executing port I/O instructions and are enabled before returning to the caller. A `nop` instruction is executed between two consecutive port operations. This gives the system enough time to perform port operation correctly.

2. `write_cmos`

   Input : `al` = index of the register to be accessed,
   `bl` = value to be written to the register

   Output : Nothing

   Description : Writes a value passed in `bl` to the CMOS register with index equal to the value stored in `al`.

`cmos_com.asm` contains subroutines which are executed when `date` or `time` commands are entered in the shell. These subroutines are:

1. `time`

   Input : Nothing

   Output : Nothing

   Description : Global variables `HRS`, `MIN`, and `SEC` are updated with the value of hour, minutes, seconds at the time of execution of the function, respectively.

2. `showtime`

   Input : Nothing

Output : Nothing

Description : Prints time in `HH:MM:SS` format. Hour is printed in 24-hour format. Executed when `time` command is entered in the shell.

3. `date`

Input : Nothing

Output : Nothing

Description : Global variables `WEEKDAY`, `DAY_OF_THE_MONTH`, `MONTH`, and `YEAR` are updated with the value of current index of weekday (Sunday = 1, Monday = 2, etc.), day of the month, month number, and last two digits of the year at the time of execution of the function, respectively.

4. `showdate`

Input : Nothing

Output : Nothing

Description : Displays current date in `Weekday, DD-MM-YY` format. Executed when `date` command is entered in the shell.

2.6.6 Random Number Generator: ../include/random

Random number generators often find application in games, cryptography, and computer simulation. The author has written a simple pseudo-random number generator, `srng`. It is described as follow:

`srng`

Input : Nothing

Output : `ax` = pseudo-random number

Description : Produces a pseudo-random number by taking the value of seconds elapsed as a seed value. This value is obtained from register 0x00 of the CMOS. This seed value is multiplied with 8999d, which is a prime number. 9923d, which is another prime number, is added to the product's low byte to produce a pseudo-random number which is returned in `ax`.

2.6.7 Sound Generator: ../include/sound

Before discussing the process of sound generation in an IBM PC, we must briefly discuss the 8254 programmable interval timer (PIT) chip. This chip was primarily designed to solve timing control problems in microcomputers. Its block diagram is shown in Figure 2.7.

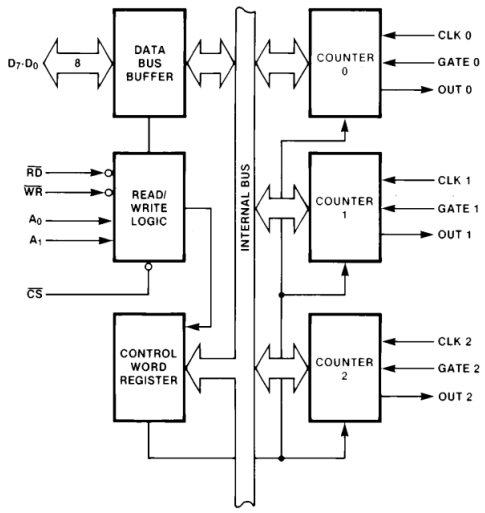


Figure 2.7: Block diagram of the 8254 PIT chip [4]

It consists of 3 counters/timers, each capable of working independently of each other. Numbering of timers start from 0. Therefore, we have timer 0, timer 1, and timer 2. The mode in which each timer operates is decided by the value which has been passed to the control word register. The format of the word which is passed on to the control word register is shown in Figure 2.8. On an IBM PC, control word register is mapped to port 0x43. Once the control word has been transmitted, the counter of each timer needs to be initialized with some value. Count down starts from this value. The counters of timers are also mapped to I/O ports starting from port 0x40. So, timer 0's counter is mapped to port 0x40, timer 1's counter is mapped to port 0x41, and timer 2's counter is mapped to port 0x42. Each timer/channel two input pins, namely `CLK` and `GATE`. The former receives clock signal from an oscillator running at approximately 1.19 MHz. The later is used to enable an output signal from the output pin, `OUT` [4].

We are interested in operating the piezoelectric speaker which is soldered onto the motherboard of PCs. This piezoelectric speaker is driven by the signal generated by timer 2, and is connected to its `OUT` line using an AND gate as shown in Figure 2.9. The second pin of this AND gate is mapped to bit 1 of port 0x61. `GATE 2` pin is mapped to bit 0 of port 0x61.

The algorithm for producing a sound from the speaker by generating a square wave of par-

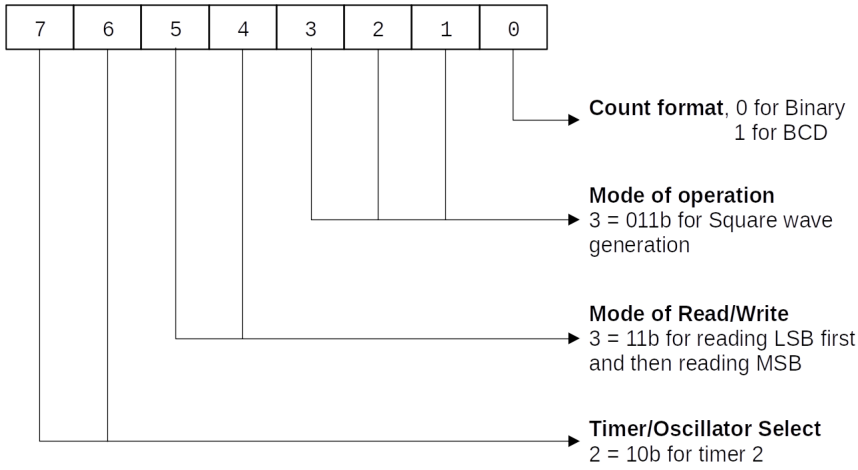


Figure 2.8: Control word format for the PIT chip

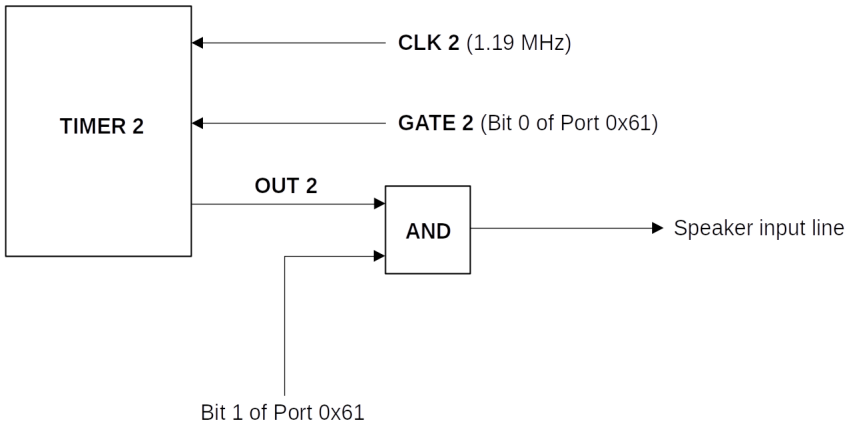


Figure 2.9: Connection of motherboard speaker with timer 2 of the PIT chip

ticular frequency from timer 2 is as follows:

Step 1: Select the frequency, f, of the sound you wish to generate.

Step 2: Compute the value to be stored in the counter using the following formula:

$$\text{Value to be stored in the the counter, } N = \frac{1.19MHz}{f}$$

Step 3: Round of N to the nearest integer.

Step 4: Output a byte to port 0x43 specifying that timer 2 is to be used as a square wave generator, with the value of latch count being transmitted in binary, LSB being transmitted first, and then MSB.

Step 5: Output the LSB of the latch count value to port 0x42.

Step 6: Output the MSB of the latch count value to port 0x42.

Step 7: Read the value at port 0x61, I.

Step 8: Set bit 0 and 1 of value I. i.e.,

$$I = 00000011b \mid I$$

Step 9: Output I to port 0x61.

Step 10: Delay while the note plays.

Step 11: Read the value at port 0x61, I.

Step 12: Clear bit 0 and 1 of value I, i.e.,

$$I = 11111100 \ \& \ I$$

Step 13: Stop the timer by outputting I to port 0x61.

This algorithm is used by the subroutines in `sound.asm` . These subroutines are:

1. `sound_tiny`

   Input : `ax` = latch count value,
   `bx` = down-counter's initial value

   Output : Nothing

   Description : Generates a sound wave equal to `ax`$\times 1.19MHz$. Delay is provided by counting down from the value present in `bx`.

2. `sound_mega`

   Input : `ax` = latch count value,
   `bx` = `sound_tiny`'s down-counter's initial value,
   `dx` = `sound_mega`'s down-counter's initial value

   Output : Nothing

   Description : Generates a sound wave equal to `ax`$\times 1.19MHz$ for a longer duration by using two nested down-counters.

2.6.8 Disk Read/Write Operations: ..include/disk

Disk read-write operations are necessary because they allow the user to store their data in a storage media which retains information even after the computer has been turned off. CrazyOS is capable of performing disk operations on floppy disks using INT 13 BIOS call. The core subroutines for checking a disks status and reading from, and writing to sectors on the disk are in `disk.asm` file. These subroutines are now described.

1. `disk_status`

   Input : Nothing

   Output : Nothing

   Description : Reads the status of last disk operation using INT 13,01 BIOS call. Disk status is returned as an 8-bit number in `al`. Each number corresponds to a particular event. For example, if `al` = 0x00, then the previous disk operation was successful. If `al` = 0x01, then an invalid parameter was passed in `ah` or other registers, and so on. These messages are printed by a call to `print_disk_status` after executing the BIOS call.

2. `disk_read`

   Input : `al` = number of sectors to be read,
   `ch` = cylinder number,
   `cl` = sector number,
   `dh` = head number,
   `dl` = drive number,
   `es:bx` = segment:offset

   Output : Nothing

   Description : Reads sectors from a disk using INT 13,02 with the parameters that have been passed to it. Sectors are read and their contents are stored at address `es:bx`. In case the carry bit is set after disk read, `disk_status` is called to print the error message.

3. `disk_write`

Input : `al` = number of sectors to be written,
`ch` = cylinder number,
`cl` = sector number,
`dh` = head number,
`dl` = drive number,
`es:bx` = segment:offset

Output : Nothing

Description : Writes the content at address `es:bx` to the disk using INT 13,03 with the parameters that have been passed to it. In case the carry bit is set after disk write, `disk_status` is called to print the error message.

The user does not need to hardcode the operating system to load his disk applications: he can simply use `disk read` command to load the application from a disk to a segment of his choice. Similarly, the user can write a segment to disk using `disk write` command in the shell. The subroutines which are executed when disk related commands are entered in the shell are present in `disk_com.asm`. The primary subroutine which is executed when upon entering `disk` command in the shell is `disk_com`. It is described below:

`disk_com`

Input : `si` = pointer to the first argument of `disk` command

Output : Nothing

Description : Executed when `disk` command is entered in the shell. Valid disk commands are `disk read`, `disk write`, and `disk -h`. Commands other than these generate an error message. Parameters for disk read/write operations are filled by `disk_rw_parameter_fill` subroutine.

2.6.9 APM Support: ../include/apm

To control the power source of computers, vendors release various software packages which monitor and control the power source and power requirements of different computer peripherals. There are two main technologies used for putting power management in the hands of the operating system: Advanced Power Management (APM), and Advanced Configuration and Power Interface (ACPI). APM is a relatively older technology. ACPI is being used in favour of

APM. However, as APM is much simpler than ACPI, the author decided to use APM for managing system power in this project.

APM was developed by Intel and Microsoft back in 1992. It enables an operating system running on an IBM compatible PC to use INT 15,53 BIOS call to manage power [5]. Before accessing the services of APM, it should be made sure that the operating system has been switched to real mode and that APM is enabled for all the devices. The algorithm to enable APM is presented below:

Step 1: Check whether APM is present or not. If APM is not present then terminated this algorithm.

Step 2: Disconnect every device from APM.

Step 3: Connect to the real mode interface for APM.

Step 4: Enable APM for all the devices.

`APM_REAL_MODE_ENABLE` present in `apm.asm` is an implementation of this algorithm. After this algorithm has been executed, services of the APM can be used using INT 15,53 BIOS call. The subroutines present in `apm.asm` are now described:

1. `APM_SERVICE_ROUTINE`

   Input : Registers `al`, `bx`, `cx`, and `dx` must be properly initialized for a particular APM service [5].

   Output : Values are returned in general purpose registers depending upon the service that has been used [5].

   Description : Used for accessing services of APM using INT 15,53 BIOS call.

2. `APM_REAL_MODE_ENABLE`

   Input : Nothing

   Output : Nothing

   Description : Enables APM for all devices connecting to its real mode interface.

3. `APM_POWER_OFF_ROUTINE`

   Input : Nothing

Output : Nothing

Description : Powers off the system provided APM is present.

4. `APM_POWER_LEVEL_ROUTINE`

Input : Nothing

Output : Nothing

Description : Prints the power level of the system. At present not function on qemu-system-i386.

`apm_com.asm` contains a single subroutine, `power_com` which is executed when `power` command is entered in the shell. Its description is given below:

`power_com`

Input : `si` = pointer to the first argument of `power` command

Output : Nothing

Description : Processes the arguments passed with `power` command. Supported arguments/options are:

`off`: Used for turning the computer off, provided it supports APM

`level`: Used for checking the charge left in the battery. Currently prints "Power level unknown".

`-h`: Used for displaying valid options to be passed with `power` command.

2.7 Applications

Early in the development of this project, the author read that an assembler, a line editor, and a shell formed the first three applications for the UNIX operating system. This inspired him to write a shell and a line editor for CrazyOS. The idea of developing an assembler was dropped because of time constraints.

The applications for CrazyOS can be divided into two categories: system applications and user applications. Those applications which are ran from the shell and have been included on the

same disk as the operating system itself are called system applications. Those applications which the programmer writes and loads on the second floppy disk are called user applications or disk applications. System application files are in the `../apps` directory (Figure 2.10), and disk applications are in the (../disk-apps) directory (Figure 2.2). As of writing of this report, there are two system applications, namely, the shell and the line editor. There is only a single user application, the sound and light program, `soundnlight.asm`. In this section, we will first discuss the aforementioned system applications, starting with the shell. Then we will briefly discuss `soundnlight.asm`.

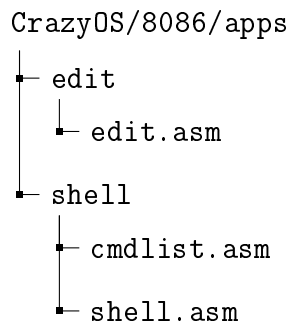


Figure 2.10: Directory tree for `CrazyOS/8086/apps` (depth = 1)

2.7.1 Shell: ../apps/shell

The shell is considered as the *mother of all programs*. It is one of the first programs which runs in most operating systems. It provides the user an environment where he can issue commands and run other programs [40]. The environment provided to the user in which he can write using a keyboard and see the characters on the screen is called a *terminal emulator* as it emulates the old teletype-writers such as ASR Model 33. In CrazyOS, the terminal emulator is implemented by the use of subroutines present in `../include/ttyio` directory, in particular by `getline` and `printf` subroutines. The shell is implemented by the source code present in `../apps/shell` directory. This directory contains two source files: `shell.asm` and `cmdlist.asm`. There is only a single subroutine present in `shell.asm`, `main`. This subroutine firstly prints the prompt, `>`, using `printf`. Then it calls `getline` subroutine which takes input from the keyboard until the enter key gets pressed. Then using a blank space as the token separator, `lexer` splits the line into command word and arguments which are then executed.

The commands which are supported by the shell are present in `cmdlist.asm`. `com` string is compared with the strings present in this file one by one until a match is found. If no match has

been found then an error message is printed on the screen. The commands which are supported at the time of writing of this report are:

1. `clear`

   Description : Clears the screen and displays prompt on the first line.

2. `date`

   Description : Displays current date in `WEEKDAY, DD-MM-YY` format.

3. `disk`

   Description : Used for performing disk operations using INT 13 BIOS call.

   Options :

   `-h` Display available options.

   `read` Format: `read D:C:H:S SEG:OFF N`.
   Read `N` sectors from the disk no. `D` starting with cylinder `C`, head `H`, and sector `S`. Contents are stored at `SEG:OFF`.

   `status` Display status of last disk operation.

   `write` Format: `write D:C:H:S SEG:OFF N`
   Write data at `SEG:OFF` to `N` sectors of disk `D` starting with cylinder `C`, head `H`, and sector `S`.

   Note `D` = drive no. (decimal),
   `C` = Cylinder no. (decimal),
   `H` = Head no. (decimal),
   `S` = Sector no. (decimal),
   `SEG` = 16-bit segment base address (hexadecimal),
   `OFF` = 16-bit offset from segment base address (hexadecimal),
   `N` = No. of sectors to be read (decimal)

4. `edit`

   Description : Runs the line editor. Used for editing the contents of the buffer which has been passed as an argument.

Options :

`SEG:OFF` Address of the buffer in whose contents will be edited. `SEG` is the 16-bit segment base address. `OFF` is the 16-bit offset. Both are hexadecimal numbers.

5. `power`

   Description : Uses APM to perform power management.

   Options :

   `-h` Displays options available with `power` command.

   `level` Displays current power level.

   `off` Turns of the computer/emulator.

6. `run`

   Description : Performs a far call to the memory address passed in the command.

   Options :

   `SEG:OFF` `SEG` = Segment base address (hexadecimal),
   `OFF` = Offset (hexadecimal)

7. `srng`

   Description : Displays a 16-bit positive random number.

8. `time`

   Description : Displays current time in `hh:mm:ss` (24-hour) format.

2.7.2 Editor: ../apps/edit

`edit` is a simple line editor which is provided with CrazyOS. This line editor has been inspired from the infamously terse `ed` line editor which is the standard editor on UNIX and UNIX like operating systems.

The line editor is started by typing `ed SEG:OFF` in the shell. Here, `SEG` is the 16-bit segment base address and `OFF` is the 16-bit offset from the base address at which the contents of file will be stored. Due to the use of segments, the size of the file cannot exceed 64 kiB or about 819

80-character wide lines. Also, each file is terminated with a magic number, 0xaa. This hex-number maps to *á* in code page 437. As in this simple project we are not using accented letters, therefore, we do not have to worry about the loss of characters which might happen to exist in the file buffer after 0xaa.

Once the editor starts, the user is told about the number of characters that have been found in the file buffer. Like `ed`, `edit` allows the user to enter a single line at a time. The line so entered should not exceed 80 character limit. If the first character of the line is an asterisk, than the subsequent characters are considered as editor commands and are not stored in the file buffer. Valid editor commands are:

1. `edit`

   Description : Allows the user to edit a previous line.

   Format : `*edit N`, where `N` is a valid positive integer.

2. `exit`

   Description : Displays the number of sectors that will be required to fit the contents of the file and then exits from the editor.

   Format : `*exit`

3. `list`

   Description : Lists the contents of the file.

   Format : `*list`

If an invalid command is passed to editor, then all the characters except for the asterisk will be saved in the file buffer.

The file can be saved permanently by using `disk write` command to write the buffer to the disk.

2.8 Disk Applications: ../disk-apps

As we have discussed before, applications which come on a separate disk are called disk applications in this project. Source code for disk applications are saved at `../disk-apps` directory. At present, the whole project has only a single disk application, namely `soundnlight.asm`. This assembly file contains code which reads a single byte from the kernel (present in block 1), and

1. displays a pixel having a color corresponding to the value of the byte that has been read using INT 10,0C BIOS call.

2. generates a note corresponding the value of the byte that has just been read using `sound_tiny` subroutine present in `../include/sound/sound.asm`.

Then the pointer `si` is incremented and the next byte of the kernel is read. The screen is assumed to have a size of 640×480 pixels. Other available screen resolutions are 640×350 and 320×200 pixels.

2.9 Building Binaries and Disk Images

The phrase "building a project" in computer science refers to building an application or producing something which will be used in the for getting the task done. The final result is ran and tested. Many times the final result is not what was required which might require that certain files be removed. Typing all of these commands in the terminal to build and run the project and clean redundant files will be tedious. To automate this process we have used GNU Make automation software which allows us to do all of the aforementioned tasks with just a few simple commands. In this project, we want disk images of two 3.5" floppy disk having a size of 1.44 MiB. The first floppy disk will be having the binary of the bootloader and CrazyOS. The second floppy disk will be having the binary of our disk applications and will be used as a permanent storage media for storing files. Therefore, during clean up procedure, every binary file except for the second disk will be removed.

To build the binaries of the bootloader and the operating system, we type go to the project directory in the terminal and then type

```
make build
```

This command builds the binaries of the bootloader and the operating system and concatenates them into 1.44 MiB file. Binaries of disk applications are also created and concatenated into another 1.44 MiB binary file. These files are produced using `../mkdisk1.asm` and `../mkdisk2.asm`, respectively. Both of these files use `times` directive we discussed in section 3.3.1. The first disk which contains the bootloader and the operating system has the name `CrazyOS.bin` and the second disk having the disk applications has the name `disk2.bin`. A file named `CrazyOS.img` is also created. It has the same contents as `CrazyOS.bin` and is created so that it can be used with emulators other than QEMU (such as PCem) which only support `.img`

files.

The operating system is ran using `make run` command. In case the programmer needs to rebuild fresh disk image, he can type `make clean` in the terminal to delete all binary images in the `../8086/build` folder except for `asm.bin`, which is the binary image of `soundnlight.asm`, and `disk2.bin`, which might be containing some files which the programmer might have written using CrazyOS.

CHAPTER 3: RESULTS AND DISCUSSION

3.1 Introduction

We have discussed the implementation of CrazyOS. However, discussing the implementations is not enough for the reader needs to get their hands dirty to understand this project. For this reason, in this chapter we will present the results of this project. These results are obtained by running the operating system in QEMU's emulation of a system based on i386 processor. We have discussed how various commands are used in the shell and present to the reader the only three known bugs known in the operating system at the time of writing of this report.

3.2 Cloning & Running CrazyOS

Since its inception, CrazyOS's version control has been done using Git and the repository has been hosted on Github. To download the source files of the project on a local machine, the user must first install Git and generate SSH keys and link them with his account on Github. Once these preliminary tasks have been done, the user has to type the following command in a terminal to clone CrazyOS:

```
$ git clone git@github.com:PraneetKapoor2619/CrazyOS.git
```

The $ should be ignored; it is just a standard way of telling readers that they should type the text ahead in a terminal.
Once the project has been cloned, the reader has to make sure that the version of CrazyOS that he uses has the short commit ID 1a17fa0. For this, the following command has to be entered in the terminal:

```
$ git reset --hard 1a17fa0
```

To move to the project directory, the following command is executed:

```
$ cd CrazyOS/8086/
```

Once in the project directory, the user must create a build directory by the following command:

```
$ mkdir build
```

The binary files and the disk images are built using `make build` command. To run the operating system, type

```
$ make run
```

To exit QEMU without powering off the emulator from CrazyOS's shell, just press `Ctrl + Alt + Q`. `build` directory is cleaned by using `make clean` command. To remove `asm.bin` and `disk2.bin` in the `build` directory, the following commands should be used:

```
$ rm build/asm.bin build/disk2.bin
```

3.3 Using the Shell

Upon running the emulation, the user will be greeted with a startup screen showing a text message, and current time and date, as shown in Figure 3.1).

Figure 3.1: CrazyOS startup screen

3.3.1 Using time, date, srng, and clear

Using `time` and `date` command the user can know the current time and date, respectively. By typing `srng` multiple times, the user can generate different pseudo-random numbers. To clear the screen, `clear` command is used. Output of these commands are shown in Figure 3.2.

(a) Using `time`, `date`, and `srng` commands

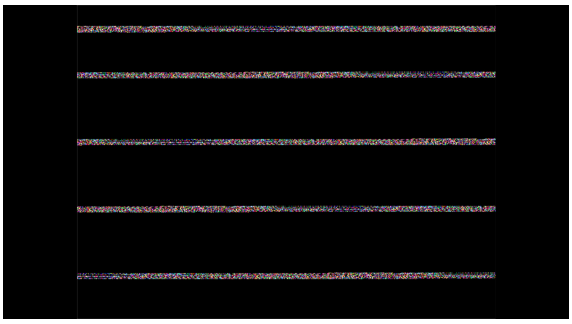
(b) Effect of `clear` command

Figure 3.2: Using `time`, `date`, `srng`, and `clear` commands

3.3.2 Running Disk Applications

The binary of `../disk-apps/soundnlight.asm` is loaded at the first sector of `disk2.bin`. The application is first read using `disk read` command. In Figure 3.3a, we have loaded the application in the 5th block. To run it, we type `run 5000:0000`. This performs an intersegment call and runs the application, thus producing coloured pixels and notes corresponding to the byte of the kernel that is being read, as shown in Figure 3.3b. As this application has no far return instruction, `retf`, we need to kill the emulation using `Ctrl + Alt + Q` key combo.

(a) Using `disk read` and `run`



(b) Running `soundnlight`

Figure 3.3: Using `disk read` and `run` commands to run a disk application

3.3.3 Power Commands

`power` command is used to manage power of the machine using APM. Its use is shown in Figure 3.4. Note that the command `power level` prints `Power level unknown`.

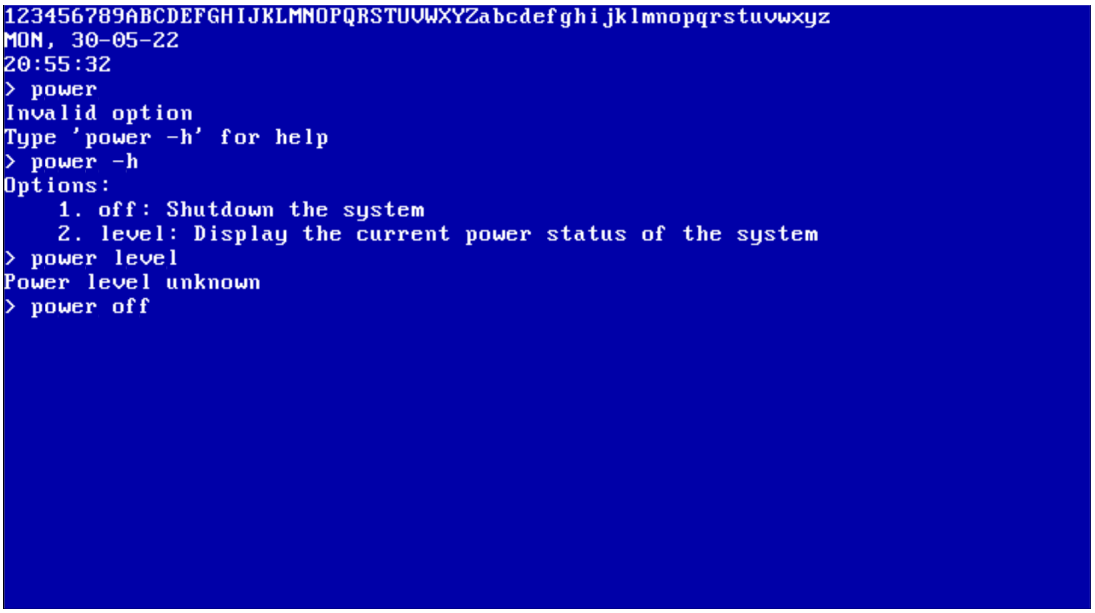
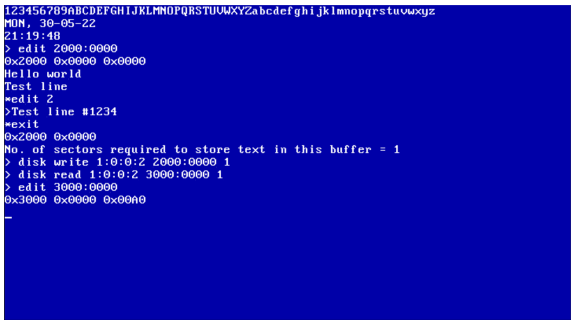


Figure 3.4: Using `power` command

3.4 Using the Editor

The line editor is ran from the shell by using the `edit` command. The address of the buffer should be passed as an argument. If the buffer is empty, then the total number of bytes is shown to be zero. Files are edited and their contents are listed using `*edit` and `*list` commands, respectively. Note that `*list` command clears the screen and then displays the contents of the buffer. To exit the editor, `*exit` command is used. Contents of the buffer are written to the disk using `disk write` command. Figure 3.5a shows these commands in action.

The same sectors of the disk to which we have written earlier can be read again and opened and listed in the editor. Figure 3.5b is a proof that the line editor and disk read/write commands are able to successfully able to create, save and edit files.



(a) Saving a new file to disk

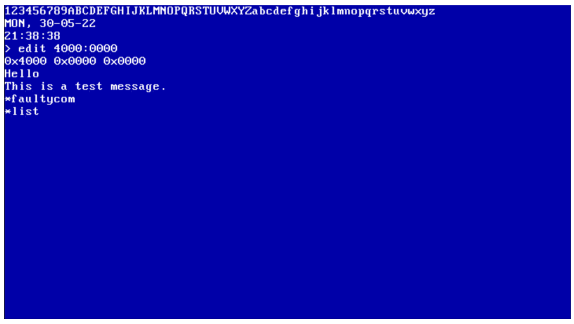
(b) Successful operation of `edit` and `disk`

Figure 3.5: Using `edit` to create and edit a file

3.5 Bugs

The project has only three known bugs which is astonishing because it has been completely written in assembly language. These bugs are:

1. `power level` command is not able to show the charge in the battery of the machine on which the emulator is running. This is shown in Figure 3.4.
2. Executing an invalid command string in the line editor insert that command string except for the asterisk. This is shown in Figure 3.6.
3. System might become unstable if an intersegment call is performed.



(a) Entering an invalid command in the editor

(b) Listing of the file reveals the bug

Figure 3.6: Bug in line editor causing it include contents of invalid commands

3.6 Discussion

CrazyOS is comparable to disk operating systems (DOS) like CP/M and MS-DOS v1.25. Both MS-DOS v1.25 and CrazyOS have many similarities, few of which are:

1. Both operating systems work in real mode.
2. The command-prompt/shell of both of these operating system has commands built into the source code of the shell itself.
3. Both operating systems make use of BIOS calls to access the hardware.
4. Both of these operating system are meant for a single user only.
5. Neither CrazyOS nor MS-DOS v1.25 has the ability to multitask between different software.

However, unlike MS-DOS, CrazyOS

1. does not have any system calls which makes it heavily dependent on the BIOS calls provided by the OEM.

2. does not support running applications having binary size greater than 64 kiB. Returning to shell from smaller applications after an intersegment call makes the shell unstable.

3. does not use proper file allocation table (FAT). Therefore, disk space is not utilized efficiently.

All modern day operating systems, like Linux kernel based distros and Windows 10, are far superior than DOS. They are one of the most complex pieces of software in the world which provide programmers and everyday users with hardware support, system calls, and file systems among many other things which allow them to use the resources of the machine efficiently and get their job done. With millions of lines of codes, even tech-gurus can only master a few integral components of the operating system in their lifetime. Clearly, the simplicity of CrazyOS comes at the cost of functionality and makes it a toy operating system which can be used for educational purposes. Porting CrazyOS for x86-64 and ARM based processors while implementing a few fundamental system calls and a file system will tremendously improve its usefulness.

CHAPTER 4: CONCLUSION & FUTURE SCOPE

4.1 Conclusion

We have successfully implemented a single-user, single-tasking, real-mode operating system named CrazyOS. The operating system comes with two applications: a shell and a line editor. Both of these applications have been demonstrated to work successfully. Various subroutines responsible for handling devices and for processing user commands form the library of the operating system. These library functions can be used to develop various disk applications which can then be run from the shell, as has been successfully demonstrated by running `soundnlight.asm` program.

4.2 Future Scope

The total lines of code which are there in the project directory can be computed with the help of `cloc` utility. It is revealed that there are in total 1718 lines of assembly code. If one excludes the number of lines of code in `soundnlight.asm` program, then the total lines of code for the project comes out to be 1690. Both of these numbers show that the operating system is small enough to be read, understood, used, and played by a student over the course of their undergraduate study. If it is incorporated correctly in courses related to computer architecture and x86 microprocessor family, this operating system can prove to be useful for disseminating practical knowledge of the aforementioned project.
The project can be further expanded in the following ways:

1. As an educational exercise, versions of CrazyOS working on computers using i386, x86-64, ARM and RISC-V processors can be developed.

2. Documentation related to each version of CrazyOS can be published online. This will help in increasing the number of people who learn from it.

3. A separate operating system can be developed for Raspberry Pi line of single board computers (SBC). These are easily and cheaply available all around the world. When playing with a toy operating system on a cheap SBC, the user will not have to worry about bricking their main computer. Furthermore, as RPis are increasingly being used in embedded

systems, a lightweight operating system will prove to be a boon for embedded systems developers.

We would conclude this report by quoting the words of Linus Torvalds and Terry Davis: This operating system has been developed because it was a fun challenge. Unlike Linux or Windows, it is not a professional operating system. In comparison to them, it is just a dirt-bike using which the user can go on accomplishing fun little challenges of their own liking.

REFERENCES

[1] *The birth of the IBM PC*. IBM Archives. URL https://www.ibm.com/ibm/history/exhibits/pc25/pc25_birth.html.

[2] *The 8086 Family Users Manual*. Intel Corporation, 1979.

[3] *8086 16-bit HMOS Microprocessor 8086/8086-2/8086-1*. Intel Corporation, 1990.

[4] *8254 Programmable Interval Timer*. Intel Corporation, 1993.

[5] *Advanced Power Management (APM): BIOS Interface Specification. Revision 1.2*. Intel Corporation, and Microsoft Corporation, 1996.

[6] *Hthreads Glossary*. Hthreads, 2006. URL https://web.archive.org/web/20070826224349/http://www.ittc.ku.edu/hybridthreads/glossary/index.php.

[7] *The NASM Documentation*. The NASM development team, 2015. URL https://www.nasm.us/xdoc/2.15.05/html/nasmdoc0.html.

[8] Steve Wozniak points at lack of creativity in Indian education system. *The Times of India*, Mar 2018. URL https://timesofindia.indiatimes.com/home/education/news/steve-wozniak-points-at-lack-of-creativity-in-indian-education-system/articleshow/63265853.cms.

[9] Popularity of Engineering streams depends on job scenario. *The Times of India*, Mar 2022. URL https://timesofindia.indiatimes.com/home/education/news/popularity-of-engineering-streams-depends-on-job-scenario/articleshow/90319832.cms.

[10] Richard Vernon Andree. *Programming the IBM 650 magnetic drum computer and data-processing machine*. Hollis Publishing Company, 1958.

[11] Ionescu Andreea. Categories and Generations of Computers. *European Journal of Computer Science and Information Technology*, 3(1):15–42, 2015.

[12] Prof. Anupam Basu et al. *Model Curriculum for Undergraduate Degree Courses in Engineering & Technology*. AICTE, 2018.

[13] Fabrice Bellard. QEMU: A Fast and Portable Dynamic Translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005.

[14] Nick Blundell. *Writing a Simple Operating System—from Scratch*. 2009.

[15] Ralph Brown and Marc Perkel. *Ralph Brown's Interrupt List: Indexed HTML Version - Release 61*. URL `https://www.ctyme.com/rbrown.htm`.

[16] Prof. S. Chapman. Tercentenary of the calculating machine. *Nature*, 150(3809):508–509, 1942.

[17] Yaohan Chu. *High-Level Language Computer Architecture*. Academic Press, 1975.

[18] Frank Cruz. *Programming the ENIAC*. Columbia University, 2013. URL `http://www.columbia.edu/cu/computinghistory/eniac.html`.

[19] Subrata Dasgupta. *It Began with Babbage: The Genesis of Computer Science*. Oxford University Press, 2014.

[20] Kyriakos Efstathiou and Marianna Efstathiou. Celestial Gearbox. *Mechanical Engineering*, 140(09):31–35, 09 2018. doi: 10.1115/1.2018-SEP1. URL `https://doi.org/10.1115/1.2018-SEP1`.

[21] Claire L. Evans. *Broad Band: The Untold Story of the Women Who Made the Internet*. Penguin, 2020.

[22] Linda Weiser Friedman. From Babbage to Babel and Beyond: A Brief History of Programming Languages. *Computer Languages*, 17(1):1–17, 1992.

[23] Eldon C Hall. *Journey to the Moon: The History of the Apollo Guidance Computer*. Aiaa, 1996.

[24] Ber Halpern. *Large Eddy Simulation of Complex Engineering and Geophysical Flows*. Cambridge University Press, 1993.

[25] Hans Dieter Hellige. *Geschichten der Informatik: Visionen, Paradigmen, Leitmotive*. Springer-Verlag, 2013.

[26] Donald Ervin Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Pearson Education, 1997.

[27] Ariel Kyrou. L'art brut est contemporain. *Multitudes*, 4(69):19–29, 2017.

[28] LionKimbro. A diagram of cylinders/tracks and sectors on a platter. Also, a diagram of four (4) stacked platters, and eight (8) read/write heads. URL `https://commons.wikimedia.org/wiki/File:Cylinder_Head_Sector.svg`.

[29] Robert L Mitchell. Cobol: Not Dead Yet. *Computerworld*, 40(41):25–37, 2006.

[30] Marcello Morelli. Dalle calcolatrici ai computer degli anni cinquanta. *FrancoAngeli, Milano (in Italian)*, 2001.

[31] James L Peterson and Abraham Silberschatz. *Operating System Concepts*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[32] Dennis M Ritchie and Brian Kernighan. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[33] T. Robinson. The Meccano Set computers: A History of Differential Analyzers Made from Children's Toys, 2005. URL `https://doi.org/10.1109/mcs.2005.1432602`.

[34] Raúl Rojas and Ulf Hashagen. The ENIAC: History, Operation and Reconstruction in VLSI. In *The First Computers: History and Architectures*, pages 121–178. MIT Press, 2002.

[35] Peter Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, 2009.

[36] Kounteya Sinha. India set to produce world's largest number of engineers. *The Times of India*, Oct 2015. URL `https://timesofindia.indiatimes.com/india/India-set-to-produce-worlds-largest-number-of-engineers/articleshow/49532113.cms`.

[37] Cliff Stoll. When slide rules ruled. *Scientific American*, 294(5):80–87, 2006.

[38] Bob Supnik. Simulators: Virtual Machines of the Past (and the Future). *Queue*, 2(5): 52–58, 2004.

[39] Andrew S. Tannenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. 2003.

[40] Linus Torvalds and David Diamond. *Just for fun: The story of an accidental revolutionary*. Harper Audio, 2001.

[41] John von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

RESEARCH PAPER

CrazyOS: A Case for Toy Operating Systems

Praneet Kapoor
Electrical and Electronics Engineering
Dr. Akhilesh Das Gupta Institute of Technology and Management
New Delhi, India
kapoorpraneet2619@gmail.com

Rachit Jain
Electrical and Electronics Engineering
Dr. Akhilesh Das Gupta Institute of Technology and Management
New Delhi, India
rachitjain24@gmail.com

Shristi Sharma
Electrical and Electronics Engineering
Dr. Akhilesh Das Gupta Institute of Technology and Management
New Delhi, India
shristisharma1808@gmail.com

Sumit Gaur
Electrical and Electronics Engineering
Dr. Akhilesh Das Gupta Institute of Technology and Management
New Delhi, India
sumit.gaur1999@gmail.com

Md. Saquib Faraz**
Electrical and Electronics Engineering
Dr. Akhilesh Das Gupta Institute of Technology and Management
New Delhi, India
mdsaquibfaraz@gmail.com

***Abstract*—Technical education of undergraduate students should ensure that they are provided with practical knowledge which makes them fit for the ever changing industry. One of the ways of accomplishing this is by attempting to include project based learning alongside regular coursework. This project is a step towards this direction. We have presented a single-user, single-tasking, real mode operating system. The operating system comes with two applications, a shell and a line editor. A library of subroutines has also been created which includes custom implementation of getchar, putchar, printf, strlen, strcmp, strcpy and atoi in x86 assembly language. With only 1690 lines of code, the source code of the operating system can be easily understood by students and then used as a sandbox for learning about the design of the 8086 microprocessor and the original IBM PC.**

***Index Terms*—Real mode, Operating system, QEMU, NASM, BIOS, Make, Makefile, shell, line editor, 8086, IBM PC.**

I. Introduction

A. Motivation

Engineering is a practical profession. A good engineer should be able to use fundamental principles to solve problems or derive practical results. The ability to do so requires experience, which is often termed as "*getting hands dirty*". It is of primary importance that an engineering student should have their hands sufficiently dirty in the profession they wish to pursue. A student interested in pursuing a career in silicon industry should have considerable experience in designing systems on an FPGA board. If he[1] has only memorized the important principles from some standard book, that student will have no value in the industry. It is because of this reason that academicians and professors all around the globe try to the best of their knowledge to keep the curriculum in sync with the advances that are being made in the industry. Despite all their efforts, there will always remain a few deficiencies in the roadmap thus produced. Education system of a few countries and the curriculum followed in a few top-tier institutions is considered as the gold-standard for technical education by other countries and institutions. An educated discussion about the about the state of engineering education in different countries across the globe will deviate us from the subject matter of this report. Therefore, we restrict our discussion to the state of technical education in India. This is appropriate as all the authors of this paper are of Indian nationality and have first hand experience of the education system.

In India, institutes offering technical education need to get certified by the All India Council for Technical Education (AICTE). AICTE has produced a model curriculum for technical education. Engineering Institutions can either follow this curriculum or develop their own using the model curriculum as a template. As the project presented in this paper is related to computer architecture and operating systems, we will take the example the course on Computer Architecture offered to Electrical Engineering undergrads [5]. There are six modules in this course. Module 1 tells about the differences between CISC and RISC, data types, system buses, and multi-bus organization. Module 2 then jumps to memory organization, followed by Module 3 discussing I/O devices, PCI and PCI Express bus. Module 4 then discusses x86 architecture, and its addressing modes and instruction. Module 5 discusses instruction level pipelining (ILP) and module 6 discusses other architectures like MIPS. Three flaws can be found in this course:

1) *Undirected and overloaded course structure*: The course, and other similar courses, can be considered to be well designed theoretically. However, they are overloaded with theoretical concepts. It would be better to just focus on the architecture of single computer, preferably IBM PC. The course can then describe the 8086 microprocessor and later describe ILP, and paging. Then the architecture of the IBM PC can be dealt with in detail.
2) *Impractical course structure*: Continuing with the ex-

[1]"He" should be read as "he or she" throughout this paper.

ample provided previously, most of the courses dealing with microprocessors start of with 8085 microprocessor. Though it is still used in legacy systems in the industry, no new projects are developed using it. Students would be better served by teaching them about ARM or RISC-V or x86 architectures as most of the computing systems today are built around them. Furthermore, it must be noted that the practical assignments given to students are trivial. Examples include writing subroutines to multiply numbers, or to move strings in 8086's assembly language. A more practical exercise would be to write subroutines to print a string on the screen or to separate words out from a string.

3) *Obsolete tools*: Most of the tools, be it teaching resources or software tools, used in Indian schools and colleges are obsolete. Instead of following texts like *The 8086 Family User's Manual* [1], students and teachers alike follow notes and presentations which don't talk about the practical side of various concepts like segmentation, effective address calculation and string operations, to name a few. Courses on C programming and data structures still require students to develop and run code in Turbo C environment. This is ridiculous because Turbo C's compiler does not follow the standard implementation of C and C++. In the course on microprocessors, codes are run on expensive trainer kits or on emulators, like emu8086, which is not the industry standard for developing 8086 assembly code.

Disgruntled by the approach taken in teaching one of his favourite subjects, one of the authors of this paper (Kapoor), decided to write his own 16-bit operating system from scratch. He considered this to be fun challenge which would allow him to tinker with the hardware and understand the architecture of 8086 microprocessor and the original IBM PC. His objective at first was to develop a UNIX like operating system. However, as UNIX and its derivatives (Linux, MINIX [10], BSD, etc.) are complex systems and developing them would require ample amount of free time, it was decided that operating system so designed would be a simple one consisting of just a primitive shell and line editor. Basic commands can be executed from shell. Students can develop the their own programs using the libraries provided by the author, assemble using NASM, create a virtual disk and run it in real mode on QEMU's emulation of i386 based IBM PC. Due to such unusual design of the operating system, it was named *CrazyOS* by its creator.

B. Literature Survey

To develop concrete objectives, a brief of review of histor of computing systems was made. History of computers involve studying their evolution. Just like a study of human history consists of the study of various civilizations and generations, a study of computer history involves the study of various generations of computers. Beginning of a new generation are marked with the usage of newer, better, and reliable technology in comparison to the previous generation. The technology most used for this classification is the hardware. By using hardware as the basis for classification, most authors divide computer history into five generations, starting in 1951 with the launch of UNIVAC I [4][8]. By using programming languages and operating systems as the basis for classification, we arrive at similar results [10]. Table I shows various generations of computers and the operating system used in those generations. After reviewing the history of computing systems, it was decided that the operating system will be developed in the spirit of MS-DOS v1.25 whose source code has been made public by Microsoft [9].

C. Objectives

The objective of this project was to develop a simple single-user, single-tasking, real mode operating system from scratch for educational purposes. This objective was achieved by completing the following goals:

1) To develop a bootloader from scratch.
2) To develop a set of libraries containing certain standard functions found in C's `stdio.h` and `string.h` libraries. This would assist the user in building disk applications for the system.
3) To develop a shell which enables the user to issue basic commands to interact with system components like disks, APM, etc.
4) To develop file system should be simple enough to understand, implement and use with programs like the line editor. For these reasons, the file system uses contiguous sector allocation [10].
5) To develop a build environment which allows the user to write and build their own disk applications, and run them using the operating system.

II. Implementation of CrazyOS

A. Set-up of the Development Environment

The development environment of CrazyOS consists of five software tools: code editor, assembler, emulator, build automation software, and source version control software.

Initially, the author of the project (Kapoor) used gVim for writing and editing source code. However, as the size of the project grew, the author found himself to be unable to work productively with gVim. For this reason, a switch to VScode was made.

NASM was chosen over GNU AS because of its support for Intel syntax [3]. Most technical institutions use Intel syntax during course on 8086 processor. Using NASM meant that students learning about x86 architecture won't have to learn a new syntax (AT&T syntax) of writing assembly code.

QEMU was used for emulating an i386 based IBM PC [6]. GNU Make was used for automatically building binaries, running the emulator, and removing the binaries. Git was used for tracking the files in this project.

B. Structure of the Project Directory

CrazyOS started out as a project to teach its author about the 8086 microprocessor and the IBM PC. The author expects to write similar toy operating systems for computers using

TABLE I: Generations of Computers on the Basis of their Hardware

Generation	Year	Hardware	Operating System
1st Generation	1951-1960	Vacuum tubes	Non-existent
2nd Generation	1959-1965	Transistors	Batch Processing
3rd Generation	1964-present	Integrated circuits	Multiprogramming, Time-sharing, UNIX
4th Generation	1971-present	VLSI, Microprocessors	Disk Operating System (DOS), Linux distros, Windows NT
5th Generation	Present-future	Parallel processing architecture	

i386, x86-64, ARM and RISC-V processors. These ambitions are reflected in the commit history and by the structure of the parent directory. Assuming the reader clones the project from Github into a directory named `CrazyOS`, the directory tree that will be obtained is shown in Figure 1. *depth = 0* indicates the number of files and subfolders that are being shown in the source tree.

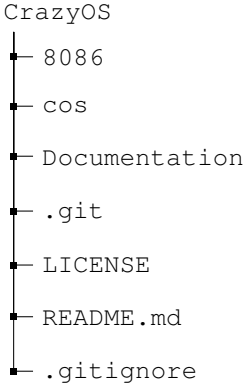


Fig. 1: Parent directory tree (depth = 0)

As has been mentioned before, the directory `../8086`[1] contains the source files for the project. The tree of this directory is given in Figure 2.

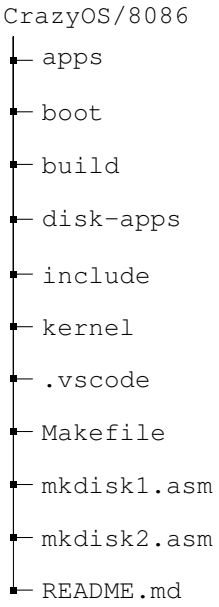


Fig. 2: Directory tree for `CrazyOS/8086` (depth = 0)

C. Bootloader and Kernel Header

The bootloader was the first program written in this project. It is contained in the `../8086/boot` directory. It is a very simple bootloader which uses standard BIOS interrupt no. 13 to load the operating system from disk into the main memory. The bootloader loads 32 sectors from the first disk at address 1000:0000. It then performs an intersegment call to `kernel_entry` subroutine present in `../8086/kernel/kernel.asm`. If `kernel.asm` forms the head of the operating system, then `kerne_entry` subroutine would be the scalp and `kernel_main` would be the cranium of the operating system. The bootloader performs an intersegment call to `kernel_entry` subroutine which has been loaded, along with the rest of the operating system, in block 1. `kernel_entry` subroutine is responsible for initializing `cs`, `ds`, `es`, and `ss`.

```
kernel_entry:
        mov ax, cs
        mov ds, ax
        mov es, ax
        mov ax, 0x9000
        mov ss, ax
        mov bp, 0x9ffe
        mov sp, bp
```

`kernel_main` calls the `main` subroutine in `../apps/shell/shell.asm`. This launches the shell from where the user can execute various commands.

D. Library Functions

In CrazyOS, library functions are placed in `../8086/include` folder. Source tree of this directory is shown in Figure 3. These functions use BIOS calls to access hardware facilities. In fact, the whole operating system is heavily dependent on BIOS interrupt calls for

[1] `..` is used to refer to the parent directory, in this case `CrazyOS`. In fact, `CrazyOS` is itself a part of some directory and therefore it can be written as `../CrazyOS`.

device management. These BIOS calls can be thought of as CrazyOS' system calls. These interrupts were studied and used by following *Ralph Brown's Interrupt List* [7]. The subroutines present in the library are analogous to functions present in the standard C library such as `printf`, `scanf`, `atoi`, etc., which a programmer often uses in their application to get the job done. Subroutines dealing with printing text on screen and taking input from the keyboard are in `../include/ttyio` directory. Subroutines performing string operations such as `strlen` (for finding length of the string), `strcmp` (for comparing two strings), and `atoi` (for converting a number in string format into a positive integer) are present in `../include/string/string.asm` file.
`..include/string` directory has another assembly source file besides `string.asm`. It is `lexer.asm` which is responsible for breaking a string of characters into separate "words" using some separator. The function used for performing this primitive lexical-analysis is `cmd_lexer`. It *Tokenizes* the string that is being pointed at by `si` using the character stored in `al` as a separator. It generates its own personal copy of the string which is stored in local label `.COPY`. This subroutine is capable of generating a maximum of 5 tokens: one command token and four argument tokens. Command token is stored in global label `com` and argument tokens are stored in `arg1`, `arg2`, `arg3`, and `4` which are also global labels.
Subroutines which allow the user to read and write to disks are in `../include/disk` directory. `../include/apm` contains subroutines for using Advanced Power Management (APM) for managing power of the device (or in this case, an emulator).
Time and date is obtained from CMOS memory on RTC on the motherboard using the subroutines present in `../include/cmos` directory. The subroutines in this directory are also used to produce pseudo-random numbers by `srng` subroutine in `../include/random/srng.asm`. These pseudo-random numbers are generated by the multiplication of seconds count in CMOS with a 16-bit prime number and then addition of the resulting product with a different 16-bit prime number.
Finally, subroutines present in `../include/sound` directory are used to produce sounds by feeding the piezoelectric speaker on the motherboard with a square wave of chosen frequency. This square wave is generated by 8254 Programmable Interrupt Timer (PIT) chip [2].

E. Shell

The shell is considered as the *mother of all programs*. It is one of the first programs which runs in most operating systems. It provides the user an environment where he can issue commands and run other programs [11]. CrazyOS' shell is implemented by the source code present in `../apps/shell` directory. This directory contains two source files: `shell.asm` and `cmdlist.asm`. There is only a single subroutine present in `shell.asm`, `main`. This subroutine firstly prints the prompt, `>`, using `printf`. Then

```
CrazyOS/8086/include
├── apm
│   ├── apm_com.asm
│   └── apm.asm
├── cmos
│   ├── cmos_com.asm
│   └── cmos.asm
├── disk
│   ├── disk_com.asm
│   └── disk.asm
├── random
│   └── srng.asm
├── sound
│   └── sound.asm
├── string
│   ├── lexer.asm
│   ├── README.txt
│   └── string.asm
└── ttyio
    ├── legacyprint.asm
    ├── printf.asm
    ├── README.txt
    ├── stdtty.asm
    └── ttyio.asm
```

Fig. 3: Directory tree for `CrazyOS/8086/include` (depth = 1)

it calls `getline` subroutine which takes input from the keyboard until the enter key gets pressed. Then using a blank space as the token separator, `lexer` splits the line into command word and arguments which are then executed.
The commands which are supported by the shell are present in `cmdlist.asm`. `com` string is compared with the strings present in this file one by one until a match is found. If no match has been found then an error message is printed on the screen.

F. File System and the Line Editor

CrazyOS uses contiguous file allocation which stores files in consecutive blocks of memory [10]. This makes the design

of the operating system even simpler. To read or write a file to a disk, the user just needs to read consecutive blocks till the end of the file has been reached. End of text files made using editor provided with CrazyOS is indicated by a byte having the value 0xaa.

This simple file system is used by the line editor whose source code is present in `../apps/edit` directory. The line editor is started by typing `ed SEG:OFF` in the shell. Here, `SEG` is the 16-bit segment base address and `OFF` is the 16-bit offset from the base address at which the contents of file will be stored. Due to the use of segments, the size of the file cannot exceed 64 kiB or about 819 80-character wide lines. Also, each file is terminated with a magic number, 0xaa. This hex-number maps to *á* in code page 437. As in this simple project we are not using accented letters, therefore, we do not have to worry about the loss of characters which might happen to exist in the file buffer after 0xaa.

Once the editor starts, the user is told about the number of characters that have been found in the file buffer. Like `ed`, `edit` allows the user to enter a single line at a time. The line so entered should not exceed 80 character limit. If the first character of the line is an asterisk, than the subsequent characters are considered as editor commands and are not stored in the file buffer. The file can be saved permanently by using `disk write` command to write the buffer to the disk.

G. Disk Applications: ../disk-apps

As we have discussed before, applications which come on a separate disk are called disk applications in this project. Source code for disk applications are saved at `../disk-apps` directory. At present, the whole project has only a single disk application, namely `soundnlight.asm`. This assembly file contains code which reads a single byte from the kernel (present in block 1), and

1) displays a pixel having a color corresponding to the value of the byte that has been read using INT 10,0C BIOS call.
2) generates a note corresponding the value of the byte that has just been read using `sound_tiny` subroutine present in `../include/sound/sound.asm`.

Then the pointer `si` is incremented and the next byte of the kernel is read. The screen is assumed to have a size of 640×480 pixels. Other available screen resolutions are 640×350 and 320×200 pixels.

III. Results & Discussion

A. Cloning & Running CrazyOS

Since its inception, CrazyOS's version control has been done using Git and the repository has been hosted on Github. To download the source files of the project on a local machine, the user must first install Git and generate SSH keys and link them with his account on Github. Once these preliminary tasks have been done, the user has to type the following command in a terminal to clone CrazyOS:

```
$ git clone git@github.com:PraneetKapoor2619/CrazyOS.git
```

The $ should be ignored; it is just a standard way of telling readers that they should type the text ahead in a terminal.

Once the project has been cloned, the reader has to make sure that the version of CrazyOS that he uses has the short commit ID 1a17fa0. For this, the following command has to be entered in the terminal:

```
$ git reset --hard 1a17fa0
```

To move to the project directory, the following command is executed:

```
$ cd CrazyOS/8086/
```

The reader can use the latest version of the operating system but it might not be stable. The reader should refer to commit messages if he wishes to the latest version of CrazyOS.

Once in the project directory, the user must create a build directory by the following command:

```
$ mkdir build
```

The binary files and the disk images are built using `make build` command. To run the operating system, type

```
$ make run
```

To exit QEMU without powering off the emulator from CrazyOS's shell, just press `Ctrl + Alt + Q`. `build` directory is cleaned by using `make clean` command. To remove `asm.bin` and `disk2.bin` in the `build` directory, the following commands should be used:

```
$ rm build/asm.bin build/disk2.bin
```

B. Using the Shell

Upon running the emulation, the user will be greeted with a startup screen showing a text message, and current time and date, as shown in Figure 4).

Fig. 4: CrazyOS startup screen

User can enter commands into the shell to get their job done. An example is shown in Figure 5, where `time`, `date`, and `srng` commands have been used.

Fig. 5: Using `time`, `date`, and `srng` commands

C. Bugs

The project has only three known bugs which is astonishing because it has been completely written in assembly language. These bugs are:

1) `power level` command is not able to show the charge in the battery of the machine on which the emulator is running..
2) Executing an invalid command string in the line editor insert that command string except for the asterisk.
3) System might become unstable if an intersegment call is performed.

D. Discussion

CrazyOS is comparable to disk operating systems (DOS) like CP/M and MS-DOS v1.25. Both MS-DOS v1.25 and CrazyOS have many similarities, few of which are:

1) Both operating systems work in real mode.
2) The command-prompt/shell of both of these operating system has commands built into the source code of the shell itself.
3) Both operating systems make use of BIOS calls to access the hardware.
4) Both of these operating system are meant for a single user only.
5) Neither CrazyOS nor MS-DOS v1.25 has the ability to multitask between different software.

However, unlike MS-DOS, CrazyOS

1) does not have any system calls which makes it heavily dependent on the BIOS calls provided by the OEM.
2) does not support running applications having binary size greater than 64 kiB. Returning to shell from smaller applications after an intersegment call makes the shell unstable.
3) does not use proper file allocation table (FAT). Therefore, disk space is not utilized efficiently.

All modern day operating systems, like Linux kernel based distros and Windows 10, are far superior than DOS. They are one of the most complex pieces of software in the world which provide programmers and everyday users with hardware support, system calls, and file systems among many other things which allow them to use the resources of the machine efficiently and get their job done. With millions of lines of codes, even tech-gurus can only master a few integral components of the operating system in their lifetime. Clearly, the simplicity of CrazyOS comes at the cost of functionality and makes it a toy operating system which can be used for educational purposes. Porting CrazyOS for x86-64 and ARM based processors while implementing a few fundamental system calls and a file system will tremendously improve its usefulness.

IV. Conclusion & Future Scope

A. Conclusion

We have successfully implemented a single-user, single-tasking, real-mode operating system named CrazyOS. The operating system comes with two applications: a shell and a line editor. Both of these applications have been demonstrated to work successfully. Various subroutines responsible for handling devices and for processing user commands form the library of the operating system. These library functions can be used to develop various disk applications which can then be run from the shell, as has been successfully demonstrated by running `soundnlight.asm` program.

B. Future Scope

The total lines of code which are there in the project directory can be computed with the help of `cloc` utility. It is revealed that there are in total 1718 lines of assembly code. If one excludes the number of lines of code in `soundnlight.asm` program, then the total lines of code for the project comes out to be 1690. Both of these numbers show that the operating system is small enough to be read, understood, used, and played by a student over the course of their undergraduate study. If it is incorporated correctly in courses related to computer architecture and x86 microprocessor family, this operating system can prove to be useful for disseminating practical knowledge of the aforementioned project.

The project can be further expanded in the following ways:

1) As an educational exercise, versions of CrazyOS working on computers using i386, x86-64, ARM and RISC-V processors can be developed.
2) Documentation related to each version of CrazyOS can be published online. This will help in increasing the number of people who learn from it.
3) A separate operating system can be developed for Raspberry Pi line of single board computers (SBC). These are easily and cheaply available all around the world. When playing with a toy operating system on a cheap SBC, the user will not have to worry about bricking their main computer. Furthermore, as RPis are increasingly being used in embedded systems, a lightweight operating system will prove to be a boon for embedded systems developers.

We would conclude this paper by quoting the words of Linus Torvalds[11] and Terry Davis This operating system has been developed because it was a fun challenge. Unlike Linux

or Windows, it is not a professional operating system. In comparison to them, it is just a dirt-bike using which the user can go on accomplishing fun little challenges of their own liking.

ACKNOWLEDGMENT

We would like to thank our family and friends for supporting us during tough times.

REFERENCES

[1] *The 8086 Family Users Manual*. Intel Corporation, 1979.

[2] *8254 Programmable Interval Timer*. Intel Corporation, 1993.

[3] *The NASM Documentation*. The NASM development team, 2015. URL https://www.nasm.us/xdoc/2.15.05/html/nasmdoc0.html.

[4] Ionescu Andreea. Categories and Generations of Computers. *European Journal of Computer Science and Information Technology*, 3(1):15–42, 2015.

[5] Prof. Anupam Basu et al. *Model Curriculum for Undergraduate Degree Courses in Engineering & Technology*. AICTE, 2018.

[6] Fabrice Bellard. Qemu, a fast and portable dynamic translator. 41(46):10–5555, 2005.

[7] Ralph Brown and Marc Perkel. *Ralph Brown's Interrupt List: Indexed HTML Version - Release 61*. URL https://www.ctyme.com/rbrown.htm.

[8] Linda Weiser Friedman. From Babbage to Babel and Beyond: A Brief History of Programming Languages. *Computer Languages*, 17(1):1–17, 1992.

[9] Tim Paterson. MS-DOS v1.25. https://github.com/microsoft/MS-DOS/tree/master/v1.25, 2018.

[10] Andrew S. Tannenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. 2003.

[11] Linus Torvalds and David Diamond. *Just for fun: The story of an accidental revolutionary*. Harper Audio, 2001.

CURRICULUM VITAE

Praneet Kapoor

Email: kapoorpraneet2619@gmail.com Mobile: +91-931-9311-759
Github: github.com/PraneetKapoor2619

Education

- **Dr. Akhilesh Das Gupta Institute of Technology and Management** New Delhi, India
  Bachelor of Technology - Electrical and Electronics Engineering; CGPA: 9.121 *August 2018 - August 2022*
  Courses: *Fundamental Circuit Theory, Analog and Digital Circuits, VLSI, Microprocessors and Microcontrollers*

Skills Summary

- **Languages**: x86 Assembly Language, AVR 8-bit Assembly Language, C, C++, Bash, Lua, Python 3, SQL
- **Frameworks**: NumPy, Matplotlib, SciPy
- **Tools**: GCC, Make, NASM, SQLite 3, Git, QEMU, VMware, Docker, LTSpice XVII, Arduino IDE
- **Platforms**: Linux, Windows, AVR 8-bit microcontrollers, 6502, 8085, x86, x86-64, Raspberry Pi 3 Model B+
- **Soft Skills**: Technical Writing, Public Speaking, Project Management

Experience

- **Kyron Healthcare Pvt. Ltd.** On-site
  Firmware Design Intern (Full-time) *Aug 2021 - Nov 2021*
  - **Medical Humidifier**: Collaborated in the development of a reliable medical humidifier.
  - **Automated Flow Sensor Testing**: Implemented a Python script to automate the testing of medical venturi.
  - **Testing and Analysis of Nasal Cannula**: Tested the efficacy of PID controllers to control flow and oxygen levels with different nasal cannulas. Data was analysed and used in selection of flow sensors and in designing of robust PID controllers.
  - **Embedded PID Controller**: Developed algorithms for improving the reliability of embedded PID using fuzzy logic.

Projects

- **CrazyOS**: Single-user, single-tasking, real mode operating system with a shell, line editor, and library consisting of various subroutines including custom implementations of `getchar`, `printf`, `putchar`, `strlen`, `strcmp`, `strcpy`, and `atoi` in 8086 assembly language. Bootloader for the operating system has also been developed from scratch. Tech: x86 Assembly, NASM, Git, QEMU (Feb 2022 - Present)
- **SHA-256 Hash Generator**: Implemented SHA-256 hash in C. Uses a Python script to read text files and feed them into the C program which then emits the hash code. Tech: C, Python (Nov 2021 - Dec 2021)
- **Remote Controlled Rover**: Built custom designed rover using PVC pipes and hardboard. Uses RPi for processing commands passed through remote connection using SSH. Arduino Uno is used for locomotion and steering using Ackermann's steering equation. Tech: C, AVR-C, Python 3, Arduino Uno, Raspberry Pi 3 Model B+, SSH (Aug 2021 - Jan 2022)
- **Poor Engineering Student's Modular Oscilloscope (PESMO)**: A low frequency oscilloscope made using analog front-end, Arduino Uno as signal sampler, and a laptop running a Python script for serial communication as a visualizer. Won Intra-IEEE Project Making Competition, 2020. Tech: LTSpice XVII, Arduino Uno, Python 3 (Sep 2020 - Oct 2020)
- **Voyager Program CRS Data Extractor and Visualizer**: Scrapped Goddard Space Flight Center's Space Physics Data Facility to extract cosmic ray data collected by Voyager probes, stored them into an SQL database, and visualized using Matplotlib. Tech: Python 3, SQL(Jul 2020 - Aug 2020)
- **LM358D Analog Computer**: Low frequency analog computer using 6 op-amps, Arduino Uno as sampler, and Arduino IDE as signal visualizer. Tech: LTSpice XVII, Arduino Uno, Arduino IDE (Feb 2020 - Mar 2020)
- **Account Management System for Travel Agency**: Implemented an account management system for a travel agency from scratch. Features include ability to enter, retrieve, and edit customer information, and a total charge calculator. Tech: C++ (Dec 2016 - Mar 2017)

Honors and Awards

- IEEE ADGITM Volunteer of the Month - March 2021
- First Position at Intra-IEEE Project Making Competition - October 2020

Volunteer Experience

- **Chaiperson of PES and Technical Activities Lead, IEEE-ADGITM** New Delhi, India
  Organized webinars on embedded systems and IoT and project making competition for over 70 students. *Jan 2021 - Jan 2022*
- **Ambassador for the IEEE PES Day 2021** New Delhi, India
  Organized two webinars on power systems, renewable energy, and sustainable development with registrations and attendance reaching over 100 students and professors. *May 2021 - April 2021*

Rachit Jain

Email: rachitjain24@gmail.com
Mobile: +91-931-837-5168

LinkedIn

Education

- **Dr. Akhilesh Das Gupta Institute of Technology and Management** — New Delhi, India
  Bachelor of Technology - Electrical and Electronics Engineering; GPA: 8.8 — *August 2018 - August 2022*
  Courses: *Fundamental Circuit Theory, Analog and Digital Circuits, Microprocessors and Micro controllers, Data Structures*

Skills Summary

- **Languages**: Python, C++, Embedded C, Bash, JAVA, Assembly
- **Frameworks**: OpenCV, Matplotlib,Arduino IDE, AVR
- **Tools**: AutoCAD, MATLAB, Simulnk, VS Code
- **Platforms**: Linux, Web, Windows, Arduino, Raspberry pi,Azure
- **Soft Skills**: Leadership, Event Management, Writing, Public Speaking, Time Management

Experience

- **Banaao Innovation Lab Pvt. Ltd.** — Full Time
  Internet Of Things Developer(Full-time) — *May 2020 - July 2020*
  - **ESP32 and RTOS**: Worked on RTOS system in ESP32 Chipset
- **Eruditre LLP.** — Full Time
  ardware and IoT Developer — *June 2020 - Aug 2020*
  - **Projects done using RFID and Vechile Tracking**: Worked on RFID based attendance System and Vehicle Tracking using GPS
- **Ved World Pvt. Ltd..** — Full Time
  Embedded System Intern — *Oct 2020 - Nov 2020*
  - **IoT Project**: Responsible for creating IoT Aura Scanner for Astrological Purposes
- **RoboSlog Pvt. Ltd.** — Full Time
  Robotics Developer — *Jan 2021 - March 2021*
  - **ESP32 Based Project**: Worked on ESP32 Based Heart ECG Devices and ESP32 OTA Facility
- **Elecbits Technologies Pvt. Ltd.** — Full Time
  Firmware Design Intern — *July 2021 - Oct 2021*
  - **Multithreading in ESP32,AWS**: Working on IoT based solution , Multi threading in ESP32 Chipsets and AWS IoT Core Data Management
- **D'N'D Rodhak (IIT Mandi)** — Full Time
  Embedded Systems Developer — *March 2021 - July 2021*
  - **Road Safety Project**: Working on Road Safety based project like Alcohol Detection and Drowsiness Detection based systems.
- **Escorts Limited** — Full Time
  Research and Development Intern — *Oct 2021 - present*
  - **Agri Tech Project**: Working on Agri-Tech IoT Based Applications

Projects

- **Fault Detection System Using Opencv**: The Project was detecting the different shapes , the color and the area of the objects passing on a conveyer belt to detect faulty Objects
- **RFID Based Car Parking System**: The Project was all about providing a automated RFID security enhanced system for vehicles that are parked in the parking areas.
- **Smart Verticle Farming Units**: The Project was creating a smart farming solution , in which set of sensor were used to track the progress of plant growing in the unit
- **Bluetooth speakers using PAM 8403 Amplifier module**: The Project is all about making a Bluetooth speaker in which the wireless signals were converted by PAM 8403 module

Publications

- **Research Paper:Topic - Blind Vision and Theft protection device in Vehicles using Embedded System**: Proposed a solution for theft protection using embedded systems.

Achievements

- **Winner at IEEE ADGITM Project Design Competition**Created a Modular Oscilloscope and Plotted its graph using Matplotlib library in Python
- **Research Paper Published in IEEE SMARTGENCON confrence 2021** Topic - Blind Vision and Theft protection device in Vehicles using Embedded System.

Volunteer Experience

- **Conducted Worshop on Embedded systems and Aurduino** — Delhi, India
  Conducted offline technical & soft-skills training impacting over 30 students. — *June 2020*

Shristi Sharma

Email: shristisharma1808@gmail.com Mobile: +91-987-1141-197
Github: github.com/shristi-1808

Education

- **Dr. Akhilesh Das Gupta Institute of Technology and Management** New Delhi, India
  Bachelor of Technology - Electrical and Electronics Engineering; CGPA: 8.689 *August 2018 - August 2022*
  Courses: *Fundamental Circuit Theory, Analog and Digital Circuits, VLSI, Microprocessors and Microcontrollers*

Skills Summary

- **Languages**: AVR C, Lua, Python 3, SQL
- **Frameworks**: NumPy, Matplotlib, OpenCV, Sphinx
- **Tools**: Python 3, SQLite 3, Git, Arduino IDE
- **Platforms**: Windows, AVR 8-bit microcontrollers, x86, Raspberry Pi 3 Model B+
- **Soft Skills**: Technical Writing, Public Speaking, Project Management

Experience

- **Central Electronics Limited** On-site
  Training and Internship(Full-time) *Oct 2021 - Nov 2021*
  Assisted in the operation of Digital Axle Counters used High Availability Single Section Digital Axle Counter (HASSDAC) at North Central Railway.

Projects

- **CrazyOS**: Assisted in the development of the technical documentation using restructured text and Sphinx of a single-user, single-tasking, real mode operating system with a shell, line editor, and library consisting of various subroutines in 8086 assembly language. Tech: Python 3, Restructured text, Sphinx (Feb 2022 - May 2022)
- **Remote Controlled Rover**: Implemented a steering mechanism using Ackermann's steering equation for a remote controlled rover. Tech: C, AVR-C, Python 3, Arduino Uno, Raspberry Pi 3 Model B+, SSH (Aug 2021 - Jan 2022)
- **Pioneer 11 Particle Flux Data Visualizer**: Scrapped Goddard Space Flight Center's Space Physics Data Facility to extract particle flux data collected by Pioneer 11, stored it into an SQL database, and visualized using Matplotlib. Tech: Python 3, SQL (Jul 2020 - Aug 2020)
- **Monochromatic Shape Detector**: Developed a program in Python to detect monochromatic shapes in a noisy video. Tech: Python 3, OpenCV, NumPy (Aug 2020 - Oct 2020)
- **Brightness Control using PWM**: Controlled the brightness of an LED strip using PWM signal generated from a 555 timer. Tech: LTSpice XVII (Feb 2020 - Mar 2020)

Honors and Awards

- Scored 125 out of 160 in Duolingo English Test - November 2021
- Presented a report titled *Current State of Drones for Management of Power Systems* - October 2020
- Second Position at Inter College Basketball Tournament - October 2019

Volunteer Experience

- **Member of the Editorial Team, IEEE-ADGITM** New Delhi, India
  Assisted in the development of technical reports for Power and Energy Society (PES), and Women in Engineering (WIE) student chapters. *Jan 2021 - Jan 2022*

- **Tuition from Grade 1 to Grade 10 (ICSE Board)** Uttar Pradesh, India
  Tutored more than 40 students from grade 1 to grade 10 in-person free of cost enabling them to clear their examinations with flying colours. *Aug 2018 - Present*

Sumit Gaur

Email: sumit.gaur1999@gmail.com
Mobile: +91-807-602-1699

Github: github.com/sumit3301
LinkedIn

Education

- **Dr. Akhilesh Das Gupta Institute of Technology and Management** — New Delhi, India
  Bachelor of Technology - Electrical and Electronics Engineering; GPA: 8.189 — *July 2018 - June 2022*
  Courses: *Fundamental Circuit Theory, Analog and Digital Circuits, Microprocessors and Micro controllers, Data Structures*

Skills Summary

- **Languages**: Python, PHP, C++, JavaScript, SQL, Bash, JAVA
- **Frameworks**: JDBC, Tkinter, Plyer ,Spring Boot,Maven,NodeJS,Numpy,Matplotlib,Seaborn
- **Tools**: Docker, GIT,MongoDB MySQL
- **Platforms**: Linux, Web, Windows, Arduino, Raspberry pi,Azure
- **Soft Skills**: Leadership, Event Management, Writing, Public Speaking, Time Management

Experience

- **LetsGrowMore** — Remote
  Data Science Intern (Full-time) — *Aug 2020 - Sep 2019*
  - **Provided comprehensive analysis**: recommend solutions to address complex problems and issues using data
  - **applied advanced analytical methods**: assessed factors impacting security of the countries, created data visualization graphics
- **Gurugram Police** — Remote
  Cybersecurity Intern — *June 2021 - July 2021*
  - **Reviewed violations of computer security**: reviewed procedures and developed mitigation plans
  - **Learned new skills**: and applied them to daily tasks improved efficiency and productivity
- **Careers360** — Remote
  Content Writer — *June 2019 - July 2019*
  - **Guided students for their college selection**: gave career guidance on website of careers 360
  - **Worked within tight deadlines**: Proved successful working within tight deadlines and fast-paced atmosphere.

Projects

- **Weather monitoring system using ESP8266 and DHT11**: In this project the data from the temprature sensor was sent to a dashboard which is sent to the adafruit API and the data is showed in a realtime web based dashboard.
- **Pomodoro Timer**: Made a desktop application called pomodoro timer which starts a 25 minute timer followed by a 5 minute timer in sets for concentration.It is made in Python with using Tkinter which is python desktop GUI library and plyer.

Publications

- **Research Paper:Modern Technology for Evolving Mass Public Transportation in Cities**: Published a research paper and proposed a solution to the problems faced by the modern transportation systems.

Certifications

- **Embedded and IoT Systems(NIELIT)**provides understanding of Embedded Systems, Architecture of Embedded System, IoT Overview and Hardware Platforms, ARM Architecture, Basic C Programming.
- **Building Web Applications in PHP(Courera)** Learned about PHP programming and developed a CRUD App using PHP and SQL database.

Volunteer Experience

- **Conducted Worshop on Embedded systems and Aurduino** — Delhi, India
  Conducted offline technical & soft-skills training impacting over 30 students. — *June 2020*

- **Volunteering in Annual Technorax v6.0 Hackathon conducted by IEEE ADGITM** — Delhi, India
  Organized online hackathon with the help of discord. — *July 2020*