Winter Internship Report: Digital Design Through Arduino

Moole Arya Manohar

B.Tech 3rd Year, Electrical Engineering

Maulana Azad National Institute of Technology (MANIT), Bhopal

Document Information

Field	Details
Student Name	Moole Arya Manohar
Scholar Number	2311301157
Programme	B.Tech, 3rd Year, Electrical Engineering
Institution	Maulana Azad National Institute of Technology (MANIT), Bhopal
Internship Host	Department of Electrical Engineering, IIT Hyderabad
Course Instructor	Dr. G. V. V. Sharma, Associate Professor, EE, IIT Hyderabad
Period	Winter Break (Post 5th Semester)
Date of Report	30 December 2025

Table of Contents

Contents

1. Abstract

This internship report documents the completion of the course "Digital Design Through Arduino" up to Chapter 8 (Embedded C Programming) conducted by Dr. G. V. V. Sharma at the Department of Electrical Engineering, IIT Hyderabad. The course provides a comprehensive introduction to digital design, logic circuits, and embedded systems programming using the Arduino Uno microcontroller, assembly language, and embedded C. The work was undertaken using a free and open-source toolchain consisting of Termux (on Android), Debian Linux, AVR-GCC, PlatformIO, and ArduinoDroid for flashing precompiled hexadecimal files to the Arduino board. This report covers the practical and theoretical aspects of eight chapters spanning installation, digital logic (seven-segment displays, Boolean algebra, Karnaugh maps), sequential circuits (flip-flops, decade counters, finite state machines), assembly language programming, and embedded C development. Particular emphasis is placed on the hands-on troubleshooting and problem-solving approach required to overcome toolchain and environment challenges typical of embedded systems development. In addition to replicating the course experiments, this internship also explored alternative workflows on resource-constrained hardware, highlighting trade-offs between convenience and control in embedded design. The successful completion of all eight chapters demonstrates the viability of professional-grade embedded systems education using only free and open-source tools on Android platforms.

2. Introduction

2.1 Motivation and Course Objective

Digital design and embedded systems form the backbone of modern electronics and IoT applications. Understanding how to implement digital logic at both the hardware and software levels is essential for electrical engineering students because most real-world systems combine microcontrollers, sensors, actuators, and communication interfaces inside a single product. Traditional approaches often rely on proprietary software such as MATLAB and Simulink, along with expensive development boards like those from National Instruments, making the learning curve steep and resource-intensive for students who do not have access to high-end machines or institutional funding.

The "Digital Design Through Arduino" course takes an alternative, inclusive approach: leveraging free and open-source Linux toolchains, low-cost Arduino boards (typically under $25), and the ubiquity of Android smartphones to democratize embedded systems education. This approach ensures that talented students from resource-constrained backgrounds can still gain professional-level skills in digital design. The course philosophy emphasizes learning by doing, as students progress from manual control of digital logic through hardware, to software-driven logic synthesis, and finally to assembly and C-level programming. This philosophy matches the current industry trend where engineers are expected to be comfortable with both low-level details (register manipulation, timing constraints) and higher-level abstractions (object-oriented design, frameworks).

The objective of this internship was to systematically work through the course materials up to Chapter 8 (Embedded C), gaining proficiency in:

- Practical digital logic implementation on breadboards and understanding how theoretical logic gates map onto actual pins and wires in a real circuit.
- Boolean algebra and Karnaugh map-based logic minimization for designing efficient combinational circuits that use fewer ICs and consume less power.
- Sequential circuits including latches, flip-flops, and counters, which are the foundational building blocks for memory, timing, and control units in digital systems.
- Low-level assembly language programming for AVR microcontrollers, focusing on how instructions directly manipulate registers, flags, and memory.
- Embedded C for microcontroller-based applications, using higher-level constructs while still controlling timing and resources with precision.

2.2 Course Context

The course is structured to be platform-agnostic and accessible on resource-constrained devices. All work was completed using:

- Hardware Platform: Arduino Uno (8-bit AVR ATmega328P microcontroller) with 32 KB flash and 2 KB SRAM.
- Development Environment: Termux with Debian GNU/Linux, running on Android without requiring desktop virtualization.
- Build System: PlatformIO and AVR-GCC toolchain for cross-compilation and firmware generation.
- Flashing Method: ArduinoDroid v6.3.1 (via USB-OTG adapter) for uploading pre-compiled hex files.
- Code Editor: Neovim, configured for syntax highlighting and efficient text editing inside Termux.

This approach enabled professional embedded systems development without requiring a traditional laptop or desktop. It demonstrated that serious firmware engineering is achievable on resource-constrained platforms, a valuable skill in contexts ranging from remote field work to developing countries with limited infrastructure. Working within these limitations also provided experience similar to embedded development in actual field conditions, where only minimal tools may be available and scripted, reproducible workflows become critical for reliability and maintainability.

3. Experimental Environment and Development Toolchain

3.1 Hardware Setup

The Arduino Uno is a microcontroller board based on the ATmega328P with the following key specifications:

- Microcontroller: ATmega328P (8-bit AVR RISC processor).
- Operating Voltage: 5 V (digital logic level).
- Analog Reference: 3.3 V option available.
- Clock Frequency: 16 MHz external crystal oscillator.
- Flash Memory: 32 KB (program storage).
- SRAM: 2 KB (runtime variables and stack).
- EEPROM: 1 KB (non-volatile data storage).
- Digital I/O Pins: 14 (6 with PWM capability).
- Analog Input Pins: 6 (A0–A5 with 10-bit ADC).

Supporting components used throughout the internship included a common-cathode seven-segment LED display (with 7 segments plus decimal point), a 7447 BCD-to-seven-segment decoder IC for combinational logic, 7474 dual D flip-flop ICs for sequential logic implementation, 220 Ω resistors for current limiting on LED segments (to protect the display and limit brightness), a solderless breadboard with 830 tie points, assorted jumper wires in multiple colors, and a USB-OTG adapter for connecting the Arduino to an Android phone. Careful attention was paid to current limiting and correct orientation of ICs to avoid accidental damage; each new circuit was first verified on paper and checked against datasheets before wiring on the breadboard.

To keep the workspace organized and reduce errors, each experiment was wired on a dedicated portion of the breadboard, and systematic color coding of jumper wires was adopted wherever possible. Red wires represented VCC (5V), black wires represented GND (ground), and other colors represented signal lines. This practice significantly reduced wiring mistakes and made debugging considerably easier when signals did not behave as expected, since visual inspection could immediately identify suspicious cross-connections.

3.2 Software Toolchain

Termux and Debian Installation

Termux is a Linux terminal emulator for Android that allows running a full Debian environment without requiring root access or virtual machines. Debian was set up in Termux using the official `github.com/gadepall/fwc1` repository, providing:

- Full GNU toolchain support including GCC, Make, Binutils, and standard Unix utilities.
- Package management via `apt`, which simplifies installation of compilers, debuggers, version control, and text editors.
- SSH and Git integration, enabling version control of internship code and remote backup to GitHub if needed.
- Complete file system access under the Termux prefix (`$PREFIX`), while still respecting Android's security sandbox.
- Access to standard development libraries (libc, libm) compatible with the ARM processor.

The installation process involved multiple steps: granting storage permissions to Termux, bootstrapping Debian within Termux, updating package lists, and verifying that critical commands like `gcc --version`, `make --version`, and `git --version` ran correctly. This initial setup served as a gentle introduction to Linux administration tasks, which are essential skills for embedded and systems engineers. Understanding package dependencies, managing library paths, and troubleshooting missing headers became necessary skills during this phase.

AVR-GCC and PlatformIO

PlatformIO is a cross-platform build system and IDE that abstracts much of the complexity of microcontroller development. The setup involved installing AVR-specific tools and configuring PlatformIO for Arduino Uno targets:

```
apt install avra avrdude gcc-avr avrlibc binutils-avr
pip install platformio
pio platform install atmelavr
pio board list
```

The build process generates `.hex` files in `.pio/build/uno/firmware.hex`, which are then moved to the Android Downloads folder for flashing from ArduinoDroid. PlatformIO's configuration file (`platformio.ini`) specifies the board type, CPU frequency,

and upload protocol, so switching projects or targets becomes straightforward. Through repeated compilations across different projects, it became clear how the toolchain functions: C or assembly sources are compiled to object files, then linked into an ELF binary with symbols and debugging information, and finally converted into a hex representation suitable for the microcontroller's flash memory.

Understanding this pipeline is important because compilation errors often trace back to missing include directories, undefined symbol references, or linker script mismatches. The ability to read and interpret compiler error messages became a critical debugging skill.

ArduinoDroid and Hex File Flashing

The Play Store version of ArduinoDroid (v7.x and above) no longer supports uploading locally compiled hex files, as the development shifted toward a cloud-based IDE. To restore this functionality, version 6.3.1 was downloaded from APKPure (a third-party APK repository) and pinned to avoid automatic updates through the Play Store. The workflow became:

1. Generate hex files using PlatformIO in Termux (output: `.pio/build/uno/firmware.hex`).
2. Move the hex file to the `~/Downloads/` folder using CX File Explorer or command line.
3. In ArduinoDroid: navigate to **Actions → Upload → Upload Precompiled**.
4. Select the hex file from the Downloads folder via the file picker.
5. Connect the Arduino Uno via USB-OTG cable and initiate the upload.

Several trial uploads were performed to understand how serial ports are detected on Android, how upload progress is reported, and how failures manifest (for example, when the cable is loose, the board is in the wrong bootloader mode, or the hex file is corrupted). These repeated tests helped build confidence in the flashing process and reduced hesitation when modifying low-level firmware. The experience also highlighted the importance of checking device connections before attempting uploads, similar to hardware debugging workflows in professional environments.

3.3 Challenges and Troubleshooting

Key environment challenges included an Ubuntu dual-boot failure on the personal laptop (despite 180 GB of free disk space, the partition manager could not allocate new partitions), hidden hex files inside the `.pio` directory that were not visible to default Android file pickers, and ArduinoDroid version incompatibility due to Play Store forced

auto-updates. These were resolved by adopting an Android-only workflow, using the CX File Explorer app to access hidden directories, and manually installing ArduinoDroid v6.3.1 from APKPure, respectively. Each issue forced a deeper understanding of how operating systems handle storage allocation, file system hierarchies, and application update policies, turning what initially seemed like obstacles into valuable learning experiences that expanded knowledge beyond the core embedded systems curriculum.

4. Chapter-Wise Work Completion

4.1 Chapter 1: Installation

The installation chapter covered Termux and Debian setup, PlatformIO and AVR toolchain installation, and ArduinoDroid configuration. Successful completion was verified by uploading a blink program to the Arduino Uno, confirming end-to-end toolchain correctness and introducing cross-compilation workflows. The LED on pin 13 blinked at regular 1-second intervals, providing visual confirmation that the compilation, linking, hex generation, and flashing all succeeded.

During this stage, particular attention was paid to understanding each component in the toolchain rather than treating the build system as a black box. For example, the distinction between `pio run` (which builds the firmware) and `pio run -t upload` (which performs both build and upload) was noted, along with how `avrdude` is invoked under the hood to communicate with the bootloader. This knowledge proved helpful later when debugging build failures (such as missing include files) or upload errors (such as connection timeouts). Understanding the build output, including warnings about unused variables or unreachable code, became important for writing efficient embedded code.

4.2 Chapter 2: Seven-Segment Display

This chapter introduced seven-segment display control using both direct pin manipulation and Arduino C code. The display was wired to Arduino pins 2–8, with the common cathode connected to ground through a current-limiting resistor. Experiments included manual segment activation by grounding individual pins, truth-table construction for displaying digits 0–9, and understanding active-low logic (where a segment illuminates when the pin is pulled LOW relative to the common cathode).

Arduino C programs used lookup tables to store the segment patterns for each digit, with each entry encoding which segments should be active. In addition to simple counting sequences, patterns such as walking segments (one lit at a time, moving around the display), alternating even and odd digits, and displaying custom shapes were implemented. These exercises strengthened intuition about how binary values on output pins translate into physical illumination patterns on the display. They also highlighted the importance of current-limiting resistors (typically 220 Ω per segment) to prevent LED burnout and the difference between common-cathode and common-anode configurations, which invert the logic required in software. Understanding these hardware-level details is critical for designing robust display interfaces.

4.3 Chapter 3: 7447 BCD-to-Seven-Segment Decoder

The 7447 IC was used to offload combinational logic from the microcontroller, demonstrating the principle of using specialized hardware to reduce software complexity. Arduino pins D2–D5 drove the BCD (Binary Coded Decimal) inputs, and the 7447 outputs were directly connected to the seven-segment display. Boolean logic expressions for each segment output (a through g) were implemented in software and tested against the datasheet truth table for all 16 possible input combinations. A custom incrementing decoder (outputs the next digit for inputs 0–9 with wraparound $9 \rightarrow 0$) was also designed as a prerequisite for later Karnaugh map exercises.

To deepen understanding, timing diagrams were sketched to visualize how changes on the BCD inputs propagate through the 7447 combinational logic and eventually update the seven-segment display. Propagation delay (typically 20–40 nanoseconds for the 7447) and setup/hold times for valid inputs were considered. Edge cases such as invalid BCD inputs (10–15, which do not represent decimal digits) were examined, and different strategies for handling them were discussed conceptually: blanking the display (driving all segments OFF), repeating the previous valid digit, or showing a special pattern such as a dash or question mark. These considerations highlight how hardware designers must anticipate edge cases and plan graceful degradation.

4.4 Chapter 4: Karnaugh Maps

Karnaugh maps were applied to minimize the incrementing decoder Boolean expressions and the 7447 segment equations, moving from raw truth tables to optimized circuits. By exploiting don't-care states for unused BCD inputs (10–15), gate count and expression complexity were significantly reduced. For example, the expression for output D (which generates the most-significant bit of the next count) was minimized from a complex sum-of-products form to a much simpler expression with fewer literals, directly translating into savings in gate count, power consumption, and propagation delay on an actual circuit.

Several variations of the same truth table were intentionally minimized using different grouping strategies in the Karnaugh map to observe how alternative but logically equivalent expressions can exist. This exercise emphasized that minimization is not always unique and that designers must balance multiple factors such as gate fan-in (the number of inputs to a gate), wiring simplicity, and ease of debugging in practice. The process of moving from unsimplified sum-of-products to compact expressions also reinforced algebraic manipulation skills that are useful across many branches of engineering. Understanding these trade-offs between theoretical optimality and practical implementation is essential for real-world digital design.

4.5 Chapter 5: 7474 D Flip-Flop and Decade Counter

Sequential logic was introduced using the 7474 dual D flip-flop, the fundamental building block for state storage in digital systems. A decade counter was built by combining three layers:

- Combinational logic (incrementing decoder) to compute the next state based on the current state.
- 7474 flip-flops to store the current state and capture the next state on clock edges.
- An Arduino-generated clock signal on pin D13 (via `digitalWrite` and `delay` calls) to drive state transitions at regular intervals.

Outputs were displayed through the 7447 decoder and seven-segment display, cycling cleanly from 0 to 9 and back to 0, illustrating synchronous state machine behavior. Setup and hold time constraints were carefully observed: the next-state inputs to the 7474 must be stable for a specified time before and after the clock edge to ensure correct operation.

Beyond the basic counter, experiments were conducted with different clock speeds (from 10 milliseconds per cycle down to 100 microseconds per cycle) to observe how human perception interacts with electronics. At very low frequencies the digit changes are easily visible, whereas at higher frequencies the display appears blurred or steady because the human eye cannot track rapid transitions (the eye's refresh rate is approximately 60 Hz). This simple experiment connects digital design to concepts from signal processing and human factors, demonstrating why visual indicators are usually updated at comfortable refresh rates. The decade counter capstone project unified all prior learning into a single working system that embodied Boolean logic, K-map minimization, sequential circuits, and real-time control.

4.6 Chapter 6: Finite State Machine Framework

The decade counter was formalized as a ten-state finite state machine with states s_0 to s_9 and well-defined transitions: $s_i \to s_{i+1}$ on every clock pulse, with $s_9 \to s_0$. While this chapter focused more on conceptual understanding than on implementing new hardware, it cemented the mapping between state diagrams (graphical representations), transition tables (tabular representations), combinational logic (computing next states), and flip-flop implementations (storing current states).

To connect hardware FSMs with software design patterns, pseudo-code state machines were written that mirror the hardware counter, using variables to represent state and switch-case constructs to describe transitions. This comparison showed that, at an abstract level, both hardware and software are solving the same state-transition problem,

even though their implementation details differ significantly. This recognition is powerful: understanding FSMs deeply at the hardware level transfers directly to writing robust state-based software, as seen in everything from protocol handlers to game engines.

4.7 Chapter 7: Assembly Language Programming

AVR assembly programming was explored with emphasis on direct register manipulation, I/O configuration using DDRx and PORTx registers, and control flow using branches and loops. Completed exercises included:

- Configuring pins as inputs or outputs using the Data Direction Register (DDRB, DDRD, etc.).
- Toggling LEDs using SBI (Set Bit in I/O) and CBI (Clear Bit in I/O) instructions.
- Driving the seven-segment display with bit patterns loaded into registers and output to ports.
- Implementing timer-based delays using the AVR timer-counter module (TCCR0B, TIFR0) for more precise timing than crude cycle counting.
- Building a memory-based decade counter by iterating through SRAM using the X register (a 16-bit register pair) as a pointer.

Detailed comments were added to each assembly program to annotate which registers were temporary storage, which were used for loop counters, and how many clock cycles each section consumed. This practice made the assembly code easier to revisit later and provided insight into how fine-grained control over timing can be achieved when needed for time-critical operations. The experience also highlighted the discipline required to manage limited registers (only 32 general-purpose registers on the ATmega328P) and avoid accidental overwrites, a skill that carries over to optimizing any resource-constrained system.

4.8 Chapter 8: Embedded C Programming

Embedded C programs were developed using AVR-GCC, balancing readability with hardware control. Key completed tasks included:

- A basic blink program using direct PORTB manipulation and `_delay_ms` from the `util/delay.h` library.
- Seven-segment display control for digits 0–9 via lookup tables stored in flash memory (const arrays).

- Initial exploration of LCD 16×2 interfacing via parallel I/O (reviewed but not fully completed due to the complexity of LCD timing protocols, which require microsecond-level precision).
- Integrating C with assembly by calling assembly delay routines from C code, demonstrating the calling convention for AVR.

To bridge the gap between C and assembly understanding, compiled binaries were inspected using `avr-objdump -d firmware.elf` to see the assembly generated by the compiler for simple C functions. Observing this mapping clarified how high-level constructs such as for loops and if statements are converted into branch instructions (conditional jumps), memory operations, and function calls. This insight helped in writing C code that remains efficient on a small 8-bit microcontroller where every byte of flash and SRAM matters. Techniques such as using `const` for lookup tables (which forces storage in flash rather than SRAM), avoiding unnecessary function calls in tight loops, and using `inline` for small helper functions became important practices.

5. Learning Outcomes and Technical Competencies

5.1 Digital Design Fundamentals

The internship strengthened foundational knowledge in number system conversions (decimal, binary, hexadecimal), truth table construction and verification, Boolean algebra simplification using axioms and theorems, Karnaugh map minimization for logic functions, the distinction between combinational and sequential circuits, and effective use of don't-care conditions in design optimization. These topics are often taught abstractly in theory courses, but implementing them on actual hardware gave much more concrete understanding of why minimization matters (cost savings, power reduction) and how errors in truth tables immediately translate into incorrect physical behavior (wrong outputs, display glitches). The hands-on experience reinforced that digital design is not purely mathematical but has real economic and physical consequences.

5.2 Practical Hardware Skills

Hands-on work built confidence in solderless breadboard wiring techniques, IC pinout interpretation from datasheets, visual debugging using LEDs and displays as output indicators, translating schematic diagrams to physical layouts, and safe handling of TTL-level digital signals (5V logic). Common pitfalls encountered and corrected during experiments included floating inputs (unconnected pins that pick up noise), missing ground connections (leading to undefined behavior), accidentally mirrored IC orientation (causing power supply reversal and instant burnout), and incorrect current-limiting resistor values. Discovering and fixing each mistake reinforced the habit of double-checking connections and verifying continuity before applying power, a discipline that transfers to larger projects.

5.3 Embedded Systems Toolchain

Competence was gained in cross-compilation targeting the AVR architecture from a Linux development environment, understanding the multi-stage compilation and linking process, generating and flashing hex files to the microcontroller, handling API changes between tool versions (as exemplified by ArduinoDroid versioning), and organizing embedded projects using PlatformIO's project structure. The ability to read and interpret compilation warnings (unused variables, unreachable code), understand linker errors (undefined symbols, section mismatches), and adjust include paths and library dependencies will be directly useful in larger embedded and systems projects. The experience with managing tool versions and avoiding breaking changes is especially relevant in production embedded systems where updating tools can inadvertently break existing firmware.

5.4 Low-Level Programming

The internship developed low-level programming skills in both assembly language and embedded C, including register-level I/O through memory-mapped ports, bitwise operations for manipulating individual bits, timing considerations for meeting hardware requirements, and pragmatic decisions about when to remain at the assembly level (for timing-critical or space-constrained code) versus when to rely on C abstractions (for complex logic and maintainability). For example, simple delay loops and bit-toggling were good candidates for assembly due to predictable timing, whereas state machine logic and lookup-table management were more readable and maintainable in C. Understanding these trade-offs is essential for writing robust embedded systems.

5.5 Problem-Solving and Debugging

Significant emphasis throughout the internship fell on systematic debugging methodologies, researching issues through datasheets and online resources, isolating root causes through systematic testing, designing practical workarounds under hardware and software constraints, and validating each subsystem independently before integrating it into larger systems. The experience of working without full-featured desktop debuggers (lacking breakpoints and single-stepping) encouraged careful reasoning about expected behavior, writing assertions and sanity checks in code, and using LEDs or UART output as observable indicators of internal state. These strategies, born from necessity, are indispensable for embedded systems where traditional debugging tools are often unavailable or impractical.

6. Connection to Broader Embedded Systems Ecosystem

Although this internship centered on the Arduino Uno and ATmega328P microcontroller, the same conceptual frameworks extend naturally to more advanced platforms such as the ESP32 (WiFi and Bluetooth enabled), STM32 "Blue Pill" (ARM Cortex-M3 processor, industrial applications), the Vaman FPGA+ARM heterogeneous board (custom hardware logic in Verilog, real-time constraints), and Raspberry Pi Pico (RP2040 ARM processor for education and maker projects). The philosophy of using free software and performing cross-compilation from lightweight environments (including Android) remains applicable across these platforms, and the skills developed with AVR-GCC translate to ARM-GCC with relatively straightforward adjustments (different register names, different instruction set, but the same build pipeline).

From a career and technical perspective, familiarity with this broader ecosystem opens doors to specialized domains like IoT systems, industrial automation and control, edge computing and machine learning inference on devices, and real-time embedded systems. For instance, the decade counter logic implemented on the Arduino Uno could later be integrated into a networked ESP32 system where counts are logged to a cloud server, or translated into hardware Verilog on an FPGA where timing guarantees are even more stringent. The mental model of moving flexibly between abstraction layers—from low-level bit manipulation to high-level software design—is directly reusable and highly valued in professional embedded systems roles.

7. Conclusion

The winter internship on "Digital Design Through Arduino" provided a coherent and well-structured journey from basic digital logic through embedded C, tightly coupling theory with hands-on practice. Working entirely from an Android-based development environment demonstrated convincingly that serious, production-quality embedded systems work is feasible without conventional PC infrastructure. The decade counter capstone project synthesized and unified concepts across Boolean logic, K-maps, sequential circuits, finite state machines, assembly language, and C, providing a multi-layered view of how the same logical behavior can be represented and implemented at different abstraction levels.

Beyond core technical skills, the internship cultivated lasting professional habits: systematic experimentation and hypothesis testing, careful documentation of procedures and findings, resilience and creativity when facing toolchain or hardware obstacles, and pragmatic decision-making when balancing multiple engineering trade-offs. These habits will prove advantageous not only in future embedded projects but also in adjacent areas such as AR-powered hardware interfaces, IoT dashboards and real-time visualization, and full-stack systems that combine firmware with web or mobile front-ends. With the solid foundations laid by this internship, the trainee is well-positioned to pursue advanced topics in microcontroller-based real-time systems, FPGA-based digital design in Verilog or VHDL, distributed embedded systems and mesh networks, and high-level applications that integrate custom firmware with cloud backends. The approach of combining free and open-source tools with accessible hardware has proven to be an effective, inclusive, and scalable model for embedded systems education.

References

1. G. V. V. Sharma, *Digital Design Through Arduino*, Department of Electrical Engineering, IIT Hyderabad, 2025. Available at: `https://github.com/gadepall/digital-design-28-01-25`.

2. G. V. V. Sharma, Course resources and related repositories. Available at: `https://github.com/gadepall`.

Report Prepared By: Moole Arya Manohar
Scholar Number: 2311301157
Programme: B.Tech 3rd Year, Electrical Engineering, MANIT Bhopal
Date: 30 December 2025

Signature: _____

Faculty Advisor: Dr. G. V. V. Sharma, Associate Professor, EE, IIT Hyderabad