Embedded Systems Engineer Assignment

1. Introduction and Objective ;

This report details the successful completion of a three-part embedded systems assignment. The project demonstrates core skills in microcontroller programming, analog circuit design, and the practical application of electrical engineering principles. The solutions were verified through simulation using industry-standard software, including Wokwi and Proteus.

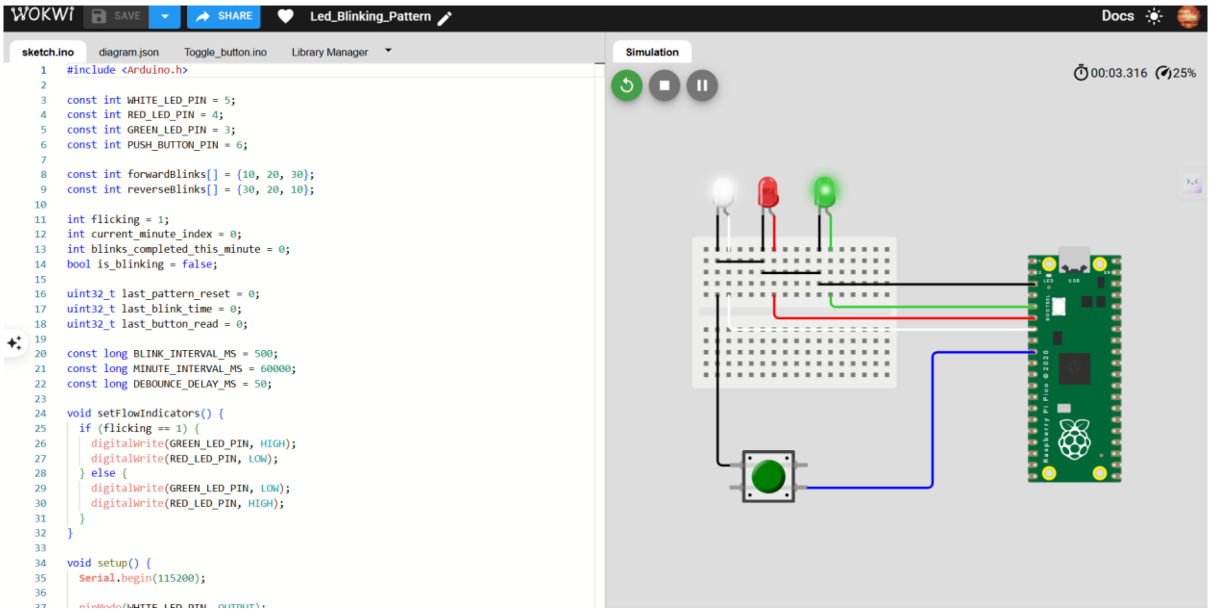
2. Q1: Raspberry Pi Pico LED Pattern Control

2.1 Objective

The objective of this project was to program a Raspberry Pi Pico to execute a specific LED blinking pattern based on time and a push-button toggle. The project was designed with non-blocking logic to ensure continuous responsiveness.

2.2 Hardware and Circuit Diagram

The project uses a Raspberry Pi Pico microcontroller with three LEDs (white, red, and green) and a push button. The circuit diagram below illustrates the correct component connections.



```cpp
#include <Arduino.h>

const int WHITE_LED_PIN = 5;
const int RED_LED_PIN = 4;
const int GREEN_LED_PIN = 3;
const int PUSH_BUTTON_PIN = 6;

const int forwardBlinks[] = {10, 20, 30};
const int reverseBlinks[] = {30, 20, 10};

int flicking = 1;
int current_minute_index = 0;
int blinks_completed_this_minute = 0;
bool is_blinking = false;

uint32_t last_pattern_reset = 0;
uint32_t last_blink_time = 0;
uint32_t last_button_read = 0;

const long BLINK_INTERVAL_MS = 500;
const long MINUTE_INTERVAL_MS = 60000;
const long DEBOUNCE_DELAY_MS = 50;

void setFlowIndicators() {
  if (flicking == 1) {
    digitalWrite(GREEN_LED_PIN, HIGH);
    digitalWrite(RED_LED_PIN, LOW);
  } else {
    digitalWrite(GREEN_LED_PIN, LOW);
    digitalWrite(RED_LED_PIN, HIGH);
  }
}

void setup() {
  Serial.begin(115200);

```

2.3 Software and Logic

The program is written in C++ for the Arduino framework. It uses a non-blocking timer with millis() to allow for simultaneous blinking control and button-press detection. The flicking variable acts as a state flag to switch between the two blinking patterns.

```
#include <Arduino.h>
```

```
// Define pin numbers for all components
const int WHITE_LED_PIN = 5;
const int RED_LED_PIN = 4;
const int GREEN_LED_PIN = 3;
const int PUSH_BUTTON_PIN = 6;

// Define blinking patterns for forward and reverse flows
const int forwardBlinks[] = {10, 20, 30};
const int reverseBlinks[] = {30, 20, 10};

// State variables for the program's logic
// flicking = 1 for forward flow, 0 for reverse flow
int flicking = 1;
int current_minute_index = 0;
int blinks_completed_this_minute = 0;
bool is_blinking = false;

// Variables for non-blocking timing using millis()
uint32_t last_pattern_reset = 0;
uint32_t last_blink_time = 0;
uint32_t last_button_read = 0;

// Constants for timing control
const long BLINK_INTERVAL_MS = 500; // Time for one state (ON or OFF)
const long MINUTE_INTERVAL_MS = 60000; // One minute in milliseconds
const long DEBOUNCE_DELAY_MS = 50; // Delay to prevent multiple button reads

// Function to set the indicator LEDs based on the 'flicking' state
void setFlowIndicators() {
  if (flicking == 1) { // Forward flow
    digitalWrite(GREEN_LED_PIN, HIGH);
    digitalWrite(RED_LED_PIN, LOW);
  } else { // Reverse flow
    digitalWrite(GREEN_LED_PIN, LOW);
    digitalWrite(RED_LED_PIN, HIGH);
  }
}

// Setup function: runs once on power-up
void setup() {
  Serial.begin(115200);

  // Configure all LED pins as outputs
  pinMode(WHITE_LED_PIN, OUTPUT);
  pinMode(RED_LED_PIN, OUTPUT);
  pinMode(GREEN_LED_PIN, OUTPUT);

  // Configure the button pin as an input with an internal pull-up resistor
  pinMode(PUSH_BUTTON_PIN, INPUT_PULLUP);

  // Initial blink of all LEDs to confirm startup
  digitalWrite(WHITE_LED_PIN, HIGH);
  digitalWrite(RED_LED_PIN, HIGH);
  digitalWrite(GREEN_LED_PIN, HIGH);
  delay(1000);
  digitalWrite(WHITE_LED_PIN, LOW);
  digitalWrite(RED_LED_PIN, LOW);
  digitalWrite(GREEN_LED_PIN, LOW);

  // Set initial state and start the main timer
  setFlowIndicators();
  last_pattern_reset = millis();
  is_blinking = true;
```

```
}

// Loop function: runs repeatedly after setup
void loop() {
  // Read button state with a debounce check
  bool buttonState = digitalRead(PUSH_BUTTON_PIN);
  if (buttonState == LOW && (millis() - last_button_read) > DEBOUNCE_DELAY_MS) {
    // Toggle the 'flicking' state (1 to 0 or 0 to 1)
    flicking = (flicking == 1) ? 0 : 1;
    setFlowIndicators();

    // Reset all timers and counters to start the new pattern
    current_minute_index = 0;
    blinks_completed_this_minute = 0;
    last_pattern_reset = millis();
    is_blinking = true;

    Serial.println(flicking == 1 ? "Switched to FORWARD flow." : "Switched to REVERSE
flow.");
    last_button_read = millis();
  }

  // Check if it's time to start a new minute's pattern
  uint32_t elapsedTime = millis() - last_pattern_reset;
  if (elapsedTime >= (current_minute_index + 1) * MINUTE_INTERVAL_MS) {
    current_minute_index++;
    // Check if the 3-minute cycle is complete
    if (current_minute_index >= 3) {
      current_minute_index = 0;
      last_pattern_reset = millis();
    }
    // Reset blink counter for the new minute
    blinks_completed_this_minute = 0;
    is_blinking = true;
  }

  // Blinking logic for the white LED
  if (is_blinking) {
    // Select the correct blinking pattern based on 'flicking' state
    const int* currentPattern = (flicking == 1) ? forwardBlinks : reverseBlinks;
    int targetBlinks = currentPattern[current_minute_index];

    // Check if we have completed all blinks for the current minute
    if (blinks_completed_this_minute < targetBlinks * 2) {
      if (millis() - last_blink_time >= BLINK_INTERVAL_MS) {
        // Toggle LED state (ON/OFF)
        digitalWrite(WHITE_LED_PIN, !digitalRead(WHITE_LED_PIN));
        last_blink_time = millis();
        blinks_completed_this_minute++;
      }
    } else {
      // Ensure LED is OFF after the pattern completes
      digitalWrite(WHITE_LED_PIN, LOW);
      is_blinking = false;
    }
  }
}
```

2.4 Simulation and Results

The project was simulated in Wokwi to demonstrate its functionality. The simulation confirmed that the code works as expected, executing the correct blinking patterns and responding immediately to the button presses.

Wokwi Simulation

3. Q2: PWM Signal Condition

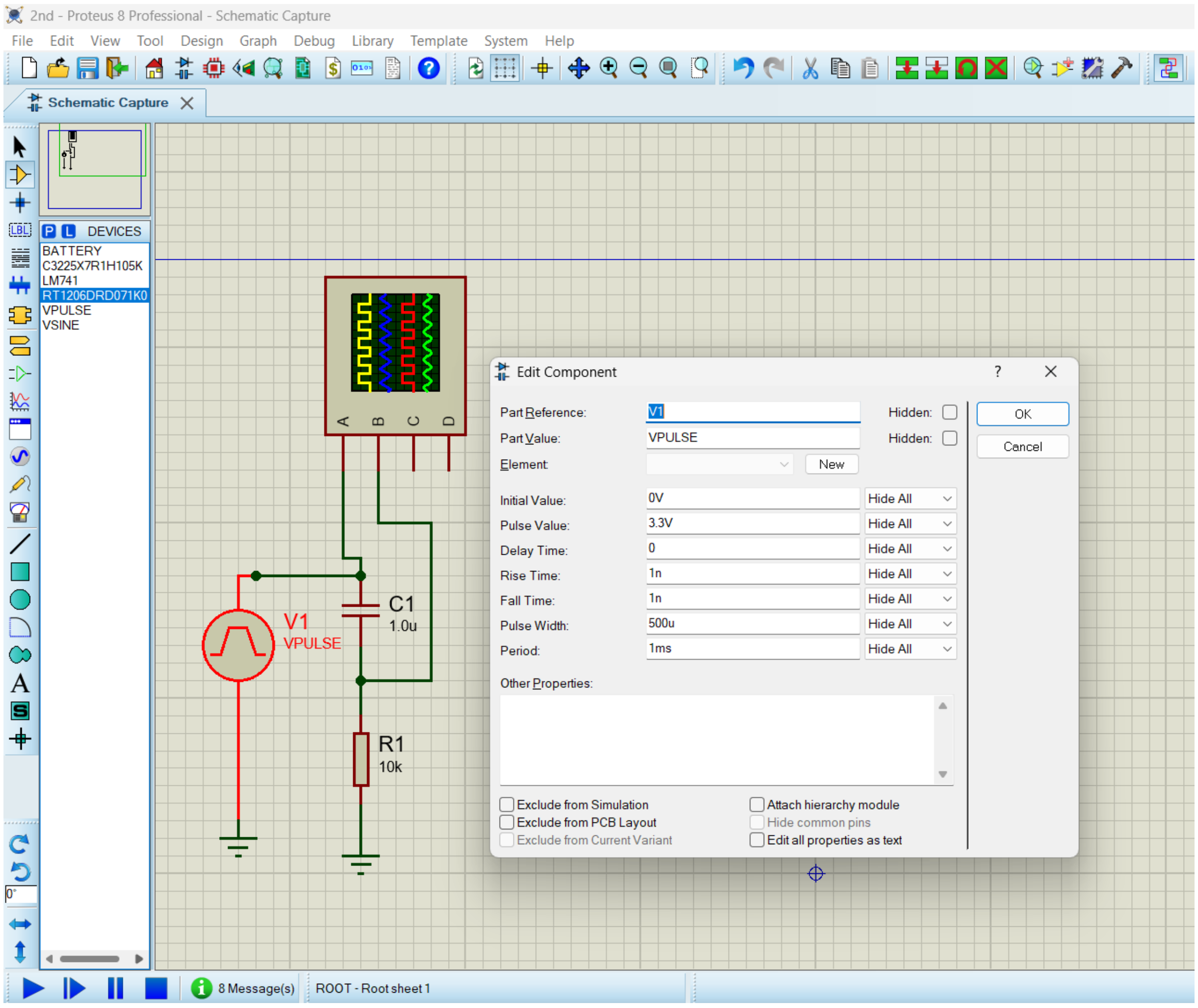
3.1 Objective

The objective was to design two analog circuits in Proteus. The first circuit shifts a 0V to 3.3V PWM signal to a -1.65V to +1.65V range, and the second shifts the signal back to the original voltage level.

3.2 Circuit Design and Simulation

The simulation was built with two interconnected circuits.

- Circuit 1: Down-Shifting Circuit (RC High-Pass Filter)
    - This circuit uses a capacitor to block the DC component and a resistor to ground to shift the signal's mean to 0V. The output of this circuit serves as the input for the second circuit.



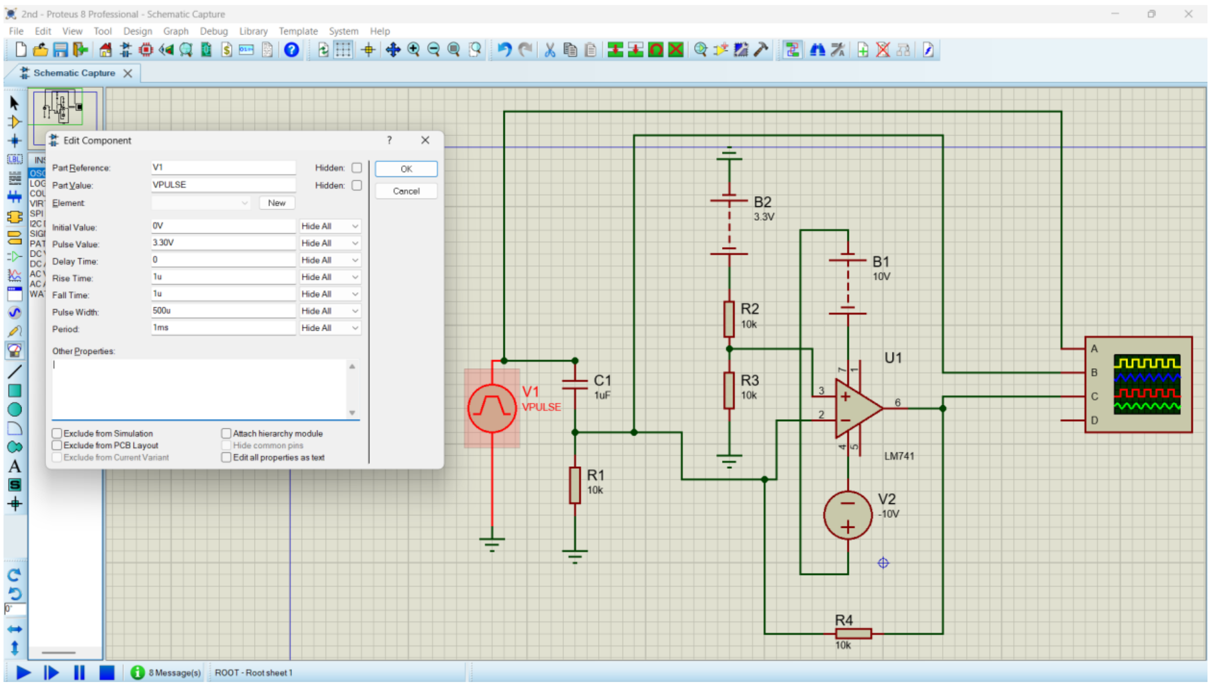
Edit Component

Part Reference:	V1	Hidden: ☐	OK
Part Value:	VPULSE	Hidden: ☐	Cancel
Element:		New	
Initial Value:	0V	Hide All	
Pulse Value:	3.3V	Hide All	
Delay Time:	0	Hide All	
Rise Time:	1n	Hide All	
Fall Time:	1n	Hide All	
Pulse Width:	500u	Hide All	
Period:	1ms	Hide All	

Other Properties:

☐ Exclude from Simulation
☐ Exclude from PCB Layout
☐ Exclude from Current Variant
☐ Attach hierarchy module
☐ Hide common pins
☐ Edit all properties as text

- Circuit 2: Up-Shifting Circuit (Op-Amp Level Shifter)
    - This circuit uses an LM741 operational amplifier to add a 1.65V DC offset to the signal, restoring it to the original voltage range.


2nd - Proteus 8 Professional - Schematic Capture

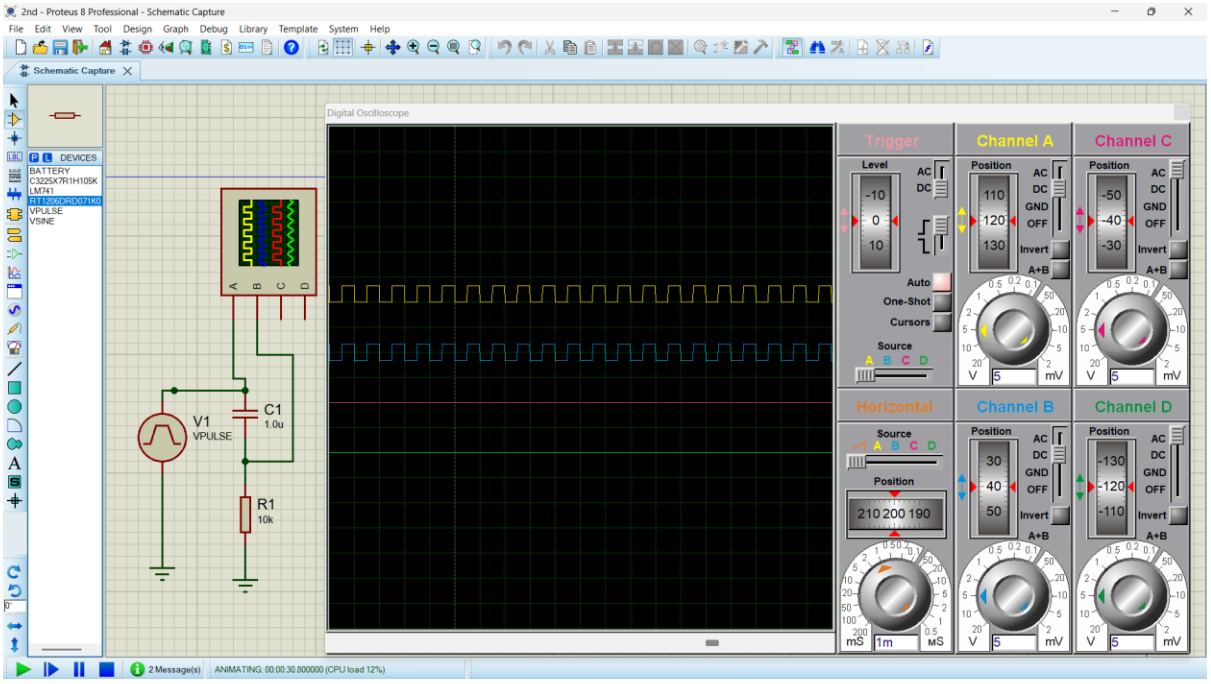
Edit Component

Part Reference: V1
Part Value: VPULSE
Element:
Initial Value: 0V
Pulse Value: 3.30V
Delay Time: 0
Rise Time: 1u
Fall Time: 1u
Pulse Width: 500u
Period: 1ms
Other Properties:

3.3 Simulation Analysis

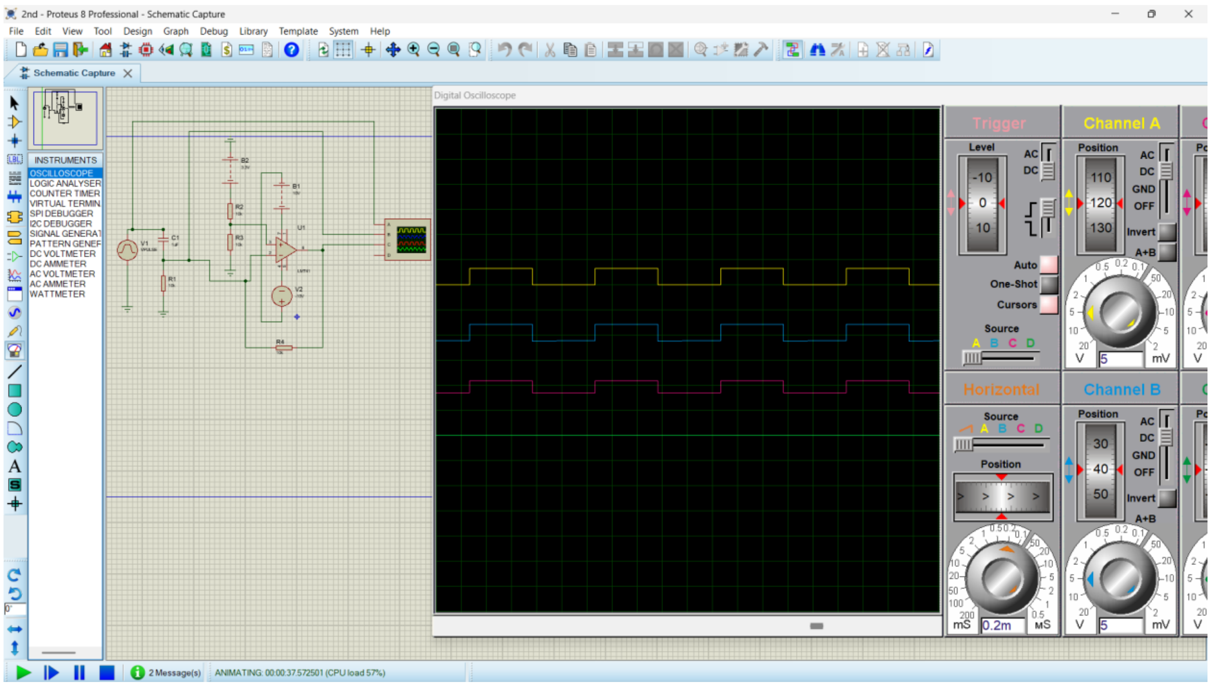
The final output was analyzed on a virtual oscilloscope to confirm the level shifting.

- **Input Waveform**: The input was a square wave correctly ranging from 0V to 3.3V.
- **Down-Shifted Waveform**: The signal at the output of the first circuit was correctly shifted to a range of -1.65V to +1.65V.


2nd - Proteus 8 Professional - Schematic Capture

ANIMATING: 00:00:30.800000 (CPU load 12%)

- **Final Output Waveform**: The output of the op-amp circuit was a perfect square wave ranging from 0V to 3.3V, confirming a successful level shift.



4. Q3: Battery Pack Topologies

4.1 Objective

The objective was to explain the pros and cons of two battery pack topologies and verify their characteristics through simulation.

4.2 Topology 1: Series-Parallel

This topology connects cells in series to increase voltage and in parallel to increase capacity.

- **Pros**: Allows for both high voltage and high capacity. Provides redundancy if a single cell fails.
- **Cons**: Complex to design and requires a sophisticated Battery Management System (BMS).

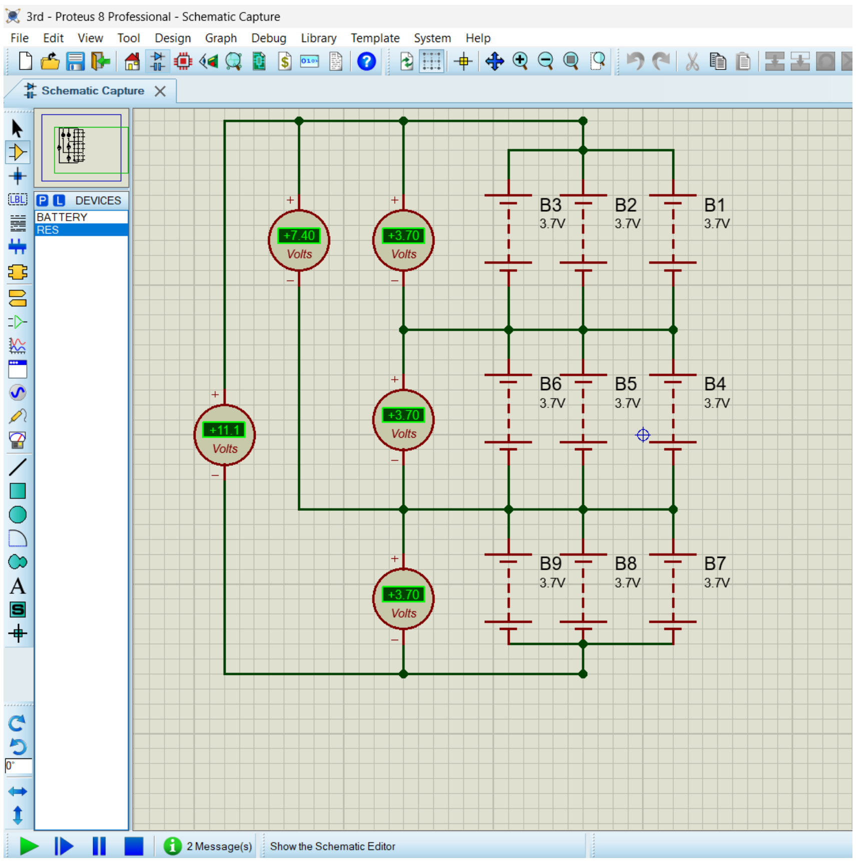
4.3 Topology 2: Parallel

This topology connects all positive terminals and all negative terminals together.

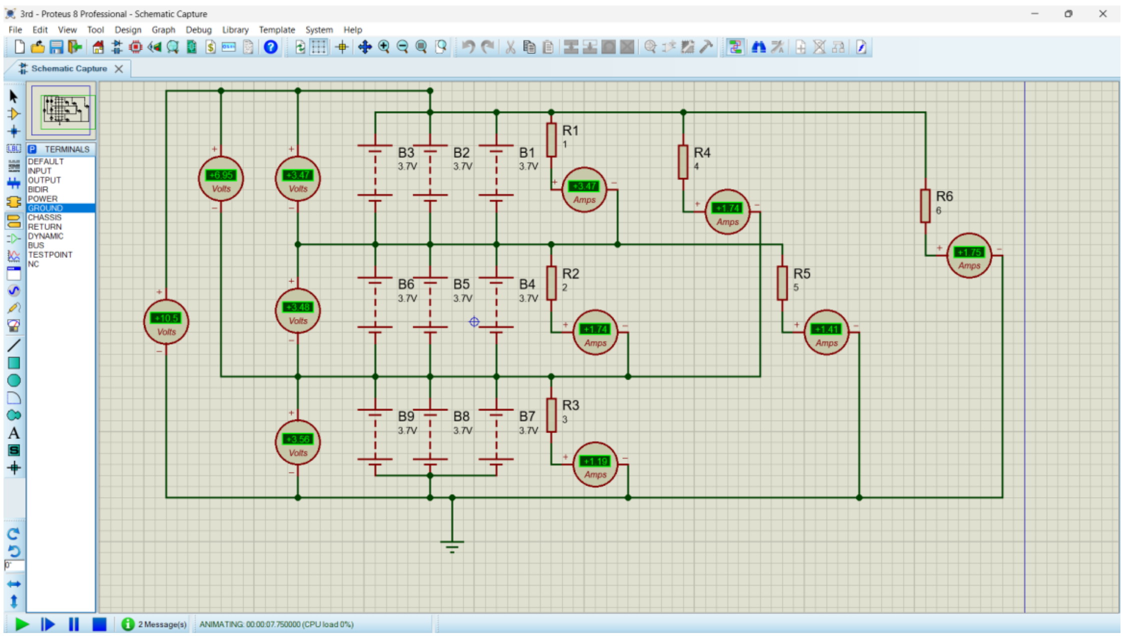
- **Pros**: Simple to wire and provides a high current capacity.
- **Cons**: The total voltage is limited to that of a single cell, and a cell fault can be a significant safety risk.

4.4 Simulation Results

Simulations for both topologies were conducted in Proteus to verify their total voltage and current.

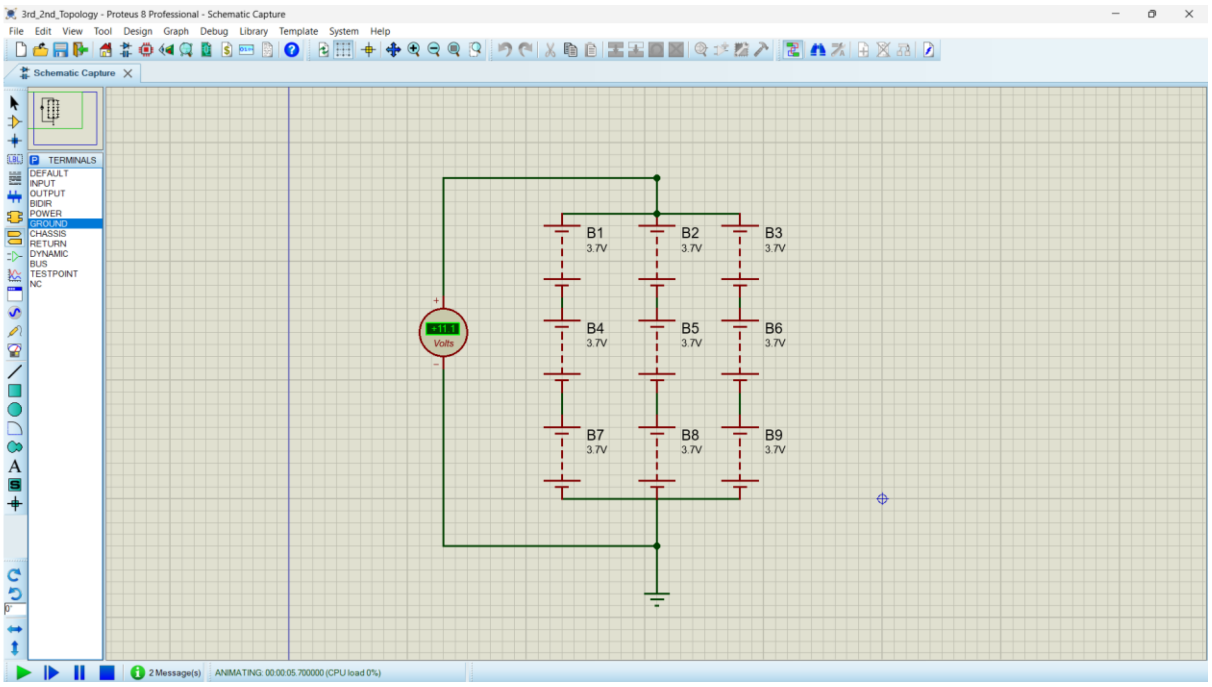


Topology 1 Series_Parallel_Without Load

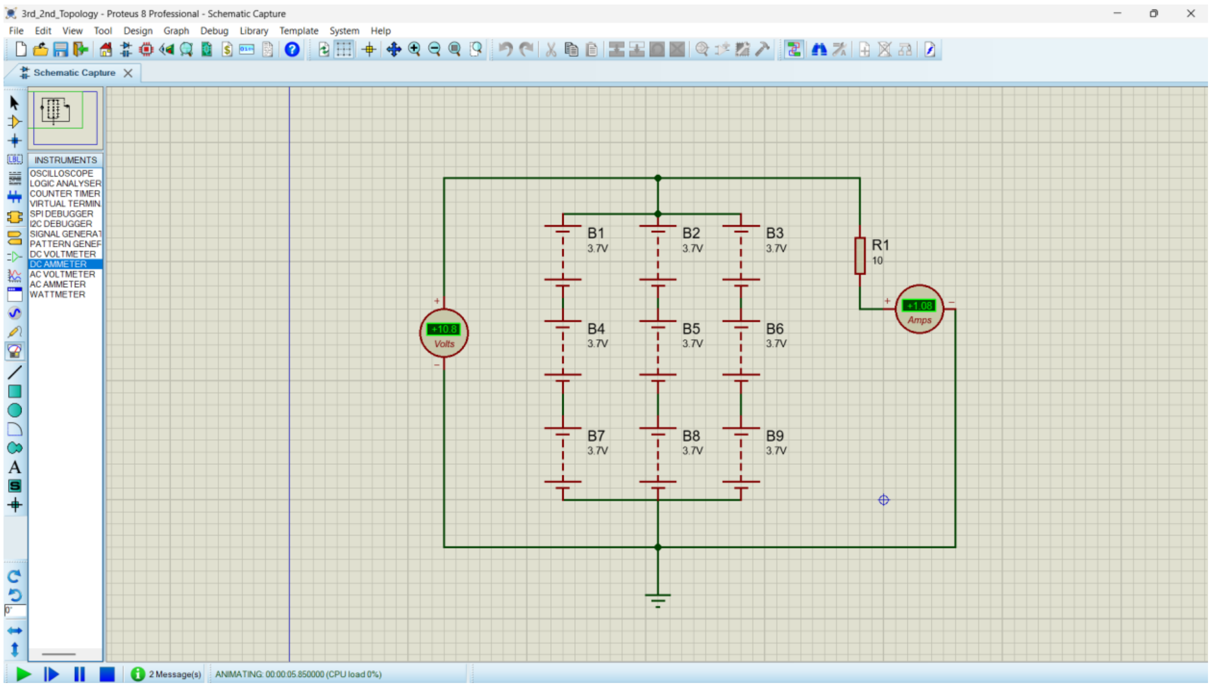


Topology 1 Series_Parallel_With Load

- **Series-Parallel Simulation**: The simulation of a 3-series, 3-parallel (3S3P) pack correctly showed a total voltage of **11.1V**.



Topology 2 Parallel_without Load



Q3.Topology 2 Parallel_with Load

- **Parallel Simulation**: The simulation of a parallel pack correctly showed a total voltage equal to a single cell's voltage while allowing for a high current draw.

From:

Shaik Althaf

Shaikalthaf1768@gmail.com