Semester Long Internship Report

On

Digital and Mixed Signal Circuits in eSim

Submitted by

Mir Mousam Ali
B.Tech(Electronics and Communication)
Aliah University Kolkata

Under the guidance of

Prof.Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

June 27, 2024

Acknowledgment

I am deeply grateful to the FOSSEE team for providing me a opportunity to embark on a semester-long internship journey. This experience has been a pivotal chapter in my academic and professional development, and I am thankful to all those who have played a role in making it enriching and fulfilling.

I extend my heartfelt appreciation to my mentors, Mr. Sumanto Kar, whose mentorship and guidance have been invaluable throughout my internship. Their unwavering support, insightful feedback, and encouragement have empowered me to navigate challenges, explore new horizons, and grow both personally and professionally. I would also like to express my gratitude to the entire FOSSEE Team at IIT Bombay for their warm welcome and collaborative spirit. Their collective expertise, diverse perspectives, and camaraderie have created an inspiring environment conducive to learning and growth. I am indebted to each member of the team for their willingness to share knowledge, offer guidance, and foster a culture of excellence.

Each task and challenge has provided me with valuable learning experiences and has equipped me with skills and insights that will undoubtedly shape my future endeavors.

Furthermore, I extend my gratitude to my academic institution for facilitating this internship opportunity and for their continued support and encouragement throughout the semester.

In conclusion, I am grateful for the privilege to have been a part of FOSSEE team as a Intern and to have had the chance to collaborate with talented professionals in a dynamic and innovative environment. This internship has been a transformative experience, and I am excited to carry forward the lessons learned and insights gained into my future endeavors. Thank you to everyone who has contributed to making my internship journey memorable and rewarding.

Contents

1 Introduction **4**
- 1.0.1 eSim . . . 4
- 1.0.2 Ngspice . . . 4
- 1.0.3 NGHDL . . . 5
- 1.0.4 Makerchip-NgVeri . . . 5

2 Problem Statement **6**
- 2.1 Approach . . . 6
- 2.2 Problem in Implementing Zilog Z80 Microprocessor using NgVeri . . . 6

3 Universal Gate **7**
- 3.1 Circuit Details . . . 7
- 3.2 VHDL Code for NAND Gate . . . 7
- 3.3 VHDL Code for NOR Gate . . . 8
- 3.4 Schematic Diagram . . . 8
- 3.5 Ngspice Plots . . . 9

4 3-Bit Ripple Counter **11**
- 4.1 Circuit Details . . . 11
- 4.2 Verilog Code . . . 11
- 4.3 Schematic Diagram . . . 12
- 4.4 Ngspice Plots . . . 12

5 8-bit ALU **13**
- 5.1 Circuit Details . . . 13
- 5.2 Verilog Code . . . 13
- 5.3 Schematic Diagram . . . 14
- 5.4 Ngspice Plots . . . 15

6 BCD to Seven Segment Decoder **18**
- 6.1 Circuit Details . . . 18
- 6.2 Verilog Code . . . 18
- 6.3 Schematic Diagram . . . 19
- 6.4 Ngspice Plots . . . 19

7 Realization of T Flip flop using JK flip flop **21**
7.1 Circuit Details 21
7.2 Verilog Code for JK Flip Flop 21
7.3 Schematic Diagram 22
7.4 Ngspice Plots 22
7.5 Python Plots 23

8 Binary to Gray-Code Converter **25**
8.1 Circuit Details 25
8.2 Verilog Code 25
8.3 Schematic Diagram 26
8.4 Ngspice Plots 26

9 Single Port Memory **28**
9.1 Circuit Details 28
9.2 Verilog Code 28
9.3 Test bench Code 29
9.4 Schematic Diagram 30
9.5 Ngspice Plots 30

10 PID Controller **33**
10.1 Circuit Details 33
10.2 Verilog Code 33
10.3 Schematic Diagram 35
10.4 Ngspice Plots 35

11 Single Stage Delta-Sigma Digital to Analog Converter **37**
11.1 Circuit Details 37
11.2 Verilog Code 37
11.3 Schematic Diagram 39
11.4 Ngspice Plots 39

12 Digital Calculator **42**
12.1 Circuit Details 42
12.2 Verilog Code 42
12.3 Schematic Diagram 43
12.4 Ngspice Plots 43

Bibliography **48**

Chapter 1

Introduction

1.0.1 eSim

The eSim circuit simulation software, developed by the FOSSEE team at IIT Bombay, revolutionizes the accessibility and functionality of electronics simulation. It represents a collaborative effort to democratize access to circuit simulation tools and bridge the gap between theory and practice in electronics. With its open-source nature, eSim fosters a culture of collaboration and customization, empowering users to tailor the software to their specific needs. Its intuitive interface and comprehensive simulation capabilities make it suitable for students, educators, and professionals alike. From simple analog circuits to complex mixed-signal and power electronics circuits, eSim caters to a diverse range of applications. Through this internship report, we delve into the transformative potential of eSim in engineering education and research, showcasing its impact on learning, experimentation, and innovation in electronics.

1.0.2 Ngspice

Ngspice is an open-source mixed-level/mixed-signal electronic circuit simulator. It allows you to simulate electric and electronic circuits, which can include a combination of components such as JFETs, bipolar transistors, MOS transistors, passive elements (like resistors, inductors, or capacitors), diodes, transmission lines, and other devicesall interconnected in a netlist.

Here are some key points about Ngspice:

1. Circuit Simulation: Ngspice numerically solves equations describing electronic circuits. It models time-varying currents, voltages, noise, and small-signal behavior.

2. Mixed-Level/Mixed-Signal: You can simulate both analog and digital circuits. From single gates to complex circuits, Ngspice handles a wide range of designs.

3. Device Models: Ngspice provides a wealth of device models for active, passive, analog, and digital elements. These models come from collections, semiconductor manufacturers, or foundries.

4. Netlist-Based Input: Instead of providing a schematic entry interface, Ngspice accepts input in the form of a netlist. Users describe their circuits using a text-based format.

5. SPICE Compatibility: Ngspice is SPICE-compatible, meaning you can apply PSPICE or LTSPICE device model parameters and netlists for simulating discrete circuits. It can also read HSPICE device libraries from semiconductor foundry Process Design Kits (PDKs) for simulating integrated circuits.

1.0.3 NGHDL

NGHDL, a cornerstone of digital design, provides engineers and researchers with a powerful platform for simulation and verification. Developed as an open-source project, NGHDL embodies collaboration and innovation, drawing upon collective expertise to deliver a sophisticated simulation environment. Rooted in Hardware Description Languages (HDLs), NGHDL offers a versatile framework for modeling complex digital systems with precision. Its robust simulation engine and comprehensive verification features enable engineers to validate designs efficiently, reducing risks and accelerating time-to-market. From basic combinational circuits to advanced processor architectures, NGHDL supports a wide range of digital designs. Throughout this internship report, we explore NGHDL's diverse applications in education, research, and industrial projects, highlighting its pivotal role in driving innovation in digital circuit design. By elucidating NGHDL's capabilities, we underscore its significance in shaping the future of digital design and verification.

1.0.4 Makerchip-NgVeri

Makerchip is a browser-based IDE (Integrated Development Environment) that allows users to simulate Verilog, System Verilog, and TL-Verilog files. It is developed using Verilator, which converts Verilog files into C++ objects. Before using NgVeri in eSim, the design can be simulated in Makerchip with random inputs to ensure that it produces the desired and consistent results. Once the design is successfully simulated, it can be used in mixed-signal designs. These models can be used in digital/mixed signal simulations.

Chapter 2

Problem Statement

The aim of this internship is to deploy digital and mixed-signal circuits within eSim by leveraging NgVeri, a utility designed to translate Verilog models into NgSpice format.

2.1 Approach

The methodology I adopted to implement the mixed-signal and digital circuit for this project can be outlined as follows:

- I began by simulating each Verilog file of the circuit design in ModelSim to confirm its correct functionality. Following this, I utilized NgVeri to validate the conversion of these Verilog files, ensuring that there were no errors in the process.
- Then that individual file is simulated and the Ngspice waveform is generated to check the desired waveform.
- then I proceeded to create the final circuit design, encompassing all the components, to visualize the complete system architecture and ensure the seamless integration of both digital and mixed-signal elements.
- Following the creation of the final circuit design, the entire system was simulated using Ngspice to validate its functionality and performance. This step ensured that the integrated digital and mixed-signal circuit operated as intended and met the project requirements.

2.2 Problem in Implementing Zilog Z80 Microprocessor using NgVeri

I try to implement the Zilog Z80 Microprocessor main programm using NgVeri along with the ALU Programm, Regiter file programm , core CPU programm. When i try to convert these programm to Verilog to NgSpice its shows some errors . Some of the compiler directives are not supported in verilator. That maybe an issue with z80.

Chapter 3

Universal Gate

3.1 Circuit Details

A universal gate refers to a logic gate that can perform all the basic logic operations, including AND, OR, NAND, and NOR. The NAND gate and NOR gate are commonly considered universal gates because they can be used to implement any logical function. By combining multiple instances of a universal gate, complex digital circuits can be constructed efficiently. Universal gates are fundamental components in digital circuit design due to their versatility and ability to simplify circuitry. Here i simulate the NAND and NOR gate using NGHDL ,then I proceeded to create the final circuit design.

3.2 VHDL Code for NAND Gate

```
library ieee;
use ieee.std_logic_1164.all;
entity nand_gate is
port(    a: in std_logic;
         b: in std_logic;
         c: out std_logic
         );
end nand_gate;
architecture beh of nand_gate is
begin
process(a, b)
begin
if (a='1' and b='1') then
        c <= '0';
else
c <= '1';
end if;
end process;
end beh;
```

3.3 VHDL Code for NOR Gate

```
library ieee;
use ieee.std_logic_1164.all;

entity nor_gate is
    port (a : in  std_logic;
          b : in  std_logic;
          c : out std_logic);
end nor_gate;

architecture rtl of nor_gate is
    begin
^^I^^Ic <= a nor b;
end rtl;
```

3.4 Schematic Diagram

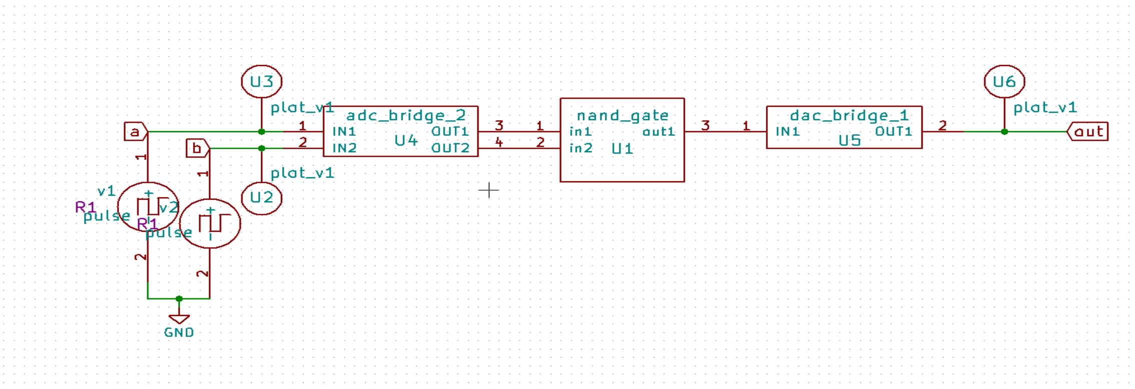


Figure 3.1: NAND Gate

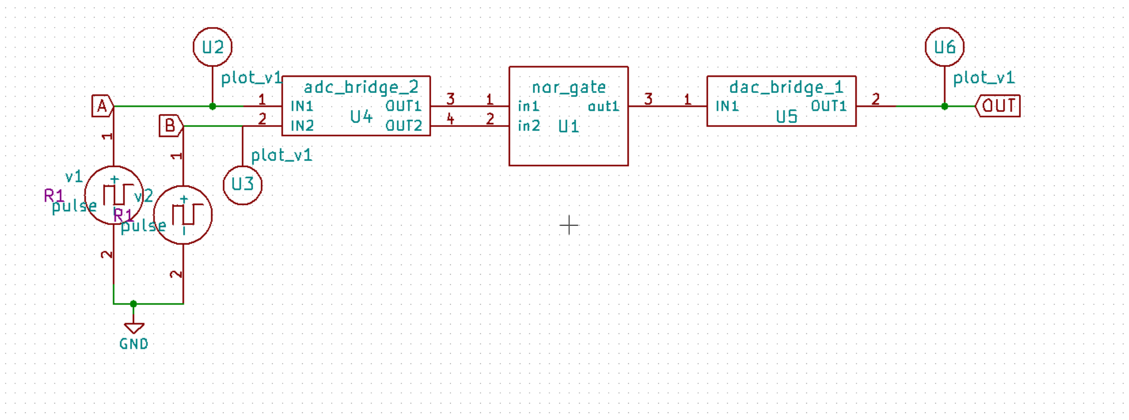


Figure 3.2: NOR Gate

3.5 Ngspice Plots

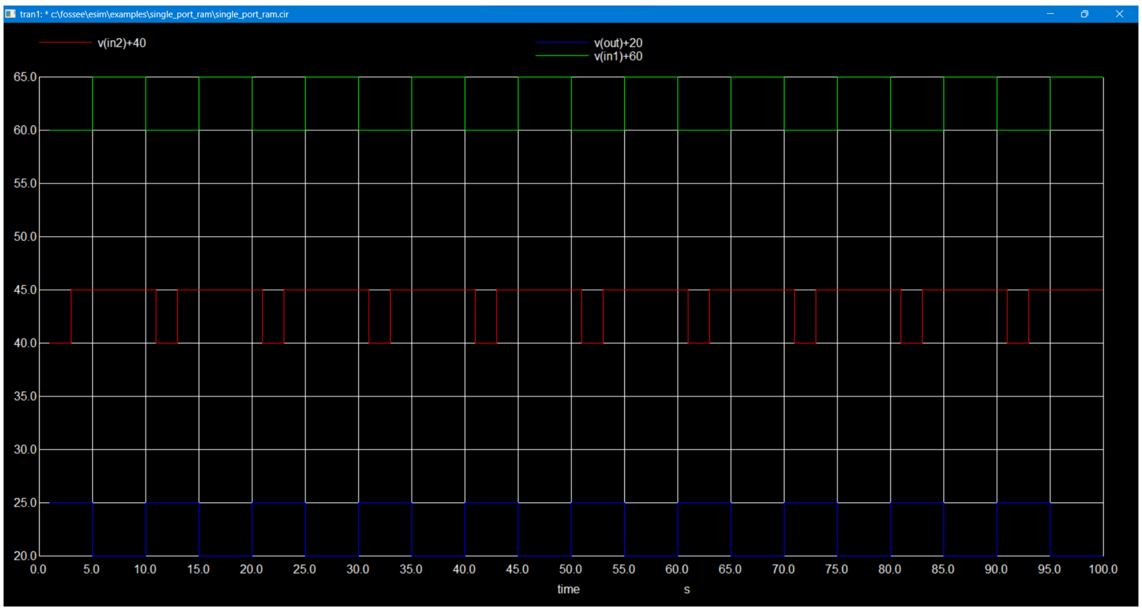


Figure 3.3: NAND Gate

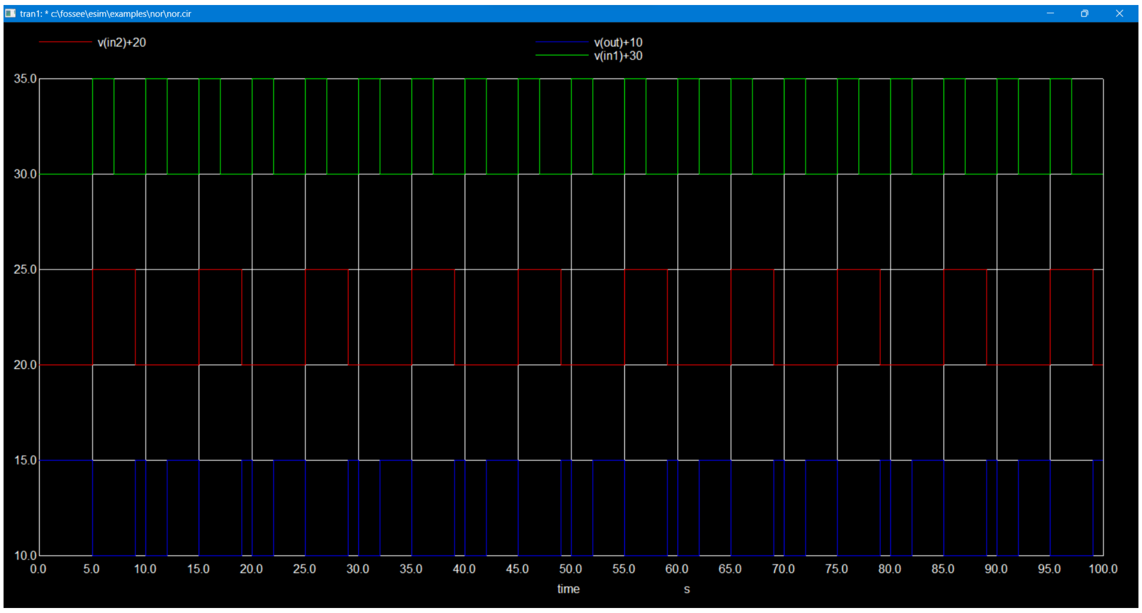


Figure 3.4: NOR Gate

Chapter 4

3-Bit Ripple Counter

4.1 Circuit Details

A 3-bit ripple counter is a digital circuit made up of three T flip-flops connected in series. The output of each flip-flop acts as the clock input to the next flip-flop in the sequence. The first flip-flop is driven by an external clock signal. Each flip-flop toggles its state on the falling or rising edge of the clock pulse, creating a binary counting sequence from 000 to 111, hence achieving a modulo-8 count. The ripple effect refers to the sequential propagation of clock pulses through the flip-flops, introducing slight delays between changes in each flip-flop's state.

4.2 Verilog Code

```verilog
`timescale 1ns/1ns
module ripple_counter(
      input clk_in, reset_in,
      output reg [2:0] Q_out);

      always @(negedge clk_in)
      begin
            if (reset_in == 1)
            begin
              Q_out = 3'b000;
            end
            else
            begin
              Q_out = Q_out + 1;
            end
      end
endmodule
```

4.3 Schematic Diagram

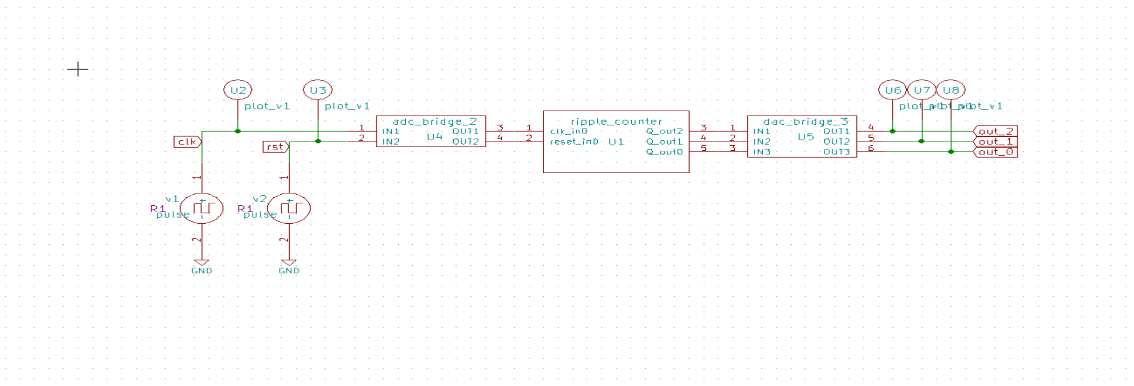


Figure 4.1: 3-bit ripple counter

4.4 Ngspice Plots

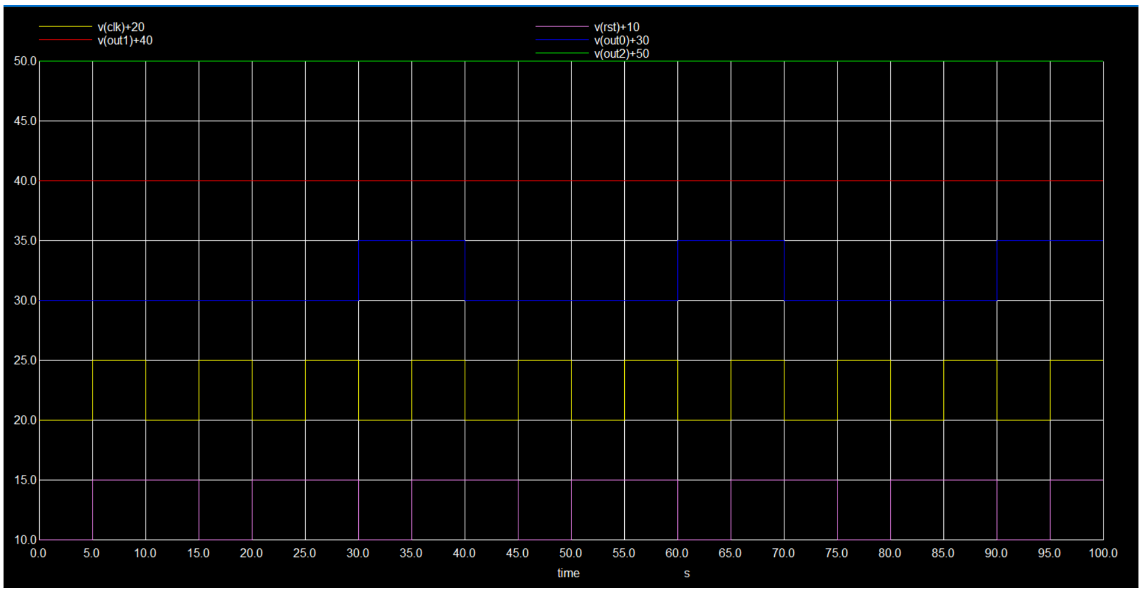


Figure 4.2: NgSpice Plot

Chapter 5

8-bit ALU

5.1 Circuit Details

An 8-bit Arithmetic Logic Unit (ALU) is a critical digital circuit used in computer processors and various digital systems to perform arithmetic and logical operations. The "8-bit" designation indicates that the ALU processes data and instructions that are 8 bits wide.The ALU processes 8 bits of data at a time, meaning it can handle numbers and binary operations on values ranging from 0 to 255 (unsigned) or -128 to 127 (signed, using two's complement representation). The provided Verilog code defines an 8-bit ALU capable of performing various arithmetic and logical operations based on a 4-bit select input. The carry output indicates the presence of a carry-out from addition operations. The placeholders for multiplication and division indicate that additional logic is needed for these operations. This ALU is a fundamental component that can be used in processors, microcontrollers, and other digital systems requiring arithmetic and logic functionalities.

5.2 Verilog Code

```
module ALU_8bit (
  input [7:0] A, B,
  input [3:0] select,
  output [7:0] out,
  output carry
);

  reg [7:0] ALU_Result;
  wire [8:0] tmp;

  assign out = ALU_Result;

  assign tmp = {1'b0, A} + {1'b0, B};
  assign carry = tmp[8];
  always @(*) begin
    case(select)
      4'b0000: ALU_Result = A + B;
      4'b0001: ALU_Result = A - B;
      // Replace with appropriate logic for 8-bit multiplication
      4'b0010: ALU_Result = 8'd0; // Placeholder for multiplication
      // Replace with appropriate logic for 8-bit division
      4'b0011: ALU_Result = 8'd0; // Placeholder for division
      4'b0100: ALU_Result = A << 1;
      4'b0101: ALU_Result = A >> 1;
      4'b0110: ALU_Result = {A[6:0], A[7]};
      4'b0111: ALU_Result = {A[0], A[7:1]};
      4'b1000: ALU_Result = A & B;
      4'b1001: ALU_Result = A | B;
      4'b1010: ALU_Result = A ^ B;
      4'b1011: ALU_Result = ~(A | B);
      4'b1100: ALU_Result = ~(A & B);
      4'b1101: ALU_Result = ~(A ^ B);
      4'b1110: ALU_Result = (A > B) ? 8'd1 : 8'd0;
      4'b1111: ALU_Result = (A == B) ? 8'd1 : 8'd0;
      default: ALU_Result = A + B;
    endcase
  end
endmodule
```

5.3 Schematic Diagram

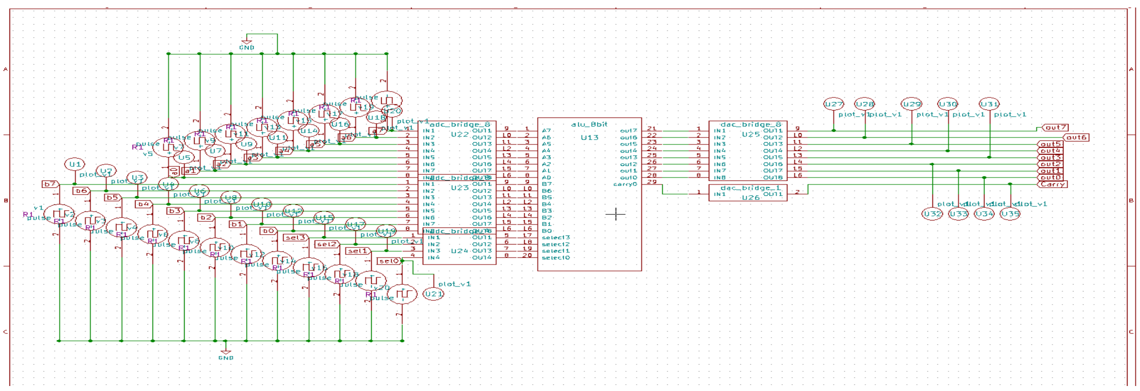


Figure 5.1: 8-bit ALU

5.4 Ngspice Plots

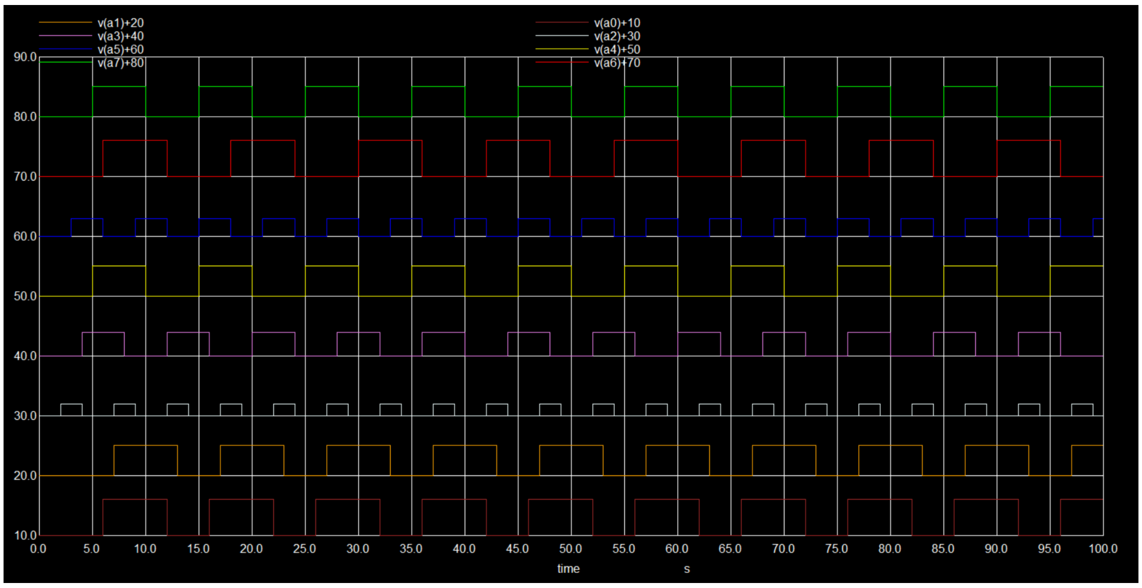


Figure 5.2: INPUT-1

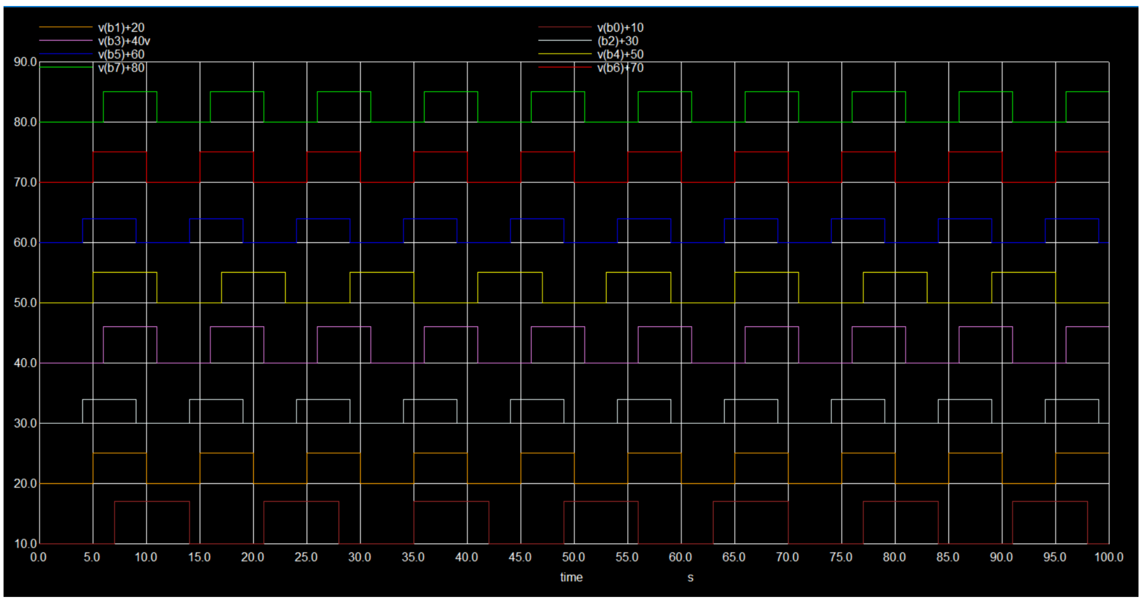


Figure 5.3: INPUT-2

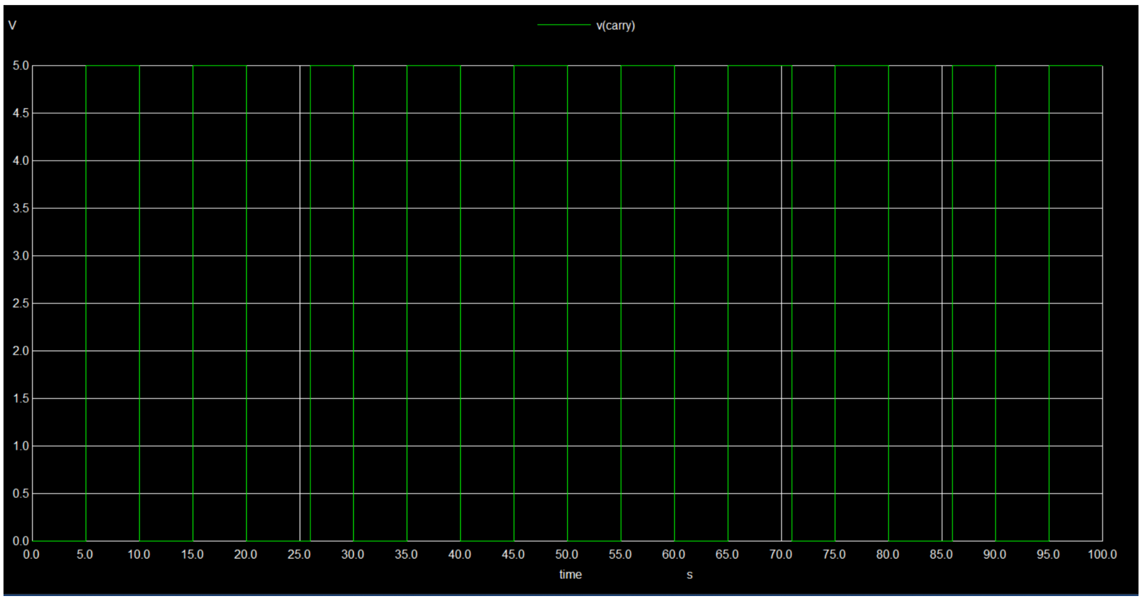


Figure 5.4: CARRY OUT

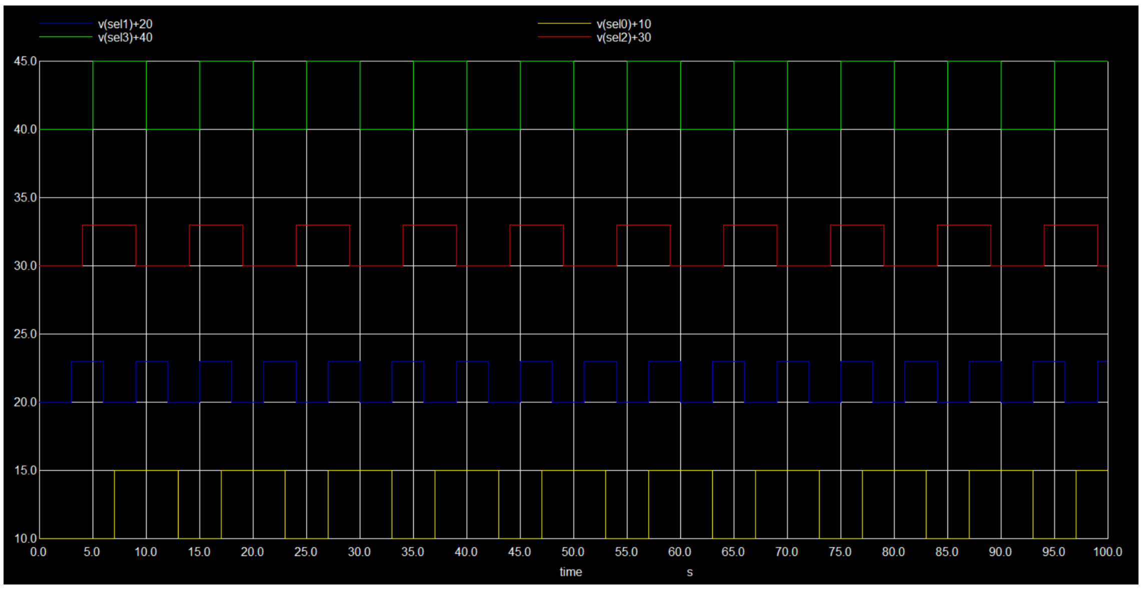


Figure 5.5: SELECTION LINE

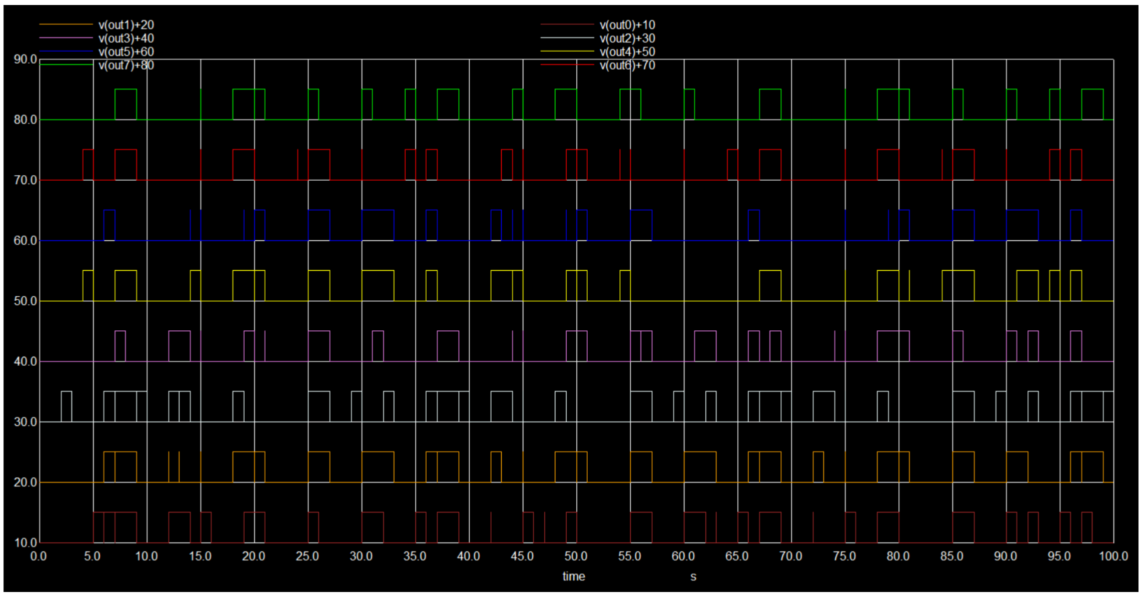


Figure 5.6: OUTPUT

Chapter 6

BCD to Seven Segment Decoder

6.1 Circuit Details

A BCD (Binary-Coded Decimal) to 7-Segment Decoder is a digital circuit that converts a BCD input (representing digits 0-9) into signals that can drive a 7-segment display. A 7-segment display consists of seven LEDs (labeled a through g) arranged to form a digit 0-9. The decoder maps each 4-bit BCD input to the appropriate combination of these seven segments to display the corresponding digit. The BCD input consists of four binary digits (bits) that represent a decimal digit. The range of BCD is from 0000 (0) to 1001 (9). The 7-segment output controls the seven LEDs to display the corresponding decimal digit. Each segment is controlled by a separate output.

6.2 Verilog Code

```
module BCD_to_7_Segment_decoder(
    input [3:0] bcd,     // 4-bit BCD input signal
    output reg [6:0] seg // 7-segment display output signal
);
always @ (bcd) begin
        case(bcd)
            4'b0000: seg = 7'b1000000;
            4'b0001: seg = 7'b1111001;
            4'b0010: seg = 7'b0100100;
            4'b0011: seg = 7'b0110000;
            4'b0100: seg = 7'b0011001;
            4'b0101: seg = 7'b0010010;
            4'b0110: seg = 7'b0000010;
            4'b0111: seg = 7'b1111000;
            4'b1000: seg = 7'b0000000;
            4'b1001: seg = 7'b0010000;
            default: seg = 7'b1111111;
        endcase
    end
endmodule
```

6.3 Schematic Diagram

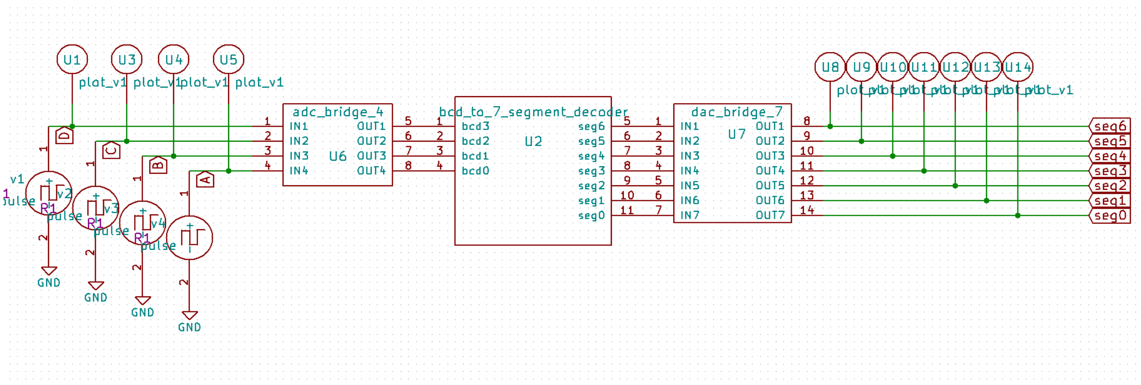


Figure 6.1: BCD to Seven Segment Decoder

6.4 Ngspice Plots

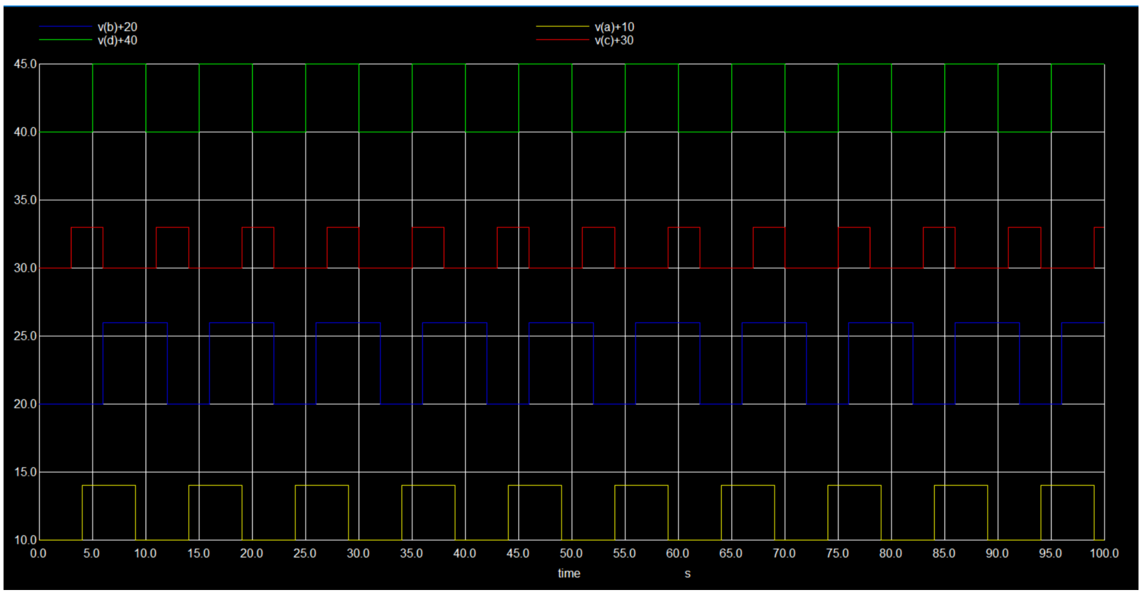


Figure 6.2: BCD INPUT

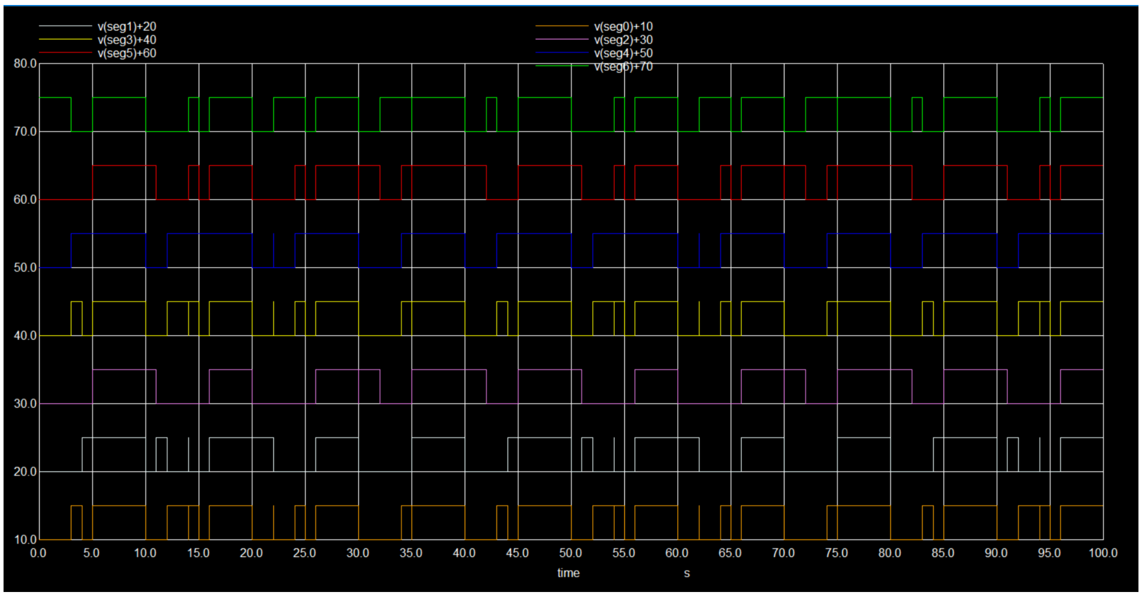


Figure 6.3: SEGMENT OUTPUT

Chapter 7

Realization of T Flip flop using JK flip flop

7.1 Circuit Details

A T flip-flop can be realized using a JK flip-flop by appropriately connecting the J and K inputs of the JK flip-flop. The T flip-flop toggles its state when its input (T) is high (1) and maintains its state when the input is low (0). This behavior can be achieved by tying the J and K inputs together and connecting them to the T input. Connect the J and K inputs of the JK flip-flop together. Use this common connection as the T input of the T flip-flop. When T = 0, both J and K inputs are 0. The JK flip-flop does not change its state. When T = 1, both J and K inputs are 1. The JK flip-flop toggles its state.

7.2 Verilog Code for JK Flip Flop

```
module JK_FlipFlop (
  input J, K, Clk,
  output reg Q,
  output reg Qbar
);

always @(posedge Clk) begin
  case ({J, K})
    2'b00: Q <= Q;
    2'b01: Q <= 1'b0;
    2'b10: Q <= 1'b1;
    2'b11: Q <= ~Q;
    default: ;
  endcase
end
```

```
assign Qbar = ~Q;

endmodule
```

7.3 Schematic Diagram

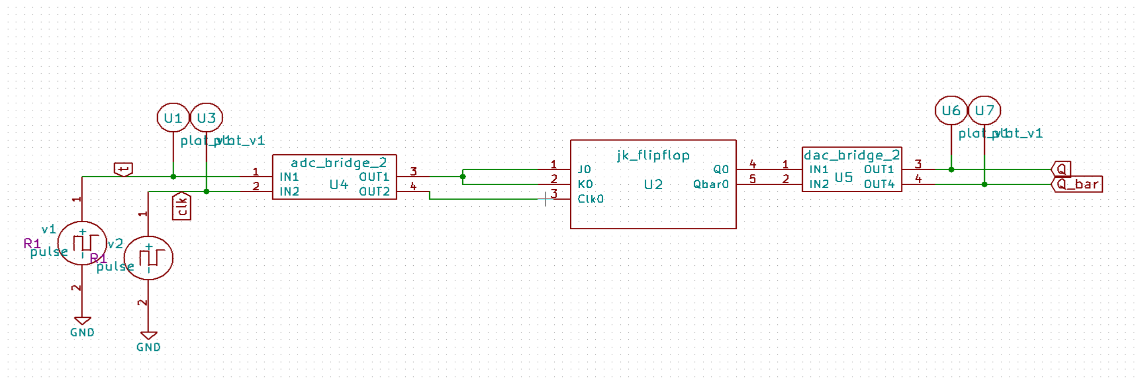


Figure 7.1: T FLIP FLOP

7.4 Ngspice Plots

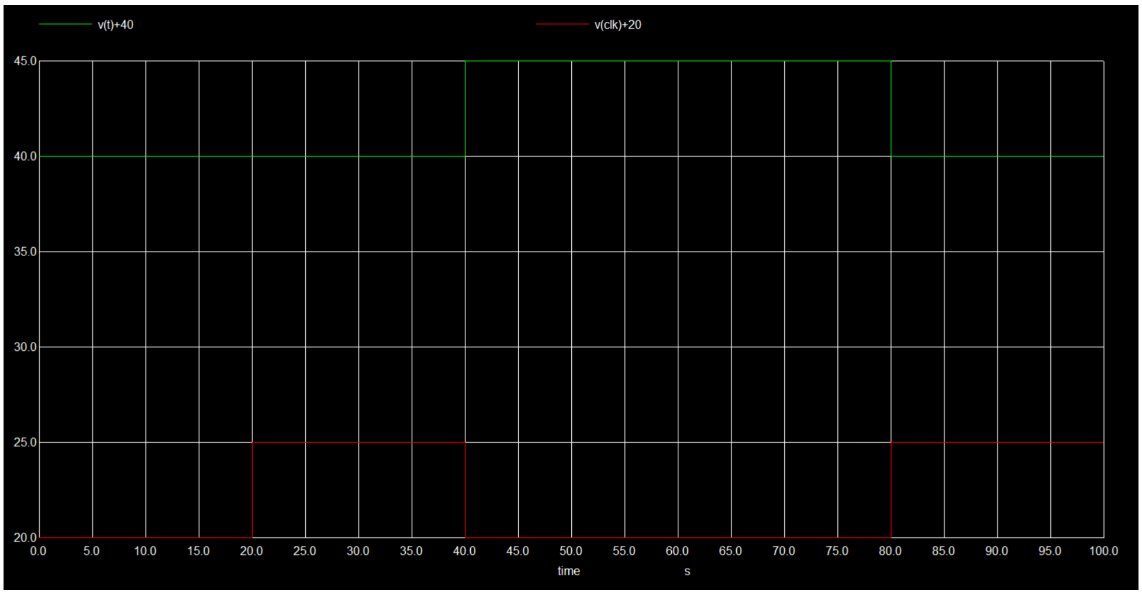


Figure 7.2: INPUT

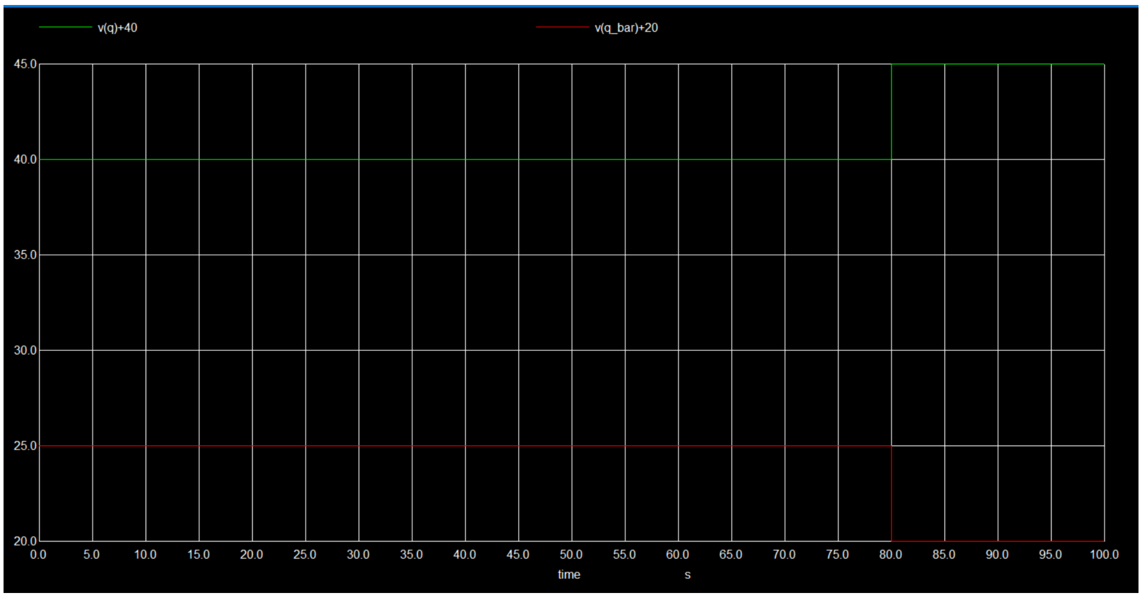
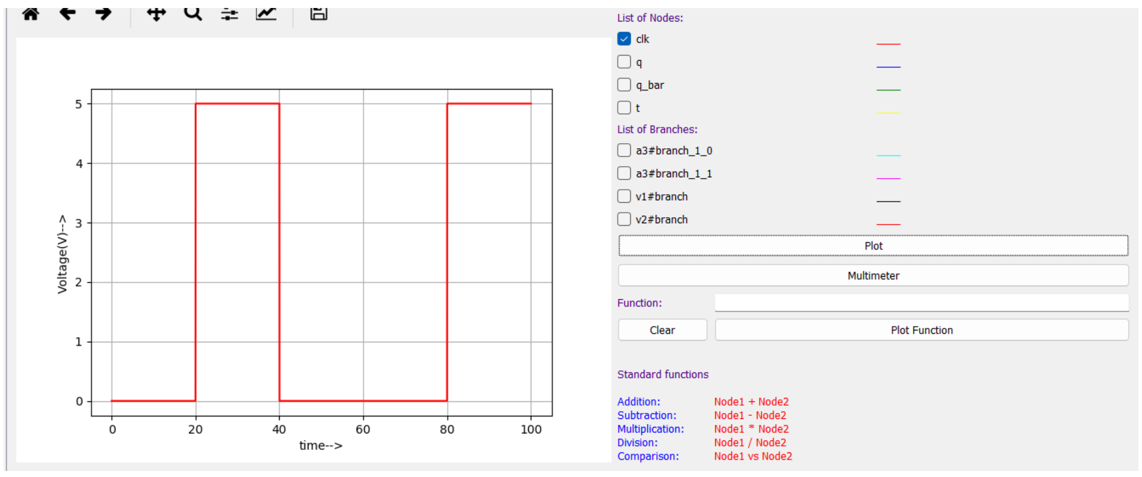


Figure 7.3: OUTPUT

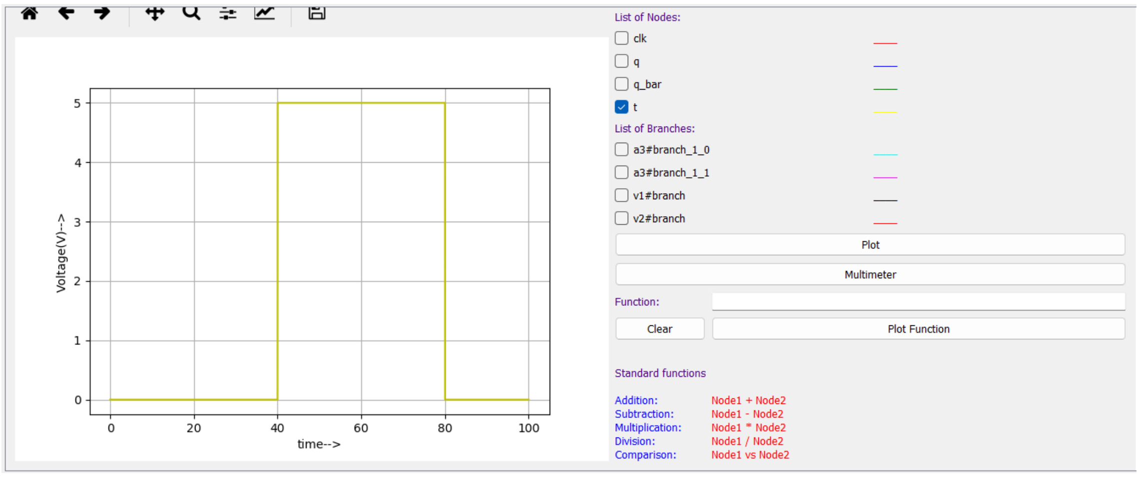
7.5 Python Plots



Standard functions

Addition: Node1 + Node2
Subtraction: Node1 - Node2
Multiplication: Node1 * Node2
Division: Node1 / Node2
Comparison: Node1 vs Node2

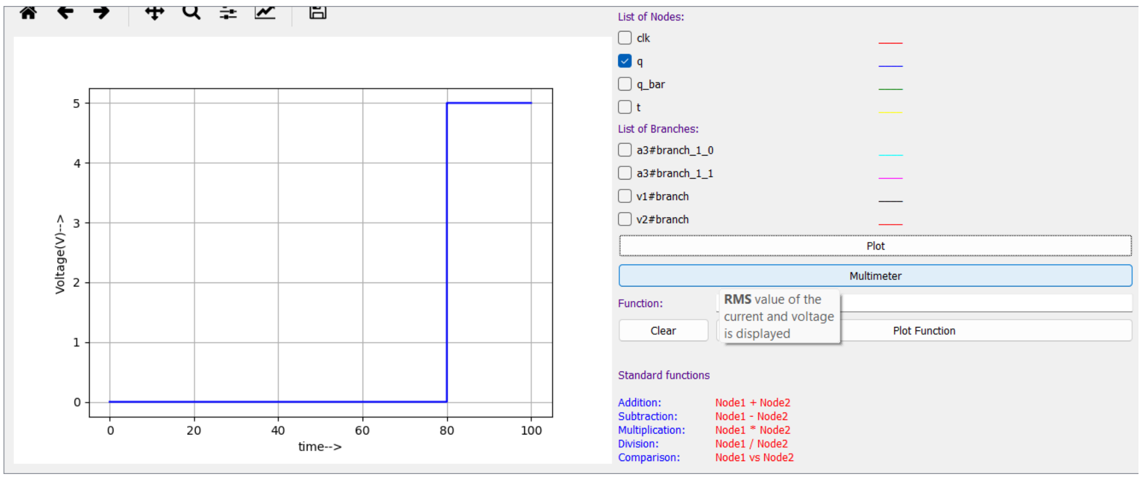
Figure 7.4: CLOCK INPUT



Standard functions

Addition:	Node1 + Node2
Subtraction:	Node1 - Node2
Multiplication:	Node1 * Node2
Division:	Node1 / Node2
Comparison:	Node1 vs Node2

Figure 7.5: T INPUT

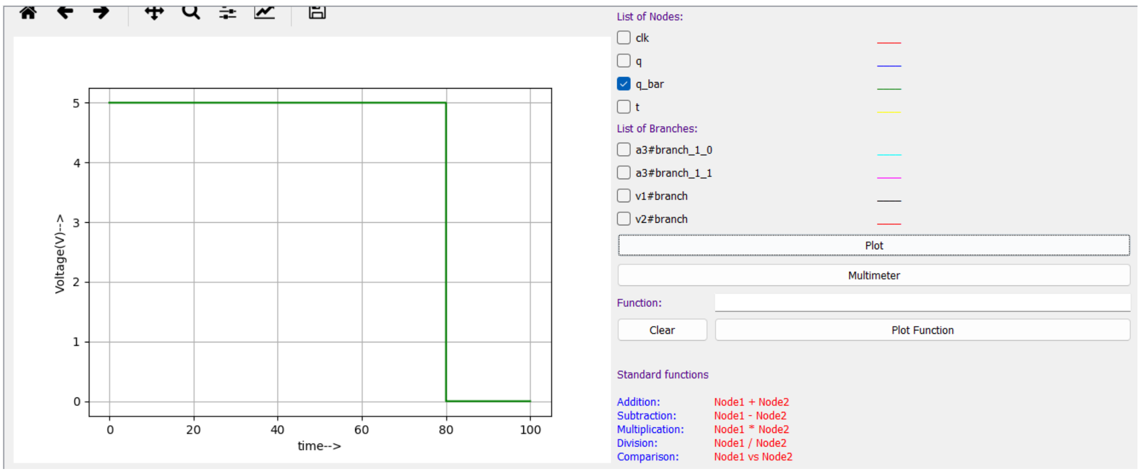


RMS value of the current and voltage is displayed

Standard functions

Addition:	Node1 + Node2
Subtraction:	Node1 - Node2
Multiplication:	Node1 * Node2
Division:	Node1 / Node2
Comparison:	Node1 vs Node2

Figure 7.6: OUTPUT -Q



Standard functions

Addition:	Node1 + Node2
Subtraction:	Node1 - Node2
Multiplication:	Node1 * Node2
Division:	Node1 / Node2
Comparison:	Node1 vs Node2

Figure 7.7: OUTPUT Q-BAR

Chapter 8

Binary to Gray-Code Converter

8.1 Circuit Details

A Binary to Gray code converter is a digital circuit that converts a binary number into its corresponding Gray code representation. Gray code is a binary numeral system where two successive values differ in only one bit, which minimizes errors in digital systems, especially during transitions. The conversion from binary to Gray code can be derived using the following logic:

- The most significant bit (MSB) of the Gray code is the same as the MSB of the binary code.
- Each subsequent bit of the Gray code can be found by XOR-ing the current binary bit with the previous binary bit.

8.2 Verilog Code

```
module binary_to_gray
        (input [3:0] bin, //binary input
         output [3:0] G //gray code output
        );
assign G[3] = bin[3];
assign G[2] = bin[3] ^ bin[2];
assign G[1] = bin[2] ^ bin[1];
assign G[0] = bin[1] ^ bin[0];
endmodule
```

8.3 Schematic Diagram

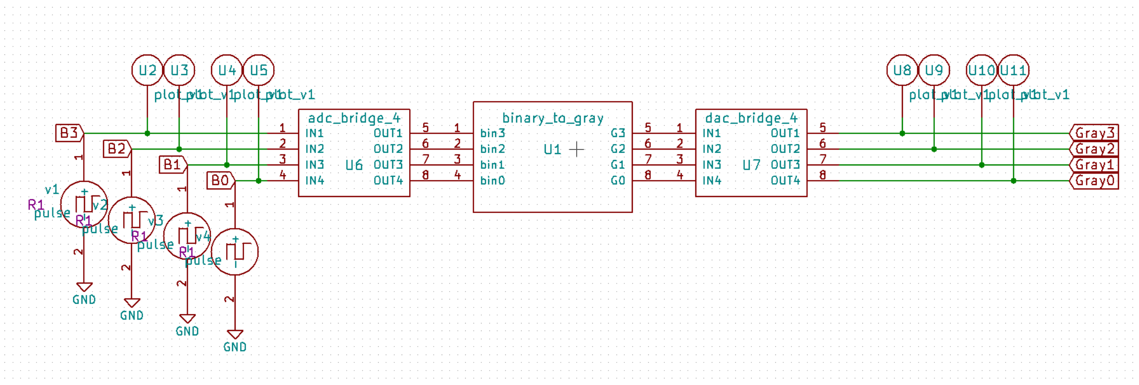


Figure 8.1: Binary to Gray-Code Converter

8.4 Ngspice Plots

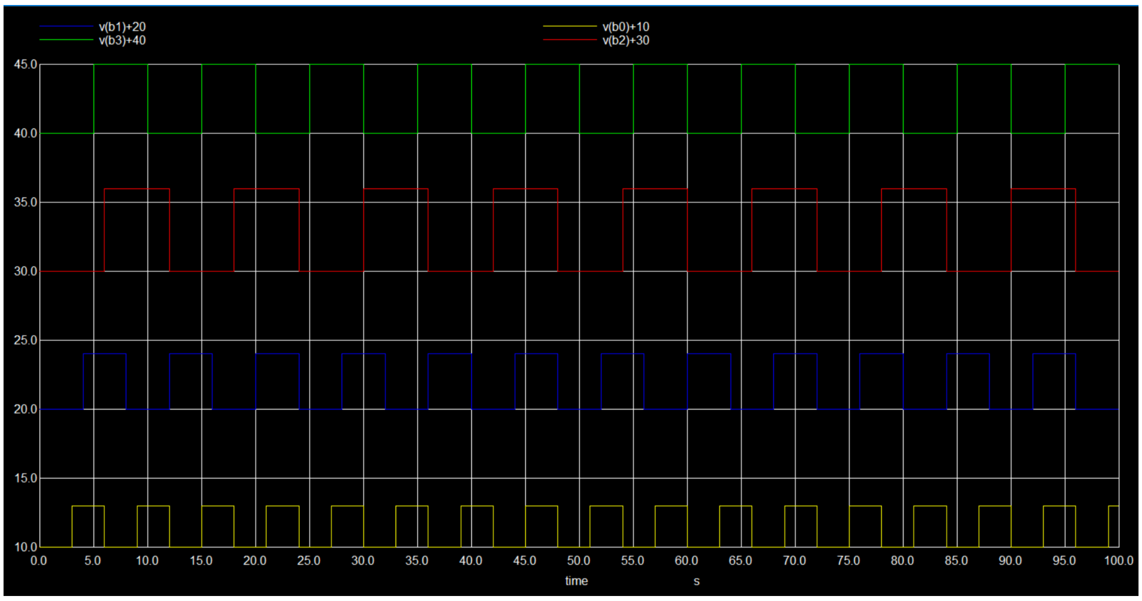


Figure 8.2: BINARY INPUT

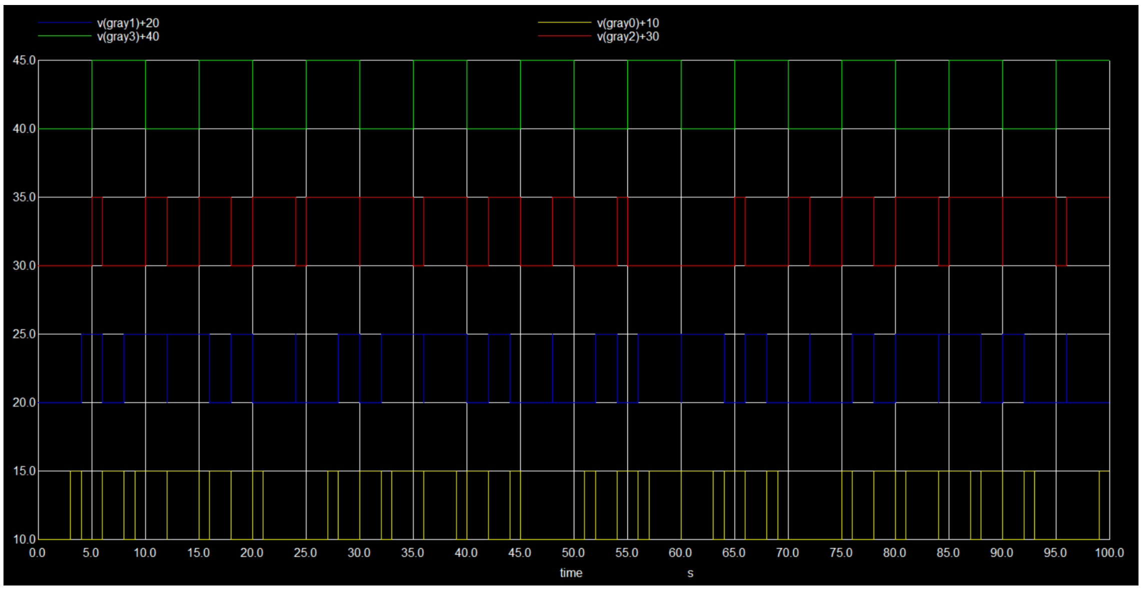


Figure 8.3: GRAY CODE OUTPUT

Chapter 9

Single Port Memory

9.1 Circuit Details

A single port memory is a type of memory that can be accessed by only one device or process at a time. In this type of memory, data can be written and read from the same port. It is generally used in applications where only one processor is used, and the memory is not required to be accessed by multiple processors simultaneously. The design of a single port memory in Verilog involves the use of a register array. The size of the register array is determined by the number of data bits that need to be stored in memory. In this blog post, we will discuss the design of a single port memory with 8-bit data and 16-bit address.

9.2 Verilog Code

```
module single_port_memory(
  input clk,        // Clock input
  input [15:0] addr,  // Address input
  input [7:0] din,    // Data input
  input wr_en,      // Write enable input
  output reg [7:0] dout // Data output
);

reg [7:0] mem [0:65535]; // Register array for memory storage

always @(posedge clk) begin
  if(wr_en)   // Write operation
    mem[addr] <= din;
  else    // Read operation
    dout <= mem[addr];
end

endmodule
```

9.3 Test bench Code

```
module single_port_RAM_tb;

    reg clk;
    reg [15:0] addr;    // Address input
    reg [7:0] din;      // Data input
    reg wr_en;          // Write enable input
    wire [7:0] dout;    // Data output

    // Instantiate the RAM module
    single_port_RAM ram (
        .clk(clk),
        .addr(addr),
        .din(din),
        .wr_en(wr_en),
        .dout(dout)
    );

    // Provide stimulus
    initial begin
        clk = 0;
        addr = 16'h0000;  // Assign an address (e.g., 0x0000)
        din = 8'hFF;      // Example data input (e.g., 0xFF)
        wr_en = 1;        // Enable write operation

        // Apply stimulus for a few clock cycles
        #10;
        clk = 1;
        #10;
        clk = 0;

        // Add more stimulus if needed...
    end

endmodule
```

9.4 Schematic Diagram

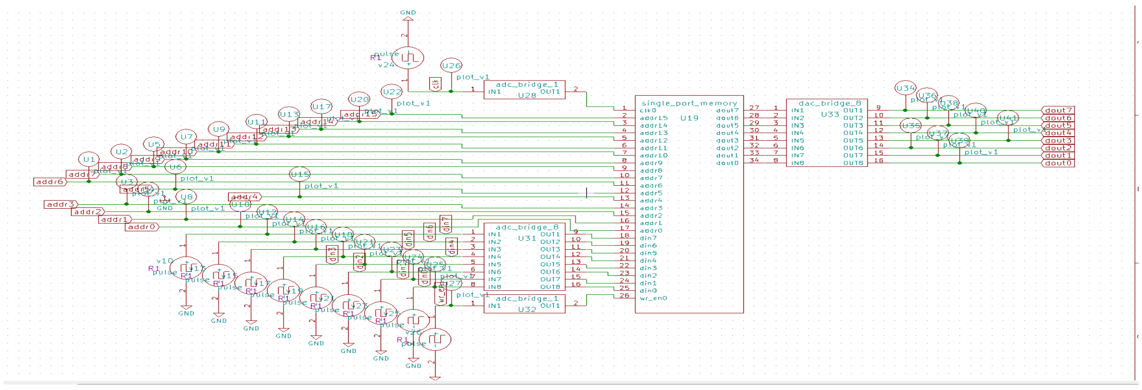


Figure 9.1: SINGLE PORT MEMORY

9.5 Ngspice Plots

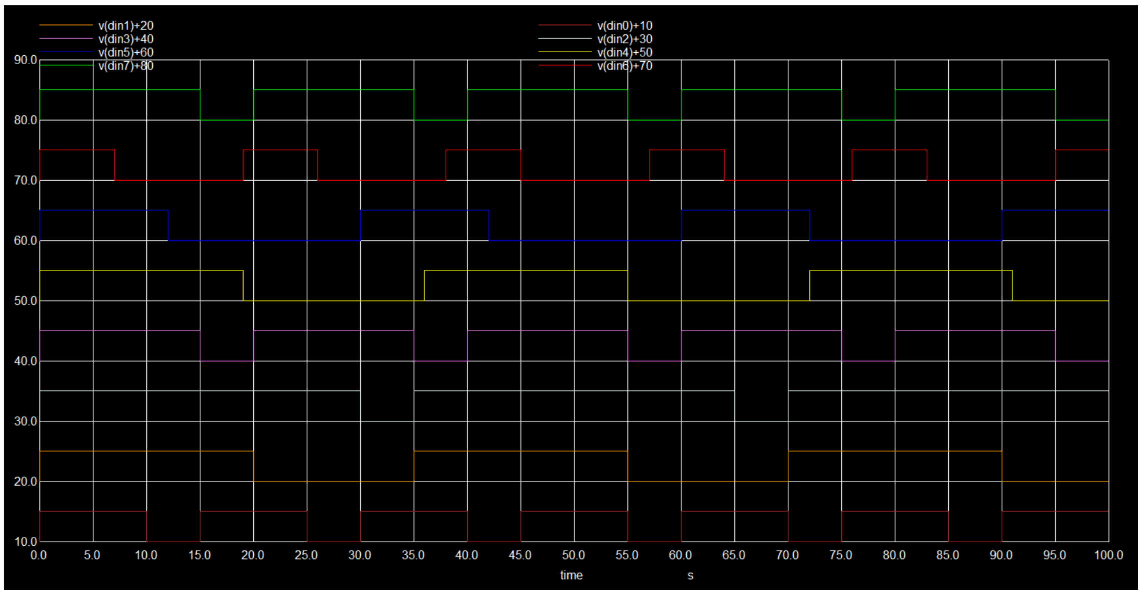


Figure 9.2: DATA INPUT

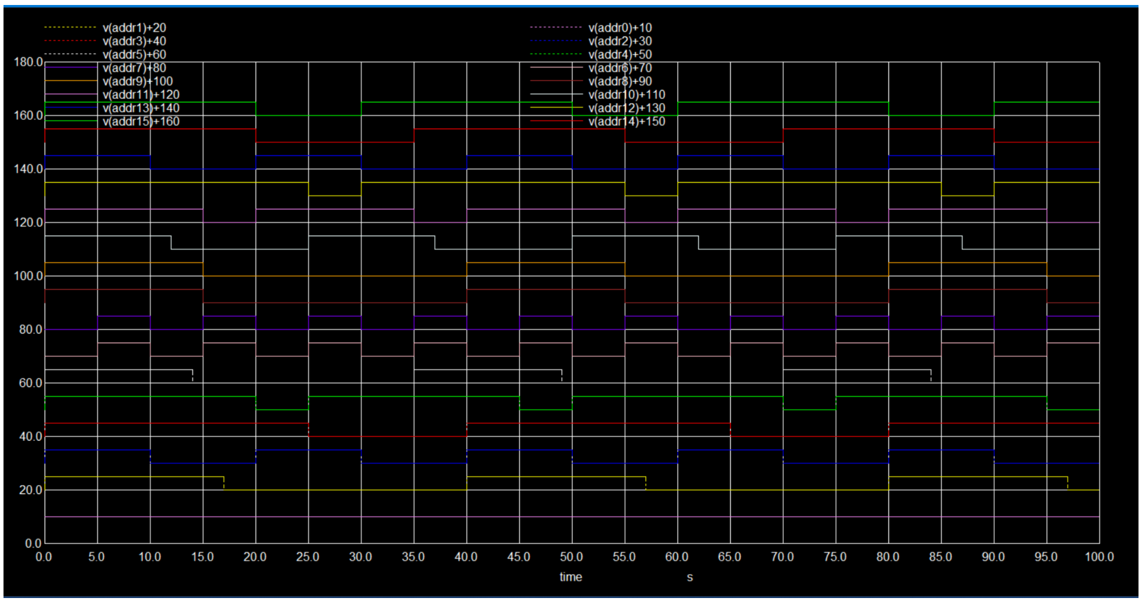


Figure 9.3: ADDRESS INPUT

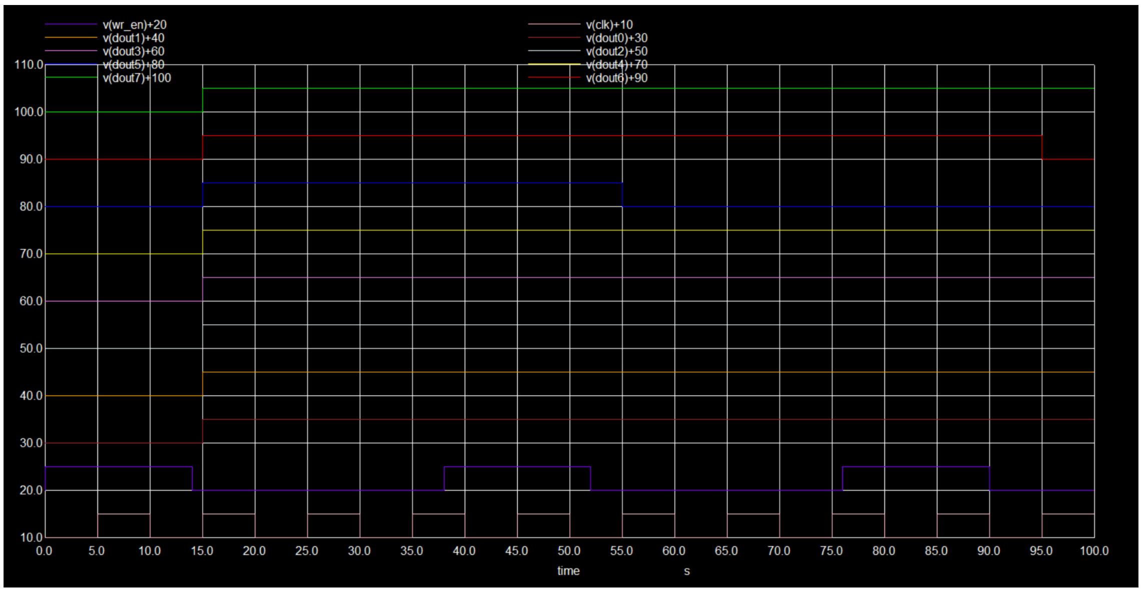
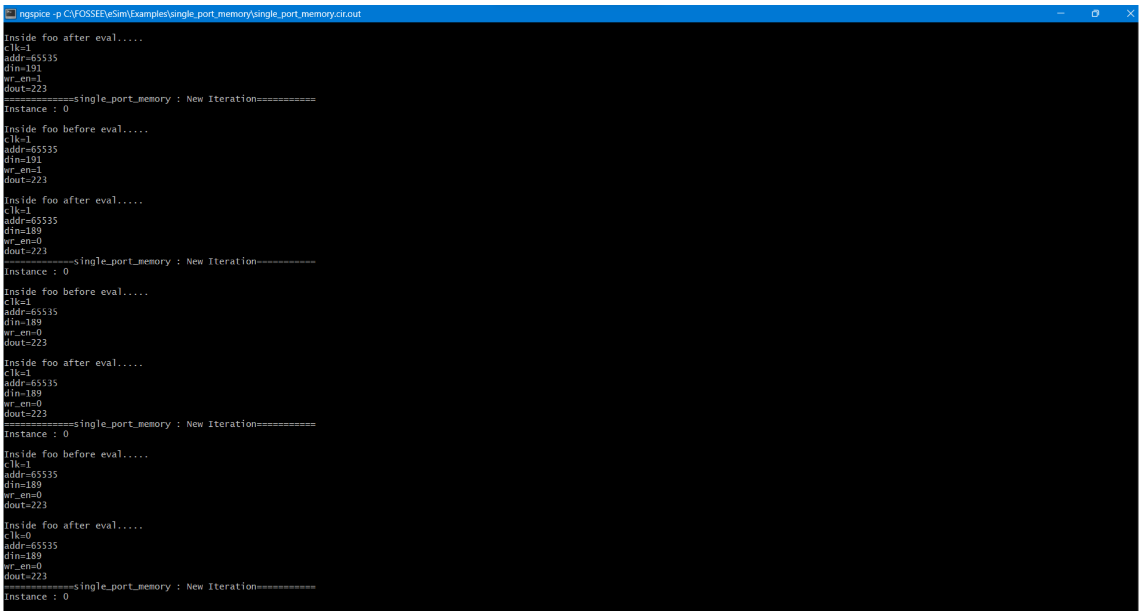


Figure 9.4: DATA OUTPUT



```
ngspice -p C:\FOSSEE\eSim\Examples\single_port_memory\single_port_memory.cir.out

Inside foo after eval.....
clk=1
addr=65535
din=191
wr_en=1
dout=223
=============single_port_memory : New Iteration===========
Instance : 0

Inside foo before eval.....
clk=1
addr=65535
din=191
wr_en=1
dout=223

Inside foo after eval.....
clk=1
addr=65535
din=189
wr_en=0
dout=223
=============single_port_memory : New Iteration===========
Instance : 0

Inside foo before eval.....
clk=1
addr=65535
din=189
wr_en=0
dout=223

Inside foo after eval.....
clk=1
addr=65535
din=189
wr_en=0
dout=223
=============single_port_memory : New Iteration===========
Instance : 0

Inside foo before eval.....
clk=1
addr=65535
din=189
wr_en=0
dout=223

Inside foo after eval.....
clk=0
addr=65535
din=189
wr_en=0
dout=223
=============single_port_memory : New Iteration===========
Instance : 0
```

Figure 9.5: NgSpice Log

Chapter 10

PID Controller

10.1 Circuit Details

A PID controller (Proportional-Integral-Derivative controller) is a control loop feedback mechanism widely used in industrial control systems. It continuously calculates an error value as the difference between a desired setpoint and a measured process variable, and applies a correction based on proportional, integral, and derivative terms, hence the name PID. Using the proportional, one can know the error and this error helps in providing the corrective response value. The proportional term is even termed as proportional gain constant. With integral, the past error values are known and integrated. When the error values are excluded from the system, then the integral value gets increased. And using the derivative, the forthcoming error values are expected depending on the current values. **Features** 16-bit arithmetic for high-precision control. Configurable Kp, Ki, Kd coefficients to adapt to various systems. Clock prescaling feature to adjust the controller's sampling rate. Includes a testbench simulating a generic linear system for validation.

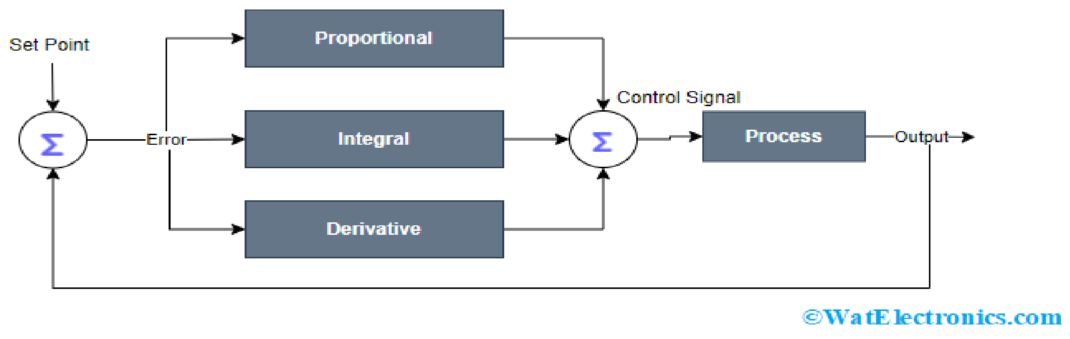


Figure 10.1: BLOCK DIAGRAM

10.2 Verilog Code

```
`timescale 1ns / 1ps

module pid_controller(
    input clk,
    input rst_n,
    input [1:0] setpoint,
    input [1:0] feedback,
    input [1:0] Kp,
    input [1:0] Ki,
    input [1:0] Kd,
    input [1:0] clk_prescaler,
    output reg [1:0] control_signal
);

    // Internal signals

    reg [1:0] prev_error = 2'b00;
    reg [3:0] integral = 4'b0000;
    reg [1:0] derivative = 2'b00;

    // Clock divider for sampling rate
    reg [1:0] clk_divider = 2'b00;
    reg sampling_flag = 0;

    always @(posedge clk or negedge rst_n) begin
        if (~rst_n)
            clk_divider <= 2'b00;
        else if (clk_divider == clk_prescaler) begin
            clk_divider <= 2'b00;
            sampling_flag <= 1;
        end else begin
            clk_divider <= clk_divider + 1;
            sampling_flag <= 0;
        end
    end

    always @(posedge clk or negedge rst_n) begin

        if (~rst_n) begin
            // Reset logic generally specific to application
        end
        else if (sampling_flag) begin

            // PID Calculation
            integral <= integral + (Ki * (setpoint - feedback));
            derivative <= Kd * ((setpoint - feedback) - prev_error);
            // Calculate control signal
            control_signal = (Kp * (setpoint - feedback)) + integral[1:0] +
             ↪  derivative;
            prev_error <= (setpoint - feedback);
        end
    end

endmodule
```

10.3 Schematic Diagram

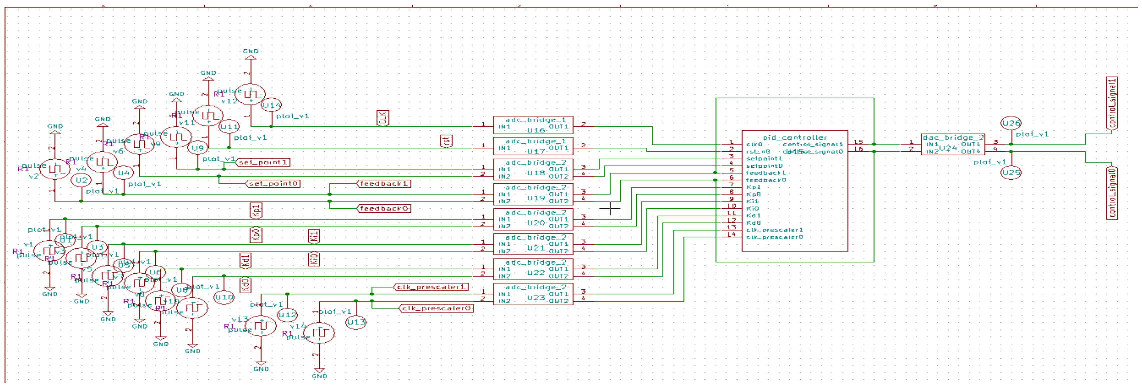


Figure 10.2: PID CONTROLLER

10.4 Ngspice Plots

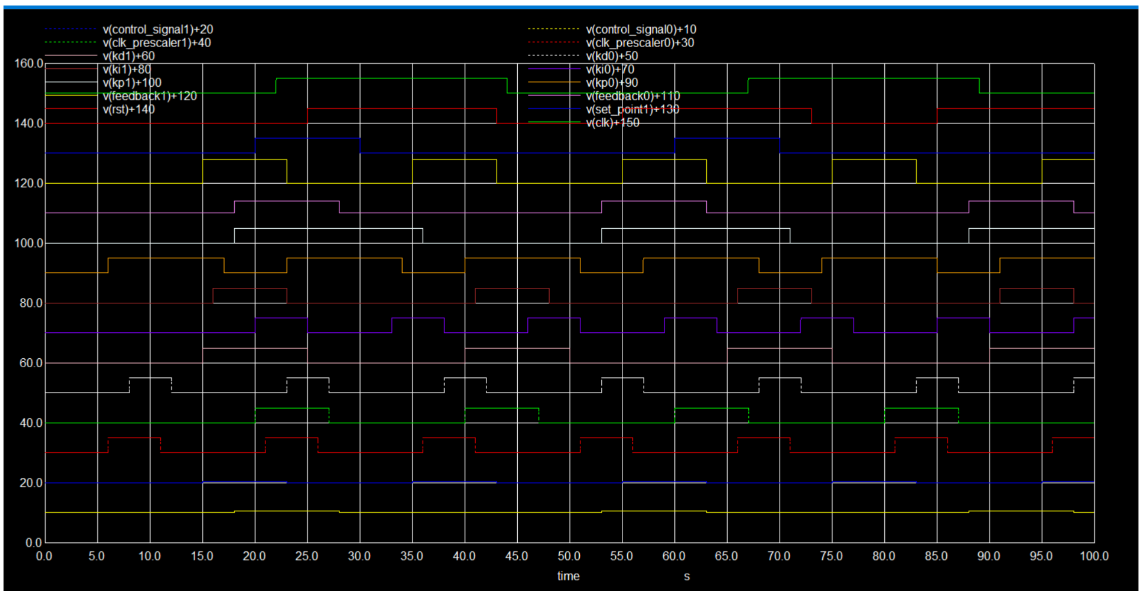
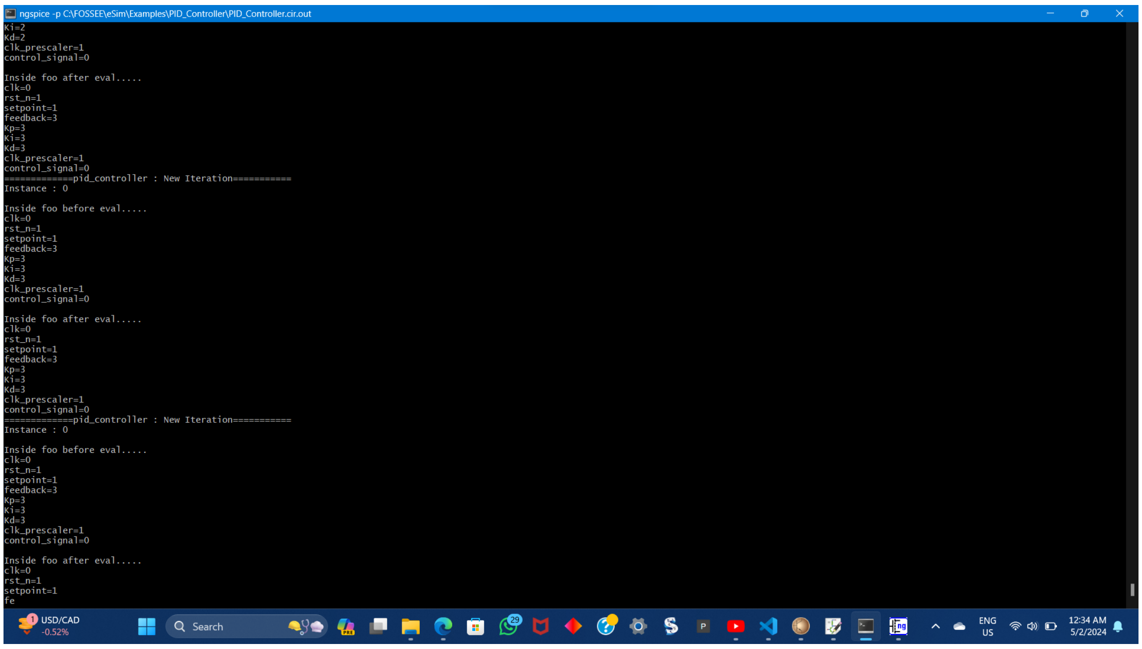


Figure 10.3: NgSpice Plot



ngspice -p C:\FOSSEE\eSim\Examples\PID_Controller\PID_Controller.cir.out

```
Ki=2
Kd=2
clk_prescaler=1
control_signal=0

Inside foo after eval.....
clk=0
rst_n=1
setpoint=1
feedback=3
Kp=3
Ki=3
Kd=3
clk_prescaler=1
control_signal=0
=============pid_controller : New Iteration===========
Instance : 0

Inside foo before eval.....
clk=0
rst_n=1
setpoint=1
feedback=3
Kp=3
Ki=3
Kd=3
clk_prescaler=1
control_signal=0

Inside foo after eval.....
clk=0
rst_n=1
setpoint=1
feedback=3
Kp=3
Ki=3
Kd=3
clk_prescaler=1
control_signal=0
=============pid_controller : New Iteration===========
Instance : 0

Inside foo before eval.....
clk=0
rst_n=1
setpoint=1
feedback=3
Kp=3
Ki=3
Kd=3
clk_prescaler=1
control_signal=0

Inside foo after eval.....
clk=0
rst_n=1
setpoint=1
fe
```

Figure 10.4: NGSPICE LOG

Chapter 11

Single Stage Delta-Sigma Digital to Analog Converter

11.1 Circuit Details

A Delta-Sigma Digital to Analog Converter (DAC) is a type of DAC that oversamples the input signal and uses noise shaping to push quantization noise out of the band of interest. This method results in high-resolution digital-to-analog conversion. A single stage Delta-Sigma DAC uses a single Delta-Sigma modulator stage to convert a digital input signal to an oversampled and noise-shaped output, which is then filtered to produce the final analog output.

11.2 Verilog Code

```
‘timescale 1ns / 1ps


module DAC #(
^^Iparameter dac_bw = 16
)(

^^Iinput^^Iwire^^I^^I^^I^^Iclk,
^^Iinput^^Iwire^^I^^I^^I^^Irst_n,
^^Iinput^^Iwire^^I[15 : 0]^^Idin,
^^Ioutput^^Iwire^^I^^I^^I^^Idout
);

^^Ilocalparam bw_ext = 2;
^^Ilocalparam bw_tot = dac_bw + bw_ext;

^^Ireg^^I^^I^^I^^I^^I^^Idout_r;
^^Ireg^^I^^I^^I^^I^^I^^Idac_dout;
```

```
^^Ireg signed^^I^^I[bw_tot-1 : 0]^^IDAC_acc_1st;

^^Iwire signed^^I^^I[bw_tot-1 : 0]^^Imax_val = (2**(dac_bw - 1) - 1);
^^Iwire signed^^I^^I[bw_tot-1 : 0]^^Imin_val = -(2**(dac_bw - 1));
^^Iwire signed^^I^^I[bw_tot-1 : 0]^^Idac_val = (!dout_r) ? max_val : min_val;

^^Iwire signed^^I^^I[bw_tot-1 : 0]^^Iin_ext = {{bw_ext{din[dac_bw - 1]}}, din};
^^Iwire signed^^I^^I[bw_tot-1 : 0]^^Idelta_s0_c0 = in_ext + dac_val;
^^Iwire signed^^I^^I[bw_tot-1 : 0]^^Idelta_s0_c1 = DAC_acc_1st + delta_s0_c0;

^^Ialways@(posedge clk)begin
^^I^^Iif(!rst_n)begin
^^I^^I^^IDAC_acc_1st <= 'd0;
^^I^^Iend else begin
^^I^^I^^IDAC_acc_1st <= delta_s0_c1;
^^I^^Iend
^^Iend

^^Ialways@(posedge clk)begin
^^I^^Iif(!rst_n)begin
^^I^^I^^Idout_r^^I^^I<= 1'b0;
^^I^^I^^Idac_dout^^I<= 1'b0;
^^I^^Iend else begin
^^I^^I^^Idout_r^^I^^I<= delta_s0_c1[bw_tot-1];
^^I^^I^^Idac_dout^^I<= ~dout_r;
^^I^^Iend
^^Iend

^^Iassign dout = dout_r;

endmodule
```

11.3 Schematic Diagram

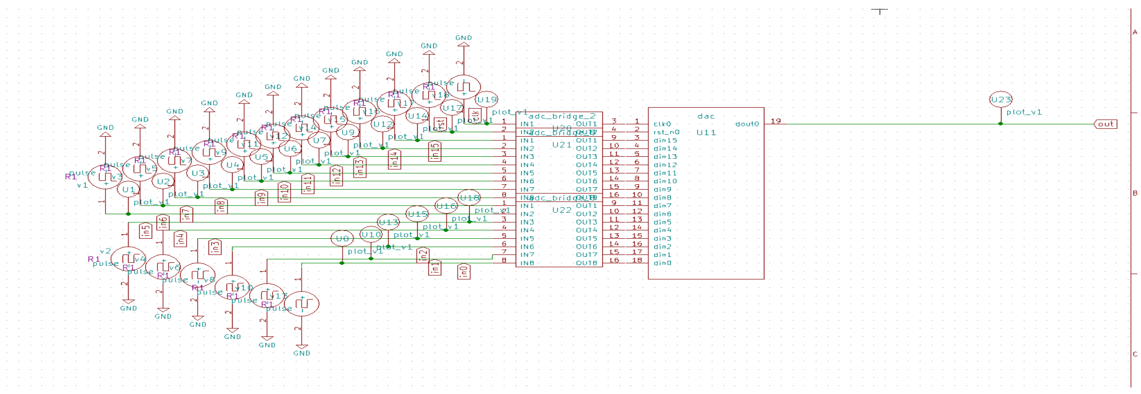


Figure 11.1: DAC Schematic Diagram

11.4 Ngspice Plots

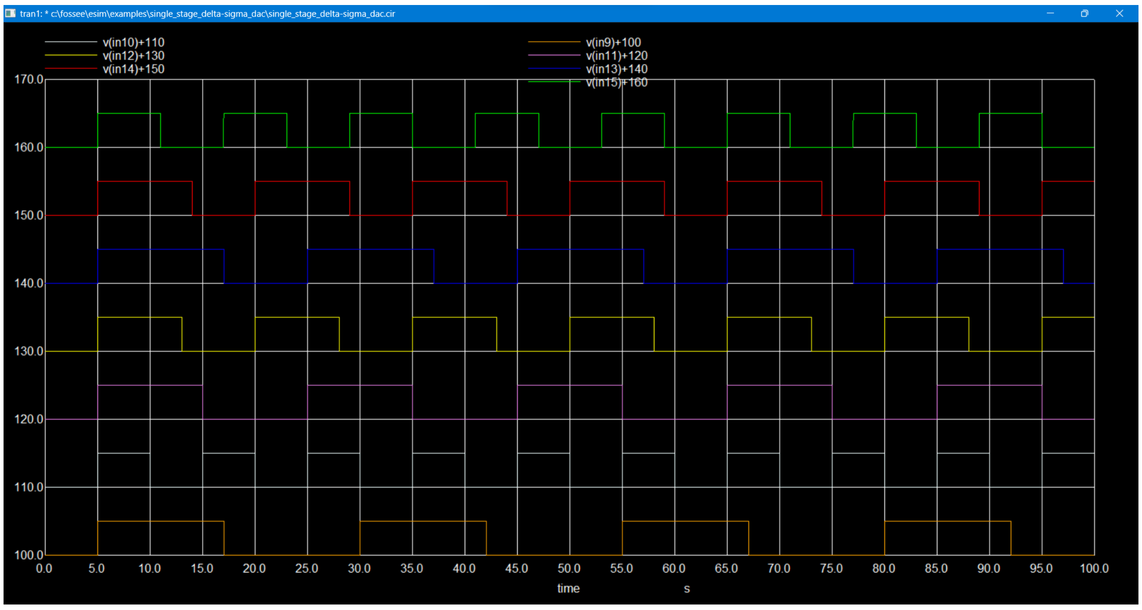


Figure 11.2: DAC INPUT

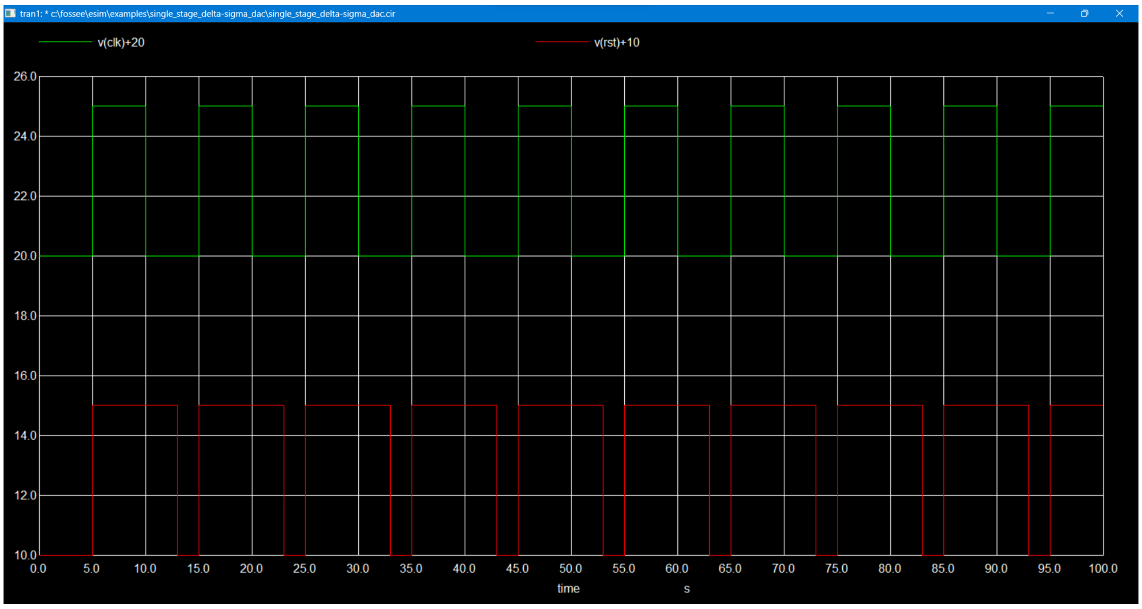


Figure 11.3: CLOCK and RESET PLOT

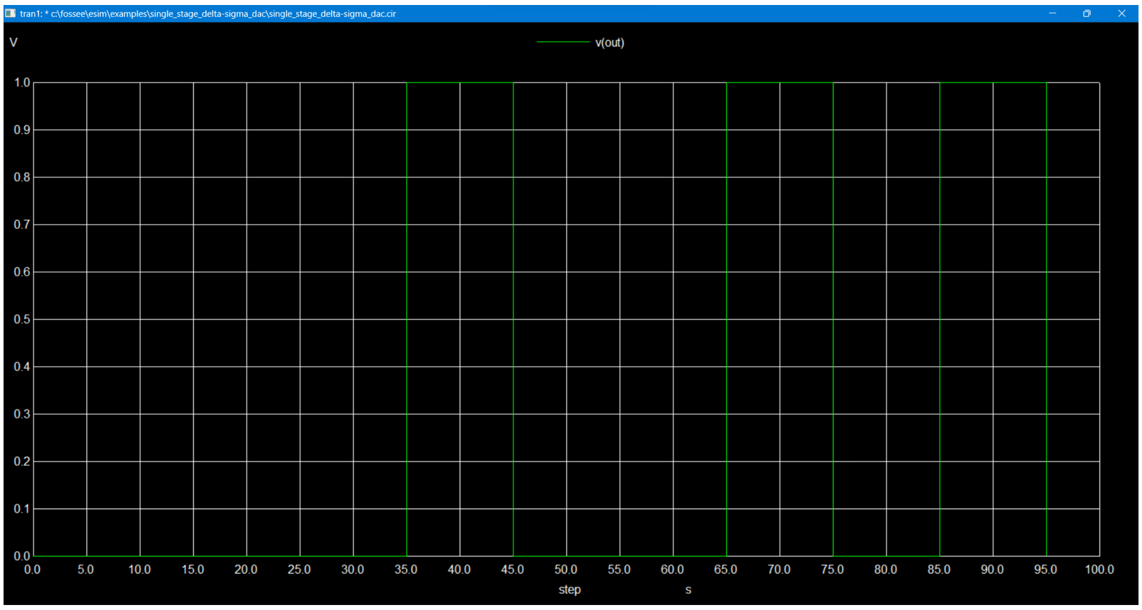


Figure 11.4: DAC OUTPUT

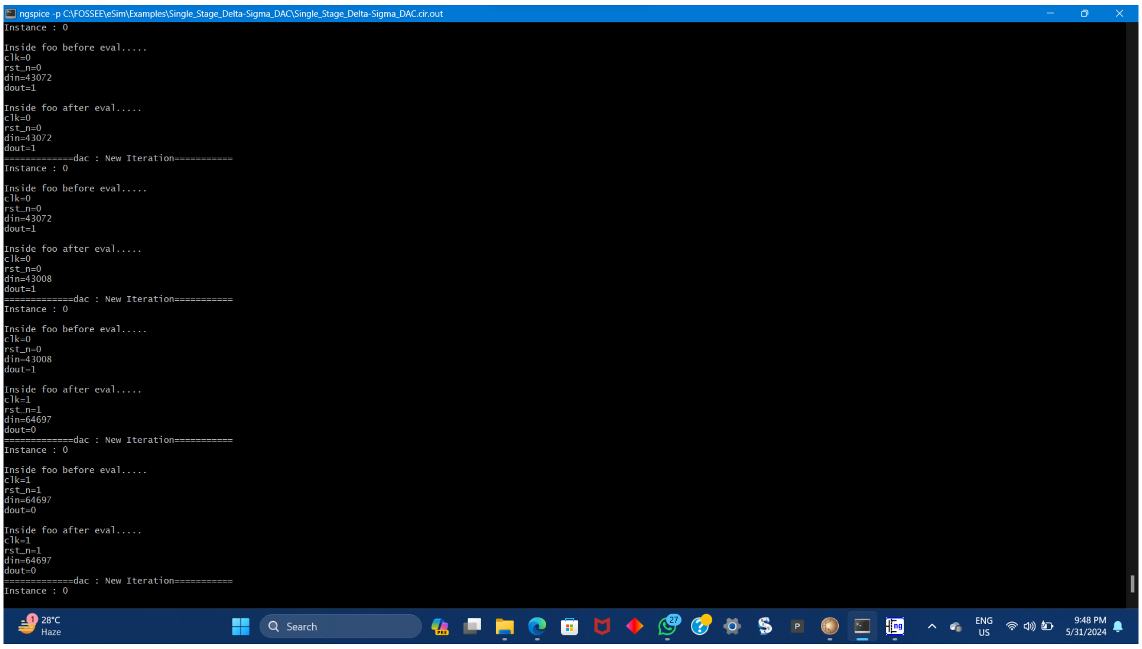

```
ngspice -p C:\FOSSEE\eSim\Examples\Single_Stage_Delta-Sigma_DAC\Single_Stage_Delta-Sigma_DAC.cir.out
Instance : 0

Inside foo before eval.....
clk=0
rst_n=0
din=43072
dout=1

Inside foo after eval.....
clk=0
rst_n=0
din=43072
dout=1
=============dac : New Iteration===========
Instance : 0

Inside foo before eval.....
clk=0
rst_n=0
din=43072
dout=1

Inside foo after eval.....
clk=0
rst_n=0
din=43008
dout=1
=============dac : New Iteration===========
Instance : 0

Inside foo before eval.....
clk=0
rst_n=0
din=43008
dout=1

Inside foo after eval.....
clk=1
rst_n=1
din=64697
dout=0
=============dac : New Iteration===========
Instance : 0

Inside foo before eval.....
clk=1
rst_n=1
din=64697
dout=0

Inside foo after eval.....
clk=1
rst_n=1
din=64697
dout=0
=============dac : New Iteration===========
Instance : 0
```

Figure 11.5: NGSPICE LOG

Chapter 12

Digital Calculator

12.1 Circuit Details

The digital calculator performs arithmetic operations on two 8-bit inputs (a and b) based on a 2-bit operation selector (opt). It supports addition, absolute subtraction, multiplication, and division.It's capable of performing four distinct arithmetic operations on two 8-bit inputs (a and b) based on a 2-bit operation selector (opt). The operations include addition (opt = 2'b00), absolute subtraction (—a-b— when opt = 2'b01), multiplication (opt = 2'b10), and division (opt = 2'b11). The result of these operations is stored in a 16-bit output register (result).

12.2 Verilog Code

```
module digital_calculator(result,a,b,opt);
input [7:0]a,b;// two 8 bit inputs
input [1:0]opt;
output reg [15:0]result;// one 16 bit output, assuming the multiplcation result
 ↪  is maximum of 16bits

always@(a,b)
begin
case(opt)
2'b00:result=a+b;//addition
2'b01:begin// modulo subtraction |a-b|
        if(a>b)
            result=a-b;
        else
            result=b-a;
      end
2'b10:result=a*b;//multiplication
2'b11:result=a/b;//divison
endcase
```

```
end
endmodule
```

12.3 Schematic Diagram

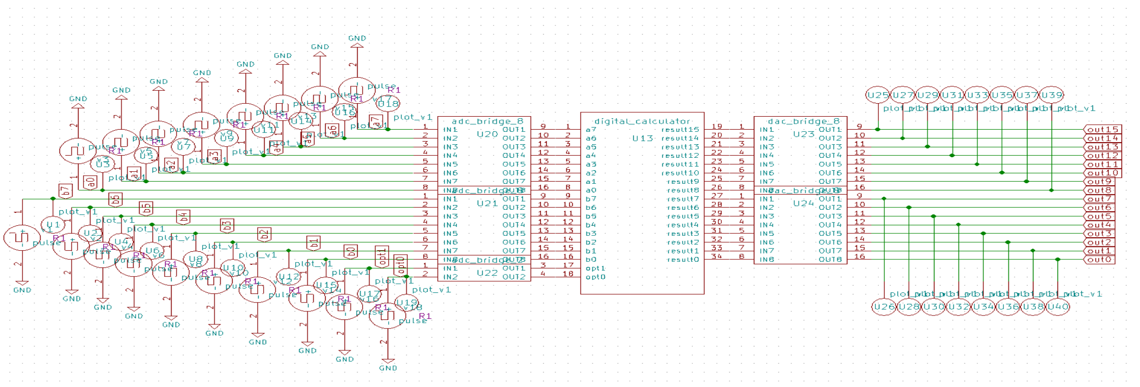


Figure 12.1: Digital calculator

12.4 Ngspice Plots

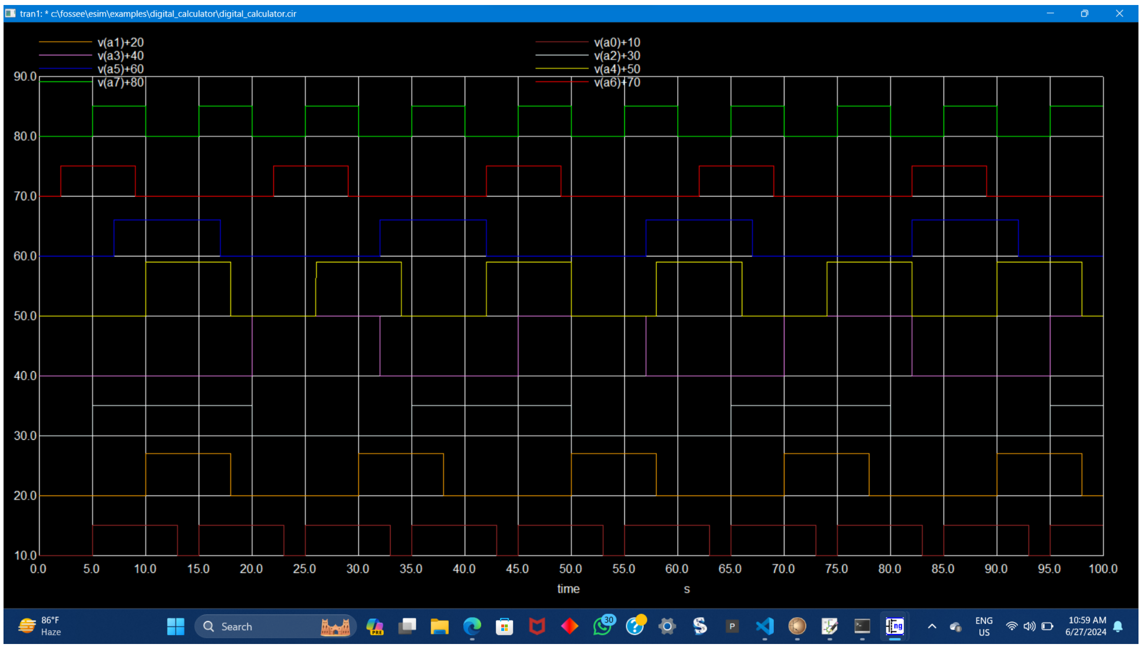


Figure 12.2: INPUT-1

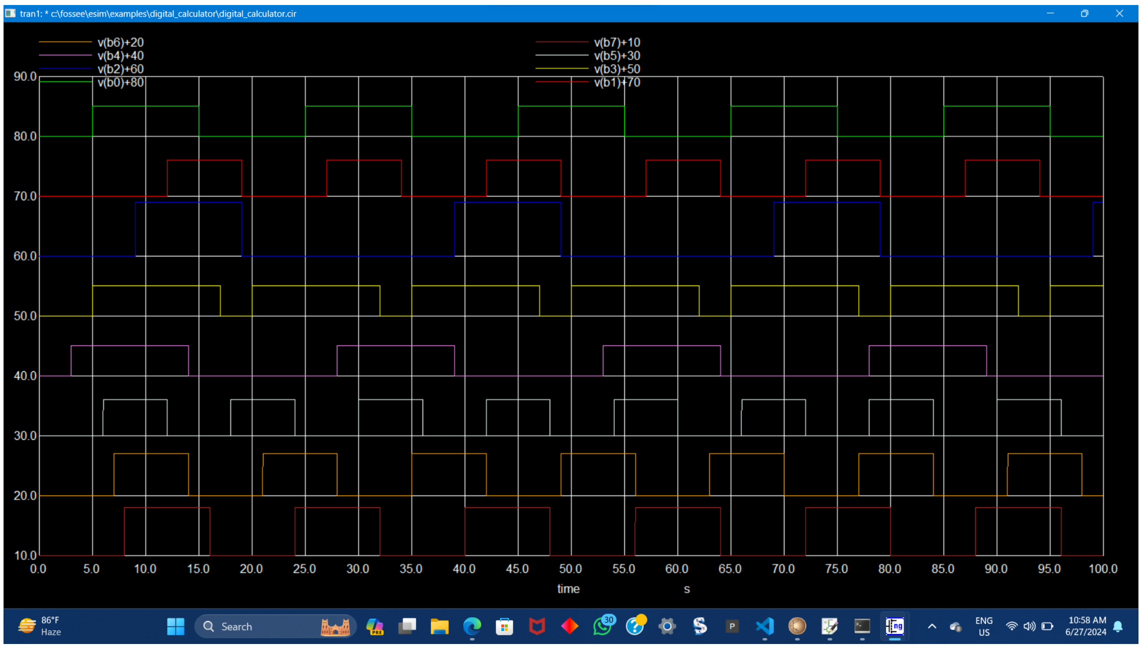


Figure 12.3: INPUT-2

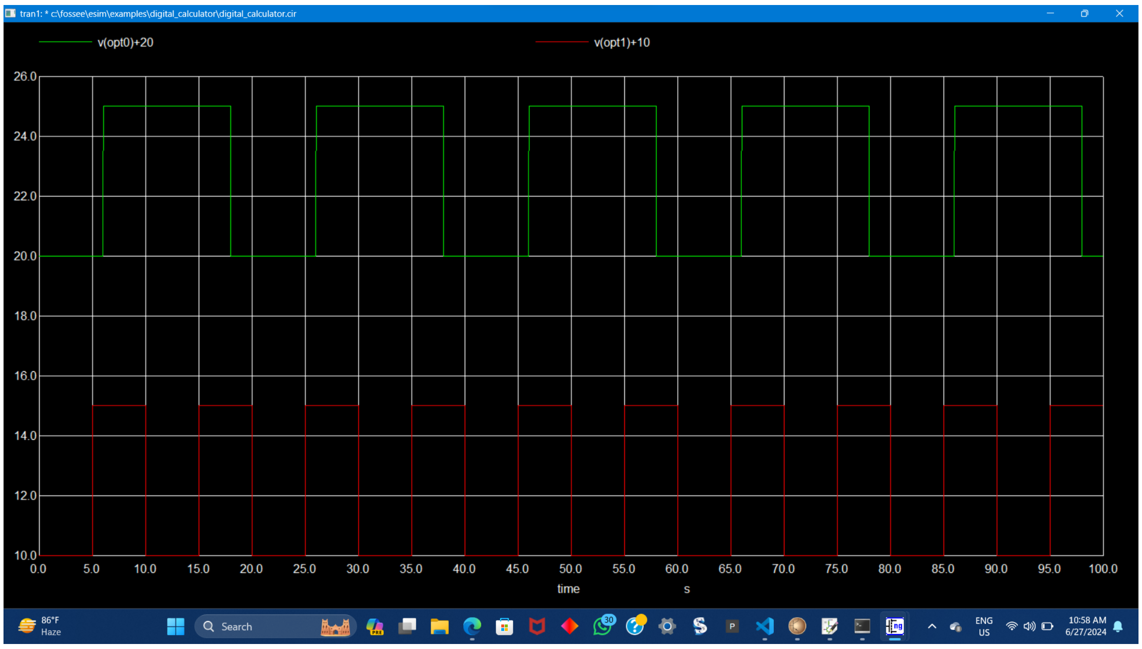


Figure 12.4: Operation Selector

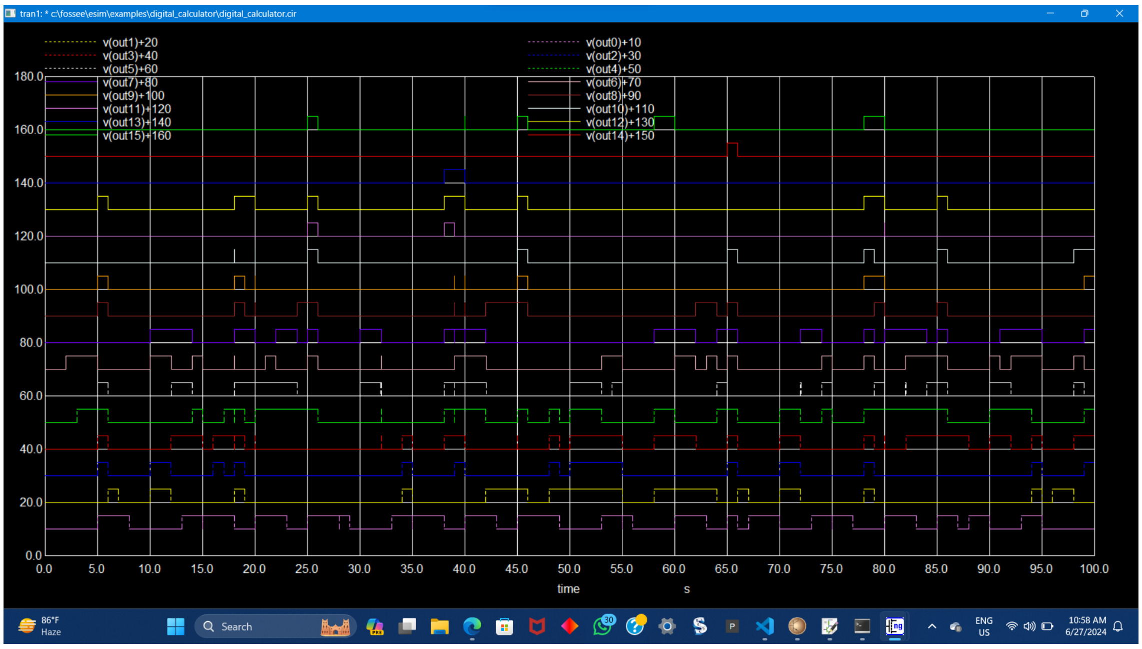


Figure 12.5: OUTPUT

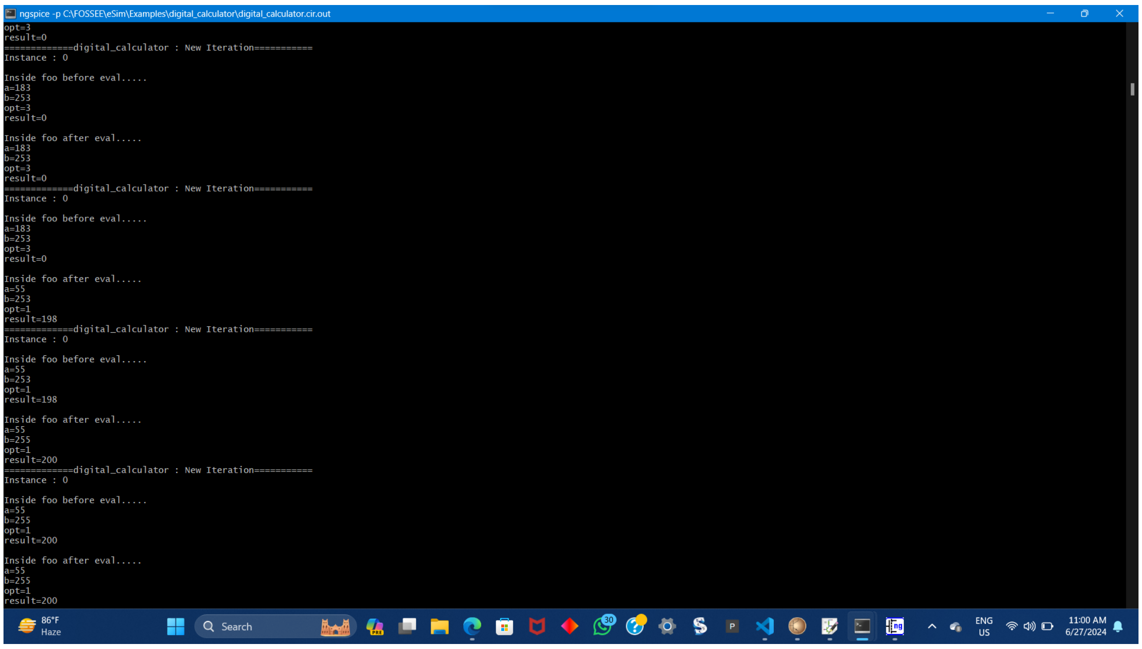
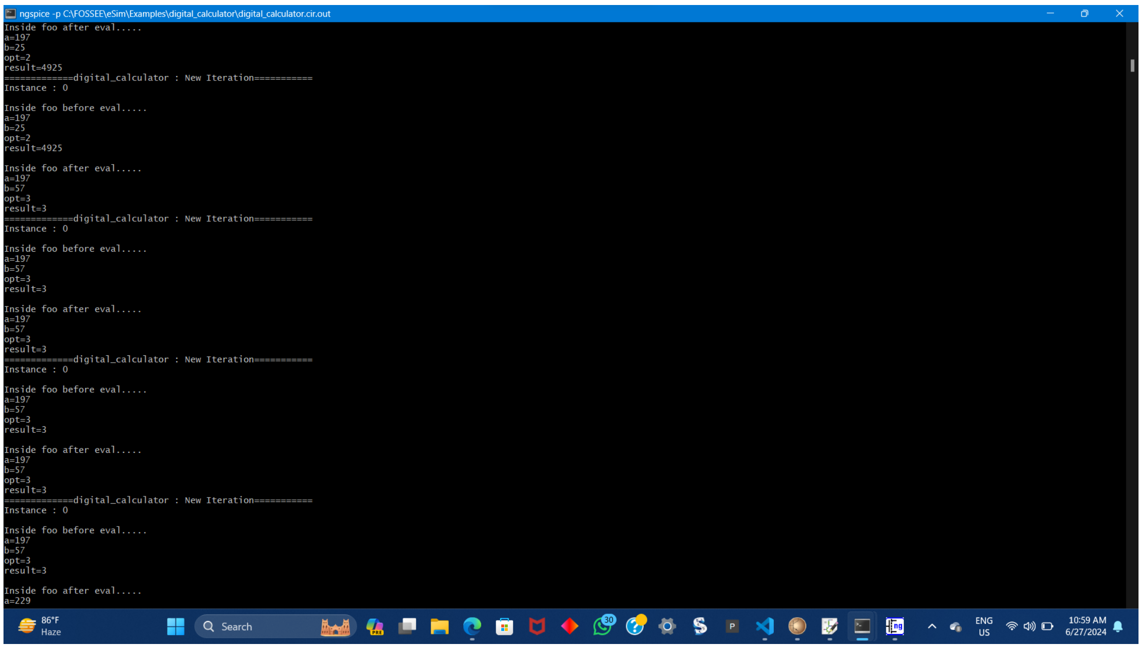
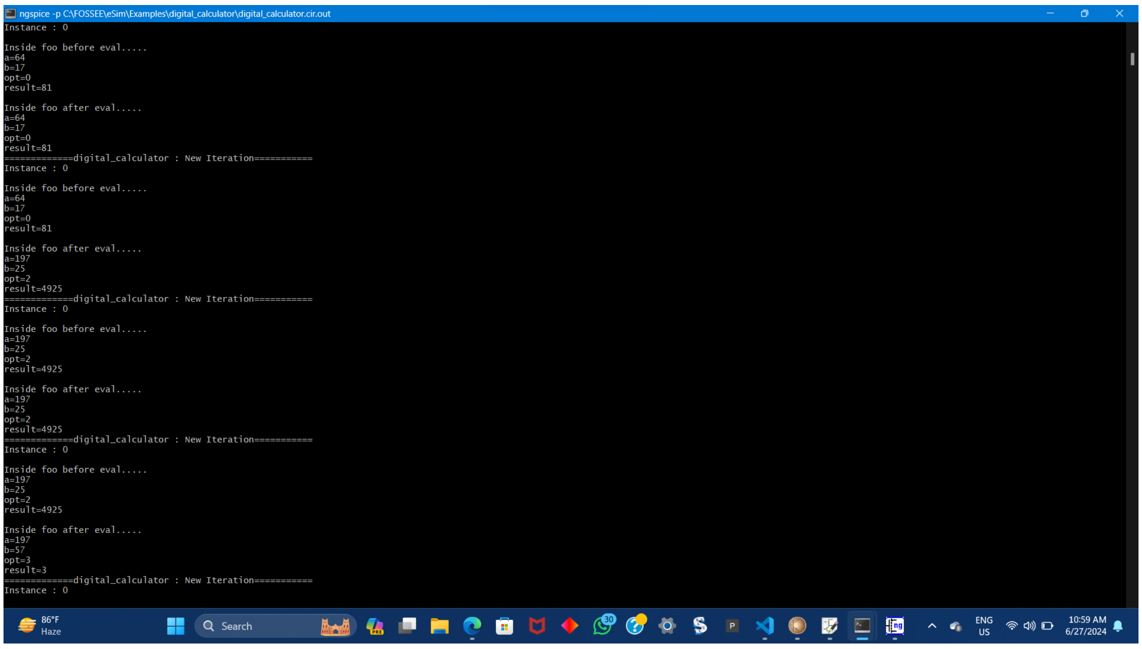

```
ngspice -p C:\FOSSEE\eSim\Examples\digital_calculator\digital_calculator.cir.out
opt=3
result=0
=============digital_calculator : New Iteration===========
Instance : 0

Inside foo before eval.....
a=183
b=253
opt=3
result=0

Inside foo after eval.....
a=183
b=253
opt=3
result=0
=============digital_calculator : New Iteration===========
Instance : 0

Inside foo before eval.....
a=183
b=253
opt=3
result=0

Inside foo after eval.....
a=55
b=253
opt=1
result=198
=============digital_calculator : New Iteration===========
Instance : 0

Inside foo before eval.....
a=55
b=253
opt=1
result=198

Inside foo after eval.....
a=55
b=255
opt=1
result=200
=============digital_calculator : New Iteration===========
Instance : 0

Inside foo before eval.....
a=55
b=255
opt=1
result=200

Inside foo after eval.....
a=55
b=255
opt=1
result=200
```

Figure 12.6: NgSpice Log


ngspice -p C:\FOSSEE\eSim\Examples\digital_calculator\digital_calculator.cir.out
86°F
Haze
Search
ENG
US
10:59 AM
6/27/2024

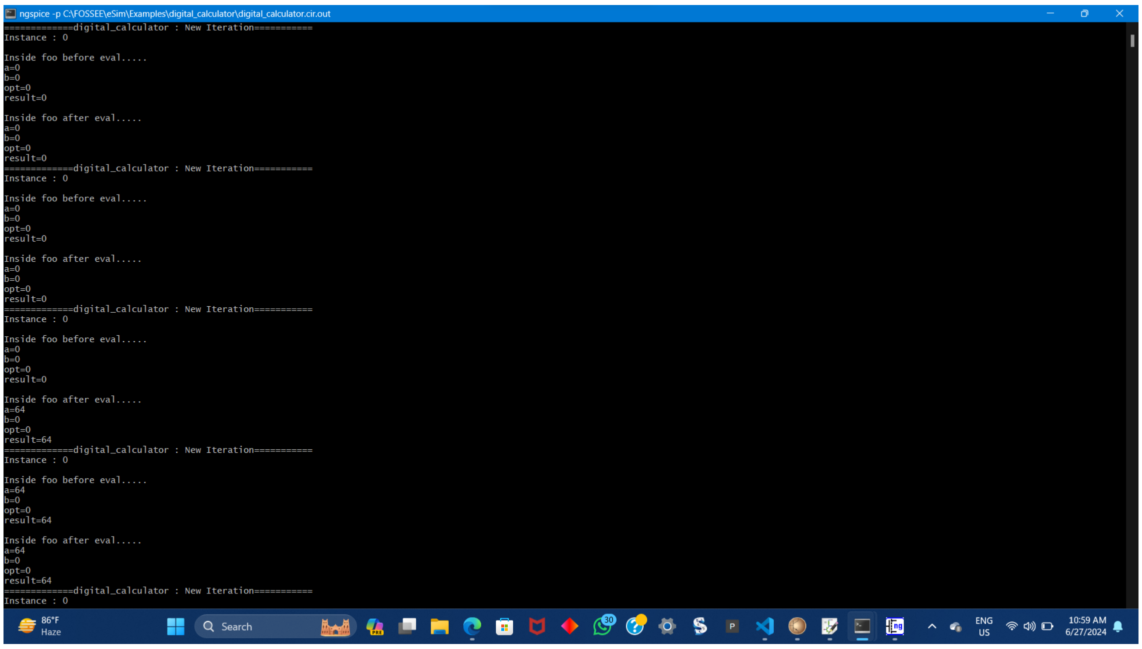
```
Inside foo after eval.....
a=197
b=25
opt=2
result=4925
============digital_calculator : New Iteration==========
Instance : 0

Inside foo before eval.....
a=197
b=25
opt=2
result=4925

Inside foo after eval.....
a=197
b=57
opt=3
result=3
============digital_calculator : New Iteration==========
Instance : 0

Inside foo before eval.....
a=197
b=57
opt=3
result=3

Inside foo after eval.....
a=197
b=57
opt=3
result=3
============digital_calculator : New Iteration==========
Instance : 0

Inside foo before eval.....
a=197
b=57
opt=3
result=3

Inside foo after eval.....
a=197
b=57
opt=3
result=3
============digital_calculator : New Iteration==========
Instance : 0

Inside foo before eval.....
a=197
b=57
opt=3
result=3

Inside foo after eval.....
a=229
```

Figure 12.7: NgSpice Log


ngspice -p C:\FOSSEE\eSim\Examples\digital_calculator\digital_calculator.cir.out
86°F
Haze
Search
ENG
US
10:59 AM
6/27/2024

```
Instance : 0

Inside foo before eval.....
a=64
b=17
opt=0
result=81

Inside foo after eval.....
a=64
b=17
opt=0
result=81
============digital_calculator : New Iteration==========
Instance : 0

Inside foo before eval.....
a=64
b=17
opt=0
result=81

Inside foo after eval.....
a=197
b=25
opt=2
result=4925
============digital_calculator : New Iteration==========
Instance : 0

Inside foo before eval.....
a=197
b=25
opt=2
result=4925

Inside foo after eval.....
a=197
b=25
opt=2
result=4925
============digital_calculator : New Iteration==========
Instance : 0

Inside foo before eval.....
a=197
b=25
opt=2
result=4925

Inside foo after eval.....
a=197
b=57
opt=3
result=3
============digital_calculator : New Iteration==========
Instance : 0
```

Figure 12.8: NgSpice Log



```
ngspice -p C:\FOSSEE\eSim\Examples\digital_calculator\digital_calculator.cir.out
=============digital_calculator : New Iteration===========
Instance : 0

Inside foo before eval.....
a=0
b=0
opt=0
result=0

Inside foo after eval.....
a=0
b=0
opt=0
result=0
=============digital_calculator : New Iteration===========
Instance : 0

Inside foo before eval.....
a=0
b=0
opt=0
result=0

Inside foo after eval.....
a=0
b=0
opt=0
result=0
=============digital_calculator : New Iteration===========
Instance : 0

Inside foo before eval.....
a=0
b=0
opt=0
result=0

Inside foo after eval.....
a=64
b=0
opt=0
result=64
=============digital_calculator : New Iteration===========
Instance : 0

Inside foo before eval.....
a=64
b=0
opt=0
result=64

Inside foo after eval.....
a=64
b=0
opt=0
result=64
=============digital_calculator : New Iteration===========
Instance : 0
```

Figure 12.9: NgSpice Log

Bibliography

[1] FOSSEE Official Website.
URL: https://fossee.in/about

[2] Implementation of Basic Logic Gates using VHDL in ModelSim in Circuit Digest Website by Raghul Saravanan. 2020.
URL: https://circuitdigest.com/microcontroller-projects/implementation-of-basic-logic-gates-using-vhdl-in-modelsim

[3] Wikipedia Official Website. 2020.
URL: https://www.geeksforgeeks.org/ripple-counter-in-digital-logic/

[4] Verilog Coding Tips and Tricks.
URL: https://verilogcodes.blogspot.com/2015/10/verilog-code-for-bcd-to-7-segment.html

[5] eSim Official website. 2020.
URL: https://esim.fossee.in/

[6] tanmay-mohapatra Flipflop Verilog github Repository.
URL: https://github.com/tanmay-mohapatra/Flipflop_Verilog/blob/main

[7] Vedant-02 Verilog-HDL-Lab-Experiments Github Repository.
URL: https://github.com/Vedant-02/Verilog-HDL-Lab-Experiments/blob/main/Binary/20to/20Gray/20Code/20Converter/bin_gray.v

[8] Github FOSSEE NGHDL Repository.
URL: https://github.com/FOSSEE/nghdl

[9] Obijuan github Repository.
URL: https://github.com/Obijuan/Z80-FPGA/tree/master/TV80-verilog

[10] roboticvedant github Repository.
URL: https://github.com/roboticvedant/Verilog-PID-Controller/blob/main/PID_controller/PID_controller.srcs/sources_1/new/pid.v

[11] briansune github Repository.
URL: https://github.com/briansune/Delta-Sigma-DAC-Verilog/blob/main/hdl/dsa_single.v

[12] Single-Port Memory by Aditya Mathur.
URL: https://www.linkedin.com/pulse/single-port-memory-aditya-mathur/

[13] ekb0412 github Repository.
URL: https://github.com/ekb0412/100DaysofRTL/blob/main/Day080-/20Single-port/20RAM/single_port_ram.v

[14] eSim winter internship reports.
URL: https://static.fossee.in/fossee/winter-intership-2023/reports/eSim

[15] armaan-says github Repository - DIGITAL-SYSTEM-DESIGN
URL: https://github.com/armaan-says/DIGITAL-SYSTEM-DESIGN