Winter Internship Report

On

Digital and Mixed Signal Circuits in eSim

Submitted by

Amisha Shyam Sakhare

Under the guidance of

Prof. Kannan M. Moudgalaya

Chemical Engineering Department

IIT Bombay

April 12, 2023

Acknowledgment

I am extremely thankful to the FOSSEE team for providing me with an incredible and educational opportunity. Their support and guidance have made a significant impact on my learning journey. I am grateful for the valuable experience they have given me.

I am deeply grateful to the FOSSEE Team at IIT Bombay for entrusting me with this project and believing in my capabilities. This opportunity has been invaluable to my growth and development, and I sincerely appreciate their support.

I would like to express my heartfelt appreciation to my mentors, Mr. Sumanto Kar, Mr. Rahul Paknikar, Mrs. Madhuri Kadam, Prof. Inderjit Singh Dhanjal, Mrs. Usha Vishwanathan, and the entire team. Their unwavering support and invaluable guidance throughout my internship have been instrumental in my growth. I am truly grateful for their wealth of knowledge and constructive suggestions that have greatly enhanced my learning experiences.

I am also deeply appreciative of my fellow friends, whose incredible knowledge and exceptional skills have been invaluable in assisting me. The energetic and friendly work environment has fostered collaboration and support.

I am committed to utilizing everything I have gained here for my personal and professional growth, as well as for the advancement of our society.

Contents

1 Introduction

1.1 eSim

eSim is a free/libre and open source Electronic Design Automation (EDA) tool developed by FOSSEE (Free and Open Source Software for Education) at IIT Bombay. It provides a comprehensive platform for circuit design, simulation, analysis, and PCB (Printed Circuit Board) design.

eSim is built using various free/libre and open source software components, including:

1. KiCad: A popular EDA suite that offers schematic capture and PCB layout tools.
2. Ngspice: A mixed-level/mixed-signal circuit simulator that can perform analog, digital, and mixed-signal simulations.
3. NGHDL: An open source VHDL simulator that enables simulation and analysis of digital circuits.
4. GHDL: Another open source VHDL simulator that supports the IEEE 1076 VHDL standard.

1.2 Ngspice

ngspice is an open-source simulation program for electric and electronic circuits based on the SPICE (Simulation Program with Integrated Circuit Emphasis) simulation engine. It provides a powerful platform for simulating a wide range of circuits, including analog, digital, and mixed-signal circuits.

Some key features and capabilities of ngspice:

1. Component Support: ngspice supports a variety of components and devices, including JFETs, bipolar and MOS transistors, passive elements such as resistors (R), inductors (L), and capacitors (C), diodes, transmission lines, and more. These elements can be interconnected in a circuit using a netlist.
2. Mixed-Signal Simulation: ngspice allows you to simulate mixed-signal circuits, which combine both analog and digital components. This enables the analysis of complex systems that involve both continuous and discrete signals.
3. Comprehensive Device Models: ngspice provides a wide range of device models for active and passive components, covering both analog and digital elements. These models are sourced from various collections, semiconductor manufacturers, and semiconductor foundries. They include accurate descriptions of device behavior and characteristics.
4. Graphical Outputs and Data Logging: The simulation results in ngspice can be visualized through graphs showing currents, voltages, and other electrical quantities. Additionally, the simulation data can be saved in a data file for further analysis and processing.
5. Event-Driven Simulation: ngspice utilizes an event-driven simulation approach, which ensures efficient and fast simulation of digital circuits. It can handle circuits ranging from simple gates to complex digital systems.

1.3 Makerchip

Makerchip is a platform that offers convenient and accessible access to various tools for digital circuit design. It provides both browser-based and desktop-based environments for coding, compiling, simulating, and debugging Verilog designs. Makerchip supports a combination of open-source tools and proprietary ones, ensuring a comprehensive range of capabilities.

Here are some key features of Makerchip:

1. Browser-Based Environment: Makerchip allows users to perform Verilog design tasks directly from their web browser. This eliminates the need for local installations and provides flexibility in accessing the tools from any device with internet access.

2. Code, Compile, Simulate, and Debug: Users can write Verilog code, compile it into a circuit representation, simulate the behavior of the design, and debug any issues directly within the Makerchip environment. This integrated workflow streamlines the design process, reducing the need for switching between different tools.

3. Seamless Integration: Makerchip offers tight integration between code, block diagrams, waveforms, and novel visualization capabilities. This integration enhances the design experience by providing a cohesive and intuitive interface for designing, visualizing, and analyzing circuits.

4. Advanced Verilog Design Capabilities: Makerchip introduces innovative features and capabilities for advanced Verilog design. It incorporates ground-breaking functionalities that facilitate complex digital circuit design tasks. These capabilities empower users to tackle sophisticated design challenges effectively.

5. User-Friendly Design Experience: Makerchip aims to make circuit design easy and enjoyable for users of all skill levels. The platform provides a user-friendly interface, intuitive workflows, and a range of helpful features that simplify the design process and enhance the overall user experience.

2 Features of eSim

1. Circuit Design and Schematic Capture: eSim provides a user-friendly interface for designing electronic circuits. It offers a schematic capture tool where users can create circuit diagrams by placing and connecting components.

2. Component Libraries: eSim includes extensive libraries of electronic components, such as resistors, capacitors, inductors, transistors, diodes, and integrated circuits (ICs). These libraries help users easily access and integrate components into their designs.

3. Symbol and Footprint Creation: Users can create custom symbols and footprints for components that are not available in the existing libraries. This allows for the inclusion of specialized or unique components in circuit designs.

4. Circuit Simulation: eSim integrates powerful simulation engines, such as Ngspice, to simulate the behavior of electronic circuits. It supports analog, digital, and mixed-signal simulations, enabling comprehensive analysis of circuit performance.

5. Waveform Viewer: The tool provides a waveform viewer that allows users to visualize simulation results in the form of waveforms. This helps in analyzing and understanding circuit behavior, including voltage levels, currents, and signal timing.

6. PCB Design and Layout: eSim seamlessly integrates with KiCad, a popular open-source PCB design tool. Users can transfer their circuit designs from eSim to KiCad for further PCB layout and routing.

7. Interactive Simulation and Analysis: eSim allows users to interactively probe and analyze circuit parameters during simulation. This feature facilitates real-time monitoring and evaluation of circuit performance.

8. Open Source and Customization: eSim is built using free/libre and open-source software, providing users with the freedom to modify and customize the tool according to their specific requirements. It encourages collaboration and community-driven development.

9. Educational Resources: eSim is developed with a focus on education. It provides educational resources like tutorials, documentation, and example circuits to help users learn and understand various aspects of circuit design and simulation.

10. Cross-Platform Support: eSim is designed to work on multiple operating systems, including Windows, Linux, and macOS, ensuring accessibility for users across different platforms.

3 Problem Statement

The objective of this internship is to implement digital and mixed signal circuits in eSim using NgVeri, a tool that converts Verilog models into NgSpice. This will enable users to simulate these circuits as examples within eSim.

3.1 Approach

The general approach I followed to implement the mixed and digital circuit for this project was as follows:

1. The list of circuits to be implemented during this internship has been finalized.
2. Each individual Verilog file of the circuit design was simulated in Makerchip to ensure correct functionality. Then, the Verilog files were fed into NgVeri to verify their conversion without any errors.
3. Simulations were performed for each individual Verilog file, and the generated Ngspice waveform was analyzed to validate that the desired waveform was obtained. This step ensured the correct behavior of each component of the mixed and digital circuit.
4. The final circuit design (schematic), including all the components was created.
5. The complete mixed and digital circuit design was simulated in Ngspice to evaluate the overall performance and behavior. This step provided a comprehensive assessment of the circuit's functionality and verified the successful implementation of the mixed and digital circuit.

4 Barrel Shifter

4.1 Circuit Details

The barrel shifter is a digital circuit implemented using pure combinational logic, which enables shifting a data word by a specified number of bits. It eliminates the need for sequential logic elements, making it a highly efficient and fast solution for shifting operations.

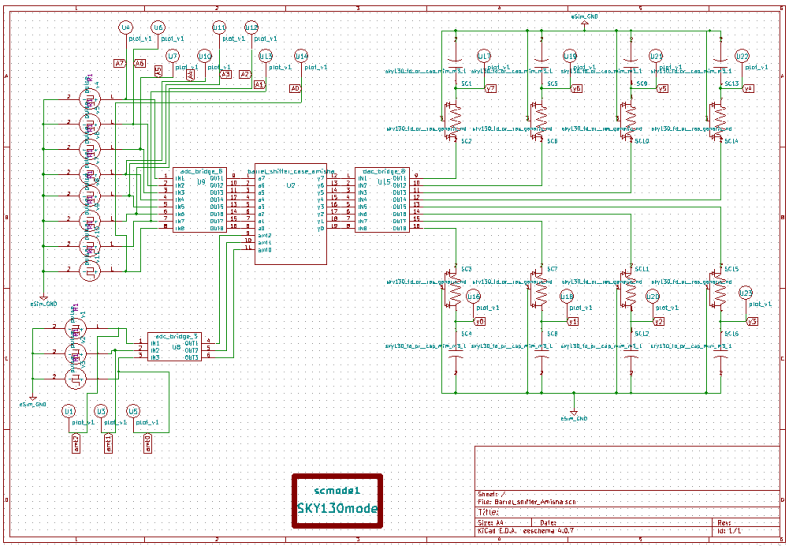
The circuit design employs a parallel structure, allowing simultaneous shifting of multiple bits in parallel. By utilizing combinational logic elements, the barrel shifter achieves a high-speed data shifting capability while maintaining simplicity in its architecture.

This circuit serves as a crucial component in digital systems and processors where efficient data manipulation and rearrangement are required.

4.2 Verilog Code

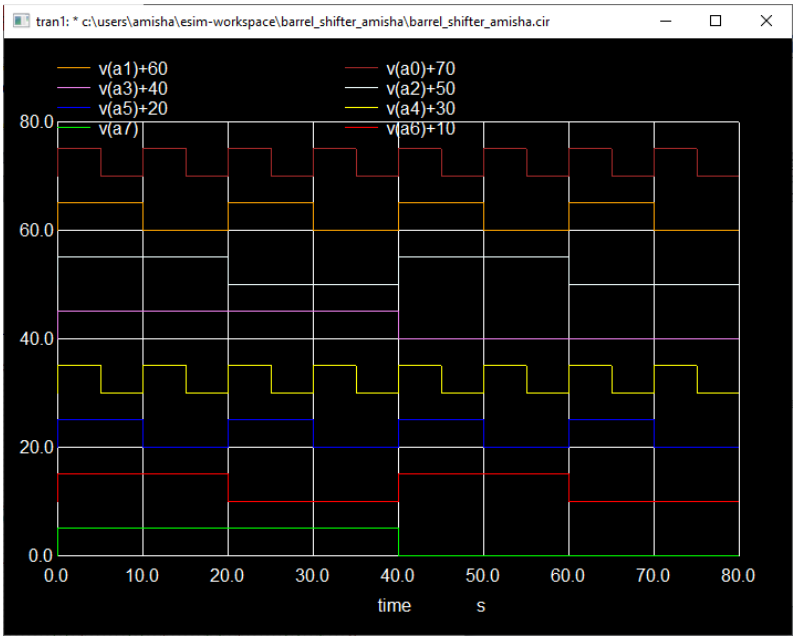
```
module barrel_shifter_case_Amisha(
    input [7:0] a,
    input [2:0] amt,
    output reg [7:0] y
    );
    always @*
    case (amt)
        3'o0: y = a;
        3'o1: y = {a[0], a[7:1]};
        3'o2: y = {a[1:0], a[7:2]};
        3'o3: y = {a[2:0], a[7:3]};
        3'o4: y = {a[3:0], a[7:4]};
        3'o5: y = {a[4:0], a[7:5]};
        3'o6: y = {a[5:0], a[7:6]};
        default: y = {a[6:0], a[7]};
    endcase
endmodule
```

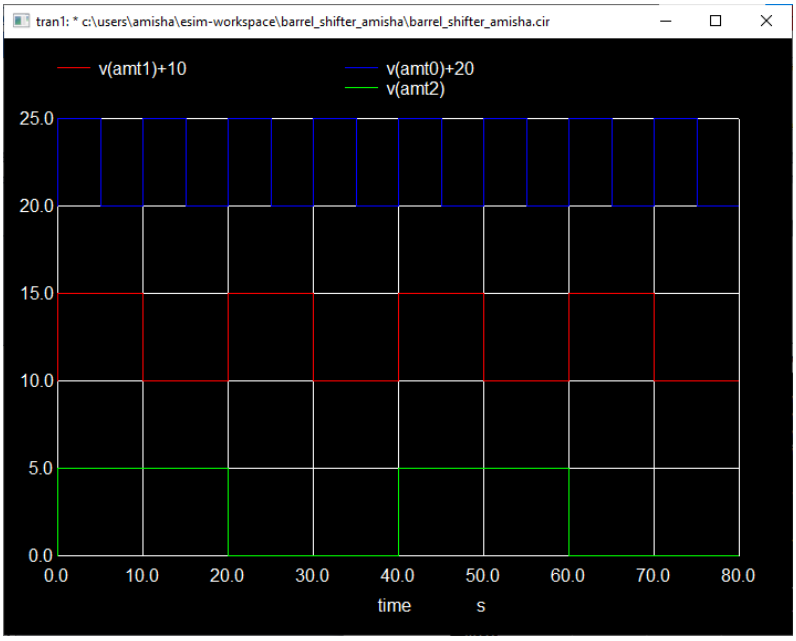
4.3 Schematic Diagram



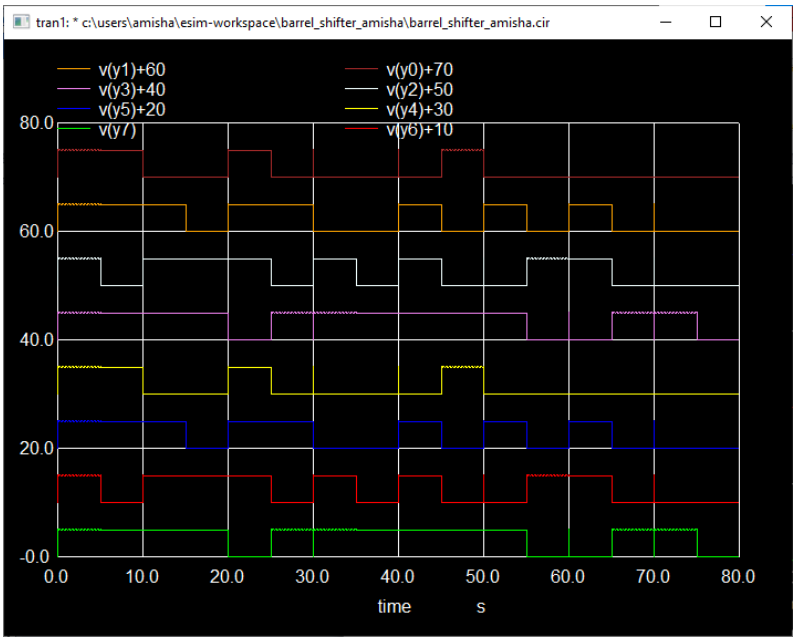
4.4 Ngspice Plots

4.4.1 Input plot





4.4.2 Output plot



5 8-bit Ripple Carry Adder

5.1 Circuit Details

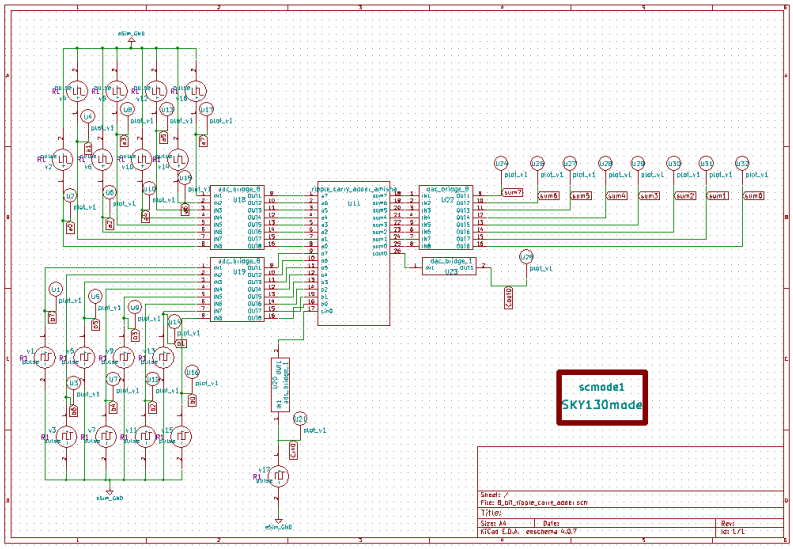
The addition process in an 8-bit ripple-carry adder follows the same principle employed in a 4-bit ripple-carry adder. Each bit from the two input sequences is added together along with the input carry, if any. This principle is extended to perform addition on two 8-bit binary digit sequences.

The 8-bit ripple-carry adder circuit consists of a cascade of full-adder modules, with each module handling the addition of a single bit. The carry output from each full-adder module is propagated to the carry input of the subsequent module, creating a ripple effect throughout the circuit. This ripple propagation can introduce delays in the calculation of the final result.

5.2 Verilog Code

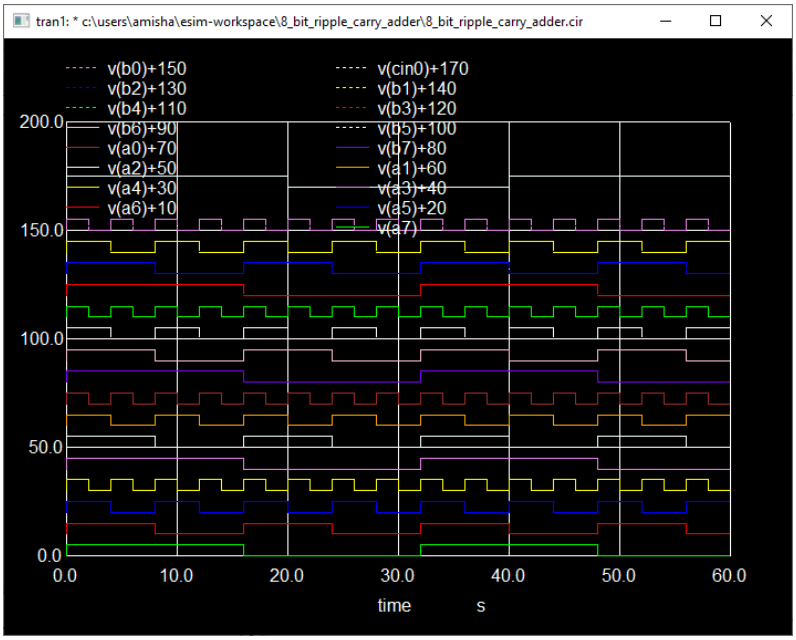
```
module full_adder_Amisha
    (
    input a,
    input b,
    input cin,
    output sum,
    output cout
    );
    assign sum=(a^b^cin);
    assign cout=((a&b)|(b&cin)|(a&cin));
endmodule

module ripple_carry_adder_Amisha(
    input [07:0] a,
    input [07:0] b,
    input cin,
    output [7:0]sum,
    output cout
    );
    wire[6:0] c;
    full_adder_Amisha a1(a[0],b[0],cin,sum[0],c[0]);
    full_adder_Amisha a2(a[1],b[1],c[0],sum[1],c[1]);
    full_adder_Amisha a3(a[2],b[2],c[1],sum[2],c[2]);
    full_adder_Amisha a4(a[3],b[3],c[2],sum[3],c[3]);
    full_adder_Amisha a5(a[4],b[4],c[3],sum[4],c[4]);
    full_adder_Amisha a6(a[5],b[5],c[4],sum[5],c[5]);
    full_adder_Amisha a7(a[6],b[6],c[5],sum[6],c[6]);
    full_adder_Amisha a8(a[7],b[7],c[6],sum[7],cout);
endmodule
```

5.3 Schematic Diagram

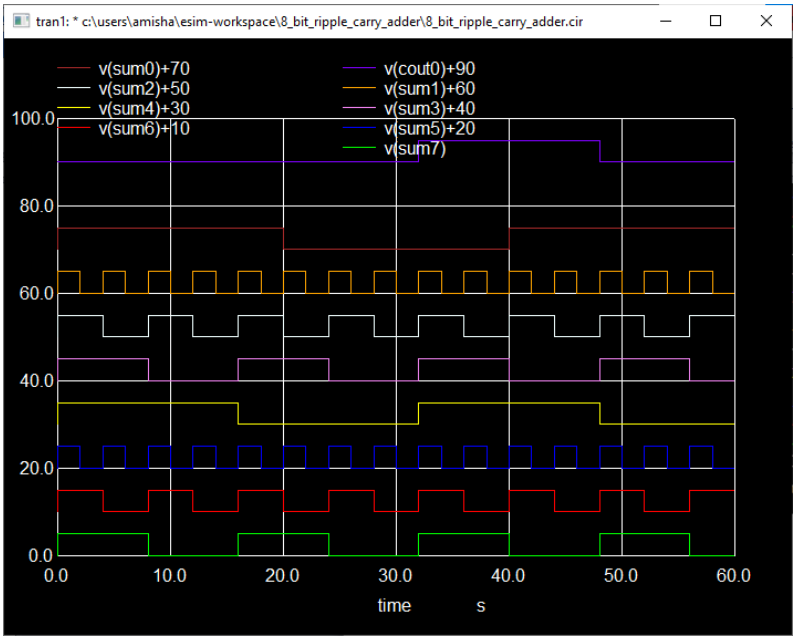


5.4 Ngspice Plots

5.4.1 Input plot



5.4.2 Output plot



6 BCD Adder

6.1 Circuit Details

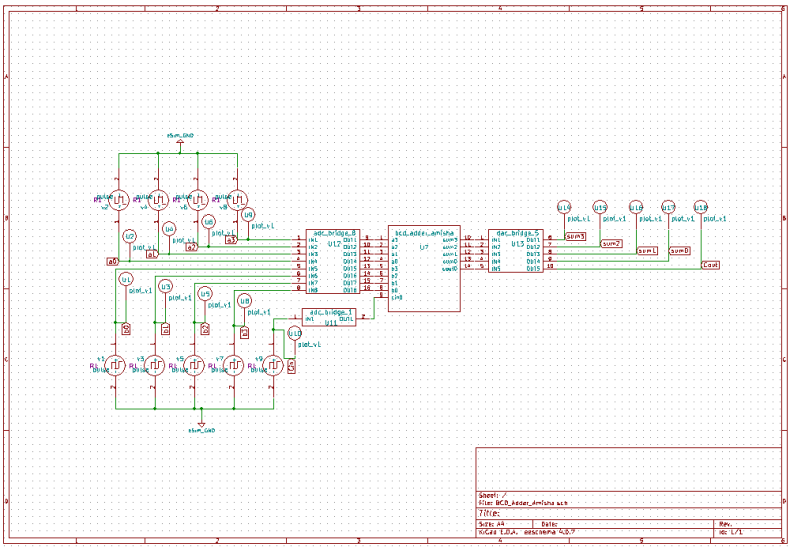
The objective of implementing a BCD adder in this internship is to design a circuit that takes two BCD (Binary Coded Decimal) numbers as inputs and produces a BCD digit as the output, along with a carry output. The BCD adder is designed to handle the scenario where the sum of the BCD digits is greater than 9, requiring conversion into BCD format.

When the sum of the BCD digits is less than or equal to 9, the circuit operates without any issues. However, when the sum exceeds 9, a conversion process is initiated. In this process, 6 is added to the sum, and only the least significant 4 bits are considered. The most significant bit (MSB) of the result is output as the carry.

6.2 Verilog Code

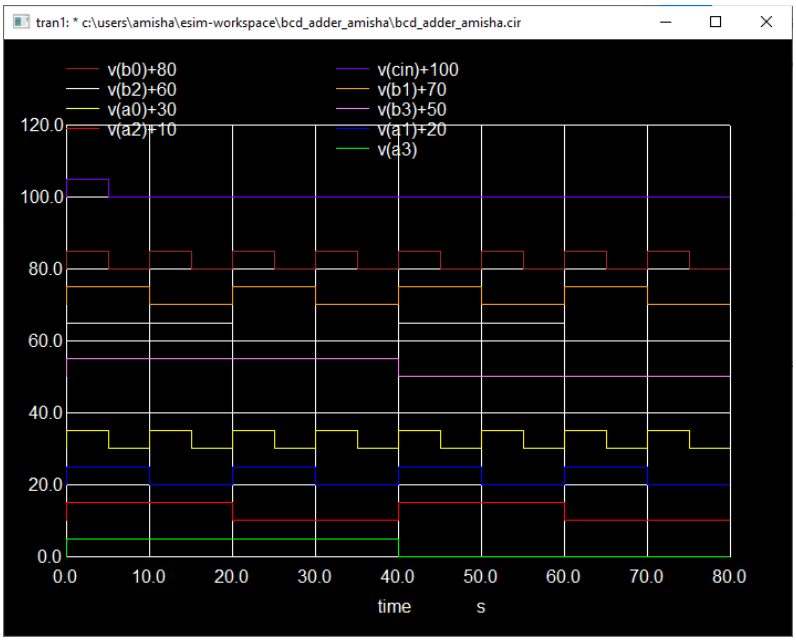
```
module bcd_adder_Amisha(a,b,cin,sum,cout);
    input [3:0] a,b;
    input cin;
    output [3:0] sum;
    output cout;
    reg [4:0] temp;
    reg [3:0] sum;
    reg cout;
    always @(a,b,cin)
    begin
        temp = a+b+cin;
        if(temp > 9)
        begin
            temp = temp+6; //add 6, if result is more than 9.
            cout = 1;  //set the carry output
            sum = temp[3:0];
        end
        else
        begin
            cout = 0;
            sum = temp[3:0];
        end
    end
endmodule
```

6.3 Schematic Diagram

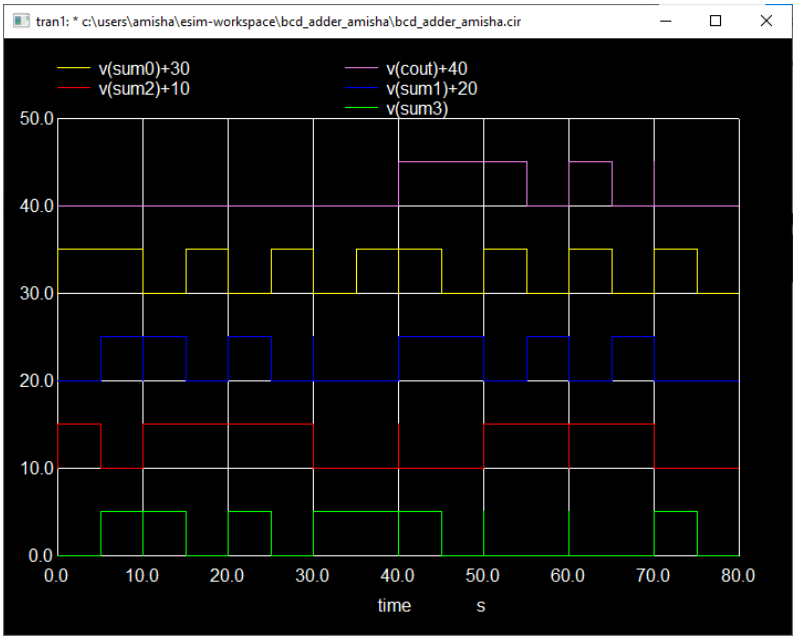


6.4 Ngspice Plots

6.4.1 Input plot



6.4.2 Output plot



7 Booth Multiplier

7.1 Circuit Details

This internship project focuses on implementing the Booth's Multiplication Algorithm for multiplying two signed numbers in a Finite State Machine (FSM) format. The specific implementation will target 3-bit operands with one sign bit.

The implemented circuit will take the following inputs:
clk: Clock signal ; rst: Reset signal ; X, Y: The two operands to be multiplied ; start: A pulse indicating the start of the computation

The outputs of the circuit will be:
Z: Product of the multiplication ; valid: Pulse indicating the availability of the final product

The implementation follows a state-based approach with the following states:

1. IDLE state: The circuit waits for the start pulse, remaining in this state until the pulse is received. Once the start pulse is detected, the circuit transitions to the START state.

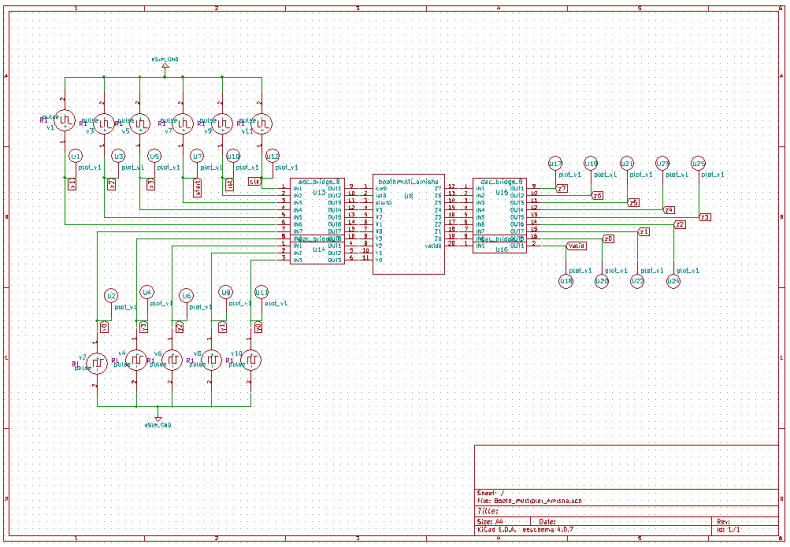
2. START state: Computation takes place within this state, utilizing Verilog logic. The operations are performed iteratively as long as the counter is less than 3 (the chosen operand size). The counter is incremented during each iteration.
   Upon completion of the third iteration (counter = 3), the valid pulse is sent, indicating that the final product is available. The circuit then transitions back to the IDLE state to wait for the next start pulse.

7.2 Verilog Code

```
module BoothMulti_Amisha(clk,rst,start,X,Y,valid,Z);
    input clk;
    input rst;
    input start;
    input signed [3:0]X,Y;
    output signed [7:0]Z;
    output valid;
    reg signed [7:0] Z,next_Z,Z_temp;
    reg next_state, pres_state;
    reg [1:0] temp,next_temp;
    reg [1:0] count,next_count;
    reg valid, next_valid;
    parameter IDLE = 1'b0;
    parameter START = 1'b1;
    always @ (posedge clk or negedge rst)
    begin
        if(!rst)
        begin
            Z          <= 8'd0;
            valid      <= 1'b0;
            pres_state <= 1'b0;
            temp       <= 2'd0;
            count      <= 2'd0;
        end
        else
        begin
            Z          <= next_Z;
            valid      <= next_valid;
```

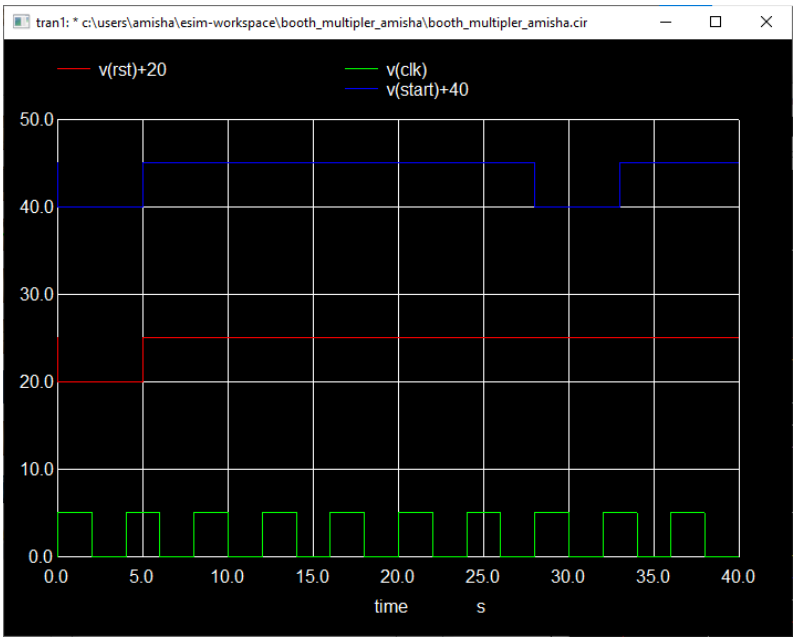
```
            pres_state <= next_state;
            temp       <= next_temp;
            count      <= next_count;
        end
    end

    always @ (*)
    begin
        case(pres_state)
        IDLE:
        begin
            next_count = 2'b0;
            next_valid = 1'b0;
            if(start)
            begin
                next_state = START;
                next_temp  = {X[0],1'b0};
                next_Z     = {4'd0,X};
            end
            else
            begin
                next_state = pres_state;
                next_temp  = 2'd0;
                next_Z     = 8'd0;
            end
        end

        START:
        begin
            case(temp)
                2'b10:   Z_temp = {Z[7:4]-Y,Z[3:0]};
                2'b01:   Z_temp = {Z[7:4]+Y,Z[3:0]};
                default: Z_temp = {Z[7:4],Z[3:0]};
            endcase
            next_temp  = {X[count+1],X[count]};
            next_count = count + 1'b1;
            next_Z     = Z_temp >>> 1;
            next_valid = (&count) ? 1'b1 : 1'b0;
            next_state = (&count) ? IDLE : pres_state;
        end
        endcase
    end
endmodule
```

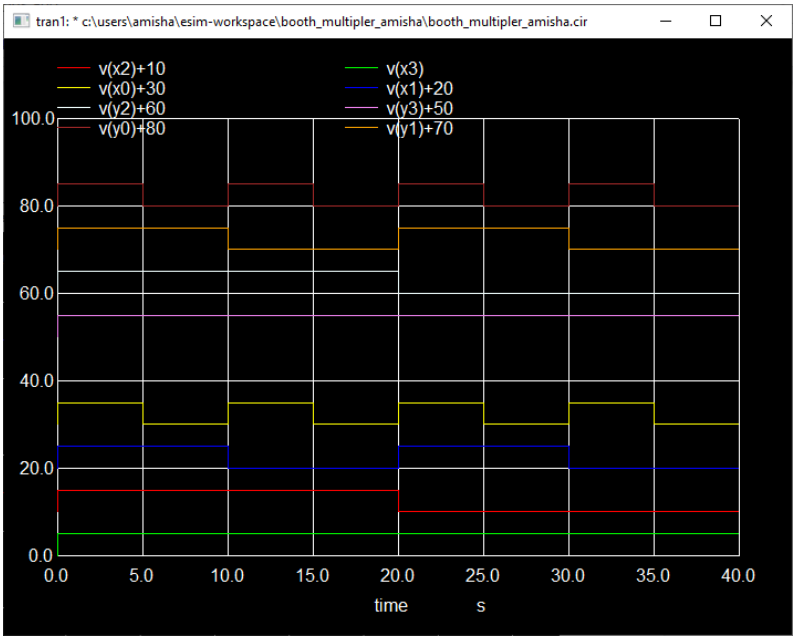
7.3 Schematic Diagram



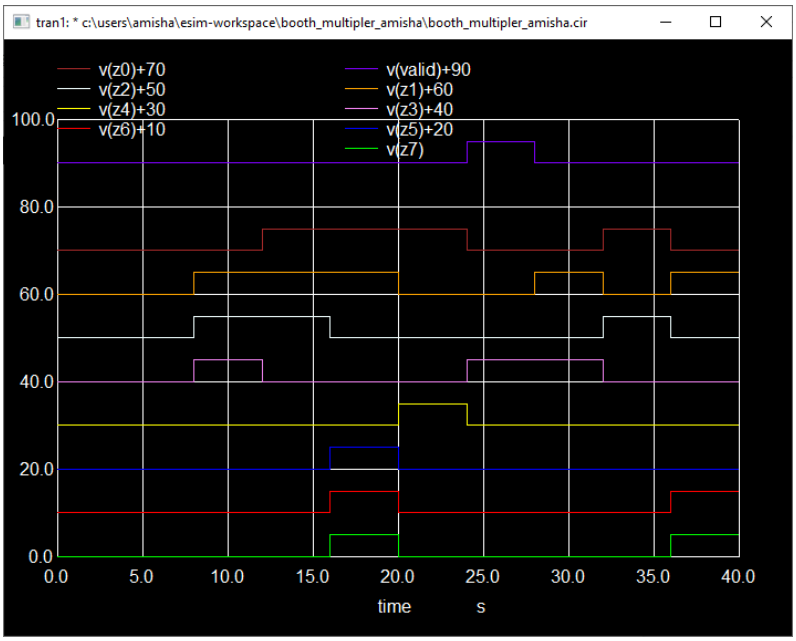
7.4 Ngspice Plots

7.4.1 Input plot





7.4.2 Output plot



8 Comparator

8.1 Circuit Details

The comparator circuit is be designed to compare the values of A and B bit by bit. It will determine the relationship between A and B and set the output signals accordingly.

The specifications for the comparator circuit are as follows:
Inputs:

- A: 2-bit input for comparison
- B: 2-bit input for comparison

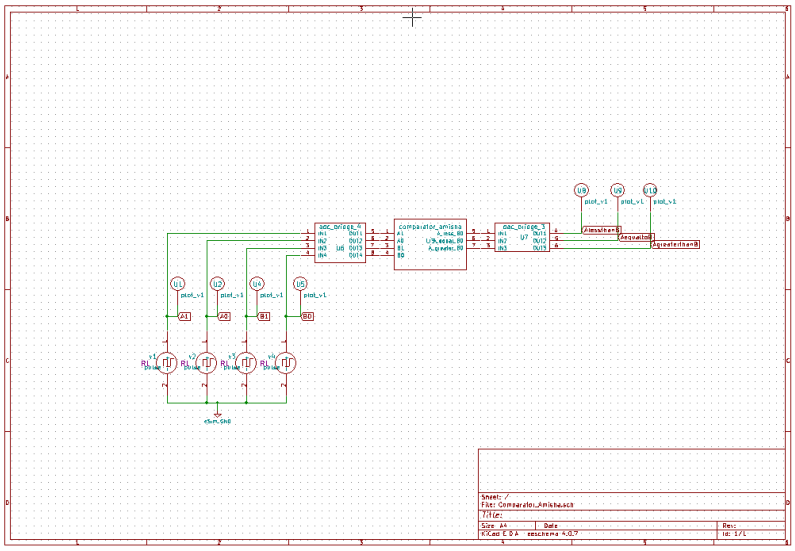
Outputs:

- *A_greater_B*: This output will be set to high if A is greater than B. Otherwise, it will be set to low.
- *A_equal_B*: This output will be set to high if A is equal to B. Otherwise, it will be set to low.
- *A_less_B*: This output will be set to high if A is less than B. Otherwise, it will be set to low.

8.2 Verilog Code

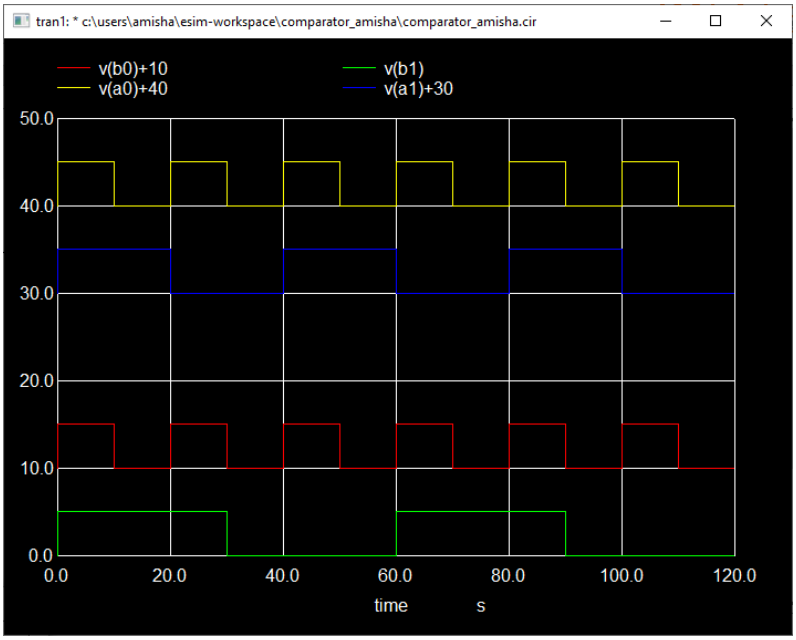
```
module Comparator_Amisha(input [1:0] A,B, output A_less_B, A_equal_B, A_greater_B);
    wire tmp1,tmp2,tmp3,tmp4,tmp5, tmp6, tmp7, tmp8;
    // A = B output
    xnor u1(tmp1,A[1],B[1]);
    xnor u2(tmp2,A[0],B[0]);
    and u3(A_equal_B,tmp1,tmp2);
    // A less than B output
    assign tmp3 = (~A[0])& (~A[1])& B[0];
    assign tmp4 = (~A[1])& B[1];
    assign tmp5 = (~A[0])& B[1]& B[0];
    assign A_less_B = tmp3 | tmp4 | tmp5;
    // A greater than B output
    assign tmp6 = (~B[0])& (~B[1])& A[0];
    assign tmp7 = (~B[1])& A[1];
    assign tmp8 = (~B[0])& A[1]& A[0];
    assign A_greater_B = tmp6 | tmp7 | tmp8;
endmodule
```

8.3 Schematic Diagram

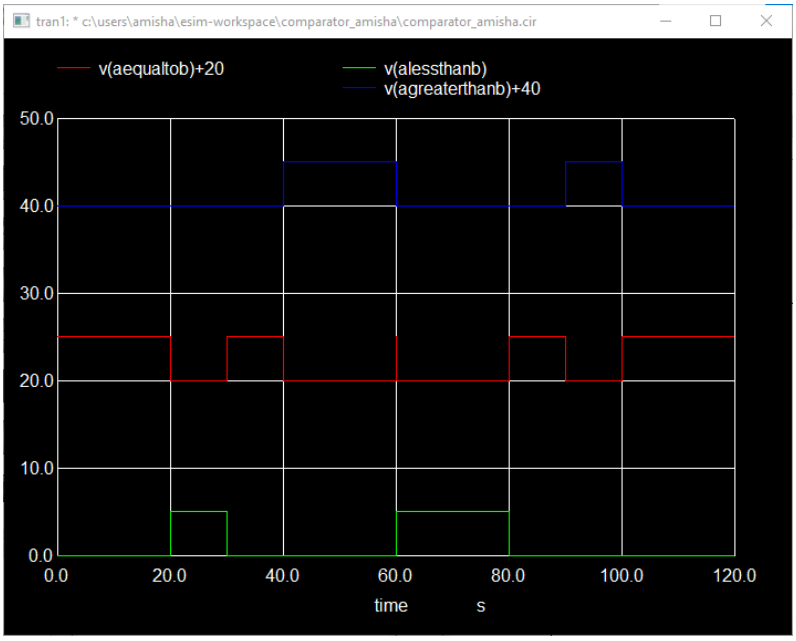


8.4 Ngspice Plots

8.4.1 Input plot



8.4.2 Output plot



9 4-bit Adder

9.1 Circuit Details

The primary function of this circuit is to perform addition on two 4-bit inputs, producing a 4-bit sum and carry bit as outputs.
Inputs:

- A: 4-bit input representing the first operand
- B: 4-bit input representing the second operand

Outputs:

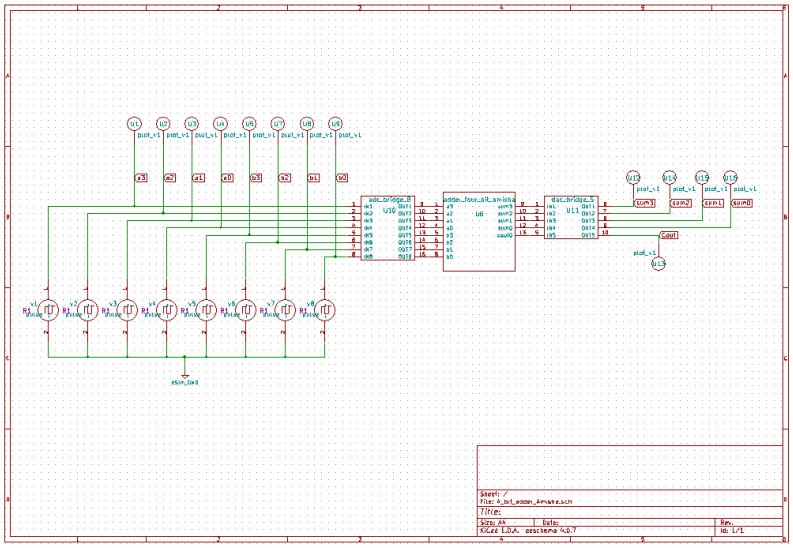
- Sum: 4-bit output representing the sum of the two operands
- Carry: Single-bit output representing the carry generated during addition

The 4-bit adder circuit will incorporate a series of full adder subcircuits to perform the addition operation. Each full adder will take in two bits from A and B, as well as the carry generated from the previous stage. The full adders will produce the corresponding sum bit and the carry bit for that stage.
The carry generated from the most significant stage (bit 3) will be the final carry output of the 4-bit adder. The sum bits from each stage will be combined to form the 4-bit sum output.

9.2 Verilog Code

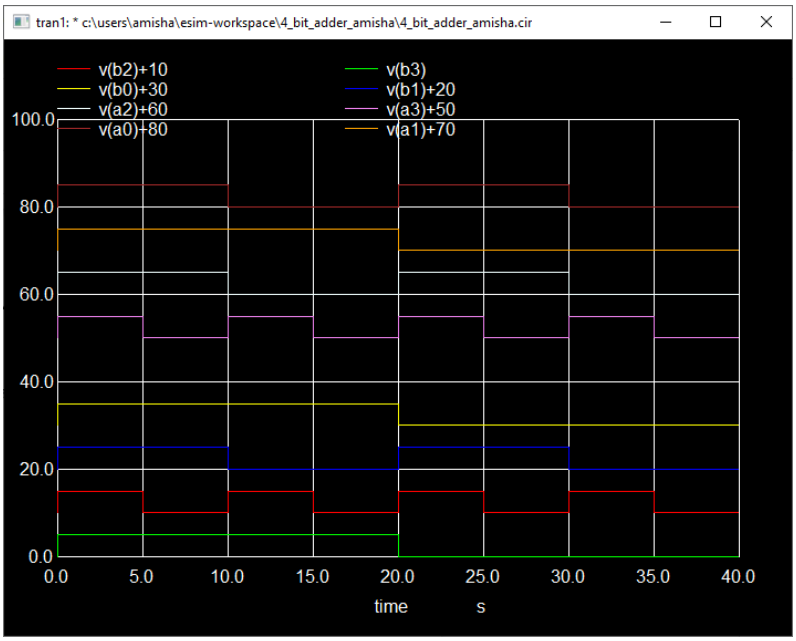
```
module full_3(a,b,cin,s,cout);
    input a,b,cin;
    output s, cout;
    assign s=a^b^cin;
    assign cout = (a&b) | (b&cin) | (cin&a);
endmodule

module adder_four_bit_Amisha(
    output [3:0]sum,
    output cout ,
    input [3:0]a,b);
    wire c1,c2,c3,c4;
    full_3 ad0( .a(a[0]), .b(b[0]),.cin(0), .s(sum[0]), .cout(c1));
    full_3 ad1( .a(a[1]), .b(b[1]),.cin(c1), .s(sum[1]), .cout(c2));
    full_3 ad2( .a(a[2]), .b(b[2]),.cin(c2), .s(sum[2]), .cout(c3));
    full_3 ad3( .a(a[3]), .b(b[3]),.cin(c3), .s(sum[3]), .cout(c4));
    assign cout= c4;
endmodule
```

9.3 Schematic Diagram

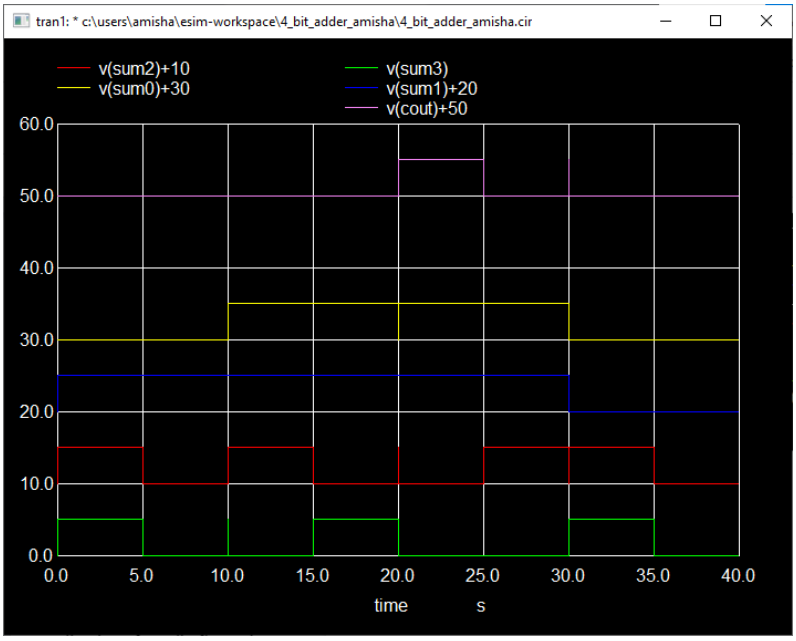


9.4 Ngspice Plots

9.4.1 Input plot



9.4.2 Output plot



10 Realization of JK Flip flop using RS flip flop

10.1 Circuit Details

The circuit will consist of an analog CMOS AND gate as the analog component and a digital flip-flop (FF) as the digital component.

The realization of the JK flip-flop involves connecting the J and K inputs to the inputs of an RS flip-flop. However, to ensure correct operation, an analog CMOS AND gate is employed as the analog component to generate the R and S inputs for the RS flip-flop.

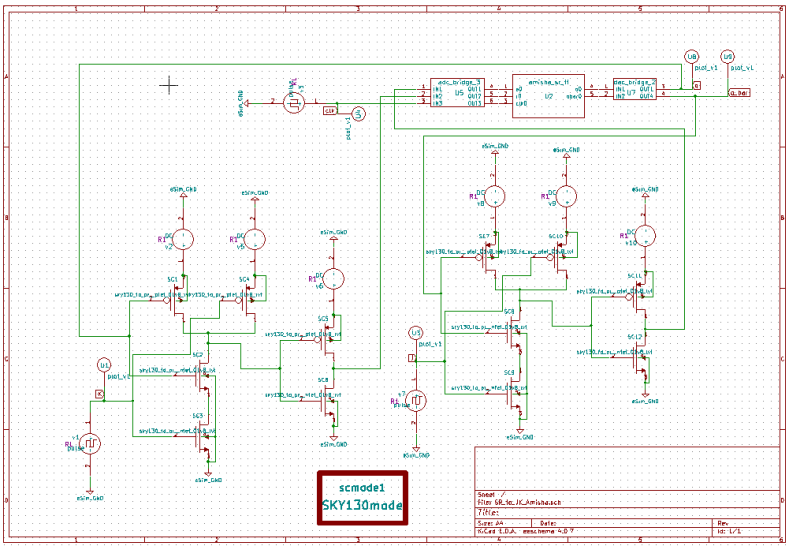
The CMOS AND gate receives the J and K inputs and produces the R (Reset) and S (Set) inputs for the RS flip-flop. The output of the CMOS AND gate is connected to the R and S inputs of the digital flip-flop, which is triggered by the CLK signal.

The digital flip-flop will store the state based on the R and S inputs when the CLK signal transitions. The output Q will represent the current state of the JK flip-flop.

10.2 Verilog Code

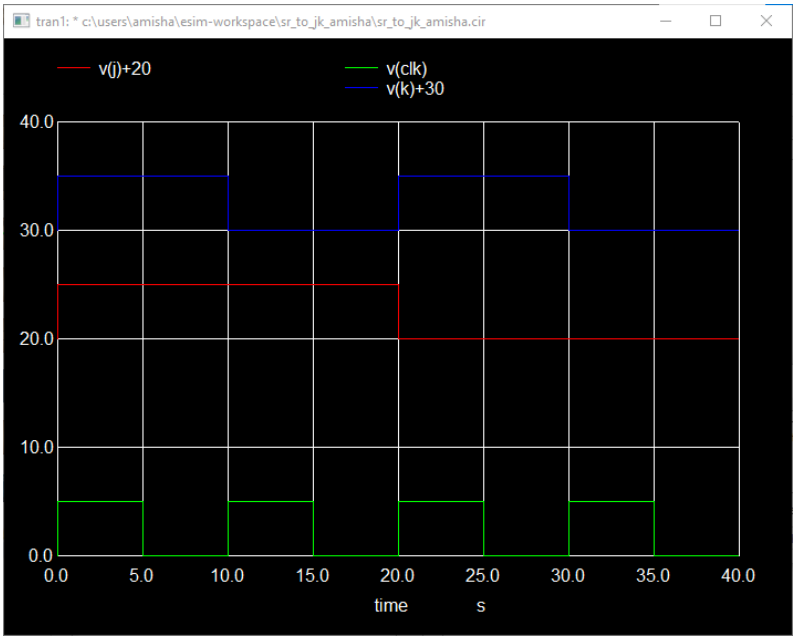
```
module Amisha_SR_ff(q, qbar, s, r, clk);
    input s,r,clk;
    output q, qbar;

    wire nand1_out; // output of nand1
    wire nand2_out; // output of nand2

    nand (nand1_out,clk,s);
    nand (nand2_out,clk,r);
    nand (q,nand1_out,qbar);
    nand (qbar,nand2_out,q);
endmodule
```

10.3 Schematic Diagram

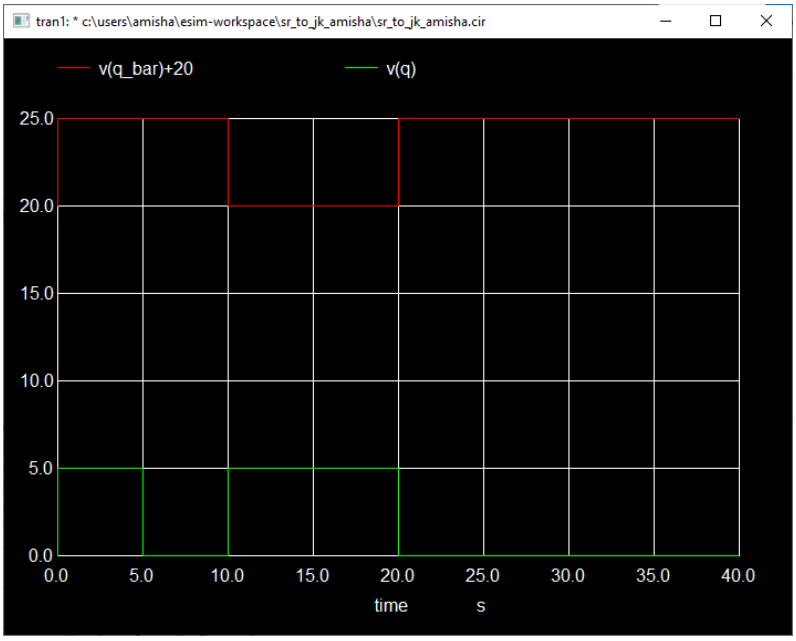


10.4 Ngspice Plots

10.4.1 Input plot



10.4.2 Output plot



11 Realization of SR Flip Flop to D Flip Flop

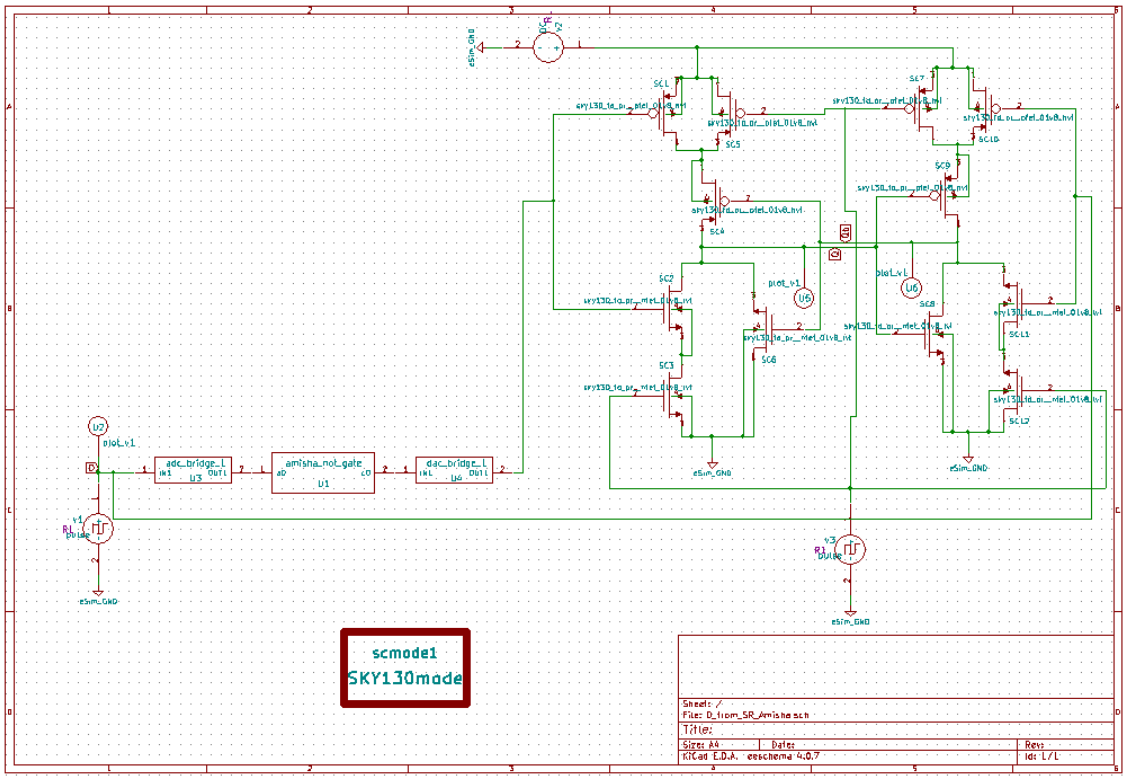
11.1 Circuit Details

The circuit will utilize the inputs S (Set) and R (Reset) as the actual inputs of the flip-flop, while D (Data) will serve as the external input. The circuit will also output Q (output) and Q' (complement of output). By implementing this mixed-signal circuit with a digital NOT gate and a CMOS analog flip-flop, we enable the conversion of inputs S and R based on the external input D and the previous output Qp.

11.2 Verilog Code

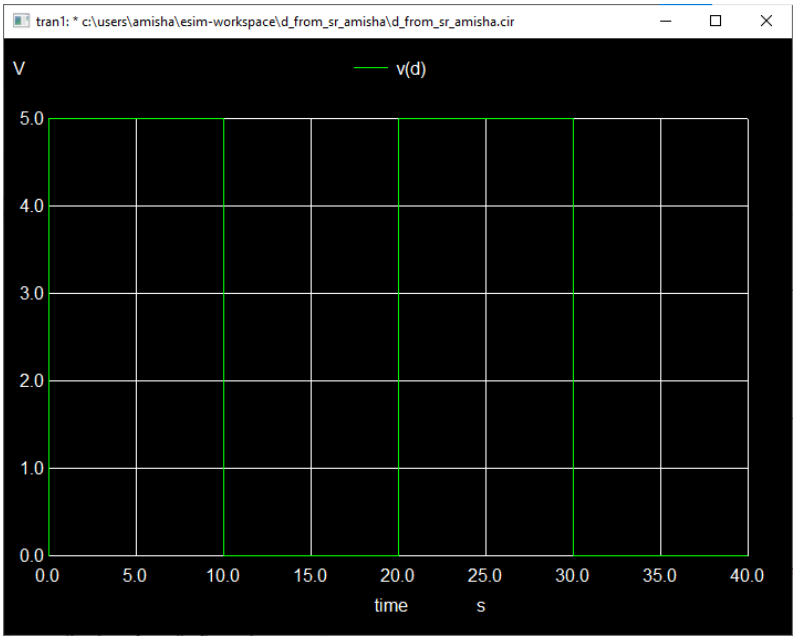
```
module amisha_not_gate(
    input a,
    output c
    );
    assign c=~a;
endmodule
```

11.3 Schematic Diagram

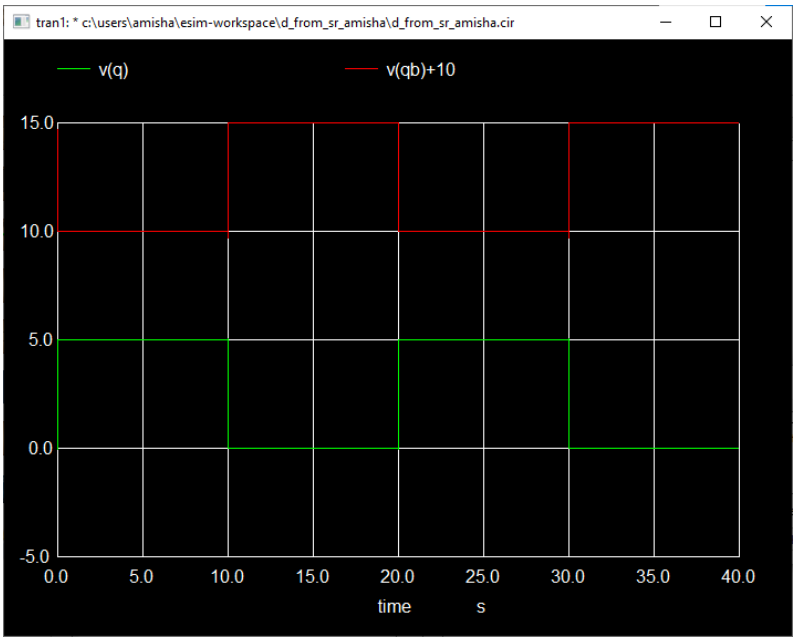


11.4 Ngspice Plots

11.4.1 Input plot



11.4.2 Output plot



12 PWM generator with Variable Duty Cycle

12.1 Circuit Details

Pulse Width Modulation generator creates a 10MHz PWM signal with variable duty cycle. The PWM generator will utilize a clock signal, an incremental clock, and a decremental clock to control the output cycle. The output cycle will depend on the activation of the "increase" or "decrease" inputs.

Inputs:

- clk: The main clock signal
- increment-clk: The incremental clock signal
- decrement-clk: The decremental clock signal

Outputs:

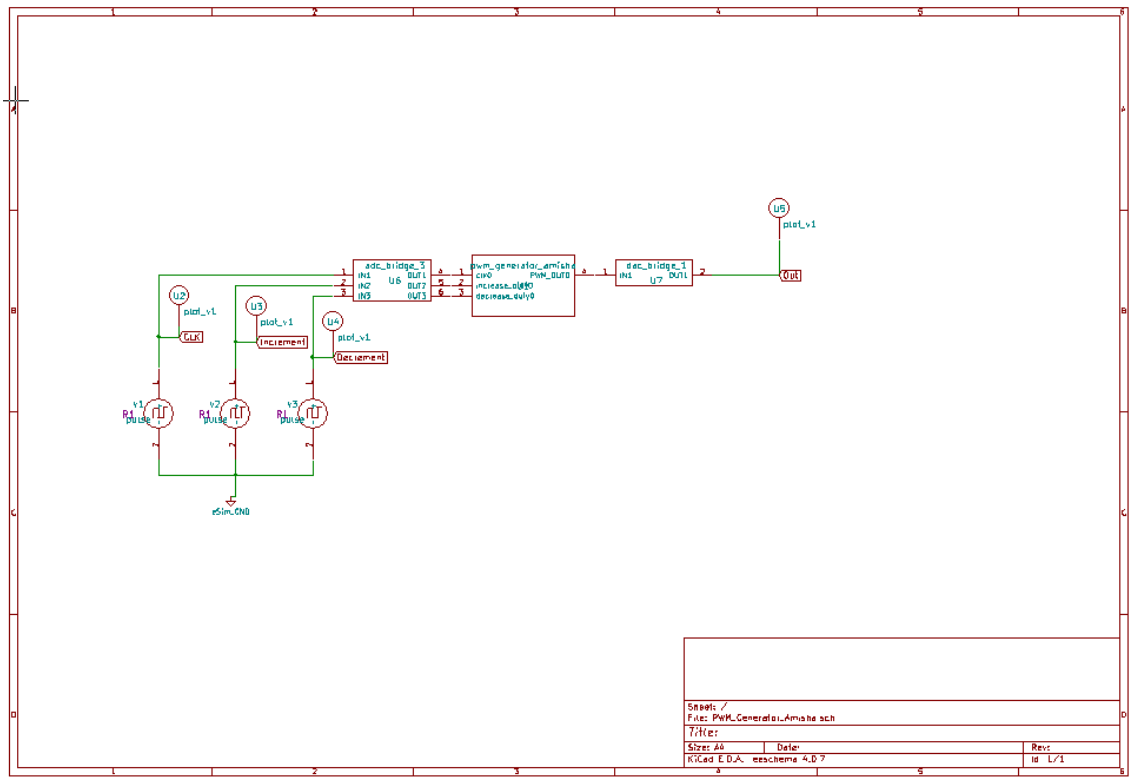
- PWM-output: The PWM output signal

12.2 Verilog Code

```
module DFF_PWM(clk,en,D,Q);
    input clk,en,D;
    output reg Q;
    always @(posedge clk)
    begin
        if(en==1)
        Q <= D;
    end
endmodule

module PWM_Generator_Amisha
(
    clk,
    increase_duty,
    decrease_duty,
    PWM_OUT
);
    input clk;
    input increase_duty;
    input decrease_duty;
    output PWM_OUT;
    wire slow_clk_enable;
    reg[27:0] counter_debounce=0;
    wire tmp1,tmp2,duty_inc;
    wire tmp3,tmp4,duty_dec;
    reg[3:0] counter_PWM=0;
    reg[3:0] DUTY_CYCLE=5;
    always @(posedge clk)
    begin
        counter_debounce <= counter_debounce + 1;
        if(counter_debounce>=1)
            counter_debounce <= 0;
    end
    assign slow_clk_enable = counter_debounce == 1 ?1:0;
    DFF_PWM PWM_DFF1(clk,slow_clk_enable,increase_duty,tmp1);
```

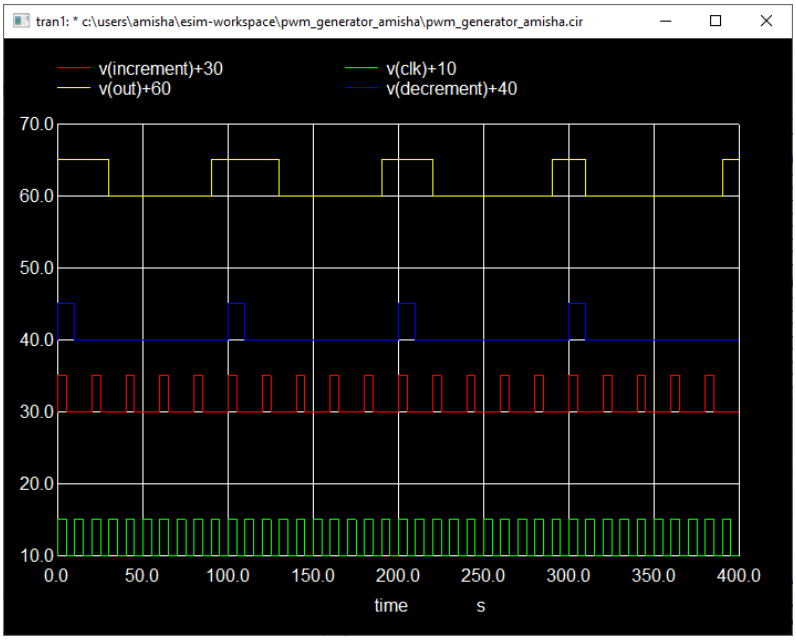
```
    DFF_PWM PWM_DFF2(clk,slow_clk_enable,tmp1, tmp2);
    assign duty_inc =  tmp1 & (~ tmp2) & slow_clk_enable;
    DFF_PWM PWM_DFF3(clk,slow_clk_enable,decrease_duty, tmp3);
    DFF_PWM PWM_DFF4(clk,slow_clk_enable,tmp3, tmp4);
    assign duty_dec =  tmp3 & (~ tmp4) & slow_clk_enable;
    always @(posedge clk)
    begin
        if(duty_inc==1 && DUTY_CYCLE <= 9)
            DUTY_CYCLE <= DUTY_CYCLE + 1;// increase duty cycle by 10%
        else if(duty_dec==1 && DUTY_CYCLE>=1)
            DUTY_CYCLE <= DUTY_CYCLE - 1;//decrease duty cycle by 10%
    end
    //10MHz PWM signal with variable duty cycle controlled by 2 buttons
    always @(posedge clk)
    begin
        counter_PWM <= counter_PWM + 1;
        if(counter_PWM>=9)
            counter_PWM <= 0;
    end
    assign PWM_OUT = counter_PWM < DUTY_CYCLE ? 1:0;
endmodule
```

12.3 Schematic Diagram

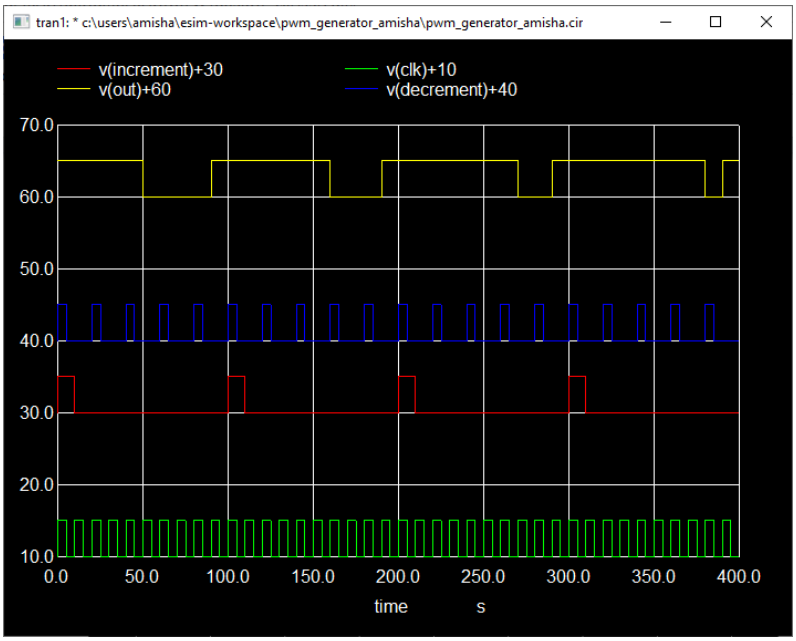


12.4 Ngspice Plots

12.4.1 Decrement PWM Output plot



12.4.2 Increment PWM Output plot



13 MIPS-16 bit Microprocessor

13.1 Circuit Details

The processor will be capable of executing a specific set of instructions, including the following sequence:

```
/* lw     $3, 0($0) --
       Loop:     slti $1, $3, 50
       beq $1, $0, Skip
       add $4, $4, $3
       addi $3, $3, 1
       beq $0, $0, Loop--
       Skip
*/
```

- Load Word (lw): Loads a word from memory into register $3, with the address specified as an offset of register $0.
- Set Less Than Immediate (slti): Sets register $1 to 1 if register $3 is less than the immediate value 50; otherwise, sets it to 0.
- Branch if Equal (beq): Branches to the "Skip" label if register $1 is equal to register $0.
- Add (add): Adds the values of registers $4 and $3 and stores the result in register $4.
- Add Immediate (addi): Adds the immediate value 1 to register $3.
- Branch (beq): Unconditionally branches to the "Loop" label.

The Verilog implementation of the 16-bit single-cycle MIPS processor will include components such as an instruction decoder, control unit, ALU (Arithmetic Logic Unit), registers, and memory. These components will work together to execute the instructions and perform the required operations.

13.2 Verilog Code

```
module instr_mem          // a synthesisable rom implementation
(
    input     [15:0]     pc,
    output wire     [15:0]          instruction
);
    wire [3 : 0] rom_addr = pc[4 : 1];
    reg [15:0] rom[15:0];
    initial
    begin
        rom[0] = 16'b1000000110000000;
        rom[1] = 16'b0010110010110010;
        rom[2] = 16'b1101110001100111;
        rom[3] = 16'b1101110111011001;
        rom[4] = 16'b1111110110110001;
        rom[5] = 16'b1100000001111011;
        rom[6] = 16'b0000000000000000;
        rom[7] = 16'b0000000000000000;
        rom[8] = 16'b0000000000000000;
        rom[9] = 16'b0000000000000000;
        rom[10] = 16'b0000000000000000;
        rom[11] = 16'b0000000000000000;
        rom[12] = 16'b0000000000000000;
```

```
        rom[13] = 16'b0000000000000000;
        rom[14] = 16'b0000000000000000;
        rom[15] = 16'b0000000000000000;
    end
    assign instruction = (pc[15:0] < 32 )? rom[rom_addr[3:0]]: 16'd0;
endmodule


module control(
    input[2:0] opcode,
    input reset,
    output reg[1:0] reg_dst,mem_to_reg,alu_op,
    output reg jump,branch,mem_read,mem_write,alu_src,reg_write,sign_or_zero
    );
    always @(*)
    begin
        if(reset == 1'b1)
        begin
            reg_dst = 2'b00;
            mem_to_reg = 2'b00;
            alu_op = 2'b00;
            jump = 1'b0;
            branch = 1'b0;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            reg_write = 1'b0;
            sign_or_zero = 1'b1;
        end
        else
        begin
            case(opcode)
            3'b000:
            begin // add
                reg_dst = 2'b01;
                mem_to_reg = 2'b00;
                alu_op = 2'b00;
                jump = 1'b0;
                branch = 1'b0;
                mem_read = 1'b0;
                mem_write = 1'b0;
                alu_src = 1'b0;
                reg_write = 1'b1;
                sign_or_zero = 1'b1;
            end
            3'b001:
            begin // sli
                reg_dst = 2'b00;
                mem_to_reg = 2'b00;
                alu_op = 2'b10;
                jump = 1'b0;
                branch = 1'b0;
                mem_read = 1'b0;
                mem_write = 1'b0;
                alu_src = 1'b1;
                reg_write = 1'b1;
```

```
    sign_or_zero = 1'b0;
end
3'b010:
begin // j
    reg_dst = 2'b00;
    mem_to_reg = 2'b00;
    alu_op = 2'b00;
    jump = 1'b1;
    branch = 1'b0;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b0;
    reg_write = 1'b0;
    sign_or_zero = 1'b1;
end
3'b011:
begin // jal
    reg_dst = 2'b10;
    mem_to_reg = 2'b10;
    alu_op = 2'b00;
    jump = 1'b1;
    branch = 1'b0;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b0;
    reg_write = 1'b1;
    sign_or_zero = 1'b1;
end
3'b100:
begin // lw
    reg_dst = 2'b00;
    mem_to_reg = 2'b01;
    alu_op = 2'b11;
    jump = 1'b0;
    branch = 1'b0;
    mem_read = 1'b1;
    mem_write = 1'b0;
    alu_src = 1'b1;
    reg_write = 1'b1;
    sign_or_zero = 1'b1;
end
3'b101:
begin // sw
    reg_dst = 2'b00;
    mem_to_reg = 2'b00;
    alu_op = 2'b11;
    jump = 1'b0;
    branch = 1'b0;
    mem_read = 1'b0;
    mem_write = 1'b1;
    alu_src = 1'b1;
    reg_write = 1'b0;
    sign_or_zero = 1'b1;
end
3'b110:
begin // beq
```

```
                reg_dst = 2'b00;
                mem_to_reg = 2'b00;
                alu_op = 2'b01;
                jump = 1'b0;
                branch = 1'b1;
                mem_read = 1'b0;
                mem_write = 1'b0;
                alu_src = 1'b0;
                reg_write = 1'b0;
                sign_or_zero = 1'b1;
            end
            3'b111:
            begin // addi
                reg_dst = 2'b00;
                mem_to_reg = 2'b00;
                alu_op = 2'b11;
                jump = 1'b0;
                branch = 1'b0;
                mem_read = 1'b0;
                mem_write = 1'b0;
                alu_src = 1'b1;
                reg_write = 1'b1;
                sign_or_zero = 1'b1;
            end
            default:
            begin
                reg_dst = 2'b01;
                mem_to_reg = 2'b00;
                alu_op = 2'b00;
                jump = 1'b0;
                branch = 1'b0;
                mem_read = 1'b0;
                mem_write = 1'b0;
                alu_src = 1'b0;
                reg_write = 1'b1;
                sign_or_zero = 1'b1;
            end
            endcase
        end
    end
endmodule


module register_file
(
    input clk,
    input rst,
    // write port
    input reg_write_en,
    input [2:0] reg_write_dest,
    input [15:0] reg_write_data,
    //read port 1
    input [2:0] reg_read_addr_1,
    output [15:0] reg_read_data_1,
    //read port 2
    input [2:0] reg_read_addr_2,
```

```
    output [15:0] reg_read_data_2
 );
    reg [15:0] reg_array [7:0];
    // write port
    //reg [2:0] i;
    always @ (posedge clk or posedge rst)
    begin
        if(rst)
        begin
            reg_array[0] <= 16'b0;
            reg_array[1] <= 16'b0;
            reg_array[2] <= 16'b0;
            reg_array[3] <= 16'b0;
            reg_array[4] <= 16'b0;
            reg_array[5] <= 16'b0;
            reg_array[6] <= 16'b0;
            reg_array[7] <= 16'b0;
        end
        else
        begin
            if(reg_write_en)
            begin
                reg_array[reg_write_dest] <= reg_write_data;
            end
        end
    end
    assign reg_read_data_1 = ( reg_read_addr_1 == 0)? 16'b0 : reg_array[reg_read_addr_1];
    assign reg_read_data_2 = ( reg_read_addr_2 == 0)? 16'b0 : reg_array[reg_read_addr_2];
endmodule


module ALUControl( ALU_Control, ALUOp, Function);
    output reg[2:0] ALU_Control;
    input [1:0] ALUOp;
    input [3:0] Function;
    wire [5:0] ALUControlIn;
    assign ALUControlIn = {ALUOp,Function};
    always @(ALUControlIn)
        casex (ALUControlIn)
            6'b11xxxx: ALU_Control=3'b000;
            6'b10xxxx: ALU_Control=3'b100;
            6'b01xxxx: ALU_Control=3'b001;
            6'b000000: ALU_Control=3'b000;
            6'b000001: ALU_Control=3'b001;
            6'b000010: ALU_Control=3'b010;
            6'b000011: ALU_Control=3'b011;
            6'b000100: ALU_Control=3'b100;
            default: ALU_Control=3'b000;
        endcase
endmodule


module JR_Control(
    input[1:0] alu_op,
    input [3:0] funct,
    output JRControl
```

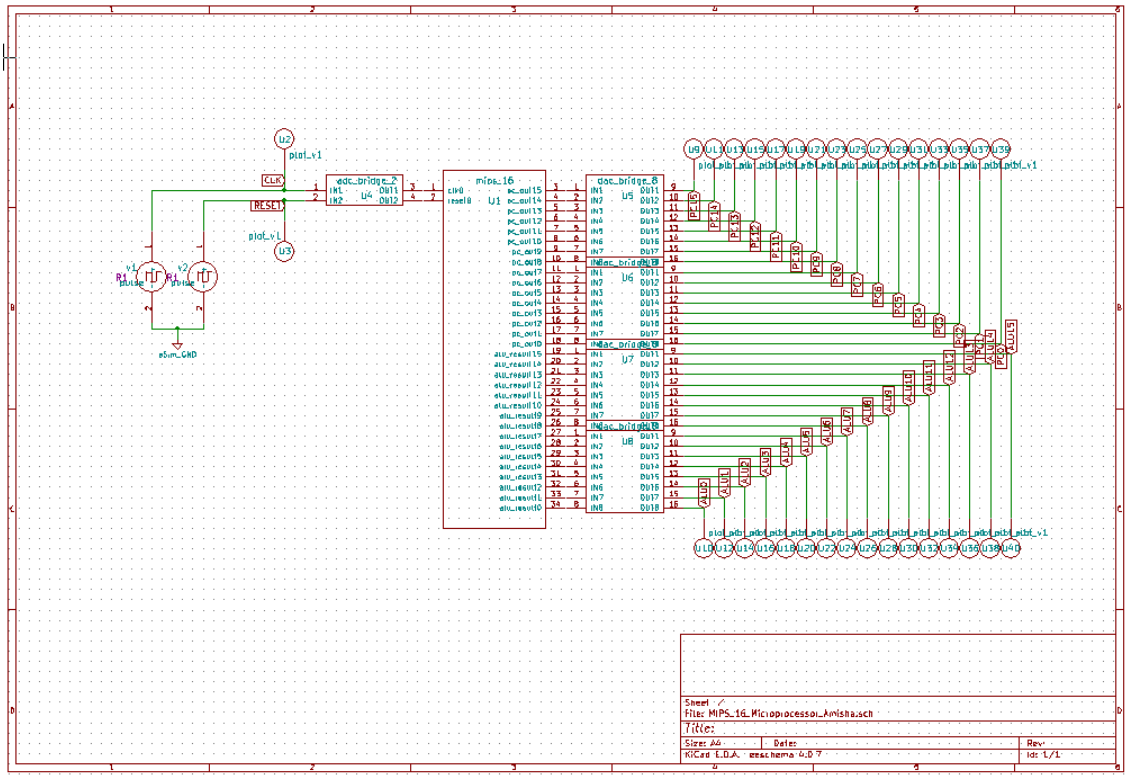
```
);
    assign JRControl = ({alu_op,funct}==6'b001000) ? 1'b1 : 1'b0;
endmodule


module ALUControl(ALU_Control, ALUOp, Function);
    output reg[2:0] ALU_Control;
    input [1:0] ALUOp;
    input [3:0] Function;
    wire [5:0] ALUControlIn;
    assign ALUControlIn = {ALUOp,Function};
    always @(ALUControlIn)
        casex (ALUControlIn)
            6'b11xxxx: ALU_Control=3'b000;
            6'b10xxxx: ALU_Control=3'b100;
            6'b01xxxx: ALU_Control=3'b001;
            6'b000000: ALU_Control=3'b000;
            6'b000001: ALU_Control=3'b001;
            6'b000010: ALU_Control=3'b010;
            6'b000011: ALU_Control=3'b011;
            6'b000100: ALU_Control=3'b100;
            default: ALU_Control=3'b000;
        endcase
endmodule


module JR_Control(
    input[1:0] alu_op,
    input [3:0] funct,
    output JRControl
);
    assign JRControl = ({alu_op,funct}==6'b001000) ? 1'b1 : 1'b0;
endmodule


module alu(
    input [15:0] a,          //src1
    input [15:0] b,          //src2
    input [2:0] alu_control,     //function sel
    output reg [15:0] result,          //result
    output zero
);
    always @(*)
    begin
        case(alu_control)
            3'b000: result = a + b; // add
            3'b001: result = a - b; // sub
            3'b010: result = a & b; // and
            3'b011: result = a | b; // or
            3'b100:
            begin
                if (a<b)
                    result = 16'd1;
                else
                    result = 16'd0;
            end
```

```
            default:result = a + b; // add
        endcase
end
    assign zero = (result==16'd0) ? 1'b1: 1'b0;
endmodule


module data_memory
(
    input clk,
    // address input, shared by read and write port
    input [15:0] mem_access_addr,
    // write port
    input [15:0] mem_write_data,
    input mem_write_en,
    input mem_read,
    // read port
    output [15:0] mem_read_data
);
    integer i;
    reg [15:0] ram [255:0];
    wire [7 : 0] ram_addr = mem_access_addr[8 : 1];
    initial
    begin
        for(i=0;i<256;i=i+1)
            ram[i] <= 16'd0;
    end
    always @(posedge clk)
    begin
        if (mem_write_en)
            ram[ram_addr] <= mem_write_data;
    end
      assign mem_read_data = (mem_read==1'b1) ? ram[ram_addr]: 16'd0;
 endmodule


module mips_16(
    input clk,reset,
    output[15:0] pc_out, alu_result
);
    reg[15:0] pc_current;
    wire signed[15:0] pc_next,pc2;
    wire [15:0] instr;
    wire[1:0] reg_dst,mem_to_reg,alu_op;
    wire jump,branch,mem_read,mem_write,alu_src,reg_write     ;
    wire     [2:0]     reg_write_dest;
    wire     [15:0] reg_write_data;
    wire     [2:0]     reg_read_addr_1;
    wire     [15:0] reg_read_data_1;
    wire     [2:0]     reg_read_addr_2;
    wire     [15:0] reg_read_data_2;
    wire [15:0] sign_ext_im,read_data2,zero_ext_im,imm_ext;
    wire JRControl;
    wire [2:0] ALU_Control;
    wire [15:0] ALU_out;
    wire zero_flag;
```

```
wire signed[15:0] im_shift_1, PC_j, PC_beq, PC_4beq,PC_4beqj,PC_jr;
wire beq_control;
wire [14:0] jump_shift_1;
wire [15:0]mem_read_data;
wire [15:0] no_sign_ext;
wire sign_or_zero;
// PC
always @(posedge clk or posedge reset)
begin
    if(reset)
        pc_current <= 16'd0;
    else
        pc_current <= pc_next;
end
// PC + 2
assign pc2 = pc_current + 16'd2;
// instruction memory
instr_mem instrucion_memory(.pc(pc_current),.instruction(instr));
// jump shift left 1
assign jump_shift_1 = {instr[13:0],1'b0};
// control unit
control control_unit(.reset(reset),.opcode(instr[15:13]),.reg_dst(reg_dst)                    ,
,.mem_read(mem_read), .mem_write(mem_write),.alu_src(alu_src)
,.reg_write(reg_write),.sign_or_zero(sign_or_zero));
// multiplexer regdest
assign reg_write_dest = (reg_dst==2'b10) ? 3'b111:
((reg_dst==2'b01) ? instr[6:4] :instr[9:7]);
// register file
assign reg_read_addr_1 = instr[12:10];
assign reg_read_addr_2 = instr[9:7];
register_file reg_file(.clk(clk),.rst(reset),.reg_write_en(reg_write),
.reg_write_dest(reg_write_dest),
.reg_write_data(reg_write_data),
.reg_read_addr_1(reg_read_addr_1),
.reg_read_data_1(reg_read_data_1),
.reg_read_addr_2(reg_read_addr_2),
.reg_read_data_2(reg_read_data_2));
assign sign_ext_im = {{9{instr[6]}},instr[6:0]};
assign zero_ext_im = {{9{1'b0}},instr[6:0]};
assign imm_ext = (sign_or_zero==1'b1) ? sign_ext_im : zero_ext_im;
// JR control
JR_Control JRControl_unit(.alu_op(alu_op),.funct(instr[3:0]),.JRControl(JRControl));
// ALU control unit
ALUControl ALU_Control_unit(.ALUOp(alu_op),.Function(instr[3:0]),.ALU_Control(ALU_Control));
// multiplexer alu_src
assign read_data2 = (alu_src==1'b1) ? imm_ext : reg_read_data_2;
// ALU
alu alu_unit(.a(reg_read_data_1),.b(read_data2),.alu_control(ALU_Control),.result(ALU_out),.ze
// immediate shift 1
assign im_shift_1 = {imm_ext[14:0],1'b0};
assign no_sign_ext = ~(im_shift_1) + 1'b1;
// PC beq add
assign PC_beq = (im_shift_1[15] == 1'b1) ? (pc2 - no_sign_ext): (pc2 +im_shift_1);
// beq control
assign beq_control = branch & zero_flag;
// PC_beq
```

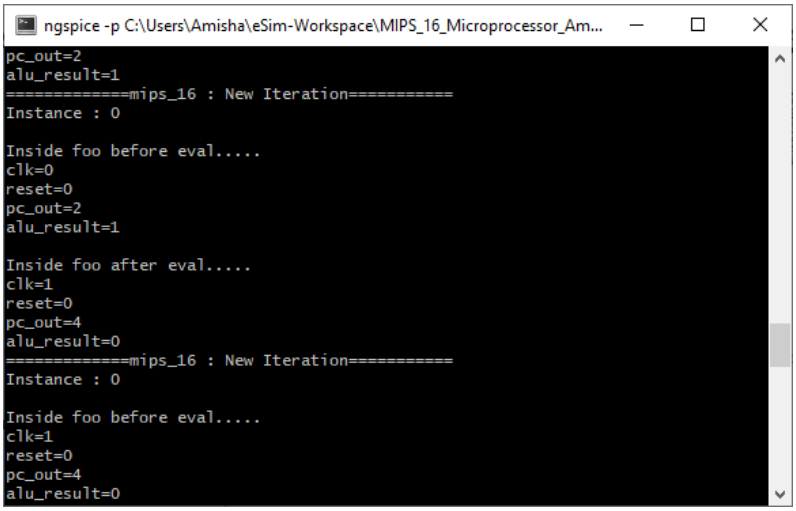
```
    assign PC_4beq = (beq_control==1'b1) ? PC_beq : pc2;
    // PC_j
    assign PC_j = {pc2[15],jump_shift_1};
    // PC_4beqj
    assign PC_4beqj = (jump == 1'b1) ? PC_j : PC_4beq;
    // PC_jr
    assign PC_jr = reg_read_data_1;
    // PC_next
    assign pc_next = (JRControl==1'b1) ? PC_jr : PC_4beqj;
    // data memory
    data_memory datamem(.clk(clk),.mem_access_addr(ALU_out),
     .mem_write_data(reg_read_data_2),.mem_write_en(mem_write),.mem_read(mem_read),
    .mem_read_data(mem_read_data));
    // write back
    assign reg_write_data = (mem_to_reg == 2'b10) ?
    pc2:((mem_to_reg == 2'b01)? mem_read_data: ALU_out);
    // output
    assign pc_out = pc_current;
    assign alu_result = ALU_out;
endmodule
```

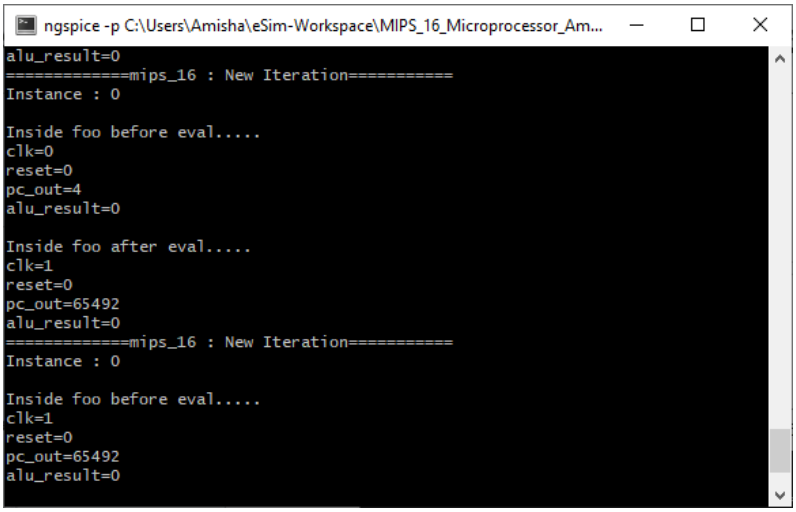
13.3 Schematic Diagram



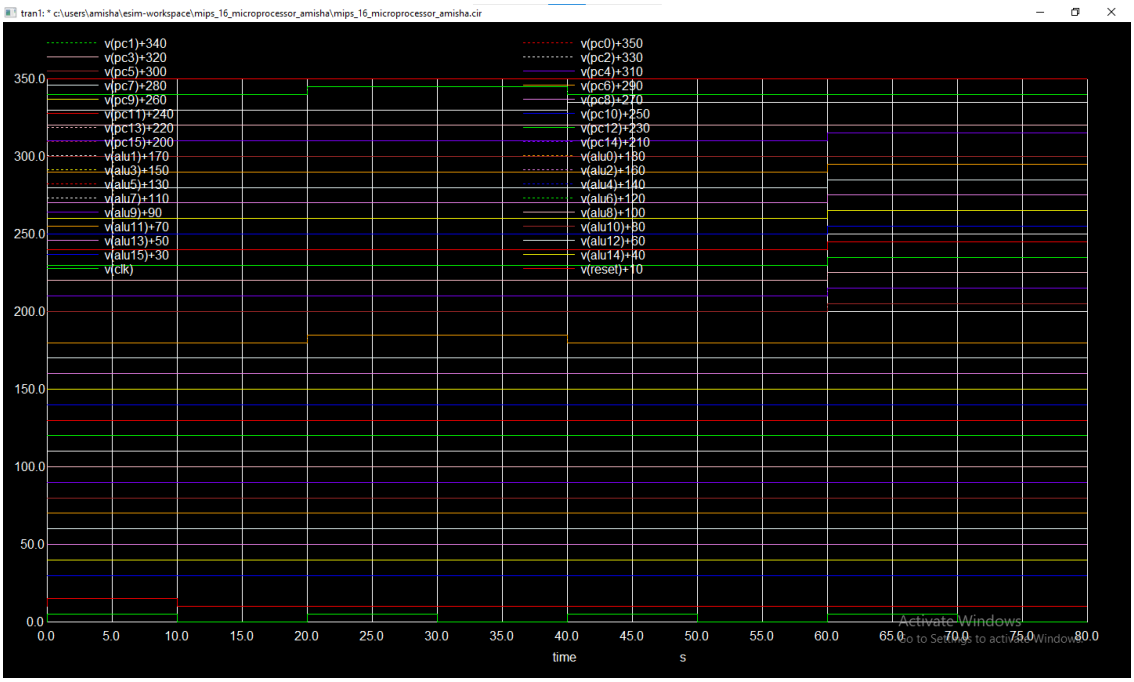
13.4 Ngspice Plots

13.4.1 NgSpice





13.4.2 Output plot



14 Bibliography

1. FOSSEE Official Website

   URL: https://fossee.in/about

2. eSim Official website

   URL: https://esim.fossee.in/

3. GitHub

   URL:https://github.com/Caskman/MIPS-Processor-in-Verilog

4. Tutorials Point

   URL:https://www.tutorialspoint.com/digital-circuits-and-their-applications

5. Wikipedia

   URL: https://en.wikipedia.org/wiki/Mixed-signal$_{i}ntegrated_{c}ircuit$

6. FPGA4Student

   URL: https://www.fpga4student.com/2017/08/verilog-code-for-pwm-generator.html