Semester Long Internship Report

On

Mixed Signal/Digital Simulation in eSim

Submitted by

Arun Pandiyan P B
Electrical and Electronics Engineering
Karpagam College of Engineering

Under the guidance of

Prof.Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

August 13, 2024

Acknowledgment

I would like to express my sincerest gratitude to the entire FOSSEE team for providing me this golden opportunity to be part of their Semester Long Internship program. The experience of working with an open-source organization such as FOSSEE has been immensely informative. It has given me the opportunity to learn about the complexities of simulation programs like eSim and develop valuable problem-solving abilities. My sincere gratitude goes out to Prof. Kannan M. Moudgalya for his visionary leadership and commitment to the advancement of this project.

In addition, I want to express my gratitude to Mr. Sumanto Kar who served as our mentors. Throughout my internship, their constant support and mentoring have been helpful. They were always there to offer advice and assist me in navigating through complexities so that I could find practical solutions whenever I ran into problems.

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 would utilize everything we got from here for our career growth as well as for the betterment of our society. Thank you to everyone who has contributed to making my internship journey memorable and rewarding.

Contents

Chapter 1

Introduction

FOSSEE (Free/Libre and Open Source Software for Education) project promotes the use of FLOSS tools to improve the quality of education in our country. It aims to reduce dependency on proprietary software in educational institutions. It encourages the use of FLOSS tools through various activities to ensure commercial software is replaced by equivalent FLOSS tools. It also develops new FLOSS tools and upgrade existing tools to meet requirements in academia and research.

The FOSSEE project is part of the National Mission on Education through Information and Communication Technology (ICT), Ministry of Human Resource Development (MHRD), Government of India.

eSim is a free/libre and open source EDA tool for circuit design, simulation, analysis and PCB design developed by FOSSEE, IIT Bombay. It is an integrated tool built using free/libre and open source software such as KiCad, Ngspice, NGHDL and GHDL.

Ngspice is a general purpose circuit simulation program for Nonlinear DC, Nonlinear transient, and Linear AC analysis. Circuits may contain Resistors, Capacitors, Inductors, mutual Inductors, independent voltage and current sources, four types of dependent sources, lossless and loss transmission lines (two separate implementations), switches, uniform distributed RC lines, and the five most common semiconductor devices: diodes, BJTs, JFETs and MOSFET.

NgVeri is a simulation tool used for verifying and analyzing digital circuits designed in Verilog, a popular hardware description language. It allows designers to simulate and test their circuit designs, ensuring they function as intended before being physically implemented.

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.

Chapter 2

4x4 Multiplier

A 4x4 multiplier is a digital circuit used to multiply two 4-bit binary numbers, producing an 8-bit binary result. The circuit typically consists of multiple smaller components such as adders and logic gates, organized in a structured manner to perform the multiplication process. Each bit of the multiplier and multiplicand contributes to the partial products, which are then summed to produce the final output. The 4x4 multiplier is efficient for small-scale multiplication tasks and serves as a fundamental building block in more complex arithmetic operations within digital systems.

2.1 Circuit Diagram

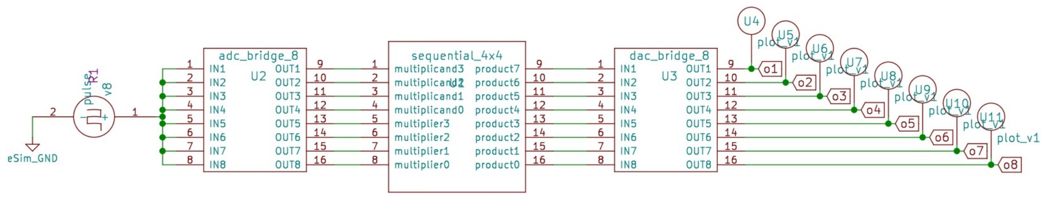

Figure 2.1: 4x4 Multiplier

Figure 2.1 illustrates the 4x4 Multiplier circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

2.2 Verilog Code

```
module sequential_4x4(
    input [3:0] multiplicand,
    input [3:0] multiplier,
    output reg [7:0] product
);
reg [7:0] temp_product;
```

```
always @ (*) begin
    temp_product = 8'b0;
    for (int i = 0; i < 4; i = i + 1) begin
        if (multiplier[i] == 1) begin
            temp_product = temp_product + (multiplicand << i);
        end
    end
end
always @ (*) begin
    product <= temp_product;
end
endmodule
```

2.3 Output Waveform

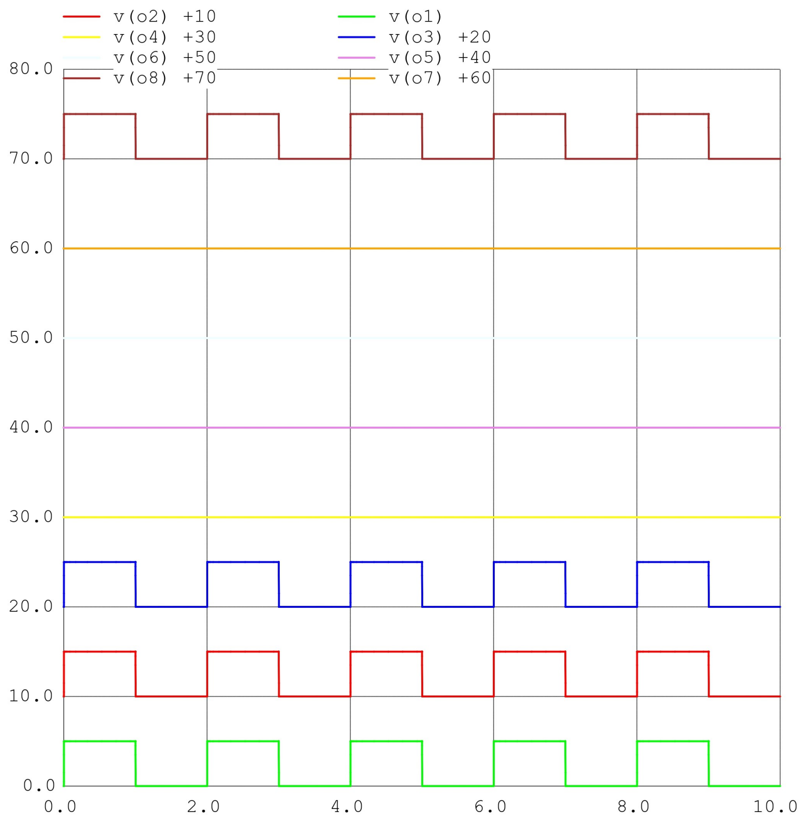

Figure 2.2: 4x4 Multiplier Output Waveform

The output waveform demonstrates the resulting waveform generated by the 4x4 Multiplier circuit using eSim and Ngspice and the output is ”1110 0001”.

Chapter 3

8x8 Multiplier

An 8x8 multiplier is a digital circuit used for multiplying two 8-bit binary numbers, producing a 16-bit binary result. This type of multiplier is integral in digital systems, enabling efficient and high-speed arithmetic operations. It leverages parallel processing techniques to handle multiple bit multiplications simultaneously, enhancing computational speed. The design of an 8x8 multiplier is more complex compared to smaller multipliers, often involving a combination of smaller multiplier units and additional logic circuits. This complexity allows it to serve as a building block for even larger multipliers. Widely used in applications such as digital signal processing, graphics processing, and embedded systems, the 8x8 multiplier is essential for tasks requiring quick and efficient data processing. While it typically consumes more power due to its complexity, optimized designs can help manage power consumption effectively.

3.1 Circuit Diagram

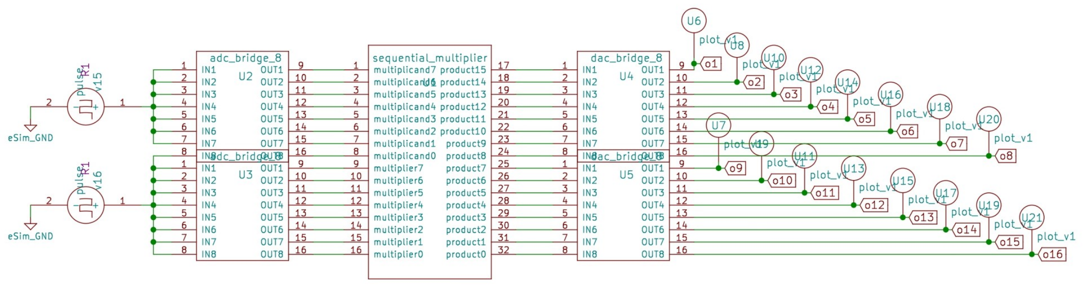

Figure 3.1: 8x8 Multiplier

Figure 3.1 illustrates the 8x8 Multiplier circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

3.2 Verilog Code

```
module sequential_multiplier(
    input [7:0] multiplicand,
    input [7:0] multiplier,
    output reg [15:0] product
);
reg [15:0] temp_product;
always @ (*) begin
    temp_product = 16'b0;
    for (int i = 0; i < 8; i = i + 1) begin
        if (multiplier[i] == 1) begin
            temp_product = temp_product + (multiplicand << i);
        end
    end
end
always @ (*) begin
    product <= temp_product;
end
endmodule
```

3.3 Output Waveform

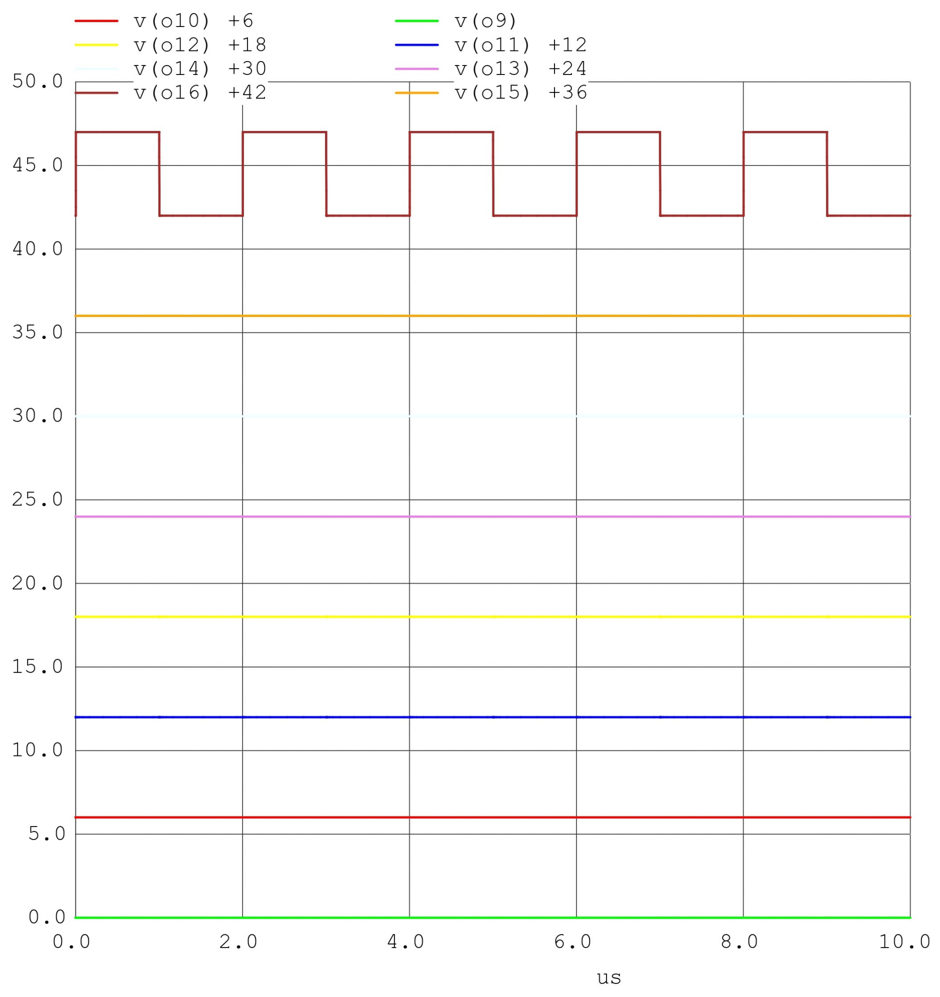

Figure 3.2: LSB bits of the 8x8 Multiplier output

Figure 3.2 represents the output ”0000 0001”, which is considered the LSB (Least Significant Bit) of the output.

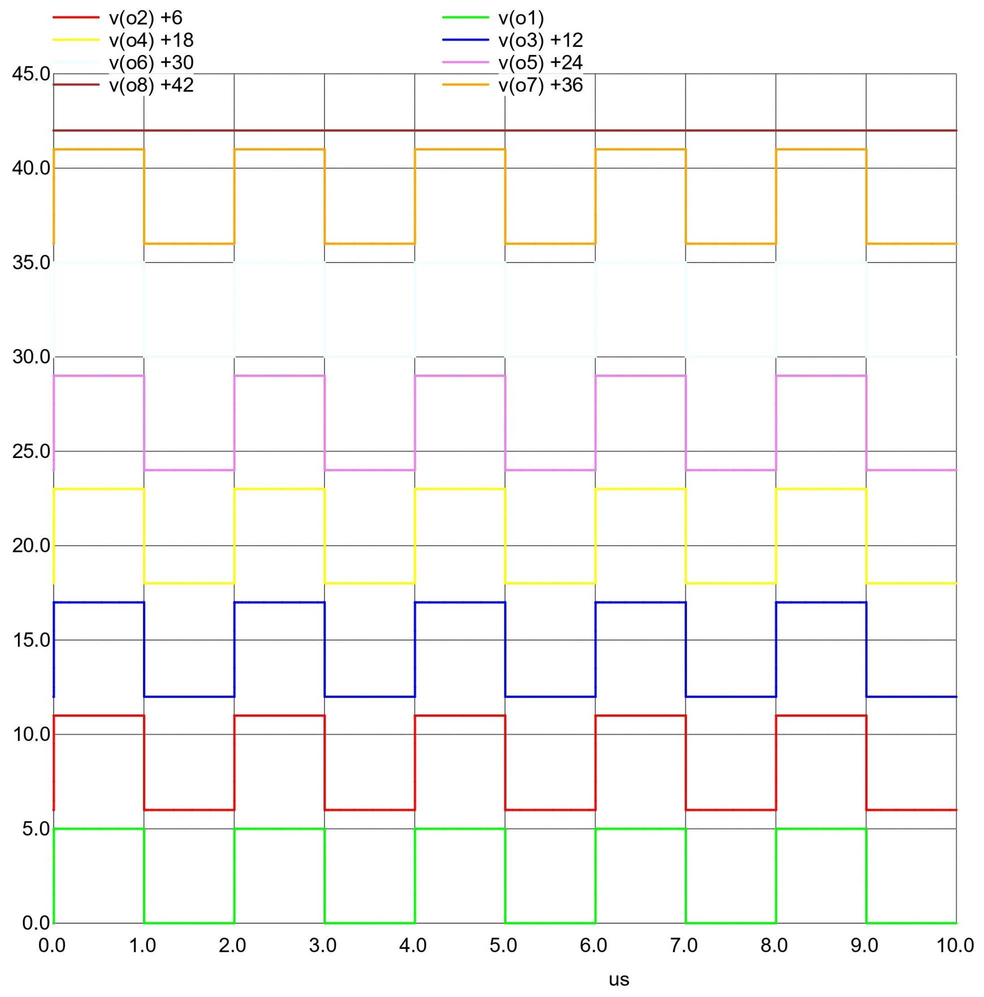

Figure 3.3: MSB bits of the 8x8 Multiplier output

Figure 3.3 represents the output ”1111 1110”, which is considered the MSB (Most Significant Bit) of the output. Cumulatively the output will be ”1111 1110 0000 0001”.

Chapter 4

Timer

A timer is a digital or analog device used to measure and control the passage of time in various applications. It can be found in a wide range of electronic devices, from household appliances to computer systems, and is crucial for tasks that require precise timing and scheduling.

4.1 Circuit Diagram

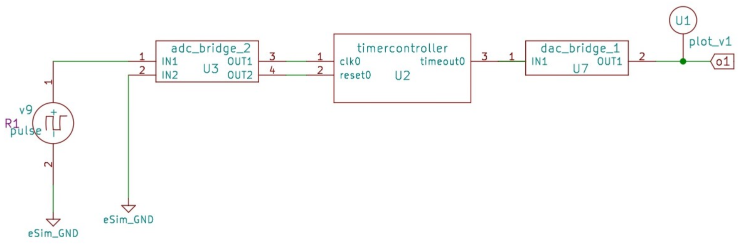

Figure 4.1: Timer circuit

Figure 4.1 illustrates the Timer circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

4.2 Verilog code

```
module TimerController  (
    input wire clk,
    input wire reset,
    output reg timeout
);
    reg [31:0] count;
    always @(posedge clk) begin
        if (reset) begin
            count <= 0;
            timeout <= 0;
        end else begin
            if (count < 100) begin
                count <= count + 1;
                timeout <= 0;
            end else begin
                timeout <= 1;
            end
        end
    end
endmodule
```

4.3 Output Waveform

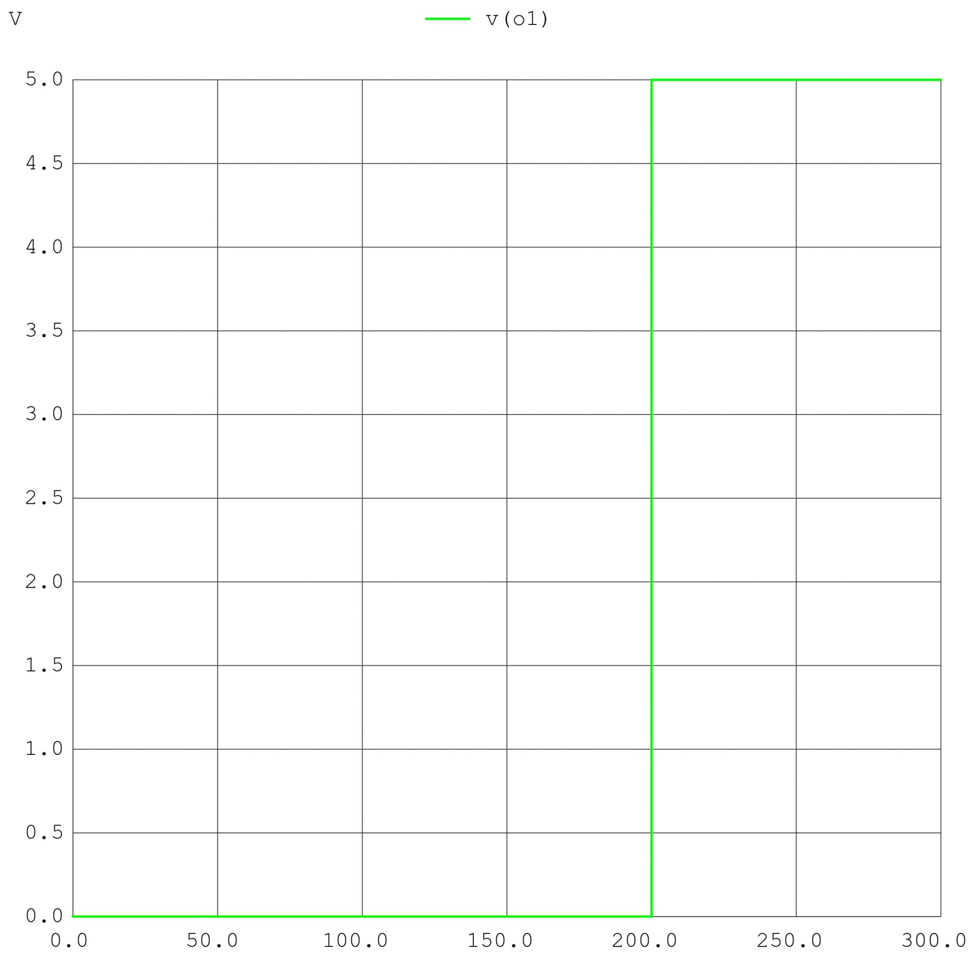

Figure 4.2: Timer Output

Figure 4.2 shows the output when the preset timer ends, setting the output to 1.

Chapter 5

Real Time Clock (RTC)

A real time clock is a specialized integrated circuit or module that keeps track of the current time and date, even when the main system or device is powered off. RTC's are crucial components in a wide range of applications, including computers, embedded systems, smartphones, and other electronic devices that require accurate timekeeping.In practical applications, a real-time clock is used for tasks such as time-stamping events, scheduling tasks and waking up systems from low-power states.

5.1 Circuit Diagram

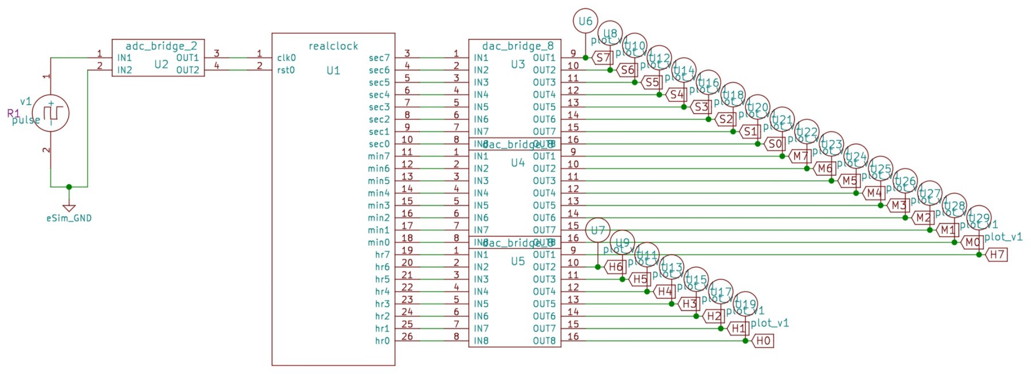

Figure 5.1: Real Time Clock Circuit

Figure 5.1 illustrates the Real Time Clock circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

5.2 Verilog Code

```
module TimerController  (
    input wire clk,
    input wire reset,
    output reg timeout
);
    reg [31:0] count;
    always @(posedge clk) begin
        if (reset) begin
            count <= 0;
            timeout <= 0;
        end else begin
            if (count < 100) begin
                count <= count + 1;
                timeout <= 0;
            end else begin
                timeout <= 1;
            end
        end
    end
endmodule
```

5.3 Output Waveform

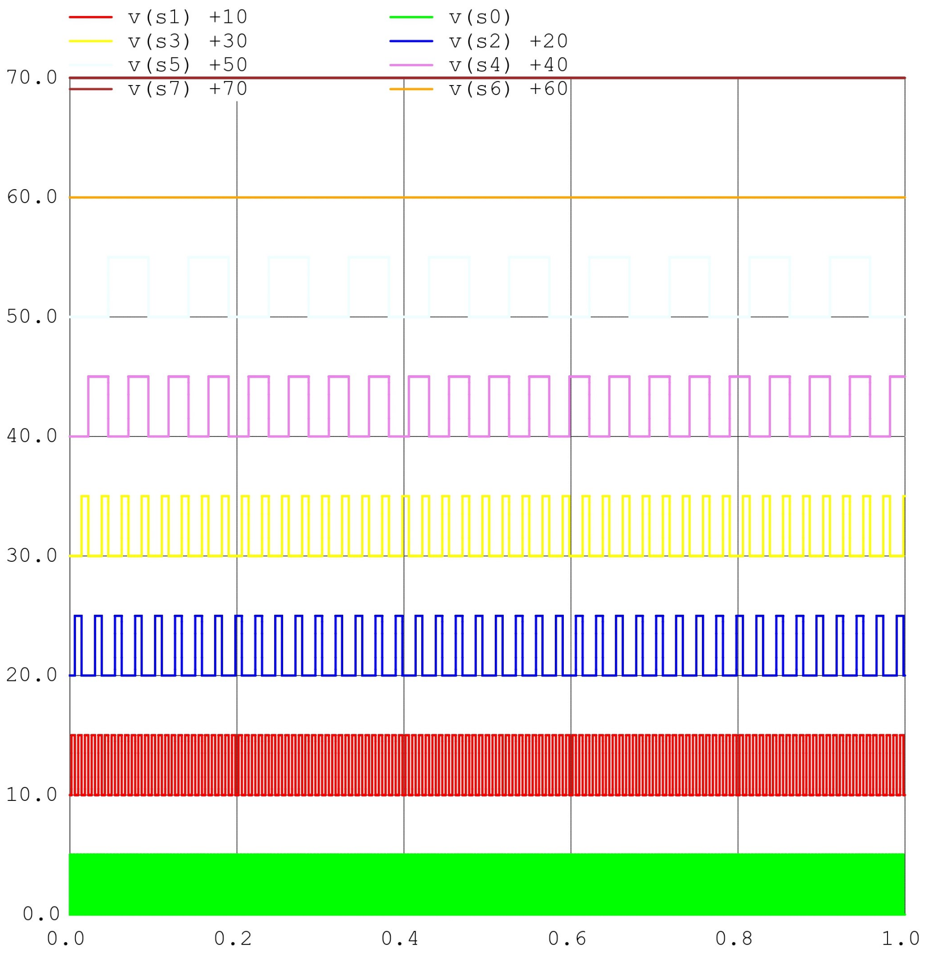

Figure 5.2: Real Time Clock Second Output

Figure 5.2 represents the second output as graph from the real-time clock circuit.

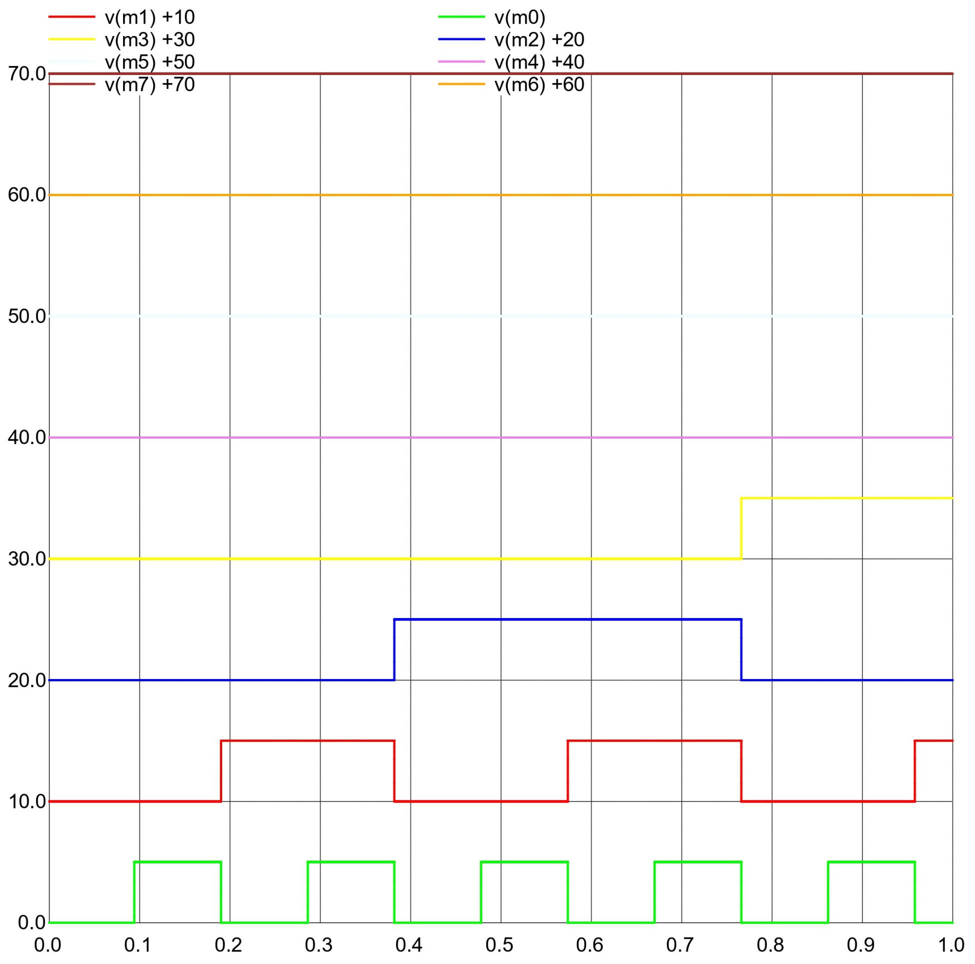

Figure 5.3: Real Time Clock Minute Output

Figure 5.3 represents the minute output as graph from the real-time clock circuit.

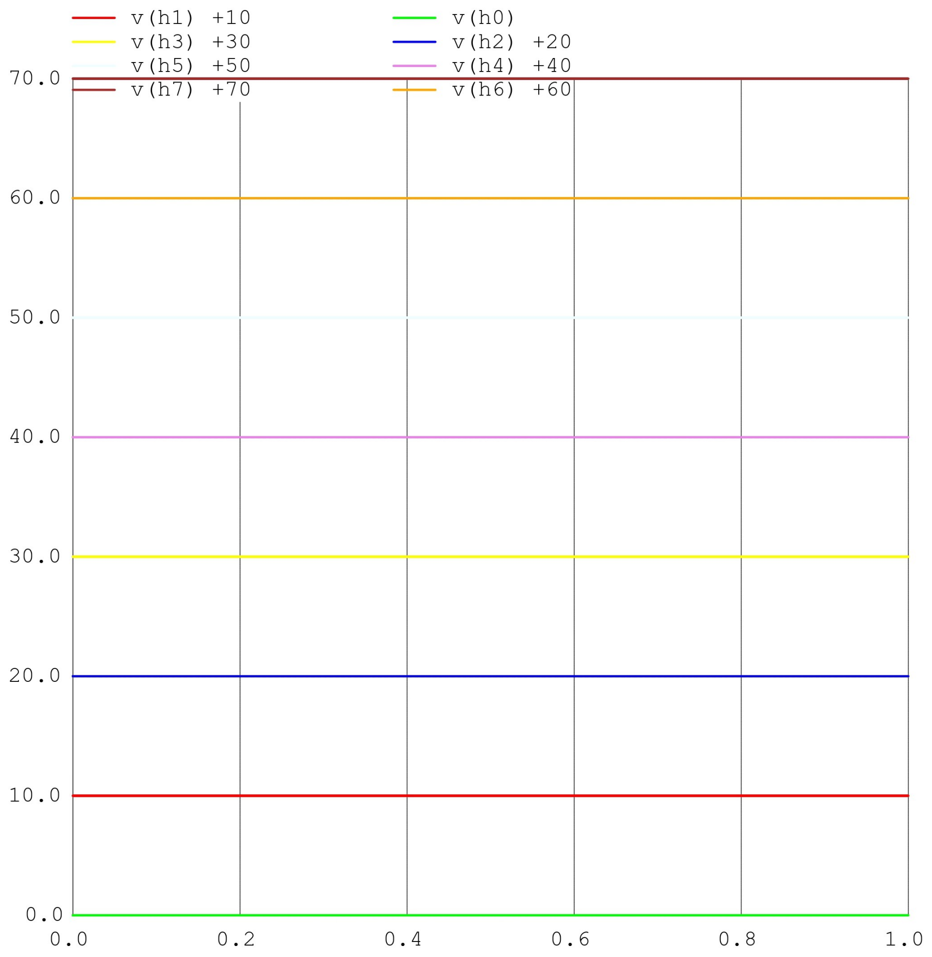

Figure 5.4: Real Time Clock Hour Output

Figure 5.4 represents the hour output as graph from the real-time clock circuit.

Chapter 6

Read Only Memory (ROM)

Read-Only Memory (ROM) is a type of non-volatile memory used in computers and electronic devices to store firmware and other essential data that must remain unchanged during normal operation. Unlike volatile memory like RAM (Random Access Memory), ROM retains its contents even when the power is turned off, making it crucial for storing permanent instructions and data. It is essential for providing a stable, unchangeable repository of crucial system instructions and firmware, ensuring the consistent and reliable operation of electronic devices.

6.1 Circuit Diagram

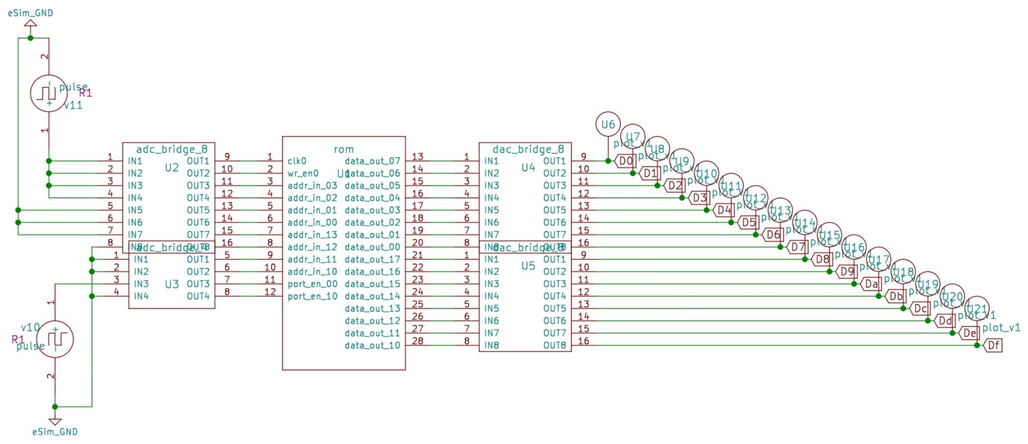

Figure 6.1: Read Only Memory Circuit

Figure 6.1 illustrates the Read Only Memory (ROM) circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

6.2 Verilog Code

```
module Rom (input clk,
         input wr_en,
         input [3:0] addr_in_0,
         input [3:0] addr_in_1,
         input port_en_0,
         input port_en_1,
         output [7:0] data_out_0,
         output [7:0] data_out_1);
reg   [4:0] raddr;
reg[7:0] rom [7:0];
 always @(posedge clk)
 begin
    if (port_en_0 == 1 && wr_en == 1)
       raddr <= addr_in_0;
 end
 always @(raddr)
 begin
    if (port_en_0 == 1 && wr_en == 1)
       case(raddr)
   4'b0000: rom[raddr] = 8'b00100100;
   4'b0001: rom[raddr] = 8'b00100011;
   4'b0010: rom[raddr] = 8'b11100010;
   4'b0011: rom[raddr] = 8'b00100001;
   4'b0100: rom[raddr] = 8'b01000101;
   4'b0101: rom[raddr] = 8'b10101110;
   4'b0110: rom[raddr] = 8'b11001011;
   4'b0111: rom[raddr] = 8'b00000000;
   4'b1000: rom[raddr] = 8'b10100011;
   4'b1001: rom[raddr] = 8'b00101010;
   4'b1010: rom[raddr] = 8'b11101100;
   4'b1011: rom[raddr] = 8'b00100010;
   4'b1100: rom[raddr] = 8'b01000000;
   4'b1101: rom[raddr] = 8'b10100000;
   4'b1110: rom[raddr] = 8'b00001100;
   4'b1111: rom[raddr] = 8'b00000000;
   default: rom[raddr] = 8'bXXXXXXXX;
       endcase
 end
assign data_out_0 = port_en_0 ? rom[addr_in_0] : 'dZ;
assign data_out_1 = port_en_1 ? rom[addr_in_1] : 'dZ;
endmodule
```

6.3 Output Waveform

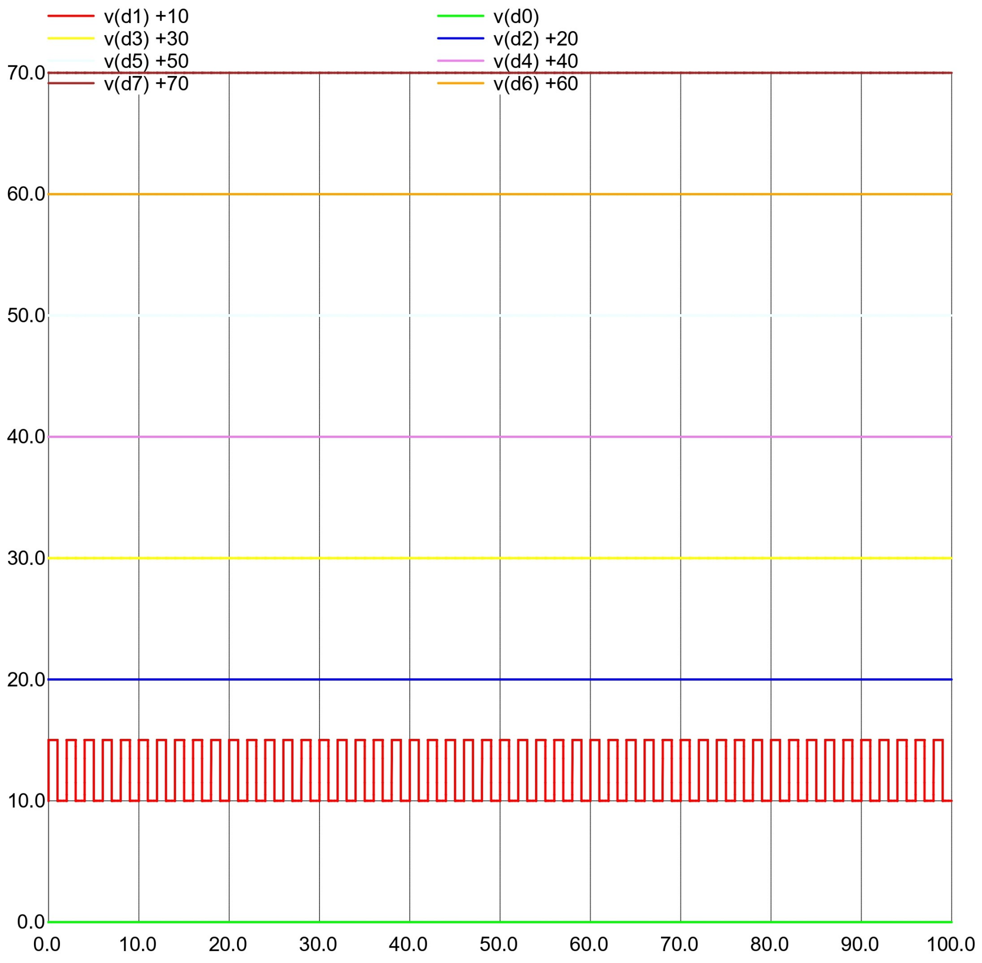

Figure 6.2: Least Significant Bit of Read Only Memory

Figure 6.2 represents the Least Significant Bit (LSB) of the Read-Only Memory (ROM). Based on the user input instruction, the output will be fetched from the ROM. The data is provided in the Verilog code Section 6.2.

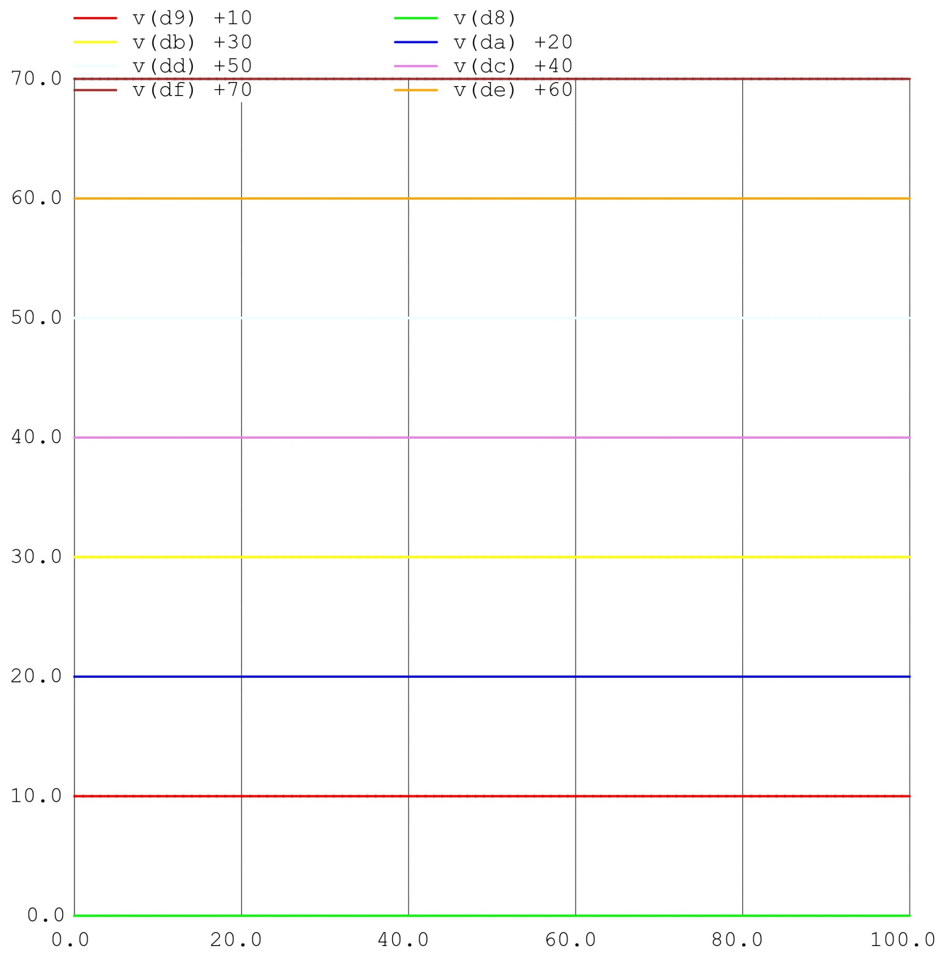

Figure 6.3: Most Significant Bit of Read Only Memory

Figure 6.3 represents the Most Significant Bit (MSB) of the Read-Only Memory (ROM). Based on the user input instruction, the output will be fetched from the ROM. The data is provided in the Verilog code above Section 6.2.

Chapter 7

Traffic Light Controller (TLC)

A traffic light controller is a digital system designed to manage the sequence of lights at an intersection to ensure smooth and safe traffic flow. It operates based on a preprogrammed schedule or real-time traffic data, controlling the red, yellow, and green lights to direct vehicles and pedestrians. The controller can be simple, with fixed timing intervals for each light, or more advanced, using sensors and algorithms to adapt to varying traffic conditions.

7.1 Circuit Diagram

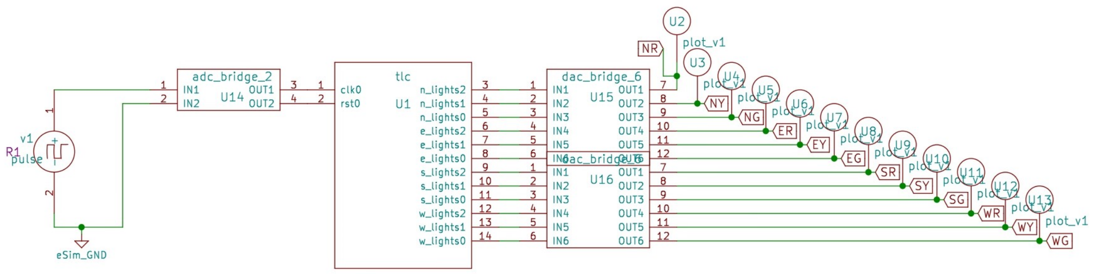

Figure 7.1: Circuit Diagram

Figure 7.1 illustrates the Traffic Light Controller (TLC) circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

7.2 Verilog Code

```
module TLC(n_lights,e_lights,s_lights,w_lights,clk,rst);
   input clk,rst;
   output reg [2:0] n_lights,e_lights,s_lights,w_lights;
   reg [2:0] state;
   reg [2:0] count;
   parameter [2:0] north=3'b000;
   parameter [2:0] north_y=3'b001;
   parameter [2:0] east=3'b010;
   parameter [2:0] east_y=3'b011;
   parameter [2:0] south=3'b100;
   parameter [2:0] south_y=3'b101;
   parameter [2:0] west=3'b110;
   parameter [2:0] west_y=3'b111;
   always @(posedge clk, posedge rst)
     begin
        if (rst)
            begin
                state=north;
                count =3'b000;
            end
        else
            begin
                case (state)
                north :
                    begin
                        if (count==3'b111)
                            begin
                            count=3'b000;
                            state=north_y;
                            end
                        else
                            begin
                            count=count+3'b001;
                            state=north;
                            end
                    end
                north_y :
                    begin
                        if (count==3'b011)
                            begin
                            count=3'b000;
                            state=east;
                            end
                        else
                            begin
                            count=count+3'b001;
                            state=north_y;
                        end
                    end
            east :
                begin
                    if (count==3'b111)
```

```
                begin
                count=3'b0;
                state=east_y;
                end
            else
                begin
                count=count+3'b001;
                state=east;
                end
            end
east_y :
        begin
            if (count==3'b011)
                begin
                count=3'b0;
                state=south;
                end
            else
                begin
                count=count+3'b001;
                state=east_y;
                end
            end
   south :
            begin
                if (count==3'b111)
                    begin
                    count=3'b0;
                    state=south_y;
                    end
                else
                    begin
                    count=count+3'b001;
                    state=south;
                end
            end
south_y :
        begin
            if (count==3'b011)
                begin
                count=3'b0;
                state=west;
                end
            else
                begin
                count=count+3'b001;
```

```
                        state=south_y;
                        end
                    end
            west :
                begin
                    if (count==3'b111)
                        begin
                        state=west_y;
                        count=3'b0;
                        end
                    else
                        begin
                        count=count+3'b001;
                        state=west;
                        end
                    end
            west_y :
                begin
                    if (count==3'b011)
                        begin
                        state=north;
                        count=3'b0;
                        end
                    else
                        begin
                        count=count+3'b001;
                        state=west_y;
                        end
                    end
            endcase
        end
    end
always @(state)
     begin
         case (state)
            north :
                begin
                    n_lights = 3'b001;
                    s_lights = 3'b100;
                    e_lights = 3'b100;
                    w_lights = 3'b100;
                end
            north_y :
                begin
                    n_lights = 3'b010;
                    s_lights = 3'b100;
                    e_lights = 3'b100;
                    w_lights = 3'b100;
                end
            east :
                begin
                    n_lights = 3'b100;
                    s_lights = 3'b001;
                    e_lights = 3'b100;
                    w_lights = 3'b100;
                end
            east_y :
                begin
                    n_lights = 3'b100;
                    s_lights = 3'b010;
                    e_lights = 3'b100;
                    w_lights = 3'b100;
                end
            west :
                begin
                    n_lights = 3'b100;
                    s_lights = 3'b100;
                    e_lights = 3'b100;
                    w_lights = 3'b001;
                end
            west_y :
                begin
                    n_lights = 3'b100;
                    s_lights = 3'b100;
                    e_lights = 3'b100;
                    w_lights = 3'b010;
                end
            south :
                begin
                    n_lights = 3'b100;
                    s_lights = 3'b100;
                    e_lights = 3'b001;
                    w_lights = 3'b100;
                end
            south_y :
                begin
                    n_lights = 3'b100;
                    s_lights = 3'b100;
                    e_lights = 3'b010;
                    w_lights = 3'b100;
                end
            endcase
```

```
    end
endmodule
```

7.3 Output Waveform

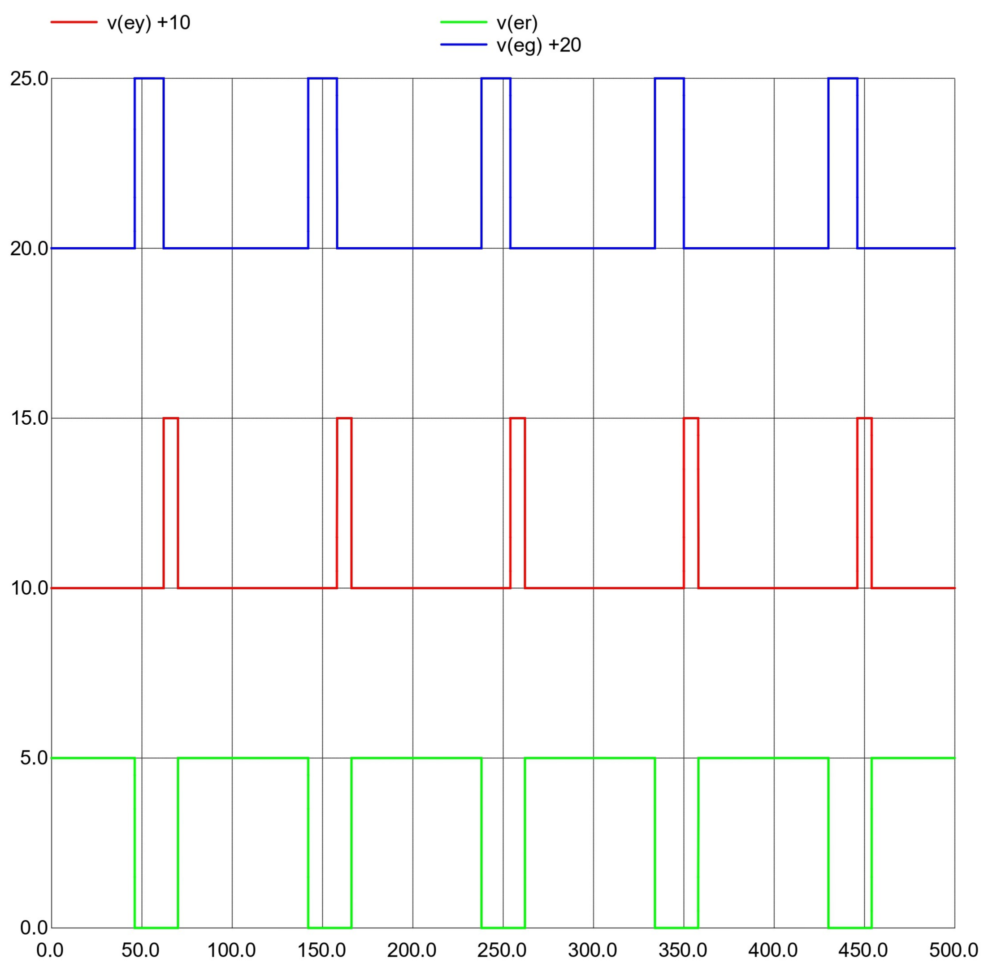

Figure 7.2: East Direction

Figure 7.2 represents the East direction as the output from the Traffic Light Controller circuit.

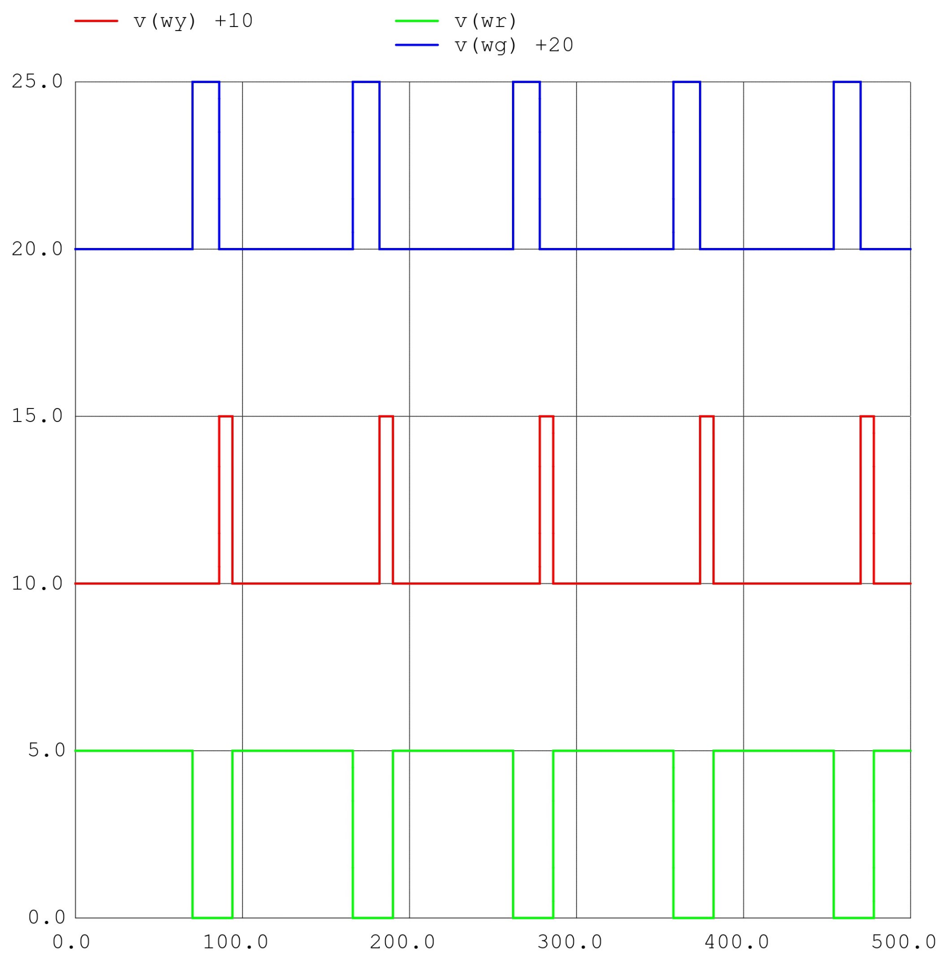

Figure 7.3: West Direction

Figure 7.3 represents the West direction as the output from the Traffic Light Controller circuit.

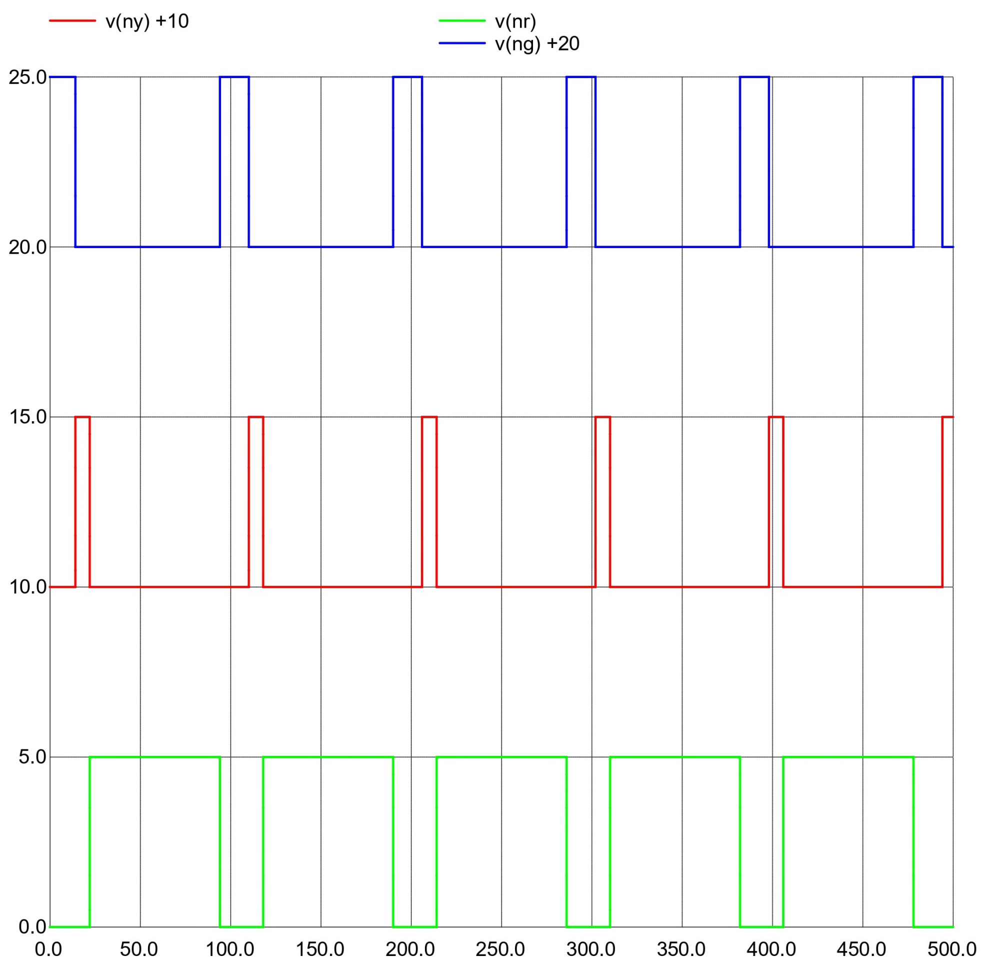

Figure 7.4: North Direction

Figure 7.4 represents the North direction as the output from the Traffic Light Controller circuit.

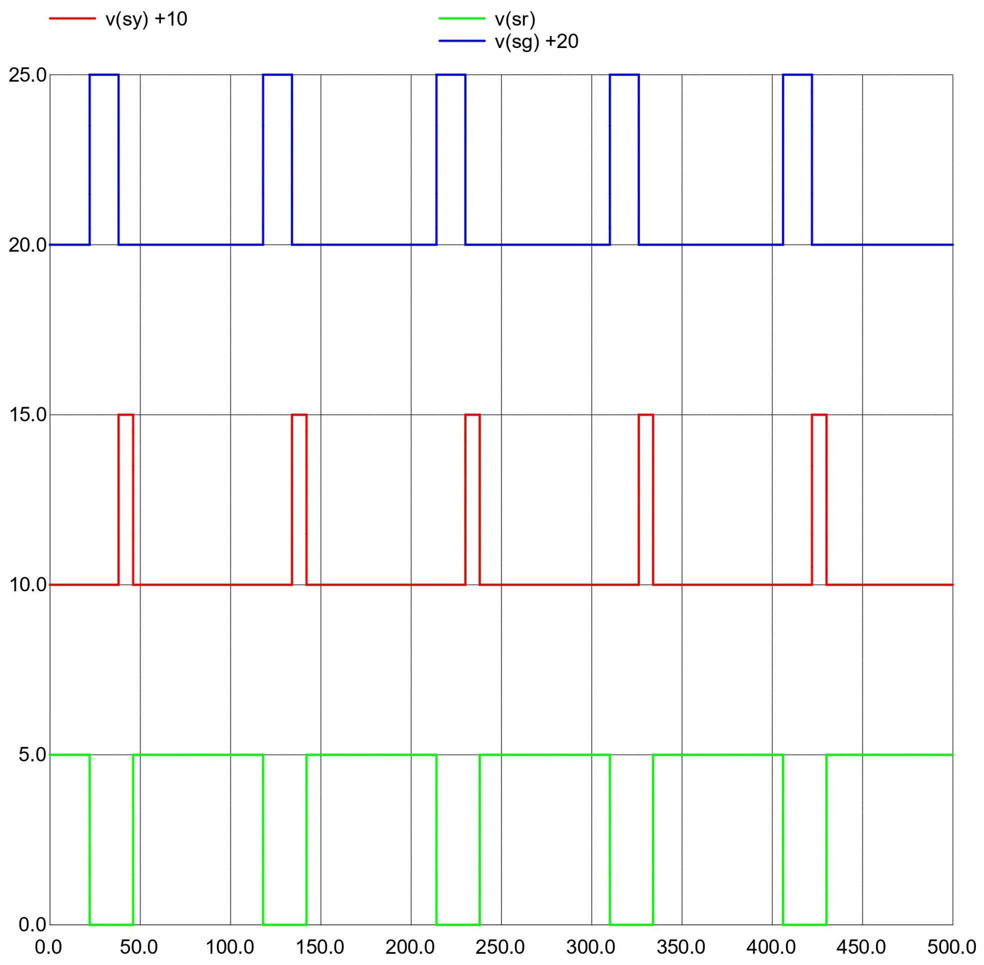

Figure 7.5: South Direction

Figure 7.5 represents the South direction as the output from the Traffic Light Controller circuit.

Chapter 8

Kogge Stone Adder

The Kogge-Stone adder is a high-performance parallel prefix adder used in digital circuits to execute fast binary addition. It is particularly favored in applications requiring rapid arithmetic operations, such as microprocessors and signal processing units. The adder is structured to compute carry signals in parallel, which significantly accelerates the addition process compared to traditional ripple carry adders. One of the key features of the Kogge-Stone adder is its logarithmic depth relative to the number of bits being added, meaning the time required to propagate carries increases logarithmically. This makes the adder extremely efficient for large bit-width additions. Additionally, the Kogge-Stone adder's design is known for having a regular structure, which simplifies its implementation and scaling in integrated circuits. Despite its complexity and higher gate count, the performance benefits often outweigh these costs, making the Kogge-Stone adder a critical component in high-speed computing systems.

8.1 Circuit Diagram

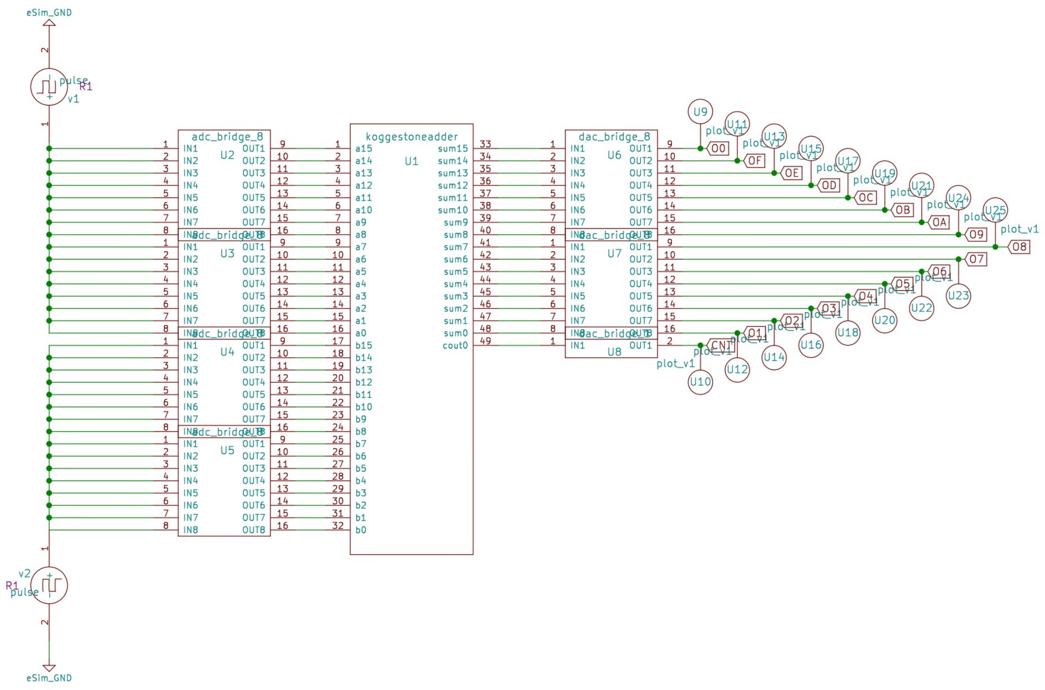
Figure 8.1: Circuit Diagram

Figure 8.1 illustrates the Kogge Stone Adder circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

8.2 Verilog Code

```
module KoggeStoneAdder(output [15:0] sum, output cout, input [15:0] a, b);
  wire cin = 1'b0;
  wire [15:0] c;
  wire [15:0] g, p;
  Square sq[15:0](g, p, a, b);
  // first line of circle
  wire [15:1] g2, p2;
  SmallCircle sc0_0(c[0], g[0]);
  BigCircle bc0[15:1](g2[15:1], p2[15:1], g[15:1], p[15:1], g[14:0], p[14:0]);
  // second line of circle
  wire [15:3] g3, p3;
  SmallCircle sc1[2:1](c[2:1], g2[2:1]);
  BigCircle bc1[15:3](g3[15:3], p3[15:3], g2[15:3], p2[15:3], g2[13:1],
  ↪  p2[13:1]);
  // third line of circle
  wire [15:7] g4, p4;
  SmallCircle sc2[6:3](c[6:3], g3[6:3]);
  BigCircle bc2[15:7](g4[15:7], p4[15:7], g3[15:7], p3[15:7], g3[11:3],
  ↪  p3[11:3]);
  // fourth line of circle
  wire [15:15] g5, p5;
  SmallCircle sc3[14:7](c[14:7], g4[14:7]);
  BigCircle bc3_15(g5[15], p5[15], g4[15], p4[15], g4[7], p4[7]);
  // fifth line of circle
  SmallCircle sc4_15(c[15], g5[15]);
  // last line - triangles
  Triangle tr0(sum[0], p[0], cin);
  Triangle tr[15:1](sum[15:1], p[15:1], c[14:0]);
  // generate cout
  buf #(1) (cout, c[15]);
endmodule

module BigCircle(output G, P, input Gi, Pi, GiPrev, PiPrev);
  wire e;
  and #(1) (e, Pi, GiPrev);
  or #(1) (G, e, Gi);
  and #(1) (P, Pi, PiPrev);
endmodule

module SmallCircle(output Ci, input Gi);
  buf #(1) (Ci, Gi);
endmodule

module Square(output G, P, input Ai, Bi);
  and #(1) (G, Ai, Bi);
  xor #(2) (P, Ai, Bi);
endmodule

module Triangle(output Si, input Pi, CiPrev);
  xor #(2) (Si, Pi, CiPrev);
endmodule
```

8.3 Output Waveform

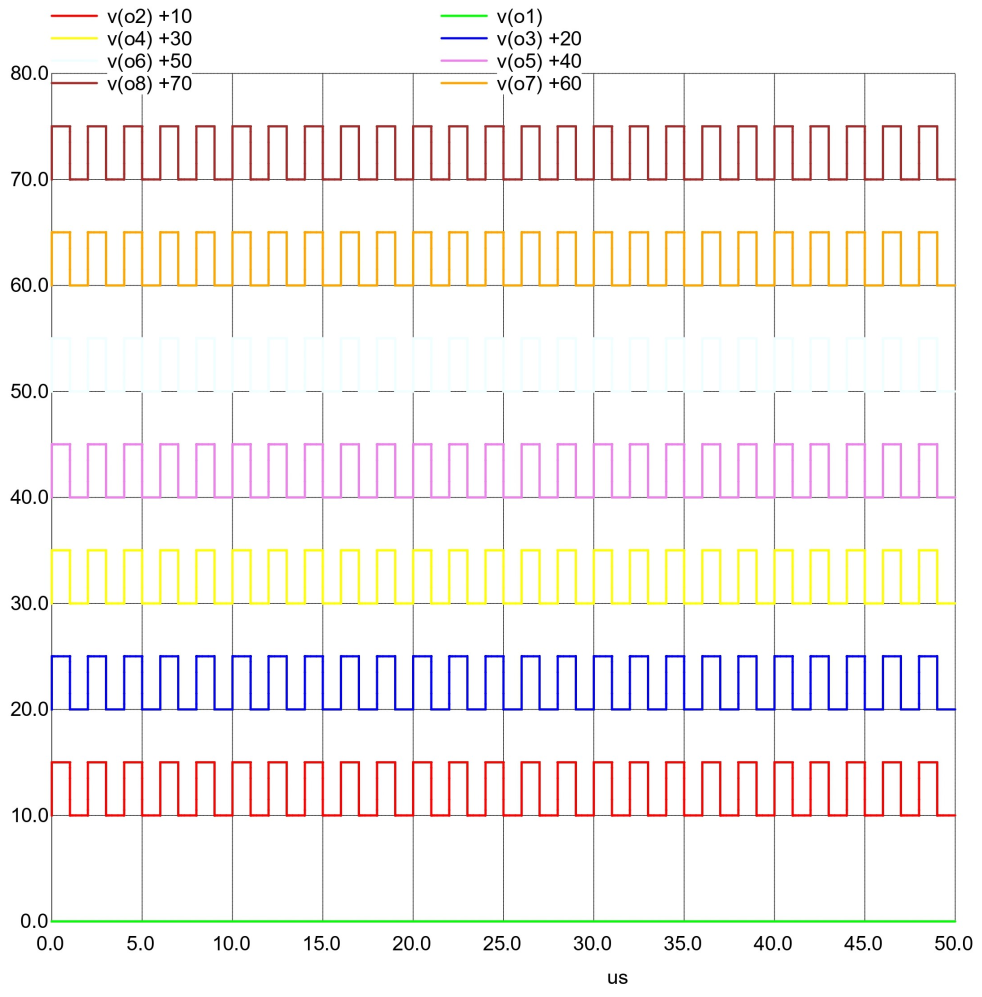

Figure 8.2: Least Significant Bit of Kogge Stone Adder

Figure 8.2 represents the output ”1111 1110”, which is considered the LSB (Least Significant Bit) of the output.

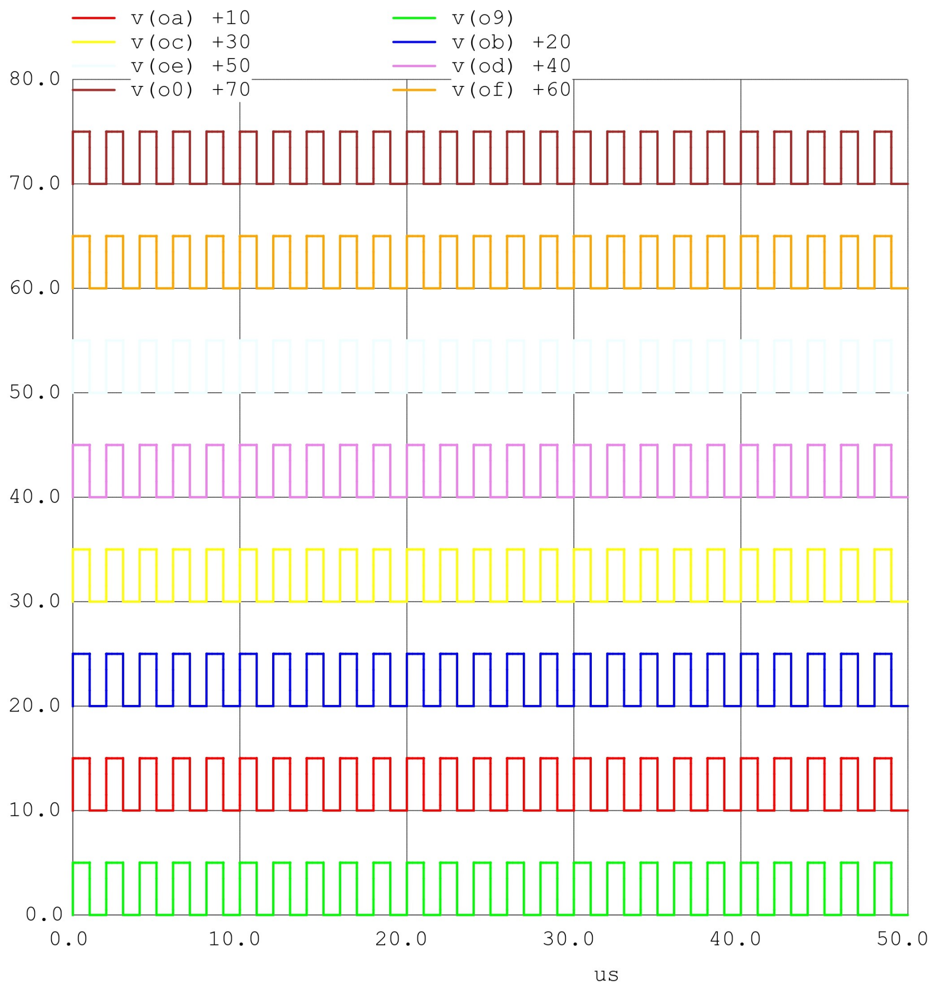

Figure 8.3: Most Significant Bit of Kogge Stone Adder

Figure 8.3 represents the output "1111 1111", which is considered the MSB (Most Significant Bit) of the output. Cumulatively the output will be " ".

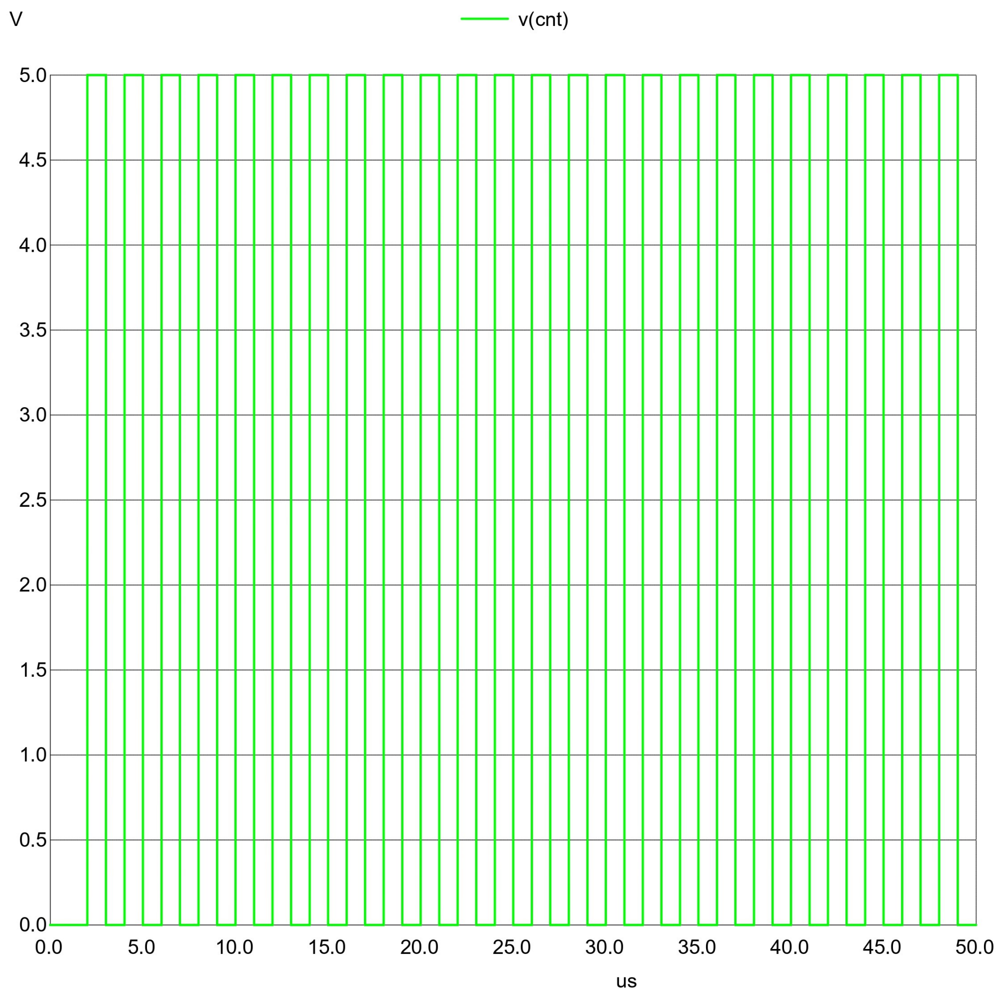

Figure 8.4: Carry bit

Figure 8.4 shows that if a Carry bit is present, the output will be generated from the Count pin.

Chapter 9

Sequence Detector

A sequence detector is a digital circuit designed to detect a specific sequence of bits within a stream of input data. It is commonly used in communication systems, digital signal processing, and control systems to identify patterns or signals that meet a predefined criterion. The sequence detector operates by comparing the incoming bit sequence with a predetermined sequence and generating an output signal when a match is found. The circuit typically consists of flip-flops, which store the state of the sequence being monitored, and combinational logic, which compares the stored sequence with the target sequence. The state transitions are managed by a finite state machine (FSM), which dictates how the circuit responds to each new bit in the input stream. The FSM ensures that the circuit can handle overlapping sequences and can reset or continue monitoring after detecting the target sequence.

9.1 Problem in implementing Sequence Detector using NgVeri

When implementing the sequence detector using NgVeri, an additional memory circuit is required to achieve the correct output, as it needs to detect either the past output or the present input.

9.2 Circuit Diagram

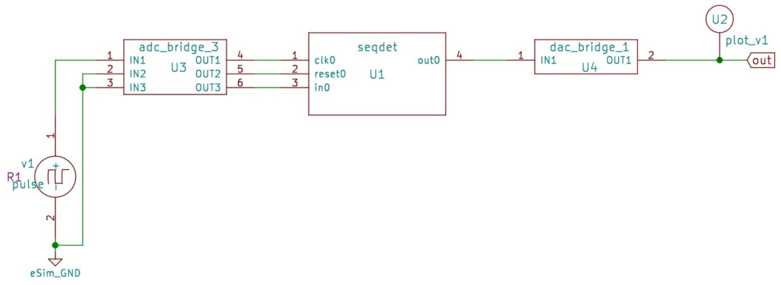

Figure 9.1: Circuit Diagram

Figure 9.1 illustrates the Sequence Detector circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

9.3 Verilog Code

```
module SeqDet (clk, reset, in, out);
parameter zero=0, one1=1, two1s=2;
output out; input clk, reset, in;
reg out; reg [1:0] state, next_state;
always @(posedge clk or posedge reset) begin
     if (reset)
     state <= zero;
     else
     state <= next_state;
     end
always @(state or in) begin
     case (state)
         zero: begin //last input was a zero out = 0;
         if (in)
            next_state=one1;
         else
            next_state=zero;
     end
     one1: begin //we've seen one 1 out = 0;
         if (in)
            next_state=two1s;
         else
            next_state=zero;
     end
     two1s: begin //we've seen at least 2 ones out = 1;
         if (in)
            next_state=two1s;
         else
            next_state=zero;
     end
     default: //in case we reach a bad state out = 0;
        next_state=zero;
     endcase
end
always @(state) begin
    case (state)
        zero: out <= 0;
        one1: out <= 0;
        two1s: out <= 1;
        default : out <= 0;
    endcase
    end
endmodule
```

9.4 Output Waveform

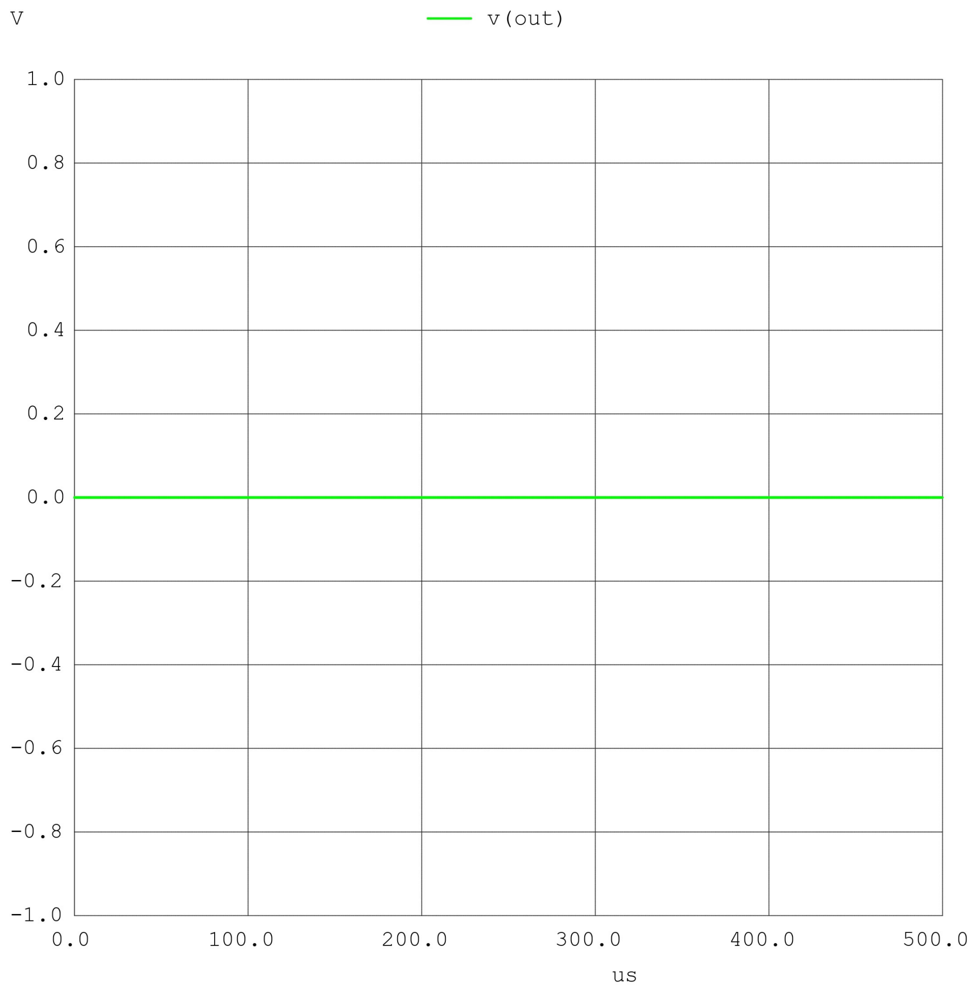

Figure 9.2: Sequence Detector Output

Figure 9.2 shows that if the required sequence is detected, the output pin generates an output as ”1” otherwise the output will be ”0”.

Chapter 10

8-Bit Microcontroller

An 8-bit microcontroller is a compact, integrated circuit designed to execute a wide range of control tasks in embedded systems. It features an 8-bit central processing unit (CPU), which means it can process 8 bits of data simultaneously, making it suitable for applications that require moderate computational power and efficiency. These microcontrollers are widely used in various devices, from home appliances and automotive systems to industrial machines and consumer electronics.

10.1 Circuit Diagram

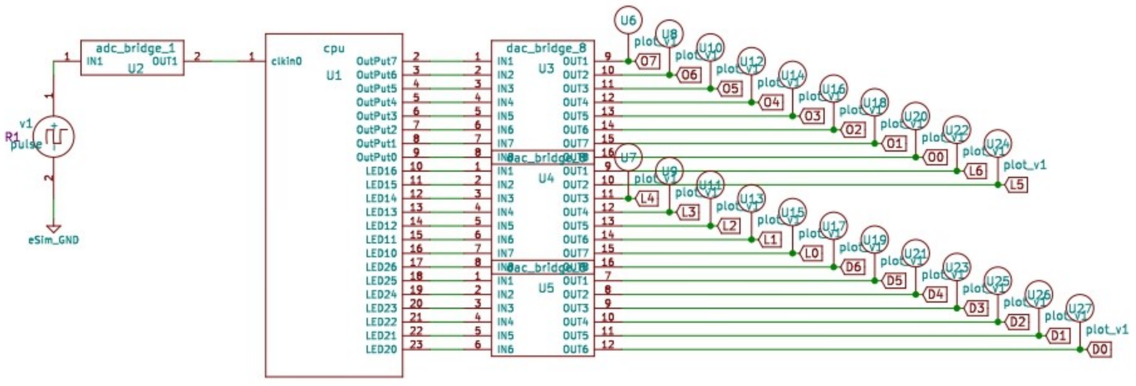

Figure 10.1: 8-Bit Microcontroller Circuit

Figure 10.1 illustrates the 8-Bit Microcontroller circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

10.2 Verilog Code

```
`include "ALU.v"
`include "PC.v"
`include "IC.v"
```

```
`include "register8.v"
`include "register4.v"
`include "tristateBuff.v"
`include "triBuff4.v"
`include "RAM.v"
`include "bcd2disp.v"
module CPU(input clkin, output [7:0] OutPut, output [6:0] LED1, output [6:0]
↪  LED2);
wire [7:0] bus;
wire [3:0] MemAddr;
wire [7:0] Aout;
wire [7:0] Bout;
wire [7:0] Instout;
wire [3:0] Pcount;
wire [3:0] Addr_in;
wire [7:0] Dispout;
wire [7:0] aluOut;
wire HLT, MI, RI, RO, IO, II, AI, AO, SO, SU, BI, OI, CE, CO, J;
wire PCrst, flag;
assign clk = (clkin & ~HLT);
register8 A(.clk(clk), .D(bus), .Q(Aout), .EI(AI));
tristateBuff triA(.data(Aout), .dataOut(bus), .enable(AO));
register8 B(.clk(clk), .D(bus), .Q(Bout), .EI(BI));
register8 InstReg(.clk(clk), .D(bus), .Q(Instout), .EI(II));
triBuff4 triInstReg(.data(Instout[3:0]), .dataOut(bus[3:0]), .enable(IO));
ALU alu(.A(Aout), .B(Bout), .op(SU), .res({flag,aluOut}));
tristateBuff tri_alu(.data(aluOut), .enable(SO), .dataOut(bus));
PC pc(.clk(clk), .rst(1'b0), .enable(CE), .jmp(J), .jmploc(bus[3:0]),
↪  .count(Pcount));
triBuff4 tripc(.data(Pcount), .dataOut(bus[3:0]), .enable(CO));
register4 MemAdd(.clk(clk), .D(bus[3:0]), .Q(Addr_in), .EI(MI));
RAM ram(.clk(~clk), .address(Addr_in), .write_enable(RI), .read_enable(RO),
↪  .data(bus));
IC ic(.clk(clk), .enable(1'b1), .Instruction(Instout[7:4]), .ctrl_wrd({HLT, MI,
↪  RI, RO, IO, II, AI, AO, SO, SU, BI, OI, CE, CO, J}));
register8 O(.clk(clk), .D(bus), .Q(OutPut), .EI(OI));
bcd2sevenseg seg0(.bcd(OutPut[3:0]), .seg(LED1));
bcd2sevenseg seg1(.bcd(OutPut[7:4]), .seg(LED2));
endmodule

module ALU(input op, input [7:0] A, input [7:0] B, output [8:0] res);
assign res[8:0] = op ? (A[7:0] - B[7:0]) : (A[7:0] + B[7:0]);
endmodule

module PC(input clk, input rst, input enable, input [3:0] jmploc, input jmp,
↪  output reg [3:0] count);
```

```
wire CLK;
assign CLK = (clk & enable);
initial begin
    count <= 4'b0000;
end
always @(posedge CLK)
begin
    if(rst)
    begin
        count <= 4'b0000;
    end
    else
    begin
        count <= count + 1;
    end
end

always @(posedge clk)
begin
    if(jmp)
    begin
        count <= jmploc;
    end
end
endmodule

module IC(input clk, input enable, input [3:0] Instruction, output reg [14:0]
↪   ctrl_wrd);
wire CLK;
assign CLK = (~clk & enable);
reg [2:0] Inst_count;
reg reset_in;
initial begin
    Inst_count <= 3'b111;
    reset_in <= 1'b0;
end
always @ (posedge CLK)
    begin
        Inst_count <= reset_in ? 3'b000 : Inst_count+1;
        reset_in <= 1'b0;
        case(Inst_count)//HLT, MI, RI, RO, IO, II, AI, AO, SO, SU, BI, OI, CE,
        ↪   CO, J;
            3'b000: ctrl_wrd <= 15'b010000000000010;
            3'b001: ctrl_wrd <= 15'b000101000000100;
            3'b010: begin
                    case(Instruction)
                    4'b0001: ctrl_wrd <= 15'b010010000000000; // LDA
                    4'b0010: ctrl_wrd <= 15'b010010000000000; //ADD
                    4'b0011: ctrl_wrd <= 15'b010010000000000;  //SUBT
                    4'b1110: ctrl_wrd <= 15'b000000010001000; // OUT
                    4'b0100: ctrl_wrd <= 15'b000010000000001; // JMP
                    4'b1111: ctrl_wrd <= 15'b100000000000000; // HLT
                    default: ctrl_wrd <= 15'b000000000000000;
                    endcase
                    end
            3'b011: begin
                    case(Instruction)
                    4'b0001: ctrl_wrd <= 15'b000100100000000; // LDA
                    4'b0010: ctrl_wrd <= 15'b000100000010000; //ADD
                    4'b0011: ctrl_wrd <= 15'b000100000010000; //SUBT
                    4'b1110: ctrl_wrd <= 15'b000000000000000; // OUT
                    4'b0100: ctrl_wrd <= 15'b000000000000000; // JMP
                    4'b1111: ctrl_wrd <= 15'b000000000000000; // HLT
                    default: ctrl_wrd <= 15'b000000000000000;
                    endcase
                    end
            3'b100: begin
                    case(Instruction)
                    4'b0001: ctrl_wrd <= 15'b000000000000000; // LDA done
                    4'b0010: ctrl_wrd <= 15'b000000101000000; //ADD
                    4'b0011: ctrl_wrd <= 15'b000000101100000; //SUBT
                    4'b1110: ctrl_wrd <= 15'b000000000000000; // OUT
                    4'b0100: ctrl_wrd <= 15'b000000000000000; // JMP
                    4'b1111: ctrl_wrd <= 15'b000000000000000; // HLT
                    default: ctrl_wrd <= 15'b000000000000000;
                    endcase
                    reset_in <= 1'b1; // Have some doubt here..... Maybe this
                    ↪  part can go wrong .....
                    end
            default: ctrl_wrd <= 15'b000000000000000;
        endcase
    end
endmodule

module register8(input clk, input [7:0] D, input EI, output reg [7:0] Q);
wire CLK;
assign CLK=(clk & EI);
always @(posedge CLK)
begin
    Q <= D;
end
endmodule
```

```
module register4(input clk, input [3:0] D, input EI, output reg [3:0] Q);
wire CLK;
assign CLK=(clk & EI);
always @(posedge CLK)
begin
    Q <= D;
end
endmodule

module tristateBuff(input [7:0] data, input enable, output [7:0] dataOut);
    assign dataOut = enable ? data : 8'bzzzzzzzz;
endmodule

module triBuff4(input [3:0] data, input enable, output [3:0] dataOut);
    assign dataOut = enable ? data : 4'bzzzz;
endmodule

module RAM(input clk, input [3:0] address, input write_enable, input read_enable,
↪   inout [7:0] data);
reg [7:0] Memory[15:0];
reg [7:0] buffer;
initial begin
    Memory[0] <= 8'b0001_1010;
    Memory[1] <= 8'b0010_1011;
    Memory[2] <= 8'b0100_0101;
    Memory[3] <= 8'b0011_1100;
    Memory[4] <= 8'b0010_1101;
    Memory[5] <= 8'b1110_0000;
    Memory[6] <= 8'b0001_1110;
    Memory[7] <= 8'b0010_1111;
    Memory[8] <= 8'b1110_0000;
    Memory[9] <= 8'b1111_0000;
    Memory[10] <= 8'b0000_0011;
    Memory[11] <= 8'b0000_0010;
    Memory[12] <= 8'b0000_0001;
    Memory[13] <= 8'b0000_0101;
    Memory[14] <= 8'b0000_1010;
    Memory[15] <= 8'b0000_1011;
end
always @(posedge clk)
begin
    if(write_enable & ~read_enable)
    begin
        Memory[address] <= data;
    end
    else
    begin
        buffer <= Memory[address];
    end
end
assign data = (read_enable & ~write_enable) ? buffer : 8'bzzzzzzzz;
endmodule

module bcd2sevenseg(
    input [3:0] bcd,
    output reg [6:0] seg
    );
always @(bcd)
begin
    case (bcd)
        0 : seg <= 7'b1111110;
        1 : seg <= 7'b0110000;
        2 : seg <= 7'b1101101;
        3 : seg <= 7'b1111001;
        4 : seg <= 7'b0110011;
        5 : seg <= 7'b1011011;
        6 : seg <= 7'b1011111;
        7 : seg <= 7'b1110000;
        8 : seg <= 7'b1111111;
        9 : seg <= 7'b1111011;
        10 : seg <= 7'b1110111;
        11 : seg <= 7'b0011111;
        12 : seg <= 7'b1001110;
        13 : seg <= 7'b0111101;
        14 : seg <= 7'b1001111;
        15 : seg <= 7'b1000111;
        default : seg <= 7'b0000000;
    endcase
end
endmodule
```

10.3 Output Waveform

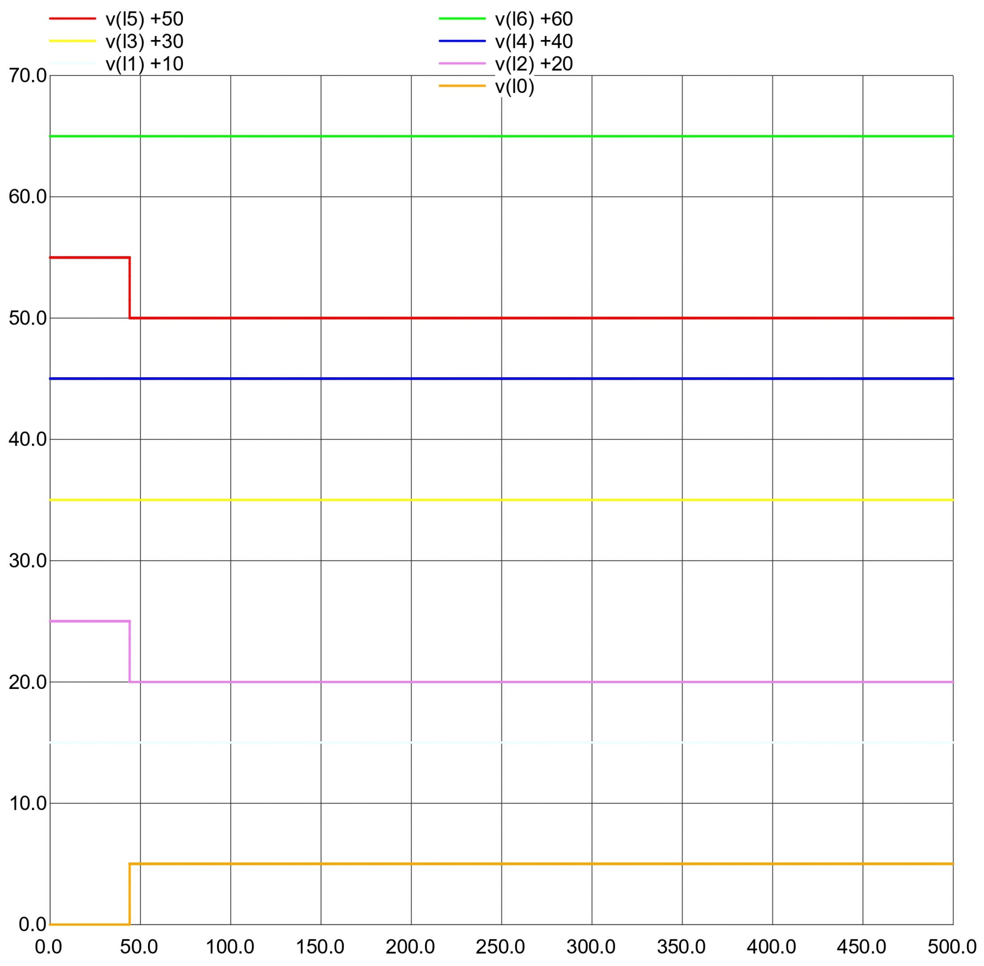

Figure 10.2: Circuit Diagram

Figure 10.2 represents the output generated from LED port 1.

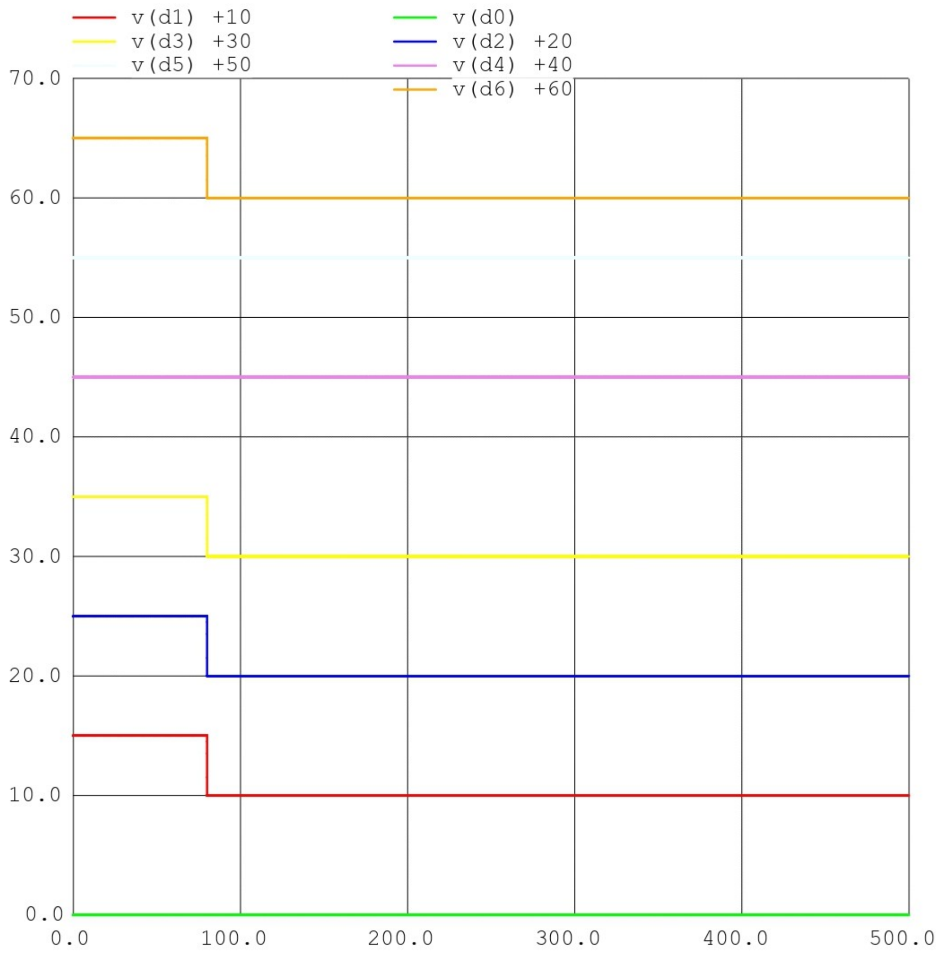

Figure 10.3: Circuit Diagram

Figure 10.3 represents the output generated from LED port 2.

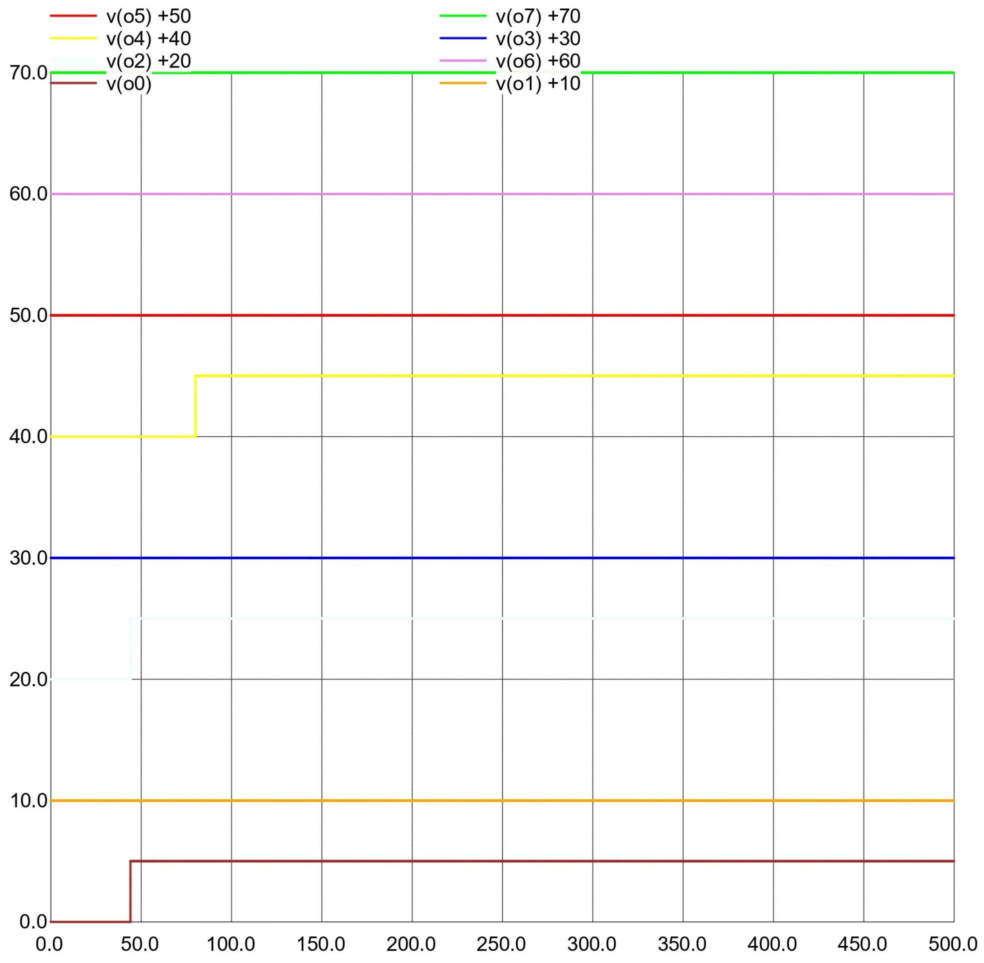

Figure 10.4: Circuit Diagram

Figure 10.4 represents the output generated from the Output Port.

Chapter 11

Single Instruction Multiple Data (SIMD)

Single Instruction, Multiple Data (SIMD) is a parallel computing architecture where a single operation is simultaneously applied to multiple data points. This approach is highly efficient for tasks involving large datasets that require the same operation across all elements, such as in graphics processing or scientific computations. By executing the same instruction across multiple data elements at once, SIMD significantly reduces the time needed for repetitive processing tasks. This architecture is widely used in applications like image processing, video encoding, and cryptography, where processing large amounts of data quickly is essential. SIMD's ability to handle multiple data streams in parallel makes it a powerful tool for improving computational efficiency in various domains.

11.1 Problem in implementing Single Instruction Multiple Data using NgVeri

While implementing SIMD using eSim and NgVeri, the expected output could not be achieved. However, using alternative software like Vivado, the correct output can be obtained.

11.2 Circuit Diagram

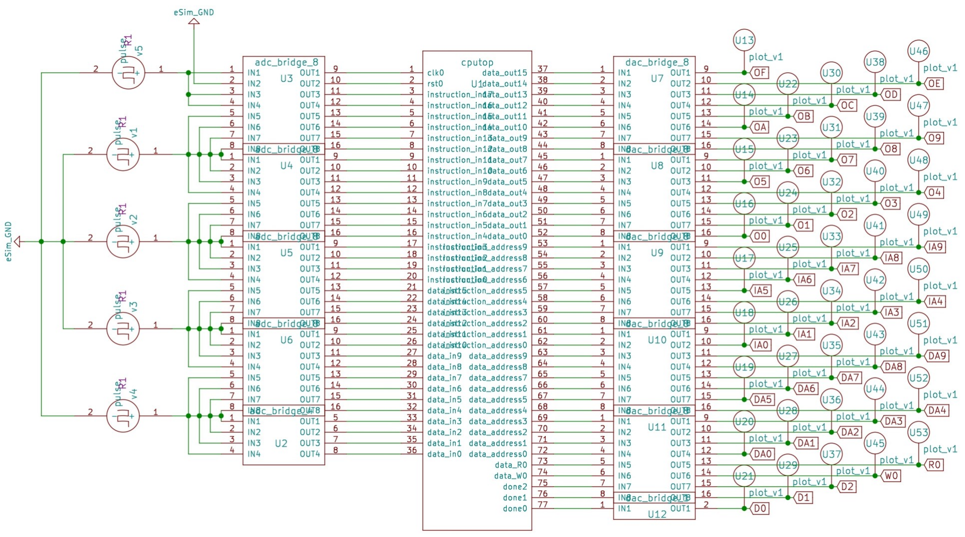

Figure 11.1: SIMD Circuit

Figure 11.1 illustrates the Single Instruction Multiple Data circuit constructed using Verilog code, with the corresponding Verilog code provided below for reference.

11.3 Verilog Code

```
module CPUtop(
        input clk,
        input rst,
        input [17:0] instruction_in,
        input [15:0] data_in,
        output [15:0] data_out,
        output [9:0] instruction_address,
        output [9:0] data_address,
        output data_R,
        output data_W,
        output done
    );
    wire [5:0] opcode = instruction_in[17:12];
    parameter [2:0] STATE_IDLE = 0;
    parameter [2:0] STATE_IF = 1;
    parameter [2:0] STATE_ID = 2;
    parameter [2:0] STATE_EX = 3;
    parameter [2:0] STATE_MEM = 4;
    parameter [2:0] STATE_WB = 5;
    parameter [2:0] STATE_HALT = 6;
```

```
reg [2:0] current_state;
reg [9:0] PC,next_PC;
reg [9:0] current_data_address;
reg rdata_en;
reg wdata_en;
reg [15:0] data_out_reg;
assign data_out = data_out_reg;
assign data_R = rdata_en;
assign data_W = wdata_en;
assign data_address = current_data_address;
reg [15:0] H[0:3];
reg [15:0] Oset[0:2];
reg [15:0] Qset[0:2];
reg [9:0]  LC;
reg [9:0] im_reg;
reg CMD_addition;
reg CMD_multiplication;
reg CMD_substruction;
reg CMD_mul_accumulation;
reg CMD_logic_shift_right;
reg CMD_logic_shift_left;
reg CMD_and;
reg CMD_or;
reg CMD_not;
reg CMD_load;
reg CMD_store;
reg CMD_set;
reg CMD_loopjump;
reg CMD_setloop;
reg [15:0] result_reg_add;
reg [15:0] result_reg_sub;
reg [15:0] result_reg_mul;
reg [15:0] result_reg_mac;
reg [15:0] result_reg_Lshift;
reg [15:0] result_reg_Rshift;
reg [15:0] result_reg_and;
reg [15:0] result_reg_or;
reg [15:0] result_reg_not;
reg [15:0] result_reg_load;
reg [15:0] result_reg_store;
reg [15:0] result_reg_set;
reg [1:0] R0,R1,R2,R3;
wire [15:0] comp_A = Hreg1?(H[R0]):((Hreg2|Hreg3)?(H[R2]):(Oreg1?(Oset[R0]):
    ((Oreg2|Oreg3)?(Oset[R2]):(Qreg1?(Qset[R0]):((Qset[R2]))))));
wire [15:0] comp_B = Hreg1?(im_reg):((Hreg2|Hreg3)?(H[R3]):(Oreg1?
    ({im_reg[7:0],im_reg[7:0]}):((Oreg2|Oreg3)?(Oset[R3]):
```

```
    (Qreg1({im_reg[3:0],im_reg[3:0],im_reg[3:0],
im_reg[3:0]}):((Qset[R3]))))));
wire [15:0] Add_Cout;
wire [15:0] Mul_Cout;
wire [15:0] MAC_Cout;
wire [15:0] MAC_A = Hreg3?(H[R1]):(Oreg3?(Oset[R1]):(Qset[R1]));
wire [15:0] MAC_B = Mul_Cout;
SIMDadd Add(
        .A(CMD_mul_accumulation?MAC_A:comp_A),
        .B(CMD_mul_accumulation?MAC_B:comp_B),
        .H(Hreg1|Hreg2|Hreg3),
        .O(Oreg1|Oreg2|Oreg3),
        .Q(Qreg1|Qreg2|Qreg3),
        .sub(CMD_substruction),
        .Cout(Add_Cout)
    );
wire [15:0] shiftin = Hreg1?(H[R3]):(Oreg1?(Oset[R3]):(Qset[R3]));
wire [15:0] shiftout;
SIMDshifter shift(
            .shiftin(shiftin),
            .H(Hreg1),
            .O(Oreg1),
            .Q(Qreg1),
            .left(CMD_logic_shift_left),
            .shiftout(shiftout)
        );
SIMDmultiply Mul(
            .mul_a(comp_A),
            .mul_b(comp_B),
            .H(Hreg1|Hreg2|Hreg3),
            .O(Oreg1|Oreg2|Oreg3),
            .Q(Qreg1|Qreg2|Qreg3),
            .mulout(Mul_Cout)
            );
assign instruction_address = PC;
assign done = current_state == STATE_HALT;
always @(posedge clk)//state machine
begin
    if (rst)
    begin
        current_state <= STATE_IDLE;
        PC <= 0;
    end
    else
    begin
        if (opcode == 63) current_state <= STATE_HALT;
        else
        if (current_state == STATE_IDLE)
        begin
            current_state <= STATE_IF;
        end
        else if (current_state == STATE_IF)
        begin
            $display("======== another instruction ========");
            $display("H00: %b",H[0]);
            $display("H01: %b",H[1]);
            $display("H10: %b",H[2]);
            $display("H11: %b",H[3]);
            $display("Oset00: %b",Oset[0]);
            $display("Oset01: %b",Oset[1]);
            $display("Oset10: %b",Oset[2]);
            $display("Qset00: %b",Qset[0]);
            $display("Qset01: %b",Qset[1]);
            $display("Qset10: %b",Qset[2]);
            $display("           -- execute --             ");
            current_state <= STATE_ID;
        end
        else if (current_state == STATE_ID)
        begin
            current_state <= STATE_EX;
        end
        else if (current_state == STATE_EX)
        begin
            current_state <= STATE_MEM;
        end
        else if (current_state == STATE_MEM)
        begin
            current_state <= STATE_WB;
        end
        else if (current_state == STATE_WB)
        begin
            current_state <= STATE_IF;
            PC <= next_PC;
        end
    end
end
always @(posedge clk)//STATE_IF
begin
    if (rst)
    begin
    end
    else
    begin
    end
end
always @(posedge clk)//STATE_ID
begin
    if (rst || current_state == STATE_IDLE || current_state == STATE_IF)
    begin
        CMD_addition <= 0;
        CMD_multiplication <= 0;
        CMD_substruction <= 0;
        CMD_mul_accumulation <= 0;
        CMD_logic_shift_right <= 0;
        CMD_logic_shift_left <= 0;
        CMD_and <= 0;
        CMD_or <= 0;
        CMD_not <= 0;
        CMD_load <= 0;
        CMD_store <= 0;
        CMD_set <= 0;
        CMD_loopjump <= 0;
        CMD_setloop <= 0;
        Hreg1<=0;
        Hreg2<=0;
        Hreg3<=0;
        Him<=0;
        Oreg1<=0;
        Oreg2<=0;
        Oreg3<=0;
        Oim<=0;
        Qreg1<=0;
        Qreg2<=0;
        Qreg3<=0;
        Qim<=0;
        im_reg <= 10'b0000000000;
        R0 <= 0;
        R1 <= 0;
        R2 <= 0;
        R3 <= 0;
    end
    else
    begin
        if (current_state == STATE_ID)
        begin
            CMD_addition <= (opcode<=5);
            CMD_substruction <= (opcode>=6)&&(opcode<=11);
            CMD_multiplication <= (opcode>=12)&&(opcode<=17);
```

```
                CMD_mul_accumulation <= (opcode>=18)&&(opcode<=20);
                CMD_logic_shift_left <= (opcode>=21)&&(opcode<=23);
                CMD_logic_shift_right <= (opcode>=24)&&(opcode<=26);
                CMD_and <= (opcode>=27)&&(opcode<=29);
                CMD_or <= (opcode>=30)&&(opcode<=32);
                CMD_not <= (opcode>=33)&&(opcode<=35);
                CMD_loopjump <= opcode==36;
                CMD_setloop <= opcode==37;
                CMD_load <= (opcode>=38)&&(opcode<=40);
                CMD_store <= (opcode>=41)&&(opcode<=43);
                CMD_set <= (opcode>=44)&&(opcode<=46);
                Hreg1<=(opcode==3)||(opcode==9)||(opcode==15)||(opcode==21)||
                    (opcode==24)||(opcode==33)||(opcode==38)||(opcode==41)||
                    (opcode==44);
                Hreg2<=(opcode==0)||(opcode==6)||(opcode==12)||(opcode==27)||
                    (opcode==30);
                Hreg3<=(opcode==18);
                Him<=(opcode==3)||(opcode==9)||(opcode==15)||(opcode==38)||
                    (opcode==41)||(opcode==44);
                Oreg1<=(opcode==4)||(opcode==10)||(opcode==16)||(opcode==22)||
                    (opcode==25)||(opcode==34)||(opcode==39)||(opcode==42)||
                    (opcode==45);
                Oreg2<=(opcode==1)||(opcode==7)||(opcode==13)||(opcode==28)||
                    (opcode==31);
                Oreg3<=(opcode==19);
                Oim<=(opcode==4)||(opcode==10)||(opcode==16)||(opcode==39)||
                    (opcode==42)||(opcode==45);
                Qreg1<=(opcode==5)||(opcode==11)||(opcode==17)||(opcode==23)||
                    (opcode==26)||(opcode==35)||(opcode==40)|| (opcode==43)||
                    (opcode==46);
                Qreg2<=(opcode==2)||(opcode==8)||(opcode==14)||(opcode==29)||
                    (opcode==32);
                Qreg3<=(opcode==20);
                Qim<=(opcode==5)||(opcode==11)||(opcode==17)||(opcode==40)||
                    (opcode==43)||(opcode==46);
                im_reg <= instruction_in[9:0];
                R0 <= instruction_in[11:10];
                R1 <= instruction_in[5:4];
                R2 <= instruction_in[3:2];
                R3 <= instruction_in[1:0];
                $display("PC: %0d : instruction = %b", PC,instruction_in);
            end
        end
end
always @(posedge clk)//STATE_EX
begin
```

```
if (rst || current_state == STATE_IDLE || current_state == STATE_IF)
begin
    result_reg_add <= 0;
    result_reg_sub <= 0;
    result_reg_mul <= 0;
    result_reg_mac <= 0;
    result_reg_Lshift <= 0;
    result_reg_Rshift <= 0;
    result_reg_and <= 0;
    result_reg_or <= 0;
    result_reg_not <= 0;
    result_reg_set <= 0;
    current_data_address <= 0;
    rdata_en <= 0;
    wdata_en <= 0;
    if (rst)
    begin
        next_PC <= 0;
    end
end
else if (current_state == STATE_EX)
begin
    if (CMD_addition) // do addition
    begin
        result_reg_add <= Add_Cout;
        if (Hreg2)
        begin
            $display("add16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
        end
        else if (Oreg2)
        begin
            $display("add8bit R%d=%b R%d=%b", R2,Oset[R2],R3,Oset[R3]);
        end
        else if (Qreg2)
        begin
            $display("add4bit R%d=%b R%d=%b", R2,Qset[R2],R3,Qset[R3]);
        end
        else if (Him)
        begin
            $display("add16bit R%d=%b im=%b", R0,H[R0],im_reg);
        end
        else if (Oim)
        begin
            $display("add8bit R%d=%b im=%b", R0,Oset[R0],im_reg);
        end
        else if (Qim)
        begin
            $display("add4bit R%d=%b im=%b", R0,Qset[R0],im_reg);
        end
    end
    else if (CMD_substruction) // do substruction
    begin
        result_reg_sub <= Add_Cout;
        if (Hreg2)
        begin
            $display("sub16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
        end
        else if (Oreg2)
        begin
            $display("sub8bit R%d=%b R%d=%b", R2,Oset[R2],R3,Oset[R3]);
        end
        else if (Qreg2)
        begin
            $display("sub4bit R%d=%b R%d=%b", R2,Qset[R2],R3,Qset[R3]);
        end
        else if (Him)
        begin
            $display("sub16bit R%d=%b im=%b", R0,H[R0],im_reg);
        end
        else if (Oim)
        begin
            $display("sub8bit R%d=%b im=%b", R0,Oset[R0],im_reg);
        end
        else if (Qim)
        begin
            $display("sub4bit R%d=%b im=%b", R0,Qset[R0],im_reg);
        end
    end
    else if (CMD_multiplication) // do multiplication
    begin
        result_reg_mul<=Mul_Cout;
        if (Hreg2)
        begin
            $display("mul16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
        end
        else if (Oreg2)
        begin
            $display("mul8bit R%d=%b R%d=%b", R2,Oset[R2],R3,Oset[R3]);
        end
        else if (Qreg2)
        begin
            $display("mul4bit R%d=%b R%d=%b", R2,Qset[R2],R3,Qset[R3]);
        end
        else if (Him)
        begin
            $display("mul16bit R%d=%b im=%b", R0,H[R0],im_reg);
        end
        else if (Oim)
        begin
            $display("mul8bit R%d=%b im=%b", R0,Oset[R0],im_reg);
        end
        else if (Qim)
        begin
            $display("mul4bit R%d=%b im=%b", R0,Qset[R0],im_reg);
        end
    end
    else if (CMD_mul_accumulation) // do mac
    begin
        result_reg_mac <= Add_Cout;
        if (Hreg3)
        begin
            $display("MAC16bit R%d=%b R%d=%b R%d=%b",
            ↪  R1,H[R1],R2,H[R2],R3,H[R3]);
        end
        else if (Oreg3)
        begin
            $display("MAC8bit R%d=%b R%d=%b R%d=%b",
            ↪  R1,Oset[R1],R2,Oset[R2],R3,Oset[R3]);
        end
        else if (Qreg3)
        begin
            $display("MAC4bit R%d=%b R%d=%b R%d=%b",
            ↪  R1,Qset[R1],R2,Qset[R2],R3,Qset[R3]);
        end
    end
    else if (CMD_logic_shift_right) // do shiftin right
    begin
        result_reg_Rshift <= shiftout;
        if (Hreg1)
        begin
            $display("Rshift16bit R%d=%b", R3,H[R3]);
        end
        else if (Oreg1)
        begin
            $display("Rshift8bit R%d=%b", R3,Oset[R3]);
        end
        else if (Qreg1)
        begin
```

```
            $display("Rshift4bit R%d=%b", R3,Qset[R3]);
        end
    end
    else if (CMD_logic_shift_left) // do shiftin left
    begin
        result_reg_Lshift <= shiftout;
        if (Hreg1)
        begin
            $display("Lshift16bit R%d=%b", R3,H[R3]);
        end
        else if (Oreg1)
        begin
            $display("Lshift8bit R%d=%b", R3,Oset[R3]);
        end
        else if (Qreg1)
        begin
            $display("Lshift4bit R%d=%b", R3,Qset[R3]);
        end
    end
    else if (CMD_and) // do and
    begin
        if (Hreg2)
        begin
            result_reg_and <= H[R2] & H[R3];
            $display("and16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
        end
        else if (Oreg2)
        begin
            result_reg_and <= Oset[R2] & Oset[R3];
            $display("and8bit R%d=%b R%d=%b", R2,Oset[R2],R3,Oset[R3]);
        end
        else if (Qreg2)
        begin
            result_reg_and <= Qset[R2] & Qset[R3];
            $display("and4bit R%d=%b R%d=%b", R2,Qset[R2],R3,Qset[R3]);
        end
    end
    else if (CMD_or) // do or
    begin
        if (Hreg2)
        begin
            result_reg_or <= H[R2] | H[R3];
            $display("or16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
        end
        else if (Oreg2)
        begin
            result_reg_or <= Oset[R2] | Oset[R3];
            $display("or8bit R%d=%b R%d=%b", R2,Oset[R2],R3,Oset[R3]);
        end
        else if (Qreg2)
        begin
            result_reg_or <= Qset[R2] | Qset[R3];
            $display("or4bit R%d=%b R%d=%b", R2,Qset[R2],R3,Qset[R3]);
        end
    end
    else if (CMD_not) // do not
    begin
        if (Hreg1)
        begin
            result_reg_not <= ~H[R3];
            $display("not16bit R%d=%b", R3,H[R3]);
        end
        else if (Oreg1)
        begin
            result_reg_not <= ~Oset[R3];
            $display("not8bit R%d=%b", R3,Oset[R3]);
        end
        else if (Qreg1)
        begin
            result_reg_not <= ~Qset[R3];
            $display("not4bit R%d=%b", R3,Qset[R3]);
        end
    end
    else if (CMD_set) // do set
    begin
        if (Hreg1)
        begin
            result_reg_set <= im_reg;
            $display("set16bit R%d im=%b", R0,im_reg);
        end
        else if (Oreg1)
        begin
            result_reg_set[7:0] <= im_reg;
            result_reg_set[15:8] <= im_reg;
            $display("set8bit R%d im=%b", R0,im_reg);
        end
        else if (Qreg1)
        begin
            result_reg_set[3:0] <= im_reg;
            result_reg_set[7:4] <= im_reg;
            result_reg_set[11:8] <= im_reg;
            result_reg_set[15:12] <= im_reg;
            $display("set4bit R%d im=%b", R0,im_reg);
        end
    end
    else if (CMD_load) // do load
    begin
        rdata_en <= 1;
        current_data_address <= im_reg;
        if (Hreg1)
        begin
            $display("load16bit R%d im=%b", R0,im_reg);
        end
        else if (Oreg1)
        begin
            $display("load8bit R%d im=%b", R0,im_reg);
        end
        else if (Qreg1)
        begin
            $display("load4bit R%d im=%b", R0,im_reg);
        end
    end
    else if (CMD_store) // do store
    begin
        wdata_en <= 1;
        rdata_en <= 1;
        current_data_address <= im_reg;
        if (Hreg1)
        begin
            data_out_reg <= H[R0];
            $display("store16bit R%d=%b im=%b", R0,H[R0],im_reg);
        end
        else if (Oreg1)
        begin
            data_out_reg <= Oset[R0];
            $display("store8bit R%d=%b im=%b", R0,Oset[R0],im_reg);
        end
        else if (Qreg1)
        begin
            data_out_reg <= Qset[R0];
            $display("store4bit R%d=%b im=%b", R0,Qset[R0],im_reg);
        end
    end
    if (CMD_loopjump)
    begin
        $display("loopjump LC=%d im=%d", LC,im_reg);
        if (LC != 0)
        begin
```

```
                next_PC <= im_reg;
                LC <= LC - 1;
            end
            else
            begin
                next_PC <= next_PC + 1;
            end
        end
        else
            next_PC <= next_PC + 1;
        if (CMD_setloop)
        begin
            $display("setloop im=%d", im_reg);
            LC <= im_reg;
        end
    end
end
always @(posedge clk)//STATE_MEM
begin
    if (rst || current_state == STATE_IDLE || current_state == STATE_IF)
    begin
    end
    else
    begin
        if (current_state == STATE_MEM)
        begin

        end
    end
end
always @(posedge clk)//STATE_WB
begin
    if (rst || current_state == STATE_IDLE || current_state == STATE_IF)
    begin
    end
    else if (current_state == STATE_WB)
    begin
        if (CMD_addition) // do addition
        begin
            if (Hreg2)
            begin
                H[R2] <= result_reg_add;
            end
            else if (Oreg2)
            begin
                Oset[R2] <= result_reg_add;
            end
            else if (Qreg2)
            begin
                Qset[R2] <= result_reg_add;
            end
            else if (Him)
            begin
                H[R0] <= result_reg_add;
            end
            else if (Oim)
            begin
                Oset[R0] <= result_reg_add;
            end
            else if (Qim)
            begin
                Qset[R0] <= result_reg_add;
            end
        end
        else if (CMD_substruction) // do substruction
        begin
            if (Hreg2)
            begin
                H[R2] <= result_reg_sub;
            end
            else if (Oreg2)
            begin
                Oset[R2] <= result_reg_sub;
            end
            else if (Qreg2)
            begin
                Qset[R2] <= result_reg_sub;
            end
            else if (Him)
            begin
                H[R0] <= result_reg_sub;
            end
            else if (Oim)
            begin
                Oset[R0] <= result_reg_sub;
            end
            else if (Qim)
            begin
                Qset[R0] <= result_reg_sub;
            end
        end
    end
```

```
else if (CMD_multiplication) // do multiplication
begin
    if (Hreg2)
    begin
        H[R2] <= result_reg_mul;
    end
    else if (Oreg2)
    begin
        Oset[R2] <= result_reg_mul;
    end
    else if (Qreg2)
    begin
        Qset[R2] <= result_reg_mul;
    end
    else if (Him)
    begin
        H[R0] <= result_reg_mul;
    end
    else if (Oim)
    begin
        Oset[R0] <= result_reg_mul;
    end
    else if (Qim)
    begin
        Qset[R0] <= result_reg_mul;
    end
end
else if (CMD_mul_accumulation) // do mac
begin
    if (Hreg3)
    begin
        H[R1] <= result_reg_mac;
    end
    else if (Oreg3)
    begin
        Oset[R1] <= result_reg_mac;
    end
    else if (Qreg3)
    begin
        Qset[R1] <= result_reg_mac;
    end
end
else if (CMD_logic_shift_right) // do shiftin right
begin
    if (Hreg1)
    begin
```

```
        H[R3] <= result_reg_Rshift;
    end
    else if (Oreg1)
    begin
        Oset[R3] <= result_reg_Rshift;
    end
    else if (Qreg1)
    begin
        Qset[R3] <= result_reg_Rshift;
    end
end
else if (CMD_logic_shift_left) // do shiftin left
begin
    if (Hreg1)
    begin
        H[R3] <= result_reg_Lshift;
    end
    else if (Oreg1)
    begin
        Oset[R3] <= result_reg_Lshift;
    end
    else if (Qreg1)
    begin
        Qset[R3] <= result_reg_Lshift;
    end
end
else if (CMD_and) // do and
begin
    if (Hreg2)
    begin
        H[R2] <= result_reg_and;
    end
    else if (Oreg2)
    begin
        Oset[R2] <= result_reg_and;
    end
    else if (Qreg2)
    begin
        Qset[R2] <= result_reg_and;
    end
end
else if (CMD_or) // do or
begin
    if (Hreg2)
    begin
        H[R2] <= result_reg_or;
    end
    else if (Oreg2)
    begin
        Oset[R2] <= result_reg_or;
    end
    else if (Qreg2)
    begin
        Qset[R2] <= result_reg_or;
    end
end
else if (CMD_not) // do not
begin
    if (Hreg1)
    begin
        H[R3] <= result_reg_not;
    end
    else if (Oreg1)
    begin
        Oset[R3] <= result_reg_not;
    end
    else if (Qreg1)
    begin
        Qset[R3] <= result_reg_not;
    end
end
else if (CMD_set) // do set
begin
    if (Hreg1)
    begin
        H[R0] <= result_reg_set;
    end
    else if (Oreg1)
    begin
        Oset[R0] <= result_reg_set;
    end
    else if (Qreg1)
    begin
        Qset[R0] <= result_reg_set;
    end
end
else if (CMD_load)
begin
    if (Hreg1)
    begin
        H[R0] <= data_in;
    end
```

```
            else if (Oreg1)
            begin
                Oset[R0] <= data_in;
            end
            else if (Qreg1)
            begin
                Qset[R0] <= data_in;
            end
        end
    end
endmodule

module SIMDadd(
        input [15:0] A,
        input [15:0] B,
        input H,
        input O,
        input Q,
        input sub,
        output [15:0] Cout
    );
    wire [15:0] B_real = sub?(~B):B;
    wire [4:0] C0 = A[3:0]   +  B_real[3:0]    + sub;
    wire [4:0] C1 = A[7:4]   +  B_real[7:4]    + (C0[4]&(O|H)) + (Q&sub);
    wire [4:0] C2 = A[11:8]  +  B_real[11:8]   + (C1[4]&H)     + ((Q|O)&sub);
    wire [4:0] C3 = A[15:12] +  B_real[15:12]  + (C2[4]&(O|H)) + (Q&sub);
    assign Cout = {C3[3:0],C2[3:0],C1[3:0],C0[3:0]};
endmodule

module SIMDmultiply(
        input [15:0] mul_a,
        input [15:0] mul_b,
        input H,
        input O,
        input Q,
        output [15:0] mulout);
    wire [15:0] sel0 = H?16'hFFFF:(O?16'h00FF:16'h000F);
    wire [15:0] sel1 = H?16'hFFFF:(O?16'h00FF:16'h00F0);
    wire [15:0] sel2 = H?16'hFFFF:(O?16'hFF00:16'h0F00);
    wire [15:0] sel3 = H?16'hFFFF:(O?16'hFF00:16'hF000);
    wire [15:0] a0 = (mul_b[0]?mul_a:16'h0000)&sel0;
    wire [15:0] a1 = (mul_b[1]?mul_a:16'h0000)&sel0;
    wire [15:0] a2 = (mul_b[2]?mul_a:16'h0000)&sel0;
    wire [15:0] a3 = (mul_b[3]?mul_a:16'h0000)&sel0;
    wire [15:0] a4 = (mul_b[4]?mul_a:16'h0000)&sel1;
    wire [15:0] a5 = (mul_b[5]?mul_a:16'h0000)&sel1;
    wire [15:0] a6 = (mul_b[6]?mul_a:16'h0000)&sel1;
    wire [15:0] a7 = (mul_b[7]?mul_a:16'h0000)&sel1;
    wire [15:0] a8 = (mul_b[8]?mul_a:16'h0000)&sel2;
    wire [15:0] a9 = (mul_b[9]?mul_a:16'h0000)&sel2;
    wire [15:0] a10 = (mul_b[10]?mul_a:16'h0000)&sel2;
    wire [15:0] a11 = (mul_b[11]?mul_a:16'h0000)&sel2;
    wire [15:0] a12 = (mul_b[12]?mul_a:16'h0000)&sel3;
    wire [15:0] a13 = (mul_b[13]?mul_a:16'h0000)&sel3;
    wire [15:0] a14 = (mul_b[14]?mul_a:16'h0000)&sel3;
    wire [15:0] a15 = (mul_b[15]?mul_a:16'h0000)&sel3;
    wire [15:0] tmp0,tmp1,tmp2,tmp3;
    wire [15:0] tmp00,tmp11;
    wire [15:0] tmp000;
    assign tmp0  = a0   + (a1<<1)   +  (a2<<2)    +  (a3<<3);
    assign tmp1  = a4   + (a5<<1)   +  (a6<<2)    +  (a7<<3);
    assign tmp2  = a8   + (a9<<1)   +  (a10<<2)   +  (a11<<3);
    assign tmp3  = a12  + (a13<<1)  +  (a14<<2)   +  (a15<<3);
    assign tmp00 = tmp0 + (tmp1<<4);
    assign tmp11 = tmp2 + (tmp3<<4);
    assign tmp000 = tmp00 + (tmp11<<8);
    wire [3:0] tmp1h,tmp1o,tmp1q;
    wire [3:0] tmp2h,tmp2o,tmp2q;
    wire [3:0] tmp3h,tmp3o,tmp3q;
    assign mulout[3:0]  = tmp0[3:0];
    assign tmp1h = tmp000[7:4];
    assign tmp2h = tmp000[11:8];
    assign tmp3h = tmp000[15:12];
    assign tmp1o = tmp00[7:4];
    assign tmp2o = tmp11[11:8];
    assign tmp3o = tmp11[15:12];
    assign tmp1q = tmp1[7:4];
    assign tmp2q = tmp2[11:8];
    assign tmp3q = tmp3[15:12];
    assign mulout[7:4]   = H?tmp1h:(O?tmp1o:tmp1q);
    assign mulout[11:8]  = H?tmp2h:(O?tmp2o:tmp2q);
    assign mulout[15:12] = H?tmp3h:(O?tmp3o:tmp3q);
endmodule

module SIMDshifter(
        input [15:0] shiftin,
        input H,
        input O,
        input Q,
        input left,
        output [15:0] shiftout);
    wire [14:0] left_shift =  shiftin[14:0];
    wire [14:0] right_shift =  shiftin[15:1];
    wire [15:0] shiftout_tmp = left?{left_shift,1'b0}:{1'b0,right_shift};
    assign shiftout[3:0]   = {(left|H|O)&shiftout_tmp[3],   shiftout_tmp[2:0]};
    assign shiftout[7:4]   = {(left|H)&shiftout_tmp[7],     shiftout_tmp[6:5],
        (!left|H|O)&shiftout_tmp[4]};
    assign shiftout[11:8]  = {(left|H|O)&shiftout_tmp[11],  shiftout_tmp[10:9],
        (!left|H)&shiftout_tmp[8]};
    assign shiftout[15:12] = {(left|H)&shiftout_tmp[15],    shiftout_tmp[14:13],
        (!left|H|O)&shiftout_tmp[12]};
endmodule
```

Chapter 12

Reference

1. https://github.com/Rohit04121998/Traffic-Controller

2. https://github.com/sudhamshu091/32-Verilog-Mini-Projects/tree/main/dual_address_rom

3. https://github.com/mongrelgem/Verilog-Adders/tree/master/Kogge-Stone%20Adder

4. https://github.com/TheSUPERCD/8bit_MicroComputer_Verilog

5. https://github.com/zslwyuan/Basic-SIMD-Processor-Verilog-Tutorial/blob/master/README.md

6. https://github.com/ARUNEMB3101/100_Days_Of_RTL_Target/blob/main/Day_32_Real_time_clock.v