



# Summer Fellowship Report

On

Mixed Signal/Digital Simulation in eSim

Submitted by

**Roshan Binu Paul**

B.Tech(Electronics and Communication)

Muthoot Institute Of Technology and Science Kochi

**Bhargav Dhoke**

B.Tech(Electronics and Telecommunication)

SGGSIE&T Nanded

**Abhinav Tripathi**

B.Tech(Electronics Engineering)

RGIPT Amethi

Under the guidance of

**Prof.Kannan M. Moudgalya**

Chemical Engineering Department

IIT Bombay

September 22, 2023

# Acknowledgment

We would like to express our gratefulness and sincere gratitude to the FOSSEE team for providing us such a wonderful opportunity to work for the welfare of humankind by contributing in the development of this open source software eSim.

First and foremost, we would like to express our gratefulness to Prof. Kannan M. Moudgalya for his valuable and constructive guidance throughout this FOSSEE fellowship program.

We would like to extend our heartfelt thanks to the whole eSim team for guiding us and providing us all the resources needed to complete this fellowship work. Special thanks to Mrs. Usha Viswanathan and Mrs. Vineeta Parmar for their extremely valuable guidance throughout this whole journey. Furthermore we would like to acknowledge and express our gratitude to our mentors Mr. Sumanto Kar and Mr. Rahul Paknikar for their invaluable support, their warm cooperation and willingness to share their knowledge and experiences with us made our journey truly enriching. Moreover, we would never forget our fellow friends for helping us to make this fellowship journey interesting and knowledgeable. It was a great experience working with them under the guidance of above mentioned people.

Lastly, we would like thank all the individual who directly or indirectly helped us in completing our fellowship. We would like to carry forward the lessons learned in this fellowship journey to make meaningful contribution in the betterment of this world.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	eSim . . . . .	5
1.2	NGHDL . . . . .	5
1.3	Icarus-Verilog . . . . .	5
1.4	Makerchip-NgVeri . . . . .	5
<b>2</b>	<b>Problem Statement</b>	<b>7</b>
2.1	Approach . . . . .	7
<b>3</b>	<b>8bit-MIPS Processor</b>	<b>8</b>
3.1	Processor Architecture . . . . .	8
3.2	Topmodule . . . . .	9
3.3	Simulation of Processor in eSim . . . . .	10
3.3.1	Outputs of Processor . . . . .	11
<b>4</b>	<b>32bit-RISC-V Processor</b>	<b>16</b>
4.1	Architecture . . . . .	16
4.2	Simulation in Icarus-Verilog . . . . .	17
4.2.1	Processor Testbench . . . . .	17
4.2.2	Output . . . . .	18
4.2.3	Register file Testbench . . . . .	19
4.2.4	Output . . . . .	21
4.3	Simulation in eSim . . . . .	22
4.3.1	Output . . . . .	23
<b>5</b>	<b>8bit-Microcomputer</b>	<b>24</b>
5.1	Simulation in Icarus-Verilog . . . . .	24
5.1.1	RAM file . . . . .	24
5.1.2	Output . . . . .	25
5.2	Simulation in eSim . . . . .	26
5.2.1	Output for opCode=5 . . . . .	27
5.2.2	Output for opCode=21 . . . . .	28
<b>6</b>	<b>8bit-RISC Processor</b>	<b>29</b>
6.1	Simulation in Icarus-Verilog . . . . .	29
6.1.1	Testbench . . . . .	29
6.1.2	Hex file . . . . .	31
6.1.3	Output . . . . .	31
6.2	Simulation in eSim . . . . .	32
6.2.1	Output . . . . .	33

<b>7</b>	<b>Digital Circuits Simulation</b>	<b>34</b>
7.1	16bit-Sklansky-Adder . . . . .	34
7.1.1	Simulation in Icarus-Verilog . . . . .	34
7.1.2	Simulation in eSim . . . . .	36
7.2	GCD Calculator . . . . .	37
7.2.1	Simulation in Icarus-Verilog . . . . .	37
7.2.2	Simulation in eSim . . . . .	39
7.3	Booth Multiplier . . . . .	40
7.3.1	Simulation in eSim . . . . .	40
7.4	Johnson Ring Counter . . . . .	41
7.4.1	Simulation in Icarus-Verilog . . . . .	41
7.4.2	Simulation in eSim . . . . .	43
7.5	Mux 8:1 . . . . .	44
7.5.1	Simulation in Icarus-Verilog . . . . .	44
7.5.2	Simulation in eSim . . . . .	46
7.6	Multiplication by Repeated Addition . . . . .	47
7.6.1	Simulation in Icarus-Verilog . . . . .	47
7.6.2	Simulation in eSim . . . . .	49
7.7	Demux 1:8 . . . . .	50
7.7.1	Simulation in Icarus-Verilog . . . . .	50
7.7.2	Simulation in eSim . . . . .	52
7.8	<b>2:1 Multiplexer</b> . . . . .	54
7.8.1	Simulation in ModelSIM . . . . .	54
7.8.2	Simulation in eSim . . . . .	56
7.9	<b>4-Bit ALU</b> . . . . .	58
7.9.1	Simulation in ModelSIM . . . . .	58
7.9.2	Simulation in eSim . . . . .	59
<b>8</b>	<b>RISC V -Processor</b>	<b>62</b>
8.1	Design of RISC V . . . . .	62
8.1.1	Arithmetic Logic Unit (ALU) . . . . .	62
8.1.2	Control Unit . . . . .	64
8.1.3	Designing Microarchitecture . . . . .	65
8.2	Simulation of RISC V . . . . .	66
8.2.1	Schematic . . . . .	66
8.2.2	Analysis . . . . .	66
8.2.3	Simulation . . . . .	67
<b>9</b>	<b>UART TX</b>	<b>68</b>
9.0.1	Schematic . . . . .	68
9.0.2	Simulation . . . . .	69
<b>10</b>	<b>Kogge Stone adder</b>	<b>70</b>
10.0.1	Schematic . . . . .	70
10.0.2	Simulation . . . . .	71
<b>11</b>	<b>4*4 Array Multiplier</b>	<b>72</b>
11.0.1	Schematic . . . . .	72
11.0.2	Simulation . . . . .	73

<b>12 Clock Divider</b>	<b>74</b>
12.0.1 Schematic . . . . .	74
12.0.2 Simulation . . . . .	74
<b>13 16 Bit Processor</b>	<b>76</b>
13.0.1 Schematic . . . . .	76
13.0.2 Simulation . . . . .	77
<b>14 CD4585 4-Bit Magnitude Comparator</b>	<b>78</b>
14.0.1 Pin Diagram . . . . .	78
14.0.2 Schematic . . . . .	78
14.0.3 Simulation . . . . .	79
<b>15 4-Bit ALU</b>	<b>81</b>
15.0.1 Schematic . . . . .	81
15.0.2 Simulation . . . . .	81
<b>16 IN74HCT21 Dual 4 Input AND Gate</b>	<b>86</b>
16.0.1 Pin Diagram . . . . .	86
16.0.2 Schematic . . . . .	86
16.0.3 Simulation . . . . .	87
<b>17 Circuits Contribution</b>	<b>88</b>
17.0.1 Roshan Binu Paul . . . . .	88
17.0.2 Bhargav Dhoke . . . . .	88
17.0.3 Abhinav Tripathi . . . . .	88
<b>Bibliography</b>	<b>89</b>

# Chapter 1

## Introduction

### 1.1 eSim

FOSSEE (Free/Libre and Open Source Software for Education) is a project part of the National Mission on Education through Information and Communication Technology (ICT), Ministry of Human Resource Development (MHRD), Government of India. FOSSEE has developed various open source tools and promotes the use of these tools in improving the quality of education and helping every individual avail these sources free of cost. The softwares is being developed in such a way that it can stay relevant with respect to the commercial softwares.

### 1.2 NGHDL

NGHDL is a mixed mode circuit simulator developed by FOSSEE, using NgSpice and GHDL. The NGHDL feature makes it easier to create user-defined models for eSims simulation of mixed-signal circuits. In NGHDL, the analogue and digital components communicate through sockets and NgSpice is used to simulate the analogue components and GHDL to simulate the digital components. This feature was added to eSim so that a user who is familiar with designing circuits in Verilog can do so with eSim. In order to write Verilog code for a digital model and install it as a model in Ngspice, NGHDL oers an interface.

### 1.3 Icarus-Verilog

Icarus Verilog is an implementation of the Verilog hardware description language compiler that generates netlists in the desired format (EDIF). It supports the 1995, 2001 and 2005 versions of the standard, portions of SystemVerilog, and some extensions. Icarus Verilog is available for Linux, FreeBSD, OpenSolaris, AIX, Microsoft Windows, and Mac OS X. Released under the GNU General Public License, Icarus Verilog is free software. As of release 0.9, Icarus is composed of a Verilog compiler (including a Verilog preprocessor) with support for plug-in backends, and a virtual machine that simulates the design.

### 1.4 Makerchip-NgVeri

Makerchip is a browser-based IDE (Integrated Development Environment) that allows users to simulate Verilog, System Verilog, and TL-Verilog files. It is developed

using Verilator, which converts Verilog files into C++ objects. Before using NgVeri in eSim, the design can be simulated in Makerchip with random inputs to ensure that it produces the desired and consistent results. Once the design is successfully simulated, it can be used in mixed-signal designs. These models can be used in digital/mixed signal simulations.

# Chapter 2

## Problem Statement

Implementing open source microcontrollers/processors using NGHDL present in eSim so that the user can simulate these processors in eSim by changing the instruction sets which can be changed from file or by providing instruction through the input pins in KiCad.

### 2.1 Approach

The general approach which I used to implement the problem statement is first searching from an open source processor on GitHub or Opencores which have any open source licensing such as MIT license. Then the implementation process is as follows:

- The verilog code, each file was tested in Icarus-Verilog and then fed to Ngveri to see if it converts correctly without throwing any error.
- Then that individual file is simulated and the Ngspice waveform is generated to check the desired waveform.
- After all the components are simulated the final cputop file is simulated in makerchip and a module is created to link the input instruction with that of the processor top file. Then the simulations and conversion takes place by adding the other files as dependencies in ngveri.
- The final object is simulated in Ngspice to check the final result of the processor instruction set.



# Chapter 3

## 8bit-MIPS Processor

It is the 8 bit MIPS processor based on the HARVARD architecture. The blocks that make up the processor are described mainly structurally, sometimes being described behaviorally or dataflow. For each major block, the architecture will be presented using schemes, the meaning of each port will also be described.

### 3.1 Processor Architecture

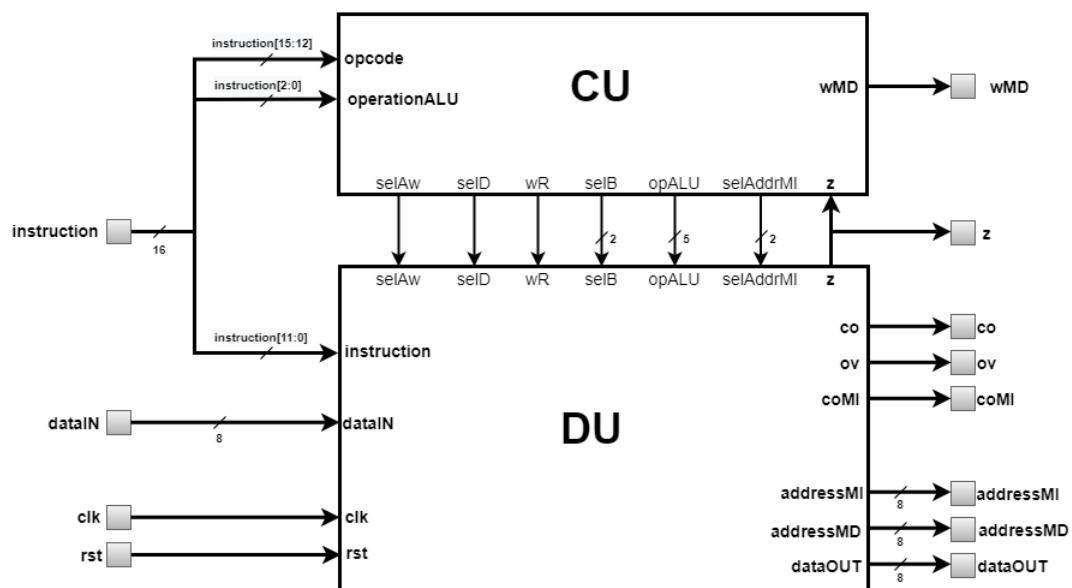


Figure 3.1: Architecture

## 3.2 Topmodule

Topmodule consists of various input and output pins.

- **addressMI[7:0]** = output pin that provides the address of the next instruction.
- **instrucion[15:0]** = input pin that receives the machine code of the current instruction.
- **addressMD[7:0]** = output pin that provides the required address of Data Memory Block to perform read/write operation.
- **dataIN[7:0]** = input pin that receives the data read from Data Memory Block as a result of using LW instruction.
- **dataOUT[7:0]** = output pin that provides the data to be written to Data Memory Block as a result of using SW instruction.
- **wMD** = output pin that provides the signal that controls the write operation in Data Memory Block.
- **co** = output pin that provides a signal that indicates an overflow in the representation of the result of an operation between unsigned integers.
- **ov** = output pin that provides a signal that indicates an overflow in the representation of the result of an operation between signed integers.
- **coMI** = output pin that provides a signal that indicates an overflow of the addressing capacity of Instruction Memory Block.
- **z** = output pin that provides a signal indicating a result equal to 0 at the output of Arithmetic-Logic Unit (ALU).
- **testCtrlSg[13:0]** = output pin that provides the control word of the current instruction to monitor the signal sent by the cu to DU.
- **clk** = input pin that receives the clock signal, the active front is the positive one.
- **rst** = input pin that receives the asynchronous initialization signal, which is active high.

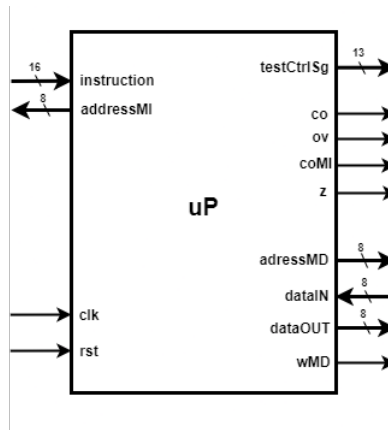


Figure 3.2: Topmodule

### 3.3 Simulation of Processor in eSim

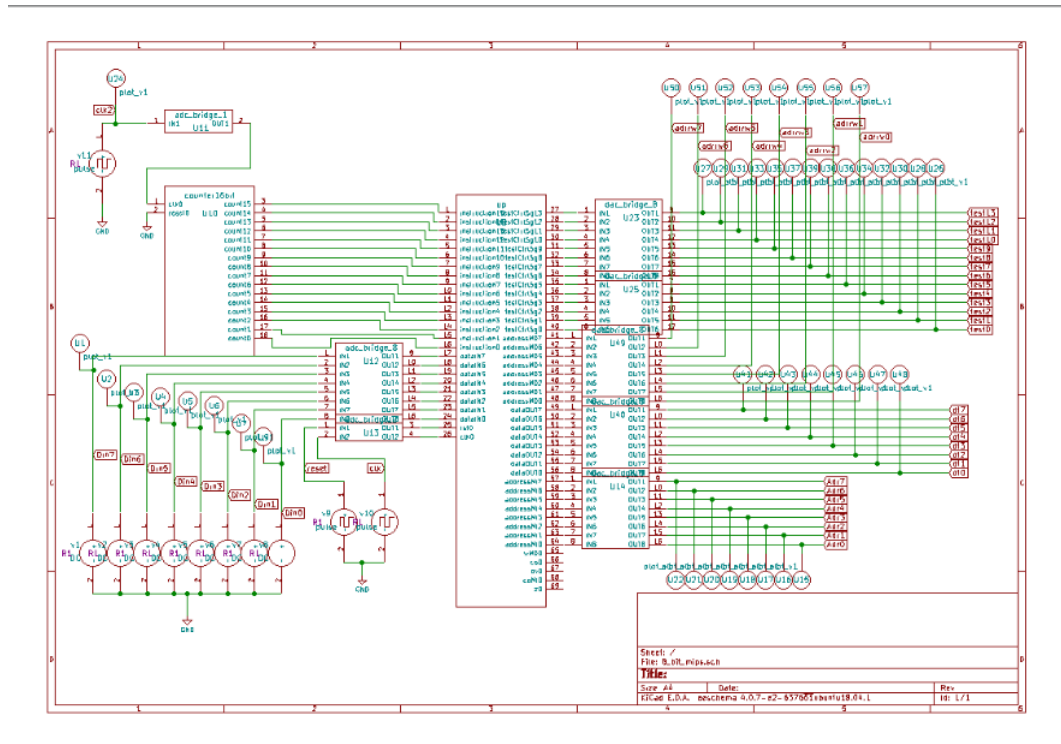


Figure 3.3: 8bit-MIPS

### 3.3.1 Outputs of Processor

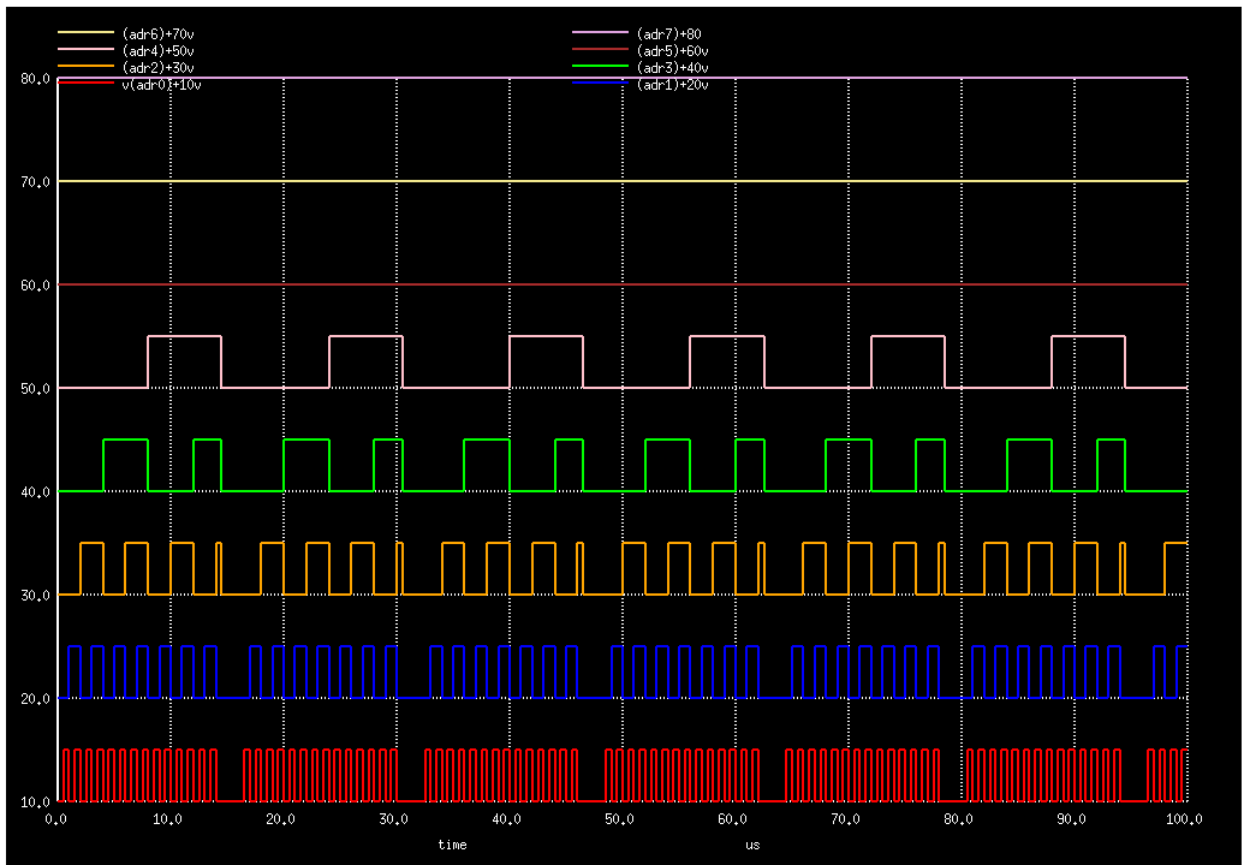


Figure 3.4: Address of next instruction

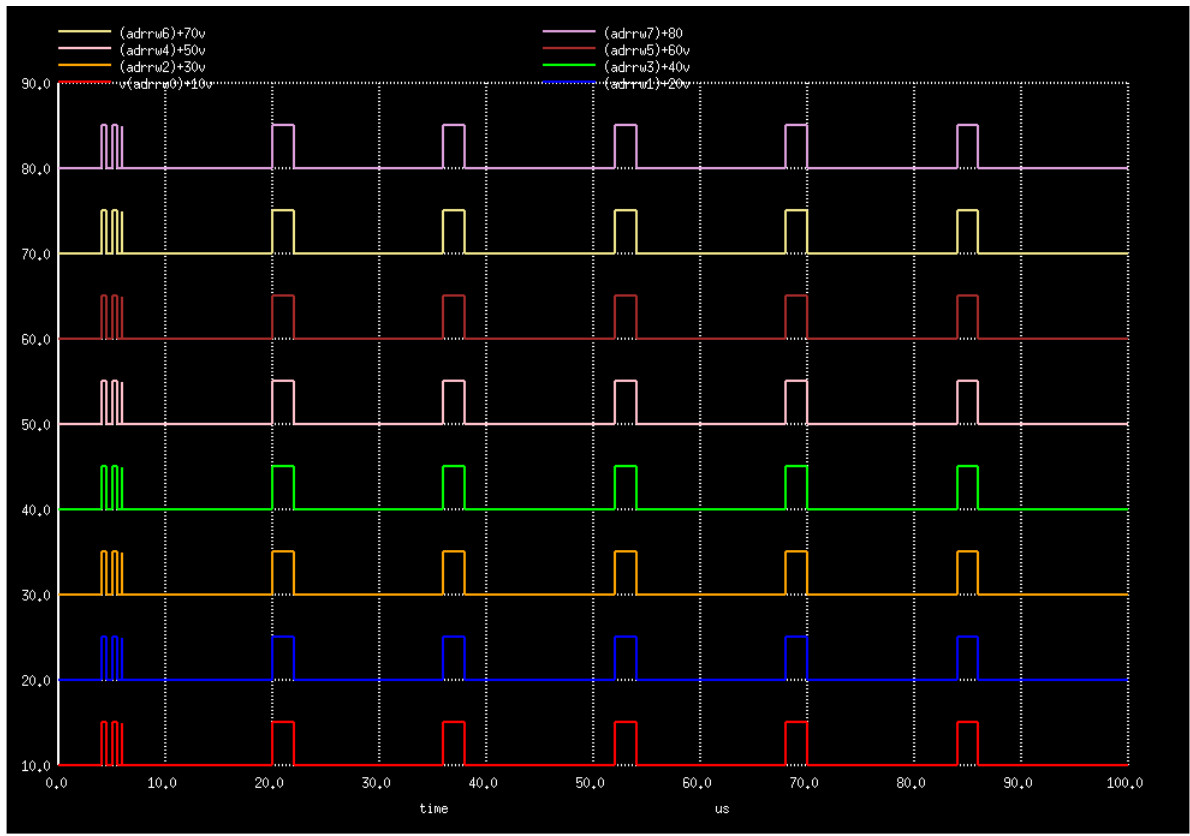


Figure 3.5: Required address of Data Memory Block to perform read/write operation

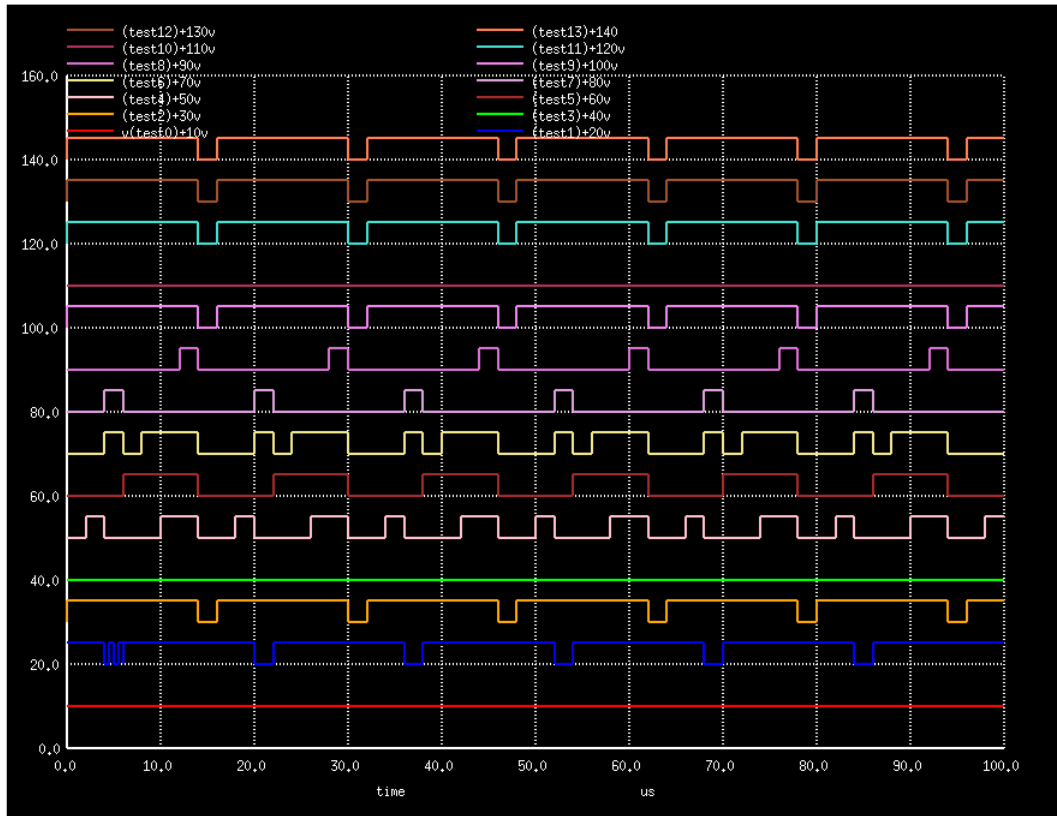


Figure 3.6: Signal sent by control unit to data unit to monitor current instruction

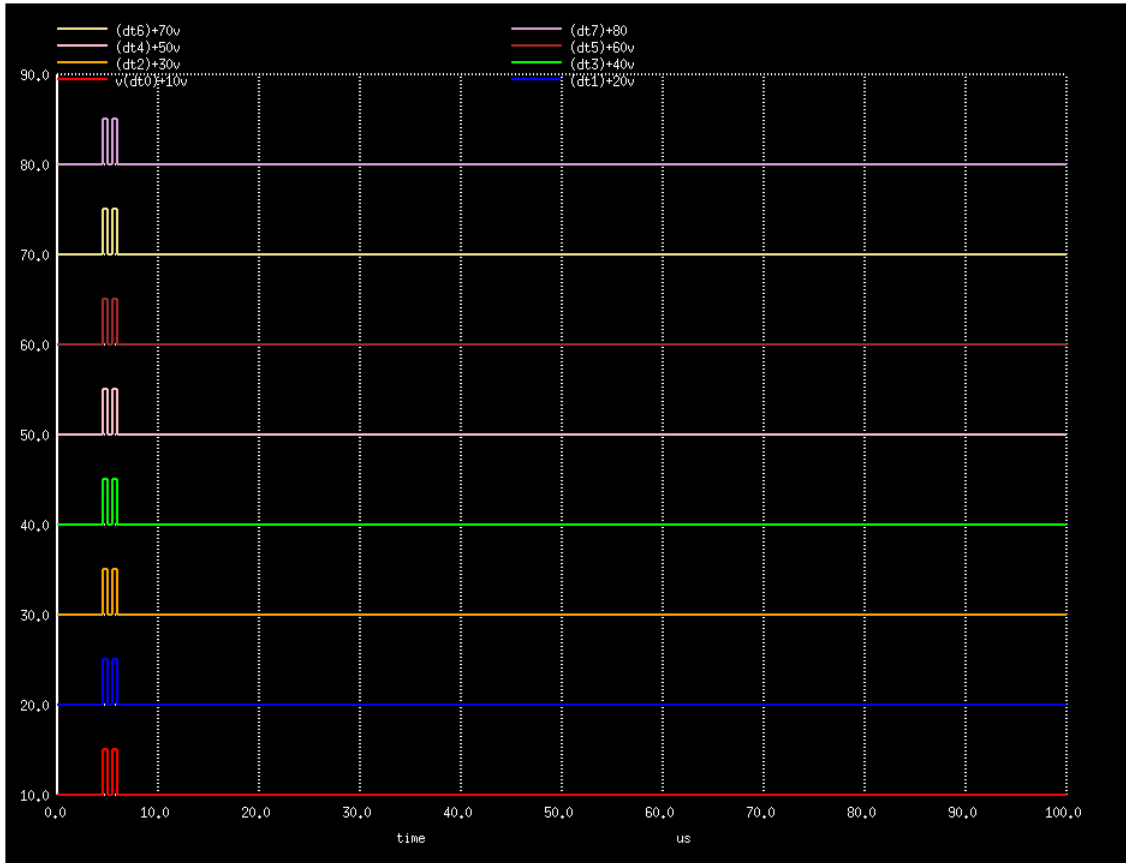


Figure 3.7: Data to be written in memory block

```
8_bit_mips.cir.out
ov=0
coMI=0
z=1
=====up : New Iteration=====
Instance : 0

Inside foo before eval.....
instruction=50
dataIN=16
rst=0
clk=0
testCtrlSg=14870
addressMI=0
dataOUT=0
addressMI=6
wMI=0
co=0
ov=0
coMI=0
z=1

Inside foo after eval.....
instruction=50
dataIN=16
rst=0
clk=1
testCtrlSg=14870
addressMI=0
dataOUT=0
addressMI=7
wMI=0
co=0
ov=0
coMI=0
z=1
=====up : New Iteration=====
Instance : 0

Inside foo before eval.....
```

Figure 3.8: Ngspice terminal



# Chapter 4

## 32bit-RISC-V Processor

RISC-V is a new instruction set architecture (ISA) that was originally designed to support computer architecture research and education. A completely open ISA that is freely available to academia and industry. It is the open source processor so that anyone can access it.

### 4.1 Architecture

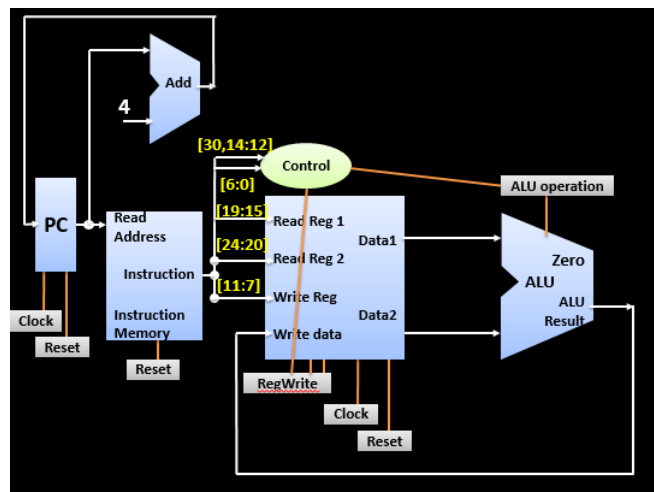


Figure 4.1: Architecture

## 4.2 Simulation in Icarus-Verilog

### 4.2.1 Processor Testbench

---

```
'include "PROCESSOR.v"

module stimulus ();

    reg clock;
    reg reset;
    wire zero;

    // Instantiating the processor!!!
    PROCESSOR test_processor(clock,reset,zero);

    initial begin
        $dumpfile("output_wave.vcd");
        $dumpvars(0,stimulus);
    end

    initial begin
        reset = 1;
        #50 reset = 0;
    end

    initial begin
        clock = 0;
        forever #20 clock = ~clock;
    end

    initial
        #300 $finish;

endmodule
```

---

## 4.2.2 Output

The processor is performing read and write operation. Processor have two 32bit register to read data and two 5bit register to write data. It takes two 32bit(Decimal) value from reg memory then write it to 5bit readregnum. As you can see in below output that the values 8 and 9 is being read by two 32bit readdata register and next it is writing this data in readregnum register which is 5bit(Hex). This whole process does not required any signal. This operation is independent of clock and reset signal.

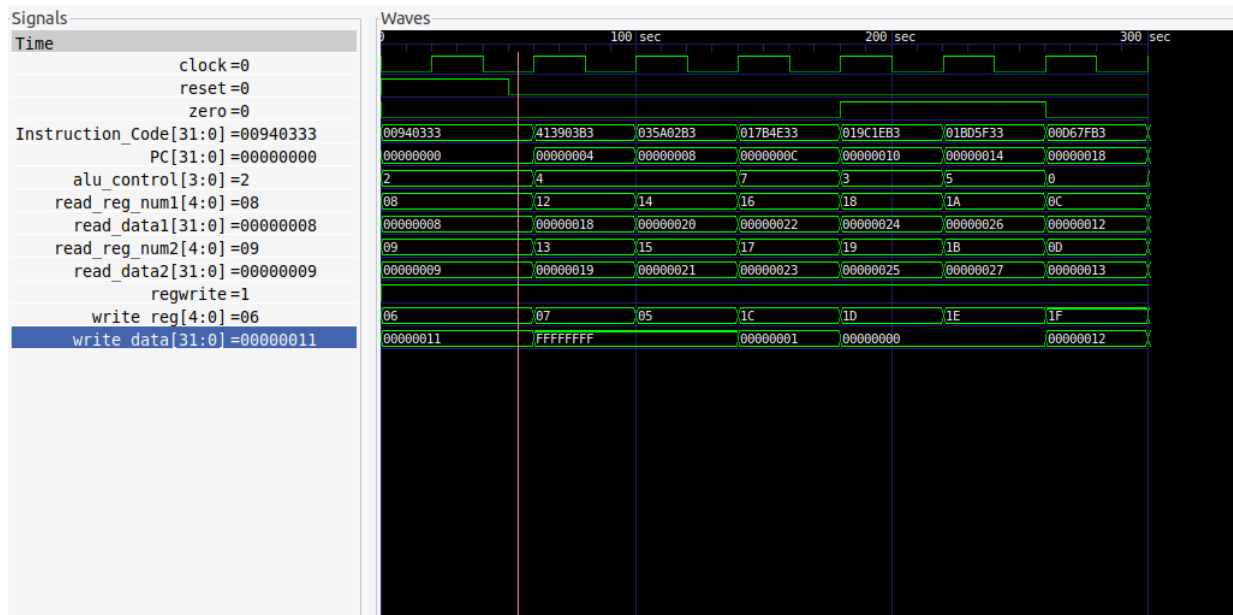


Figure 4.2: GTKwave

## 4.2.3 Register file Testbench

---

```
'include "REG_FILE.v"

module stimulus ();
    reg [4:0] read_reg_num1;
    reg [4:0] read_reg_num2;
    reg [4:0] write_reg;
    reg [31:0] write_data;
    wire [31:0] read_data1;
    wire [31:0] read_data2;
    reg regwrite;
    reg clock;
    reg reset;

    // Instantiating register file module
    REG_FILE REG_FILE_module(
        read_reg_num1,
        read_reg_num2,
        write_reg,
        write_data,
        read_data1,
        read_data2,
        regwrite,
        clock,
        reset);

    // Setting up output waveform
    initial begin
        $dumpfile("output_wave.vcd");
        $dumpvars(0, stimulus);
    end

    // Initializing the registers
    initial begin
        reset = 1;
        #10 reset = 0;
    end

    // Writing values to different registers
    initial begin
        regwrite = 0;
        #10
        regwrite = 1; write_data = 20; write_reg = 0;
        #10
        regwrite = 1; write_data = 30; write_reg = 1;
        #10
        regwrite = 1; write_data = 30; write_reg = 1;
        #10;
    end
end
```

```

end

// Reading values of different registers
initial begin

    #10 read_reg_num1 = 0; read_reg_num2 = 0;
    #10 read_reg_num1 = 0; read_reg_num2 = 1;
    #10 read_reg_num1 = 0; read_reg_num2 = 1;
    #10 read_reg_num1 = 1; read_reg_num2 = 2;

end

// Setting up clock
initial begin

    clock = 0;
    forever #10 clock = ~clock;

end

initial begin
    #200 $finish;
end

endmodule

```

---

A register file can read two registers and write in to one register. The 32bit-RISC-V register file contains total of 32 registers each of size 32-bit. Hence 5-bits are used to specify the register numbers that are to be read or written. Register Write: Register writes are controlled by a control signal RegWrite. Additionally the register file has a clock signal. The write should happen if RegWrite signal is made 1 and if there is positive edge of clock. As we can see in the below output that clk is 1 and regwrite is also 1. So it will write data in write reg[4:0]=01 from readregnum2[4:0]=01.

# 4.2.4 Output

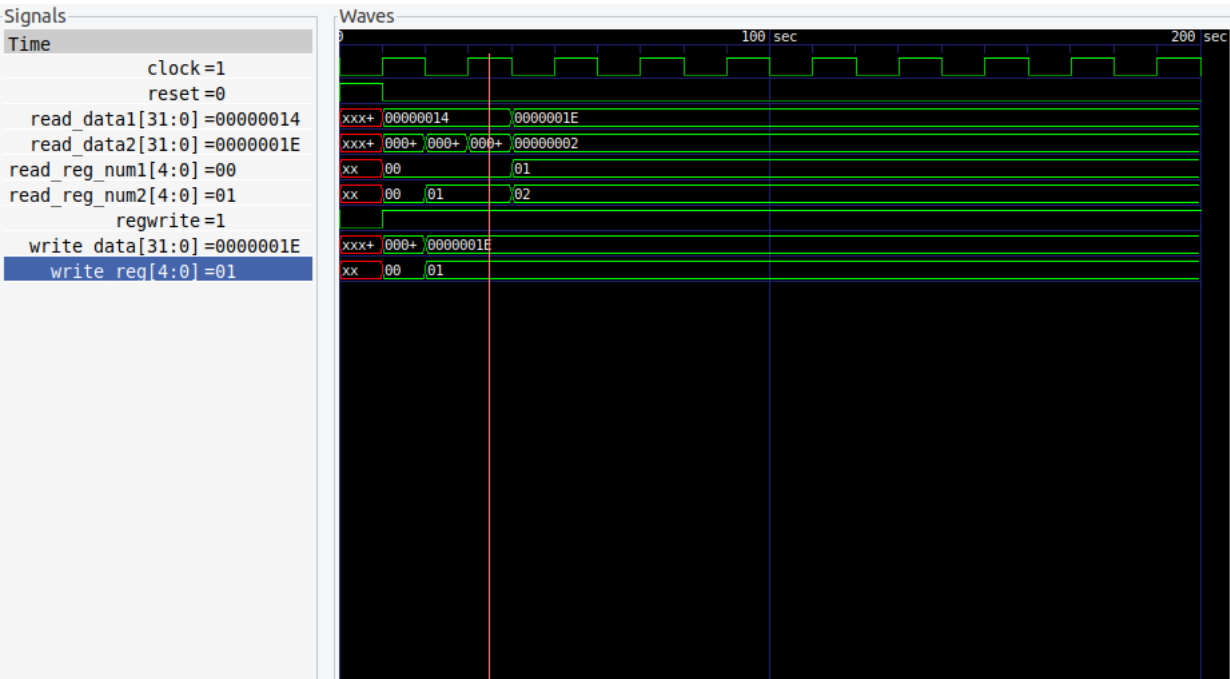


Figure 4.3: Register file Output

### 4.3 Simulation in eSim

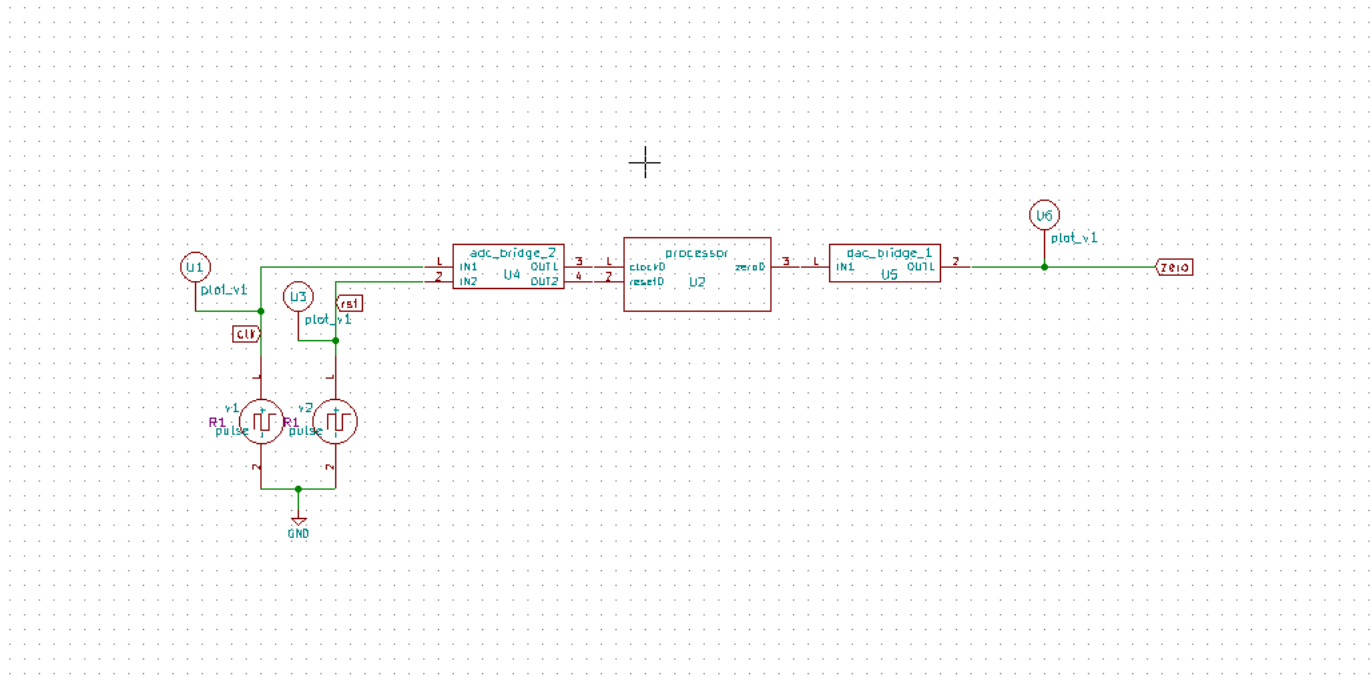


Figure 4.4: 32bit-RISC-V

### 4.3.1 Output

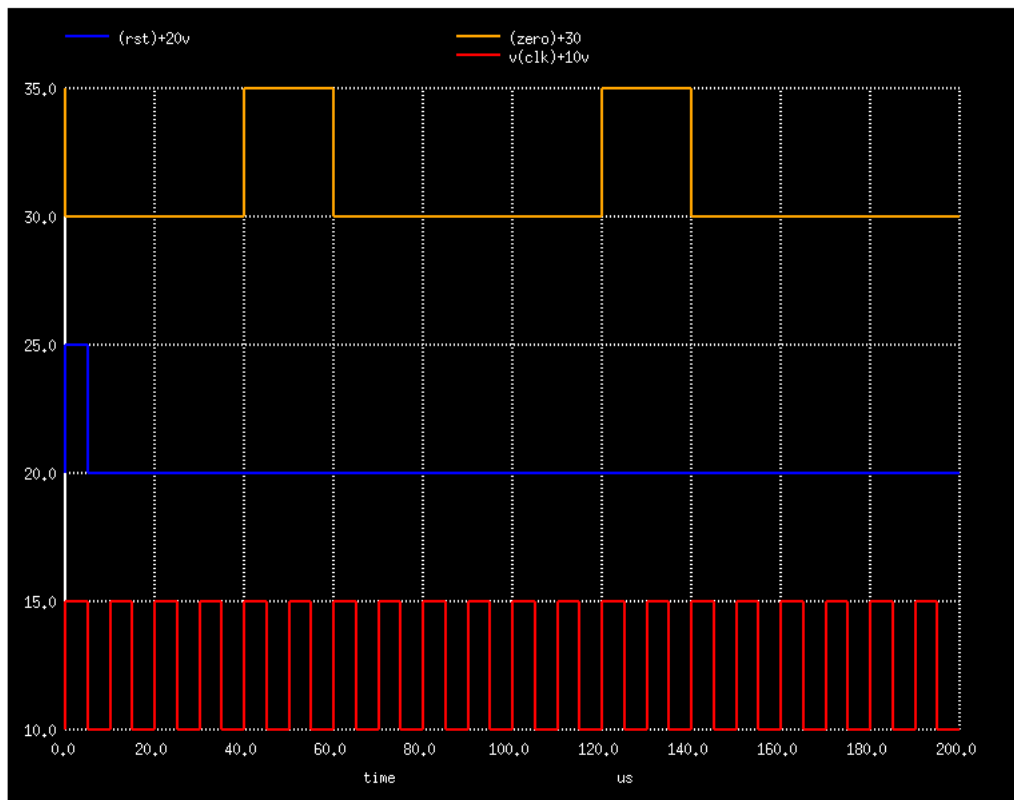


Figure 4.5: Zero signal indicating output



# Chapter 5

## 8bit-Microcomputer

This computer is based on the Von Neumann architecture. The von Neumann architecture is a computer design where program instructions and data are stored in the same memory. It consists of a CPU that executes instructions, a memory that holds instructions and data, and I/O devices for input and output. Instructions are fetched, decoded, and executed sequentially. The architecture has a bottleneck in data transfer between the CPU and memory. Despite limitations, it is widely used and forms the basis of most modern computers.

### 5.1 Simulation in Icarus-Verilog

#### 5.1.1 RAM file

---

```
'timescale 1ns/1ps
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
end
```

```

always @(posedge clk)
begin
    if(write_enable & ~read_enable)
    begin
        Memory[address] <= data;
    end
    else
    begin
        buffer <= Memory[address];
    end
end
end

assign data = (read_enable & ~write_enable) ? buffer : 8'bzzzzzzzz;

endmodule

```

---

## 5.1.2 Output

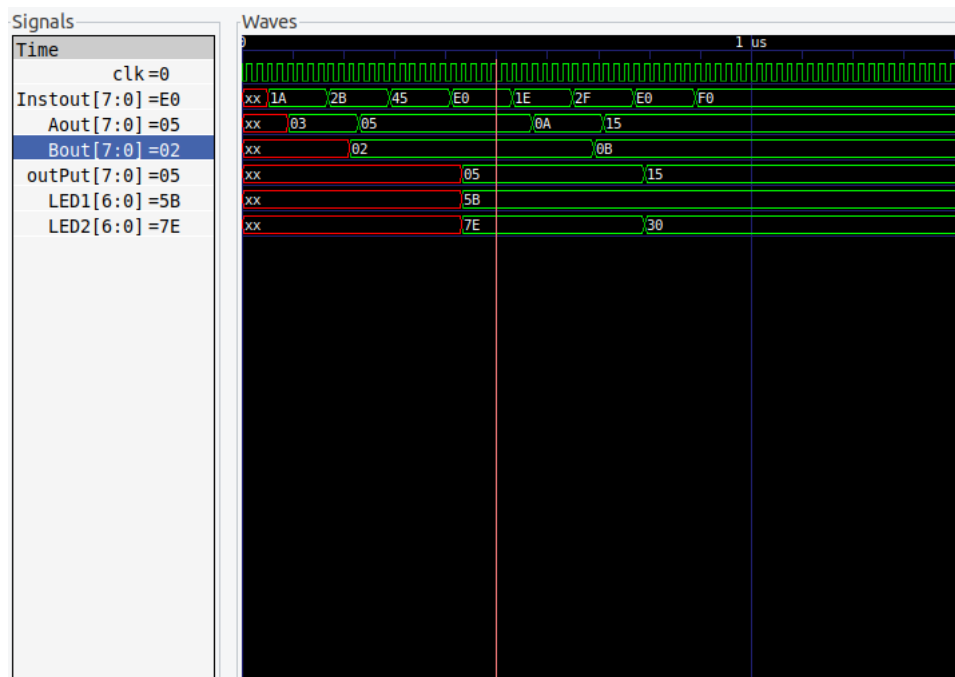


Figure 5.1: For opCode=5

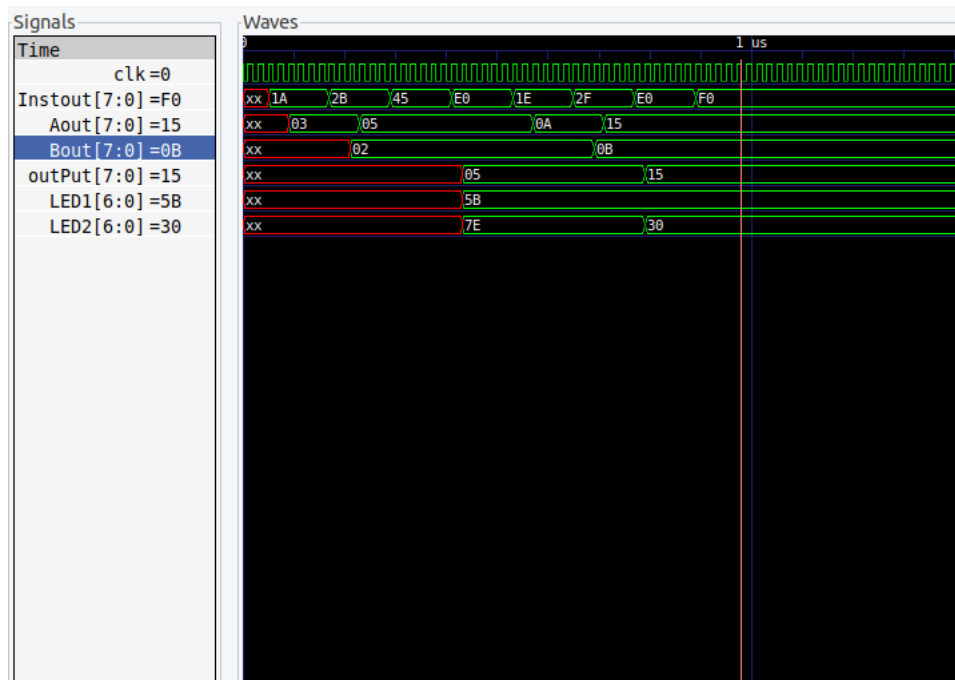


Figure 5.2: For opCode=21

## 5.2 Simulation in eSim

Top module is created using Makerchip and Ngveri. In top module at input there is 8bit opCode and one clock signal. At output we have two LED 7bit each, we will verify the output on LED. opCode is fed by 8bit counter in order to get different value for opCode. We are primarily looking for opCode 5 and 21. So we can verify them with output obtain in Icarus-Verilog. The outputs we obtained on eSim are in decimal value and in Icarus-Verilog it is hex value which is same.

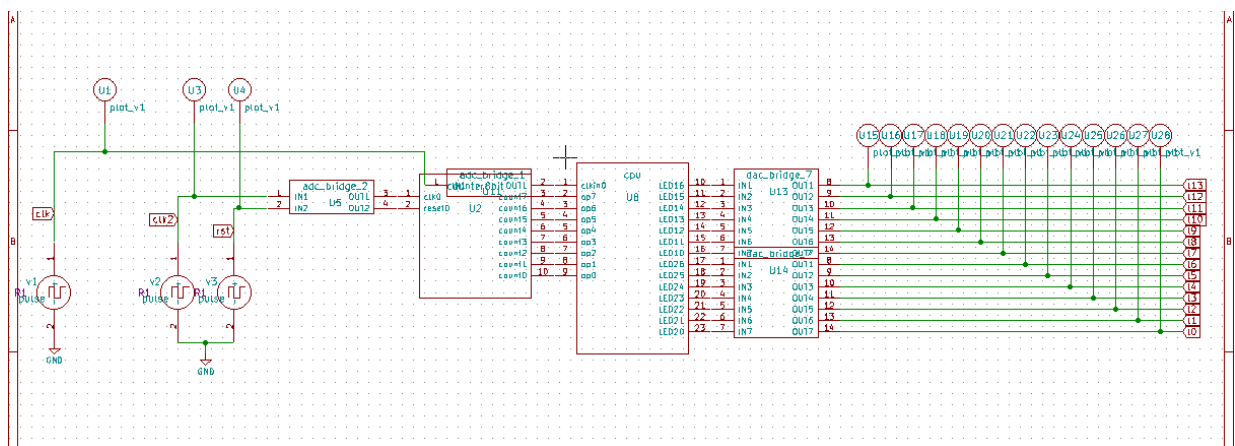


Figure 5.3: 8bit-Microcomputer model

### 5.2.1 Output for opCode=5

```
8bitcomputer.cir.out
Instance : 0
Inside foo before eval.....
clk=1
reset=0
count=5

Inside foo after eval.....
clk=0
reset=0
count=5
=====cpu : New Iteration=====
Instance : 0

Inside foo before eval.....
clk=1
op=5
LED1=91
LED2=126

Inside foo after eval.....
clk=1
op=5
LED1=91
LED2=126
=====cpu : New Iteration=====
Instance : 0
```

Figure 5.4: NgSpice

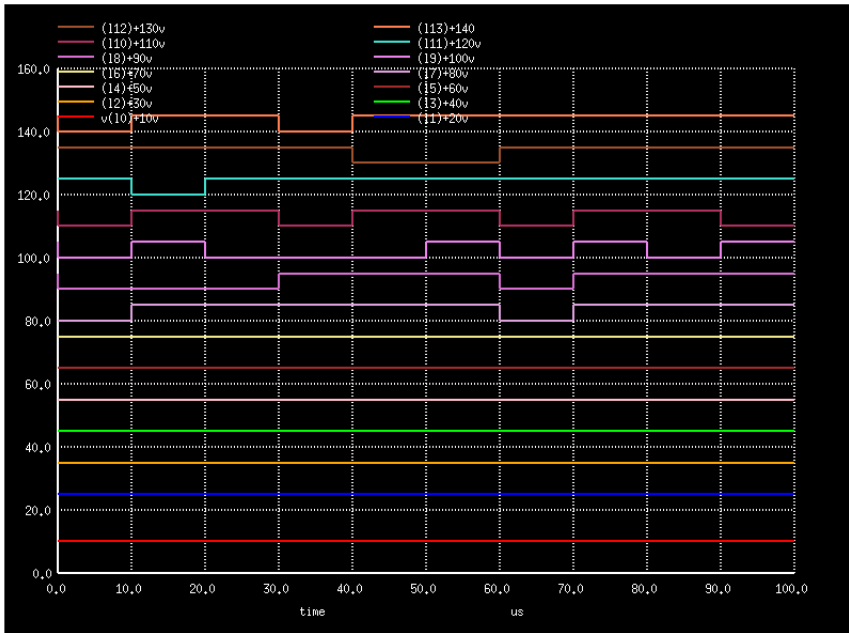


Figure 5.5: Waveform

## 5.2.2 Output for opCode=21

```
8bitcomputer.cir.out

Inside foo after eval.....
clk=1
reset=0
count=21
=====cpu : New Iteration=====
Instance : 0

Inside foo before eval.....
clk=1
op=21
LED1=91
LED2=48

Inside foo after eval.....
clk=0
op=21
LED1=91
LED2=48
=====counter8bit : New Iteration=====
Instance : 0

Inside foo before eval.....
clk=1
reset=0
count=21
```

Figure 5.6: NgSpice

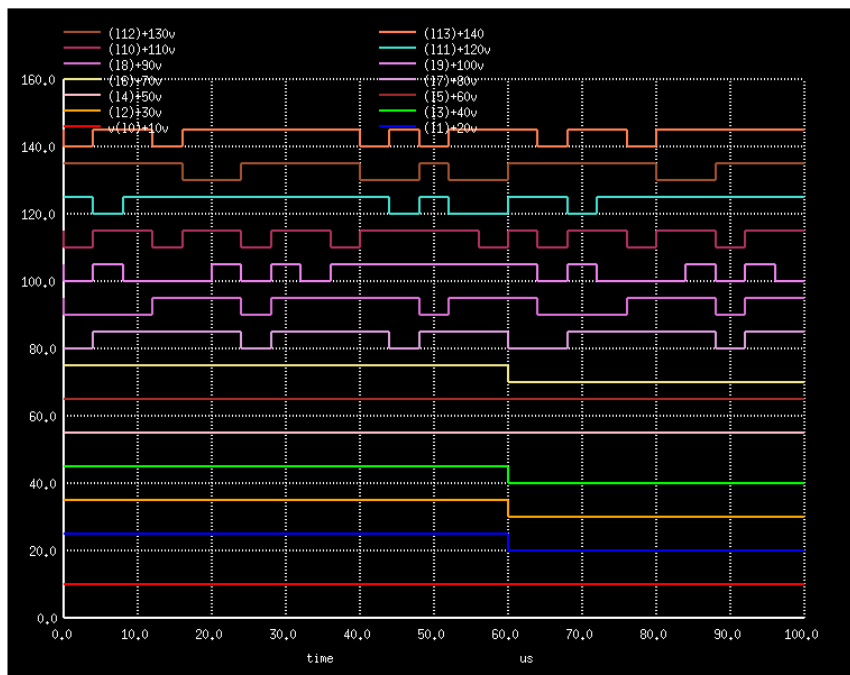


Figure 5.7: Waveform

# Chapter 6

## 8bit-RISC Processor

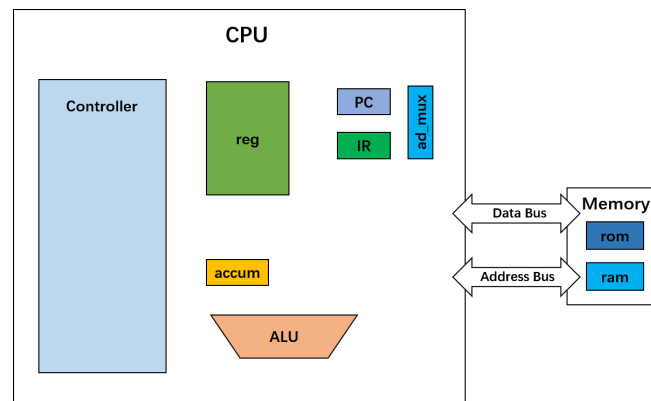


Figure 6.1: Block Diagram

### 6.1 Simulation in Icarus-Verilog

#### 6.1.1 Testbench

```
'timescale 1ps / 1ps
'include "core.v"
module core_tb_00 ;

    reg rst, clk;
    reg we;
    reg [4:0] address;
    reg [7:0] di;
    wire [7:0] dout;
    wire write_r, read_r, PC_en, ac_ena, ram_ena, rom_ena, ram_write, ram_read,
        ↪ rom_read, ad_sel;
    wire [1:0] fetch;
    wire [7:0] data;
    wire [7:0] addr;
    wire [7:0] accum_out;
    wire [7:0] alu_out;
    wire [7:0] ir_ad;
    wire [7:0] pc_ad;
```

```

wire [4:0] reg_ad;
wire [2:0] ins;

core DUT(clk, rst, write_r, read_r, PC_en, ac_ena, ram_ena, rom_ena,
    ↪ ram_write, ram_read, rom_read, ad_sel, fetch, data, addr, accum_out,
    ↪ alu_out, ir_ad, pc_ad, reg_ad, ins, address, di, dout, we);
initial
begin
^^I clk = 1'b0 ;
^^I # 150 ;
// 50 ps, single loop till start period.
    repeat(99)
        begin
^^I clk = 1'b1 ;
^^I #50 clk = 1'b0 ;
^^I #50 ;
// 9950 ps, repeat pattern in loop.
            end
^^I clk = 1'b1 ;
^^I # 50 ;
// dumped values till 10 ns
        end

// "Constant Pattern"
// Start Time = 0 ps, End Time = 10 ns, Period = 0 ps
    initial
        begin
^^I rst = 1'b0 ;
^^I # 100;
^^I rst=1'b1;
^^I # 9000 ;
// dumped values till 10 ns
            end
        initial begin
            we=1;
            address=0;
            di=0;

            #100 we=0;
            #100 address=5'd1;
            #110 address=5'd1;
            #120 address=5'd2;
            #130 address=5'd3;
            #140 address=5'd4;
            #150 address=5'd25;
        end

    initial
        begin

```

```

    $dumpfile("8bit.vcd");
    $dumpvars(0, core_tb_00);
    #2000;
    $finish;
end

always #5 clk=~clk;
endmodule

```

---

## 6.1.2 Hex file

---

```

@000
A
@001
B C D E F
10 11 12 13 14
@0b
15 16 17 18 19
1A 1B 1C 1D 1E
@015
1F 20 21 22 23

```

---

## 6.1.3 Output

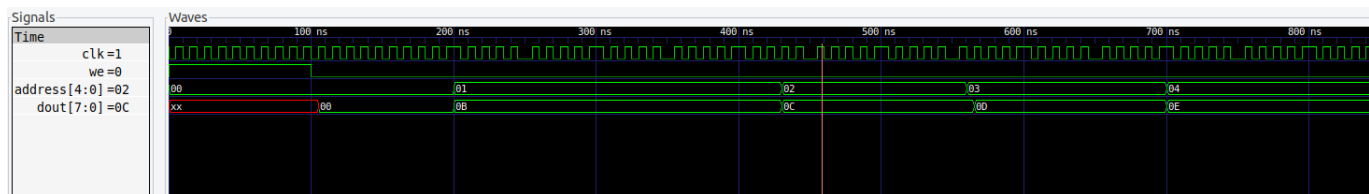


Figure 6.2: Waveform



## 6.2 Simulation in eSim

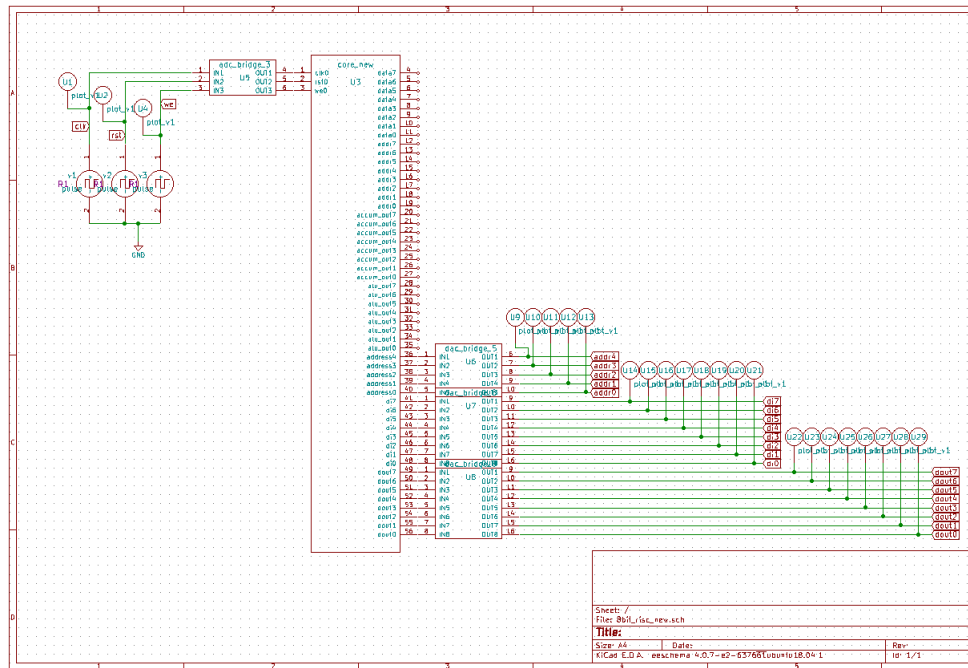
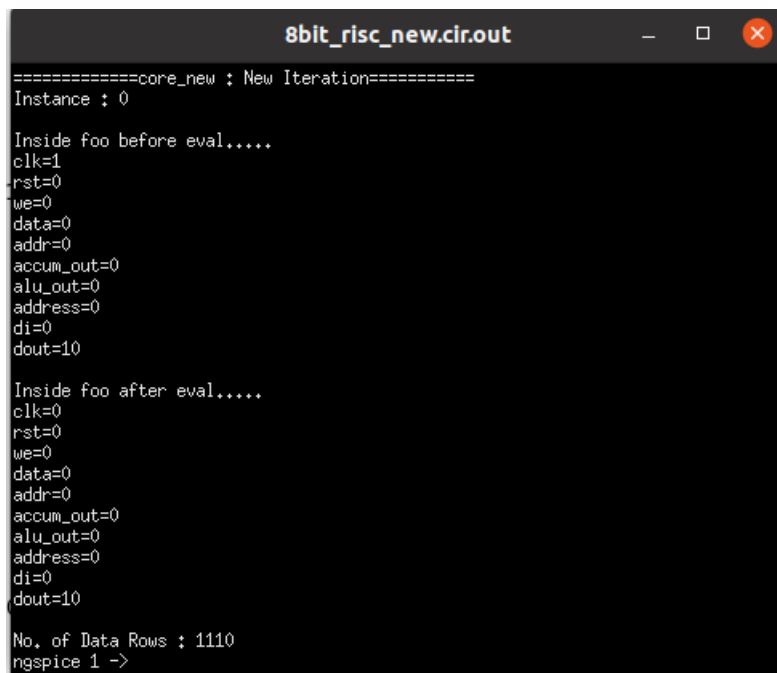


Figure 6.3: 8bit-RISC Processor

### 6.2.1 Output



```
8bit_risc_new.cir.out
=====core_new : New Iteration=====
Instance : 0

Inside foo before eval.....
clk=1
rst=0
we=0
data=0
addr=0
accum_out=0
alu_out=0
address=0
di=0
dout=10

Inside foo after eval.....
clk=0
rst=0
we=0
data=0
addr=0
accum_out=0
alu_out=0
address=0
di=0
dout=10

No. of Data Rows : 1110
ngspice 1 ->
```

Figure 6.4: Reading Number 10

# Chapter 7

# Digital Circuits Simulation

## 7.1 16bit-Sklansky-Adder

### 7.1.1 Simulation in Icarus-Verilog

## Testbench

```
'include "Sklansky.v"

module top;

wire [15:0] Sum;
wire cout;
reg [15:0] A;
reg [15:0] B;
reg cin;

sklansky adder(cin,A,B,Sum,cout);

initial
begin
    cin = 1'b0;
    #0 A=50; B=50;
    #4 A=47; B=47;
    #8 A=46; B=46;
    #12 A=49; B=49;
end

initial
begin
    $monitor ($time, " Input: A = %d   B = %d   \n\t\t\t\t\t Output: A+B = %d\n",A,B,Sum);
    $dumpfile("Sklansky.vcd");
    $dumpvars;
end

endmodule
```

```

bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/16bit-Sklansky-Adder$ iverilog Sklansky_tb.v
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/16bit-Sklansky-Adder$ vvp a.out
VCD info: dumpfile Sklansky.vcd opened for output.
      0 Input: A =   50   B =   50
        Output: A+B =  100
      4 Input: A =   47   B =   47
        Output: A+B =   94
     12 Input: A =   46   B =   46
        Output: A+B =   92
     24 Input: A =   49   B =   49
        Output: A+B =   98
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/16bit-Sklansky-Adder$

```

Figure 7.1: Icarus terminal

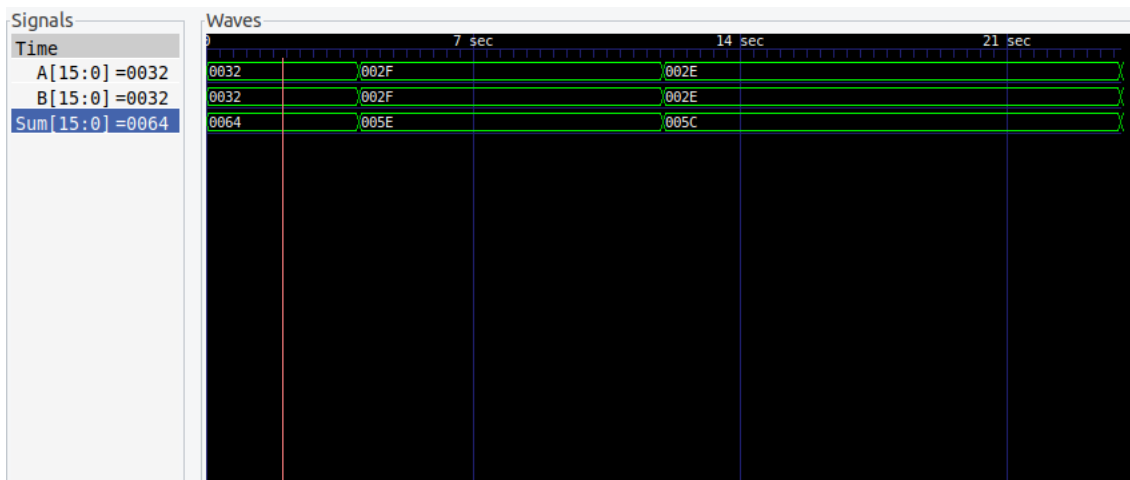


Figure 7.2: GTKwave

## 7.1.2 Simulation in eSim

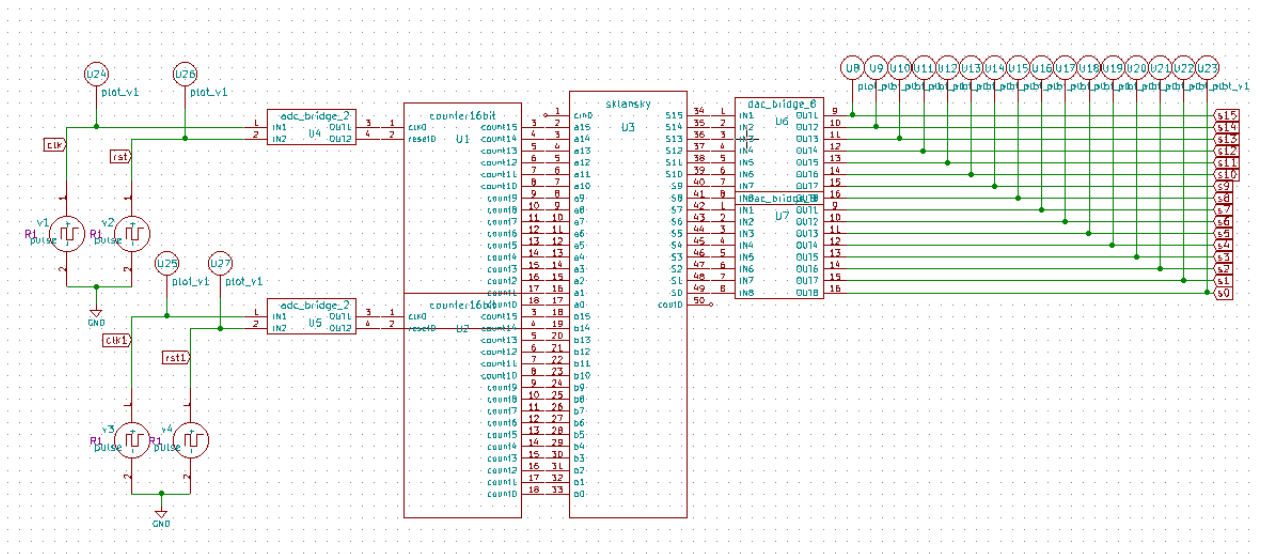


Figure 7.3: Schematic of Sklansky Adder

```

16bit_sklansky.cir.out
Inside foo after eval.....
clk=1
reset=0
count=48
=====sklansky : New Iteration=====
Instance : 0

Inside foo before eval.....
cin=0
a=47
b=47
S=94
cout=0

Inside foo after eval.....
cin=0
a=48
b=48
S=96
cout=0
=====counter16bit : New Iteration=====
Instance : 0

Inside foo before eval.....
clk=1
reset=0
count=48

Inside foo after eval.....
clk=1

```

Figure 7.4: NgSpice

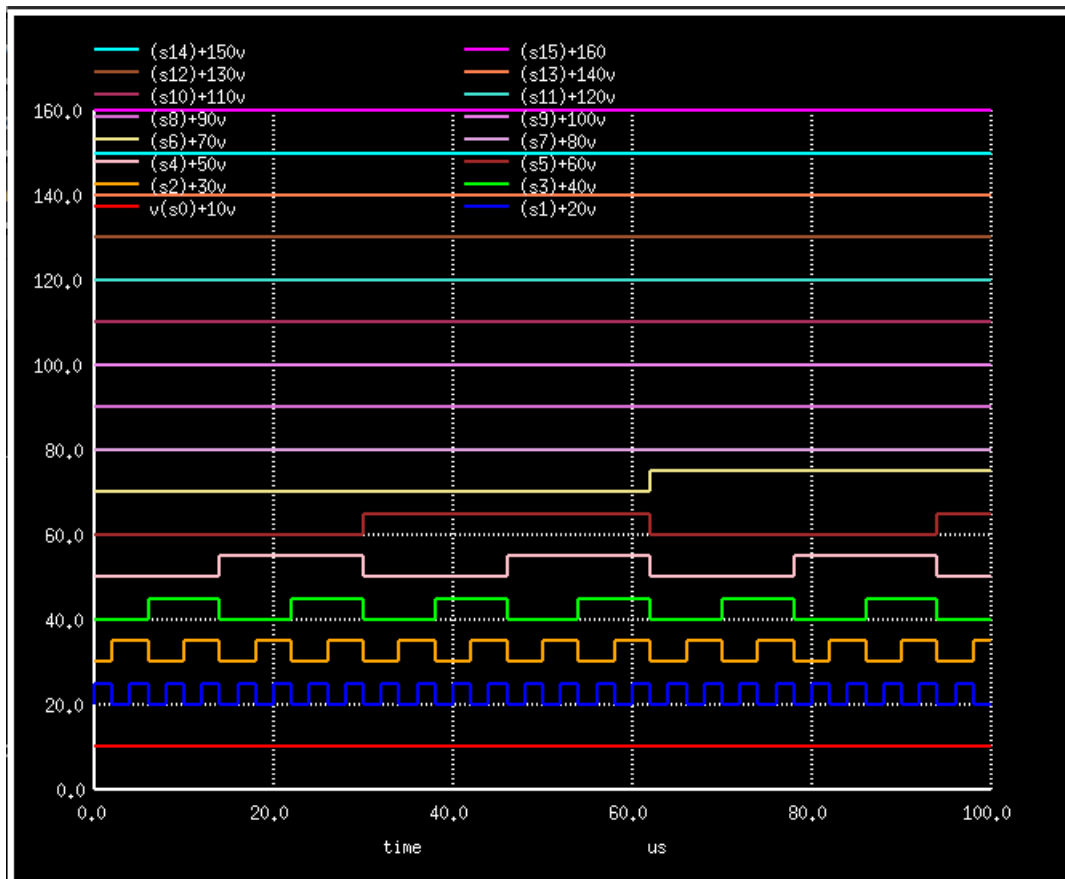


Figure 7.5: Waveform

## 7.2 GCD Calculator

### 7.2.1 Simulation in Icarus-Verilog

#### Testbench

```

include "gcd_datapath.v"
include "controller.v"

module test;
  reg [15:0] data_in;
  reg clk, start;
  wire done;

  reg [15:0] A, B;

  gcd_datapath DP (gt, lt, eq, ldA, ldB, sel1, sel2, sel_in, data_in, clk);
  controller CON (ldA, ldB, sel1, sel2, sel_in, done, clk, lt, gt, eq, start);

  initial
  begin
    clk = 1'b0;
    #3 start = 1'b1;
    #1000 $finish;
  end
endmodule

```

```

    end
always #5 clk = ~clk;

initial
    begin
        #12 data_in = 182;
        #10 data_in = 195;
    end

initial
    begin
        $monitor ($time, " %d %b", DP.Aout, done);
        $dumpfile ("gcd.vcd"); $dumpvars (0, test);
    end
endmodule

```

---

```

bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/gcd_computaion$ iverilog gcd_tb.v
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/gcd_computaion$ vvp a.out
VCD info: dumpfile gcd.vcd opened for output.
      0      x x
      5      x 0
     15    182 0
     45    169 0
     55    156 0
     65    143 0
     75    130 0
     85    117 0
     95    104 0
    105     91 0
    115     78 0
    125     65 0
    135     52 0
    145     39 0
    155     26 0
    165     13 0
    167     13 1
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/gcd_computaion$

```

Figure 7.6: Icarus terminal

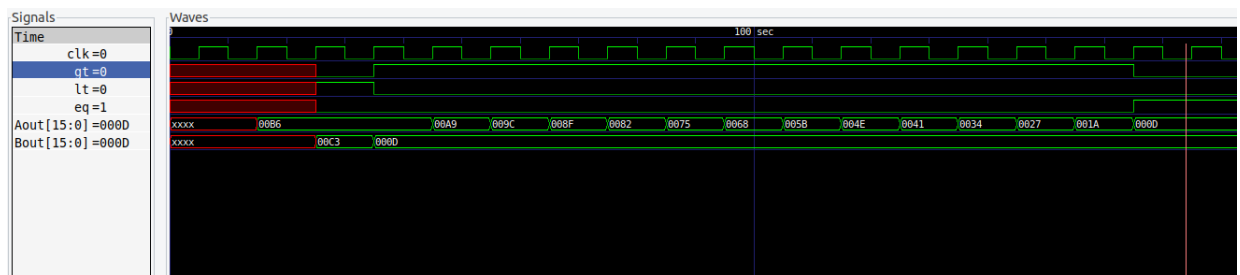


Figure 7.7: GTKwave

## 7.2.2 Simulation in eSim

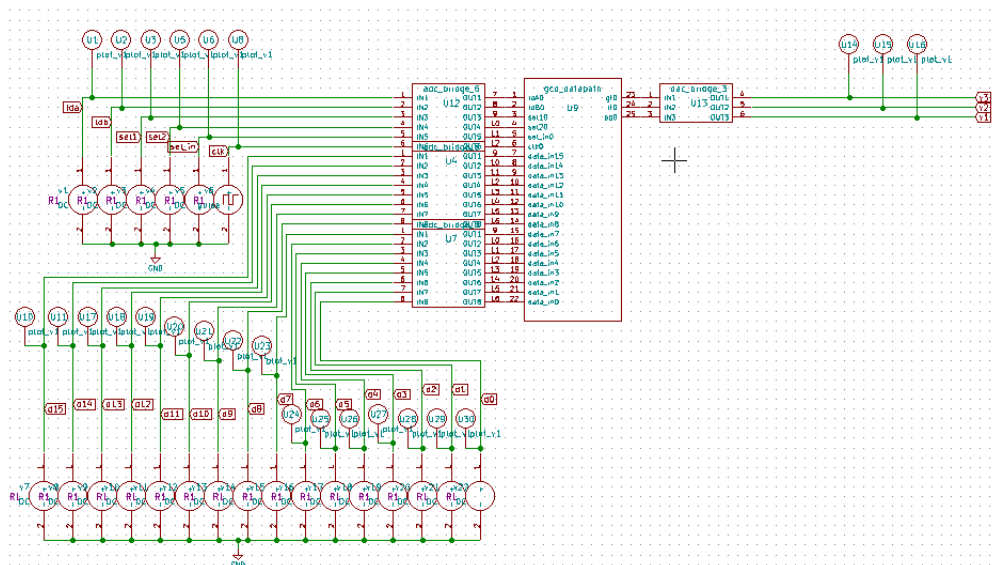


Figure 7.8: Schematic of GCD Calculator

```

gcd_calculator.cir.out
Inside foo before eval.....
ldA=0
ldB=1
sel1=1
sel2=0
sel_in=0
clk=1
data_in=15
gt=0
lt=0
eq=1

Inside foo after eval.....
ldA=0
ldB=1
sel1=1
sel2=0
sel_in=0
clk=0
data_in=15
gt=0
lt=0
eq=1

No. of Data Rows : 1110
ngspice 1 ->

```

Figure 7.9: NgSpice



## 7.3 Booth Multiplier

### 7.3.1 Simulation in eSim

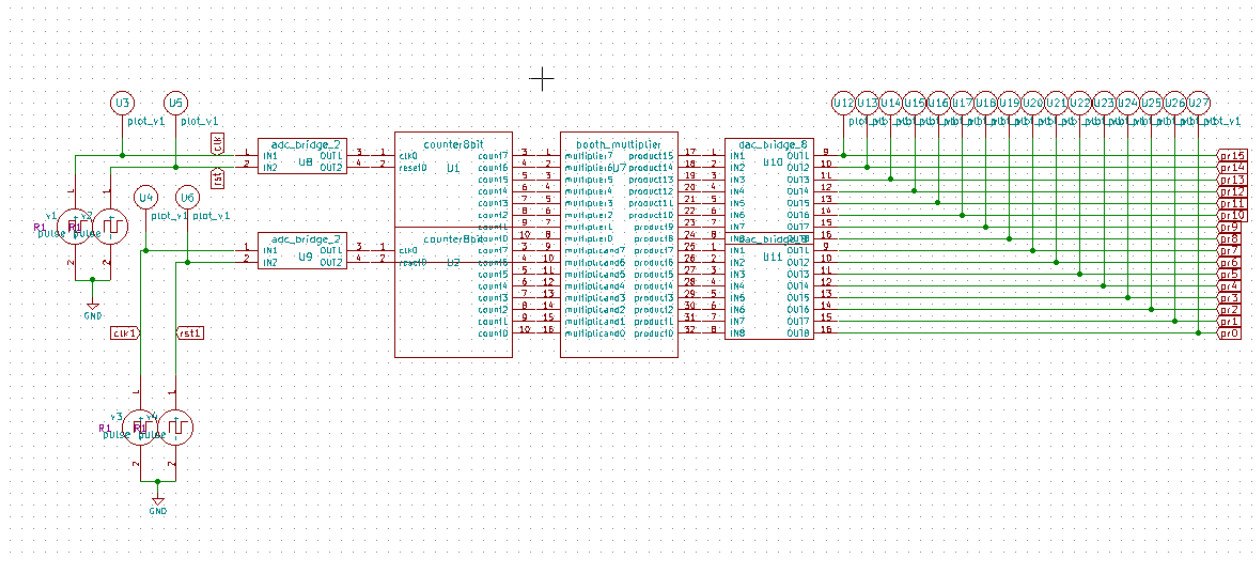


Figure 7.10: Schematic of Booth Multiplier

```
boot_multiplier.cir.out

Inside foo after eval.....
clk=1
reset=0
count=12
=====booth_multiplier : New Iteration=====
Instance : 0

Inside foo before eval.....
multiplier=11
multiplicand=9
product=99

Inside foo after eval.....
multiplier=12
multiplicand=9
product=108
=====counter8bit : New Iteration=====
Instance : 1

Inside foo before eval.....
clk=0
reset=0
count=9

Inside foo after eval.....
clk=1
reset=0
count=10
=====counter8bit : New Iteration=====
Instance : 1
```

Figure 7.11: NgSpice

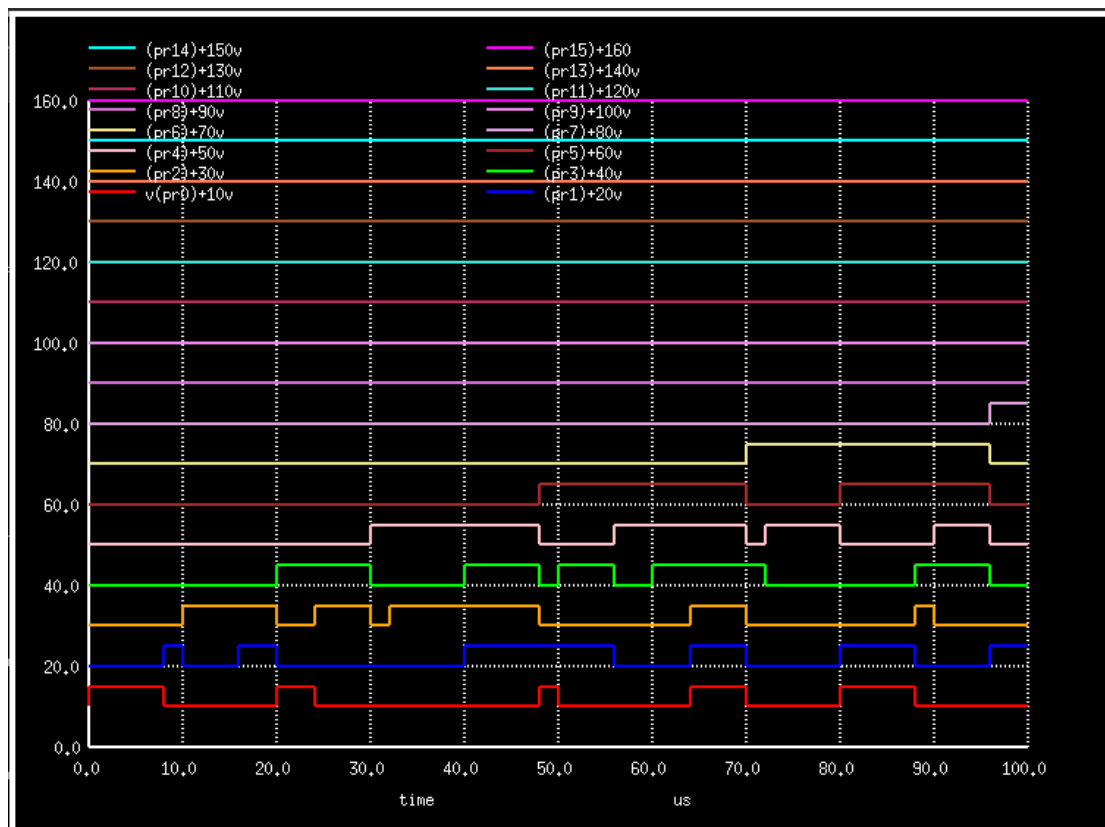


Figure 7.12: Waveform

## 7.4 Johnson Ring Counter

### 7.4.1 Simulation in Icarus-Verilog

#### Testbench

```

include "johnson.v"

module tb();

    reg clk,rst;
    wire[3:0]count;

    johnson DUT(clk, rst, count);

    always #5 clk=!clk;
    initial
    begin
        clk=0;
        rst=1;
        @(negedge clk);
        rst=0;
    end

    initial

```

```

begin
    $dumpfile("dump.vcd");
    $dumpvars();

    repeat(20)
        @(negedge clk);
    $finish();
end
endmodule

```

---

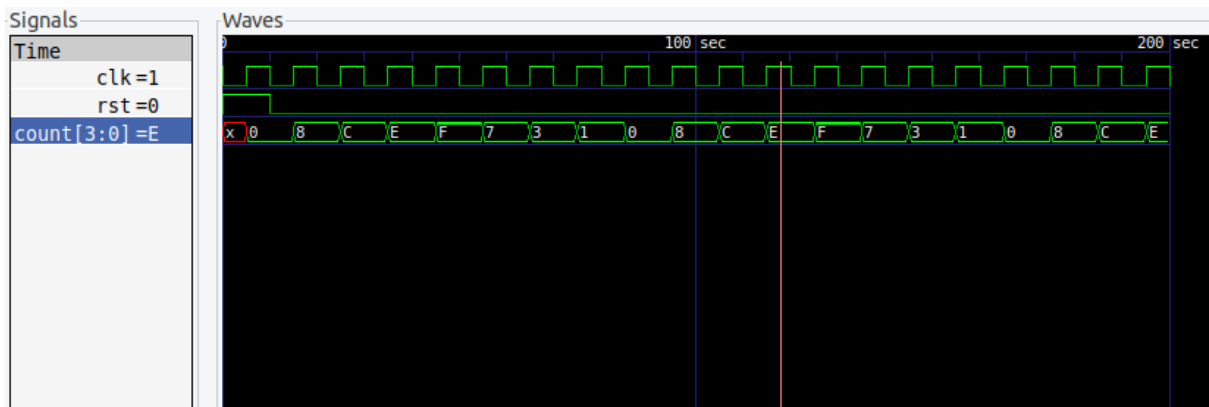


Figure 7.13: GTKwave

## 7.4.2 Simulation in eSim

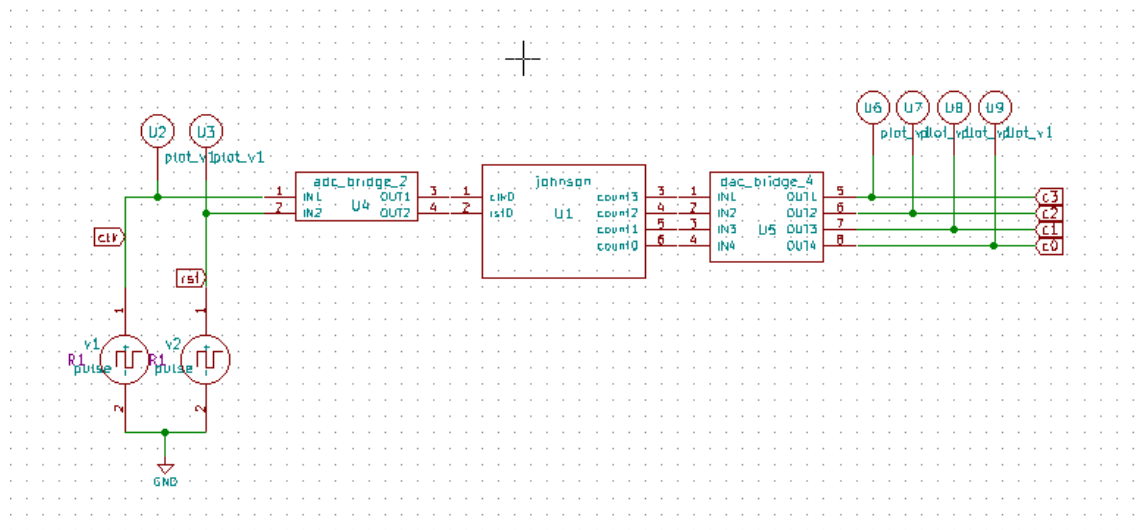


Figure 7.14: Schematic of Johnson Ring Counter

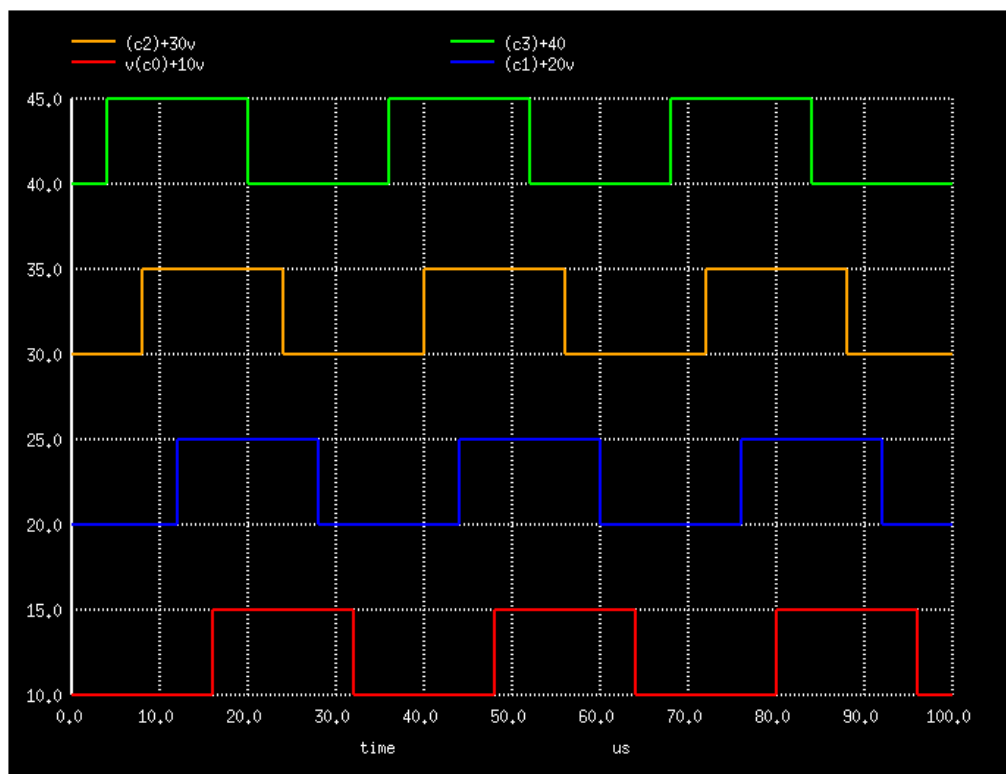


Figure 7.15: Waveform

## 7.5 Mux 8:1

### 7.5.1 Simulation in Icarus-Verilog

#### Testbench

---

```
'timescale 1s/100ms
'include "MUX8to1.v"

module MUX8to1_tb();
  ^^Ireg s0;
  ^^Ireg s1;
  ^^Ireg s2;
  ^^Ireg a;
  ^^Ireg b;
  ^^Ireg c;
  ^^Ireg d;
  ^^Ireg e;
  ^^Ireg f;
  ^^Ireg g;
  ^^Ireg h;
  ^^Iwire y;

  ^^IMUX8to1 myMUX(s0, s1, s2, a, b, c, d, e, f, g, h, y);

  ^^Iinitial
  ^^Ibegin
    ^^Ia = 1;
    ^^Ib = 0;
    ^^Ic = 1;
    ^^Id = 0;
    ^^Ie = 1;
    ^^If = 0;
    ^^Ig = 1;
    ^^Ih = 0;
    ^^I$monitor("s0 = %b, s1 = %b, s2 = %b, y = %b", s0, s1, s2, y);
    ^^I$dumpfile ("MUX8to1.vcd");
    ^^I$dumpvars (0, MUX8to1_tb);
    ^^Is0 = 0; s1 = 0; s2 = 0; #1;
    ^^Is0 = 0; s1 = 0; s2 = 1; #1;
    ^^Is0 = 0; s1 = 1; s2 = 0; #1;
    ^^Is0 = 0; s1 = 1; s2 = 1; #1;
    ^^Is0 = 1; s1 = 0; s2 = 0; #1;
    ^^Is0 = 1; s1 = 0; s2 = 1; #1;
    ^^Is0 = 1; s1 = 1; s2 = 0; #1;
    ^^Is0 = 1; s1 = 1; s2 = 1; #1;
    ^^I$finish;
  ^^Iend
endmodule
```

---

```
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/Mux$ iverilog MUX8to1_tb.v
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/Mux$ vvp a.out
VCD info: dumpfile MUX8to1.vcd opened for output.
s0 = 0, s1 = 0, s2 = 0, y = 1
s0 = 0, s1 = 0, s2 = 1, y = 0
s0 = 0, s1 = 1, s2 = 0, y = 1
s0 = 0, s1 = 1, s2 = 1, y = 0
s0 = 1, s1 = 0, s2 = 0, y = 1
s0 = 1, s1 = 0, s2 = 1, y = 0
s0 = 1, s1 = 1, s2 = 0, y = 1
s0 = 1, s1 = 1, s2 = 1, y = 0
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/Mux$
```

Figure 7.16: Icarus terminal

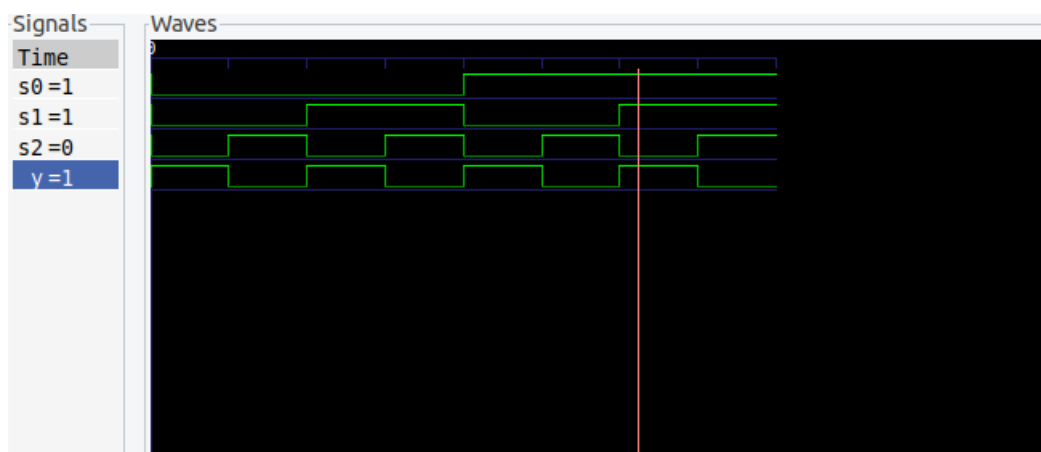


Figure 7.17: GTKwave

## 7.5.2 Simulation in eSim

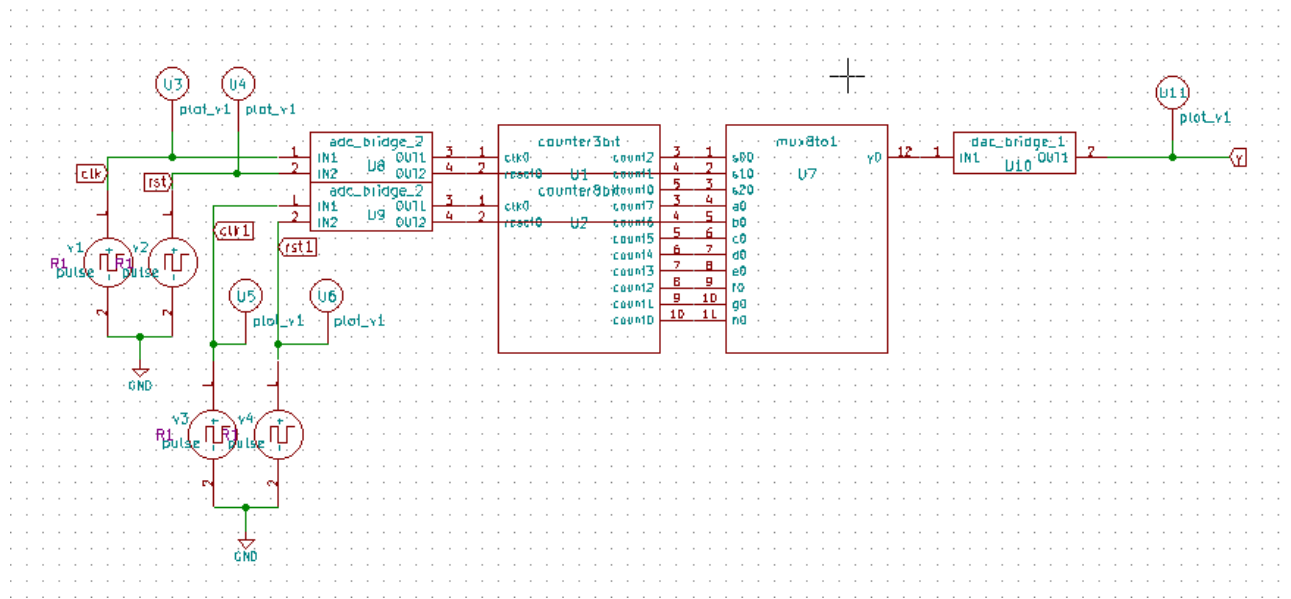


Figure 7.18: Schematic of MUX8to1

```

mux_8to1.cir.out
-----
Inside foo after eval.....
clk=1
reset=0
count=23
=====mux8to1 : New Iteration=====
Instance : 0

Inside foo before eval.....
s0=1
s1=1
s2=0
a=0
b=0
c=0
d=1
e=0
f=1
g=1
h=0
y=1

Inside foo after eval.....
s0=1
s1=1
s2=1
a=0
b=0
c=0

```

Figure 7.19: NgSpice

## 7.6 Multiplication by Repeated Addition

### 7.6.1 Simulation in Icarus-Verilog

#### Testbench

---

```
'include "MUL_datapath.v"
'include "controller.v"

module MUL_test;
    reg [15:0] data_in;
    reg clk, start;
    wire done;
    reg [15:0] A, P, B;

    MUL_datapath DP (eqz, LdA, LdB, LdP, clrP, decB, data_in, clk);
    controller CON (LdA, LdB, LdP, clrP, decB, done, clk, eqz, start);

    initial
        begin
            clk = 1'b0;
            #3 start = 1'b1;
            #500 $finish;
        end

    always #5 clk = ~clk;

    initial
        begin
            #17 data_in = 17;
            #10 data_in = 10;
        end

    initial
        begin
            $monitor ($time, " %d %b", DP.Y, done);
            $dumpfile ("mul.vcd");
            $dumpvars (0, MUL_test);
        end
endmodule
```

---



```

bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/Multiplication_byrepeated_addition$ iverilog MUL_test.v
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/Multiplication_byrepeated_addition$ vvp a.out
VCD info: dumpfile mul.vcd opened for output.
      0      x x
     35      0 x
     45      17 x
     55      34 x
     65      51 x
     75      68 x
     85      85 x
     95     102 x
    105     119 x
    115     136 x
    125     153 x
    135     170 x
    138     170 1
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/Multiplication_byrepeated_addition$

```

Figure 7.20: Icarus terminal

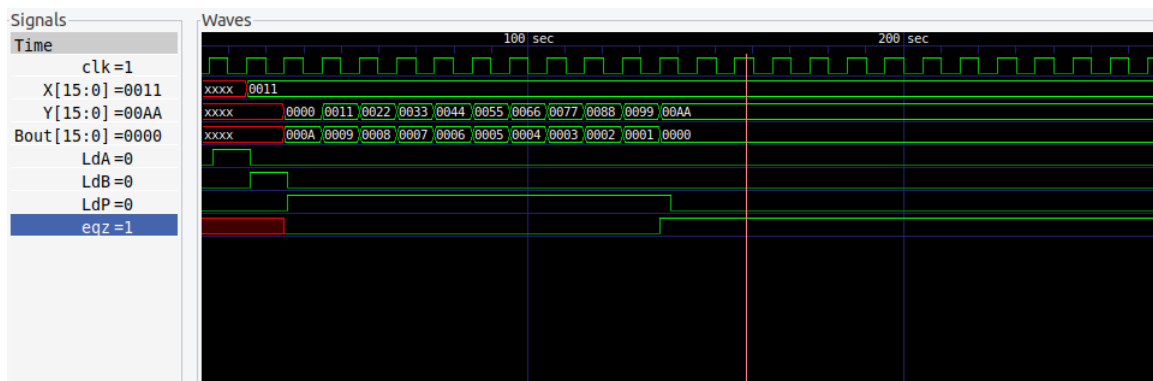


Figure 7.21: GTKwave

### 7.6.2 Simulation in eSim

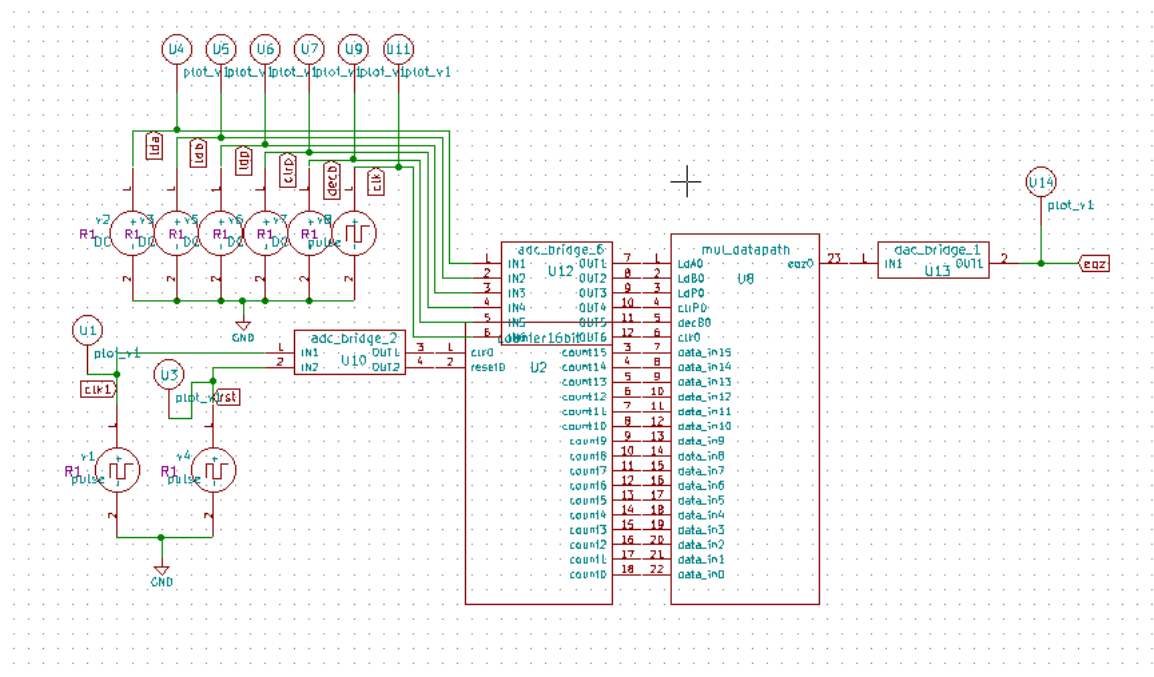


Figure 7.22: Schematic of Multiplication by Repeated Addition

```
mul_by_rep_add.cir.out
reset=0
count=25
=====mul_datapath : New Iteration=====
Instance : 0

Inside foo before eval.....
LdA=0
LdB=0
LdP=0
clnP=0
decB=0
clk=1
data_in=25
eqz=1

Inside foo after eval.....
LdA=0
LdB=0
LdP=0
clnP=0
decB=0
clk=0
data_in=25
eqz=1

No. of Data Rows : 1110
ngspice 1 ->
```

Figure 7.23: NgSpice

## 7.7 Demux 1:8

### 7.7.1 Simulation in Icarus-Verilog

#### Testbench

---

```
'include "demux_8to1.v"

module demux_8to1_tb;

    // Inputs
    reg [2:0] select;
    reg data;

    // ys
    wire [7:0] y;

    // Instantiate the demux_8to1 module
    demux_8to1 dut (
        .select(select),
        .data(data),
        .y(y)
    );

    // Clock generation
    reg clk;
    always #5 clk = ~clk;

    // Testbench stimulus
    initial begin
        $dumpfile("demux_8to1_tb.vcd"); // Specify the VCD file
        $dumpvars(0, demux_8to1_tb);   // Dump all variables

        clk = 0;
        select = 0;
        data = 0;

        // Test Case 1
        select = 3'b000; // Select the first y line
        data = 1; // Set data input to 1
        #10; // Wait for some time
        $display("y: %b", y); // Display the y value
        // Expected y: 00000001

        // Test Case 2
        select = 3'b001; // Select the second y line
        data = 1; // Set data input to 1
        #10; // Wait for some time
        $display("y: %b", y); // Display the y value
        // Expected y: 00000010
```

```

// Test Case 3
select = 3'b010; // Select the third y line
data = 0; // Set data input to 0
#10; // Wait for some time
$display("y: %b", y); // Display the y value
// Expected y: 00000100

// Test Case 4
select = 3'b011; // Select the fourth y line
data = 1; // Set data input to 1
#10; // Wait for some time
$display("y: %b", y); // Display the y value
// Expected y: 00001000

// Test Case 5
select = 3'b100; // Select the fifth y line
data = 0; // Set data input to 0
#10; // Wait for some time
$display("y: %b", y); // Display the y value
// Expected y: 00010000

// Test Case 6
select = 3'b101; // Select the sixth y line
data = 0; // Set data input to 0
#10; // Wait for some time
$display("y: %b", y); // Display the y value
// Expected y: 00100000

// Test Case 7
select = 3'b110; // Select the seventh y line
data = 1; // Set data input to 1
#10; // Wait for some time
$display("y: %b", y); // Display the y value
// Expected y: 01000000

// Test Case 8
select = 3'b111; // Select the eighth y line
data = 0; // Set data input to 0
#10; // Wait for some time
$display("y: %b", y); // Display the y value
// Expected y: 10000000

#100 $finish; // End the simulation
end

endmodule

```

---

```

bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/demux$ iverilog demux_8to1_tb.v
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/demux$ vvp a.out
VCD info: dumpfile demux_8to1_tb.vcd opened for output.
y: 00000001
y: 00000010
y: 00000100
y: 00001000
y: 00010000
y: 00100000
y: 01000000
y: 10000000
bhargav@bhargav-VirtualBox:~/Documents/icarus_iverilog/demux$

```

Figure 7.24: Icarus terminal

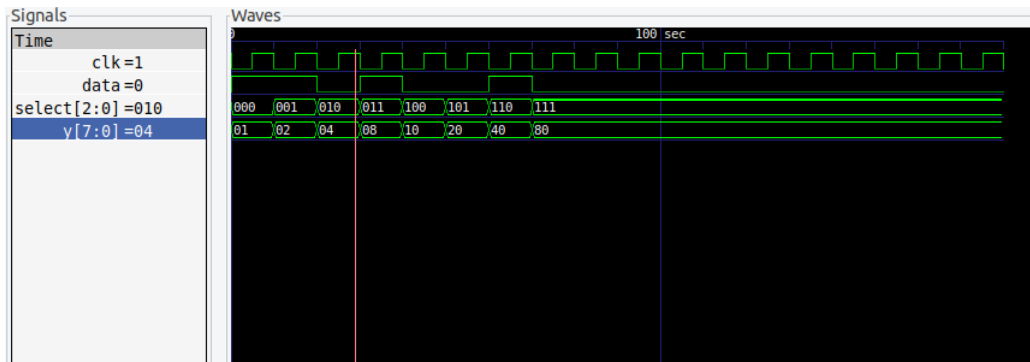


Figure 7.25: GTKWave

## 7.7.2 Simulation in eSim

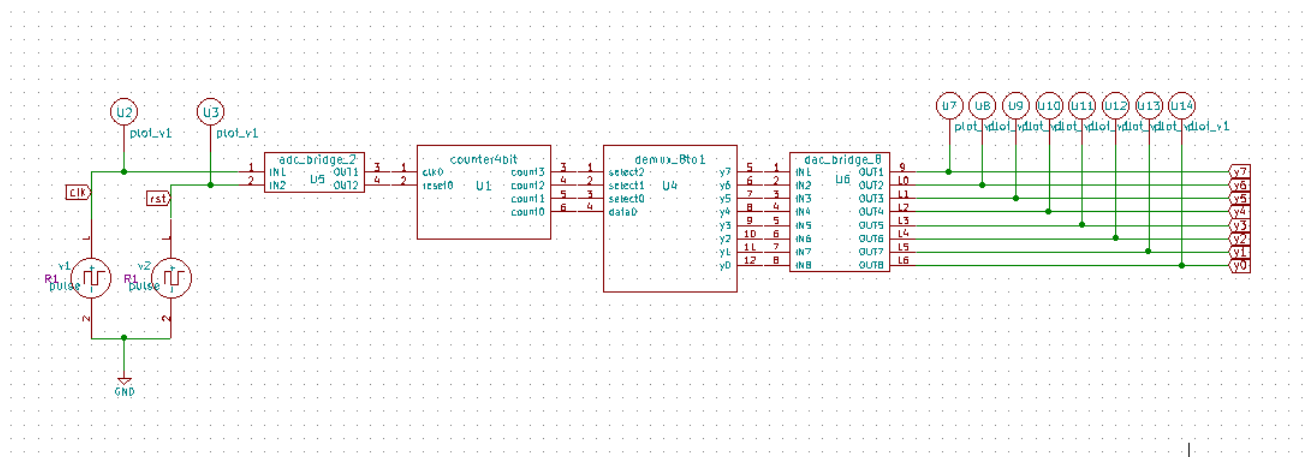


Figure 7.26: Schematic of DEMUX1to8

```

demux_8to1.cir.out
clk=1
reset=0
count=9
=====demux_8to1 : New Iteration=====
Instance : 0

Inside foo before eval.....
select=4
data=0
y=16

Inside foo after eval.....
select=4
data=1
y=16
=====counter4bit : New Iteration=====
Instance : 0

Inside foo before eval.....
clk=1
reset=0
count=9

Inside foo after eval.....
clk=1
reset=0
count=9

```

Figure 7.27: NgSpice

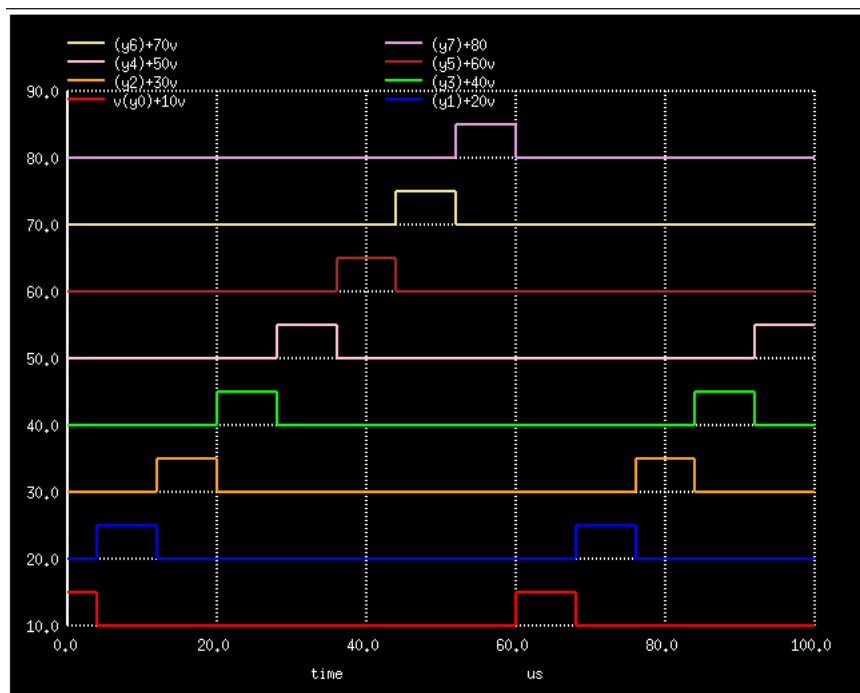


Figure 7.28: Waveform

## 7.8 2:1 Multiplexer

### 7.8.1 Simulation in ModelSIM

#### Verilog Code

---

```
module mux_1bit (  
    input a,  
    input b,  
    input x,  
    output y);  
wire not_x;  
wire bit1;  
wire bit2;  
  
not not1 (not_x,x);  
and and1 (bit1,a,not_x);  
and and2 (bit2,b,x);  
or or1 (y,bit1,bit2);  
  
endmodule
```

---

#### Testbench

---

```
module tb_mux ();  
reg a;  
reg b;  
reg x;  
wire y;  
  
mux_1bit MUX1(  
    .a(a),  
    .b(b),  
    .x(x),  
    .y(y)  
);  
  
initial begin  
    #1; a=1;b=0;x=1;  
    #1; a=0;b=1;x=0;  
    #1; a=0;b=1;x=1;  
    #1; a=0;b=0;x=1;  
    #1; a=1;b=1;x=1;  
    #1; a=0;b=1;x=0;  
    #5; $stop;  
end  
endmodule
```

---

Waveform

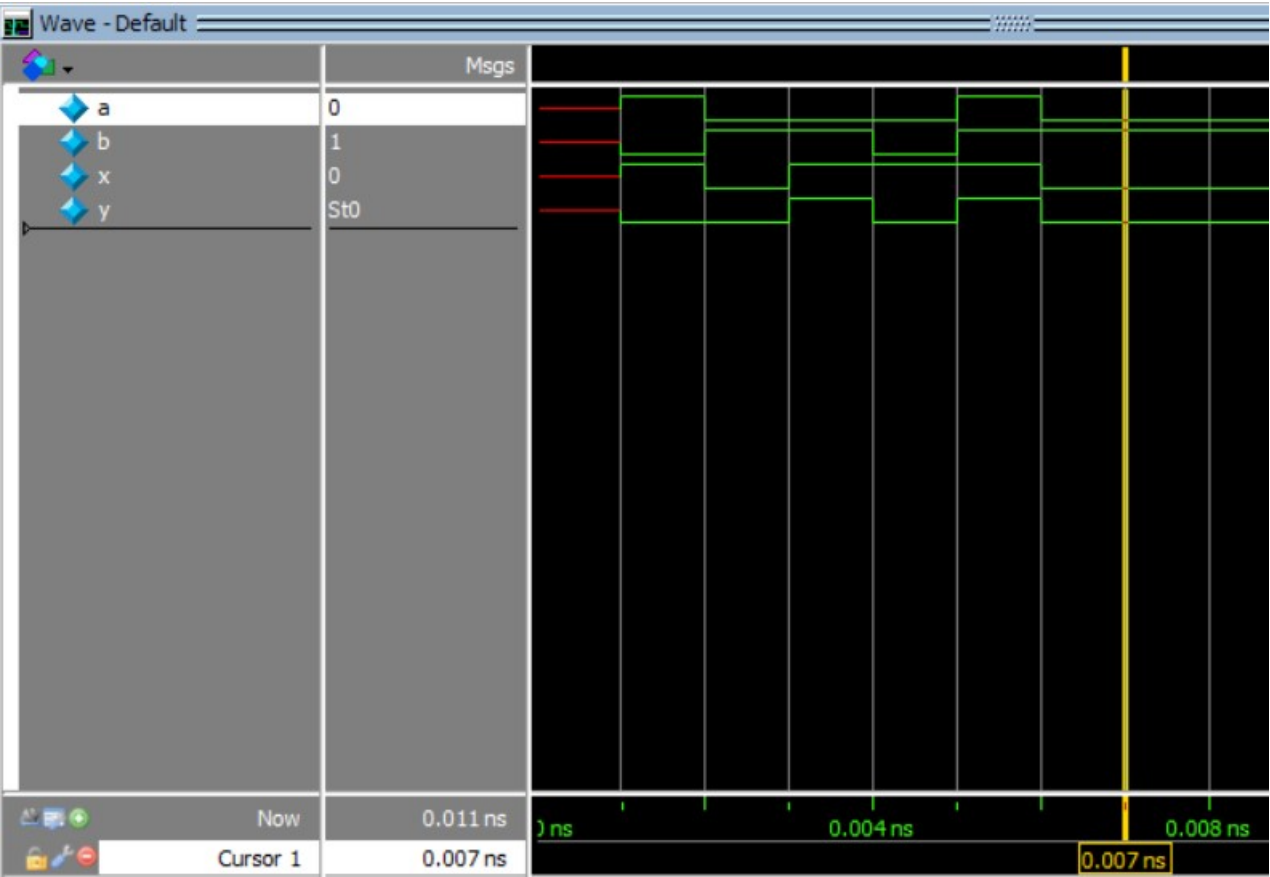


Figure 7.29: ModelSIM Waveform



## 7.8.2 Simulation in eSim

### Test Circuit Schematics

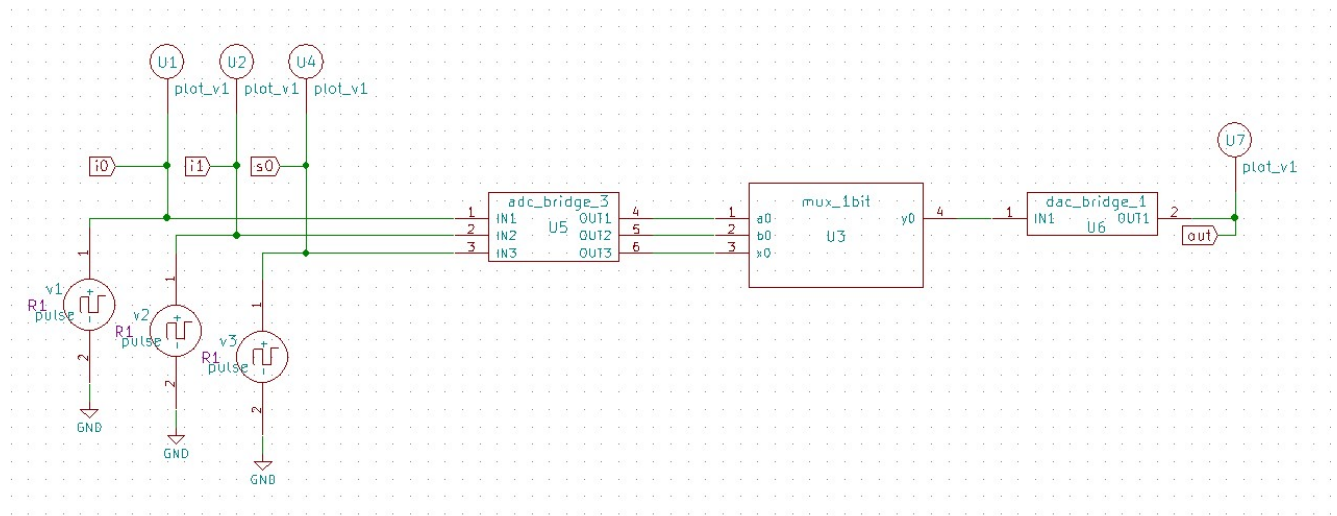


Figure 7.30: MUX Test Circuit

### NgSpice Terminal

```
ngspice -p E:\FOSSEE\Digital\mux_1bit\mux_1bit.cir.out

Inside foo after eval.....
a=1
b=1
x=0
y=1
=====mux_1bit : New Iteration=====
Instance : 0

Inside foo before eval.....
a=1
b=1
x=0
y=1

Inside foo after eval.....
a=1
b=1
x=1
y=1
=====mux_1bit : New Iteration=====
Instance : 0

Inside foo before eval|
```

Figure 7.31: NgSpice

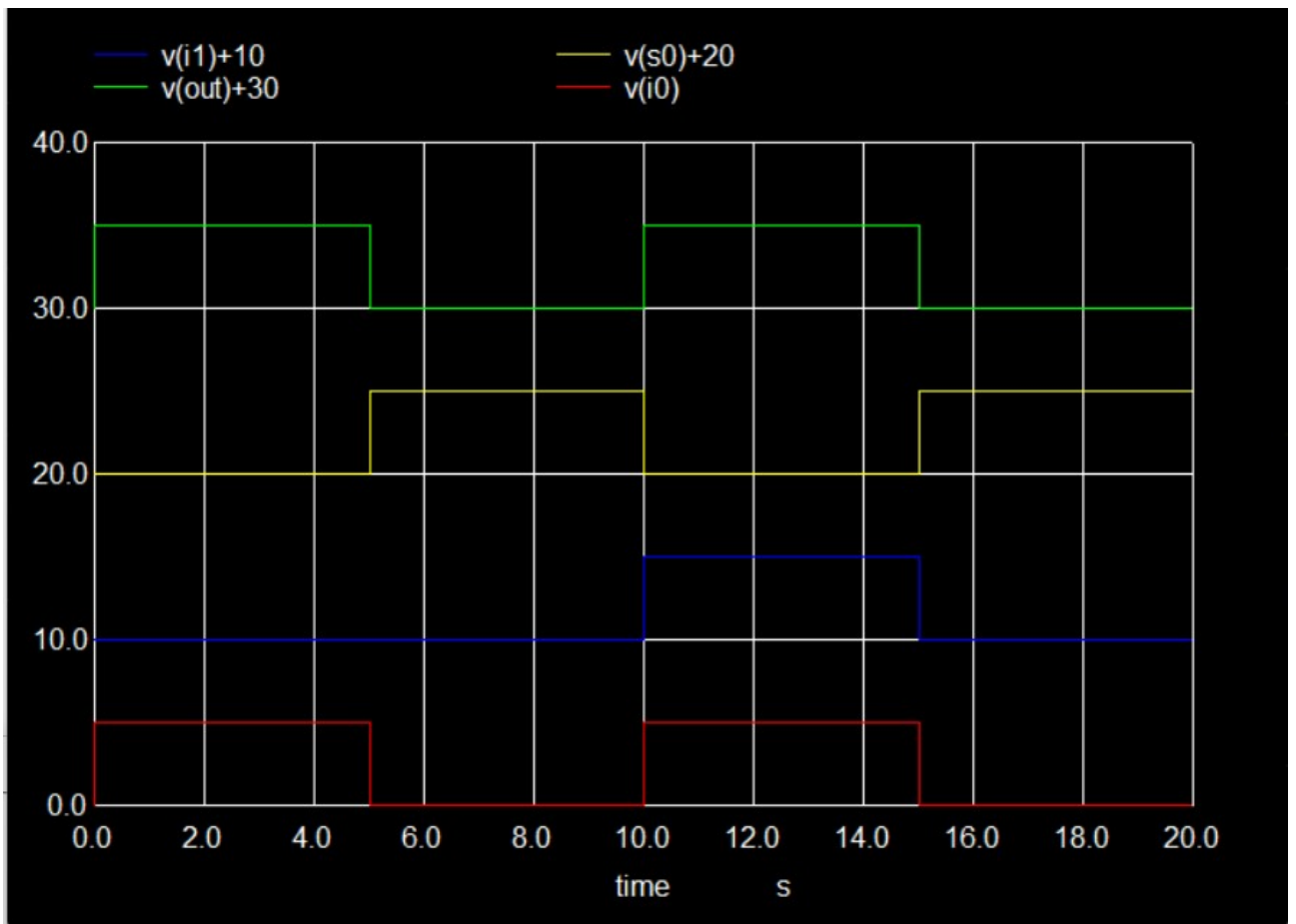


Figure 7.32: Waveform

## 7.9 4-Bit ALU

### 7.9.1 Simulation in ModelSIM

#### Verilog Code

---

```
module ALU_4bit(  
    input [3:0] A,  
    input [3:0] B,  
    input [3:0] opcode,  
    output [3:0] out,  
    output zero  
);  
    reg [4:0] result;  
    assign out = result;  
    always @(*)  
    begin  
        case (opcode)  
            4'b0000 : result = (A + B);  
            4'b0001 : result = (A - B);  
            4'b0010 : result = (A & B);  
            4'b0011 : result = (A | B);  
            4'b0100 : result = (A ^ B);  
            4'b0101 : result = (~A);  
            4'b0110 : result = (~B);  
            4'b0111 : result = (A >> 1);  
            4'b1000 : result = (A << 1);  
            4'b1001 : result = (B >> 1);  
            4'b1010 : result = (B << 1);  
            default : result = 4'b0000;  
        endcase  
    end  
  
    assign zero = (result == 4'b0000);  
  
endmodule
```

---

# 7.9.2 Simulation in eSim

## Test Circuit Schematics

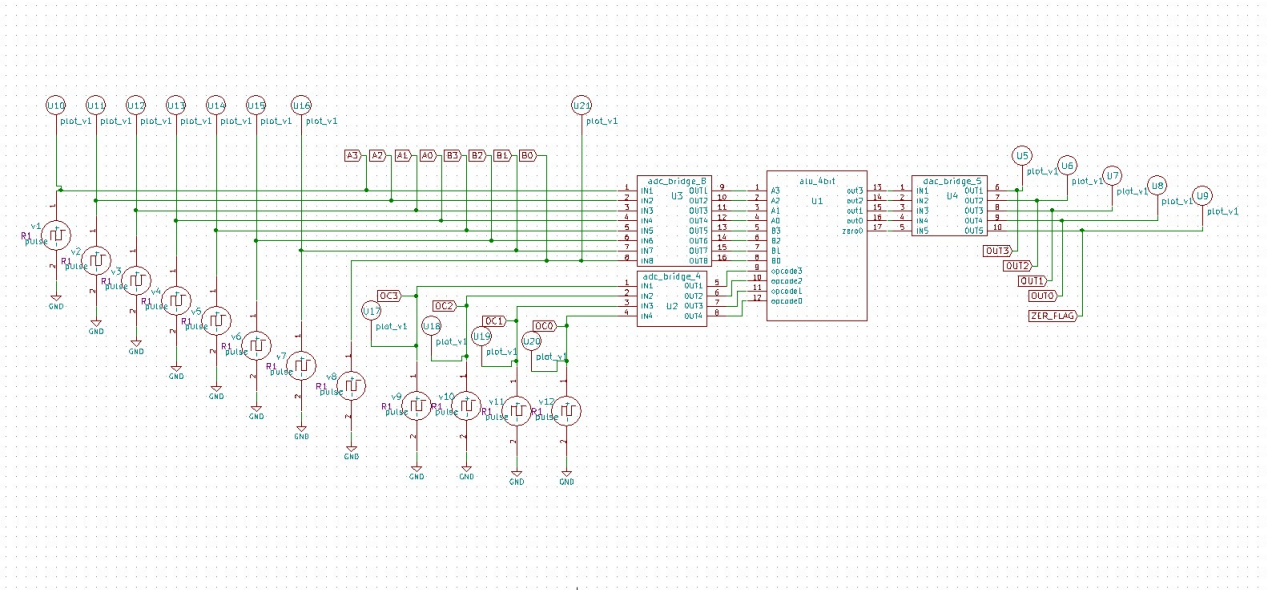


Figure 7.33: ALU Test Circuit

## NgSpice Terminal

```
ngspice -p E:\FOSSEE\Digital\TEST_ALU\TEST_ALU.cir.out

Inside foo after eval.....
A=5
B=4
opcode=1
out=1
zero=0
=====alu_4bit : New Iteration=====
Instance : 0

Inside foo before eval.....
A=5
B=4
opcode=1
out=1
zero=0

Inside foo after eval.....
A=15
B=15
opcode=1
out=0
zero=1
=====
```

Figure 7.34: NgSpice

## NgSpice Plots

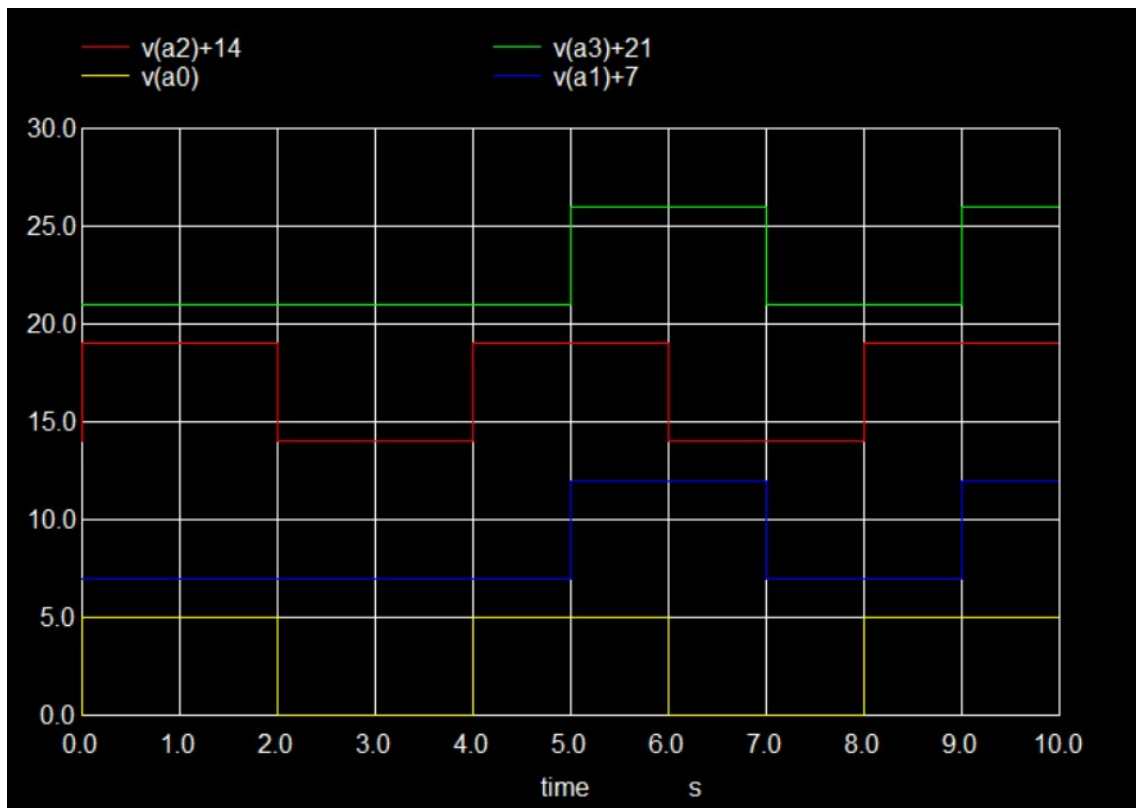


Figure 7.35: ALU Input A[3:0]

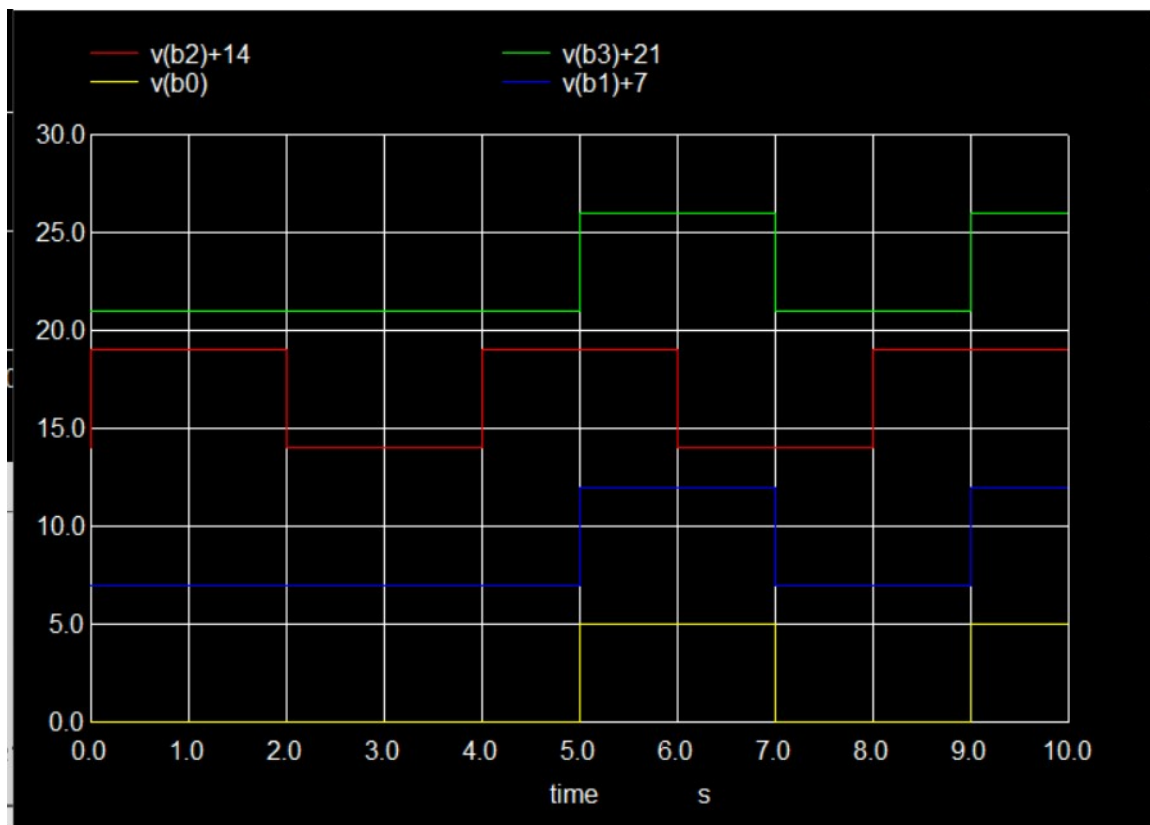


Figure 7.36: ALU Input B[3:0]

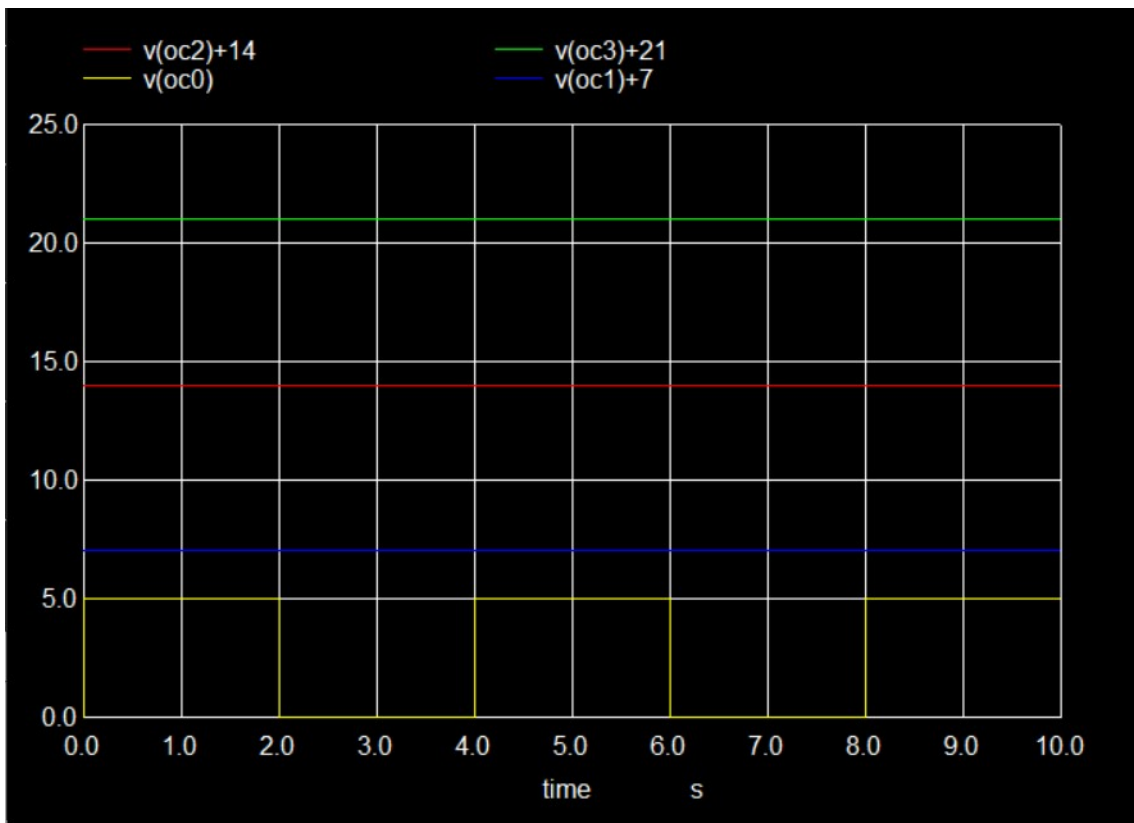


Figure 7.37: ALU OpCode

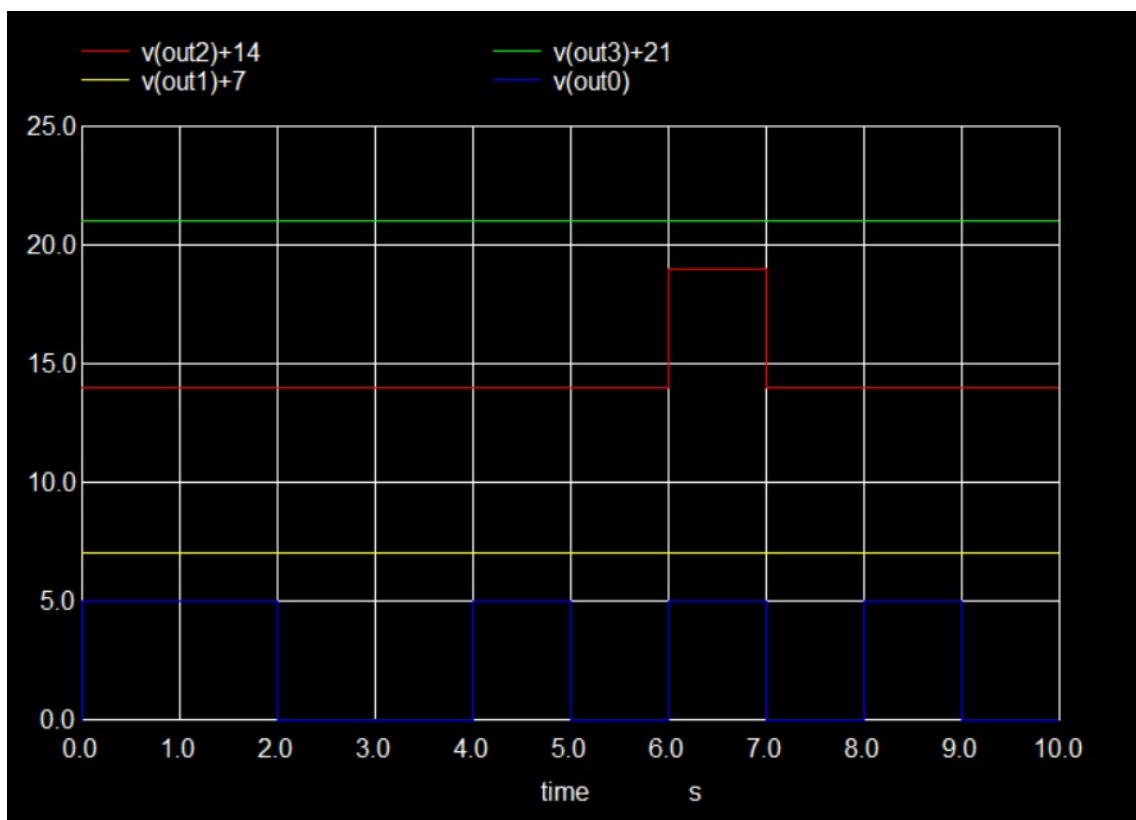


Figure 7.38: ALU Output

# Chapter 8

## RISC V -Processor

Simulation of RISC V single cycle processor of 32bit which is designed in Verilog. RISC-V uses 32-bit instructions. RISC-V consists of defining the following instruction formats: R-type, I-type, S-Type, B-Type, U-type, and J-type. R-type instructions operate on three registers. I-type, S-type and B-type instructions operate on two registers and a 12-bit immediate. U-type and J-type (jump) instructions operate on one 20-bit immediate.[2]

### 8.1 Design of RISC V

#### 8.1.1 Arithmetic Logic Unit (ALU)

Design an ALU that can perform a subset of the ALU operations of a full Processor ALU.

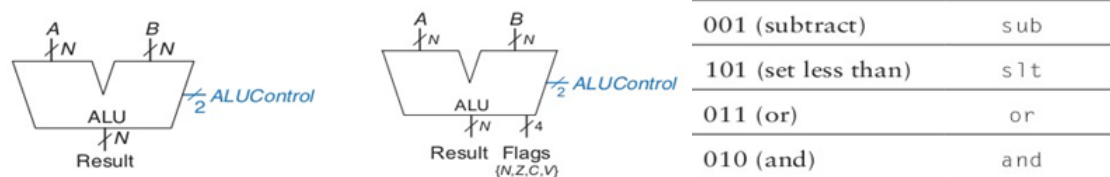


Figure 8.1: Block Diagram of simple ALU and instructions [2]

The ALU will generate a 32-bit output that we will call Result and an additional 1-bit flag Zero that will be set to logic-1 if all the bits of Result are 0. The different operations will be selected by a 3-bit control signal called ALUControl according to the table.

Comparison	Signed	Unsigned
$=$	$Z$	$Z$
$\neq$	$\overline{Z}$	$\overline{Z}$
$<$	$N \oplus V$	$\overline{C}$
$\leq$	$Z + (N \oplus V)$	$Z + \overline{C}$
$>$	$\overline{Z} \bullet \overline{(N \oplus V)}$	$\overline{Z} \bullet C$
$\geq$	$\overline{(N \oplus V)}$	$C$

We have two types of instructions. The three instructions add, sub, and slt are arithmetic operations, whereas the two remaining and, or are logical operations. Therefore, we have two separate groups of operations. Also, the above diagram shows the completed alu with carry, overflow, negative and zero flags.



## 8.1.2 Control Unit

The control unit computes the control signals based on the opcode and funct fields of the instruction, Instr[31:25], Instr[14:12] and Instr[6:0]. Most of the control information comes from the opcode, but for further operations function fields are used.[2]

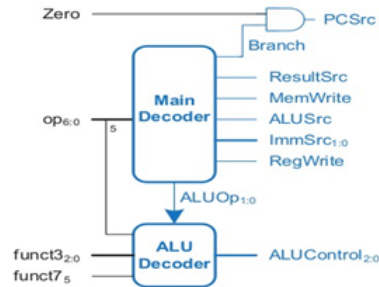


Figure 8.3: Block Diagram of control unit implemented [2]

Instruction	Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
lw	0000011	1	00	1	0	1	0	00
sw	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
beq	1100011	0	10	0	0	x	1	01

Figure 8.4: Table of main decoder implemented [2]

ALUOp	funct3	{op <sub>5</sub> , funct <sub>7:5</sub> }	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt
	110	x	011 (or)	or
	111	x	010 (and)	and

Figure 8.5: Table of alu decoder implemented [2]

### 8.1.3 Designing Microarchitecture

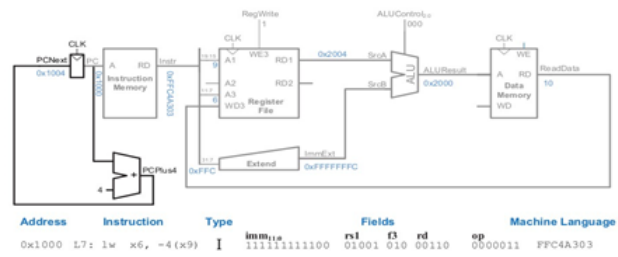


Figure 8.6: Datapath for the load instruction [2]

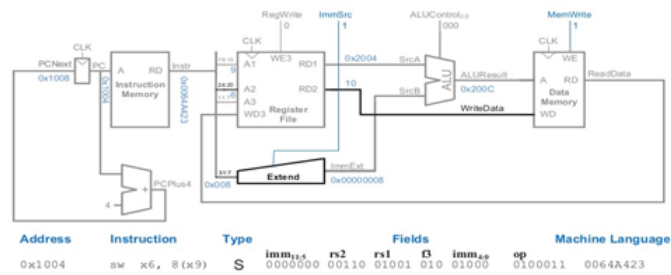


Figure 8.7: Datapath for Store Word Instruction [2]

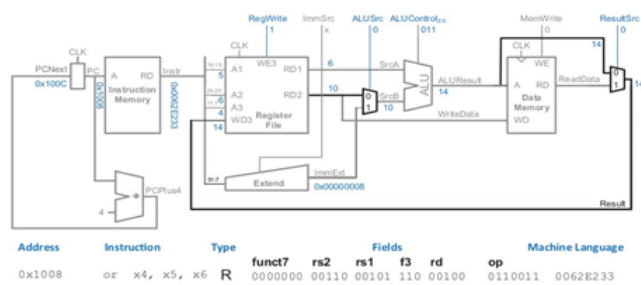


Figure 8.8: Datapath for R-Type Instruction [2]

## 8.2 Simulation of RISC V

### 8.2.1 Schematic

The RISC V Single Cycle Top module is first simulated in Makerchip and after adding other dependency files the model is created using NgVeri and implemented in eSim. The inputs for the model are clk and rst. We are using adc and dac to convert the analog and digital signals. Also, plot V1 is used to plot the signals. The output of the model is the Result wire from the mux.

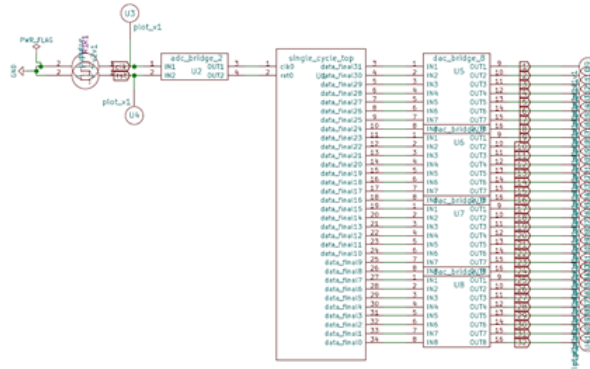


Figure 8.9: Schematic of the RISC V Processor in eSim

### 8.2.2 Analysis

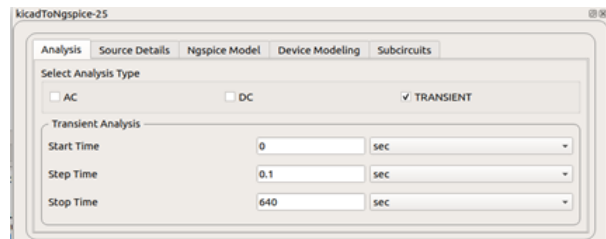


Figure 8.10: Transient analysis

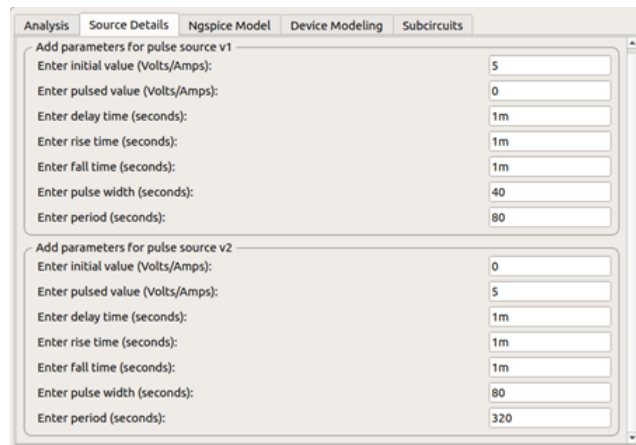


Figure 8.11: Source details used

### 8.2.3 Simulation

We also have to add an instruction file to simulate the model. Here we are using 0062E3B3 Instruction which refers to or operation of 5 and 4 which gives the answers as 5.

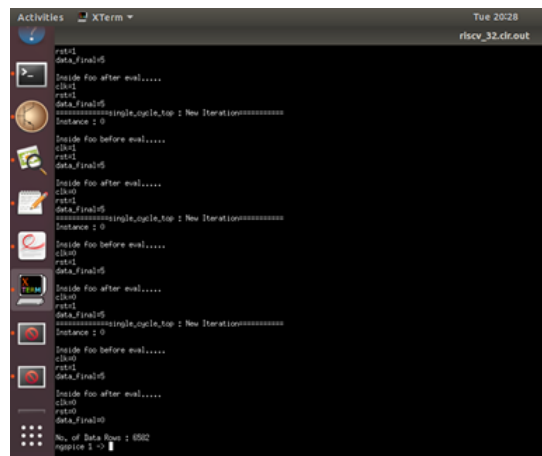


Figure 8.12: Simulation of RISC V Processor in eSim

# Chapter 9

## UART TX

A universal asynchronous receiver-transmitter (UART) is a computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable. It sends data bits one by one, from the least significant to the most significant, framed by start and stop bits so that precise timing is handled by the communication channel.[3]

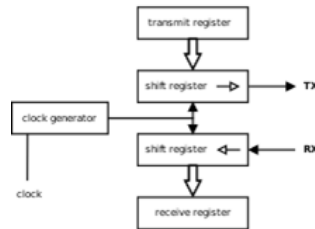


Figure 9.1: Block diagram for a UART  
[3]

Transmission operation is simpler as the timing does not have to be determined from the line state, nor is it bound to any fixed timing intervals. As soon as the sending system deposits a character in the shift register (after completion of the previous character), the UART generates a start bit, shifts the required number of data bits out to the line, generates and sends the parity bit (if used), and sends the stop bits. Since transmission of a single or multiple characters may take a long time relative to CPU speeds, a UART maintains a flag showing busy status so that the host system knows if there is at least one character in the transmit buffer or shift register.[3]

### 9.0.1 Schematic

The uart-tx model is simulated in Makerchip-NgVeri and the model is created. The model is imported to the schematic and adc and dac are used to convert the analog and digital signals. It has inputs such as clk, rst, enable and data line. While the output includes data line and busy line.[4]

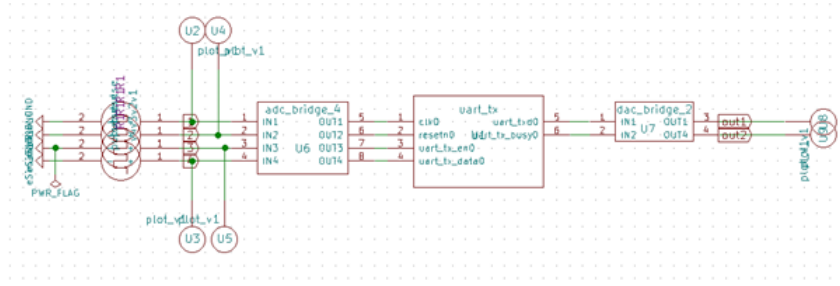


Figure 9.2: Schematic of uart-tx model implemented

## 9.0.2 Simulation

Transient analysis is used here (.tran 0.1e-00 160e-00 0e-00) and sources are varied for the successful simulation and analysis of the model. The simulation results are shown below.

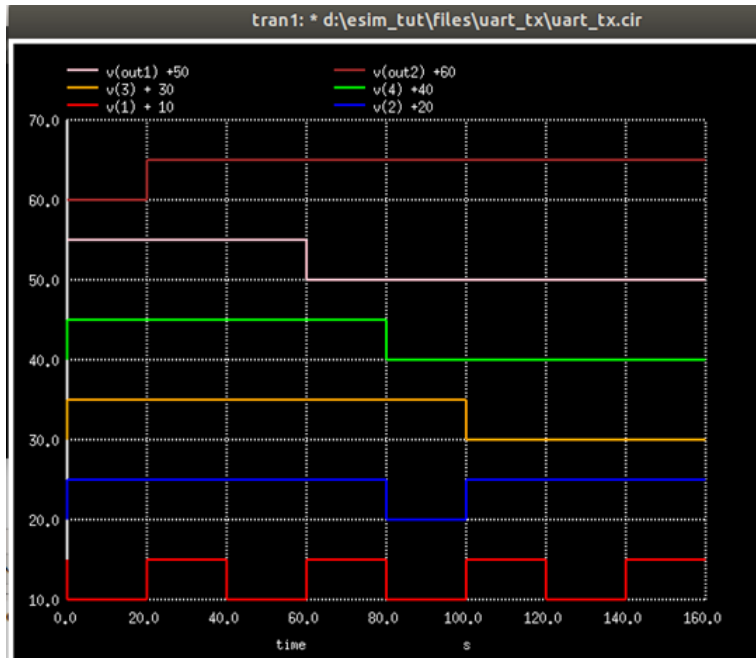


Figure 9.3: Simulation of uart-tx model implemented

# Chapter 10

## Kogge Stone adder

In computing, the KoggeStone adder (KSA or KS) is a parallel prefix form carry look-ahead adder.

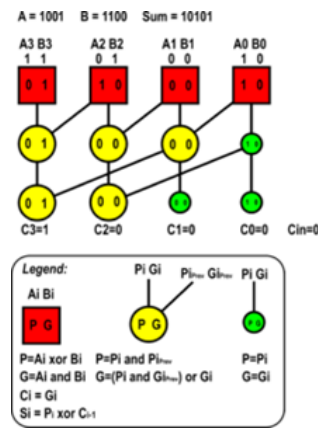


Figure 10.1: : carry generator of a 4-bit KoggeStone adder with zero carry-in, Radix-2, valency-2

[5]

Each vertical stage produces a "propagate" and a "generate" bit, as shown. The culminating generate bits (the carries) are produced in the last stage (vertically), and these bits are XOR'd with the initial propagate after the input (the red boxes) to produce the sum bits. E.g., the first (least-significant) sum bit is calculated by XORing the propagate in the farthest-right red box (a "1") with the carry-in (a "0"), producing a "1". The second bit is calculated by XORing the propagate in second box from the right (a "0") with C0 (a "0"), producing a "0".[5]

### 10.0.1 Schematic

The kogge stone module is first simulated in Makerchip and after adding other dependency files the model is created using NgVeri and implemented in eSim. The inputs for the model are 8bit numbers a and b . We are using adc and dac to convert the analog and digital signals. Also, plot V1 is used to plot the signals. The output of the model contains sum of 8bits and a carry port.[6]

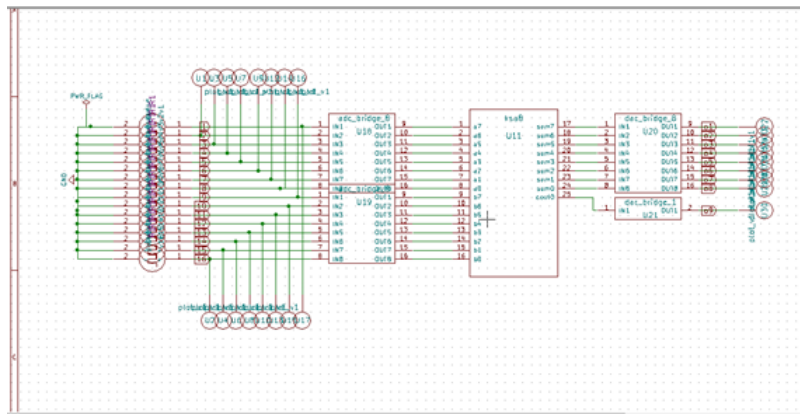


Figure 10.2: Kogge Stone Adder schematic in eSim

### 10.0.2 Simulation

Transient analysis is used here (.tran 0.1e-00 160e-00 0e-00) and sources are varied for the successful simulation and analysis of the model. The simulation results are shown below.

```

=====k1a8 : New Iteration=====
Instance : 0

Inside foo before eval....
w01
w02
sum=0
cout=0

Inside foo after eval....
w01
w04
sum=3
cout=0

=====k1a8 : New Iteration=====
Instance : 0

Inside foo before eval....
w01
w04
sum=3
cout=0

Inside foo after eval....
w01
w04
sum=3
cout=0

=====k1a8 : New Iteration=====
Instance : 0

Inside foo before eval....
w01
w04
sum=3
cout=0

Inside foo after eval....
w01
w04
sum=247
cout=0

=====k1a8 : New Iteration=====
Instance : 0

Inside foo before eval....

```

Figure 10.3: Kogge Stone Adder simulation in eSim



# Chapter 11

## 4\*4 Array Multiplier

An array multiplier is a digital combinational circuit used for multiplying two binary numbers by employing an array of full adders and half adders. This array is used for the nearly simultaneous addition of the various product terms involved. To form the various product terms, an array of AND gates is used before the Adder array. For implementation of array multiplier with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in figure. The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3c2c1c0.[7]

$$\begin{array}{r} \begin{array}{cc} \text{b1} & \text{b0} \\ \text{a1} & \text{a0} \end{array} \\ \hline \begin{array}{cc} \text{a0b1} & \text{a0b0} \end{array} \\ \begin{array}{cc} \text{a1b1} & \text{a1b0} \end{array} \\ \hline \begin{array}{cccc} \text{c3} & \text{c2} & \text{c1} & \text{c0} \end{array} \end{array}$$

Figure 11.1: : 2bit array multiplication  
[7]

### 11.0.1 Schematic

The array multiplier Verilog code is simulated and created the model using Makerchip-NgVeri . The inputs for the model are 4bit numbers a and b . We are using adc and dac to convert the analog and digital signals. Also, plot V1 is used to plot the signals. The output of the model is product of 8bits. [8]

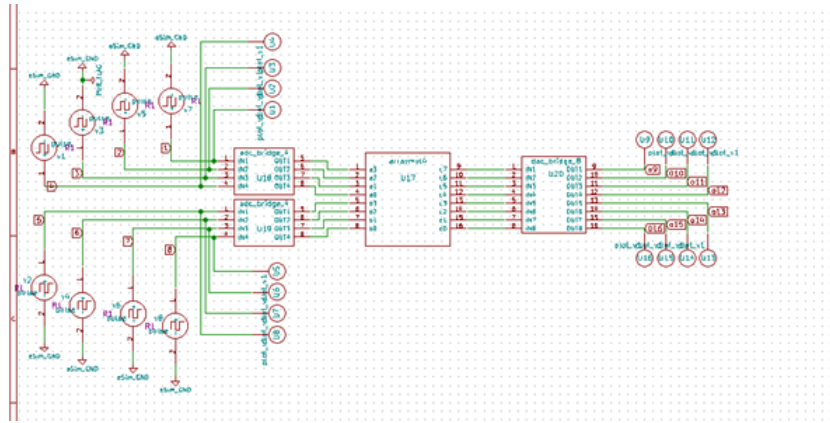


Figure 11.2: Schematic of array multiplier

## 11.0.2 Simulation

Transient analysis is used here (.tran 0.1e-00 160e-00 0e-00) and sources are varied for the successful simulation and analysis of the model. The simulation results are shown below.

```

Activities  Xterm  (nu
multab

Inside foo before eval.....
a[7]
b[11]
c[77]

Inside foo after eval.....
a[7]
b[11]
c[77]

=====arraymult4 : New Iteration=====
Instance : 0

Inside foo before eval.....
a[7]
b[11]
c[77]

Inside foo after eval.....
a[7]
b[11]
c[77]

=====arraymult4 : New Iteration=====
Instance : 0

Inside foo before eval.....
a[7]
b[11]
c[77]

Inside foo after eval.....
a[3]
b[10]
c[30]

=====arraymult4 : New Iteration=====
Instance : 0

Inside foo before eval.....
a[3]
b[10]
c[30]

Inside foo after eval.....
a[3]
b[14]
c[42]

```

Figure 11.3: Simulation of array multiplier

# Chapter 12

## Clock Divider

A frequency divider, also called a clock divider or scaler or prescaler, is a circuit that takes an input signal of a frequency,  $f_{in}$ , and generates an output signal of a frequency:

$$f_{out} = f_{in}/n$$

where  $n$  is an integer. Phase-locked loop frequency synthesizers make use of frequency dividers to generate a frequency that is a multiple of a reference frequency. Frequency dividers can be implemented for both analog and digital applications. [9]

### 12.0.1 Schematic

The clock divider model is simulated in Makerchip-NgVeri and the model is created. The model is imported to the schematic and adc and dac are used to convert the analog and digital signals. It has inputs such as  $clk$  and  $rst$ . While the output is the divided clock.[10]

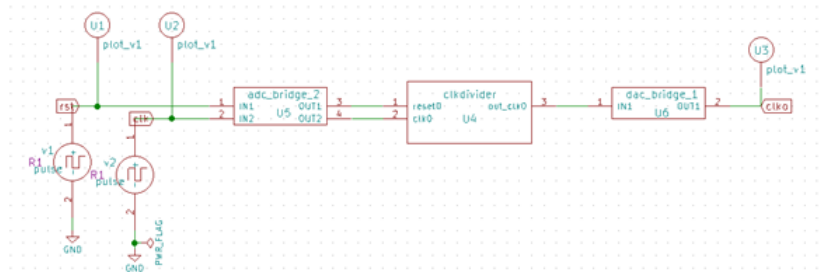


Figure 12.1: Schematic of clock -divider model implemented

### 12.0.2 Simulation

Transient analysis is used here (.tran 0.1e-00 160e-00 0e-00) and sources are varied for the successful simulation and analysis of the model. The simulation results are shown below.

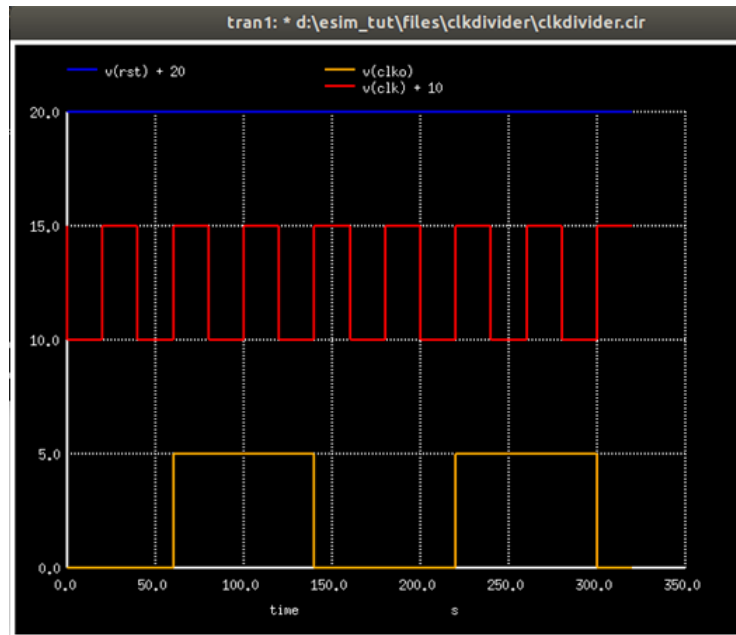


Figure 12.2: Simulation of clock-divider model implemented

# Chapter 13

## 16 Bit Processor

Calcu-16 has eight general purpose registers named r0-r7. Each register is 16-bits wide. There are two other registers, the IR (Instruction Register 26-bit) and the PC (Program Counter 16-bit).

---

### Register Format

0:3	4:6	7:9	10:12
op-code	registers		unused

0000	000	000	000	0000000000000000
r1	r2	r3		

### Immediate/Memory Format

0:3	4:6	7:9	10:25
op-code	registers	immediate/addr	

0000	000	000	0000000000000000
r1	r2		

---

### 13.0.1 Schematic

The processor model is simulated in Makerchip-NgVeri and the model is created. The model is imported to the schematic and adc and dac are used to convert the analog and digital signals. It has inputs is clk. While the output includes.[11]

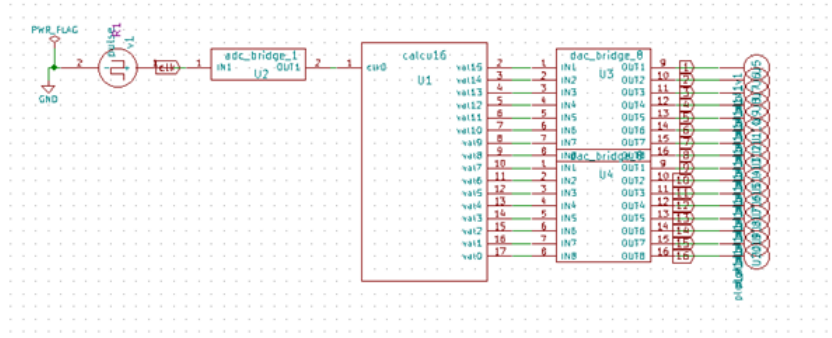


Figure 13.1: Schematic of 16bit processor model implemented

## 13.0.2 Simulation

Transient analysis is used here (.tran 0.1e-00 160e-00 0e-00) and sources are varied for the successful simulation and analysis of the model. The bin file has to be added for simulation .

---

The code and result are shown below :

0010000111000000000000000001	addi r0, r7, 1	- Load 1 into r0.
0010001111000000000000000001	addi r1, r7, 1	- Load 1 into r1.
0010010111111111111111111111	addi r2, r7, 65535	- Load 65535 into r2.
0001001000001000000000000000	add r1, r0, r1	- Add r0 to r1.
010000101000000000000000100	jeq r1, r2, 4	- Hang if r1 == r2.
0011000000000000000000000011	jmp 3	- Jump to address 3.

---

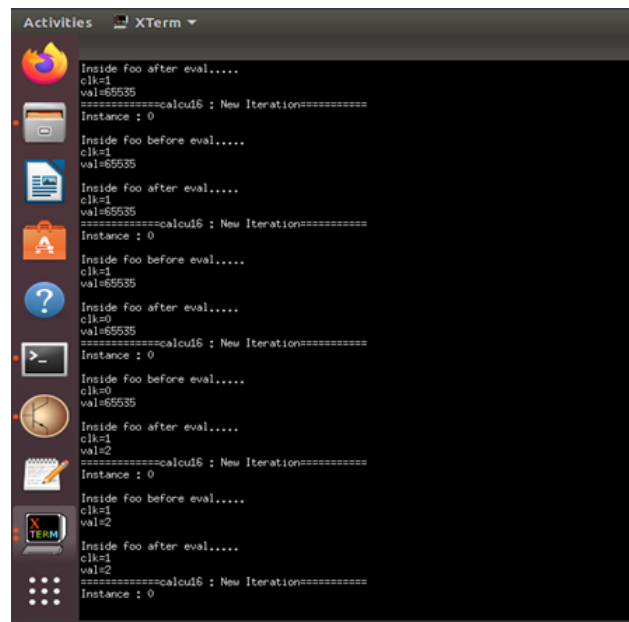


Figure 13.2: Simulation of 16bit processor model implemented

## Chapter 14

# CD4585 4-Bit Magnitude Comparator

CD4585BMS is a 4-bit magnitude comparator designed for use in computer and logic applications that require the comparison of two 4-bit words. This logic circuit determines whether one 4-bit word (Binary or BCD) is less than, equal to or greater than a second 4-bit word.

### 14.0.1 Pin Diagram

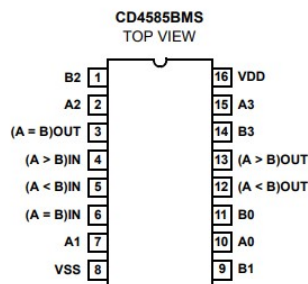


Figure 14.1: Pin Diagram of CD4585

### 14.0.2 Schematic

The processor model is simulated in Makerchip-NgVeri and the model is created. The model is imported to the schematic and adc and dac are used to convert the analog and digital signals.





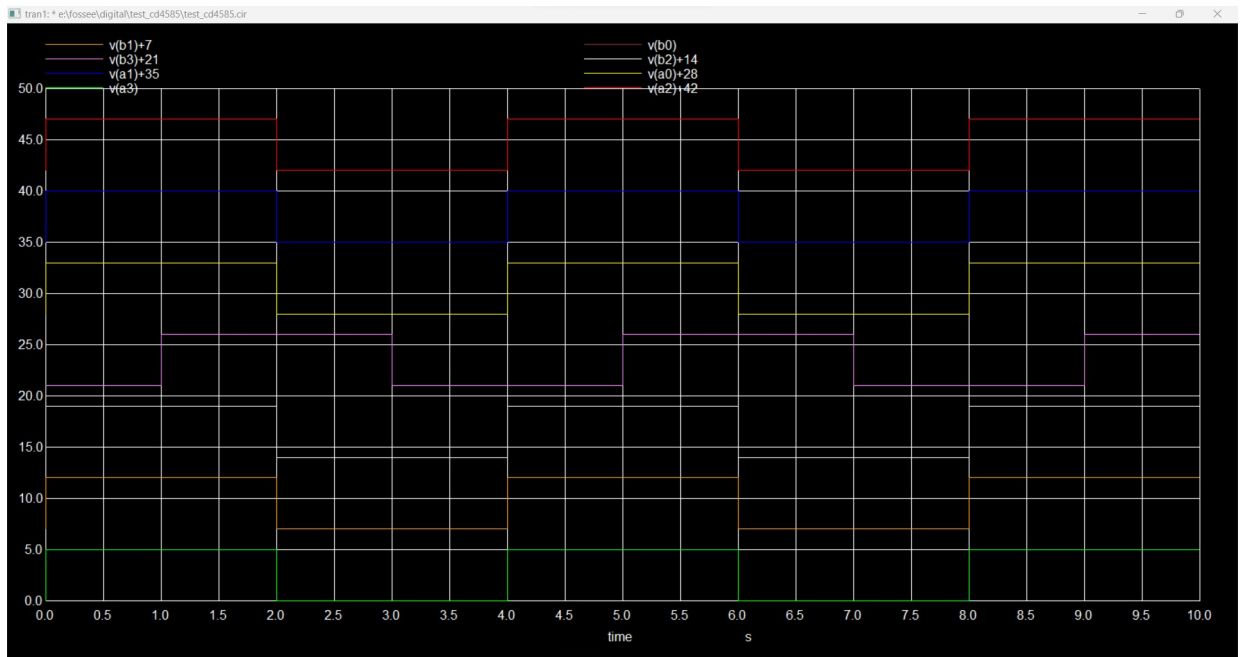


Figure 14.3: Input Waveform

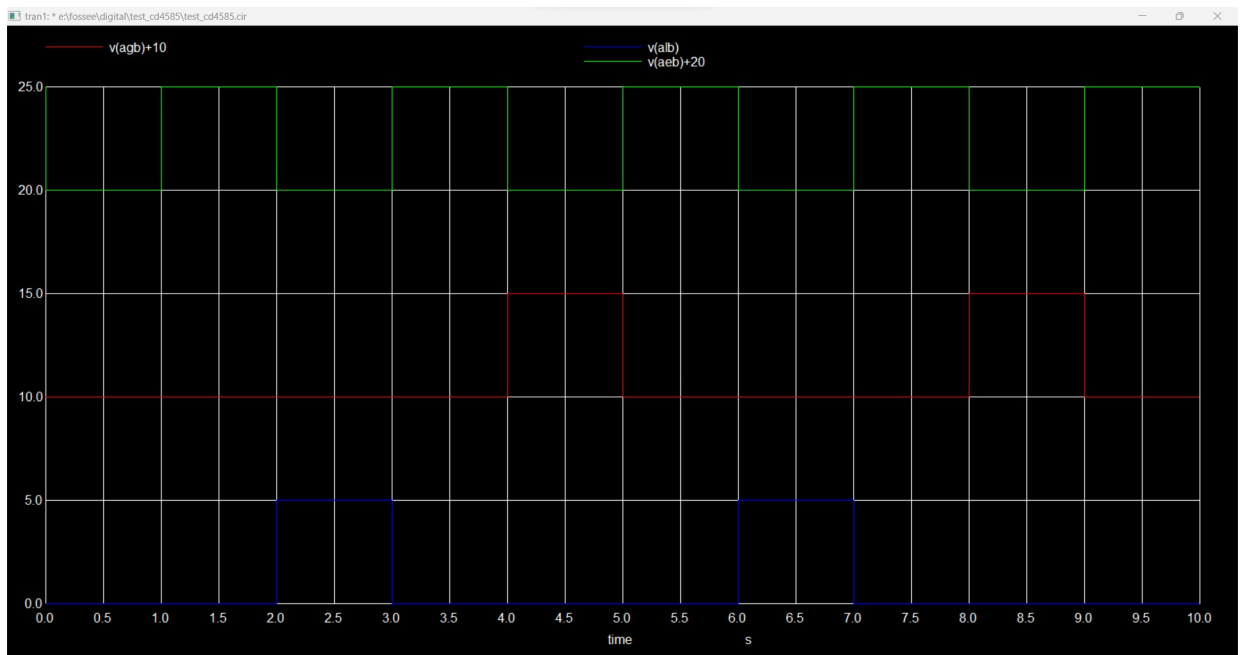


Figure 14.4: Input Waveform

# Chapter 15

## 4-Bit ALU

This ALU takes 4 bit input and processes the input bits dependin-  
gupon the OPCODE.

### 15.0.1 Schematic

The ALU model is simulated in Makerchip-NgVeri and the model is  
created. The model is imported to the schematic and adc and dac are  
used to convert the analog and digital signals.

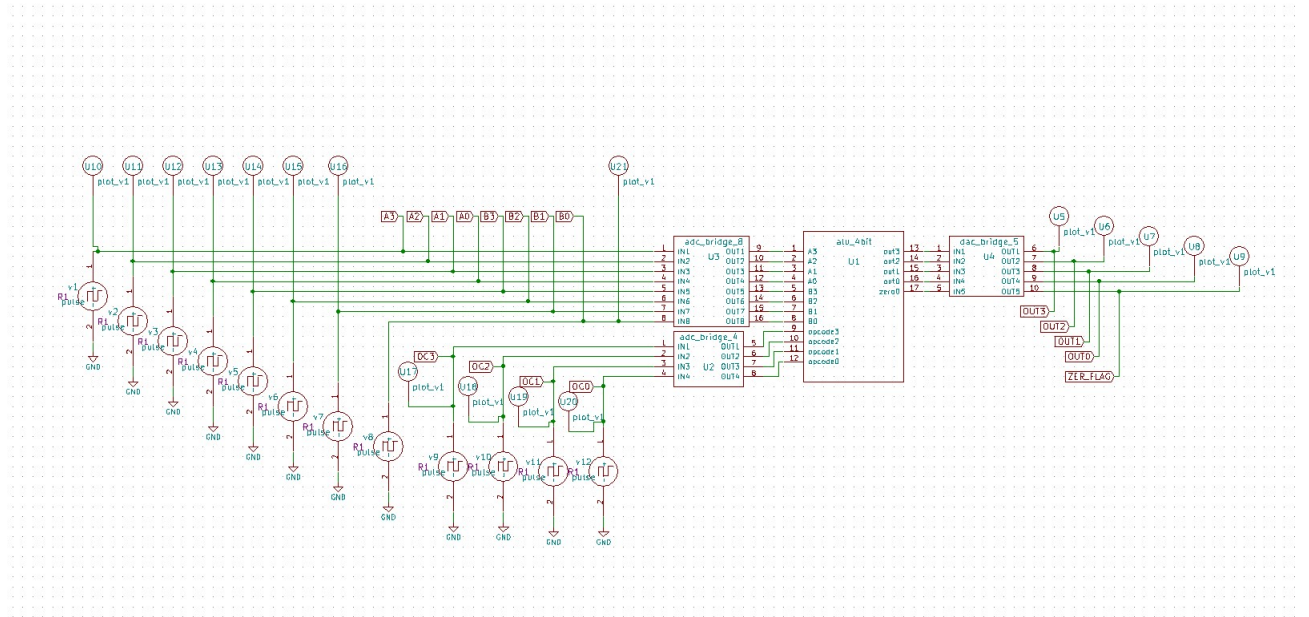


Figure 15.1: ALU Test Circuit

### 15.0.2 Simulation

We have run this ALU for addition, the OpCode, inputs and output  
waveforms are given below:

The code is shown below :

```
module ALU_4bit(  
    input [3:0] A,  
    input [3:0] B,  
    input [3:0] opcode,  
    output [3:0] out,  
    output zero  
);  
    reg [4:0] result;  
    assign out = result;  
    always @(*)  
    begin  
        case (opcode)  
            4'b0000 : result = (A + B);  
            4'b0001 : result = (A - B);  
            4'b0010 : result = (A & B);  
            4'b0011 : result = (A | B);  
            4'b0100 : result = (A ^ B);  
            4'b0101 : result = (~A);  
            4'b0110 : result = (~B);  
            4'b0111 : result = (A >> 1);  
            4'b1000 : result = (A << 1);  
            4'b1001 : result = (B >> 1);  
            4'b1010 : result = (B << 1);  
            default : result = 4'b0000;  
        endcase  
    end  
  
    assign zero = (result == 4'b0000);  
  
endmodule
```

---

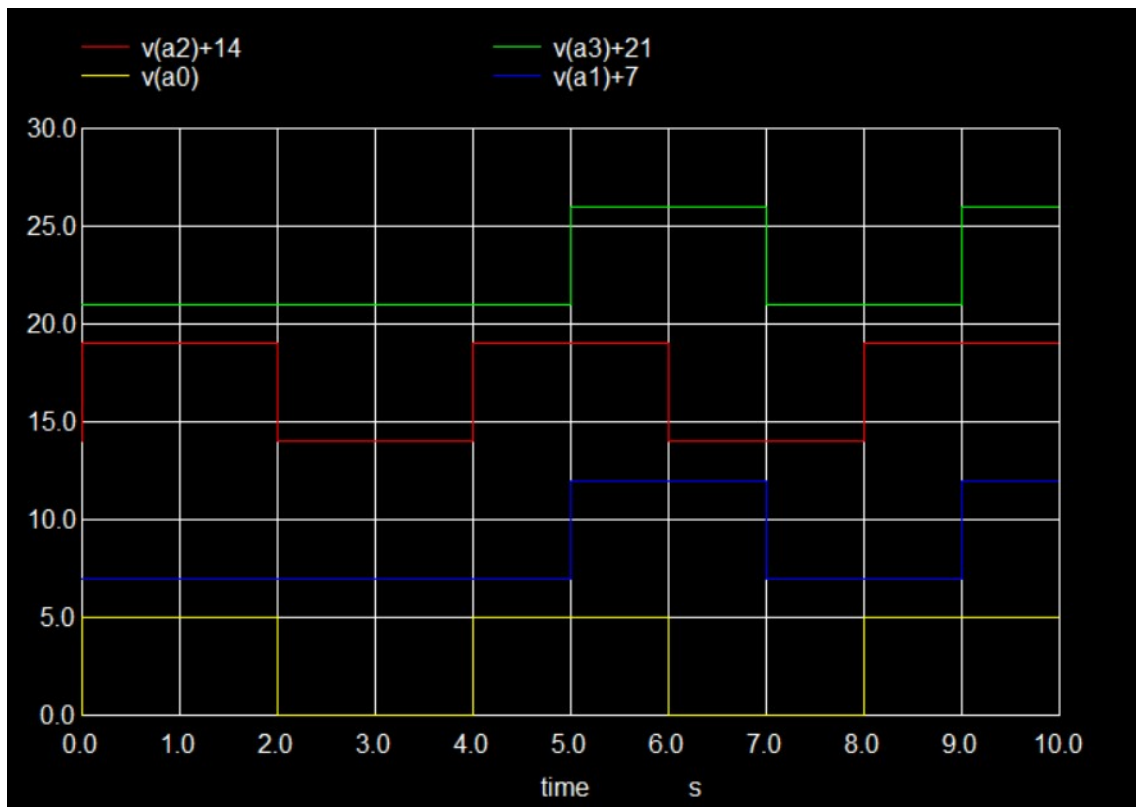


Figure 15.2: Input Waveform A

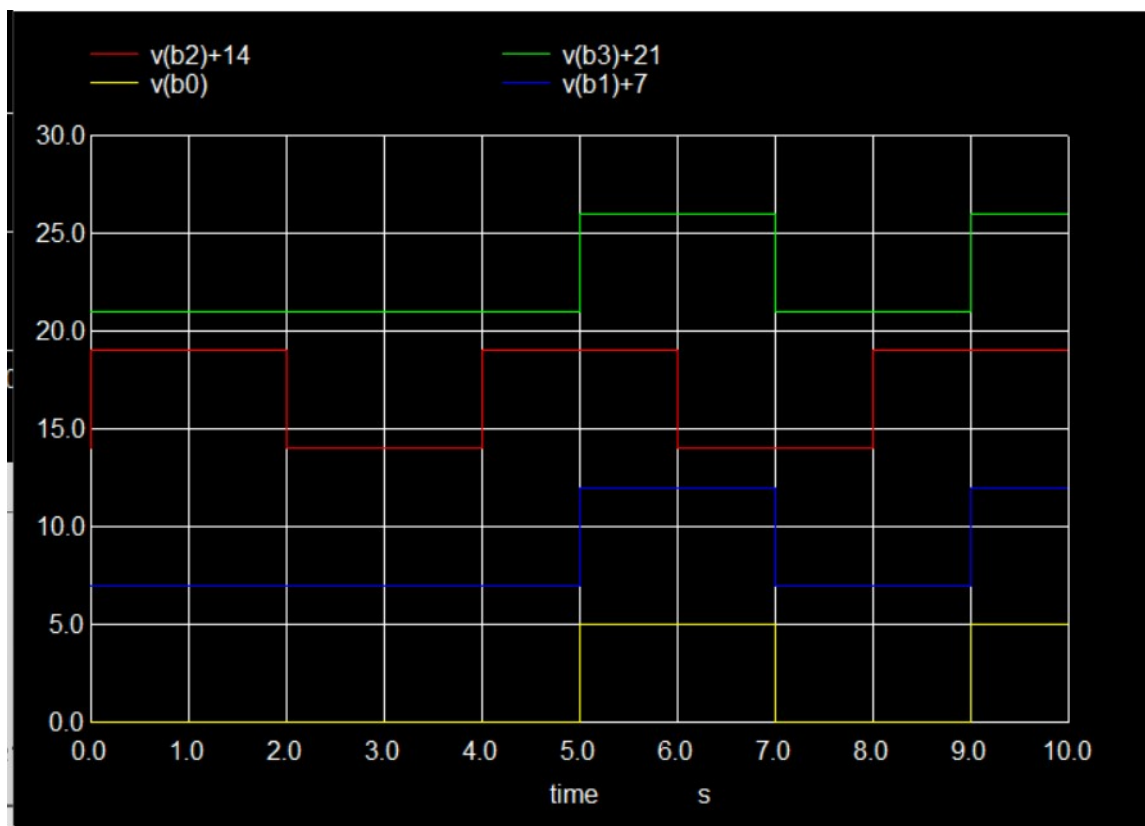


Figure 15.3: Input Waveform B

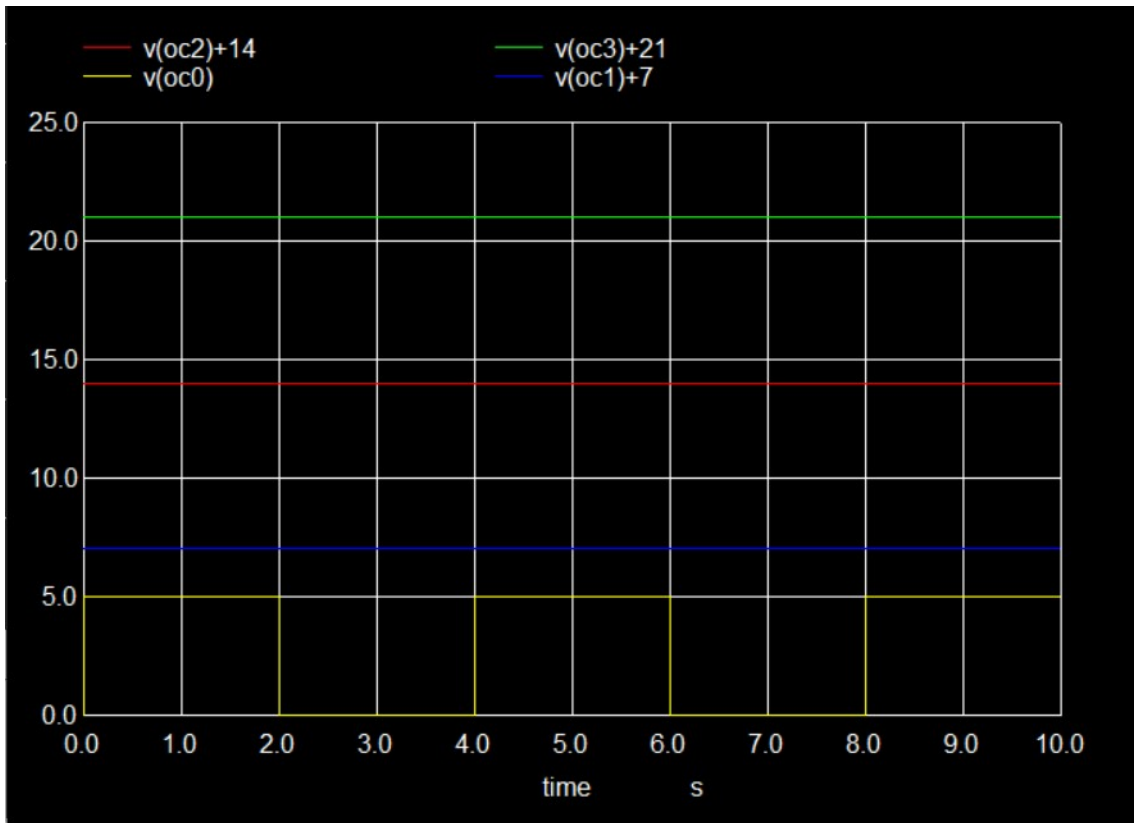


Figure 15.4: Waveform OpCode

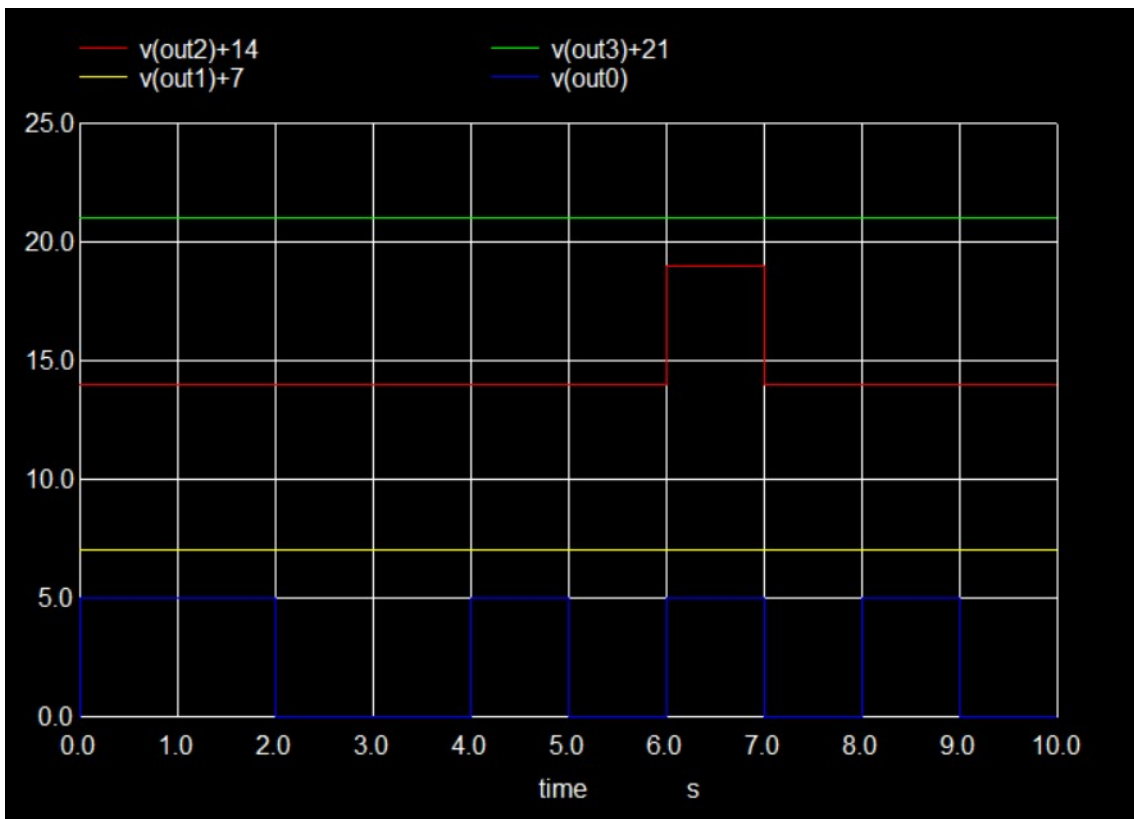


Figure 15.5: Output Waveform

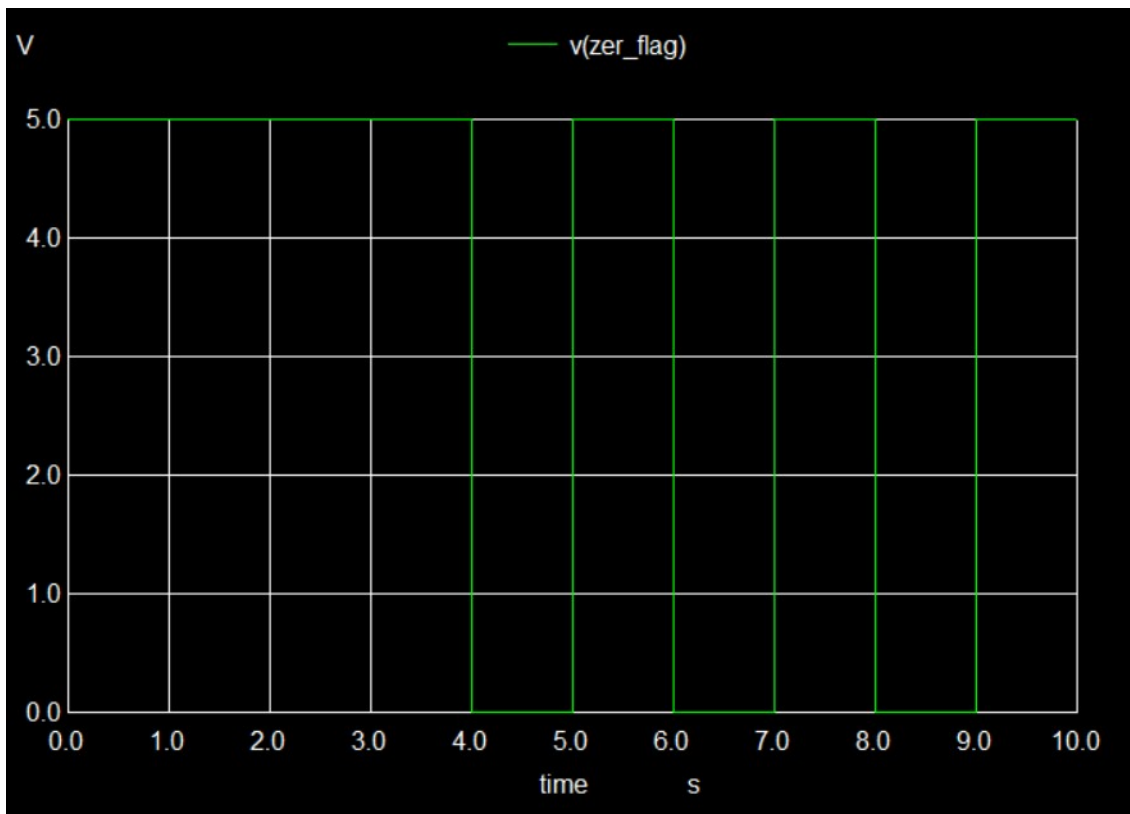


Figure 15.6: ZeroFlag Waveform

```
ngspice -p E:\FOSSEE\Digital\TEST_ALU\TEST_ALU.cir.out

Inside foo after eval.....
A=5
B=4
opcode=1
out=1
zero=0
=====alu_4bit : New Iteration=====
Instance : 0

Inside foo before eval.....
A=5
B=4
opcode=1
out=1
zero=0

Inside foo after eval.....
A=15
B=15
opcode=1
out=0
zero=1
=====
```

Figure 15.7: NgSpice Output

# Chapter 16

## IN74HCT21 Dual 4 Input AND Gate

The IN74HCT21 is high-speed Si-gate CMOS device and is pin compatible with low power Schottky TTL (LSTTL) . The device provide the Dual 4-input AND function.

### 16.0.1 Pin Diagram

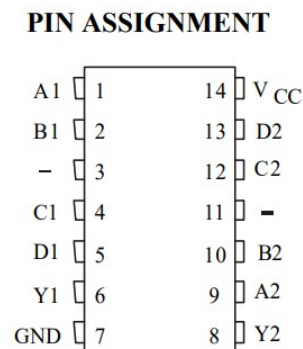


Figure 16.1: Pin Diagram of 74HCT21

### 16.0.2 Schematic

The processor model is simulated in Makerchip-NgVeri and the model is created. The model is imported to the schematic and adc and dac are used to convert the analog and digital signals.

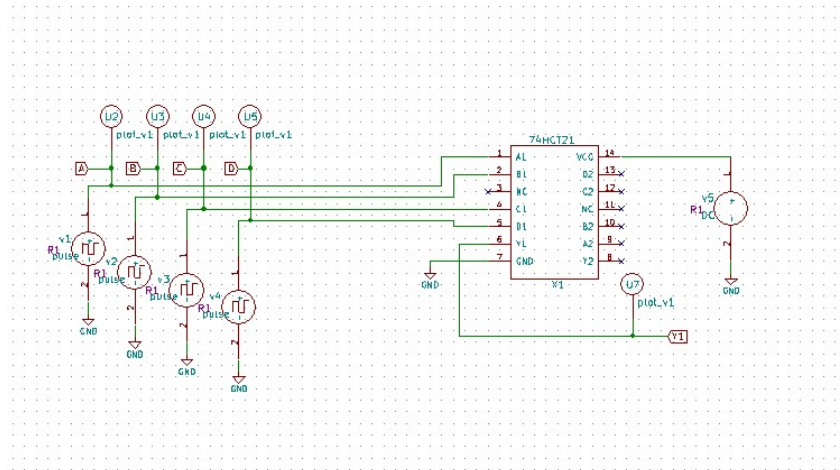


Figure 16.2: Test Circuit for 74HCT21

### 16.0.3 Simulation

Input and Output Waveforms are given below

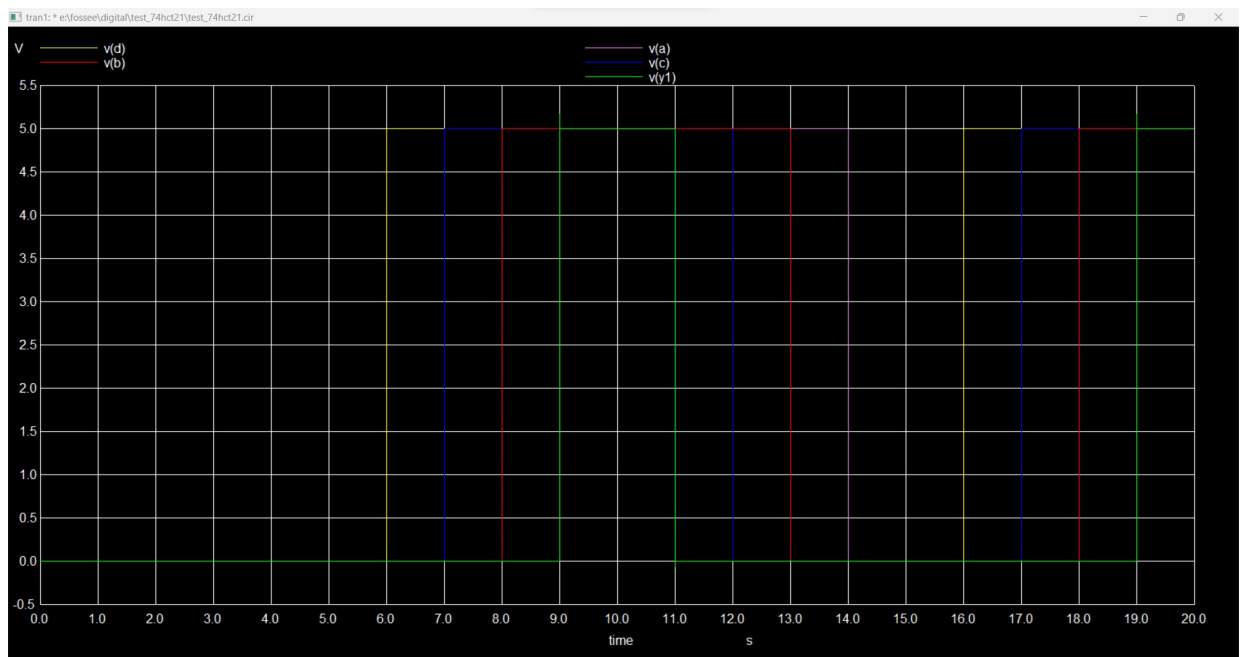


Figure 16.3: Input and Output Waveform



# Chapter 17

## Circuits Contribution

### 17.0.1 Roshan Binu Paul

1. Kogge Stone Adder
2. UART TX
3. 4\*4 Array Multiplier
4. RISC V Processor
5. Clock Divider

### 17.0.2 Bhargav Dhoke

1. 8-Bit MIPS Processor
2. 32-Bit RISC V Processor
3. 8bit-Microcomputer
4. 16bit-Sklansky-Adder
5. GCD Calculator
6. Booth Multiplier
7. Johnson Ring Counter
8. Mux 8:1
9. Multiplication by repeated addition
10. Demux 1:8

### 17.0.3 Abhinav Tripathi

1. 2:1 Multiplexer
2. 4-Bit ALU
3. IN74HCT21 Dual 4 Input AND gates
4. 16 Bit Processor

# Bibliography

- [1] FOSSEE Official Website. 2020.  
URL: <https://fossee.in/about>
  
- [2] RISC V Single Cycle Core.  
URL: [https://github.com/merldsu/RISCV\\_Single\\_Cycle\\_Core](https://github.com/merldsu/RISCV_Single_Cycle_Core)
  
- [3] Wikipedia Official Website.  
URL: [https://en.wikipedia.org/wiki/Universal\\_asynchronous\\_receiver-transmitter](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter)
  
- [4] UART.  
URL: [https://github.com/ben-marshall/uart/blob/master/rtl/uart\\_tx.v](https://github.com/ben-marshall/uart/blob/master/rtl/uart_tx.v)
  
- [5] Wikipedia Official Website.  
URL: [https://en.wikipedia.org/wiki/Kogge%E2%80%93Stone\\_adder](https://en.wikipedia.org/wiki/Kogge%E2%80%93Stone_adder)
  
- [6] Kogge Stone Adder.  
URL: <https://github.com/mongrelgem/Verilog-Adders/tree/master/Kogge-Stone%20Adder>
  
- [7] Geeksforgeeks Official Website.  
URL: <https://www.geeksforgeeks.org/array-multiplier-in-digital-logic/>
  
- [8] Array Multiplier.  
URL: <https://github.com/sudhamshu091/32-Verilog-Mini-Projects/blob/main/Array%20Multiplier/>

array\_multiplier.v

- [9] Wikipedia Official Website.[https://en.wikipedia.org/wiki/Frequency\\_divider](https://en.wikipedia.org/wiki/Frequency_divider)
  
- [10] Clock Divider.  
URL:<https://github.com/snbk001/Verilog-Design-Examples/blob/main/Clock%20Divider/clkdivider.v>
  
- [11] 16 Bit Processor.  
URL:<https://github.com/joe-legg/Calcu-16/tree/master>
  
- [12] eSim Official website. 2020.  
URL: <https://esim.fossee.in/>
  
- [13] 8-bit-MIPS-Processor  
URL:<https://github.com/GabrielGiurgica/8-bit-MIPS-Processor/tree/main>
  
- [14] 32bit-RISC-V  
URL:<https://github.com/ash-olakangal/RISC-V-Processor/tree/main>
  
- [15] 8bit-Microcomputer  
URL:[https://github.com/TheSUPERCD/8bit\\_MicroComputer\\_Verilog/tree/master](https://github.com/TheSUPERCD/8bit_MicroComputer_Verilog/tree/master)
  
- [16] Icarus-Verilog  
URL:<https://sourceforge.net/projects/iverilog/>
  
- [17] How to install iverilog and run  
URL:[https://iverilog.fandom.com/wiki/Installation\\_Guide](https://iverilog.fandom.com/wiki/Installation_Guide)