Winter Fellowship Report

On

Implementing processors in eSim

Submitted by

Subhradip Chakraborty

Under the guidance of

Prof.Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

April 12, 2023

Acknowledgment

I would like to express my gratitude to the FOSSEE team for providing me with an excellent opportunity. This internship has provided a deep insight into the technology that industry is focused on and also it gave me opportunity to promote and help developing an opensource community.My heartfelt thanks go to Prof. Kannan M. Moudgalya for his invaluable and constructive input during the planning and execution of our research work.
I want to extend my gratitude to the eSim team for providing me with all the necessary resources and guidance throughout the project. Our mentors Mr. Sumanto Kar, Mr. Rahul Paknikar, and Mrs. Usha Vishwanathan, as well as the entire team helped and supported me throught the internship period, which will undoubtedly benefit me in the near future. Their dedication and hard work made my fellowship experience a joyous one. I would like to give a special mention to Mr. Sumanto Kar for his willingness to share his knowledge and assist me whenever I was stuck in executing the problem statement.
I plan upon using the knowledge and experiences gained here to enhance my career growth and contribute to the betterment of society. The wonderful experiences I gained here during the as fresher trying to get into the VLSI industry will be cherished for a lifetime.

Contents

Chapter 1

Introduction

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]

1.1 eSim

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.

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 eSim's 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 offers an interface.

1.3 Makerchip

Makerchip is an online browser based IDE which is developed on verilator which is used to perform simulation of Verilog/ System Verilog/ TL-Verilog based files. Verilator is a tool which converts the Verilog files to C++ object. Before converting a verilog file in ngveri in eSim first the design is simulated here with random inputs

to check whether the design is producing consistent and desired results. If the design is correctly simulated we proceed with our mixed signal design.

1.4 EDA Playground

EDA Playground is an online browser based emulator which is used to edit, simulate and synthesize SystemVerilog, Verilog, VHDL and other HDLs. I have used the EDA Playground as it is free of cost and effective in writing testbenches for the complex processor design that we are going to implement on eSim. The testbench execution capability is also used to verify some of the open source processor testbenches which were used to find the functioning of the processor.

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 by file was simulated in makerchip 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

Basic-SIMD-Processor

It is a simple SIMD processor which is designed in verilog and has a 16-bit SIMD ALU. The ALU operation takes two clock cycles, the first cycle is used to load values into the registers and the second is used to perform the operations. The instruction used by this processor is 18 bit. This is an open source processor available on github (link: https://github.com/zslwyuan/Basic-SIMD-Processor-Verilog-Tutorial). This project was chosen because this processor is used for layout design and post layout simulations were performed.

3.1 Block Diagram

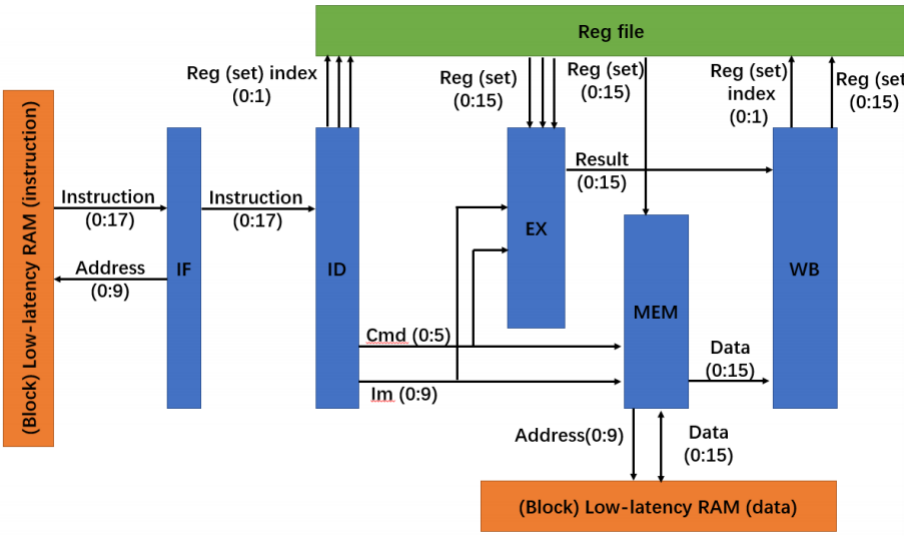

Figure 3.1: Block Diagram Implementation of Processor [3]

In a simple processor without a pipeline the processor is divided into 5 stages. The five stages are Information Fetch (IF), Information Decode (ID), Execute (EX), Memory (MEM) and Write Back (WB). A 10 bit address is used to fetch the instruction set from the instruction memory and this operation is performed in the IF stage. In the next stage the instruction is decoded and some of the control registers

will be set. In the EX stage mainly the ALU operations are performed or some control signals like jump are implemented. If the information contains any store or load then the MEM stage is used to perform those operations. The WB stage is the last stage where the data processed is written back to the registers.

3.2 Simulation of SIMD ALU

3.2.1 SIMD Adder

The SIMD adder is first simulated in makerchip and then implemented in eSim. It is based on 4 4 bit adders which supports subtraction also. There are three control signals H,O and Q which are used to control the different width of the input signal. This control supports the forwarding of the carry bit.

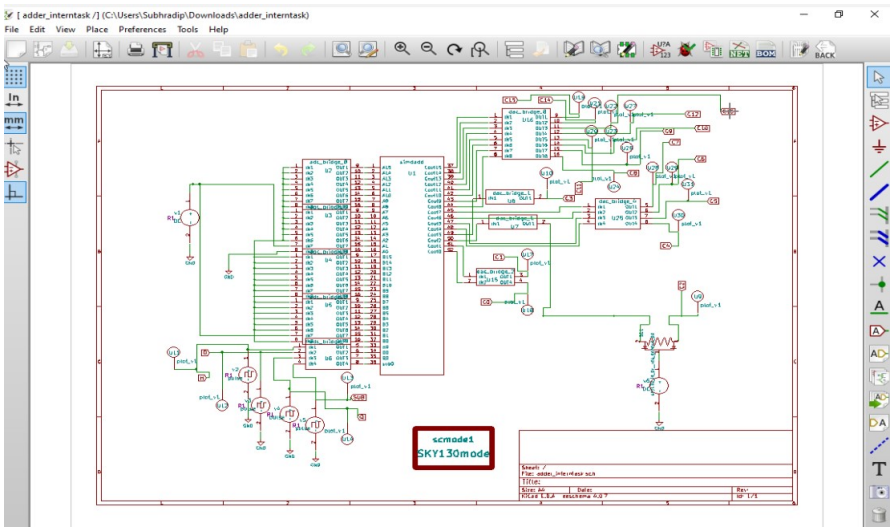

Figure 3.2: Implemented the Adder circuit in eSim

Figure 3.3: Simulation of Adder circuit in eSim

3.2.2 SIMD Shifter

The SIMD shifter is based on two 16 bit shifters which support different widths. In this block the shifter uses two control signals H and O which are used to determine whether the MSB of the 4 bit block should take 0 or the value from the LSB in front of it. Similarly like the adder circuit we simulate the shifter also.

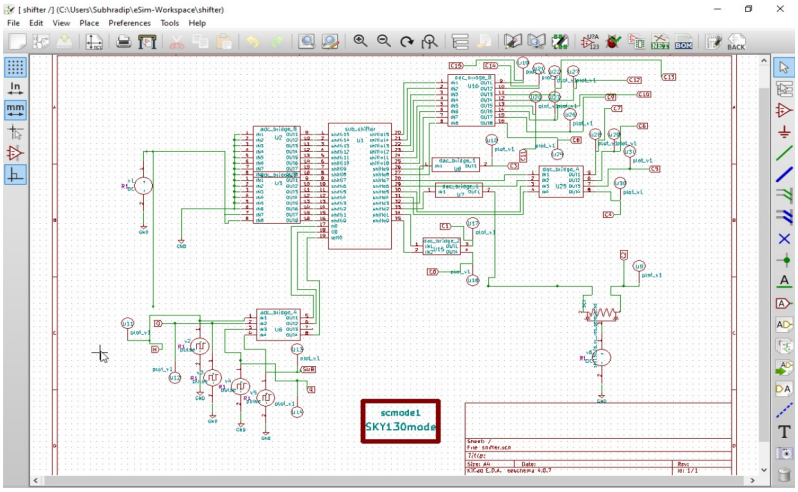

Figure 3.4: Implemented the Shifter circuit in eSim

3.2.3 SIMD Multiply

The SIMD multiplier is based on the SIMD adder and SIMD shifter. Here also we have three control signals H,O,Q and are used to control the input width. It is implemented as a typical multiplier circuit where the adder is utilized and the least significant 16 bits of the sum is the output of multiplication. Similarly like the adder circuit we simulate the multiplier also.

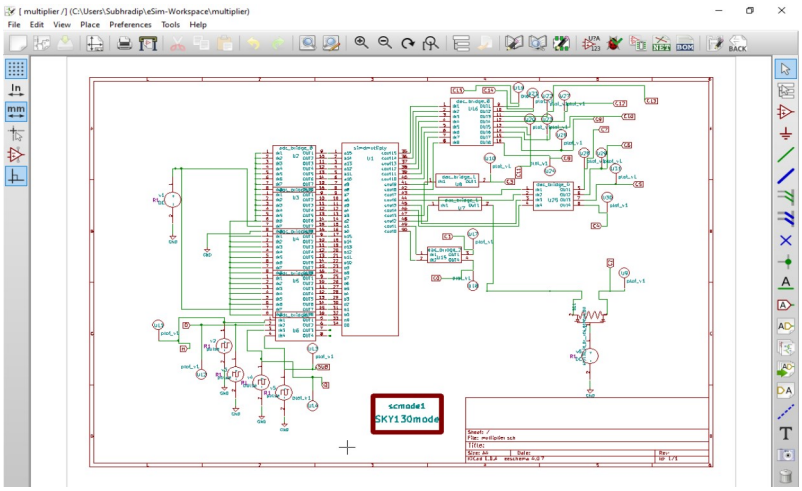

Figure 3.5: Implemented the Multiplier circuit in eSim

3.3 Implmentation of the processor

The cputop file is has to be simulated for the processor perform the operations. Our objective is to implement it on eSim, so we cannot directly use the top file, we have to design a code which provides the instruction set to the top file just like a test-bench. The first approach which I used to implement this technique was to design a connector block having the input ports as values and instruction set and output is connected to the top file manually in Kicad. The problem with this approach is that because of the two separate blocks there are timing issues which interfered with the desired output. The second approach that i used is that to call the top file inside this conncector file and add those files as dependencies during conversion. This approach worked and gave us the correct output.

The authors of the opensource processor has provided us with the sample instuction sets those are:

```
INST_MEM[0]  = 18'b100110_00_0000000000;  // load MEM0 into H0
 ↪  load16bit R0 im=0000000000
INST_MEM[1]  = 18'b100110_01_0000000001;  // load MEM1 into H1
 ↪  load16bit R1 im=0000000001
INST_MEM[2]  = 18'b100110_10_0000000010;  // load MEM2 into H2
 ↪  load16bit R2 im=0000000010
INST_MEM[3]  = 18'b000000_00000000_00_01; // add H1 to H0
 ↪  add16bit R0=0000000000000101 R1=0000000000001111
INST_MEM[4]  = 18'b000000_00000000_10_00; // add H0 to H2
 ↪  add16bit R2=0000000000000100 R0=0000000000010100
INST_MEM[5]  = 18'b101001_00_0000000000;  // store H0 back into MEM0
 ↪  store16bit R0=0000000000010100 im=0000000000
INST_MEM[6]  = 18'b101100_01_0100100010;  // set H1 to 0100100010
 ↪  set16bit R1 im=0100100010
INST_MEM[7]  = 18'b000000_00000000_01_10; // add H2 to H1
 ↪  add16bit R1=0000000100100010 R2=0000000000011000
INST_MEM[8]  = 18'b101001_01_0000000011;  // store H1 back into MEM3
 ↪  store16bit R1=0000000100111010 im=0000000011
INST_MEM[9]  = 18'b101001_10_0000000100;  // store H2 back into MEM4
 ↪  store16bit R2=0000000000011000 im=0000000100
INST_MEM[10] = 18'b101001_00_0000000101;  // store H0 back into MEM5
 ↪  store16bit R0=0000000000010100 im=0000000101

INST_MEM[11]  = 18'b100111_01_0000000010; // load MEM2 into O1
 ↪  load8bit R1 im=0000000010
INST_MEM[12]  = 18'b100111_10_0000000011; // load MEM3 into O2
 ↪  load8bit R2 im=0000000011
INST_MEM[13]  = 18'b100111_00_0000000100; // load MEM4 into O0
 ↪  load8bit R0 im=0000000100
```

```
INST_MEM[14]  = 18'b101101_01_0001011010; // set each register in O1    set8bit
 ↪  R1 im=0001011010

INST_MEM[15]  = 18'b100101_00_0000000010; // setloop = 2               setloop
 ↪  im=   2

INST_MEM[16]  = 18'b000001_00000000_00_01; // add O0 to O1             add8bit
 ↪  R0=0000000000011000 R1=0101101001011010
INST_MEM[17]  = 18'b000001_00000000_10_00; // add O2 to O0             add8bit
 ↪  R2=0000000100111010 R0=0101101001110010
INST_MEM[18]  = 18'b000111_00000000_10_01; // sub O1 from O2           sub8bit
 ↪  R2=0101101110101100 R1=0101101001011010
INST_MEM[19]  = 18'b001101_00000000_10_01; // mul O2 with O1           mul8bit
 ↪  R2=0000000101010010 R1=0101101001011010
INST_MEM[20]  = 18'b101010_01_0000000101; // store O1 MEM5 into
 ↪  store8bit R1=0101101001011010 im=0000000101
INST_MEM[21]  = 18'b101010_10_0000000110; // store O2 MEM6 into O2
 ↪  store8bit R2=0101101011010100 im=0000000110
INST_MEM[22]  = 18'b101010_00_0000000111; // store O0 MEM7 into O0
 ↪  store8bit R0=0101101001110010 im=0000000111

INST_MEM[23]  = 18'b100110_01_0000000101; // load MEM5 into H1
 ↪  load16bit R1 im=0000000101
INST_MEM[24]  = 18'b100110_10_0000000111; // load MEM7 into H2
 ↪  load16bit R2 im=0000000111
INST_MEM[25]  = 18'b001100_00000000_10_01; // mul H2 with H1
 ↪  mul16bit R2=0101101001110010 R1=0101101001011010
INST_MEM[26]  = 18'b000110_00000000_10_01; // sub H1 from H2
 ↪  sub16bit R2=1110000000010100 R1=0101101001011010
INST_MEM[27]  = 18'b011000_0000000000_01; // shift right H1
 ↪  Rshift16bit R1=0101101001011010
INST_MEM[28]  = 18'b010101_0000000000_10; // shift left H2
 ↪  Lshift16bit R2=1000010110111010
INST_MEM[29]  = 18'b000011_10_0000001110; // add im_number into H2
 ↪  add16bit R2=0000101101110100 im=0000001110
INST_MEM[30]  = 18'b001001_10_0000001110; // sub im_number from H2
 ↪  sub16bit R2=0000101110000010 im=0000001110
INST_MEM[31]  = 18'b100001_0000000000_10; // not H2
 ↪  not16bit R2=0000101101110100
INST_MEM[32]  = 18'b011011_00000000_01_00; // H1 and H0
 ↪  and16bit R1=0010110100101101 R0=0000000000010100
INST_MEM[33]  = 18'b011110_00000000_10_01;  // H2 or H1                or16bit
 ↪  R2=1111010010001011 R1=0000000000000100
```

```
INST_MEM[34]  = 18'b101100_00_0000001111;  // set H0 to 0000001111
 ↪  set16bit R0 im=0000001111
INST_MEM[35]  = 18'b101100_01_0000000100;  // set H1 to 0000000100
 ↪  set16bit R1 im=0000000100
INST_MEM[36]  = 18'b101100_10_0000000010;  // set H2 to 0000000010
 ↪  set16bit R2 im=0000000010
INST_MEM[37]  = 18'b010010_000000_00_01_10;  // H0+H1*H2
 ↪  MAC16bit R0=0000000000001111 R1=0000000000000100 R2=0000000000000010
INST_MEM[38]  = 18'b001001_00_0000001000;  // sub im_number from H0
 ↪  sub16bit R0=0000000000010111 im=0000001000
INST_MEM[39]  = 18'b001111_01_0000001101;  // mul H1 with im_number
 ↪  mul16bit R1=0000000000000100 im=0000001101
INST_MEM[40]  = 18'b101001_01_0000000111;  // store H1 back into MEM7
 ↪  store16bit R1=0000000000110100 im=0000000111
INST_MEM[41]  = 18'b101001_10_0000000100;  // store H2 back into MEM4
 ↪  store16bit R2=0000000000000010 im=0000000100
INST_MEM[42]  = 18'b101001_00_0000001000;  // store H0 back into MEM8
 ↪  store16bit R0=0000000000001111 im=0000001000

INST_MEM[43]  = 18'b100111_01_0000000110; // load MEM6 into O1
 ↪  load8bit R1 im=0000000110
INST_MEM[44]  = 18'b100111_10_0000000111; // load MEM7 into O2
 ↪  load8bit R2 im=0000000111
INST_MEM[45]  = 18'b100111_00_0000001000; // load MEM8 into O0
 ↪  load8bit R0 im=0000001000
INST_MEM[46]  = 18'b011001_0000000000_01; // shift right O1
 ↪  Rshift8bit R1=0101101011010100
INST_MEM[47]  = 18'b010110_0000000000_10; // shift left O2
 ↪  Lshift8bit R2=0000000000110100
INST_MEM[48]  = 18'b000100_10_0000001110; // add im_number into O2      add8bit
 ↪  R2=0000000001101000 im=0000001110
INST_MEM[49]  = 18'b001010_10_0000001110; // sub im_number from O2      sub8bit
 ↪  R2=0000111001110110 im=0000001110
INST_MEM[50]  = 18'b100010_0000000000_10; // not O2                     not8bit
 ↪  R2=0000000001101000
INST_MEM[51]  = 18'b011100_00000000_01_00; // O1 and O0                 and8bit
 ↪  R1=0010110101101010 R0=0000000000001111
INST_MEM[52]  = 18'b011111_00000000_10_01;  // O2 or O1                 or8bit
 ↪  R2=1111111110010111 R1=0000000000001010
INST_MEM[53]  = 18'b101101_00_0000001111;  // set O0 to 0000001111      set8bit
 ↪  R0 im=0000001111
INST_MEM[54]  = 18'b101101_01_0000000100;  // set O1 to 0000000100      set8bit
 ↪  R1 im=0000000100
```

```
INST_MEM[55]  = 18'b101101_10_0000000010;  // set O2 to 0000000010      set8bit
↪  R2 im=0000000010
INST_MEM[56]  = 18'b010011_000000_00_01_10;  // O0+O1*O2                 MAC8bit
↪  R0=0000111100001111 R1=0000010000000100 R2=0000001000000010
INST_MEM[57]  = 18'b001010_00_0000001000;  // sub im_number from O0      sub8bit
↪  R0=0001011100010111 im=0000001000
INST_MEM[58]  = 18'b010000_01_0000001101;  // mul O1 with im_number      mul8bit
↪  R1=0000010000000100 im=0000001101
INST_MEM[59]  = 18'b101010_01_0000001001; // store O1 into MEM9
↪  store8bit R1=0011010000110100 im=0000001001
INST_MEM[60]  = 18'b101010_10_0000000110; // store O2 into MEM6
↪  store8bit R2=0000001000000010 im=0000000110
INST_MEM[61]  = 18'b101010_00_0000000111; // store O0 into MEM7
↪  store8bit R0=0000111100001111 im=0000000111


INST_MEM[62]  = 18'b101000_01_0000001001; // load MEM9 into Q1
↪  load4bit R1 im=0000001001
INST_MEM[63]  = 18'b101000_10_0000000110; // load MEM6 into Q2
↪  load4bit R2 im=0000000110
INST_MEM[64]  = 18'b101000_00_0000000111; // load MEM7 into Q0
↪  load4bit R0 im=0000000111
INST_MEM[65]  = 18'b101110_10_0001011010; // set each register in Q2    set4bit
↪  R2 im=0001011010
INST_MEM[66]  = 18'b011010_0000000000_01; // shift right Q1
↪  Rshift4bit R1=0011010000110100
INST_MEM[67]  = 18'b010111_0000000000_10; // shift left Q2
↪  Lshift4bit R2=1010101010101010
INST_MEM[68]  = 18'b000101_10_0000001110; // add im_number into Q2      add4bit
↪  R2=0100010001000100 im=0000001110
INST_MEM[69]  = 18'b001011_10_0000001110; // sub im_number from Q2      sub4bit
↪  R2=0010001000100010 im=0000001110
INST_MEM[70]  = 18'b100011_0000000000_10; // not Q2                     not4bit
↪  R2=0100010001000100
INST_MEM[71]  = 18'b011101_00000000_01_00; // Q1 and Q0                 and4bit
↪  R1=0001001000010010 R0=0000111100001111
INST_MEM[72]  = 18'b100000_00000000_10_01;  // Q2 or Q1                 or4bit
↪  R2=1011101110111011 R1=0000001000000010
INST_MEM[73]  = 18'b101110_00_0000001111;  // set Q0 to 0000001111      set4bit
↪  R0 im=0000001111
INST_MEM[74]  = 18'b101110_01_0000000100;  // set Q1 to 0000000100      set4bit
↪  R1 im=0000000100
INST_MEM[75]  = 18'b101110_10_0000000010;  // set Q2 to 0000000010      set4bit
↪  R2 im=0000000010
```

```
INST_MEM[76]  = 18'b010100_000000_00_01_10;  // Q0+Q1*Q2                MAC4bit
 ↪  R0=1111111111111111 R1=0100010001000100 R2=0010001000100010
INST_MEM[77]  = 18'b001011_00_0000001000;  // sub im_number from Q0     sub4bit
 ↪  R0=0111011101110111 im=0000001000
INST_MEM[78]  = 18'b010001_01_0000000101;  // mul Q1 with im_number     mul4bit
 ↪  R1=0100010001000100 im=0000000101
INST_MEM[79]  = 18'b101110_01_0001011010; // set each register in Q1    set4bit
 ↪  R1 im=0001011010
INST_MEM[80]  = 18'b000010_00000000_00_01; // add Q0 to Q1              add4bit
 ↪  R0=1111111111111111 R1=1010101010101010
INST_MEM[81]  = 18'b000010_00000000_10_00; // add Q2 to Q0              add4bit
 ↪  R2=0010001000100010 R0=1001100110011001
INST_MEM[82]  = 18'b001000_00000000_10_01; // sub Q1 from Q2            sub4bit
 ↪  R2=1011101110111011 R1=1010101010101010
INST_MEM[83]  = 18'b001110_00000000_10_01; // mul Q2 with Q1            mul4bit
 ↪  R2=0001000100010001 R1=1010101010101010
INST_MEM[84]  = 18'b101011_01_0000001001; // store Q1 into MEM9
 ↪  store4bit R1=1010101010101010 im=0000001001
INST_MEM[85]  = 18'b101011_10_0000000110; // store Q2 into MEM6
 ↪  store4bit R2=1010101010101010 im=0000000110
INST_MEM[86]  = 18'b101011_00_0000000111; // store Q0 into MEM7
 ↪  store4bit R0=1001100110011001 im=0000000111

INST_MEM[87]  = 18'b100100_00_0000010000; // jump to 16
 ↪  loopjump LC=   x im=  16

INST_MEM[88]  = 18'b111111_000000000000; // halt

DATA_MEM[0] = 5;
DATA_MEM[1] = 15;
DATA_MEM[2] = 4;
```

3.3.1 Simulation in Makerchip

The code that i used to connect the top file of the processor is written below. Here only few of the instruction sets were taken to check the working of the processor whether it provided us a correct output. In this instrction set choses it add two numbers a and b and store it in register which is indicated by the makerchip waveform given below.

```
module fossee_proc1(
input clk,
input rst,
input [15:0] a,
input [15:0] b,
input [15:0] c,
input [15:0] inst_in,
output  reg done,
output reg [17:0] instruction_in,
output reg [15:0] data_in,
output reg [15:0] data_out,
output reg [9:0] instruction_addr,
output reg [9:0] data_addr,
output reg data_R,
output reg data_W
);

reg [17:0] INST_MEM[1023:0];
reg [15:0] DATA_MEM[1023:0];
//wire done;
//reg [17:0] instruction_in;
//reg [15:0] data_in;
//wire [15:0] data_out;
//wire [9:0] instruction_addr;
//wire [9:0] data_addr;
//wire data_R;
//wire data_W;

CPUtop uut (
.clk        (clk),          // system clock
.rst        (rst),          // system reset (active high)
.instruction_in (instruction_in),
.data_in(data_in),
.data_out(data_out),
.instruction_addr(instruction_addr),
.data_addr(data_addr),
.data_R(data_R),
.data_W(data_W),
```

```
.done(done)
);

initial begin
INST_MEM[0]  = 18'b100110_00_0000000000;  // load MEM0 into H0          load16bit
 ↪  R0 im=0000000000
INST_MEM[1]  = 18'b100110_01_0000000001;  // load MEM1 into H1          load16bit
 ↪  R1 im=0000000001
INST_MEM[2]  = 18'b100110_10_0000000010;  // load MEM2 into H2          load16bit
 ↪  R2 im=0000000010
//INST_MEM[3]  = 18'b000000_00000000_00_01; // add H1 to H0
 ↪  add16bit R0=0000000000000101 R1=0000000000001111
//INST_MEM[4]  = 18'b000000_00000000_10_00; // add H0 to H2
 ↪  add16bit R2=0000000000000100 R0=0000000000010100
INST_MEM[4]  = 18'b101001_00_0000000000;  // store H0 back into MEM0
 ↪  store16bit R0=0000000000010100 im=0000000000
//INST_MEM[6]  = 18'b101100_01_0100100010;  // set H1 to 0100100010
 ↪  set16bit R1 im=0100100010
//INST_MEM[7]  = 18'b000000_00000000_01_10; // add H2 to H1
 ↪  add16bit R1=0000000100100010 R2=0000000000011000
INST_MEM[5]  = 18'b101001_01_0000000011;  // store H1 back into MEM3
 ↪  store16bit R1=0000000100111010 im=0000000011
INST_MEM[6]  = 18'b101001_10_0000000100;  // store H2 back into MEM4
 ↪  store16bit R2=0000000000011000 im=0000000100
INST_MEM[7] = 18'b101001_00_0000000101;  // store H0 back into MEM5    store16bit
 ↪  R0=0000000000010100 im=0000000101

end

always @(negedge clk)
begin
    DATA_MEM[0] = a;
DATA_MEM[1] = b;
DATA_MEM[2] = c;
INST_MEM[3] = inst_in;
end

always @(negedge clk)
begin
if (data_R)
begin
    if (data_W)
begin
    DATA_MEM[data_addr] <= data_out;
    $display("write mem %d: %b",data_addr,data_out);
```

```
end
    else
        data_in <= DATA_MEM[data_addr];
end
end

always @(negedge clk)
begin
instruction_in <= INST_MEM[instruction_addr];
$display("inst_addr %d: %b",instruction_addr,INST_MEM[instruction_addr]);
end

endmodule
```

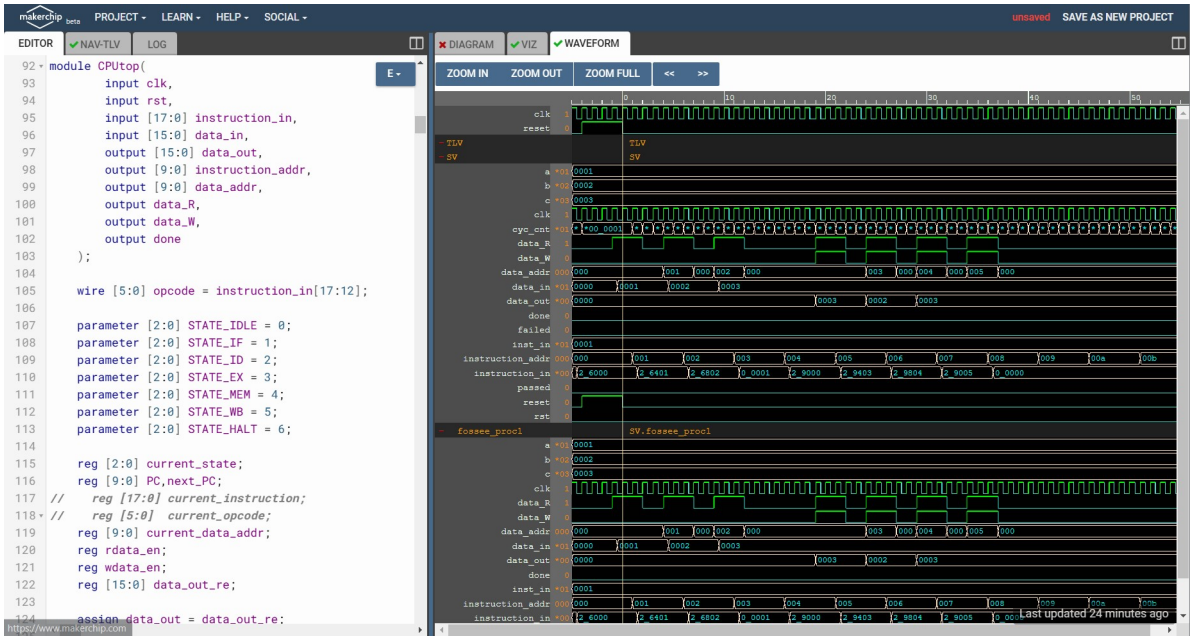

Figure 3.6: Simulation of the design in Makerchip

3.3.2 Simulation in eSim

Now we convert our verilog model to ngspice model which can be simulated. Given below shows that we have got our desired result in ngspice plot.

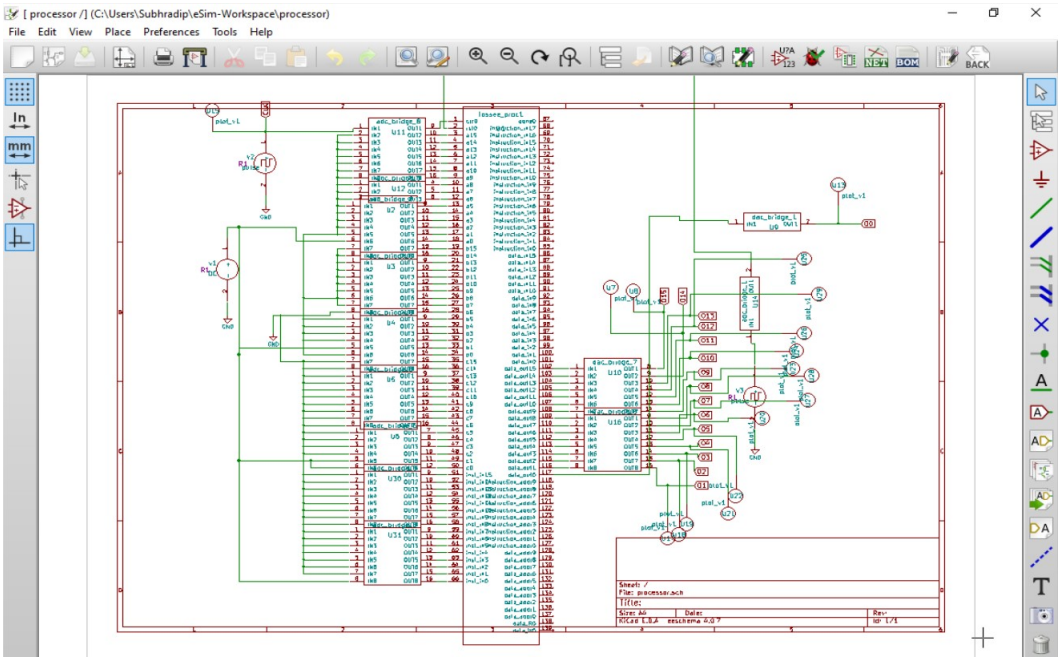

Figure 3.7: Circuit Design of Processor

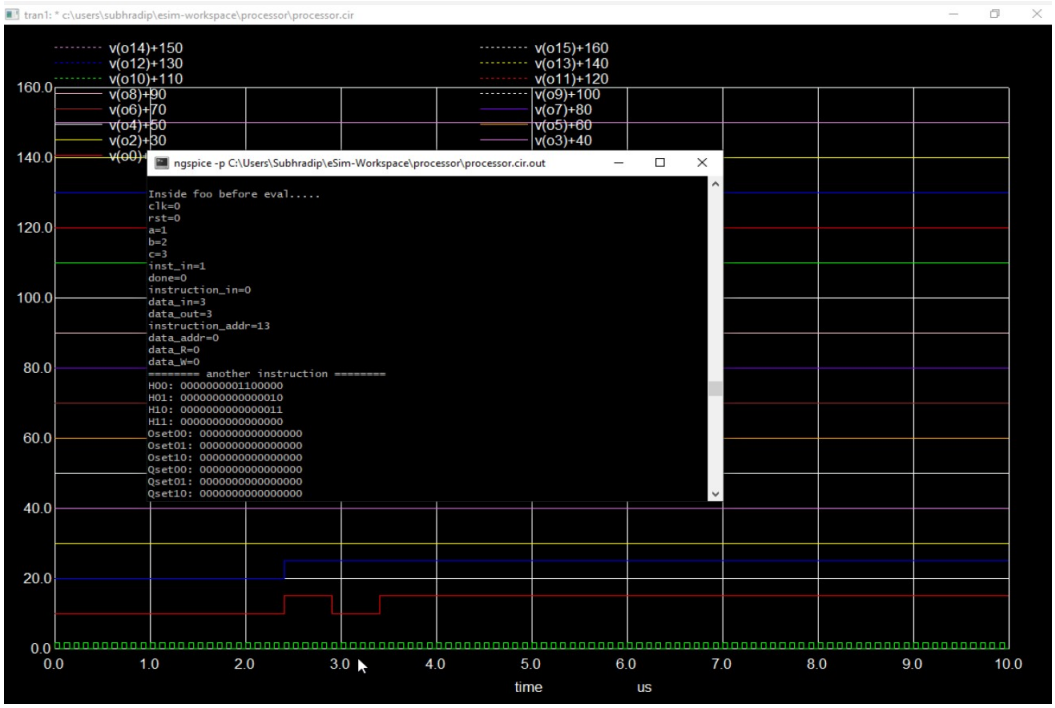

Figure 3.8: Simulation of the processor on eSim

Chapter 4

MIPS Processor

This open source processor was taken from opencores website and it is aimed at implementing a Microprocessor without Interlocked Pipe Stages (MIPS processor) with Reduced Instruction Set (RISC) architecture. The processor is coded in Verilog Hardware Description language with a 16 bit instruction set. This processor is a pipeline based processor with main 5 stages : the information fetch (IF), information decode (ID), execute (EX), memory (MEM) and write back (WB). The advantage of the pipeline is that it breaks a complex task into various stages that increases the speed and reduces the complexity. In the pipeline as soon as one instruction finishes processing in 1 stage the other data is sent to that stage simultaneously which increases the speed. But due to this there are various hazards which occur in the processor. The three main type of hazards are data hazard, control hazard and structural hazard. To avoid this hazard we add stall cycles.

4.1 Architecture

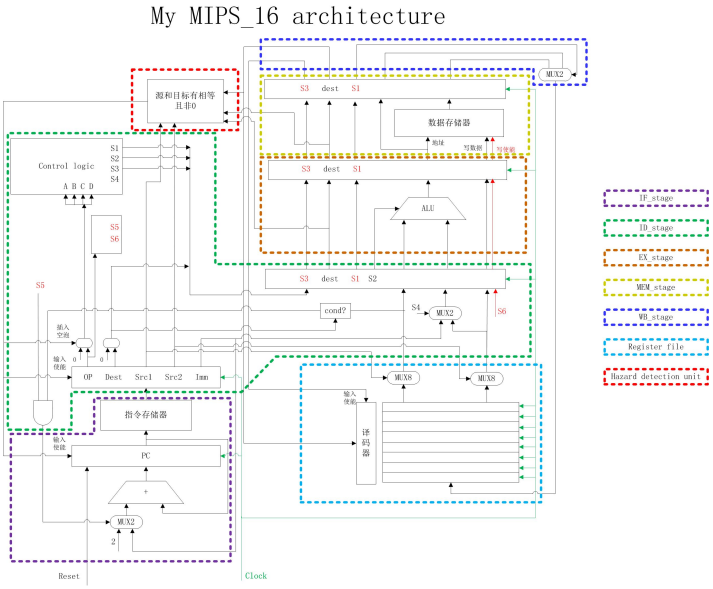

Figure 4.1: The architecture of the processor as mentioned by the authors [4]

4.2 Simulation of the processor

There are multiple verilog files in this project so we can simulate all of them separately and then combine. This could be one approach of proceeding forward but the this is time taking and not much useful in terms of proceeding forward. The approach i used was to simulate the ALU file separately as it performs all the arithmatic and logical operations so it is important to check that file. Then i simulated the test benches provided by the authors of the ALU operations to verify the makerchip and EDAplayground result. The whole processor is then simulated on EDAplayground and a testbench was written to validate the processor. Now some changes were done to simulate it esim and makerchip which you can easily find out once you go through the codes.

4.2.1 Simulation of ALU

First we simulate it in makerchip using custom values or using random values. Then we run the testbenches of ALU provided by the author to check all the operations at once.

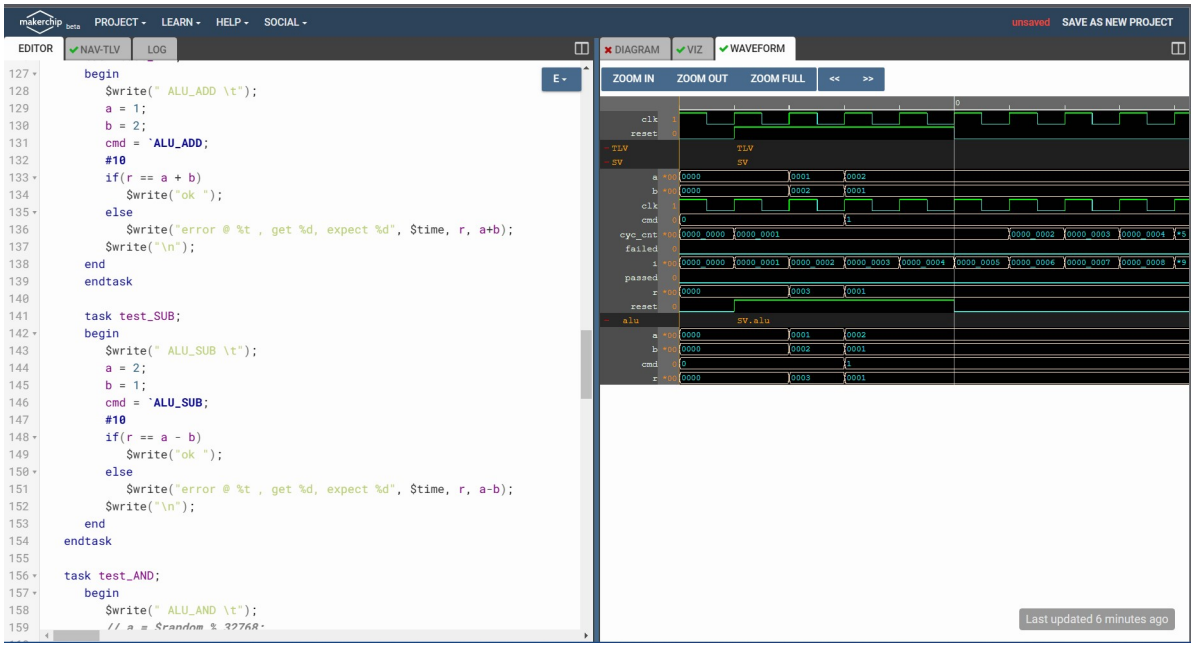

Figure 4.2: Simulation of ALU in makerchip

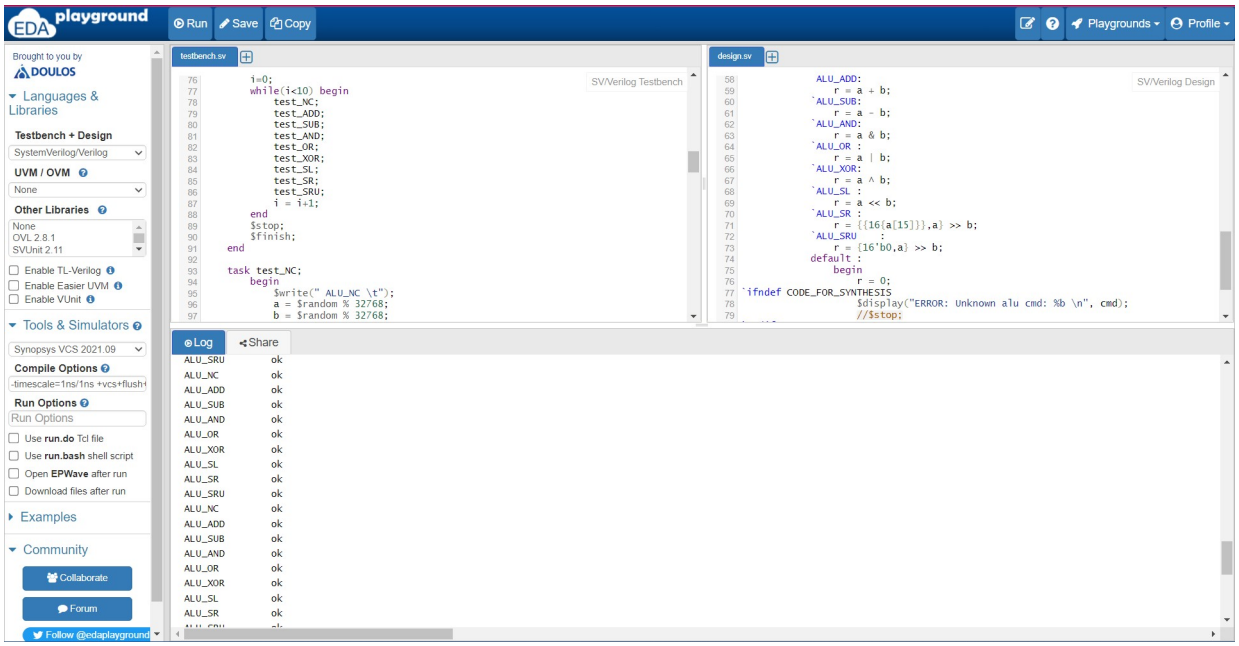

Figure 4.3: Simulation of ALU using testbench

4.2.2 Simulation of processor using testbench

```
module mips_16_core_top_tb_0_v;
reg clk;
reg rst;

reg [15:0] instruction [18:0];
integer j;

reg [15:0] output_res [7:0];

wire [`PC_WIDTH-1:0] pc;

parameter CLK_PERIOD = 10;
always #(CLK_PERIOD /2)
    clk =~clk;
integer i;
integer test;

mips_16_core_top uut (
    .clk(clk),
    .rst(rst),
    .pc(pc)
);

initial begin
```

```
    instruction[0] = 16'b1001001000001000;
    instruction[1] = 16'b1001010001001000;
    instruction[2] = 16'b1001011010001000;
    instruction[3] = 16'b0001100010011000;
    instruction[4] = 16'b1011100001000010;
    instruction[5] = 16'b1010101001000010;
    instruction[6] = 16'b0010110100101000;
    instruction[7] = 16'b1100000110111000;
    instruction[8] = 16'b1001111111000001;
    instruction[9] = 0;
    instruction[10] = 0;
    instruction[11] = 0;
    instruction[12] = 0;
    instruction[13] = 0;
    instruction[14] = 0;
    instruction[15] = 0;
    instruction[16] = 0;
    instruction[17] = 0;
    instruction[18] = 0;
end


initial begin
    clk = 0;
    rst = 0;
    #100;
    #(CLK_PERIOD/2)
    #1
    display_debug_message;
    test1;
    $stop;
end

task display_debug_message;
    begin
        $display("\n**************************");
        $display("mips_16 core test");
        $display("**************************\n");
    end
endtask

task sys_reset;
    begin
        rst = 0;
        #(CLK_PERIOD*1) rst = 1;
        #(CLK_PERIOD*1) rst = 0;
```

```
    end
endtask


task test1;
    begin
        for (j = 0 ; j< 19; j = j+1) begin
            uut.IF_stage_inst.imem.rom[j] = instruction[j];
            $display("instruction",j,"  ",instruction[j]);
            $display("rom",j,"  ",uut.IF_stage_inst.imem.rom[j]);
        end
        $display("rom load successfully\n");
        $display("running test1\n");
        sys_reset;
        #1

        #(CLK_PERIOD*100)
        display_all_regs;
        $display("ram[10] = %d", uut.MEM_stage_inst.dmem.ram[10]);
        sys_reset;

    end
endtask

task display_all_regs;
    begin
        $display("display_all_regs:");
        $display("-----------------------------");
        $display("R0\tR1\tR2\tR3\tR4\tR5\tR6\tR7");
        for(i=0; i<8; i=i+1) begin
        output_res[i] = uut.register_file_inst.reg_array[i];
        $display("The output R",i," :",output_res[i]);
        end
        $display("\n-----------------------------");
    end
endtask

endmodule
```

Figure 4.4: Simulation of processor using testbench

4.2.3 Simulation of processor in makerchip

In this section we change the code so that it can be simulated in verilator. This code is used in eSim to simulate the processor.

```
module connector(
input ck,
output reg [15:0] res,
output reg [3:0] resou
);

reg rs;
reg [15:0] instruct [18:0];
integer j = 0;
wire [7:0] pc;
integer CLPERIOD = 10;
integer i = 0;
integer tes;
integer tes_complete;
integer loa;
integer k = 0;
integer pri = 0;
reg [15:0] vah [7:0];

mips_16_core_top uut (
.clk(ck),
.rst(rs),
.pc(pc)
);

initial begin
$readmemb("test0.txt",instrut);
```

```
end

initial begin
tes_complete = 1;
end

always@(posedge ck) begin
if (j<19) begin
uut.IF_stage_inst.imem.rom[j] = instruct[j];
j= j+ 1;
end
else
    loa = 1;
end


always@(posedge ck) begin
if ((tes_complete == 1) && (loa == 1)) begin
if (i<=80) begin
vah[i] = uut.check_inst.re_array[i];
i = i + 1;
end
else
pri = 1;
end
end

always@(posedge ck) begin
if ((pri == 1) && (k <8)) begin
    res = vah[k];
    resou = k;
    k = k + 1;
end
end

endmodule
```

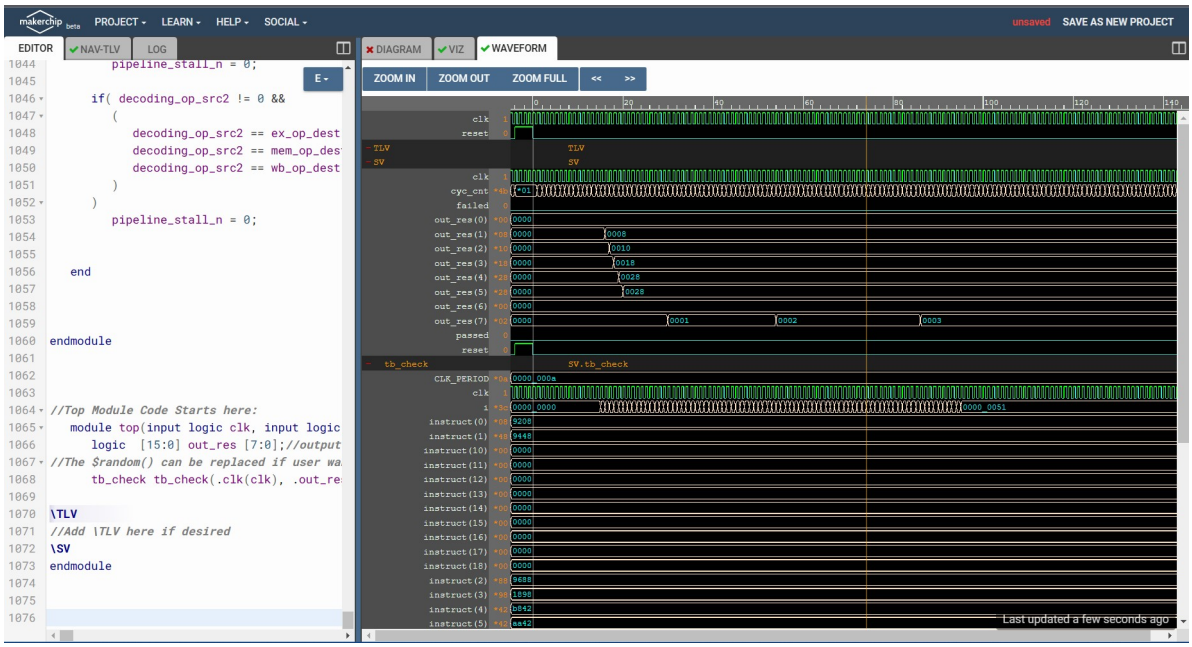

Figure 4.5: Simulation of processor in makerchip

4.2.4 Simulation of processor in eSim

I simulated the processor in eSim and got the desired register values which matched with that of the makerchip as well as the testbench Implemented.

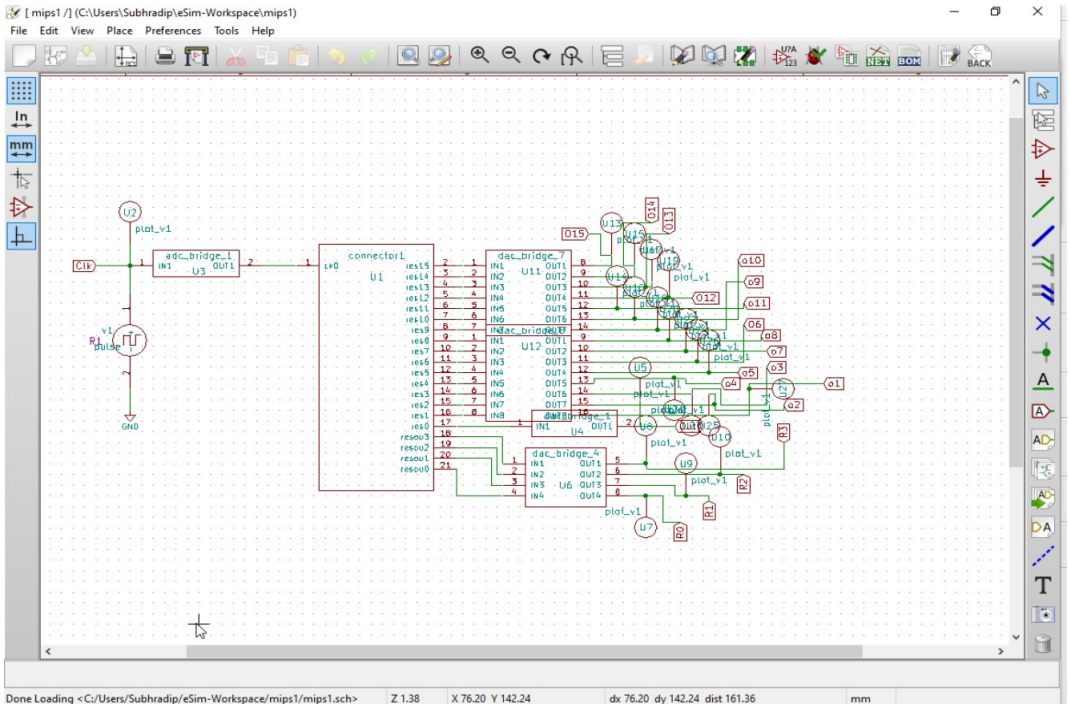

Figure 4.6: Circuit Design used for simulation

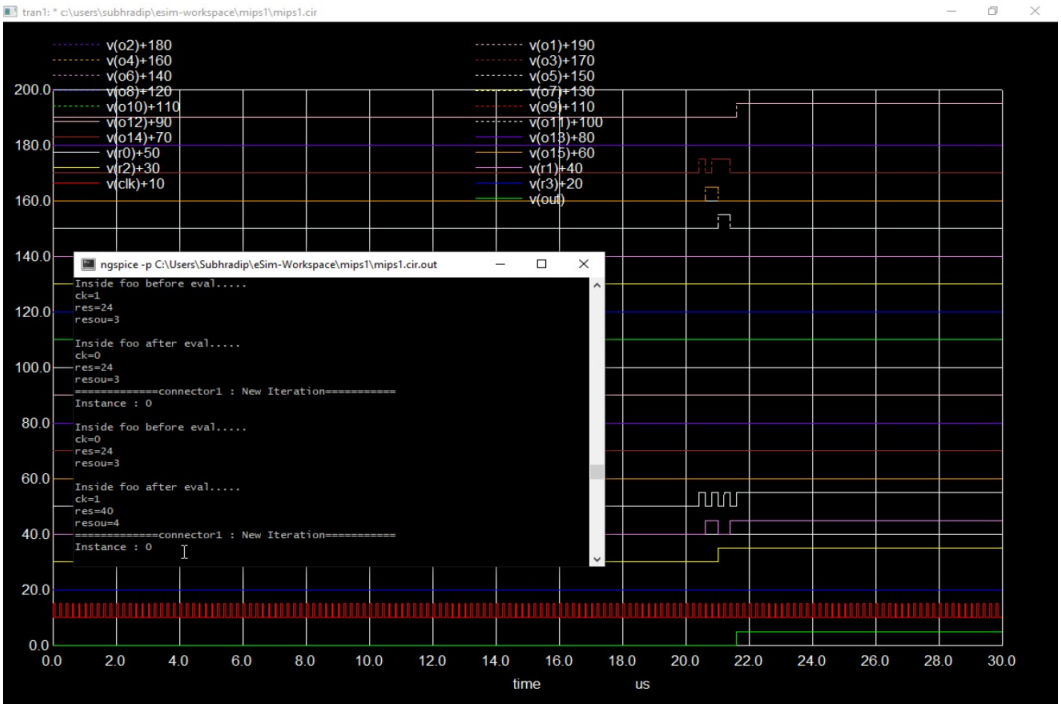

Figure 4.7: Simulation of the processor

Chapter 5

Ngveri Keywords In eSim-2.3

To convert a Verilog model into Ngspice we use the makerchip and ngveri. The makerchip is used to first simulate the file and check the resultant waveform and then the verilog file is converted. After a successful generation of the desired waveform we save the verilog file, then add the dependency files (if any) in Ngveri and convert the file into Ngspice.

While performing the conversion some of the keywords in Ngveri might throw an error while simulating. These are mainly the names and types of the input and output ports declared in the verilog file.

- All the input, output and inout ports should avoid names which include input, output, reg, wire, read, write, execute, decode, fetch as substrings.
- All variable names that are declared as default variables in verilog at the start of the code should not be used to define the size of any vector in the input and output ports.
- Should not use the unknown type (x) in variables it may throw an error while converting as it is not supported by verilator.
- Avoid using module names such as register it throws an error that file and module name are not the same although we used the same module and file name. The keyword register creates a problem.
- Do Not use a two dimensional vector example memory as an input and output port of a module, it will throw an error.
- Do Not use @ or wait commands which are not supported by verilator.

If you find any other keywords or variable names which throws an error while converting from verilog to Ngspice do let us know.

Chapter 6

Simulating Digital Circuits for MU

These are some of the cirucits which were asked by Mumbai University to be simulated and tested. I first coded all the circuits in verilog and then it was simulated in makerchip for simualtion.

6.0.1 Mux

```
module mux (
 input [3:0] a,
 input [3:0] b,
 input [3:0] c,
 input [3:0] d,
 input [1:0] sel,
 output reg [3:0] out
);

   always @ (a or b or c or d or sel) begin
      case (sel)
         2'b00 : out <= a;
         2'b01 : out <= b;
         2'b10 : out <= c;
         2'b11 : out <= d;
      endcase
   end

endmodule
```

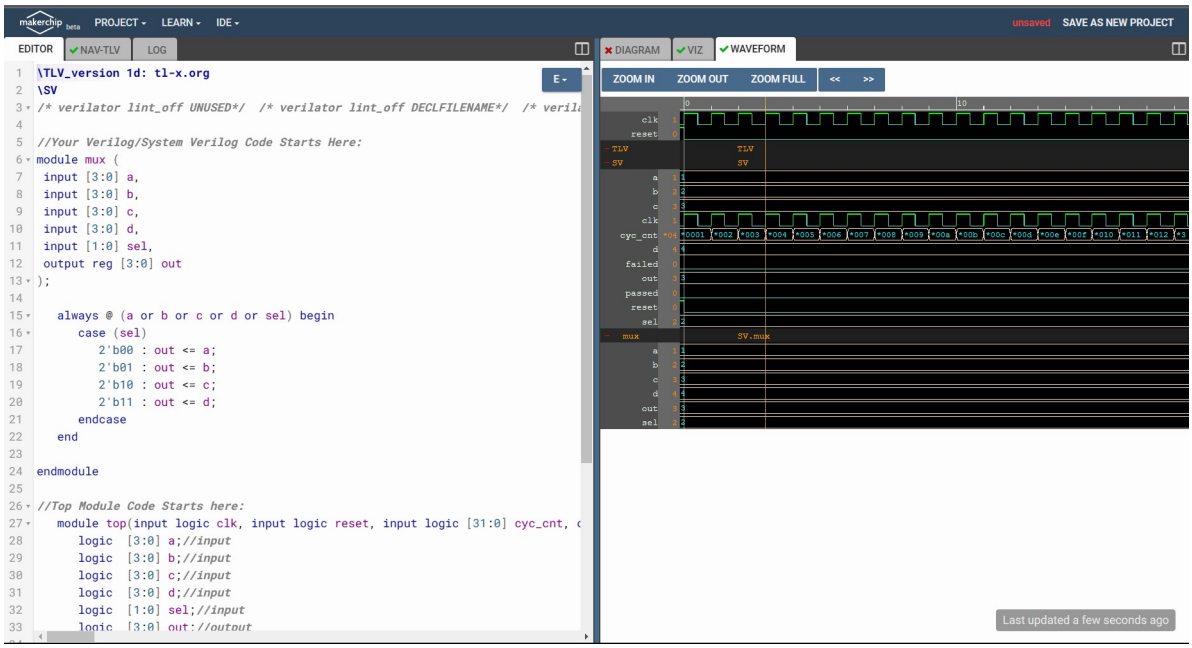

Figure 6.1: Simulation of MUX

6.0.2 DeMux

```
module Demux(
output reg [3:0] Y0,
output reg [3:0] Y1,
output reg [3:0] Y2,
output reg [3:0] Y3,
input [1:0] sel,
input [3:0] din);


always @(sel) begin
case (sel)
    2'b00 : begin
    Y0 = din;
    Y1 = 0;
    Y2 = 0;
    Y3 = 0;
    end
    2'b01 : begin
    Y0 = 0;
    Y1 = din;
    Y2 = 0;
    Y3 = 0;
    end
    2'b10 : begin
```

```
    Y0 = din;
    Y1 = 0;
    Y2 = din;
    Y3 = 0;
    end
    2'b11 : begin
    Y0 = 0;
    Y1 = 0;
    Y2 = din;
    Y3 = 0;
    end
endcase
end


endmodule
```

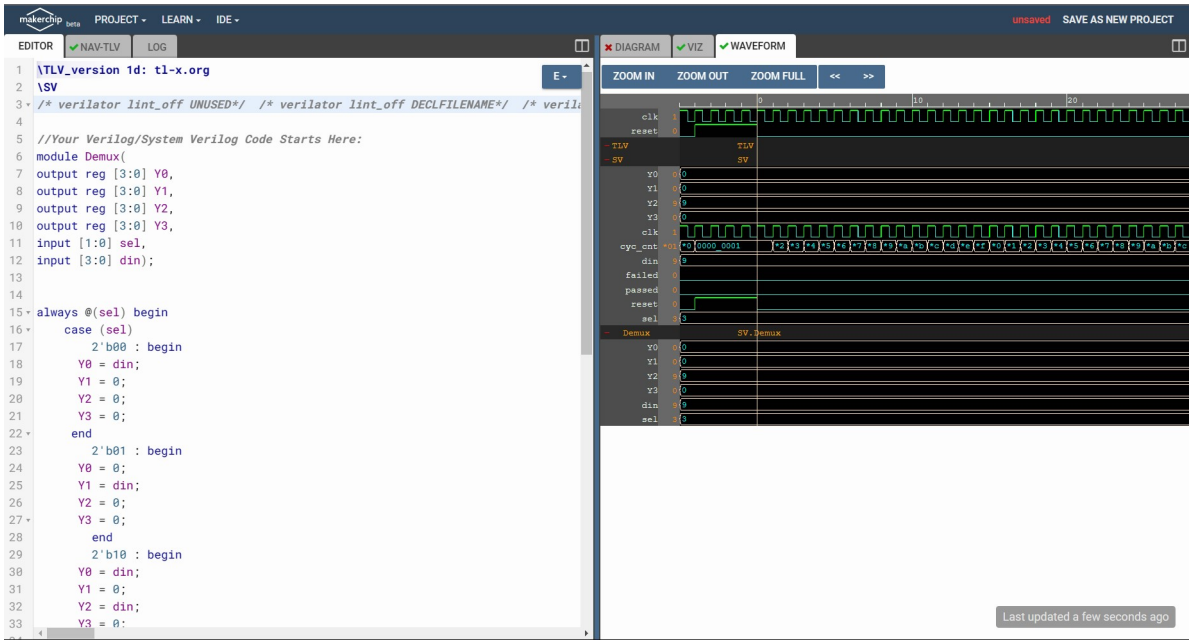

Figure 6.2: Simulation of DEMUX

6.0.3 Decoder IC 74138

```
module dec74138(
input a,
input b,
input c,
input e1_bar,
input e2_bar,
```

```
input e3,
output reg [7:0] out
);


always@(a or b or c) begin

if ((e1_bar == 0) && (e2_bar == 0) && (e3 == 1)) begin
    out=8'b11111111;
        case ({a,b,c})
            3'b000: out[0]=1'b0;
            3'b001: out[1]=1'b0;
            3'b010: out[2]=1'b0;
            3'b011: out[3]=1'b0;
            3'b100: out[4]=1'b0;
            3'b101: out[5]=1'b0;
            3'b110: out[6]=1'b0;
            3'b111: out[7]=1'b0;
        endcase
end
else
    out = 8'b11111111;
end

endmodule
```

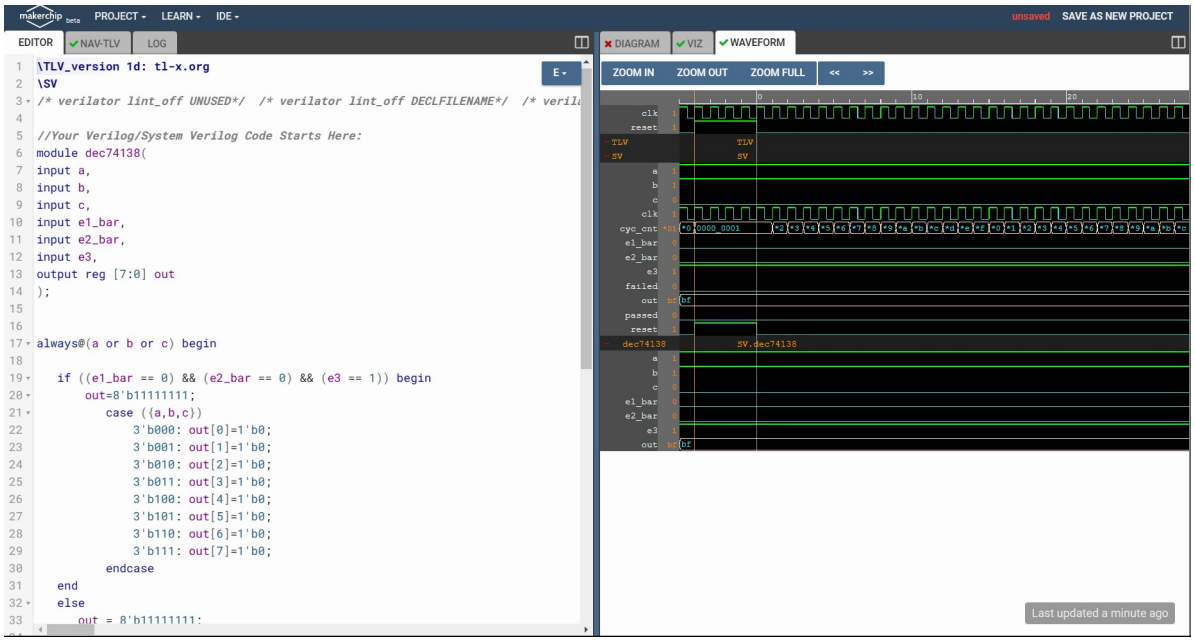

Figure 6.3: Simulation of DECODER IC

6.0.4 D Flip Flop

```
module d_ff(
input D
input clk
input reset
output reg Q
);


always @(posedge clk)
begin
 if(reset==1'b1)
  Q <= 1'b0;
 else
  Q <= D;
end


endmodule
```

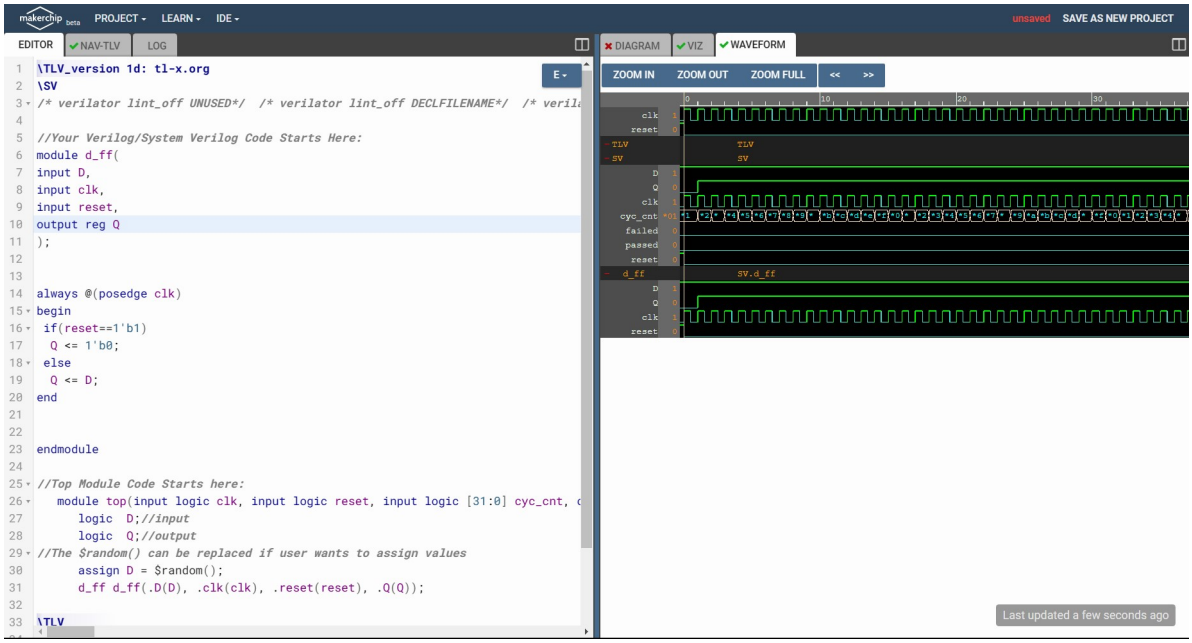

Figure 6.4: Simulation of D Flip Flop

6.0.5 JK Flip Flop

```
module jk_ff (
input j,
```

```
input k,
input clk,
output reg q,
output qb
);

   always @ (posedge clk)  begin
      case ({j,k})
         2'b00 :  q <= q;
         2'b01 :  q <= 0;
         2'b10 :  q <= 1;
         2'b11 :  q <= ~q;
      endcase
   end

assign qb = ~q;

endmodule
```

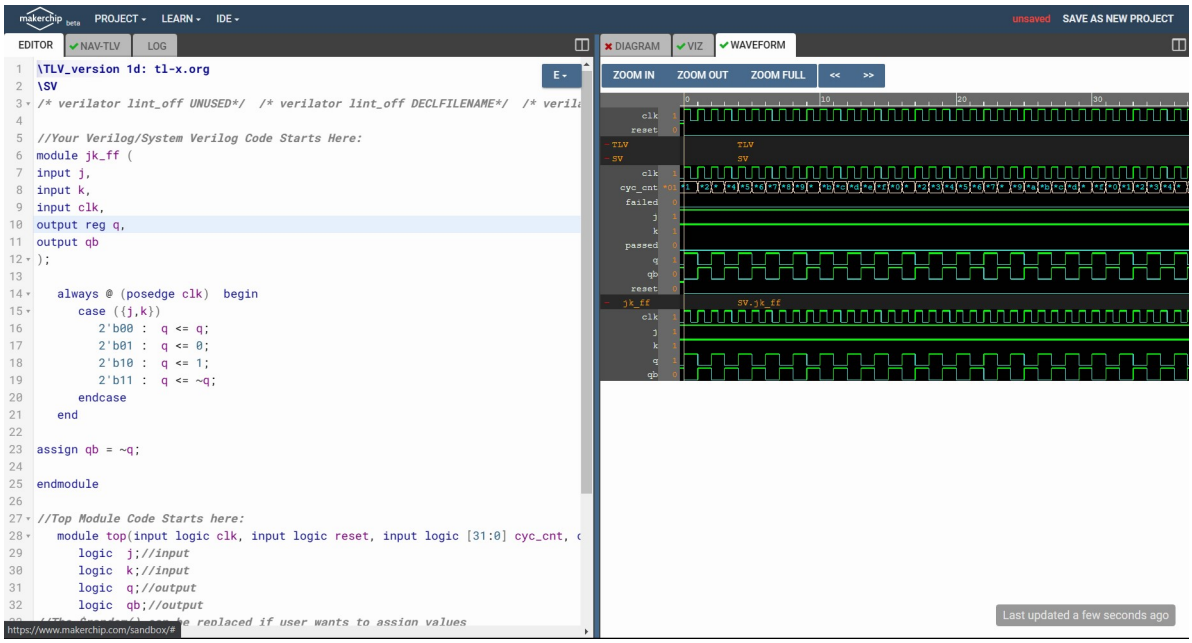

Figure 6.5: Simulation of JK Flip Flop

6.0.6 SR Flip Flop

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

```
wire out1;
wire out2;

nand (out1,clk,s);
nand (out2,clk,r);
nand (q,out1,qbar);
nand (qbar,out2,q);

endmodule
```

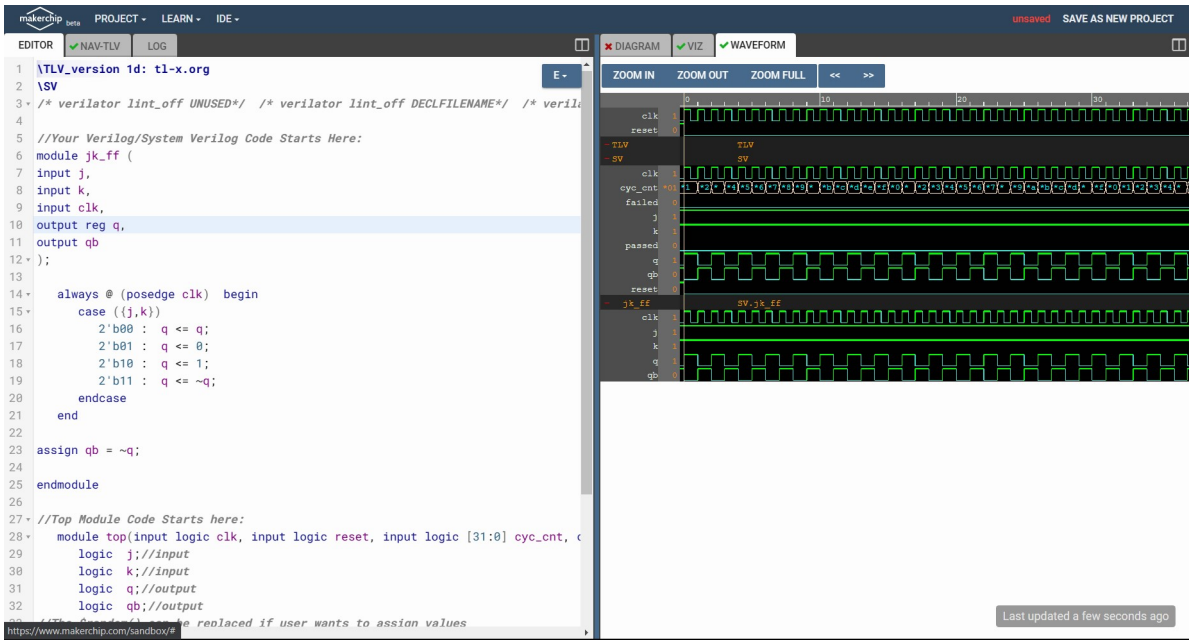

Figure 6.6: Simulation of SR Flip Flop

6.0.7 Universal Shift Register

```
module universal_shift(
input clk,
input reset,
input right_sel,
input din,
output reg [3:0] dout,
output s_left,
output s_right
);

always@(posedge clk) begin
if (reset == 1)
```

```
    dout <= 0;
else begin
    if (right_sel == 1)
        dout <= {din,dout[3:1]};
    else
        dout <= {dout[2:0],din};
end
end

assign s_left = dout[0];
assign s_right = dout[3];

endmodule
```

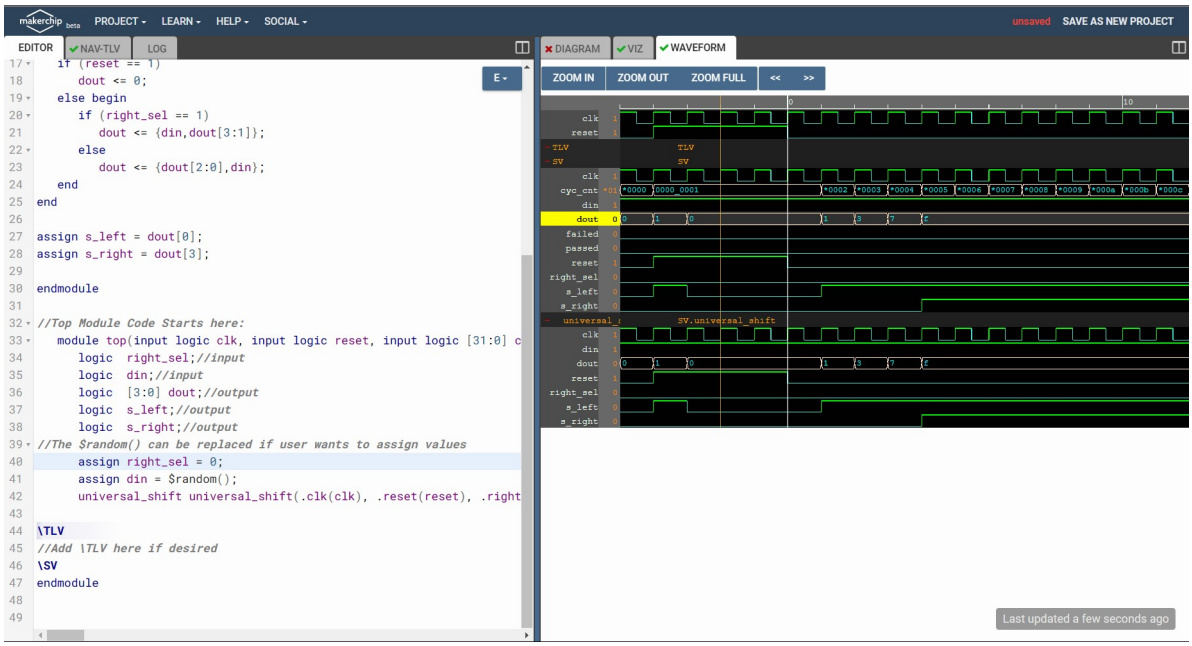

Figure 6.7: Simulation of Left Shift Register

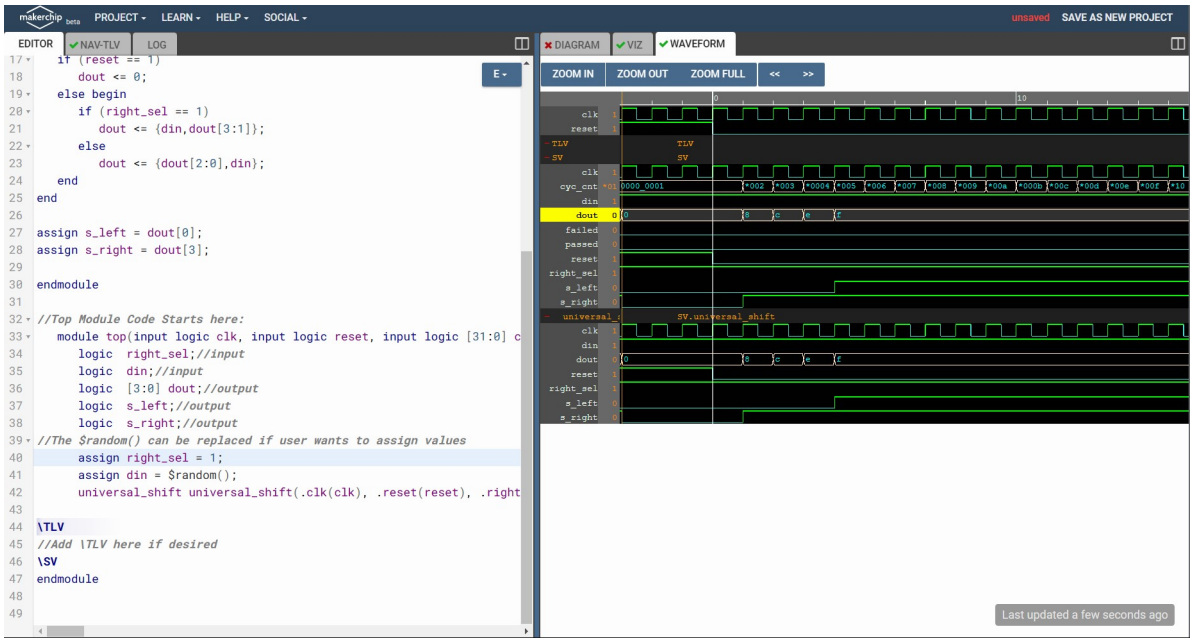

Figure 6.8: Simulation of Right Shift Register

6.0.8 Updown Counter

```
module upordown(
input clk,
input reset,
input up_high,
output reg [3:0] count
);


always@(posedge clk) begin
if(reset == 1)
    count <= 0;
else begin
    if(up_high == 1) begin
        if(count == 15)
            count <= 0;
        else
            count <= count + 1;
    end
    else begin
        if (count == 0)
            count <= 4'd15;
        else
            count <= count - 1;
    end
```

```
end
end

endmodule
```

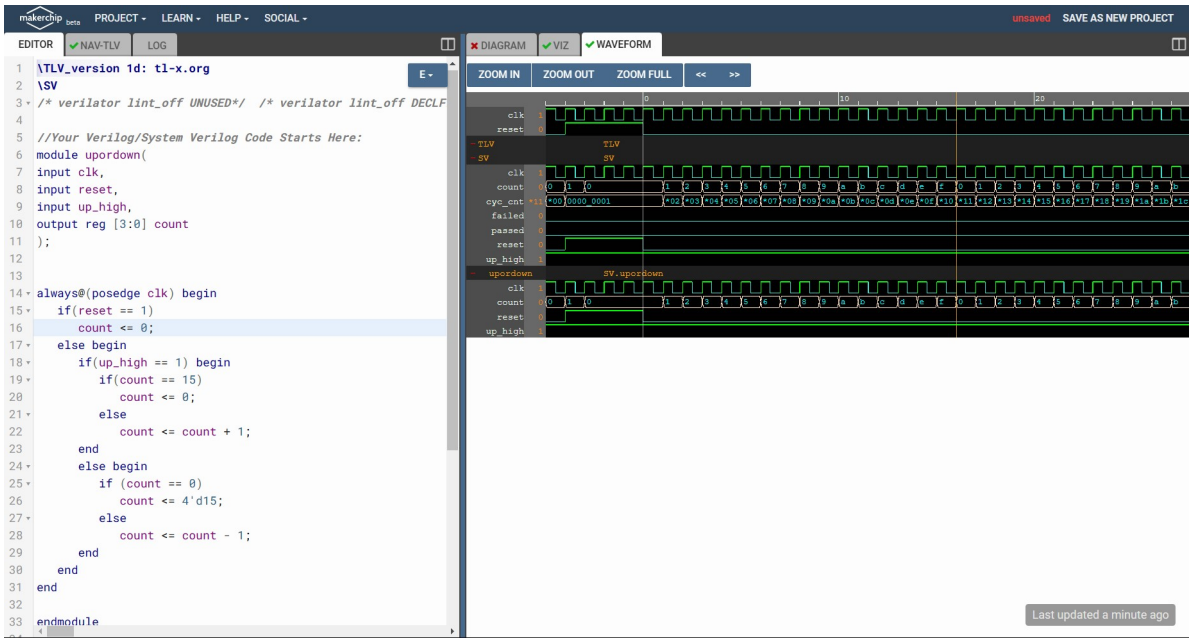

Figure 6.9: Simulation of Updown Counter

Bibliography

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

[2] Wikipedia Official Website.
URL: https://en.wikipedia.org/wiki/KiCad/

[3] Processor 1 SIMD.
URL: https://github.com/zslwyuan/Basic-SIMD-Processor-Verilog-Tutorial

[4] opencores Official Website.
URL: https://opencores.org/

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

[6] FOSSEE official webpage.
URL: https://fossee.in/fellowship/2019

[7] Geeks For Geeks official webpage.
URL: https://www.geeksforgeeks.org/bcd-to-7-segment-decoder

[8] Chip Verify official webpage.
URL: https://www.chipverify.com/