USC University of Southern California

USC Viterbi School of Engineering

Ming Hsieh Department of Electrical Engineering

EE577A

FINAL PROJECT REPORT

Design of a General Purpose CPU

Submitted By

Youngseok Lee - 4930239194

Narayana Reddy Lekkala - 9623274062

Chirag Ahuja - 5920609598

Phase 2 Part 1

A. Introduction

The core brain of the computer is the Central Processing unit. In this project, a pipelined microprocessor design has been obtained to process instructions such as arithmetic/logical operations, memory operations such as Store and Load word.

In order to increase the throughput of our design, a 5-stage pipeline has been used such that an output is present at the end of every clock cycle. Below is the diagram of 5-stage pipelined CPU where the Instruction Fetch and Instruction Decode stage has been completed using software script by Perl i.e. A Perl script fetches and decodes the instruction and generates a vector file. The rest of the stages are generated through the Hardware.

The vector file is provided as an input to the Register File stage which is further sent out to the Execution stage or the ALU according to its functionality such as OR, XOR, AND, MULTIPLY and ADD. The memory stage is a 512 bit SRAM facilitating the processing of STORE and LOAD instructions. The Write back stage writes the data back to the Register File.

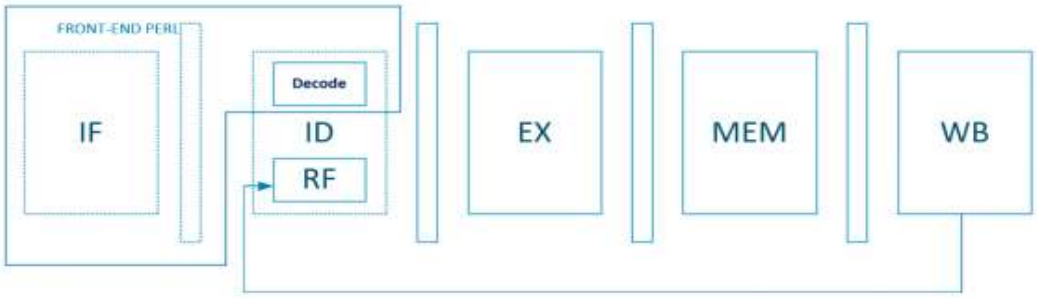
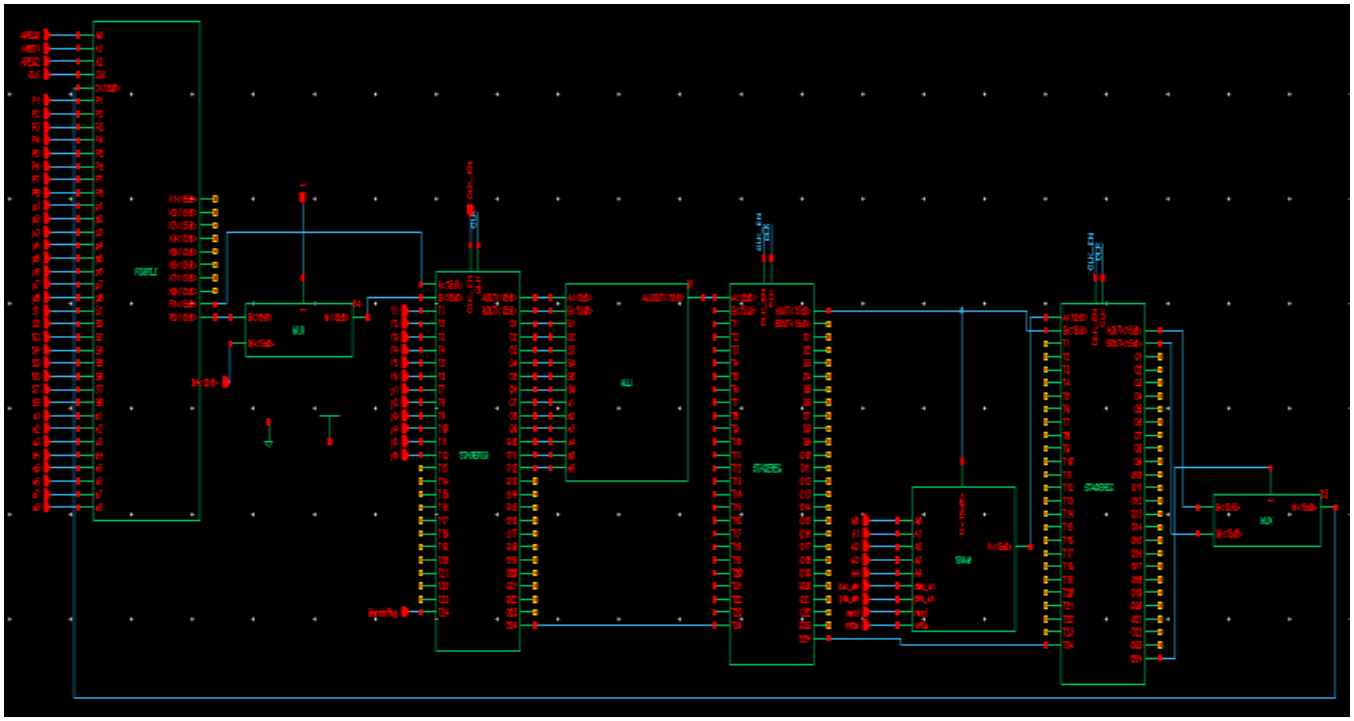


Figure 1. Block Diagram of a general purpose microprocessor

B. Data path design of the Processor

Description

The Design below is the schematic design of the microprocessor. The first block is register file (RF) with multiple input pins where the data is fed by a Perl-generated vector file. The working principle is discussed in introduction. The time that takes per cycle depends on the maximum clock value of any stage because the global clock should meet the minimum time for all the circuitry. *Note: Individual components are shown in Appendix at the end.*



Register File

Register Files are 8 16-bit registers that are consecutive D flip flops connected in series with mux select lines. Upon loading any registers specified by the address bit, the 16-bit values will directly be inputted into stage registers for ALU.

MUX 6 to 1

In case an immediate data comes directly from the Perl scripting file, immediate 16-bit values are written into the stage registers of ALU throughput the 6-to-1 MUX designed in the schematic above.

Stage Register

There are 3 stage registers with size as 56 bits. The purpose of having the stage registers is to store the input values from an instruction (32-bits) and the address bit values (3-bits) that indicate where the processed output should be written. Additional 20 bits are added in order to specify the kind of instruction, the control signals of mux, and SRAM signals.

ALU

The designed ALU supports instructions such as bitwise AND, OR, XOR, ADD and MUL with values from RF or/and immediate data through Perl code.

The three basic instructions AND, OR, and XOR are combined into one block. Later, this logic is clock-gated to save power and made dynamic to make the circuit faster. 32-bit Carry Ripple Adder has been used for the adder. 2's Compliment 8-bit Multiplier has been designed for multiplication. All arithmetic/logic results are inserted into ALU MUX, which only outputs the desired instruction results based on instruction type.

SRAM

Detailed descriptions of SRAM were discussed in Lab2.

C. Perl Scripting

Perl code for Data dependency

If the current instruction has the write register which matches the read register of the subsequent instruction or the instruction alternate to the current instruction, there would be a data dependency existing in the pipeline. The data dependency can be removed by inserting NOPs.

Algorithm used

- **For dependency between current and subsequent instruction: Insert 2 NOPs.**
- **For dependency between current and alternate instruction: Insert 1 NOP.**

The process of inserting NOPs to remove data dependency is called **Local Instruction Scheduling.**

Local Instruction Scheduling using Perl Code

The Perl code is used to read the existing/provided cmd.txt and verify all the dependencies. It then generates an updated cmd.txt that will have required (inserted) NOPs with removed dependencies based on the algorithm discussed above.

This updated cmd.txt is further used to generate the vector file which is provided as an input to the hardware stage i.e. Register File.

Perl code to remove dependency

Scheduling NOPs.pl

```perl
my $commandFile = "cmd.txt";
my $updatedcommandFile = "cmdUpdated.txt";

unless(open commandFILE, $commandFile) {
 die "Unable to open $commandFile";
}

unless(open updatedcommandFile, '>'.$updatedcommandFile) {
 die "Unable to create $updatedcommandFile";
}

my @cmdArray; #Initializing an array for the instructions in the cmd.txt
$row = 0;

while(my $line = <commandFILE>) {

    $line =~ s/^\s+|\s+$//g; #Truncating white spaces from the beginning or the end of the line.
    @value = split(/ /, $line); #Storing the operands of the instruction in an array.

    foreach $column (@value)
    {
        push @{$cmdarray[$row]}, $column; #Pushing elements into the array of commands
    }
    $row = $row + 1;
}

#Initializing a set of flags
my $fns = 0; #Flag for First and Second instruction dependency
my $fnt = 0; #Flag for First and Third instruction dependency
my $snt = 0; #Flag for Second and Third instruction dependency
my $extra = 0;

foreach $row (0..@cmdarray-1)
{
    $fns = 0;
    $fnt = 0;
    $snt = 0;

    foreach $column (0..@{$cmdarray[$row]}-1)
    {
        if ($column == 0)
        {
            print updatedcommandFile $cmdarray[$row][$column];
            if($cmdarray[$row][$column] eq "NOP")
            {
                print updatedcommandFile "\n";
            }
            else
            {
                print updatedcommandFile " ";
            }
        }

        #Checking for the dependencies in the cmd.txt file
        elsif($column == 1)
        {
            #Checking for the dependencies between First instruction and Second instruction -- Start
            if((length($cmdarray[$row][$column]) > 1)&&(length($cmdarray[$row+1][$column]) > 1))
            {
                if(($cmdarray[$row][$column] eq $cmdarray[$row+1][$column+1])||($cmdarray[$row][$column] eq $cmdarray[$row+1][$column+2]))
                {
                    $fns = 1;
                }
            }
            elsif((length($cmdarray[$row][$column]) > 1)&&(length($cmdarray[$row+1][$column]) == 1))
            {
```

Declaring flag variables for three types of dependencies.

```
            if(($cmdarray[$row][$column] eq $cmdarray[$row+1][$column+1]))
            {
                $fns = 1;
            }
        }
        elsif(($length($cmdarray[$row][$column]) == 1)&&($length($cmdarray[$row+1][$column]) > 1))
        {
            if(($cmdarray[$row][$column+1] eq $cmdarray[$row+1][$column+1]))||($cmdarray[$row][$column+1] eq $cmdarray[$row+1][$column+2]))
            {
                $fns = 1;
            }
        }
        #Checking for the dependencies between First instruction and Second instruction -- End

        #Checking for the dependencies between First instruction and Third instruction -- Start
        if(($length($cmdarray[$row][$column]) > 1)&&($length($cmdarray[$row+2][$column]) > 1))
        {
            if(($cmdarray[$row][$column] eq $cmdarray[$row+2][$column+1]))||($cmdarray[$row][$column] eq $cmdarray[$row+2][$column+2]))
            {
                $fnt = 1;
            }
        }
        elsif(($length($cmdarray[$row][$column]) > 1)&&($length($cmdarray[$row+2][$column]) == 1))
        {
            if(($cmdarray[$row][$column] eq $cmdarray[$row+2][$column+1]))
            {
                $fnt = 1;
            }
        }
        elsif(($length($cmdarray[$row][$column]) == 1) && ($length($cmdarray[$row+2][$column]) > 1))
        {
            if(($cmdarray[$row][$column+1] eq $cmdarray[$row+2][$column+1]))||($cmdarray[$row][$column+1] eq $cmdarray[$row+2][$column+2]))
            {
                $fnt = 1;
            }
        }
        #Checking for the dependencies between First instruction and Third instruction -- End

        #Checking for the dependencies between Second instruction and Third instruction -- Start
        if(($length($cmdarray[$row+1][$column]) > 1) && ($length($cmdarray[$row+2][$column]) > 1))
        {
            if(($cmdarray[$row+1][$column] eq $cmdarray[$row+2][$column+1]))||($cmdarray[$row+1][$column] eq $cmdarray[$row+2][$column+2]))
            {
                $snt = 1;
            }
        }
        elsif(($length($cmdarray[$row+1][$column]) > 1) && ($length($cmdarray[$row+2][$column]) == 1))
        {
            if(($cmdarray[$row+1][$column] eq $cmdarray[$row+2][$column+1]))
            {
                $snt = 1;
            }
        }
        elsif(($length($cmdarray[$row][$column]) == 1) && ($length($cmdarray[$row+2][$column]) > 1))
        {
            if(($cmdarray[$row][$column+1] eq $cmdarray[$row+2][$column+1]))||($cmdarray[$row][$column+1] eq $cmdarray[$row+2][$column+2]))
            {
                $snt = 1;
            }
        }
        #Checking for the dependencies between Second instruction and Third instruction -- End

        foreach $columnTemp (1..$($cmdarray[$row])-1)
        {
            print updatedcommandFile $cmdarray[$row][$columnTemp];
            print updatedcommandFile " ";
        }

        print updatedcommandFile "\n";

        if (($fns == 1 && $errs == 0)
        {
            print updatedcommandFile "NOP\nNOP\n";
            last;
        }

        if (($fns == 0) && ($fnt == 1) && ($snt == 0))
        {
            print updatedcommandFile "NOP\n";
            last;
        }

        if (($fnt == 1) && ($snt == 1))
        {
            foreach $columnTemp (0..$($cmdarray[$row+1])-1)
            {
                print updatedcommandFile $cmdarray[$row+1][$columnTemp];
                print updatedcommandFile " ";
            }
            print updatedcommandFile "\n";
            print updatedcommandFile "NOP\nNOP\n";
            $row = $row + 1;
            $errs = 1;
            last;
        }

        if(($fns == 0) && ($fnt == 0))
        {
            last;
```

Logic to check the Register IDs of the destination register of the current instruction with the source register of the next instruction

Cmd.txt without Local Instruction Scheduling

```
STOREI 0AH #0002
STOREI 0BH #001f
STOREI 2 10H #0F0F #F0F0
LOAD $1 0AH
LOAD $2 0BH
LOAD $3 10H
LOAD $4 11H
MUL $5 $1 $2
ADD $6 $3 $5
NOP
STORE 00H $5
STORE 01H $6
ORI $5 $3 #AAAA
XOR $6 $2 $4
AND $7 $5 $6
STORE 02H $5
STORE 03H $6
STORE 04H $7
LOAD $0 00H
LOAD $0 01H
LOAD $0 02H
LOAD $0 03H
LOAD $0 04H
```

Cmd.txt after Local Instruction Scheduling

cmd.pl | cmd.txt | cmdUpdated.txt

```
STOREI 0AH #0002
STOREI 0BH #001f
STOREI 2 10H #0F0F #F0F0
LOAD $1 0AH
LOAD $2 0BH
LOAD $3 10H
LOAD $4 11H
MUL $5 $1 $2
NOP
NOP
ADD $6 $3 $5
NOP
STORE 00H $5
STORE 01H $6
ORI $5 $3 #AAAA
XOR $6 $2 $4
NOP
NOP
AND $7 $5 $6
STORE 02H $5
STORE 03H $6
STORE 04H $7
LOAD $0 00H
LOAD $0 01H
LOAD $0 02H
LOAD $0 03H
LOAD $0 04H
```

Perl code to generate Vector File

A part of the Perl script is shown here. The file is attached with the report.

```perl
if($pass == 0)
{
    # Filling IMs
    if(index($value[0], "I") != -1)
    {
        print vectorFILE $tunit;
        print vectorFILE "\t";
        @dataArray = split //, $value[2];
        print vectorFILE $dataArray[1];
        print vectorFILE " ";
        print vectorFILE $dataArray[2];
        print vectorFILE " ";
        print vectorFILE $dataArray[3];
        print vectorFILE " "
        print vectorFILE $dataArray[4];
        print vectorFILE "\t";
    }
    else
    {
        print vectorFILE $tunit;
        print vectorFILE "\t0 0 0 0\t";
    }

    #Filling AREG
    if(index($value[1], "S") != -1)
    {
        @reg = split //, $value[1];

        print vectorFILE $regwrite[$reg[1]][0];
        print vectorFILE " ";
        print vectorFILE $regwrite[$reg[1]][1];
        print vectorFILE " ";
        print vectorFILE $regwrite[$reg[1]][2];
```

```perl
#Filling Y and y
if((index($value[0], "I") != -1)||($value[0] eq "STORE")||($value[0] eq "LOAD"))
{
    print vectorFILE "0 1 1 1 1 1\t";
    print vectorFILE "1 0 0 0 0 0\t";
}
else
{
    if($value[0] eq "MUL")
    {
        print vectorFILE "1 0 1 1 1 1\t";
        print vectorFILE "0 1 0 0 0 0\t";
    }
    if($value[0] eq "ADD")
    {
        print vectorFILE "1 1 0 1 1 1\t";
        print vectorFILE "0 0 1 0 0 0\t";
    }
    if($value[0] eq "AND")
    {
        print vectorFILE "1 1 1 0 1 1\t";
        print vectorFILE "0 0 0 1 0 0\t";
    }
    if($value[0] eq "OR")
    {
        print vectorFILE "1 1 1 1 0 1\t";
        print vectorFILE "0 0 0 0 1 0\t";
    }
    if($value[0] eq "XOR")
    {
        print vectorFILE "1 1 1 1 1 0\t";
```

```perl
#Filling CLK_En
print vectorFILE "1\t";

#Filling MemToReg

if(($value[0] eq "STORE")||($value[0] eq "LOAD"))
{
    print vectorFILE "0\t";
}
else
{
    print vectorFILE "1\t";
}

#Filling A0-A4

if((index($value[1], "H") != -1)||(index($value[2], "H") != -1))
{
    if(index($value[1], "H") != -1)
    {
        @add = split //, $value[1];
    }
    else
    {
        @add = split //, $value[2];
    }

    print vectorFILE $add[0];
    print vectorFILE " ";
    print vectorFILE getBinary($add[1]);
```

Logic for the mux values for different operations in the vector file

Logic to Fill the Immediate data from the instructions in the vector file

Logic for extracting the address for STORE and LOAD instructions.

Vector File

FINALPROJECT577A (1).vec

```
radix          4 4 4 4  1 1 1 1  1 1 1 1 1 1 1  1 1 1 1 1 1 1  1 1 1  1 1 1 1 1  1 1 1 1   1 1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1 1
io          i i i i  i i i i  i i i i i i i  i i i i i i i  i i i  i i i i i  i i i i   i i i i i i i i i  i i i i i i i i i  i i i i i i i i i  i i i i i i i i i
vname   IM<[15:12]> IM<[11:8]> IM<[7:4]> IM<[3:0]>    AREG0 AREG1 AREG2 I   Y1 Y2 Y3 Y4 Y5 Y6  y1 y2 y3 y4 y5 y6 CLK CLK_EN MemtoReg A0 A1 A2 A3 A4 pre_en dec_er
slope   0.01
vih     1.8
tunit   ns

0.1     0 0 0 0                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   0 1   0              0 0 0 0 0        1
;STOREI 0AH #0002

13      0 0 0 2                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   0 1   0              0 1 0 1 0        1
16      0 0 0 2                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   1 1   0              0 1 0 1 0        1


19      0 0 0 2                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   0 1   0               0 1 0 1 0
22      0 0 0 2                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   1 1   0               0 1 0 1 0
25      0 0 0 2                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   0 1   0               0 1 0 1 0

;STOREI

13      0 0 1 F                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   0 1   0              0 1 0 1 1        1
16      0 0 1 F                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   1 1   0              0 1 0 1 1        1



28      0 0 1 F                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   1 1   0              0 1 0 1 1        0
31      0 0 1 F                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   0 1   0              0 1 0 1 1        1
34      0 0 1 F                                 0 0 0  1                0 1 1 1 1 1  1 0 0 0 0 0   1 1   0              0 1 0 1 1        1
```

STORE Immediate Data in the SRAM

LOAD Data from the SRAM into the Register

```
;LOAD

82      0 0 0 0                                 1 1 0  1                0 1 1 1 1 1  1 0 0 0 0 0   1 1   0              1 0 0 0 1        0
85      0 0 0 0                                 1 1 0  1                0 1 1 1 1 1  1 0 0 0 0 0   0 1   0              1 0 0 0 1        1
88      0 0 0 0                                 1 1 0  1                0 1 1 1 1 1  1 0 0 0 0 0   1 1   0              1 0 0 0 1        1

64      0 0 0 0                                 1 1 0  1                0 1 1 1 1 1  1 0 0 0 0 0   1 1   0              1 0 0 0 1        1
67      0 0 0 0                                 1 1 0  1                0 1 1 1 1 1  1 0 0 0 0 0   0 1   0              1 0 0 0 1        1
70      0 0 0 0                                 1 1 0  1                0 1 1 1 1 1  1 0 0 0 0 0   1 1   0              1 0 0 0 1        1

;MUL WITH NOPS

97      0 0 0 0                                 0 0 0  0                1 0 1 1 1 1  0 1 0 0 0 0   0 1   1              0 1 0 0 1        1 (
100     0 0 0 0                                 0 0 0  0                1 0 1 1 1 1  0 1 0 0 0 0   1 1   1              0 1 0 0 1        1 (
103     0 0 0 0                                 0 0 0  0                1 0 1 1 1 1  0 1 0 0 0 0   0 1   1              0 1 0 0 1        1 (
107     0 0 0 0                                 0 0 0  0                1 0 1 1 1 1  0 1 0 0 0 0   1 1   1              0 1 0 0 1        1 (
110     0 0 0 0                                 0 0 0  0                1 0 1 1 1 1  0 1 0 0 0 0   0 1   1              0 1 0 0 1        1 (
113     0 0 0 0                                 0 0 0  0                1 0 1 1 1 1  0 1 0 0 0 0   1 1   1              0 1 0 0 1        1 (
116     0 0 0 0                                 0 0 0  0                1 0 1 1 1 1  0 1 0 0 0 0   0 1   1              0 1 0 0 1        1 (
```

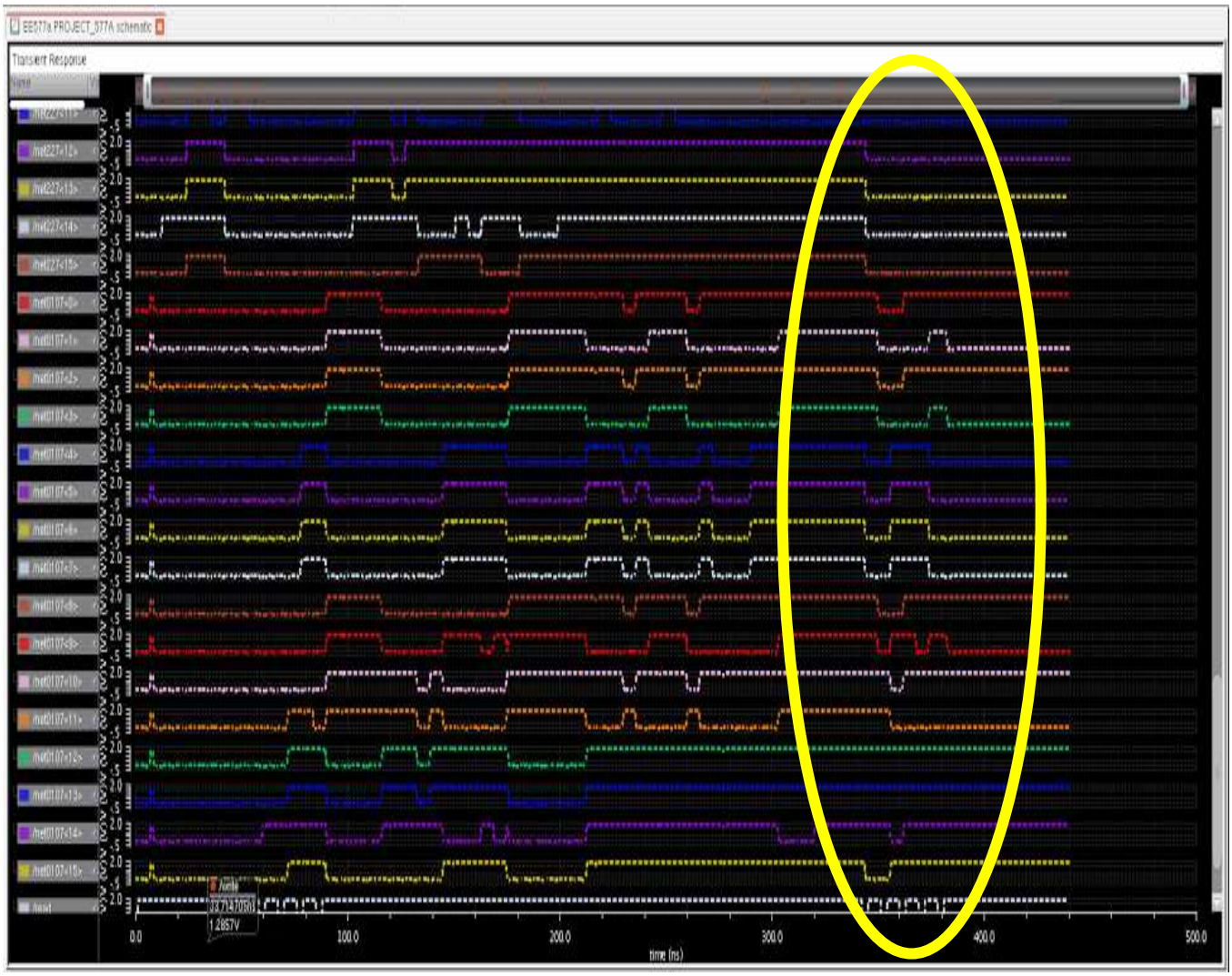
Arithmetic Operation such as MULTIPLY values in two registers and write result into third Register

D. Functionality of the Design/Waveforms

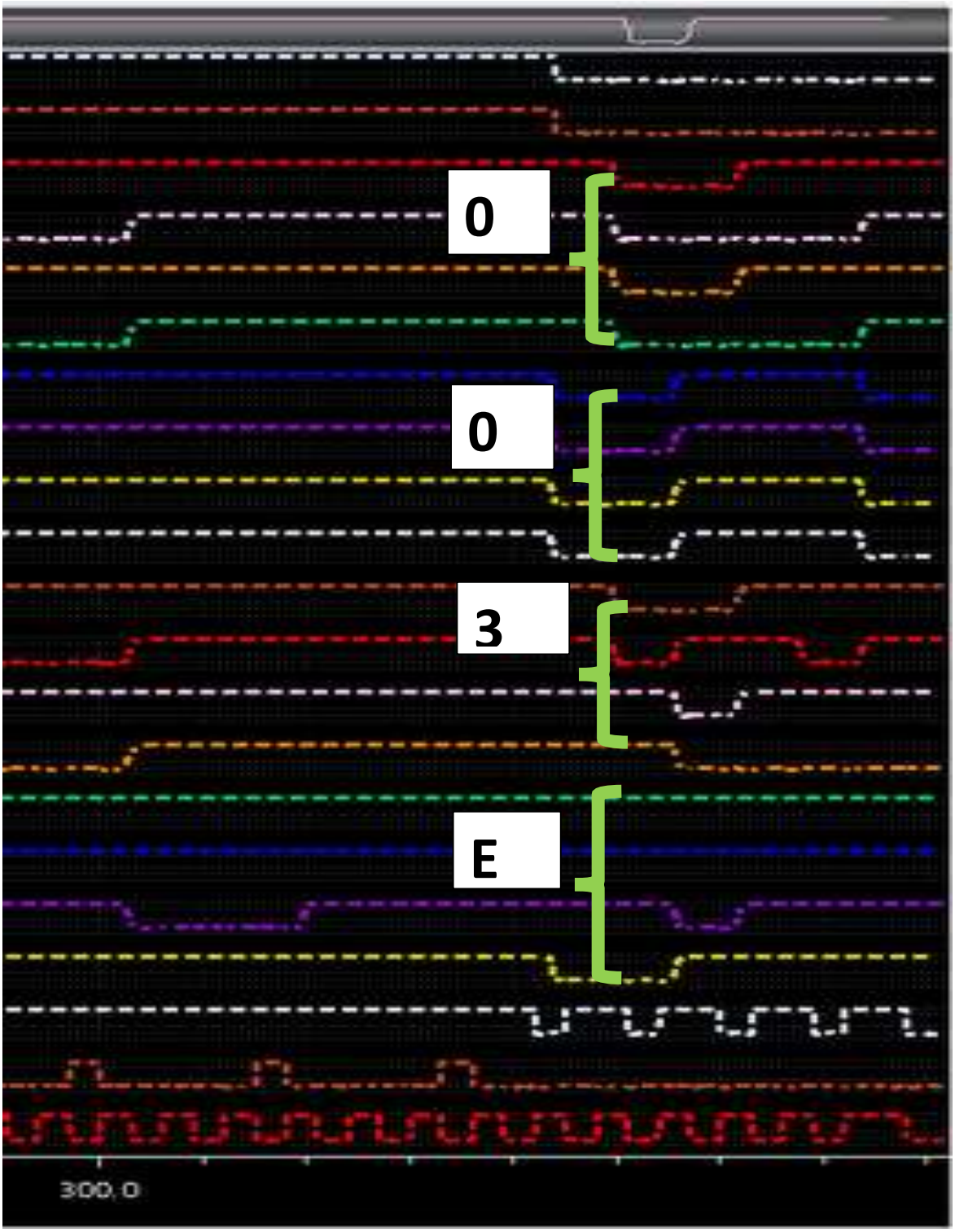
Waveform verification with vector file

Vector file (as shown in the report previously) generated through the Perl code is used to verify the functionality of the design.

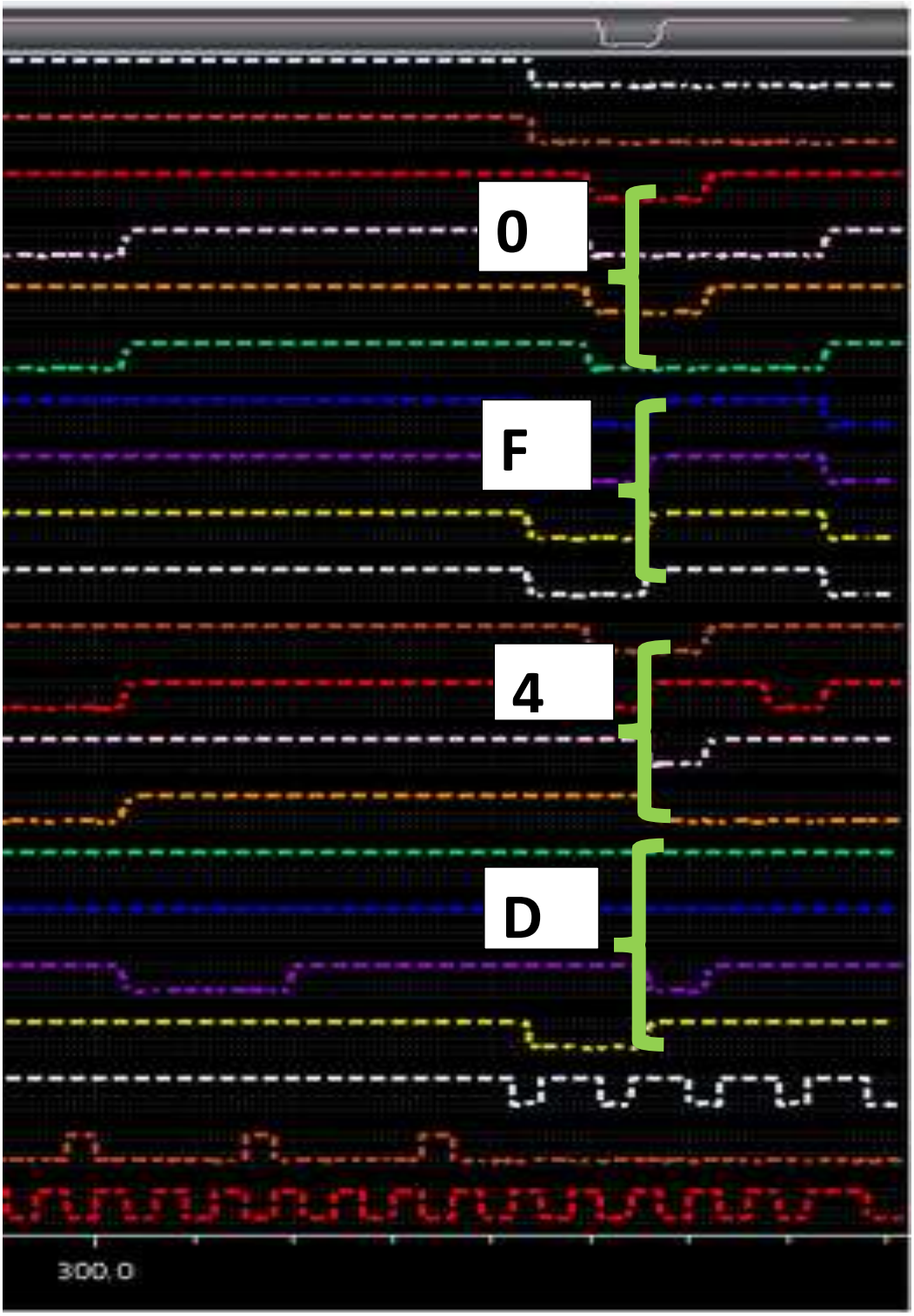
The set of output waveforms are shown with the highlighted oval below and is also shown in detail, data-wise.



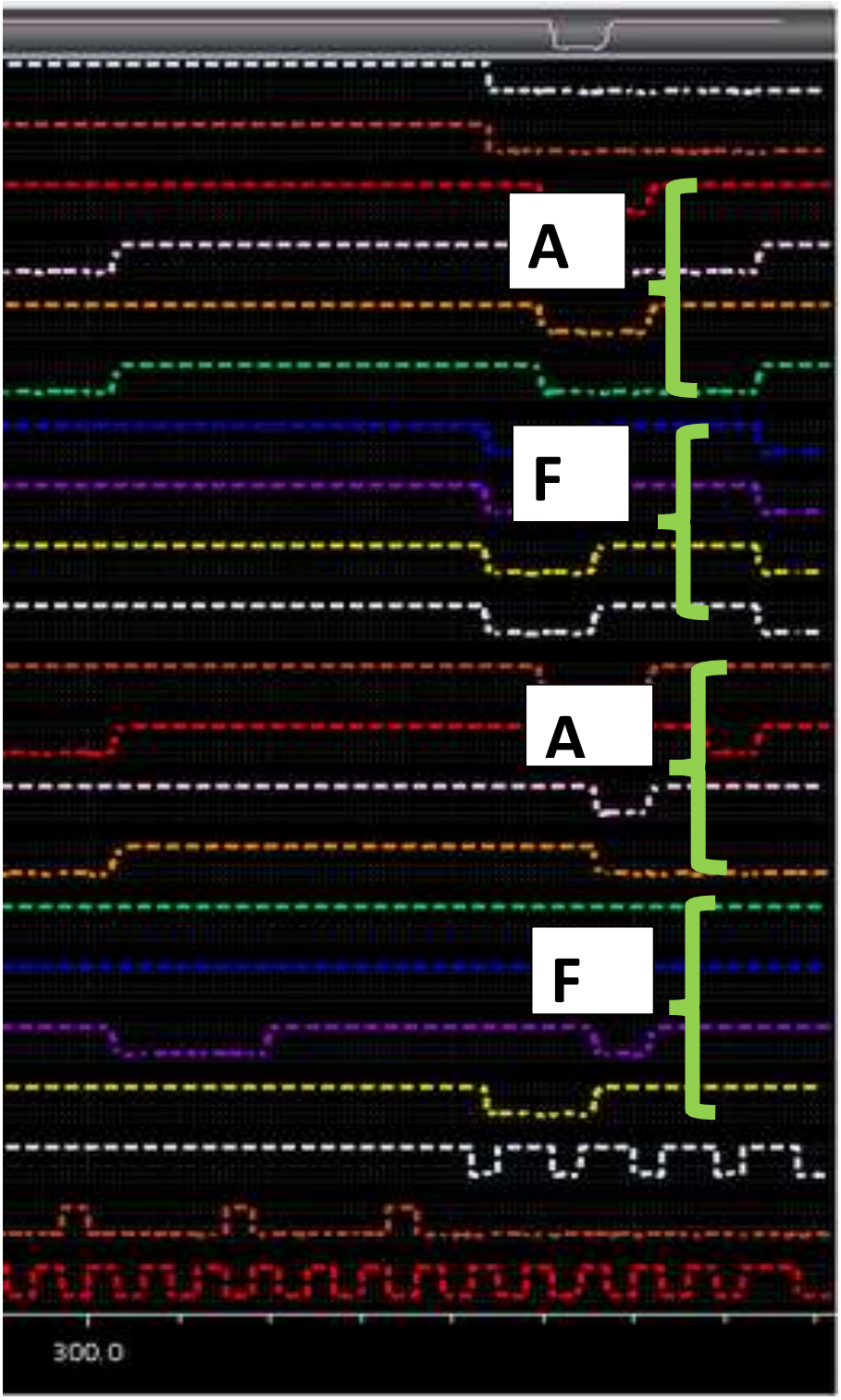
Result of LOAD $0 00H



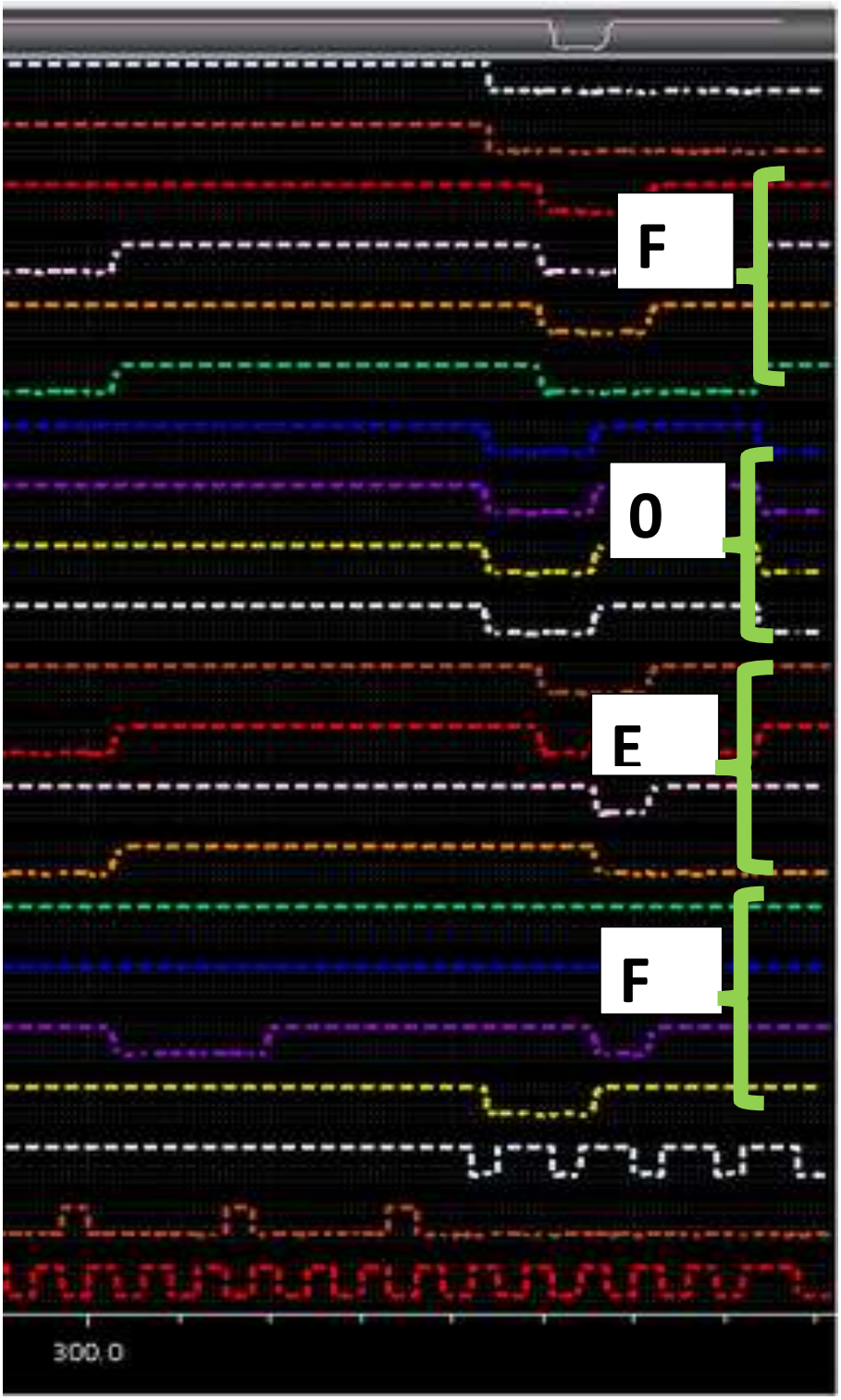
Result of LOAD $0 01H



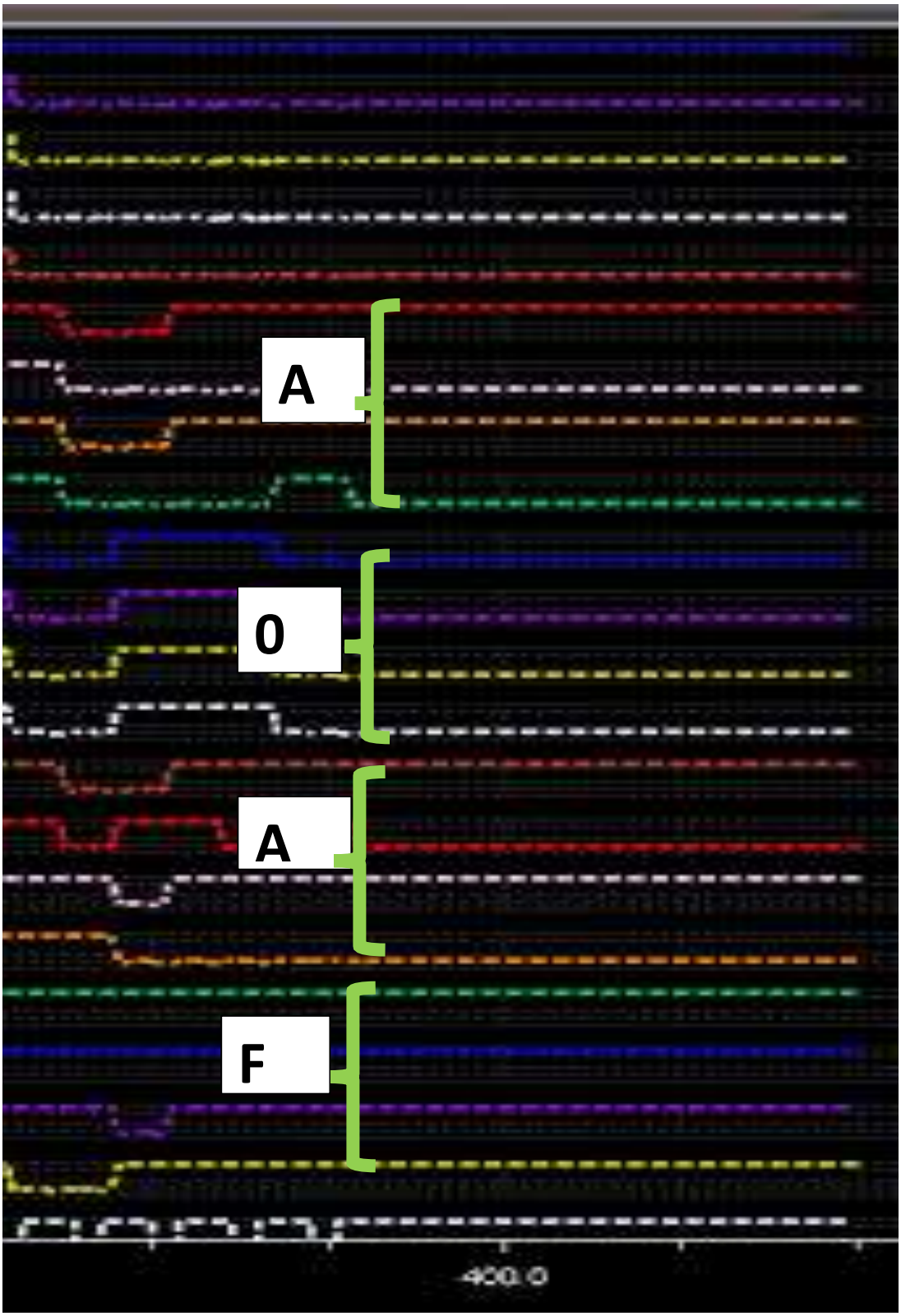
Result of LOAD $0 02H



Result of LOAD $0 03H



Result of LOAD $0 04H



Phase 2 Part 2

A. Layout of the Pipelined Processor

All the layouts of individual components along with their LVS matches are displayed in the Appendix section at the end of this report.

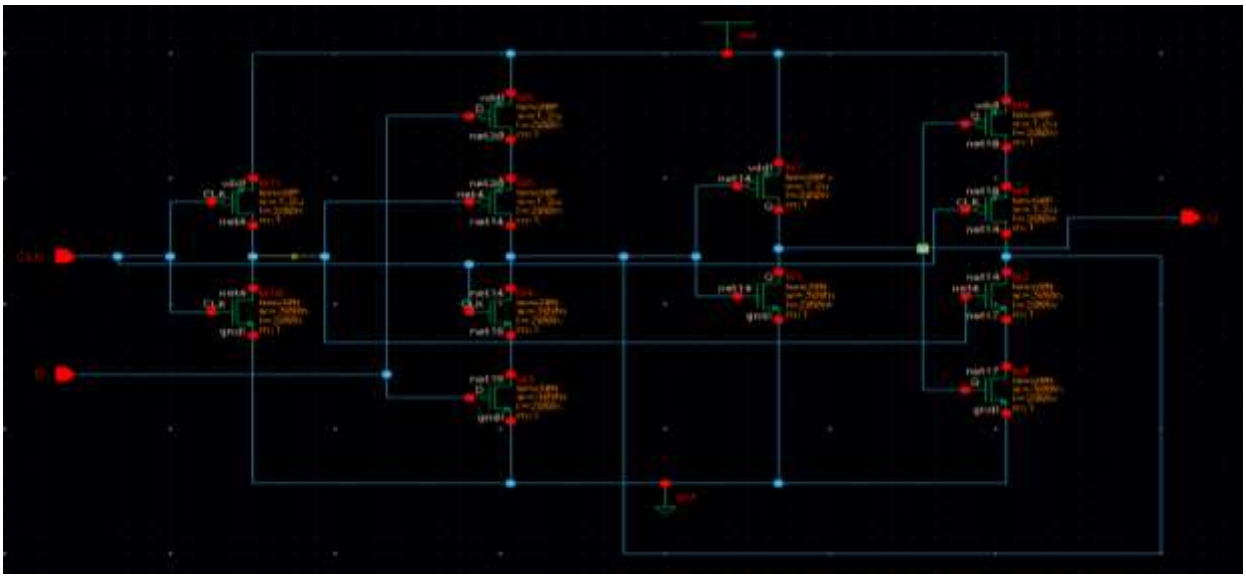
The Layout of the Data-path is still in progress and will be shown at the time of Demo with the LVS match.

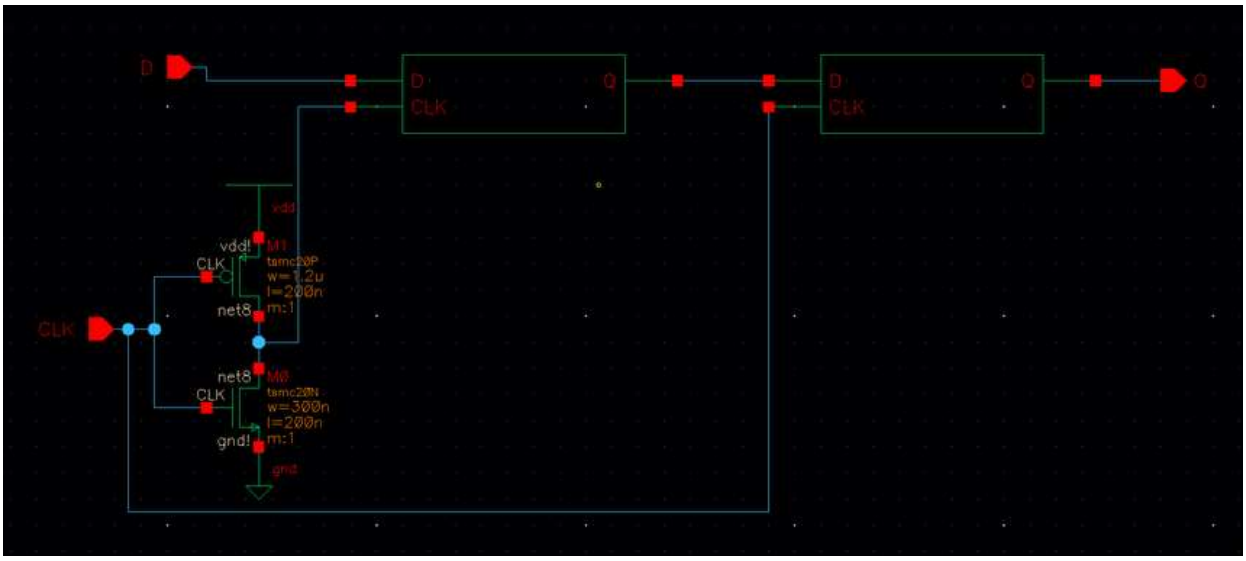
For the power optimization and dynamic, please see below.

B. D Flip Flop Optimization

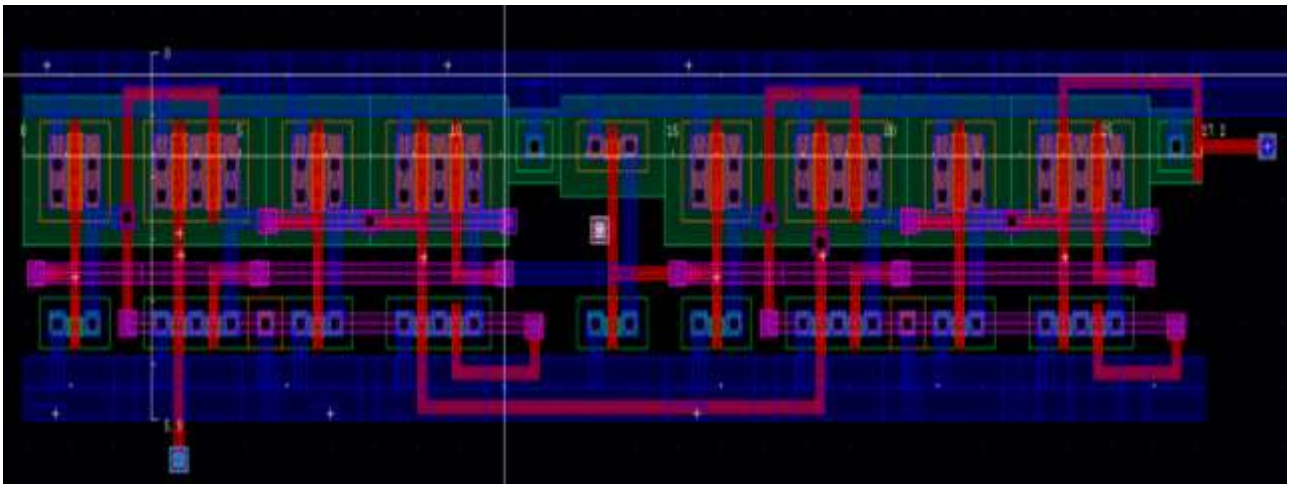
Optimized Design of the D Flip Flop

The D Flip Flop design has been optimized using the full custom design of transistors and without using any gates. A D-Latch is built using transistors and two such latches are connected in master and slave configurations to obtain the D-Flip Flop such that it replaces the need of SET and RESET PINS.



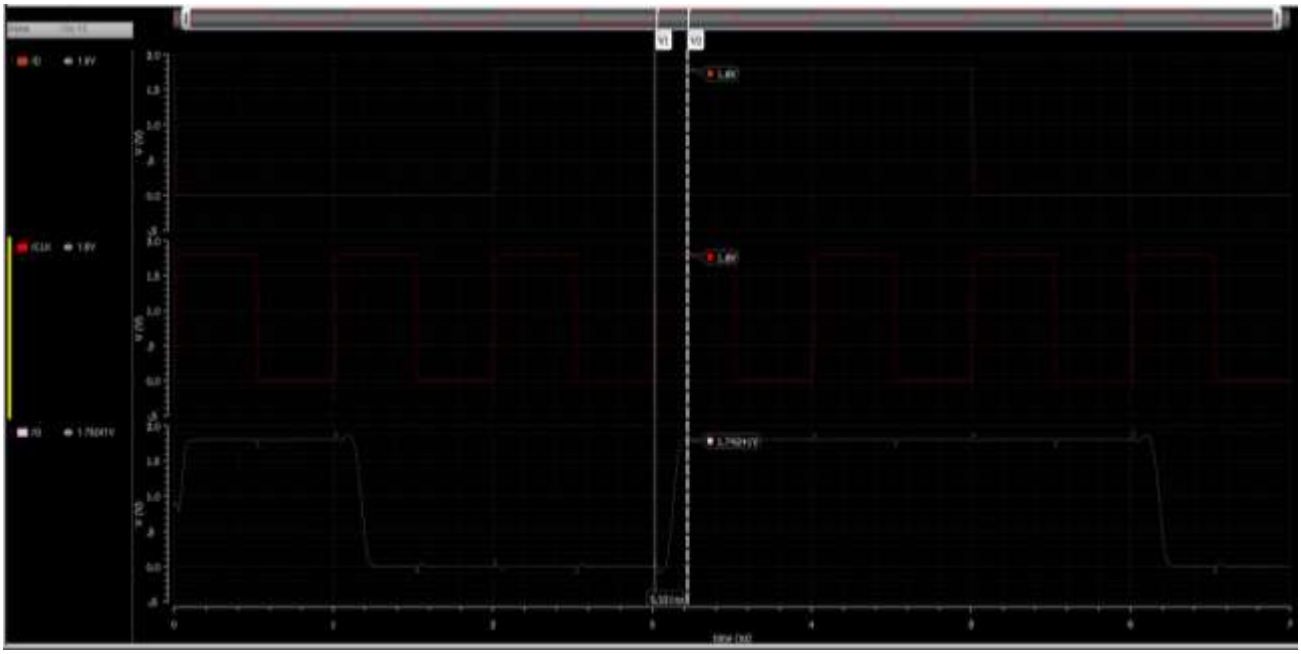


Layout of D Flip Flop



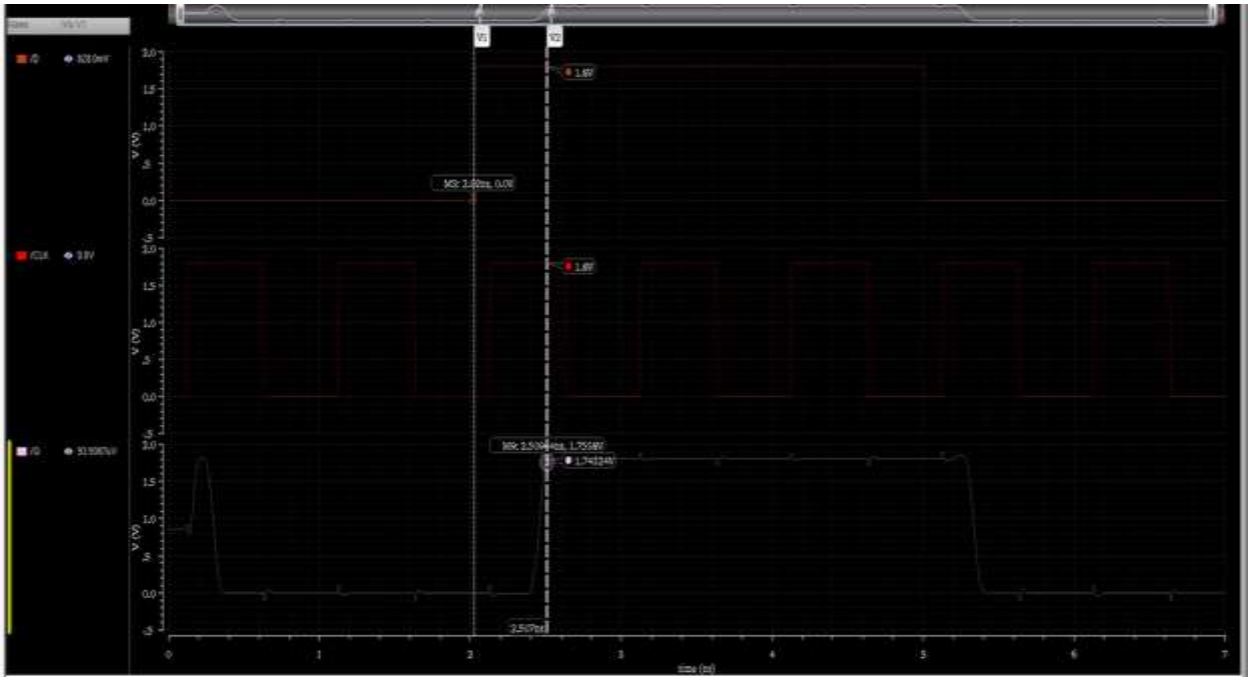
Optimized Delay of D Flip Flop – Calculation

Clock to Q Delay



t_CQ = 3.221 ns - 3.02ns = 200 ps

D to Q Delay



t_DQ = 2.508 ns - 2.024ns = 484 ps

Delay Parameter	Before Optimization	After Optimization
Clock to Q Delay	600 ps	200 ps
D to Q Delay	890 ps	484 ps

C. Power Optimization

Power optimization using Clock Gating

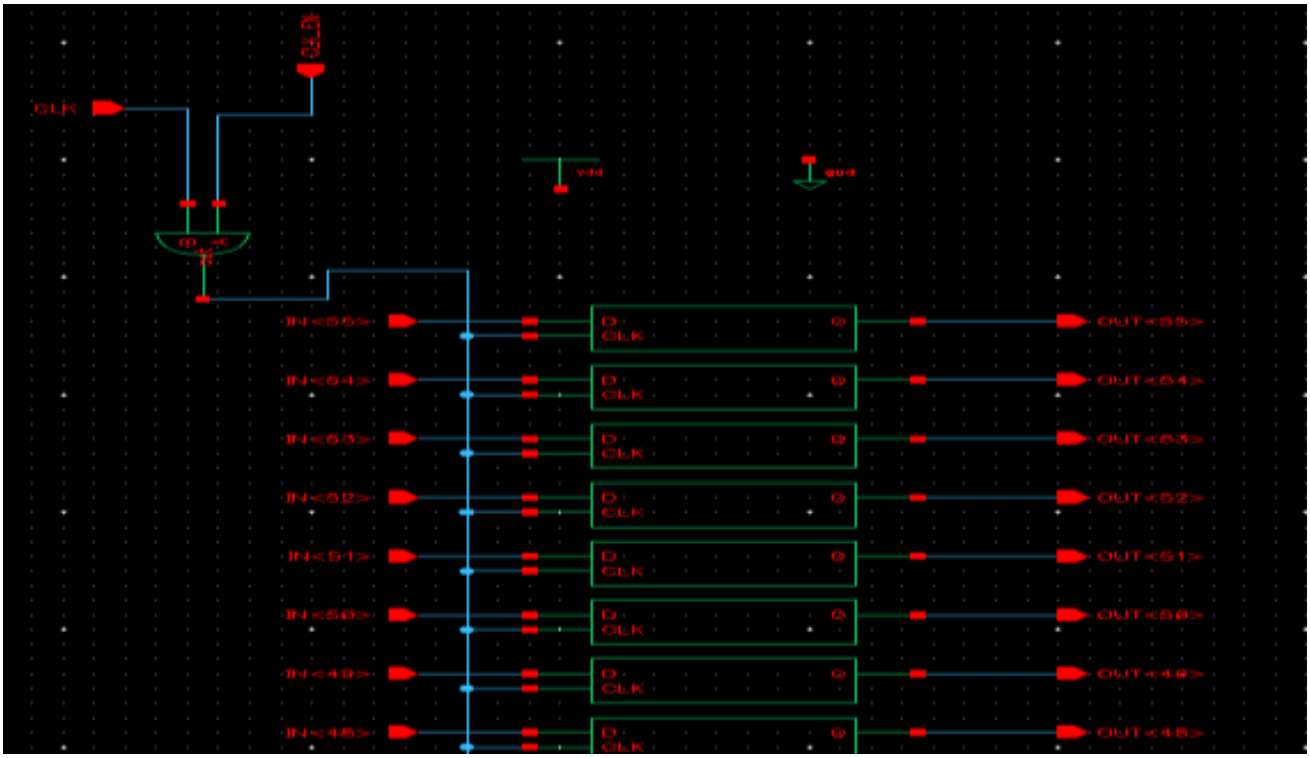
A lot of power is saved in the circuit using the concept of Power gating learnt in one of the units of Power Optimization.

The Clock signal is masked with Enable signal by AND operation therefore only if Clock Enable is 1, Clock is applied else if Clock Enable is 0, the Clock is Masked through the circuit.

This is done in following two cases.

1. The D flip flop hold their previous values and the output remains steady and doesn't change at all.

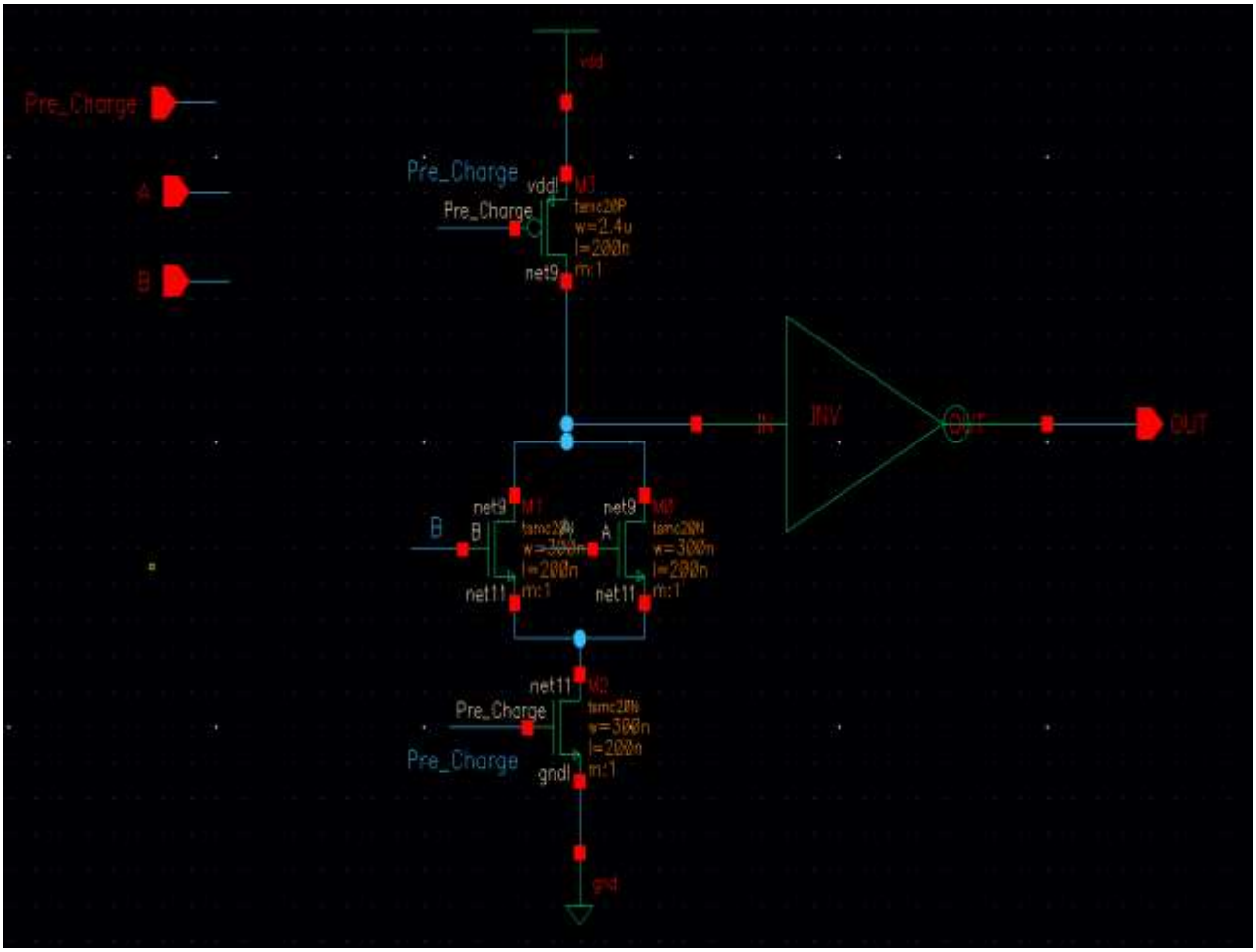
2. There may be a situation where we might not need the output of the data-path. In that case masking the clock will save power both in the flip flops and data-path.



D. Dynamic Logic

Since most of the operations are arithmetic and logical operations therefore we tried to implement Dynamic logic in the ALU stage. Dynamic logic helps to make the design faster by reducing the delay. But it affects the power consumption and makes the circuit even more power hungry.

Implementation of Dynamic Logic in OR circuit



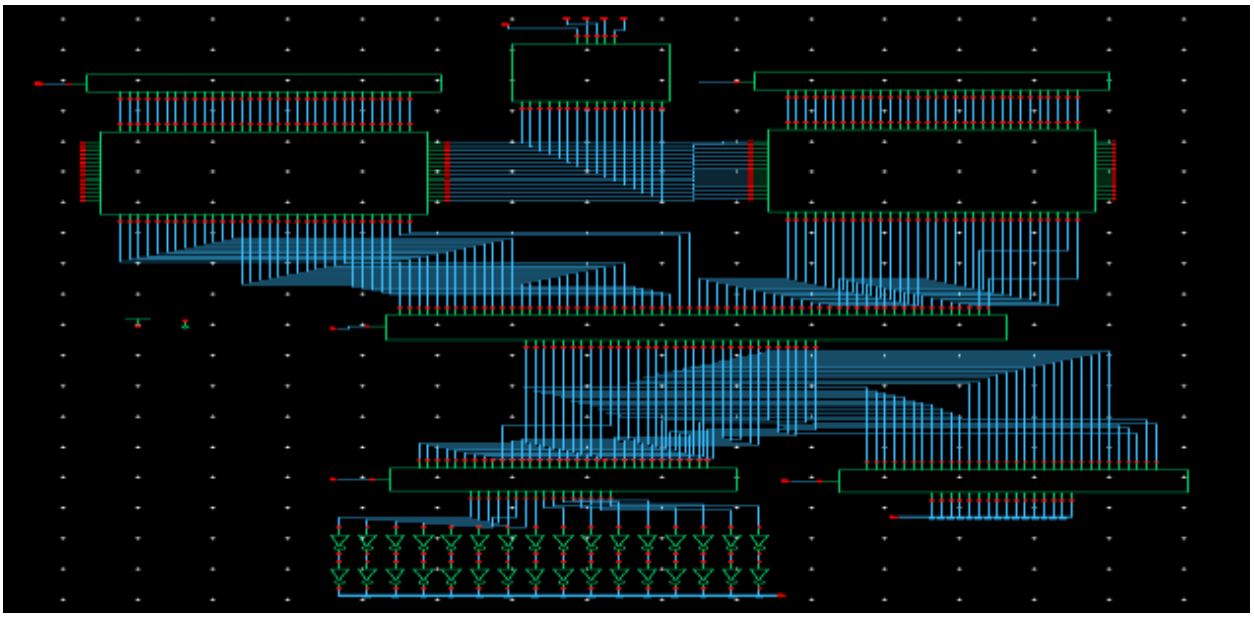
Note: We discontinued using Dynamic logic in the rest of the circuits because it posed a lot of issues in clocking the circuit and also was producing glitches at the output.

APPENDIX

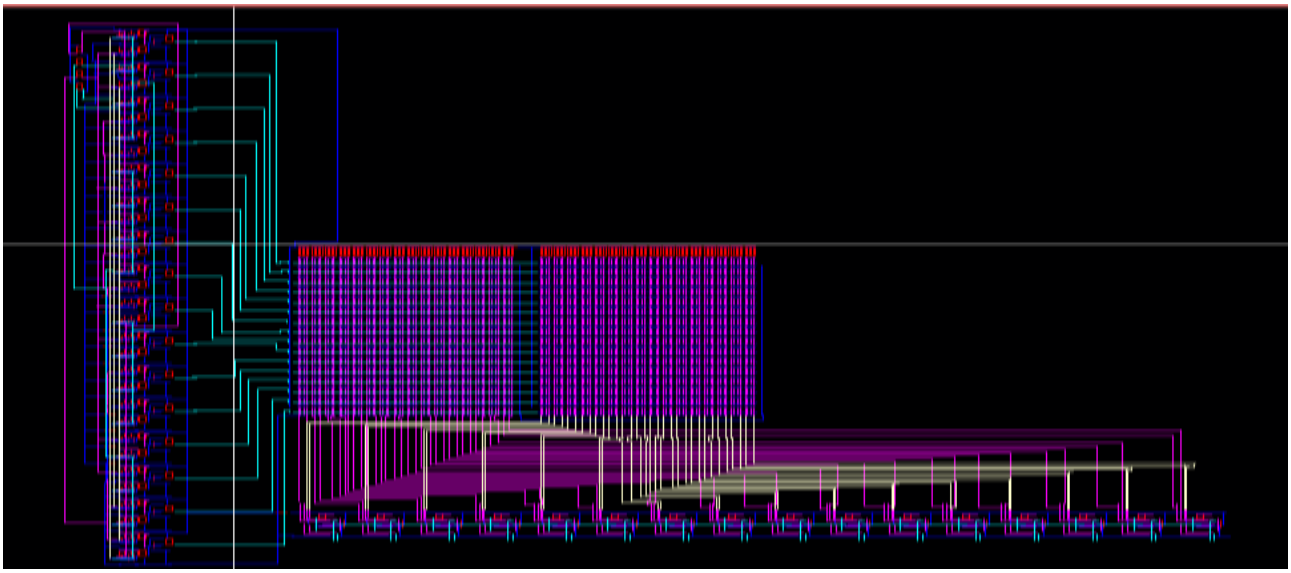
<u>Schematics, Layouts and LVS Matches of Individual components</u>

SRAM

Schematic

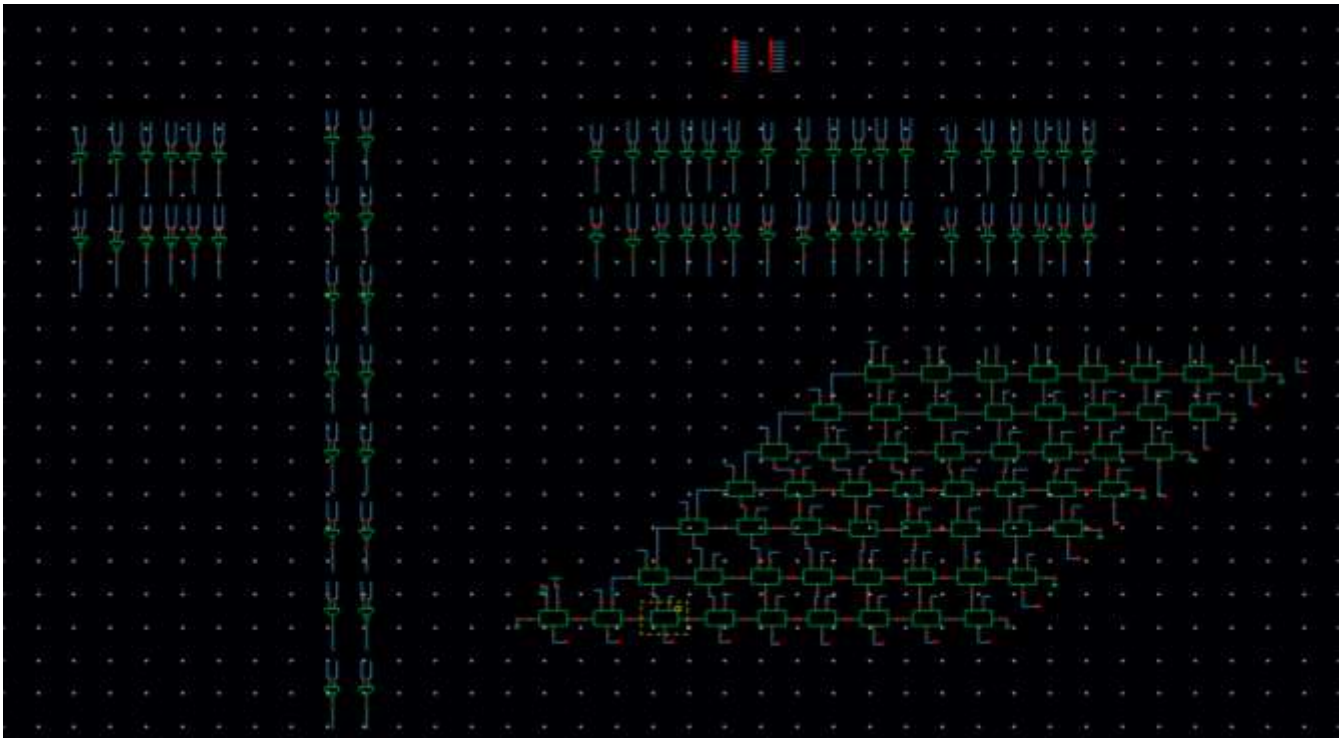


Layout

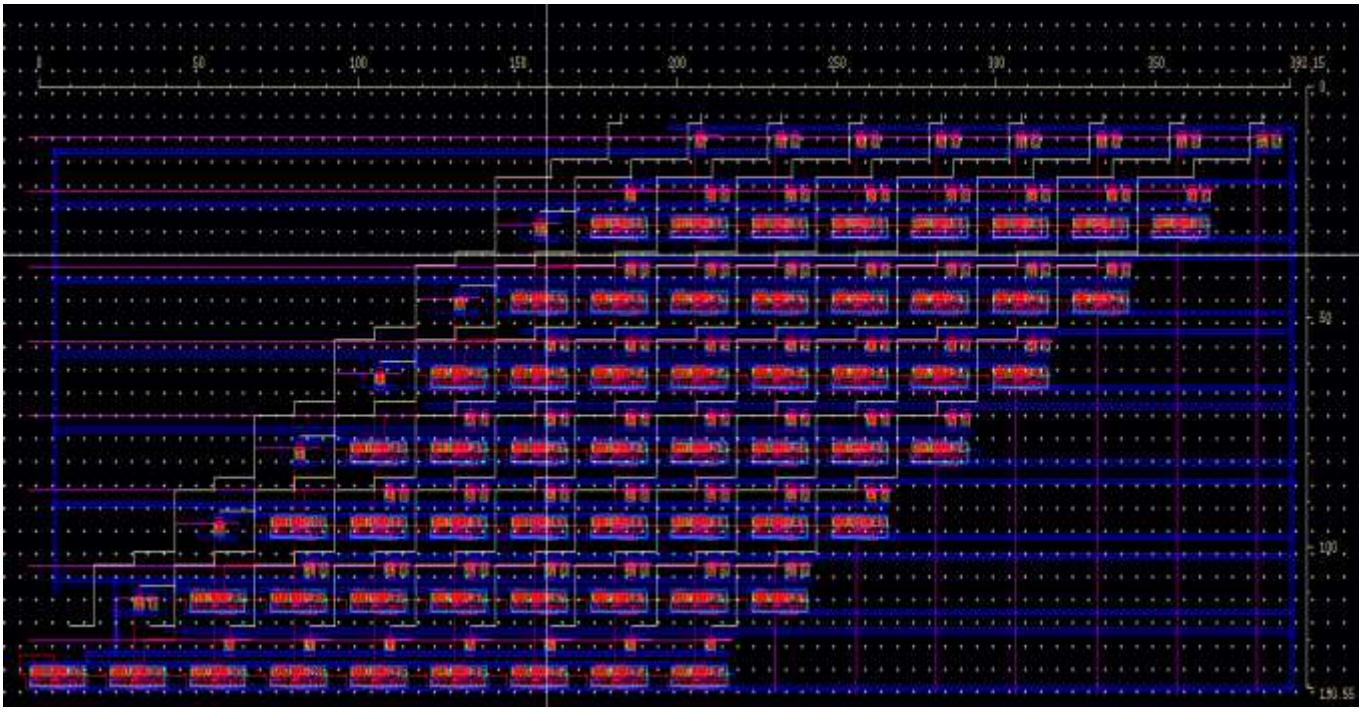


MULTIPLIER

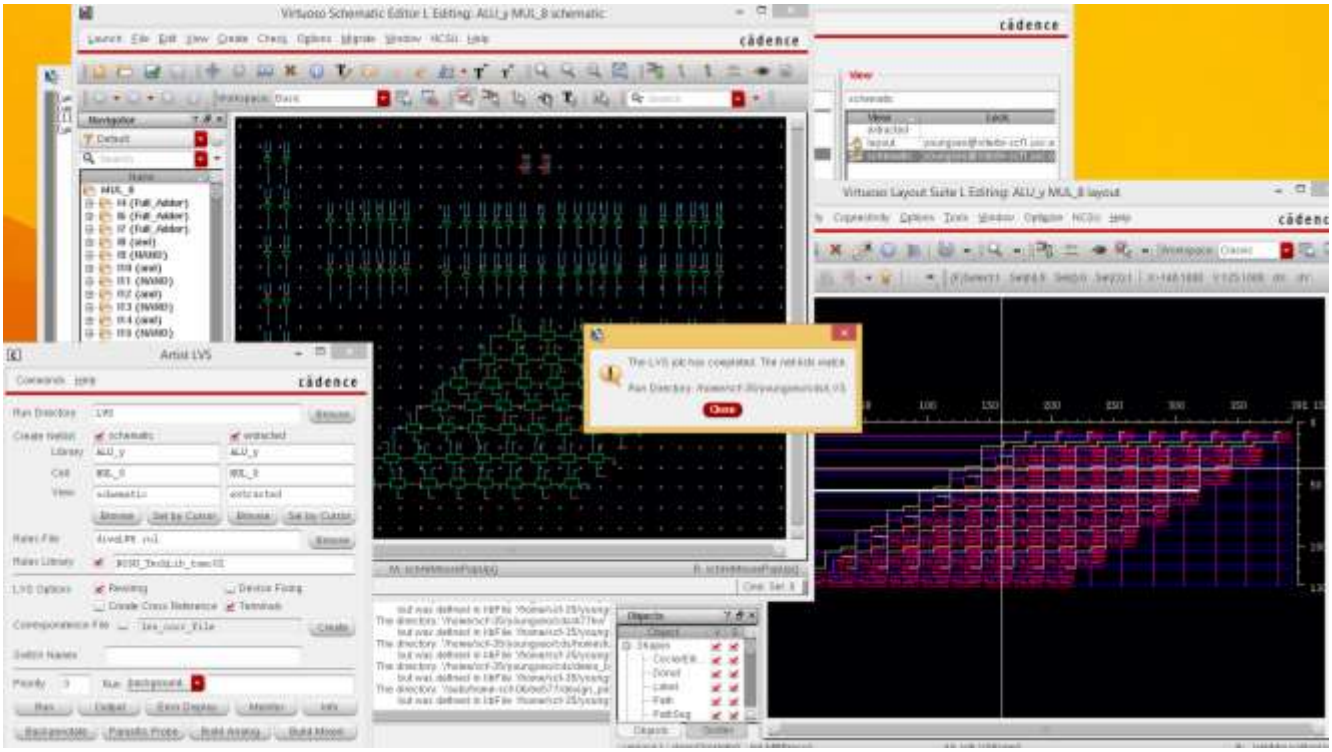
Schematic



Layout

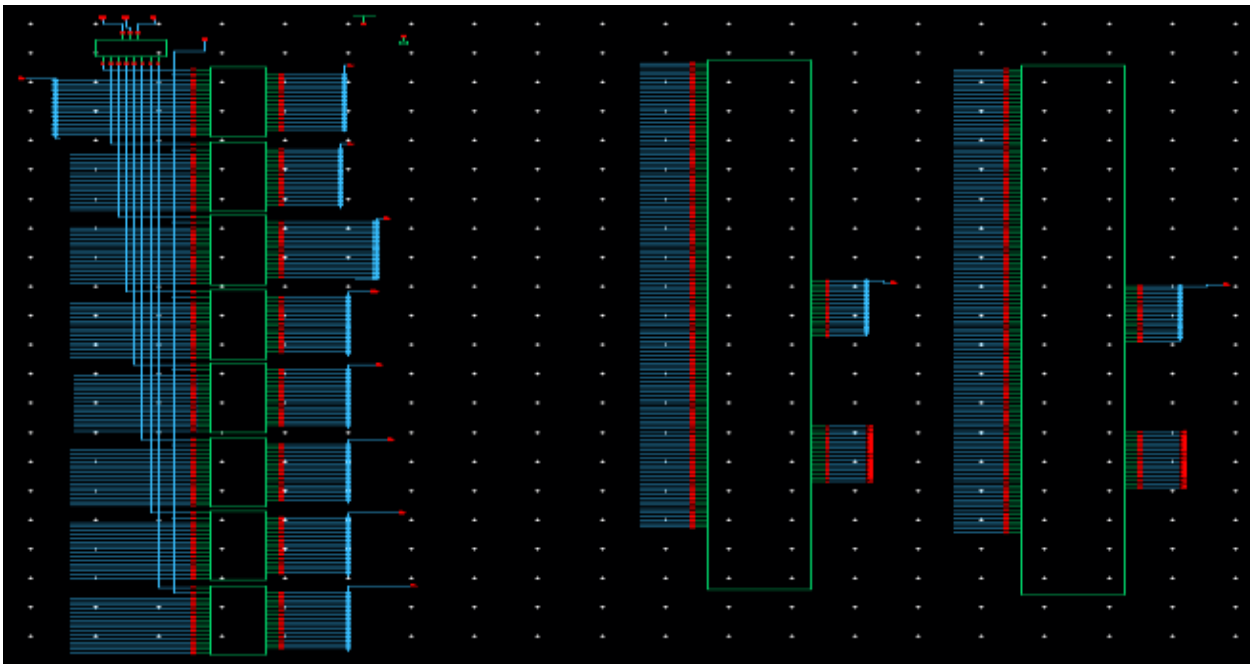


LVS Match

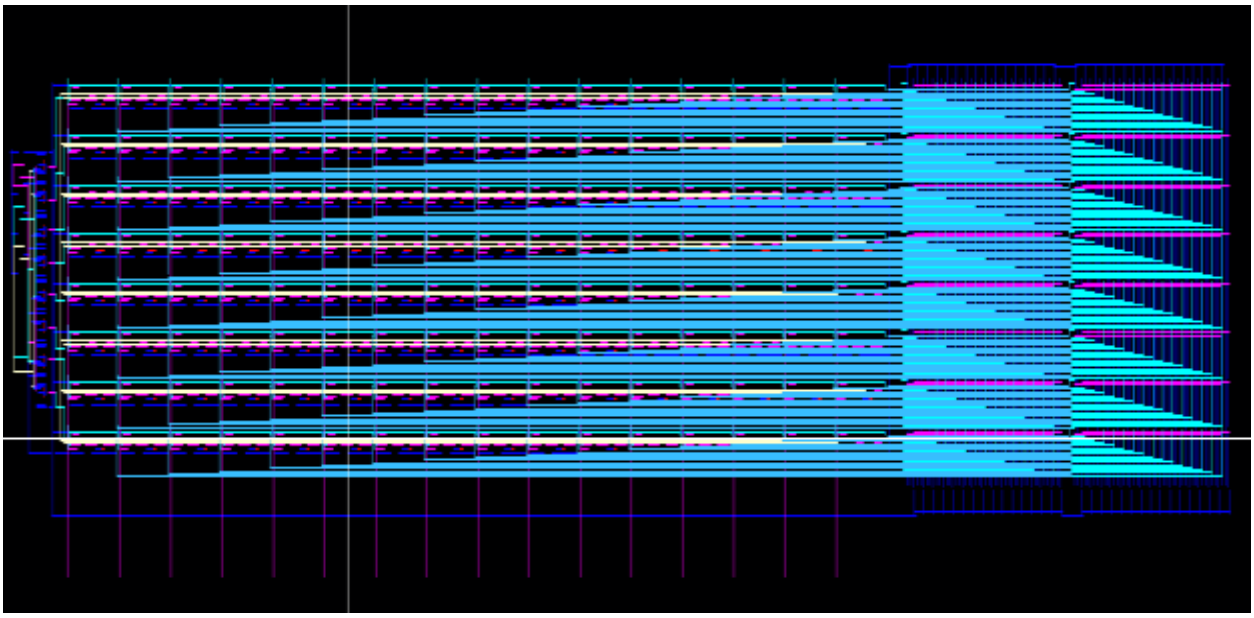


REGISTER FILE

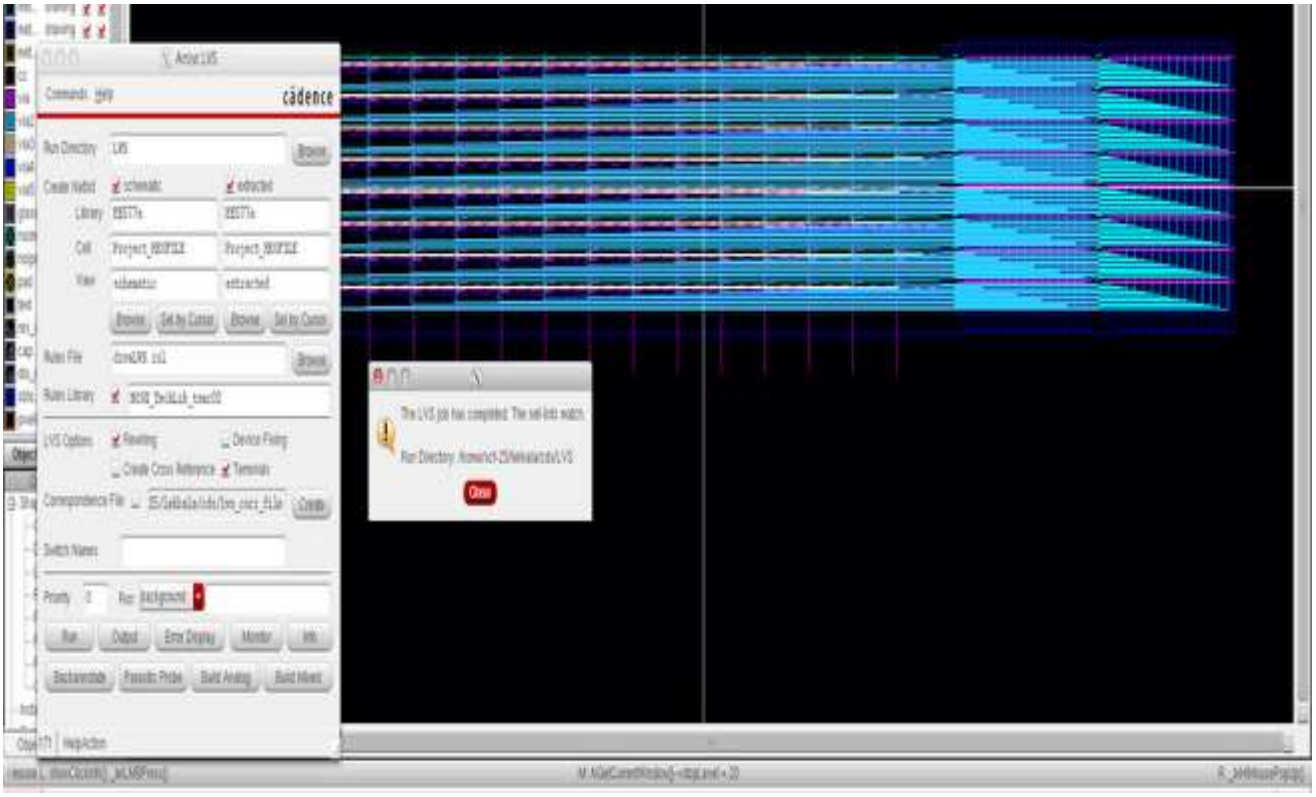
Schematic



Layout

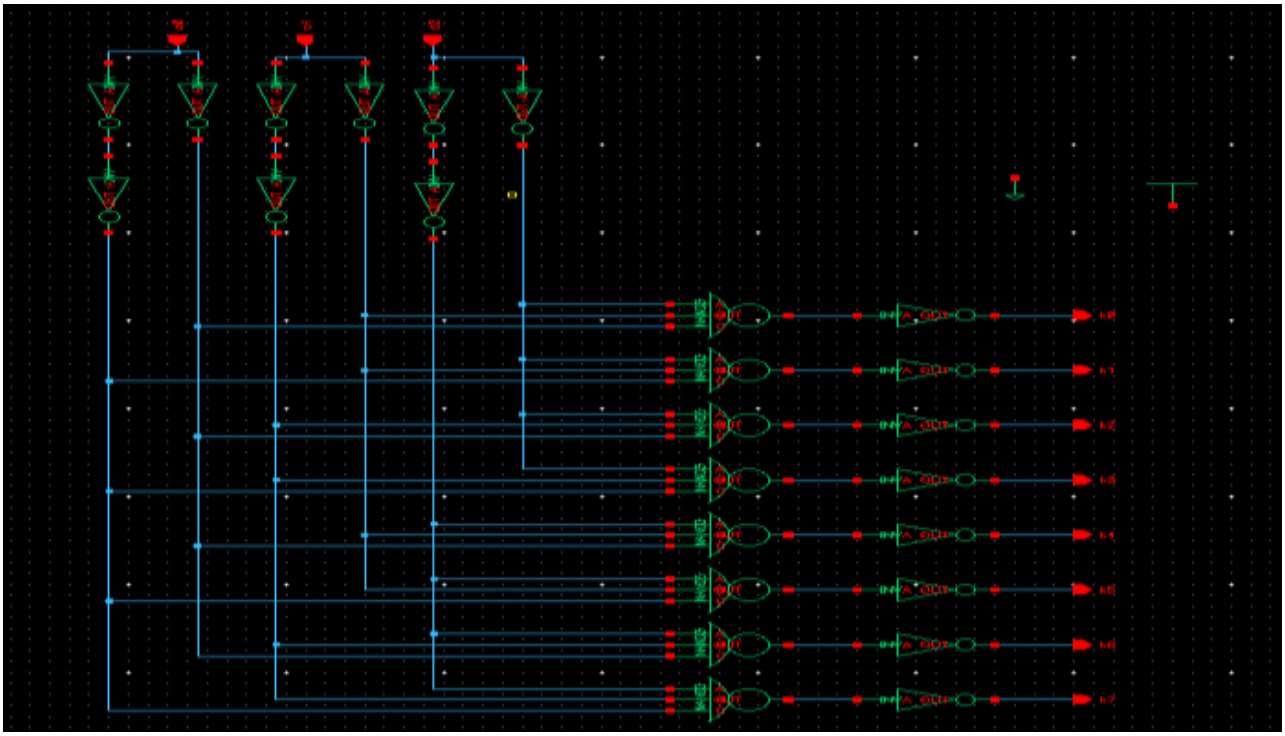


LVS Match

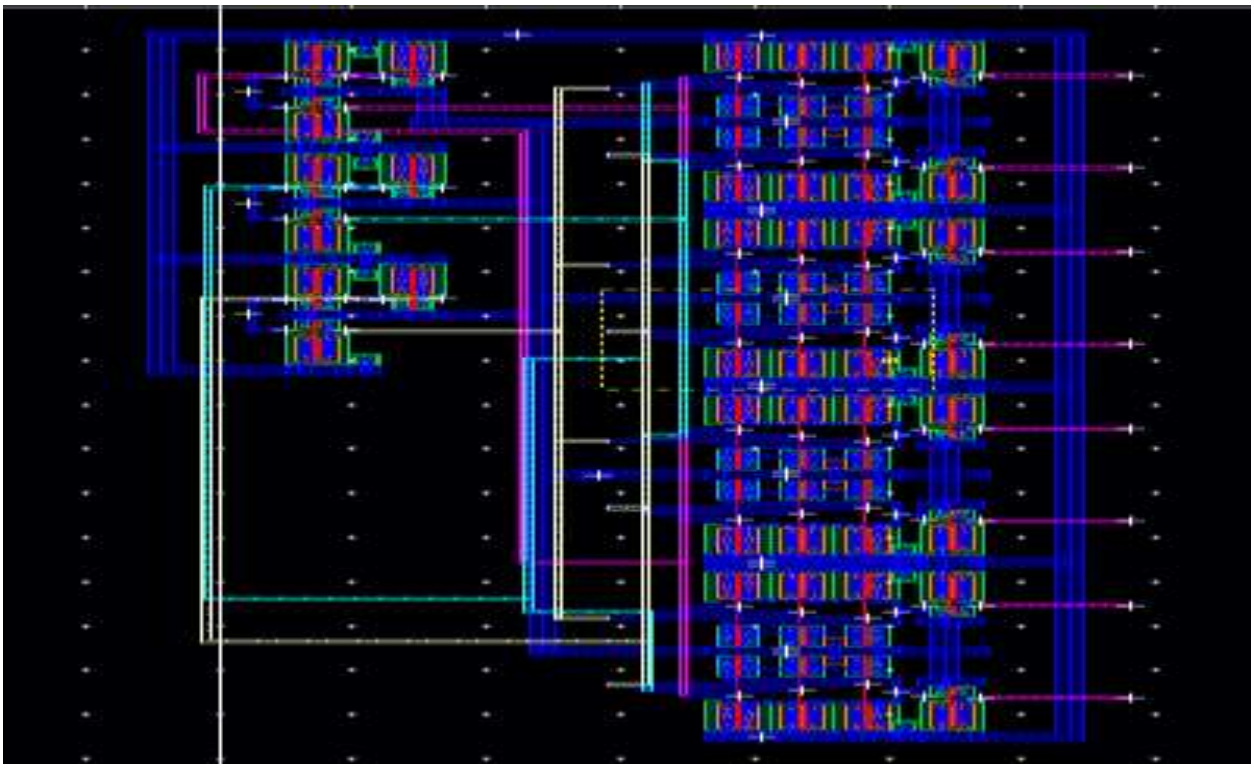


DECODER

Schematic

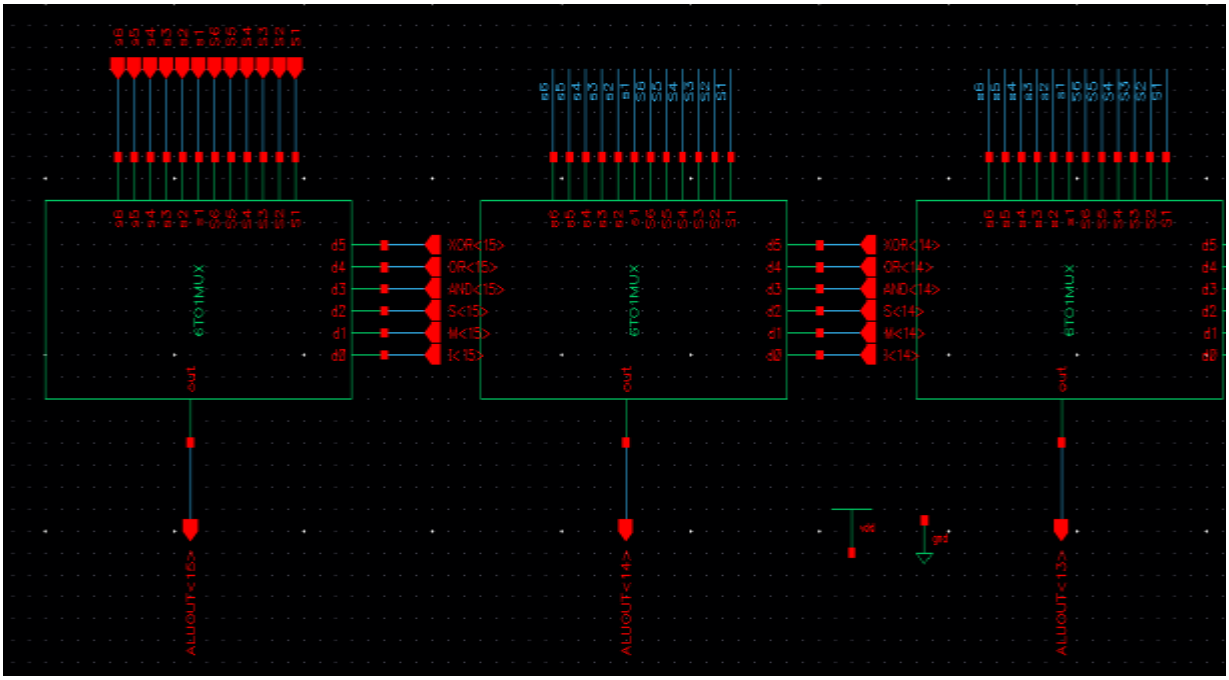


Layout

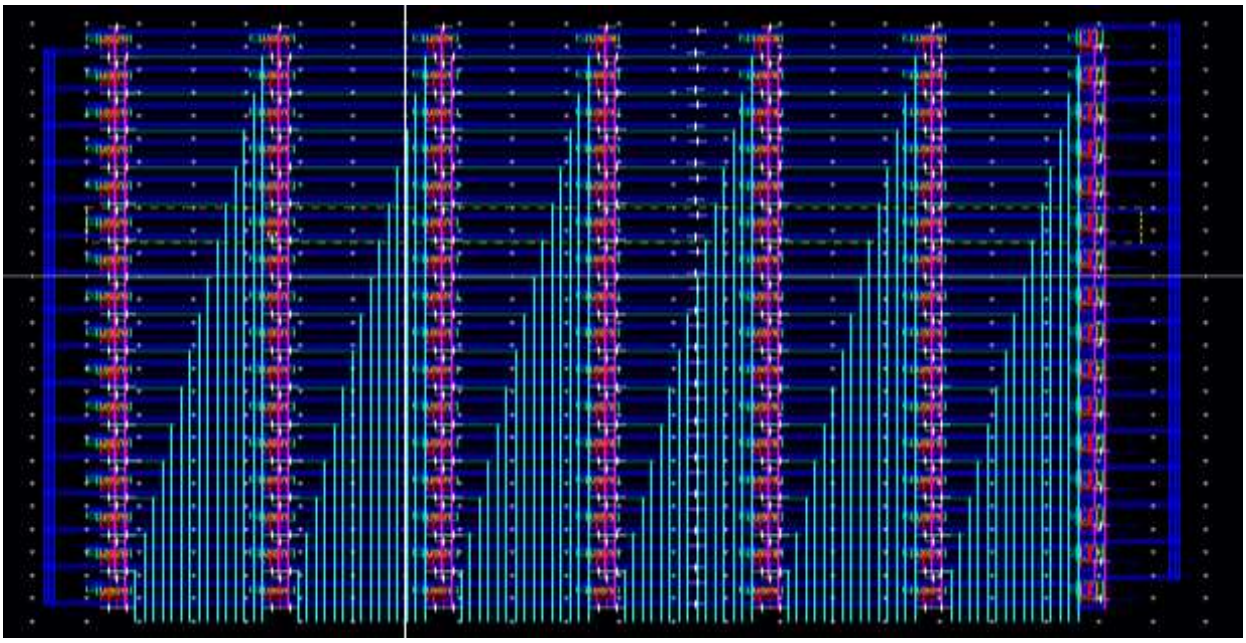


MULTIPLEXERS

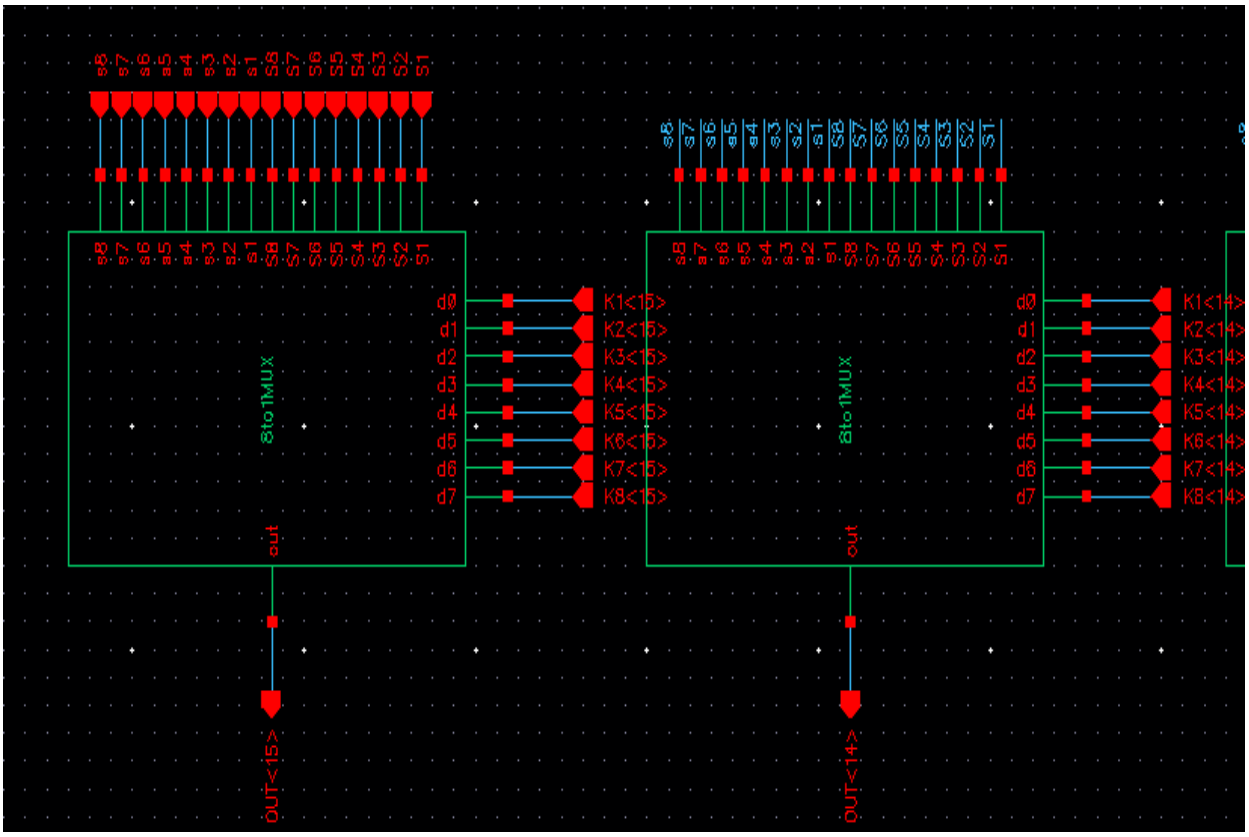
Schematic of 6 TO 1 MUX



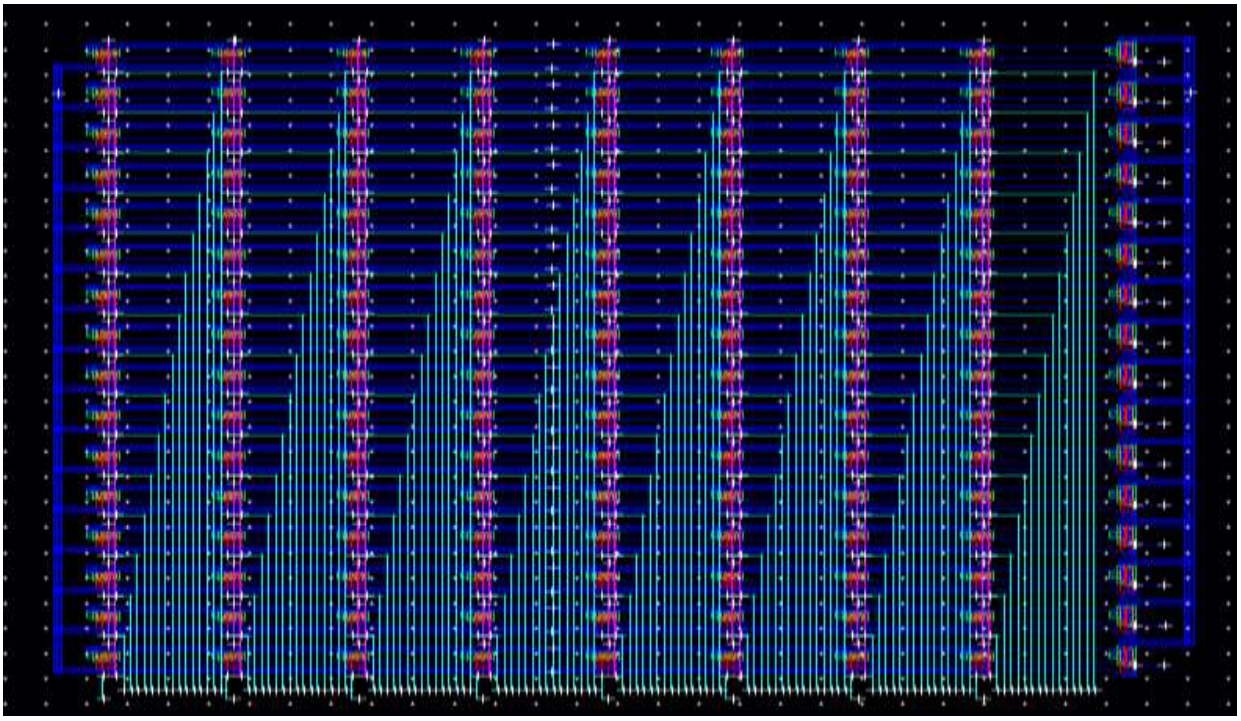
Layout



Schematic of 8 TO 1 MUX

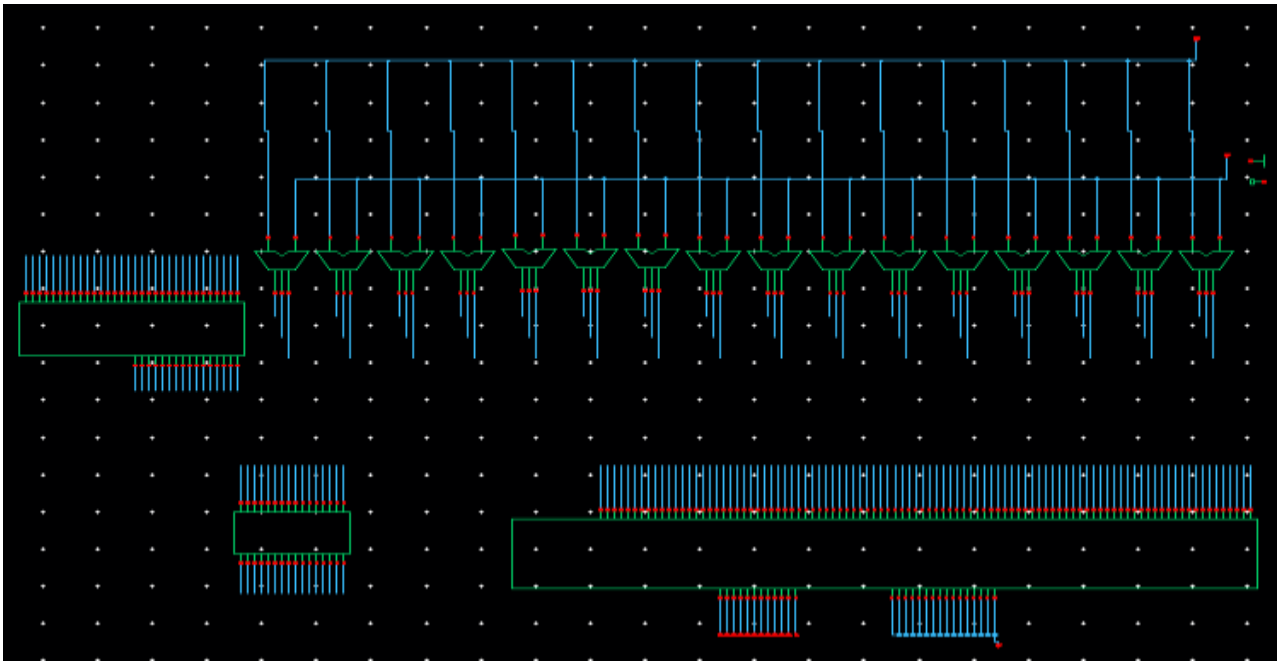


Layout



ALU (Including MUL, ADD, AND, OR, XOR)

Schematic



Layout