EE 577A Spring 2020

Professor: Peter A. Beerel

Final Project
Design of a General-Purpose Microprocessor Using Software and Hardware Components

Phase III: Final Report

Jianqi Zhang<jianqiz@usc.edu>, 1052509893 Yaxin Deng<yaxinden@usc.edu>, 4704064815

2020/05/08

Abstract

Phase I

In this part, we design the 5 stages pipeline CPU. There are five stages in the pipeline structure of this project, namely, IF, ID, EX, MEM, and WB. The front-end Python script will function as IF and decode logic in the ID stage, whereas the rest of the stages and Register File (RF) will be implemented in hardware in Cadence. Also, we can simulate the final result of the instruction by checking the value stored in register file and memory. We use the testbench generated by Python script to simulate the design and achieve 1ns clock period execution.

Phase II

In this part, we optimize the total design and achieve 800 ps clock period execution. We achieve Out-of-Order multiplier execution and divided MUL/MULI instruction into 4 periods. We optimize Register File design by using clock gating. We also optimize ALU and memory power consumption by adding enable signals for modules. In addition, we explore DFF optimization and make a trade-off between DFF's performance, area and power.

Phase III

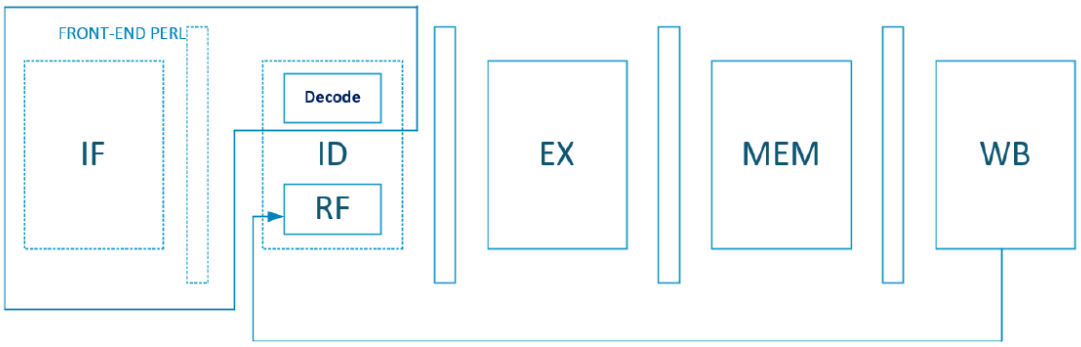
In this part, we add a clock tree to simulate the real-life execution. We also design the control logic of memory and implement it by hardware. Finally, we finish the layout of out-of-order multiplier and SRAM and estimate other modules' area by putting all components together without interconnection. The final result of our design is shown below:
Clock Cycle: 800 ps
Execution Time: 30.78ns
Power Consumption: 1.134×10^{-3}W
Total area (Estimated): $77.17\mu m \times 87.69\mu m = 6767.0373\mu m^2$

Contents

Part 1 Overview and Pipeline Design

1.1 Overview

There are five stages in the pipeline structure of this project, namely, IF, ID, EX, MEM, and WB. The front-end Perl/Python script will function as IF and decode logic in the ID stage, whereas the rest of the stages and Register File (RF) will be implemented in hardware in Cadence.



In IF (Instruction Fetch) stage and part of ID (Instruction Decode) stage, we use python script to analysis the given instructions and generate control signals as inputs of the cadence hardware design. Also, we can simulate the final result of the instruction by checking the value stored in register file and memory.

The input signals of CPU are listed in the table.

SIGNAL	DESCRIPTION	RADIX
RESET	Reset	1
CLK	Clock	1
WB	Write Back Control	1
MEMREAD	Memory Read Enable	1
MEMWRITE	Memory Write Enable	1
REG_DST	Register Destination Control	1
ALU_SRC	ALU Source Control	1
ALU_OP	ALU Operation Control	3
RS	RS #	3
RT	RT #	3
IMME_NUM	Immediate Number	16
ADDR	Memory Address	5

And the overview schematic is shown below.

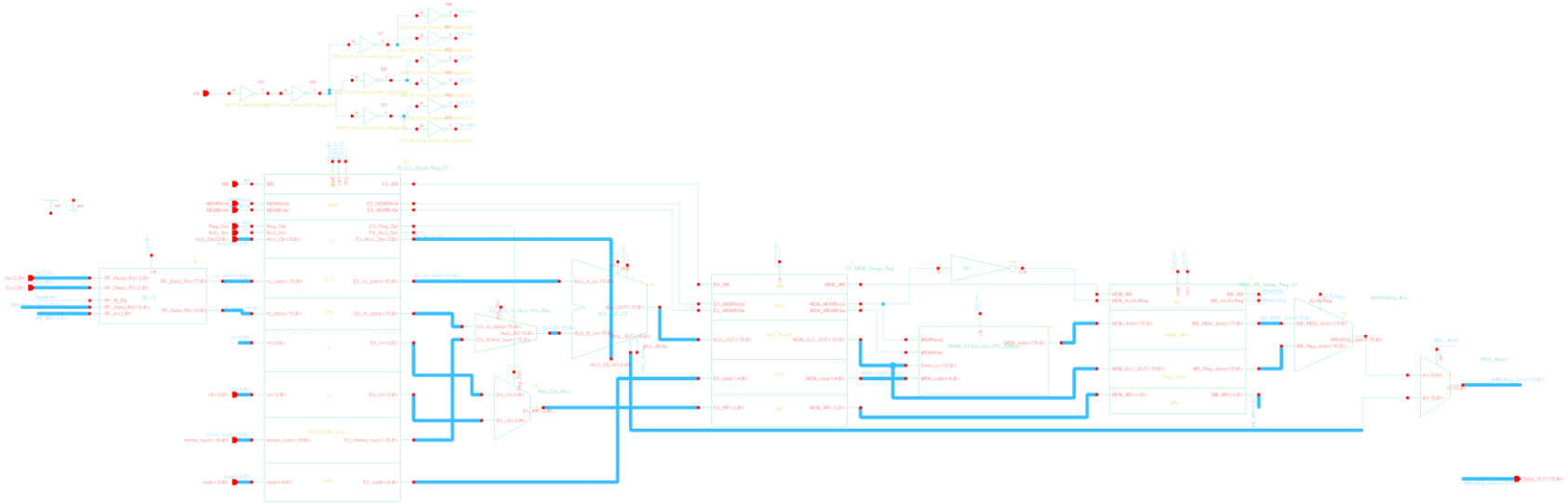


Figure 1.1.1 5 Stage Pipeline CPU Schematic

1.2 Instruction Fetching & Decoding Stage

In this stage, python scripts fetch the given ISA, and then decode the instruction into input signals. According to rs # and rt # from input, Register File will read the RS data and RT data. Also, WB stage maybe write data into register file at same time.

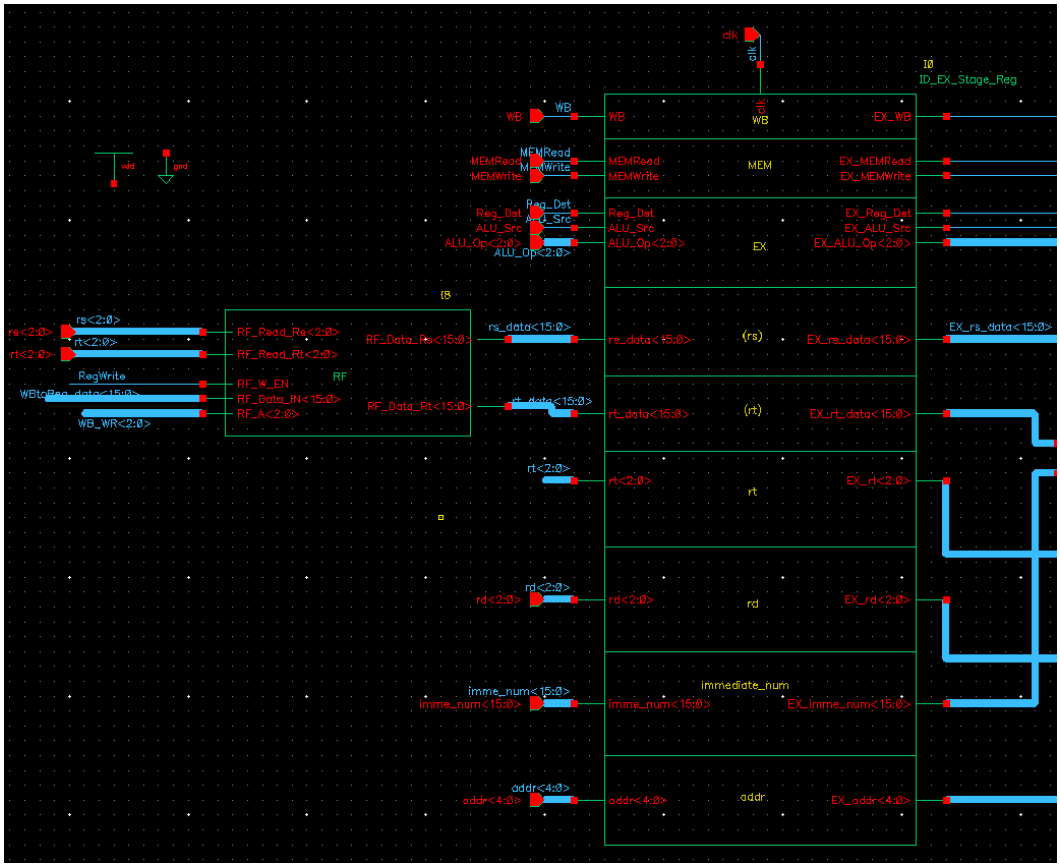


Figure 1.2.1 IF & ID Stage Schematic

1.3 Execution Stage

In EX stage, ALU is the main part of design. Several signals to ALU will be carried from input to EX stage. ALU_Op will determine the operation type of this instruction. ALU_Src serves as controlling the second source of ALU, which is one of RT value or immediate number. Write back register # will also produced in this stage, which is controlled by Reg_Dst signal to choose RT # or RD # for writing bace to register file.

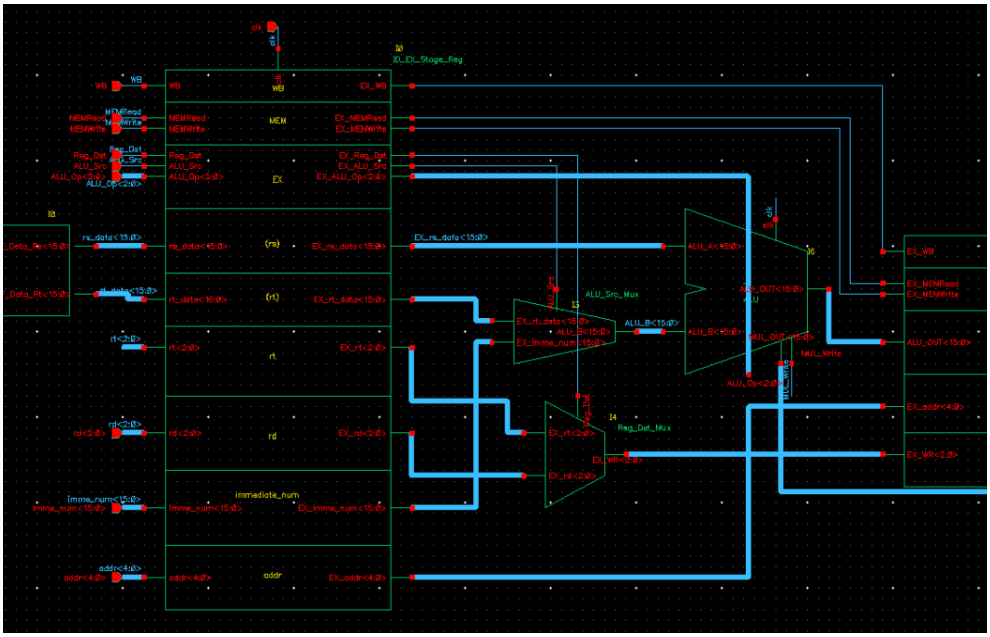
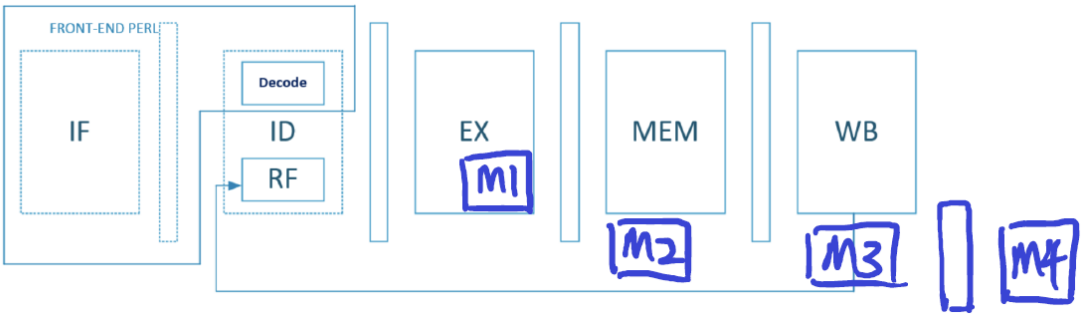
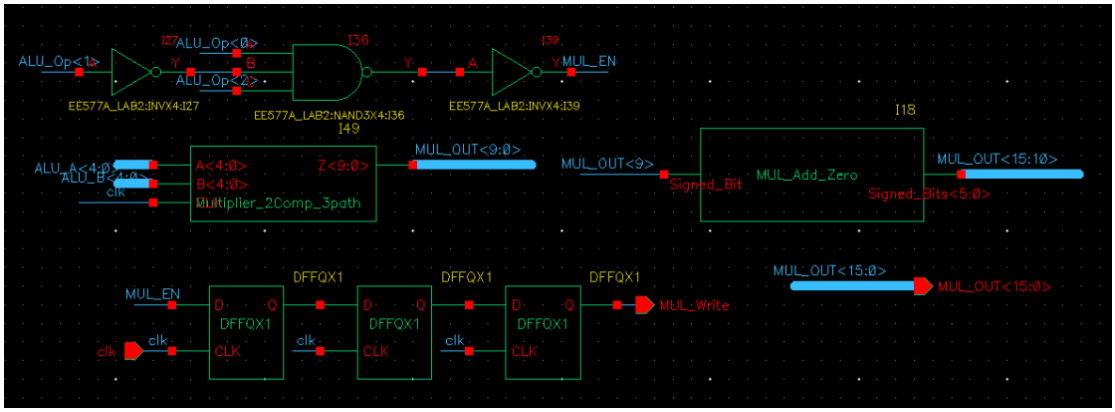


Figure 1.3.1 ID & EX Stage Schematic

Here, we use Out-of-Order execution for MUL/MULI commands to improve the performance. As you can see in the figure below, 5-bit multiplier is a separate pipeline which costs 4 clocks to execution. In order to keep the order of commitment, we need to add a fake 'NOP' instruction after every MUL/MULI instruction. This instruction carry the write register of original MUL/MULI so that the computation result will write back to register file.





1.4 Memory Stage

SRAM is the main part of this stage. MEMRead and MEMWrite serve as enabling read and write.

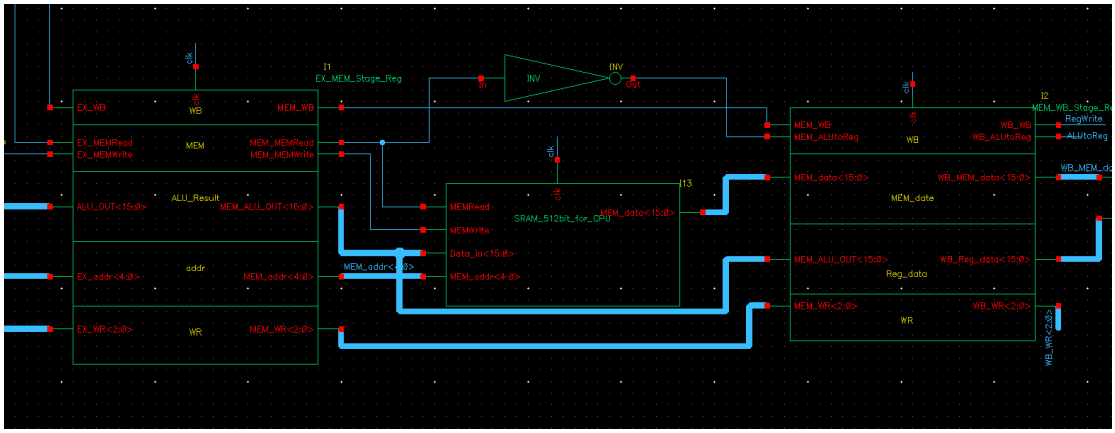


Figure 1.4.1 EX & MEM Stage Schematic

1.5 Write Back Stage

At this stage, data produced by ALU will be written back to Register File. A mux controlled by ALUtoReg will choose write back data from memory or data from ALU. Another mux controlled by MUL_Write signal, which produced in the EX stage to mark MUL/MULI instruction and write back multiplier computation result into register file.

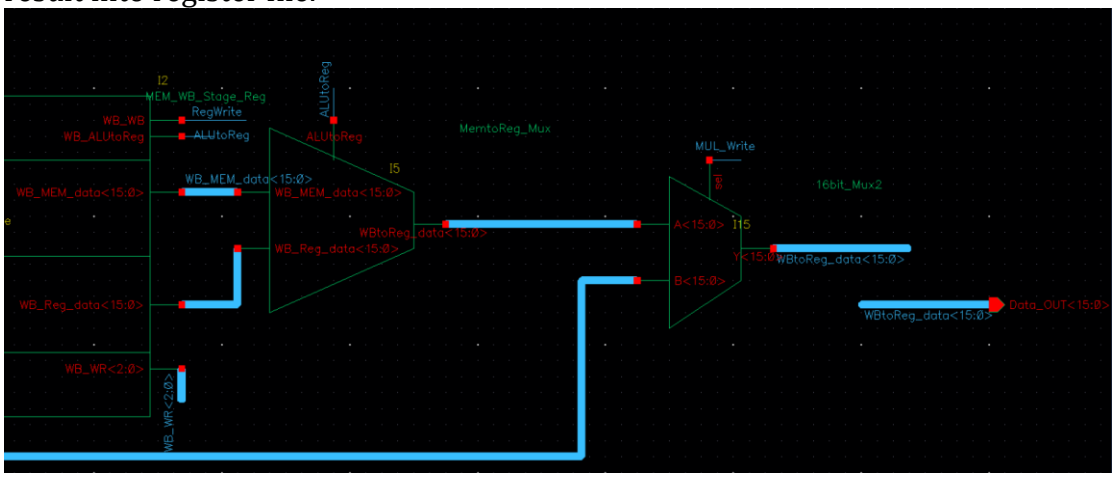


Figure 1.5.1 MEM & WB Stage Schematic

Part 2 Instruction Fetching & Decoding and Automated Result Verification (Perl/Python Scripting)

2.1 Instruction Decode

In python script, we define a class Inst, which describes behave and attribute of an instruction.

```
class Inst(object):
    def __init__(self):
        self.type = 'NOP'
        self.str_inst = 'NOP'
        self.CtrlSignal = {
            'WB':'0', 'MEMRead':'0', 'MEMWrite':'0', 'Reg_Dst':'0', 'ALU_Src':'0', 'ALU_Op':'000'
            }
        self.reg = {
                'rs':0, 'rt':0, 'rd':0
            }
        self.ISA_Type = {
            'LOAD':self.LOAD, 'STORE':self.STORE, 'NOP':self.NOP
            }
        self.imme_num = 0
        self.addr = '00'
        self.isimme = False
        self.bl = 1
        self.burstOp_flag = False
        self.error_flag = False
        self.error_code = ''
        self.error_meg = ''
        self.read_reg_1 = None
        self.read_reg_2 = None
        self.write_reg = None
        self.imme_num_list = []
        self.reg_lists = []
```

At first, we split the ISA by '\n' and for every instruction, we extract the type, operations and numbers, respectively, which is implemented by character matching.

```
def format(self):
    self.list_str_inst = self.str_inst.split(' ')
    for op in self.list_str_inst:
        if op.isalpha() is True:
            if op.endswith('I') is True:
                self.isimme = True
                self.type = op.strip('I')
            else:
                self.isimme = False
                self.type = op
            if op == 'SFR' or op == 'SFL':
                self.isimme = True
        if op.isdigit() is True:
            self.bl = int(op)
        if op.startswith('$') is True:
            self.reg_lists.append(int(re.findall(r"\d", op)[0]))
        if op.endswith('H') is True:
            self.addr = op.strip('H')
        if op.startswith('#') is True:
            self.imme_num_list.append(int(op.strip('#'),16))
    if self.type is None:
        self.error_flag = True
        self.error_code = '002'
        self.error_meg = 'Error' + str(self.error_code) + ': Command ' + self.str_inst + ' is an unsupported operation.'
```

According to the type of instruction, we can generate the corresponding signals.

```
def LOAD(self):
    self.reg['rt'] = self.reg_lists[0]
    self.write_reg = self.reg['rt']
    self.CtrlSignal['WB'] = '1'
    if self.isimme is True:#LOADI $R #xxxx              Load xxxx into register $R
        self.CtrlSignal['ALU_Src'] = '1'
        self.CtrlSignal['ALU_Op'] = '001'  # OR
        self.imme_num = self.imme_num_list[0]
    else: #LOAD $R xxH                  Load word xxH into register $R
        self.CtrlSignal['MEMRead'] = '1'
```

```
def STORE(self):
    self.CtrlSignal['MEMWrite'] = '1'
    self.CtrlSignal['ALU_Op'] = '001'  # OR
    if self.isimme is True:#STOREI {bl} xxH #xxxx {#xxxx}          Store xxxx into word xxH
        self.CtrlSignal['ALU_Src'] = '1'
        self.imme_num = self.imme_num_list[0]
        if self.bl == 1:
            pass
        elif self.bl == 2:
            self.burstOp_flag = True
            if bin(int((self.addr),16)).endswith('0') is False:
                self.error_flag = True
                self.error_code = '001'
                self.error_meg = 'Error' + str(self.error_code) + ': Command ' + self.str_inst + ' is not aligned properly.'

        elif self.bl == 4:
            self.burstOp_flag = True
            if bin(int((self.addr),16)).endswith('00') is False:
                self.error_flag = True
                self.error_code = '001'
                self.error_meg = 'Error' + str(self.error_code) + ': Command ' + self.str_inst + ' is not aligned properly.'
            else:
                self.error_flag = True
                self.error_code = '000'
                self.error_meg = 'Error' + str(self.error_code) + ': Command ' + self.str_inst + ' has invalid burst length.'
    else:
        self.reg['rs'] = self.reg_lists[0]
        self.read_reg_1 = self.reg['rs']
```

```
def R_TYPE(self):
    self.CtrlSignal['WB'] = '1'
    self.reg['rd'] = self.reg_lists[0]
    self.reg['rs'] = self.reg_lists[1]
    self.CtrlSignal['Reg_Dst']  = '1'
    self.read_reg_1 = self.reg['rs']
    self.write_reg = self.reg['rd']
    if self.isimme is True:
        self.imme_num = self.imme_num_list[0]
        self.CtrlSignal['ALU_Src'] = '1'
    else:
        self.reg['rt'] = self.reg_lists[2]
        self.read_reg_2 = self.reg['rt']
    if self.type == 'ADD':
        self.CtrlSignal['ALU_Op'] = '100'
    elif self.type == 'MUL':
        self.CtrlSignal['ALU_Op'] = '101'
    elif self.type == 'AND':
        self.CtrlSignal['ALU_Op'] = '000'
    elif self.type == 'OR':
        self.CtrlSignal['ALU_Op'] = '001'
    elif self.type == 'MIN':
        self.CtrlSignal['ALU_Op'] = '111'
    elif self.type == 'SFL':
        self.CtrlSignal['ALU_Op'] = '010'
    elif self.type == 'SFR':
        self.CtrlSignal['ALU_Op'] = '011'
    else:
        self.error_flag = True
        self.error_code = '002'
        self.error_meg = 'Error' + str(self.error_code) + ': Command ' + self.str_inst + ' is an unsupported operation.'
```

2.2 Burst Operation

Then, we create class ISA, which describes a set of instruction. In this class, we will solve the burst operation and data dependency issue.

For burst operation, we need to insert 1 or 3 STORE instructions after the original STORE. When the instruction is decoded, we will give a flag of burst operation. And we traverse the instructions set, we can find this flag and add another STORE instruction after that.

```
    def burstOperation(self):
        for inst in self.list_inst:
            rank = self.list_inst.index(inst)
            if inst.burstOp_flag is True:
                if inst.bl == 2:
                    self.insertSTOREI(rank+1, inst.imme_num_list[1], hex(int(inst.addr,16)+1)[2:])
                elif inst.bl == 4:
                    self.insertSTOREI(rank+1, inst.imme_num_list[1], hex(int(inst.addr,16)+1)[2:])
                    self.insertSTOREI(rank+2, inst.imme_num_list[2], hex(int(inst.addr,16)+2)[2:])
                    self.insertSTOREI(rank+3, inst.imme_num_list[3], hex(int(inst.addr,16)+3)[2:])
```

2.3 Data Dependency

When the two instructions are so closed and one depends on the other one, we need to insert NOP instruction to avoid dependency. In this function, we traverse all instructions in ISA. For every instruction, we check the very next of it and the next of next of it. Because if an instruction is going to write into register and another instruction is going to read, the later one should read after the first finishing. We should check the read register number, the write register number

and write enable signal to determine whether to add a NOP.

```
def solvedependency(self):
    rank = -1
    while rank < (len(self.list_inst) - 2):
        rank += 1
        inst = self.list_inst[rank]
        next_ins = self.list_inst[rank+1]
        if inst.type == 'STORE' and next_ins.type == 'LOAD':
            self.insertNOP(rank+1)
            continue
        if inst.CtrlSignal['WB'] == '1':
            if inst.write_reg is None:
                continue
            elif inst.write_reg == 0:
                continue
            else:
                if next_ins.read_reg_1 is not None :
                    if next_ins.read_reg_1 == inst.write_reg:
                        self.insertNOP(rank+1)
                        self.insertNOP(rank+1)
                    elif next_ins.read_reg_2 is not None:
                        if next_ins.read_reg_2 == inst.write_reg:
                            self.insertNOP(rank+1)
                            self.insertNOP(rank+1)
                if self.list_inst[rank+2] is not None:
                    next_next_ins = self.list_inst[rank+2]
                    if next_next_ins.read_reg_1 is not None:
                        if next_next_ins.read_reg_1 == inst.write_reg:
                            if next_ins.type != 'NOP':
                                self.insertNOP(rank+1)
                        elif next_next_ins.read_reg_2 is not None:
                            if next_next_ins.read_reg_2 == inst.write_reg:
                                if next_ins.type != 'NOP':
                                    self.insertNOP(rank+1)
        else:
            continue
```

2.4 Error Report

In instruction decode stage, we will judge the bl and address of STORE instruction.

```
    if self.bl == 1:
        pass
    elif self.bl == 2:
        self.burstOp_flag = True
        if bin(int((self.addr),16)).endswith('0') is False:
            self.error_flag = True
            self.error_code = '001'
            self.error_meg = 'Error' + str(self.error_code) + ': Command ' + self.str_inst + ' is not aligned properly.'

    elif self.bl == 4:
        self.burstOp_flag = True
        if bin(int((self.addr),16)).endswith('00') is False:
            self.error_flag = True
            self.error_code = '001'
            self.error_meg = 'Error' + str(self.error_code) + ': Command ' + self.str_inst + ' is not aligned properly.'
        else:
            self.error_flag = True
            self.error_code = '000'
            self.error_meg = 'Error' + str(self.error_code) + ': Command ' + self.str_inst + ' has invalid burst length.'
```

2.5 Testbench Generation

For every instruction, we write vector file for one clock.

```
def generateTestbench(self):
    self.vecstr += 'radix 1 1 1 1 1 1 1 1 111 111 111 111 4444 14\n'
    self.vecstr += 'io i i i i i i i i i i i i i i i i\n'
    self.vecstr += 'vname Reset clk WB MEMRead MEMWrite Reg_Dst ALU_Src ALU_Op<[2:0]> rs<[2:0]> rt<[2:0]> rd<[2:0]> imme_num<[15:0]> addr<[4:0]>\n'
    self.vecstr += 'tunit ns\n'
    self.vecstr += 'slope 0.005\n'
    self.vecstr += 'vih 1\n'
    self.vecstr += 'vil 0\n'
    self.vecstr += '\n'
    self.time = 0
    self.vecstr += '{:>4}'.format(str(self.time)) + '\t0\t0\t0\t0\t0\t0\t0\t000\t000\t000\t000\t0000\t00\n'
    self.time += self.clk
    for inst in self.list_inst:
        if inst.error_flag is True:
            print(inst.error_meg)
            continue
        self.vecstr += '{:>4}'.format(str(self.time)) + '\t' + '1\t' + '1\t' + inst.CtrlSignal['WB'] + '\t' + inst.CtrlSignal['MEMRead'] + '\t' + inst.
        CtrlSignal['MEMWrite'] + '\t' + inst.CtrlSignal['Reg_Dst'] + '\t' + inst.CtrlSignal['ALU_Src'] + '\t' + inst.CtrlSignal['ALU_Op'] + '\t' + '
        {:03b}'.format(inst.reg['rs'])  + '\t' + '{:03b}'.format(inst.reg['rt']) + '\t' + '{:03b}'.format(inst.reg['rd']) + '\t' + '{:04x}'.format
        (inst.imme_num).upper() + '\t' + inst.addr + '\t;' + inst.str_inst + '\n'
        self.time += float(self.clk)/2

        self.vecstr += '{:>4}'.format(str(self.time)) + '\t' + '1\t' + '0\t' + inst.CtrlSignal['WB'] + '\t' + inst.CtrlSignal['MEMRead'] + '\t' + inst.
        CtrlSignal['MEMWrite'] + '\t' + inst.CtrlSignal['Reg_Dst'] + '\t' + inst.CtrlSignal['ALU_Src'] + '\t' + inst.CtrlSignal['ALU_Op'] + '\t' + '
        {:03b}'.format(inst.reg['rs'])  + '\t' + '{:03b}'.format(inst.reg['rt']) + '\t' + '{:03b}'.format(inst.reg['rd']) + '\t' + '{:04x}'.format
        (inst.imme_num).upper() + '\t' + inst.addr + '\t;' + inst.str_inst + '\n'
        self.time += float(self.clk)/2
    return self.vecstr
```

The testbench is shown below.

Demo Instruction Set

```
STOREI 0AH #ff0f
STOREI 0BH #0014
STOREI 2 18H #000B #00EE
LOAD $7 0AH
LOAD $2 0BH
LOAD $3 19H
LOAD $4 18H
MUL $5 $7 $2
MUL $6 $3 $5
MIN $1 $7 $2
STORE 1FH $5
STORE 00H $6
SFL $5 $3 #0005
OR $6 $5 $4
ANDI $6 $6 #00CC
ADD $6 $6 $4
STORE 09H $5
STORE 10H $6
LOAD $1 00H
LOAD $1 1FH
LOAD $1 09H
LOAD $1 10H
```

```
radix 1 1 1 1 1 1 1 111 111 111 111 4444 14
io i i i i i i i i i i i i i i i i
vname Reset clk WB MEMRead MEMWrite Reg_Dst ALU_Src ALU_Op<[2:0]> rs<[2:0]> rt<[2:0]> rd<[2:0]> imme_num<[15:0]> addr<[4:0]>
tunit ns
slope 0.01
vih 1
vil 0

   0    0   0   0   0   0   0   0    000 000 000 000 0000    00
0.80    1   1   0   0   1   0   1    001 000 000 000 FF0F    0A  ;STOREI 0AH #ff0f
1.20    1   0   0   0   1   0   1    001 000 000 000 FF0F    0A  ;
1.60    1   1   0   0   1   0   1    001 000 000 000 0014    0B  ;STOREI 0BH #0014
2.00    1   0   0   0   1   0   1    001 000 000 000 0014    0B  ;
2.40    1   1   0   0   1   0   1    001 000 000 000 000B    18  ;STOREI 2 18H #000B #00EE
2.80    1   0   0   0   1   0   1    001 000 000 000 000B    18  ;
3.20    1   1   0   0   1   0   1    001 000 000 000 00EE    19  ;Inserted Burst STOREI Instruction
3.60    1   0   0   0   1   0   1    001 000 000 000 00EE    19  ;
4.00    1   1   1   1   0   0   0    000 000 111 000 0000    0A  ;LOAD $7 0AH
4.40    1   0   1   1   0   0   0    000 000 111 000 0000    0A  ;
4.80    1   1   1   1   0   0   0    000 000 010 000 0000    0B  ;LOAD $2 0BH
5.20    1   0   1   1   0   0   0    000 000 010 000 0000    0B  ;
5.60    1   1   1   1   0   0   0    000 000 011 000 0000    19  ;LOAD $3 19H
6.00    1   0   1   1   0   0   0    000 000 011 000 0000    19  ;
6.40    1   1   1   1   0   0   0    000 000 100 000 0000    18  ;LOAD $4 18H
6.80    1   0   1   1   0   0   0    000 000 100 000 0000    18  ;
7.20    1   1   0   0   0   1   0    101 111 010 101 0000    00  ;MUL $5 $7 $2
7.60    1   0   0   0   0   1   0    101 111 010 101 0000    00  ;
8.00    1   1   1   0   0   0   0    000 000 101 000 0000    00  ;Inserted Second MUL Write Inst
8.40    1   0   1   0   0   0   0    000 000 101 000 0000    00  ;
8.80    1   1   0   0   0   0   0    000 000 000 000 0000    00  ;Inserted NOP
9.20    1   0   0   0   0   0   0    000 000 000 000 0000    00  ;
9.60    1   1   0   0   0   0   0    000 000 000 000 0000    00  ;Inserted NOP
10.00   1   0   0   0   0   0   0    000 000 000 000 0000    00  ;
10.40   1   1   0   0   0   1   0    101 011 101 110 0000    00  ;MUL $6 $3 $5
10.80   1   0   0   0   0   1   0    101 011 101 110 0000    00  ;
11.20   1   1   1   0   0   0   0    000 000 110 000 0000    00  ;Inserted Second MUL Write Inst
11.60   1   0   1   0   0   0   0    000 000 110 000 0000    00  ;
12.00   1   1   1   0   0   1   0    110 111 010 001 0000    00  ;MIN $1 $7 $2
12.40   1   0   1   0   0   1   0    110 111 010 001 0000    00  ;
12.80   1   1   0   0   1   0   0    001 101 000 000 0000    1F  ;STORE 1FH $5
```

```
40 12.80  1  1  0  0  1  0  0   001 101 000 000 0000   1F  ;STORE 1FH $5
41 13.20  1  0  0  0  1  0  0   001 101 000 000 0000   1F  ;
42 13.60  1  1  0  0  1  0  0   001 110 000 000 0000   00  ;STORE 00H $6
43 14.00  1  0  0  0  1  0  0   001 110 000 000 0000   00  ;
44 14.40  1  1  1  0  0  0  1   010 011 101 000 0005   00  ;SFL $5 $3 #0005
45 14.80  1  0  1  0  0  0  1   010 011 101 000 0005   00  ;
46 15.20  1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
47 15.60  1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
48 16.00  1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
49 16.40  1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
50 16.80  1  1  1  0  0  1  0   001 101 100 110 0000   00  ;OR $6 $5 $4
51 17.20  1  0  1  0  0  1  0   001 101 100 110 0000   00  ;
52 17.60  1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
53 18.00  1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
54 18.40  1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
55 18.80  1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
56 19.20  1  1  1  0  0  0  1   000 110 110 000 00CC   00  ;ANDI $6 $6 #00CC
57 19.60  1  0  1  0  0  0  1   000 110 110 000 00CC   00  ;
58 20.00  1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
59 20.40  1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
60 20.80  1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
61 21.20  1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
62 21.60  1  1  1  0  0  1  0   100 110 100 110 0000   00  ;ADD $6 $6 $4
63 22.00  1  0  1  0  0  1  0   100 110 100 110 0000   00  ;
64 22.40  1  1  0  0  1  0  0   001 101 000 000 0000   09  ;STORE 09H $5
65 22.80  1  0  0  0  1  0  0   001 101 000 000 0000   09  ;
66 23.20  1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
67 23.60  1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
68 24.00  1  1  0  0  1  0  0   001 110 000 000 0000   10  ;STORE 10H $6
69 24.40  1  0  0  0  1  0  0   001 110 000 000 0000   10  ;
70 24.80  1  1  1  1  0  0  0   000 000 001 000 0000   00  ;LOAD $1 00H
71 25.20  1  0  1  1  0  0  0   000 000 001 000 0000   00  ;
72 25.60  1  1  1  1  0  0  0   000 000 001 000 0000   1F  ;LOAD $1 1FH
73 26.00  1  0  1  1  0  0  0   000 000 001 000 0000   1F  ;
74 26.40  1  1  1  1  0  0  0   000 000 001 000 0000   09  ;LOAD $1 09H
75 26.80  1  0  1  1  0  0  0   000 000 001 000 0000   09  ;
76 27.20  1  1  1  1  0  0  0   000 000 001 000 0000   10  ;LOAD $1 10H
77 27.60  1  0  1  1  0  0  0   000 000 001 000 0000   10  ;
78 28.00  1  1  0  0  0  0  0   000 000 000 000 0000   00  ;NOP
79 28.40  1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
```

2.6 Automated Result Verification

We create a class CPU, which contains its ISA, a virtual RF and a virtual Memory. According to the instruction's behave, we do the same operation and restore the result in RF and Memory to generate golden result file.

```python
def exe_inst(self, inst):
    inst_type = inst.type
    self.RF[0] = 0
    if inst.imme_num > 32767:
        inst.imme_num = inst.imme_num - 65536
    if inst_type =='STORE':
        if inst.isimme is True:
            self.MEM[int(inst.addr, 16)] = inst.imme_num
        else:
            self.MEM[int(inst.addr, 16)] = self.RF[inst.read_reg_1]
    elif inst_type == 'LOAD':
        if inst.isimme is True:
            self.RF[inst.write_reg] = inst.imme_num
        else:
            self.RF[inst.write_reg] = self.MEM[int(inst.addr, 16)]
    elif inst_type == 'AND':
        if inst.isimme is True:
            self.RF[inst.write_reg] = self.RF[inst.read_reg_1] & inst.imme_num
        else:
            self.RF[inst.write_reg] = self.RF[inst.read_reg_1] & self.RF[inst.read_reg_2]
    elif inst_type == 'OR':
        if inst.isimme is True:
            self.RF[inst.write_reg] = self.RF[inst.read_reg_1] | inst.imme_num
        else:
            self.RF[inst.write_reg] = self.RF[inst.read_reg_1] | self.RF[inst.read_reg_2]
    elif inst_type == 'ADD':
        if inst.isimme is True:
            self.RF[inst.write_reg] = self.RF[inst.read_reg_1] + inst.imme_num
        else:
            self.RF[inst.write_reg] = self.RF[inst.read_reg_1] + self.RF[inst.read_reg_2]
```

```python
elif inst_type == 'MUL':
    if inst.isimme is True:
        self.RF[inst.write_reg] = self.RF[inst.read_reg_1] * inst.imme_num
    else:
        self.RF[inst.write_reg] = self.RF[inst.read_reg_1] * self.RF[inst.read_reg_2]
elif inst_type == 'MIN':
    if inst.isimme is True:
        if (self.RF[inst.read_reg_1] > inst.imme_num):
            self.RF[inst.write_reg] = inst.imme_num
        else:
            self.RF[inst.write_reg] = self.RF[inst.read_reg_1]
    else:
        if (self.RF[inst.read_reg_1] > self.RF[inst.read_reg_2]):
            self.RF[inst.write_reg] = self.RF[inst.read_reg_2]
        else:
            self.RF[inst.write_reg] = self.RF[inst.read_reg_1]
elif inst_type == 'SFL':
    self.RF[inst.write_reg] = self.RF[inst.read_reg_1] << inst.imme_num
elif inst_type == 'SFR':
    self.RF[inst.write_reg] = self.RF[inst.read_reg_1] >> inst.imme_num
```

```
######################  RF Content  ######################
  $0:  Decimal:    0  Hex: 0000
  $1:  Decimal:  211  Hex: 00D3
  $2:  Decimal:   20  Hex: 0014
  $3:  Decimal:  238  Hex: 00EE
  $4:  Decimal:   11  Hex: 000B
  $5:  Decimal: 7616  Hex: 1DC0
  $6:  Decimal:  211  Hex: 00D3
  $7:  Decimal: -241  Hex: FF0F

######################  MEM Content  ######################
   0H:Decimal:  168     Hex: 00A8    |     1H:Decimal:    0     Hex: 0000    |     2H:Decimal:    0     Hex: 0000    |     3H:Decimal:    0     Hex: 0000    |
   4H:Decimal:    0     Hex: 0000    |     5H:Decimal:    0     Hex: 0000    |     6H:Decimal:    0     Hex: 0000    |     7H:Decimal:    0     Hex: 0000    |
   8H:Decimal:    0     Hex: 0000    |     9H:Decimal: 7616     Hex: 1DC0    |     AH:Decimal: -241     Hex: FF0F    |     BH:Decimal:   20     Hex: 0014    |
   CH:Decimal:    0     Hex: 0000    |     DH:Decimal:    0     Hex: 0000    |     EH:Decimal:    0     Hex: 0000    |     FH:Decimal:    0     Hex: 0000    |
  10H:Decimal:  211     Hex: 00D3    |    11H:Decimal:    0     Hex: 0000    |    12H:Decimal:    0     Hex: 0000    |    13H:Decimal:    0     Hex: 0000    |
  14H:Decimal:    0     Hex: 0000    |    15H:Decimal:    0     Hex: 0000    |    16H:Decimal:    0     Hex: 0000    |    17H:Decimal:    0     Hex: 0000    |
  18H:Decimal:   11     Hex: 000B    |    19H:Decimal:  238     Hex: 00EE    |    1AH:Decimal:    0     Hex: 0000    |    1BH:Decimal:    0     Hex: 0000    |
  1CH:Decimal:    0     Hex: 0000    |    1DH:Decimal:    0     Hex: 0000    |    1EH:Decimal:    0     Hex: 0000    |    1FH:Decimal: -180     Hex: FF4C    |
```

Part 3 Register File

3.1 Schematic Design

The schematic level design is shown below in Figure 3.1.1, and the input signal is the Rt[2:0], Rs[2:0], Data_IN[15:0], Write_EN, Read_EN, and output signals Data_Rs[15:0] and Data_Rt[15:0].

For designing the register file in the ID stage, we use 128 bit DFF to store 8 different 16 bit data inside the register file. Inside the RF, I use 8 AND gates to combine the Write_EN signal and output from 3-to-8 decoder to connect 8 words inside the register file to control the CLK for DFF. Once the proper wordline and Write_EN signal turns high, then the informations will be written inside the matching word line inside this register file.

For Read data, I use 2 8-to-1 demux to read out the data inside the register file for Rs and Rt address, and use 2 16 bit 2-to-1 mux to make sure ID stage output the correct data if the Rs or Rt address is matching with the latest data address will be updated from the WB stage.

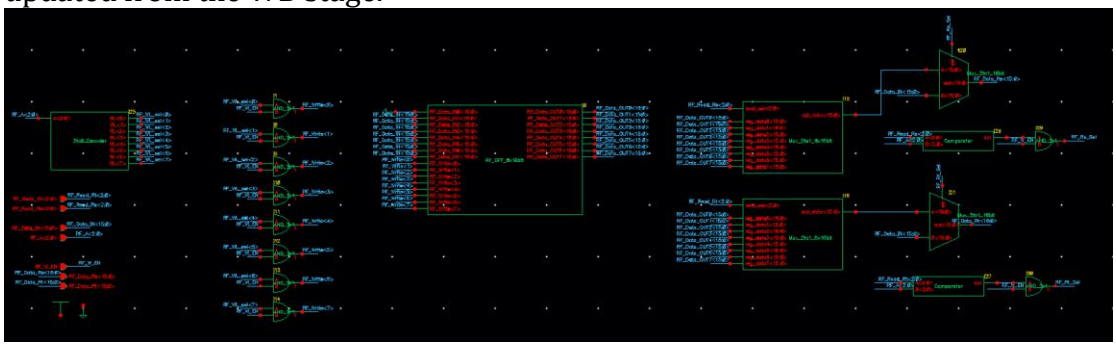


Figure 3.1.1: Schematic Design for Register File and ID stage

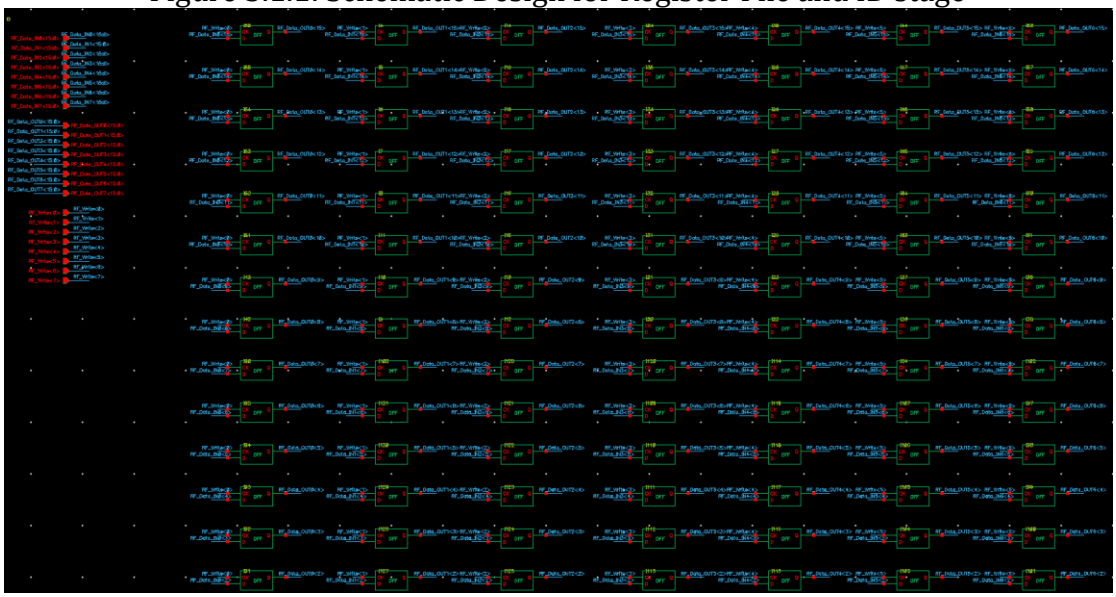


Figure 3.1.1.a: 128 bit DFF for Register File

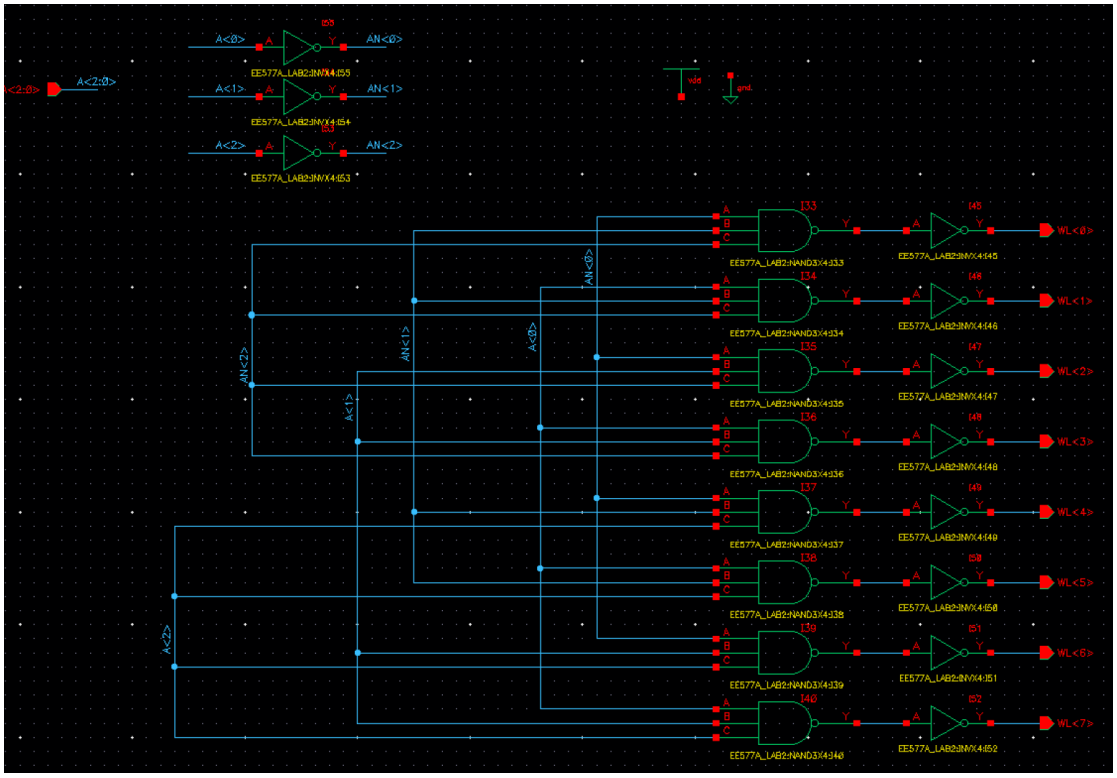


Figure 3.1.1.b: 3 to 8 Decoder for ID Stage

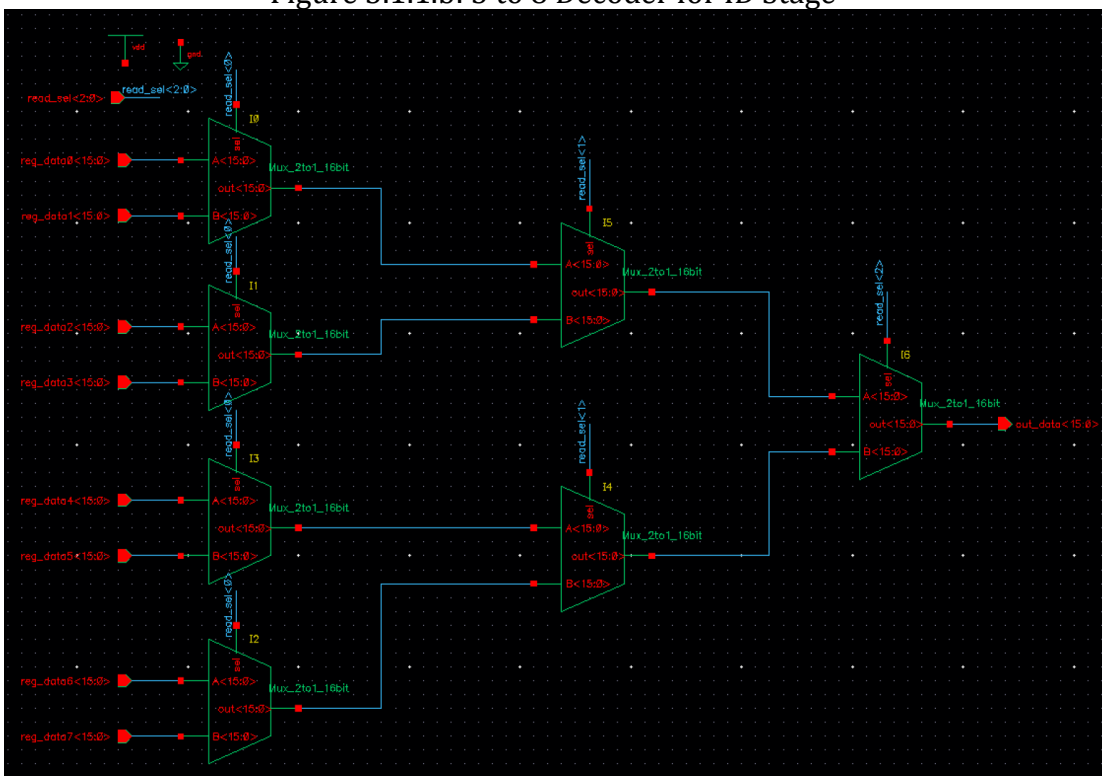


Figure 3.1.c: 16 bit DeMux for ID Stage

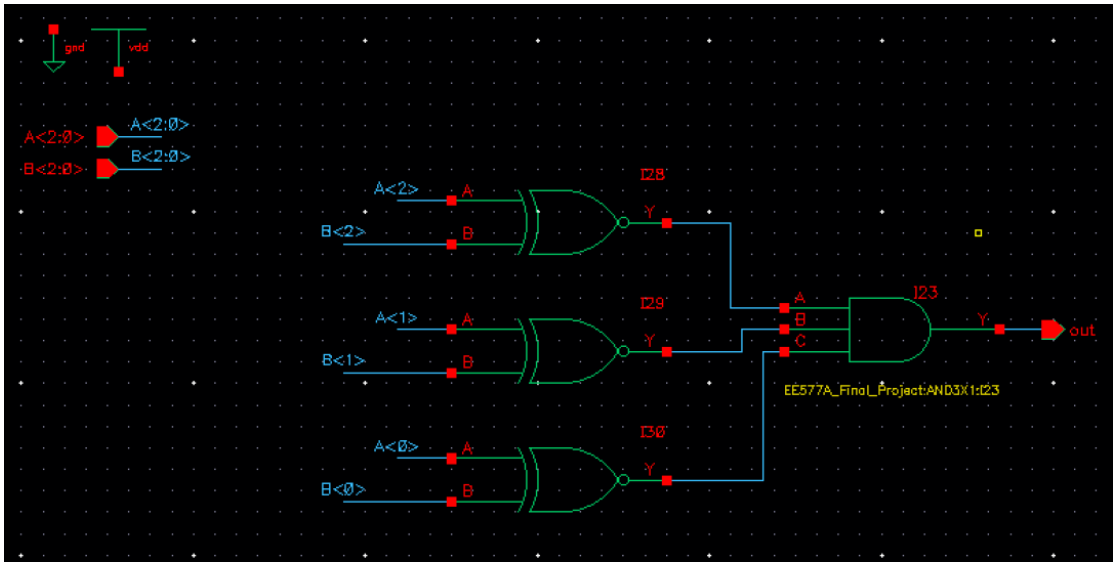


Figure 3.1.dL Address Comparator for ID Stage

3.2 Functionality Test

After, I write the vector file to brevly test the function for my register file, and the output result shown below in Figure 3.2 below. In this test, I write data '1234' into address '0' at 0ns, data '2345' into address '1' at 5ns, data '3456' into address '2' at 10ns, and data 'ab52' into address '6' at 15ns. Then I start reading the address for Rt for'0' and Rs for '1' at 20ns, and read '2' and '6' at 25ns. Finally test the read and write function at the same time at 30ns, which I need to write the latest update data at address '0'.

From the output waveform shown below in Figure 3.2, we can tell that the data can be written and read from the register properly, and the ID stage can transport the correct data even it needs to use the latest update written from WB stage.

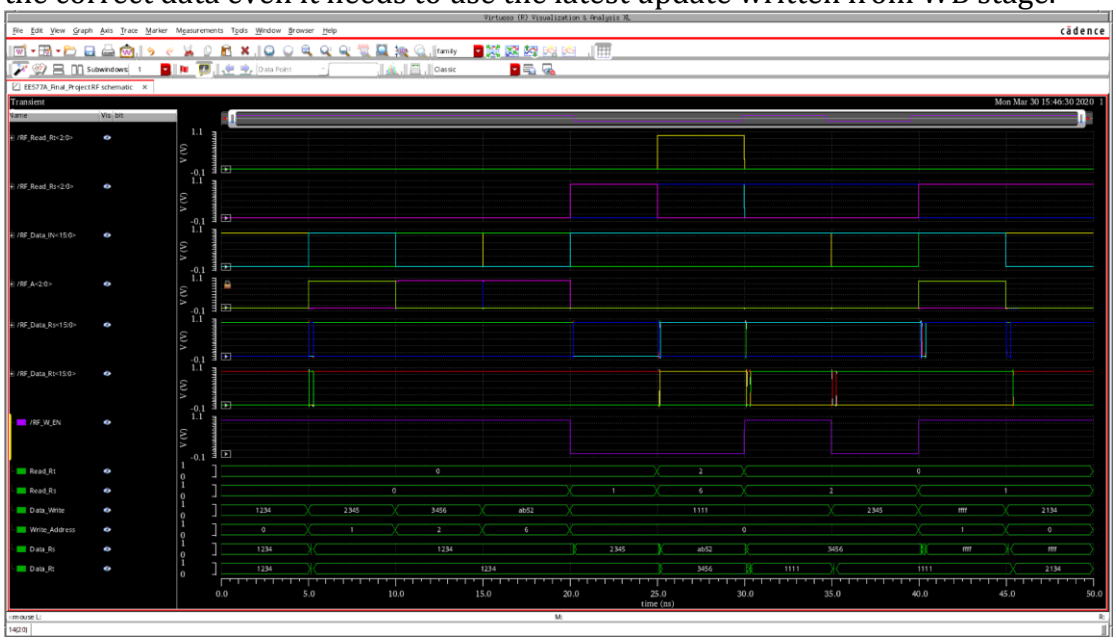


Figure 3.2: Output Waveform for Function Test at ID stage

Part 4 Execution Stage (ALU Design)

4.1 Overview

There is 5 parts of ALU, which are 16-bit Carry Skip Adder, 5-bit multiplier, 16-bit AND, 16-bit OR and 16-bit Shifter. ALU_Op signals produced instruction decode will control the output result of ALU. To keep the delay minimized, we let the result of adder and multiplier go through least number of MUX. The control logic is shown below.

Operation	*ALU_Op<2>*	*ALU_Op<1>*	*ALU_Op<0>*
AND	0	0	0
OR	0	0	1
SFL	0	1	0
SFR	0	1	1
ADD	1	0	0
MUL	1	0	1
MIN	1	1	0
	1	1	1

The overview schematic is shown below.

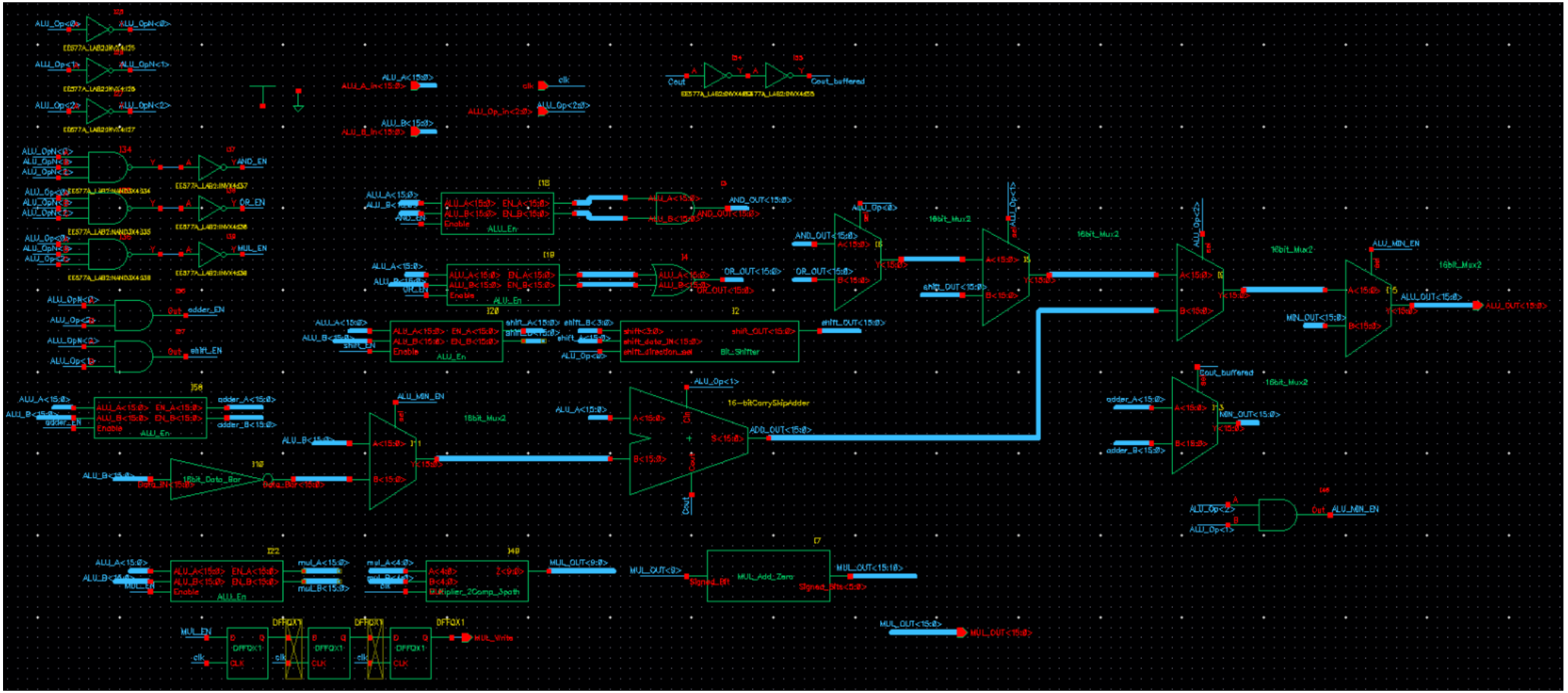


Figure 4.1.1 Overview of ALU Schematic

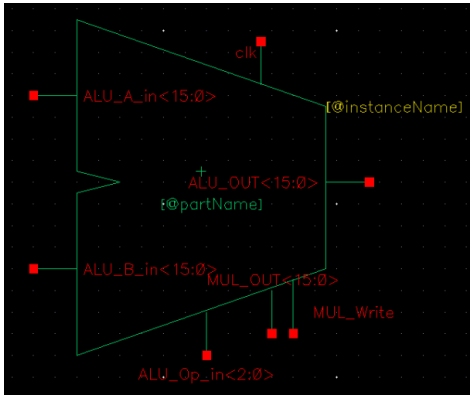


Figure 4.1.2 Symbol of ALU

4.2 Adder

we use it to design a 16-bit Carry-Skip Adder. At this point there is no need to optimize the design of the Propagate signal block. The 16-bit Carry-Skip adder consists of 4 4-bit ripple carry adders, 4 multiplexers and 4 propagation blocks. For 4-bit ripple carry adders, we use 4 1-bit full adders and connect their C_{out} into C_{in} of the next stage. The schematic and result are shown below.

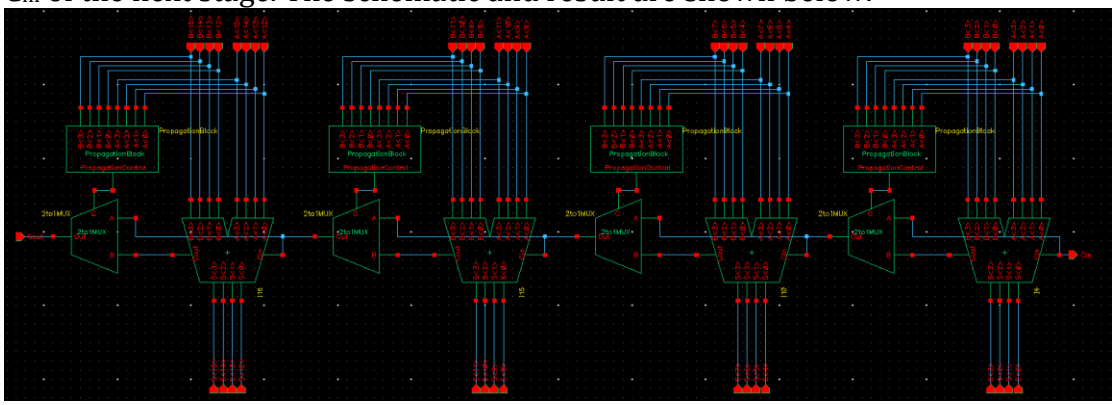


Figure 4.2.1 Schematic and Symbol of 16-bit Carry-Skip Adder

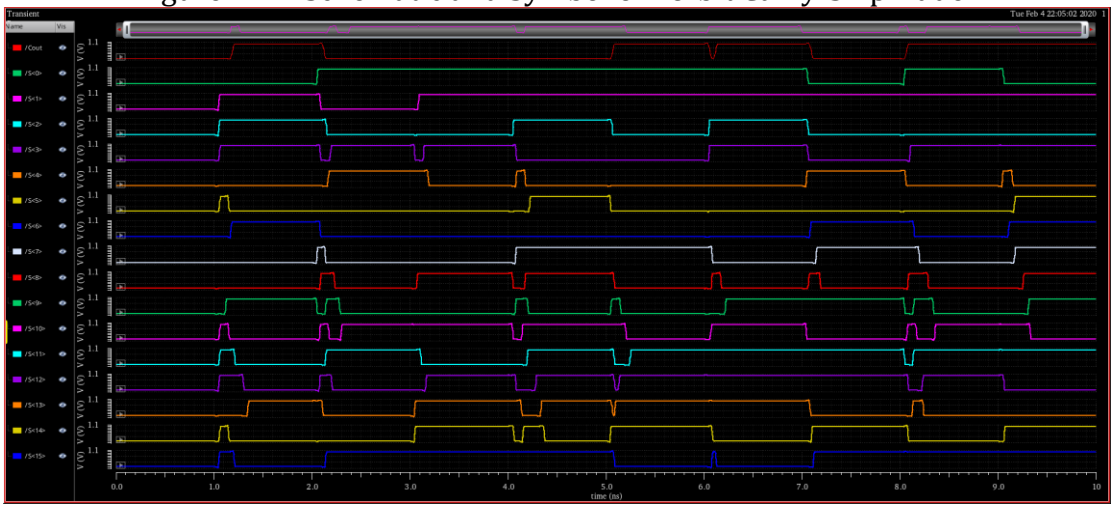


Figure 4.2.2 Result Waveform of 16-bit Carry-Skip Adder Functionality Test

4.3 Multiplier

4.3.1 Multiplier Schematic Design

For design the Multiplier, we use the balanced DFF separate the FAs with the following design shown in the Figure 4.7.1.1, each red line represents one of the pipeline registers. In this structure, we will be able to balance the pipeline optimization time between each stage with clock cycle 420ps. The critical path for this pipeline Multiplier is AND+3FA between each stage 2 and 3 of this multiplier pipelines.

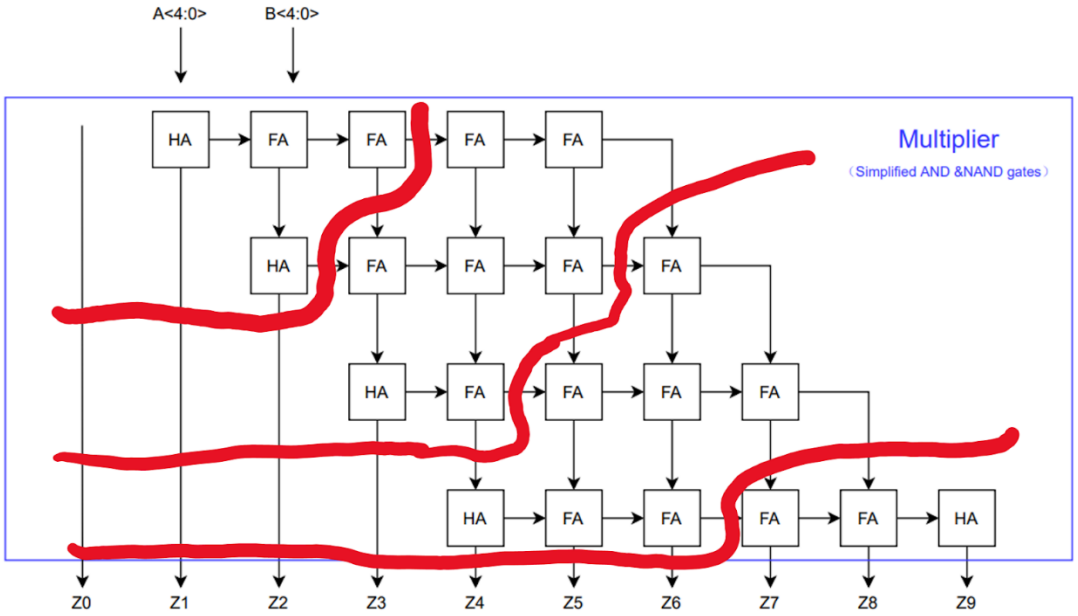


Figure 4.3.1.1: Simplified Design Diagram for Balanced Stage

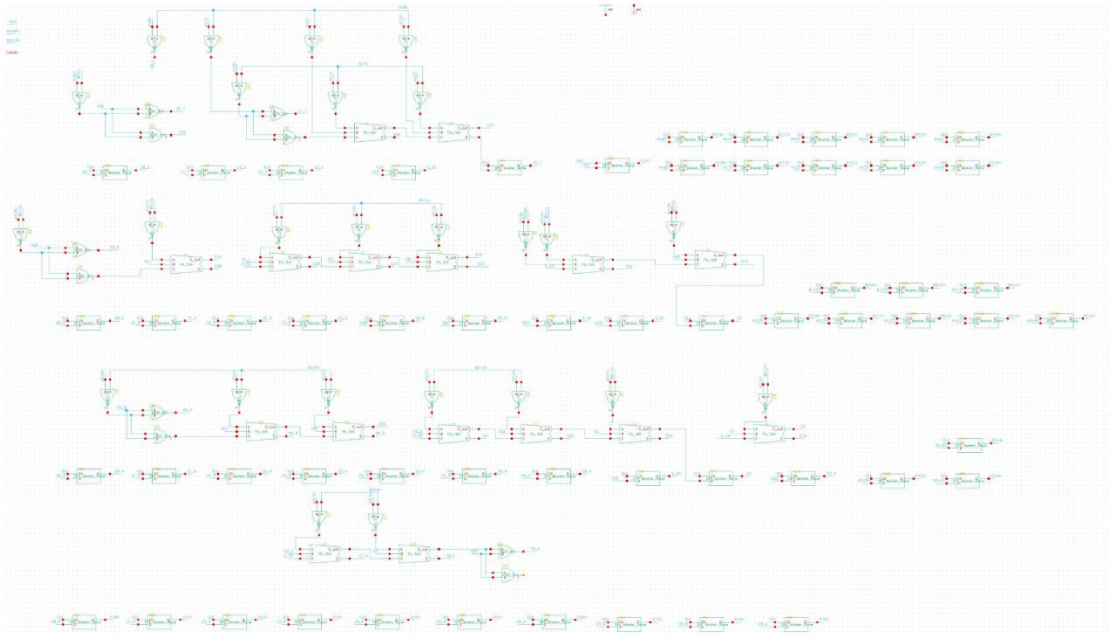


Figure 4.3.1.2: Schematic Design for Balanced Stage

Because the result of multiplier is 10 bits, but ALU output is 16 bits, we need to give the Most 5 Significant Bits with the sign bit, which is the original MSB of output. So we build a block to implement this.

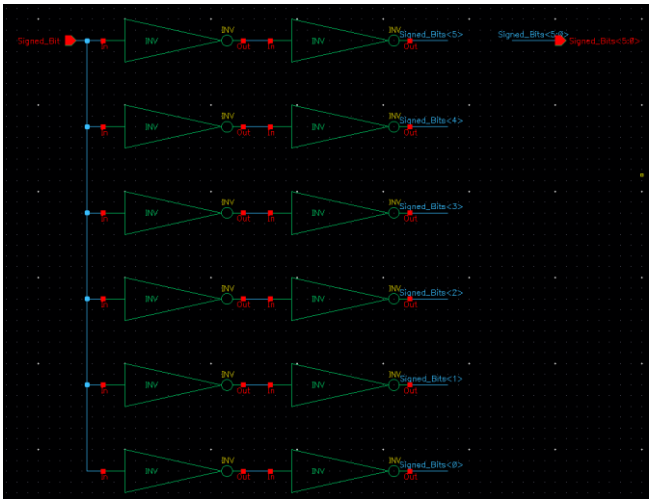


Figure 4.3.1.3 Multiplier MSB Producer

Figure 4.3.1.4: Multiplier Schematic Overview

4.3.2 Multiplier Functionality Test

By using the vector file below shown in Figure 4.3.2.1 for functionality test, we can prove that the Multiplier itself is working properly, and the generation waveform is showing in Figure 4.3.2.2 below, comparing with the output bus shown in golden result 4.3.2.3.

```
radix 14 14 1
io i i i
vname A<[4:0]> B<[4:0]> CLK
tunit ps
slope 5
vih 1
vil 0

0    00   00   0
210.0    1E   10   1
420.0    1E   10   0
630.0    0B   17   1
840.0    0B   17   0
1050.0   16   06   1
1260.0   16   06   0
1470.0   14   16   1
1680.0   14   16   0
1890.0   1B   1D   1
2100.0   1B   1D   0
2310.0   13   1E   1
2520.0   13   1E   0
2730.0   09   14   1
2940.0   09   14   0
3150.0   00   0D   1
3360.0   00   0D   0
3570.0   17   0E   1
3780.0   17   0E   0
3990.0   0A   05   1
4200.0   0A   05   0
4410.0   1F   1E   1
4620.0   1F   1E   0
4830.0   1F   1D   1
5040.0   1F   1D   0
5250.0   1F   01   1
5460.0   1F   01   0
5670.0   1F   03   1
5880.0   1F   03   0
6090.0   1F   07   1
6300.0   1F   07   0
6510.0   00   00   1
6720.0   00   00   0
6930.0   00   00   1
7140.0   00   00   0
7350.0   00   00   1
7560.0   00   00   0
7770.0   00   00   1
7980.0   00   00   0
8190.0   00   00   1
8400.0   00   00   0
```

Figure 4.3.2.1: Vector File For Function Test

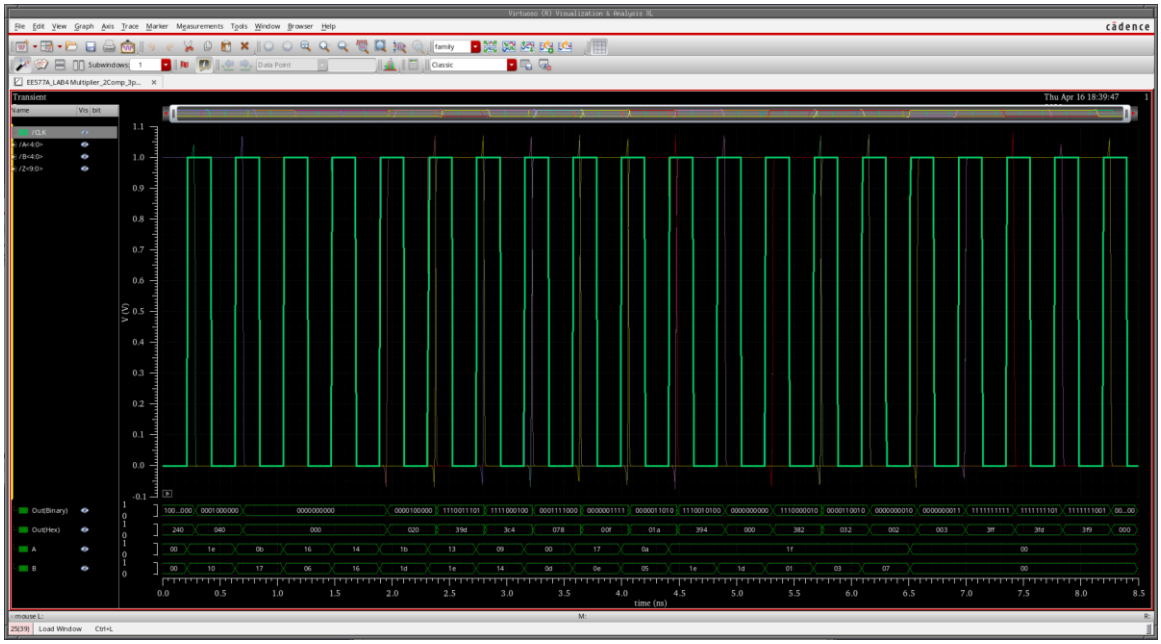


Figure 4.3.2.2: Output Waveform for Multiplier

op1(dec)	op2(dec)	result(dec)	op1(bin)	op2(bin)	result(bin)
-2	-16	32	11110	10000	0000100000
11	-9	-99	01011	10111	1110011101
-10	6	-60	10110	00110	1111000100
-12	-10	120	10100	10110	0001111000
-5	-3	15	11011	11101	0000001111
-13	-2	26	10011	11110	0000011010
9	-12	-108	01001	10100	1110010100
0	13	0	00000	01101	0000000000
-9	14	-126	10111	01110	1110000010
10	5	50	01010	00101	0000110010
-1	-2	2	11111	11110	0000000010
-1	-3	3	11111	11101	0000000011
-1	1	-1	11111	00001	1111111111
-1	3	-3	11111	00011	1111111101
-1	7	-7	11111	00111	1111111001

Figure 4.3.2.3: Golden Result

4.4 AND & OR Operation

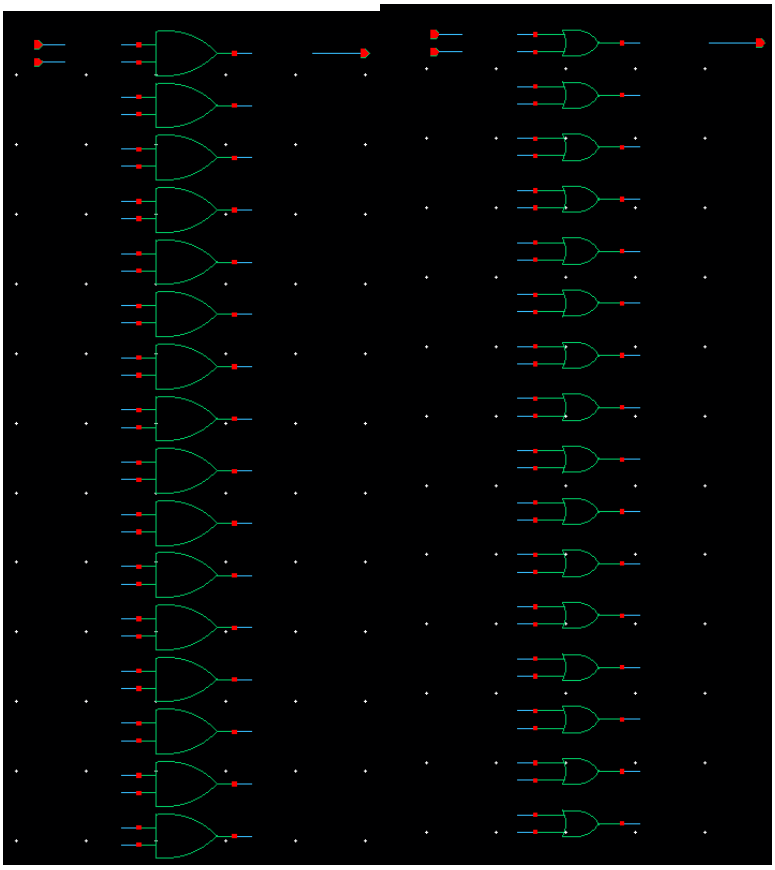


Figure 4.4.1 16-bit AND and OR Schematic

4.5 Comparator

We can use 16-bit Carry Skip Adder to implement comparator.We can invert a number, say, input_B, and keep input_A as same. If we do the comparison operation, we can give 1 to Cin. Then the adder will calculate

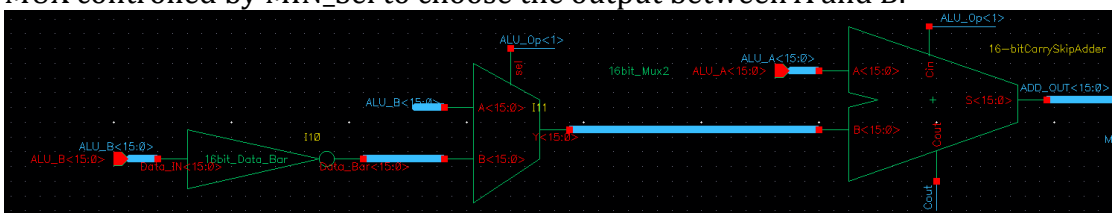
$$(\mathrm{A})_{\mathrm{binary}} + \overline{(\mathrm{B})_{\mathrm{binary}}} + 1 = (\mathrm{A} - \mathrm{B})_{\mathrm{binary}}$$

The MIN_Sel signal from adder will serve as the result of the comparison between A and B. Min_Sel can be produced by this equation:

$$\mathrm{MIN\_Sel} = \mathrm{S}_{15} \oplus \mathrm{Cout}_{15} \oplus \mathrm{Cout}_{16},$$

where S_{15} is the MSB of result.

If MIN_Sel = 0, it means A ≥ B and if MIN_Sel = 1, it means A < B. We can use a MUX controlled by MIN_Sel to choose the output between A and B.



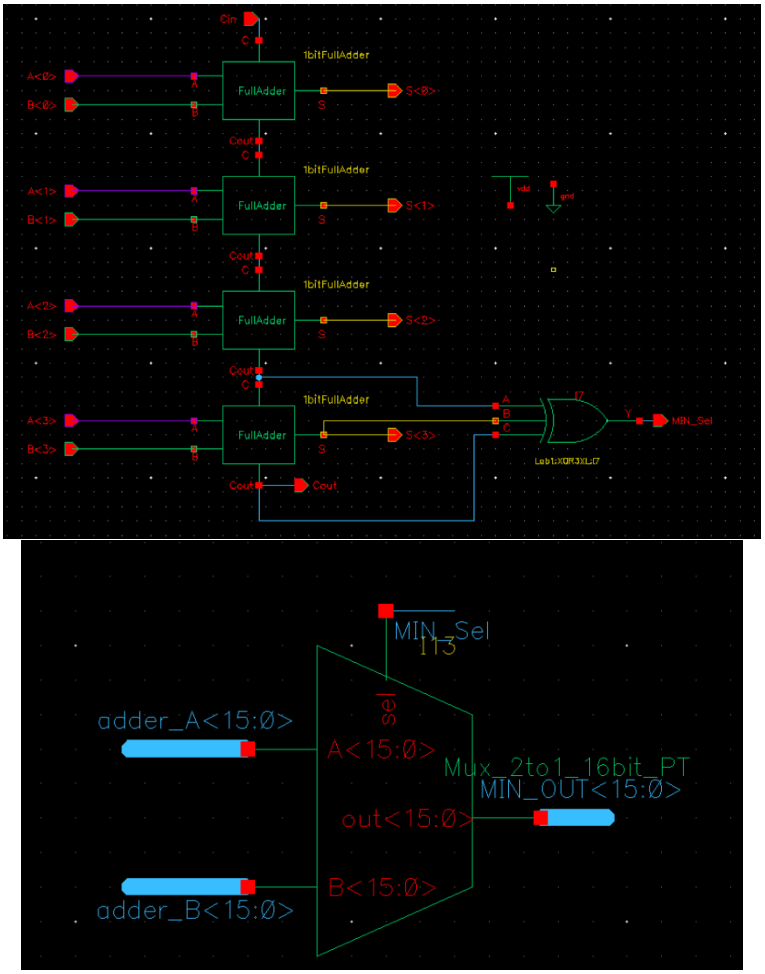


Figure 4.5.1 16-bit Comparator Schematic

4.6 Shifter Design

4.6.1 Schematic Design

In this project, we are required to complete a shifter to satisfy 16 bit binary data, and shift the data to right or left. We design our bit shifter inside ALU by using two Barrel shifters, and use a 2 to 1 MUX to select the direction for shift left or shift right (Figure 4.5.1). For data shift right, we are required to fill the most significant bit by using the original most significant bit due to this signed number (shown in Figure 4.5.2). For shift left, shifter will fill logic 0 to the least significant bit (shown in Figure 4.5.3). All the mux used in this shifter are 2 to 1 CMOS MUX on the standard library.

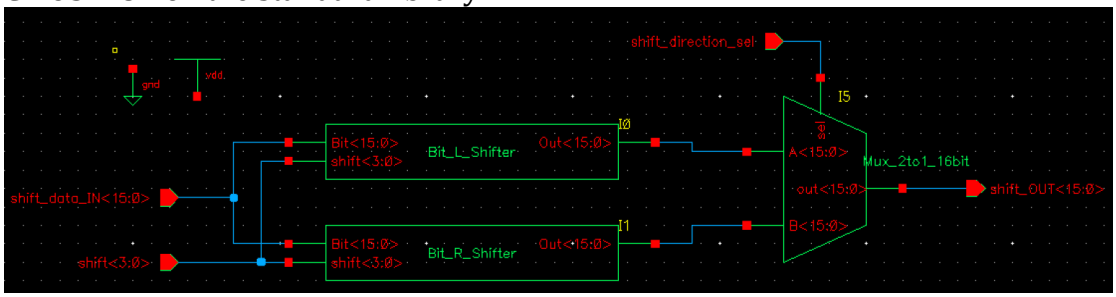


Figure 4.5.1: Schematic for Bit Shifter

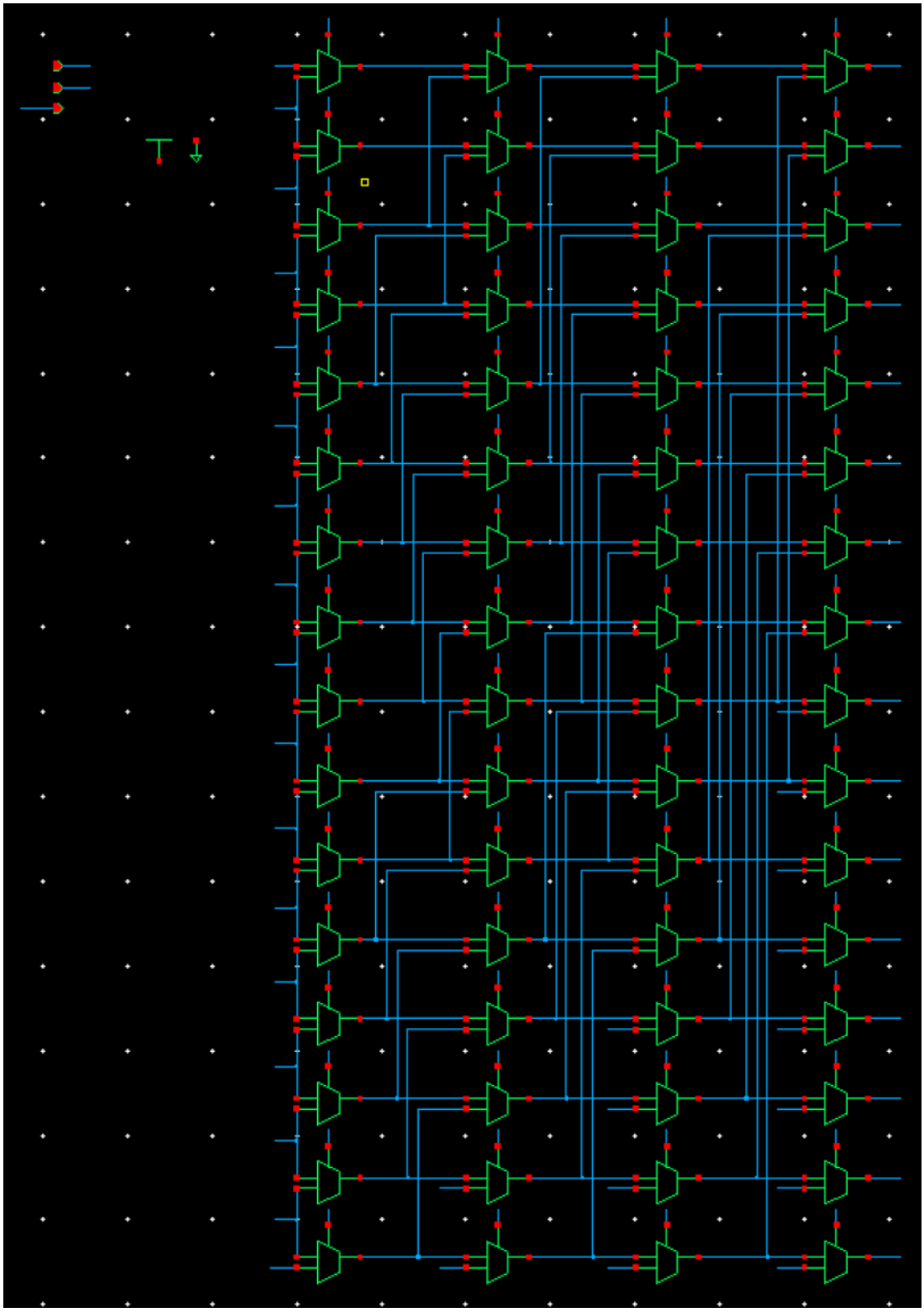


Figure 4.5.2 Overall Image for Left Bit Shifter

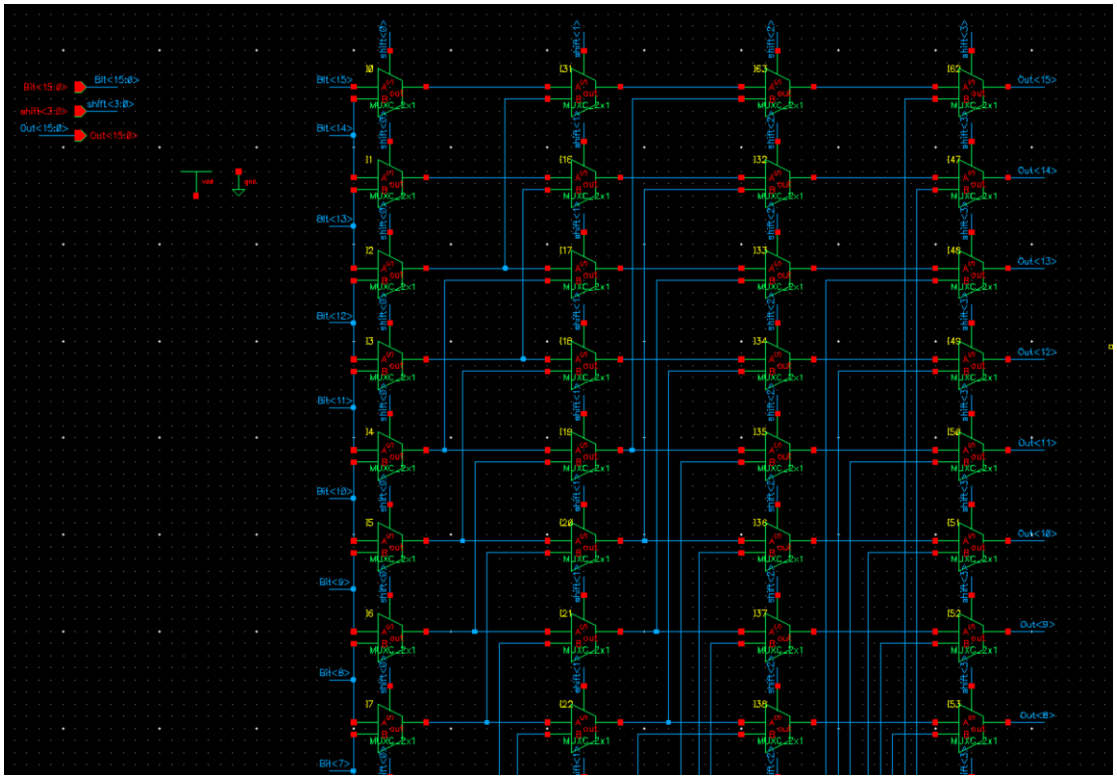


Figure 4.5.2.a Detail Image for Left Shifter

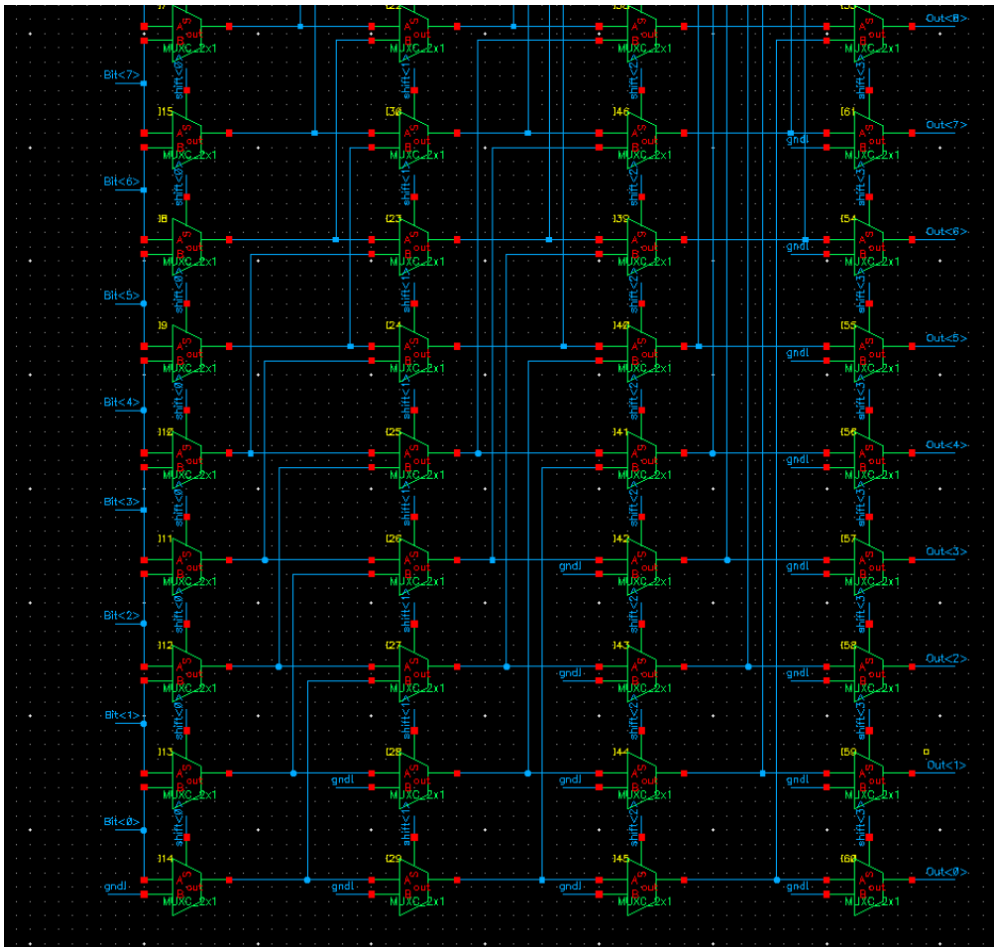


Figure 4.5.2.a Detail Image for Left Shifter

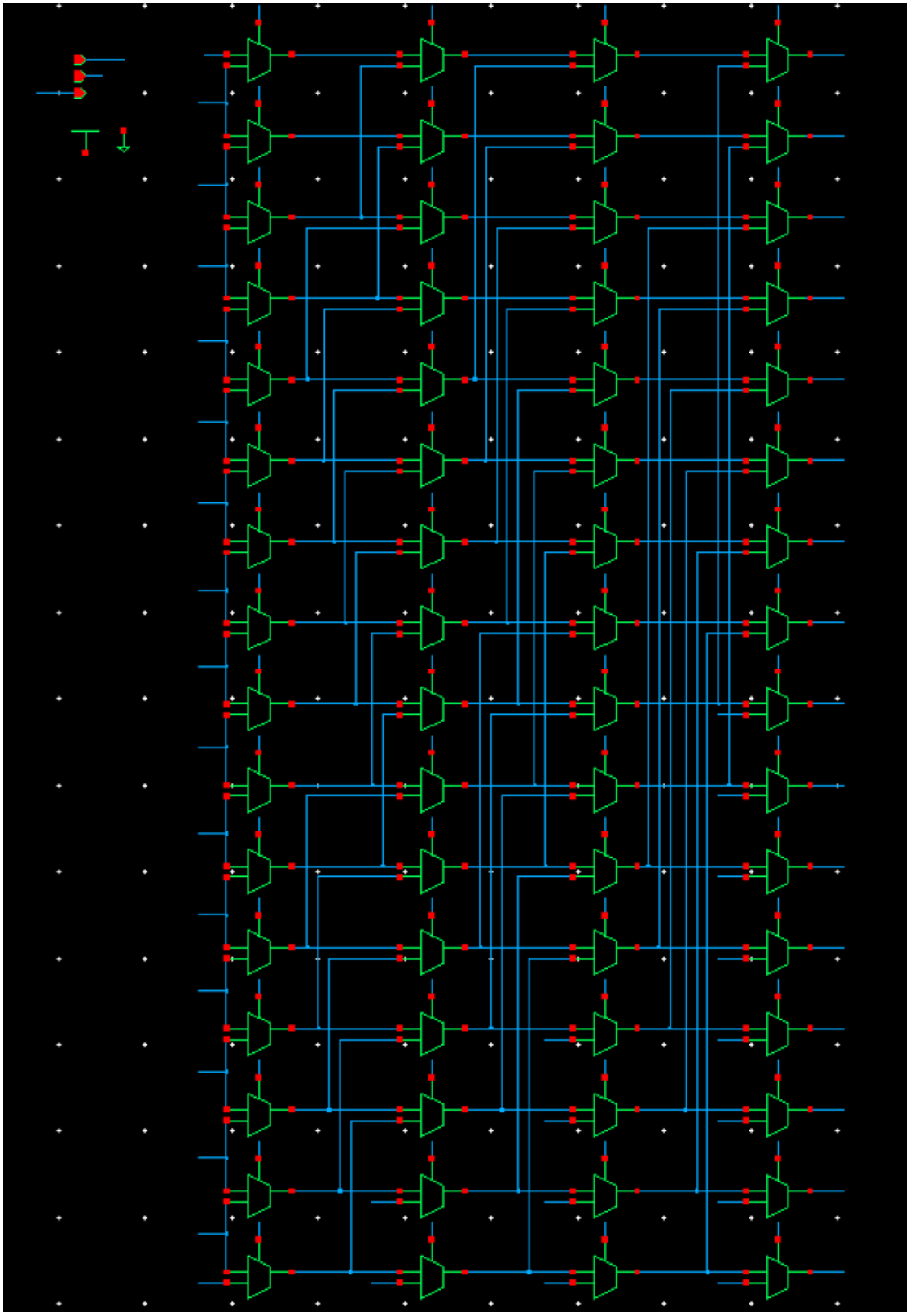


Figure 4.5.3 Overall Image for Right Bit Shifter

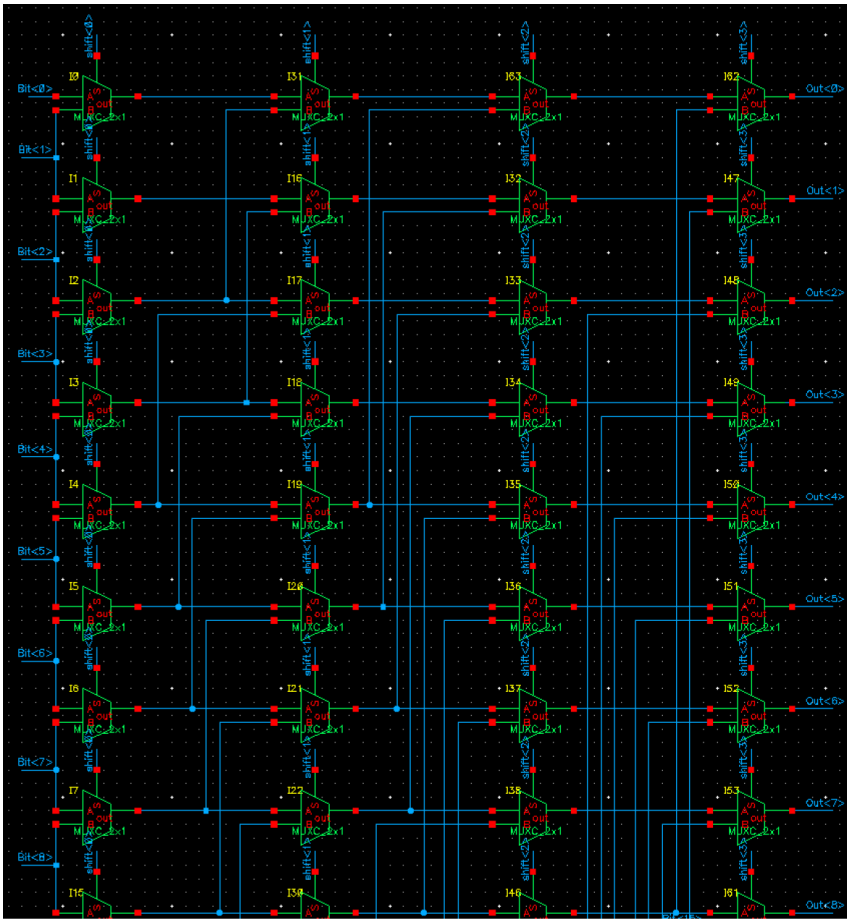


Figure 4.5.3.a Detail Image for Right Shifter

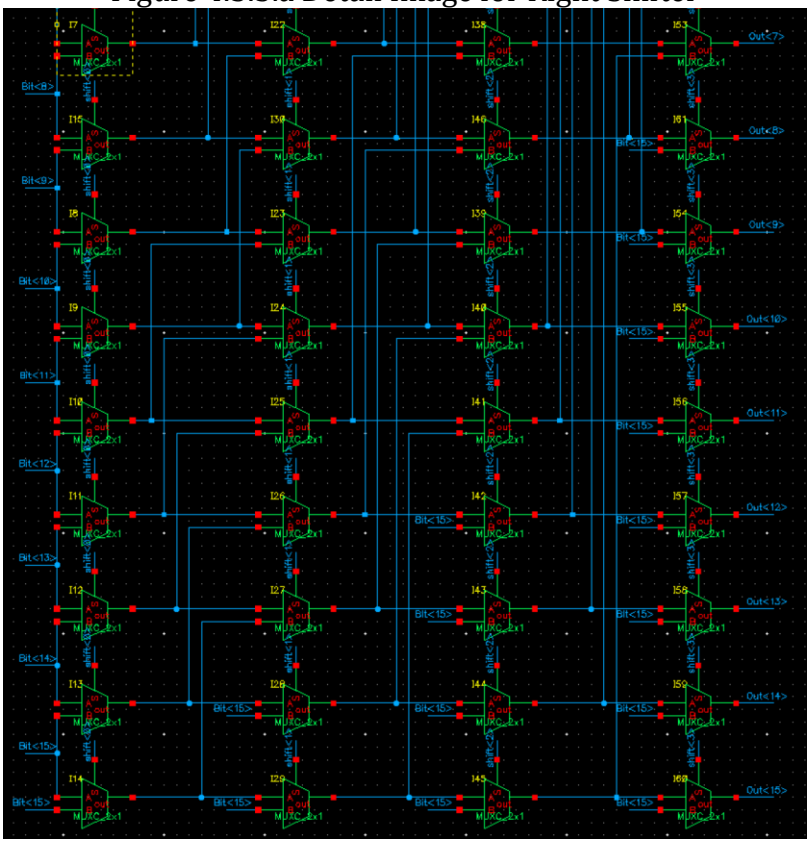


Figure 4.5.3.b Detail Image for Right Shifter

4.6.2 Schematic Functionality Test

For testing the function for this bit shifter, I write a vector file to test the shift left and shift right function. From the output waveform shown in figure 5.3.4.a and 5.3.4.b, the function for shifting 16 bit to left and right can be proved.

From the following waveform, we can tell the data is being shifted to the required direction with the proper number of the shift. Thus, this bit shifter is working properly.

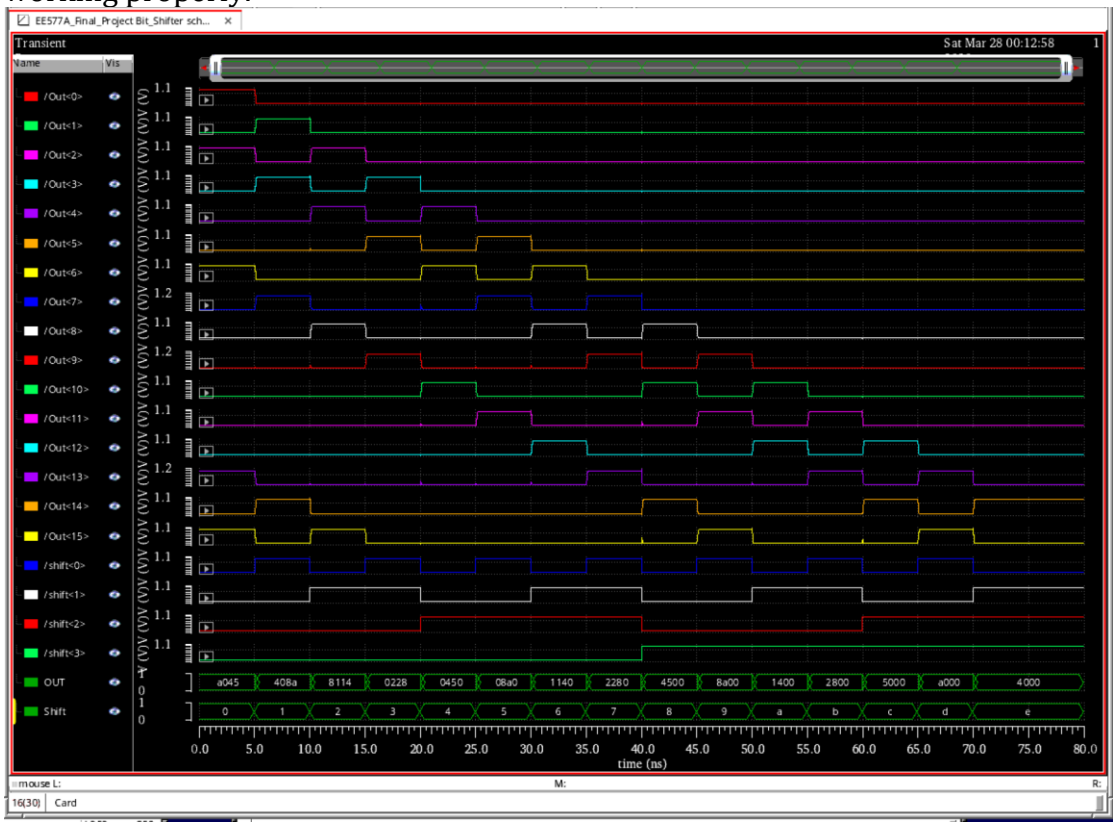


Figure 4.5.4 a: Shift left Function Test

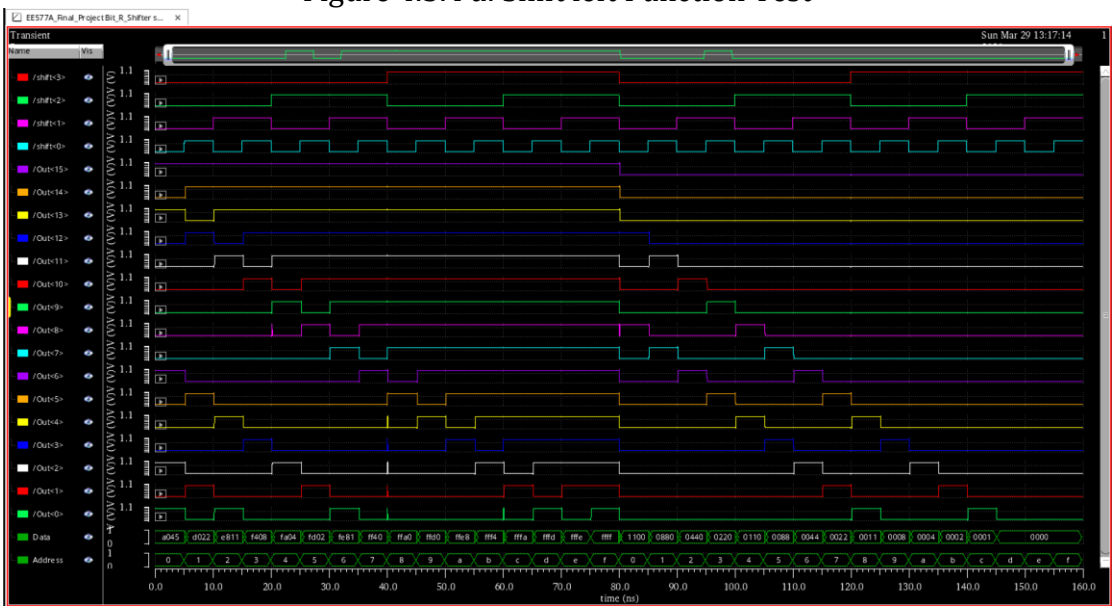


Figure 4.5.4.b: Shift Right Function Test

Part 5 Memory (SRAM)

5.1 Schematic Design

For Memory inside this pipeline design, we use the 512 bit SRAM memory based on the structure we created in LAB2, and add the INV to form the AN[5:0] and AND gates to control signal for the Write and Read enable with decoder. The new design is shown in Figure 5.1.1 below.

For making sure the memory can finish precharge and write/read in one clock cycle, we implement the circuit shown in Figure 5.1.1.c below. This structure can add delay after the command is given to SRAM, and help to control the Sense Amplifier and write circuit to open and close after the precharge is done inside this circuit.

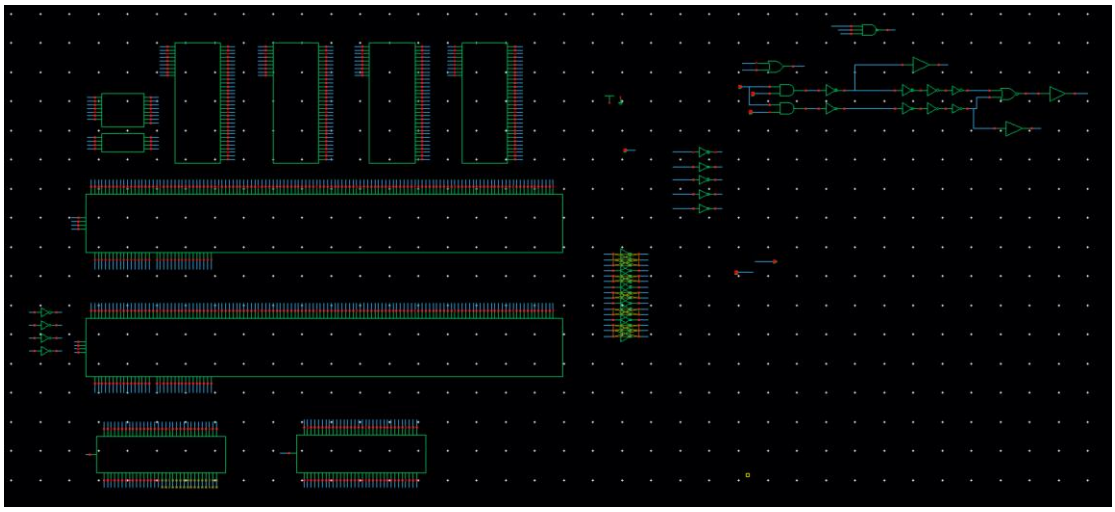


Figure 5.1.1: SRAM Design for Memory in Pipeline

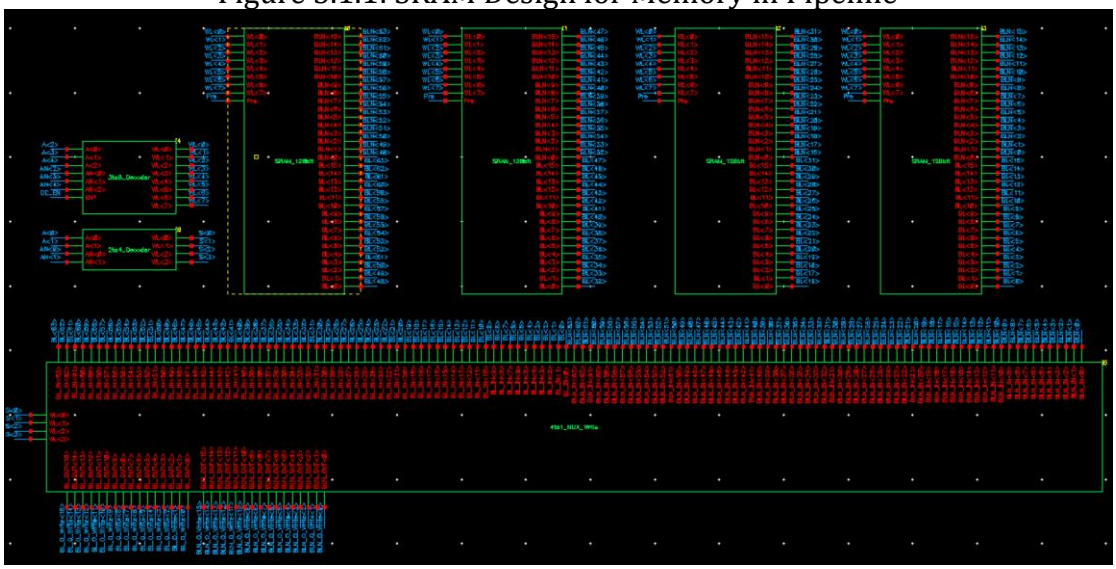


Figure 5.1.1.a: Detail Information for SRAM based Memory

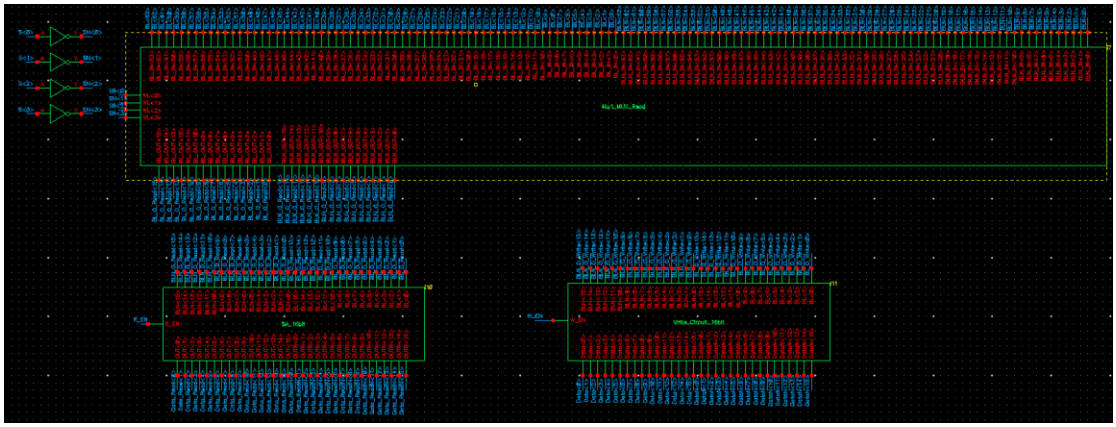


Figure 5.1.1.b: Detail Information for SRAM based Memory

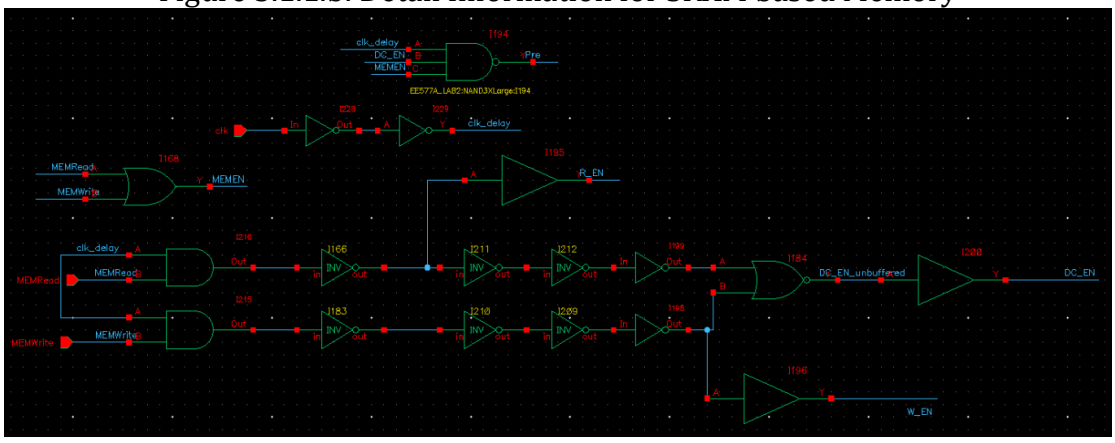
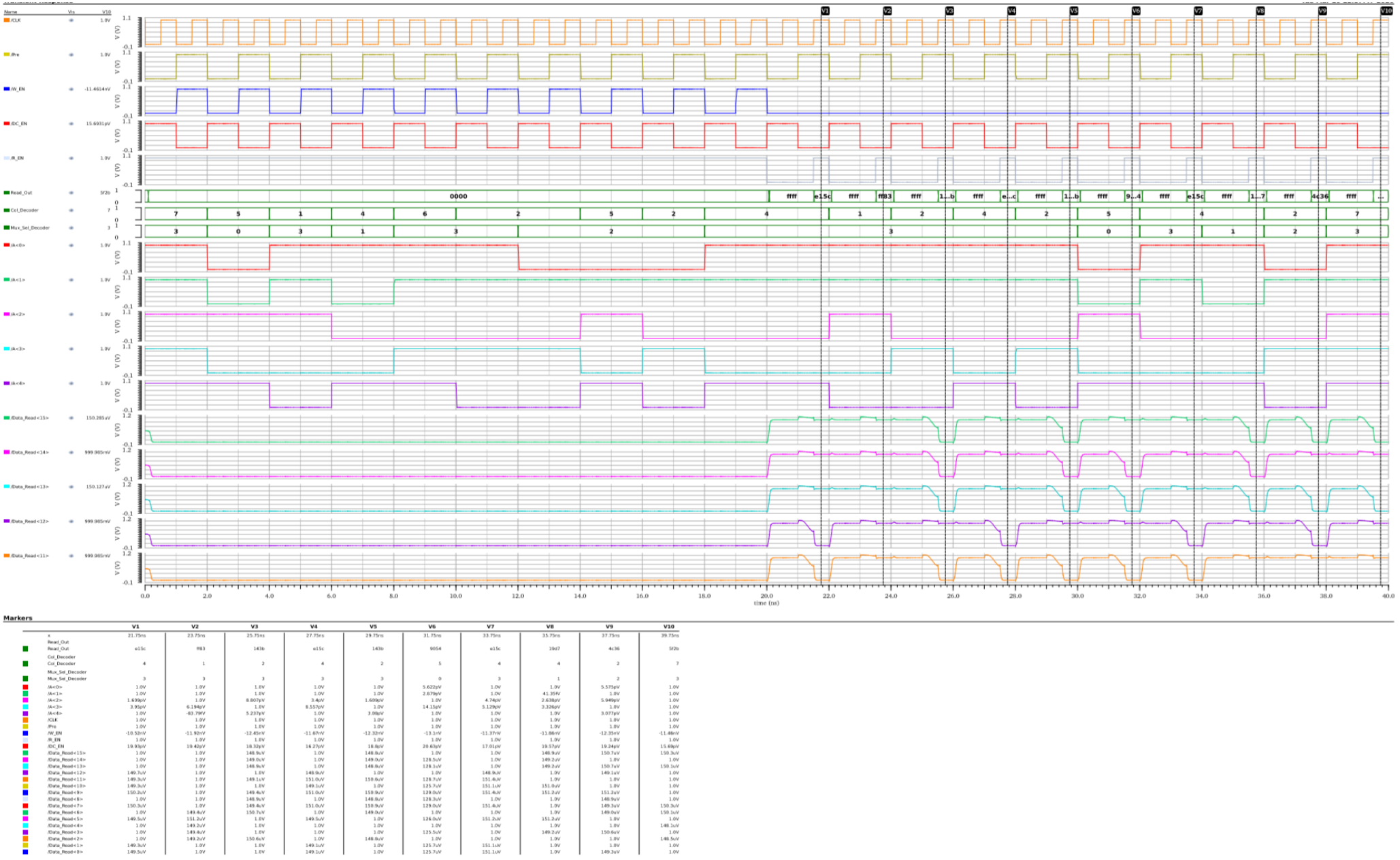


Figure 5.1.1.c: Signal Generating circuit in SRAM

5.2 Functionality Test

For testing the updated SRAM for memory, I modified the previous vector file and tested the new SRAM, I get the following output waveform shown in Figure 5.2.1 and compare with the output result I get from Lab2. They output the same result, and this matches our requirement for memory in this pipeline design.
In this part, we try to achieve the zero skew clock tree design. For equally balanced the design load that connect to our 180 DFF and other logic gates design, we evenly split the 180 DFFs to 4 different loops while trying to balance the load for logic gates. However, this part still need to be implement when it comes to the layout design. The unbalanced gate can be filled up by the actual wire connection.



Markers

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
x	21.75ns	23.75ns	25.75ns	27.75ns	29.75ns	31.75ns	33.75ns	35.75ns	37.75ns	39.75ns
Read_Out										
Read_Out	e15c	f883	143b	e15c	143b	9054	e15c	19d7	4c36	5f2b
Col_Decoder										
Col_Decoder	4	1	2	4	2	5	4	4	2	7
Mux_Sel_Decoder										
Mux_Sel_Decoder	3	3	3	3	3	0	3	1	2	3
/A<0>	1.0V	1.0V	1.0V	1.0V	1.0V	5.622pV	1.0V	1.0V	5.575pV	1.0V
/A<1>	1.0V	1.0V	1.0V	1.0V	1.0V	2.879pV	1.0V	41.35fV	1.0V	1.0V
/A<2>	1.609pV	1.0V	8.807pV	3.4pV	1.609pV	1.0V	4.74pV	2.638pV	5.949pV	1.0V
/A<3>	3.95pV	6.194pV	1.0V	8.557pV	1.0V	14.15pV	5.129pV	3.326pV	1.0V	1.0V
/A<4>	1.0V	-83.79fV	5.237pV	1.0V	3.98pV	1.0V	1.0V	1.0V	3.077pV	1.0V
/CLK	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V
/Pre	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V
/W_EN	-10.52nV	-11.92nV	-12.45nV	-11.67nV	-12.32nV	-13.1nV	-11.37nV	-11.86nV	-12.35nV	-11.46nV
/R_EN	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V
/DC_EN	19.93pV	19.42pV	18.32pV	16.27pV	18.8pV	20.63pV	17.01pV	19.57pV	19.24pV	15.69pV
/Data_Read<15>	1.0V	1.0V	148.9uV	1.0V	148.8uV	1.0V	1.0V	148.9uV	150.7uV	150.3uV
/Data_Read<14>	1.0V	1.0V	149.0uV	1.0V	149.0uV	128.5uV	1.0V	149.2uV	1.0V	1.0V
/Data_Read<13>	1.0V	1.0V	148.9uV	1.0V	148.8uV	128.1uV	1.0V	149.2uV	150.7uV	150.1uV
/Data_Read<12>	149.7uV	1.0V	1.0V	148.9uV	1.0V	1.0V	148.9uV	1.0V	149.1uV	1.0V
/Data_Read<11>	149.3uV	1.0V	149.1uV	151.0uV	150.6uV	128.7uV	151.4uV	1.0V	1.0V	1.0V
/Data_Read<10>	149.3uV	1.0V	1.0V	149.1uV	1.0V	125.7uV	151.1uV	151.0uV	1.0V	1.0V
/Data_Read<9>	150.2uV	1.0V	149.4uV	151.0uV	150.9uV	129.0uV	151.4uV	151.2uV	151.2uV	1.0V
/Data_Read<8>	1.0V	1.0V	148.9uV	1.0V	148.8uV	128.3uV	1.0V	1.0V	148.9uV	1.0V
/Data_Read<7>	150.3uV	1.0V	149.4uV	151.0uV	150.9uV	129.0uV	151.4uV	1.0V	149.3uV	150.3uV
/Data_Read<6>	1.0V	149.4uV	150.7uV	1.0V	149.0uV	1.0V	1.0V	1.0V	149.0uV	150.1uV
/Data_Read<5>	149.5uV	151.2uV	1.0V	149.1uV	1.0V	126.0uV	151.2uV	151.2uV	1.0V	1.0V
/Data_Read<4>	1.0V	149.2uV	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	1.0V	148.1uV
/Data_Read<3>	1.0V	149.4uV	1.0V	1.0V	1.0V	125.5uV	1.0V	149.2uV	150.6uV	1.0V
/Data_Read<2>	1.0V	149.2uV	150.6uV	1.0V	148.8uV	1.0V	1.0V	1.0V	1.0V	148.5uV
/Data_Read<1>	149.3uV	1.0V	1.0V	149.1uV	1.0V	125.7uV	151.1uV	1.0V	1.0V	1.0V
/Data_Read<0>	149.5uV	1.0V	1.0V	149.1uV	1.0V	125.7uV	151.1uV	1.0V	149.3uV	1.0V

Figure 5.2.1: Output Waveform for Memory Function Test

Part 6 Optimization

6.1 Power Optimization

For improving the power changing in this CPU design, we implement multiple structures inside this design to minimized the power generated during the switching, and eliminated the open parts during the process.

6.1.1 Clock Gating for register file

Clock gating is a way to minimized the switching activities for our signal, in this part, we use the latch + AND gate structure in front of the signal to minimized the power consumption inside this pipeline design.

As shown below in Figure 6.1.1.1, we add this clock gating structure in front of the write Enable signal input to register file DFF to control the enable signal turn on and off. In this case, we can minimized the power format during switching.

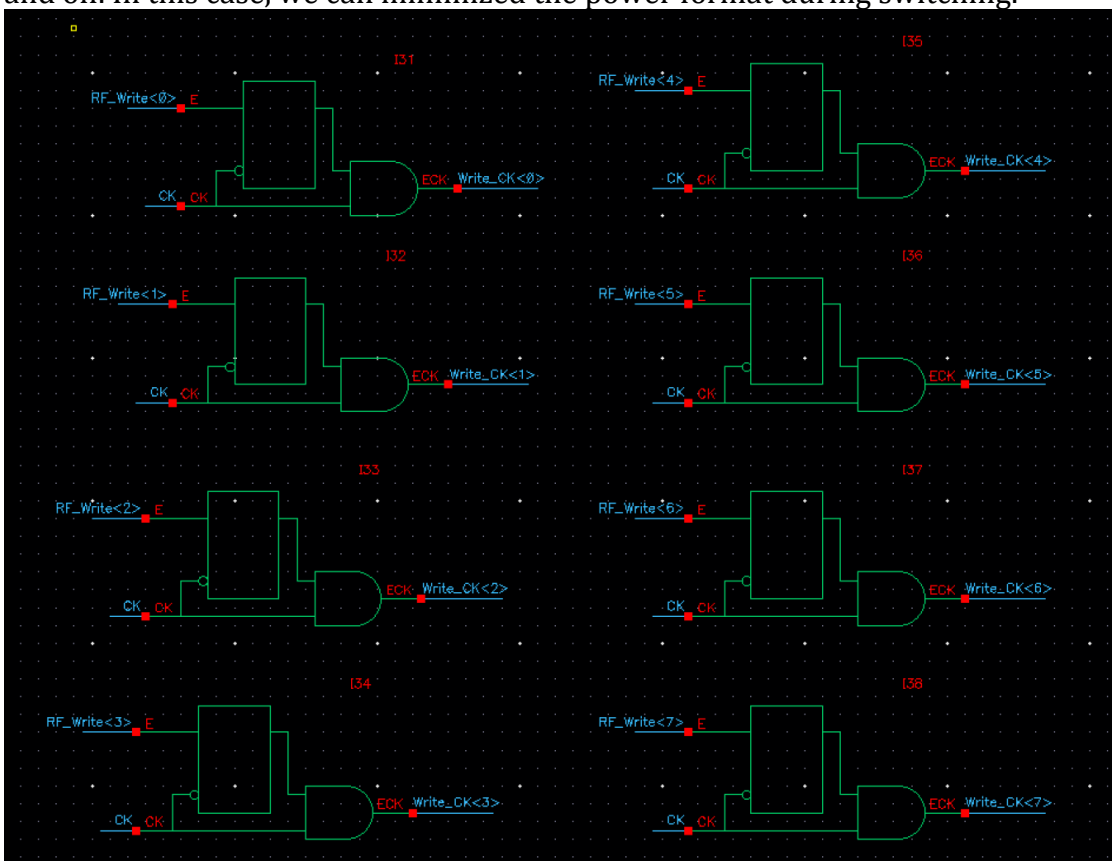


Figure 6.1.1.1: Clock Gating Design for Register File

6.1.2 Clock gating for SRAM Control Signal

Similar to the clock gating insider register file, we implement an NAND gate to NAND signal for clock, Decoder_EN and MEMEN signal to optimized the precharge cycle for memory stage shown below in Figure 6.1.1.2. The spike and period for memory precharge can be reduce, and power is decreasing.

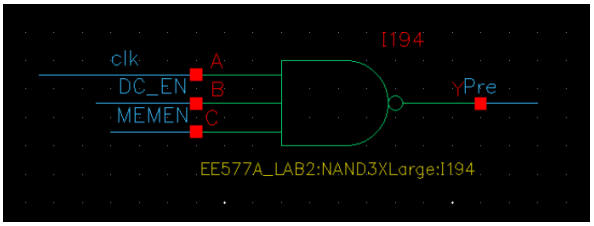


Figure 6.1.2.1: 3 input NAND gate Control Precharge Signal

6.1.3 Clock Gating for modules in ALU

For saving more power, we close part of the circuit inside ALU when the command is not require for that particular calculation. As shown in Figure 6.1.3.1 below, instead of turning on all parts of the ALU, the NAND gates will decode the ALU_OP address and turn on the EN pin for the proper circuit. In this optimization, most of the ALU parts can maintain close when not working.

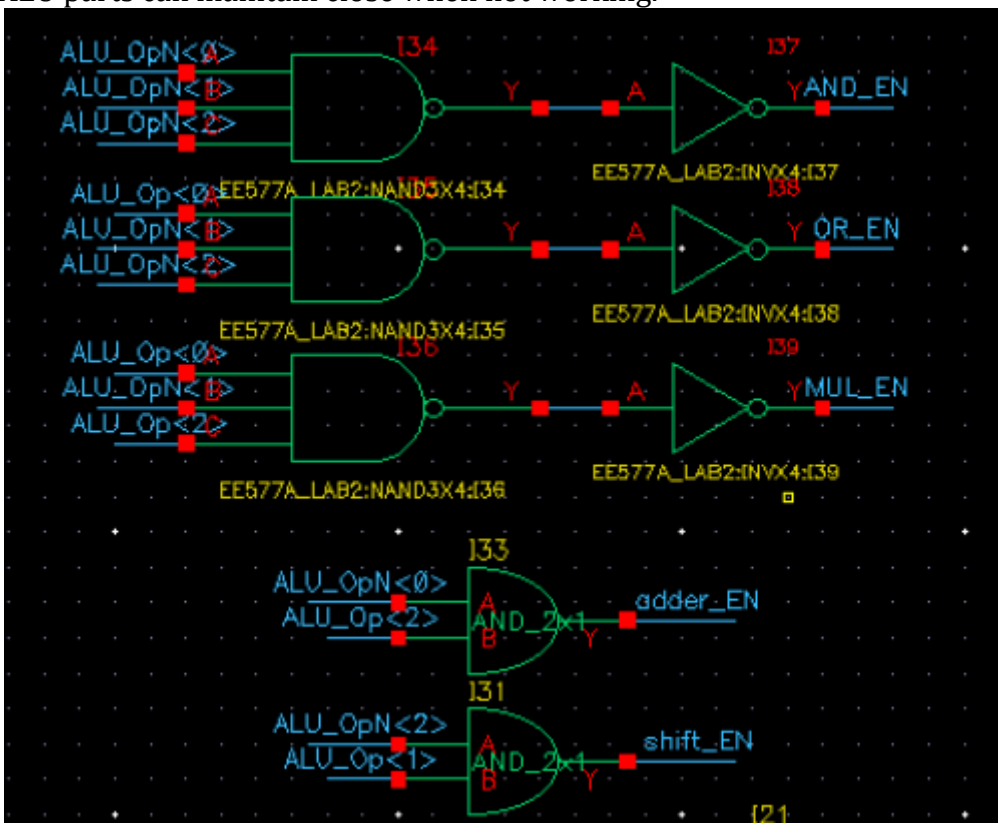


Figure 6.1.3.1: ALU Enable Signal For Power Reduction

6.1.4 DFF Optimization

For DFF optimization, we choose this DFF shown below in Figure 6.1.2.1 for all of our Register File and stage registers. This type of the DFF provides average power around 3.463E-6W per switching with a small layout area 5.814μm^2. From using this DFF, we can see the rising and fall time for the switching is around 62ps. Also, this structure provide us a CMOS logic for the DFF, and we do not need to consider the conflict caused by the setup and hold time dynamic DFF need to consider.

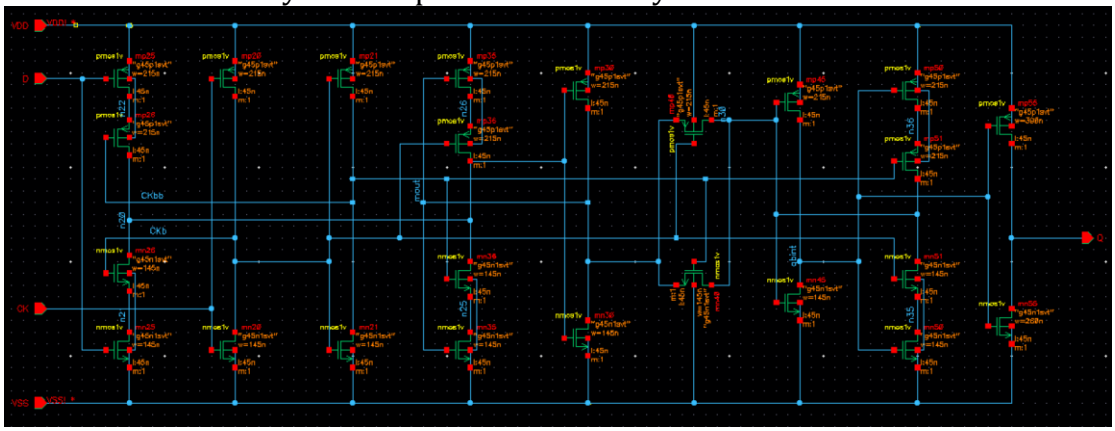


Figure 6.1.4.1: Schematic for DFF

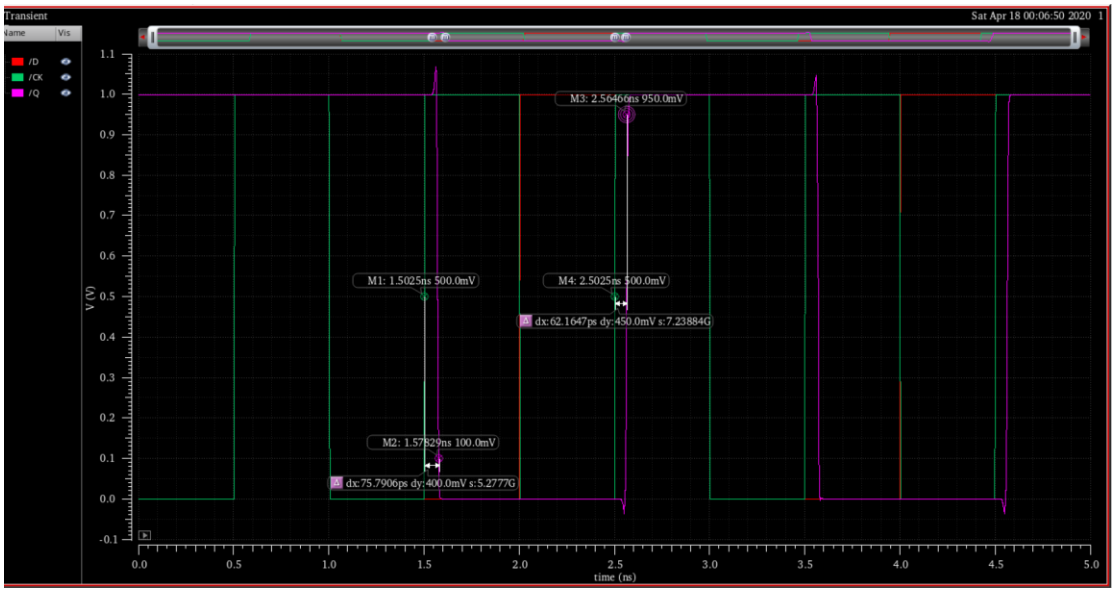


Figure 6.1.4.2: Functionality and Delay Test for this DFF

6.2 Clock Tree

For simulate closer to the realistic situation, we implement clock tree to distribute the ideal signal generated from the CLK pin to this pipeline system's clock circuitry. This

Clock Tree design in this circuit, we use Inverters to fanout the ideal reference clock signal cascaded and synthesize with each stage of our pipeline. Most of the clock tree optimization can be presented from layout structure.

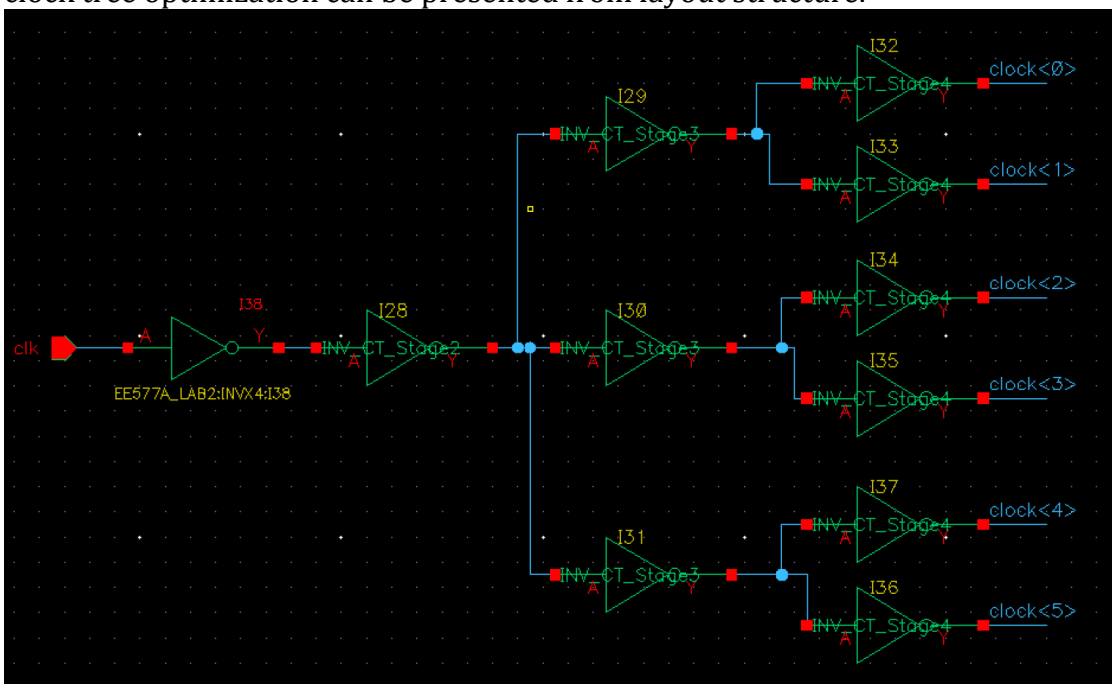


Figure 6.2.1: Clock Tree Design

As shown above in Figure 6.2.1, we split our clock structure into 6 different lines, and each lines will connect to capacitance around 32 DFFX1 gate capacitance. Thus, for achieved the design purpose, I calculate the design and have the length and width shown in the following table:

	Stage 1 INV	*Stage 2 INV*	*Stage 3 INV*	*Stage4 INV*
NMOS W/L [nm]	180/45	810/45	810/45	1225/45

PMOS W/L [nm]	260/45	1170/45	1215/45	2450/45
Multiplier	4	4	4	4

Part 7 Schematic Simulation

7.1 Functionality Test

Based on the design above, we can use the testbench generated in Part2 to test the functionality.

In Phase II, we optimize design to support 800ps clock period execution.

DIY Instruction Set

```
STOREI 0AH #001f
STOREI 0BH #0002
STOREI 2 10H #000F #00FE
LOAD $1 0AH
LOAD $2 0BH
LOAD $3 10H
LOAD $4 11H

MUL $5 $1 $2
ADD $6 $3 $4
NOP
STORE 00H $5
STORE 01H $6

SFL $5 $3 #0002
OR $6 $2 $4
AND $7 $5 $3
STORE 02H $5
STORE 03H $6
STORE 04H $7

MIN $1 $5 $6
MIN $7 $6 $5
NOP
STORE 05H $1
STORE 06H $7

MULI $5 $1 #000A
ADDI $6 $3 #1234
NOP
STORE 07H $5
STORE 08H $6

SFR $5 $3 #0003
ORI $6 $2 #000F
ANDI $7 $5 #FF00
STORE 0CH $5
STORE 0DH $6
STORE 0EH $7

LOAD $0 00H
LOAD $0 01H
LOAD $0 02H
LOAD $0 03H
LOAD $0 04H
LOAD $0 05H
LOAD $0 06H
LOAD $0 07H
LOAD $0 08H
LOAD $0 09H
LOAD $0 0AH
LOAD $0 0BH
LOAD $0 0CH
LOAD $0 0DH
LOAD $0 0EH
```

```
radix 1 1 1 1 1 1 1 111 111 111 111 4444 14
io i i i i i i i i i i i i i i
vname Reset clk WB MEMRead MEMWrite Reg_Dst ALU_Src ALU_Op<[2:0]> rs<[2:0]> rt<[2:0]> rd<[2:0]>
imme_num<[15:0]> addr<[4:0]>
tunit ns
slope 0.01
vih 1
vil 0

   0    0   0   0   0   0   0   0   000 000 000 000 0000    00
0.80    1   1   0   0   1   0   1   001 000 000 000 001F    0A  ;STOREI 0AH #001f
1.20    1   0   0   0   1   0   1   001 000 000 000 001F    0A  ;
1.60    1   1   0   0   1   0   1   001 000 000 000 0002    0B  ;STOREI 0BH #0002
2.00    1   0   0   0   1   0   1   001 000 000 000 0002    0B  ;
2.40    1   1   0   0   1   0   1   001 000 000 000 000F    10  ;STOREI 2 10H #000F #00FE
2.80    1   0   0   0   1   0   1   001 000 000 000 000F    10  ;
3.20    1   1   0   0   1   0   1   001 000 000 000 00FE    11  ;Inserted Burst STOREI Instruction
3.60    1   0   0   0   1   0   1   001 000 000 000 00FE    11  ;
4.00    1   1   1   1   0   0   0   000 000 001 000 0000    0A  ;LOAD $1 0AH
4.40    1   0   1   1   0   0   0   000 000 001 000 0000    0A  ;
4.80    1   1   1   1   0   0   0   000 000 010 000 0000    0B  ;LOAD $2 0BH
5.20    1   0   1   1   0   0   0   000 000 010 000 0000    0B  ;
5.60    1   1   1   1   0   0   0   000 000 011 000 0000    10  ;LOAD $3 10H
6.00    1   0   1   1   0   0   0   000 000 011 000 0000    10  ;
6.40    1   1   1   1   0   0   0   000 000 100 000 0000    11  ;LOAD $4 11H
6.80    1   0   1   1   0   0   0   000 000 100 000 0000    11  ;
7.20    1   1   0   0   0   0   0   000 000 000 000 0000    00  ;Inserted NOP
7.60    1   0   0   0   0   0   0   000 000 000 000 0000    00  ;
8.00    1   1   0   0   0   1   0   101 001 010 101 0000    00  ;MUL $5 $1 $2
8.40    1   0   0   0   0   1   0   101 001 010 101 0000    00  ;
```

```
30 8.80  1 1 1 0 0 0 0  000 000 101 000 0000  00  ;Inserted Second MUL Write Inst
31 9.20  1 0 1 0 0 0 0  000 000 101 000 0000  00  ;
32 9.60  1 1 1 0 0 1 0  100 011 100 110 0000  00  ;ADD $6 $3 $4
33 10.00 1 0 1 0 0 1 0  100 011 100 110 0000  00  ;
34 10.40 1 1 0 0 0 0 0  000 000 000 000 0000  00  ;NOP
35 10.80 1 0 0 0 0 0 0  000 000 000 000 0000  00  ;
36 11.20 1 1 0 0 1 0 0  001 101 000 000 0000  00  ;STORE 00H $5
37 11.60 1 0 0 0 1 0 0  001 101 000 000 0000  00  ;
38 12.00 1 1 0 0 1 0 0  001 110 000 000 0000  01  ;STORE 01H $6
39 12.40 1 0 0 0 1 0 0  001 110 000 000 0000  01  ;
40 12.80 1 1 1 0 0 0 1  010 011 101 000 0002  00  ;SFL $5 $3 #0002
41 13.20 1 0 1 0 0 0 1  010 011 101 000 0002  00  ;
42 13.60 1 1 0 0 0 0 0  000 000 000 000 0000  00  ;Inserted NOP
43 14.00 1 0 0 0 0 0 0  000 000 000 000 0000  00  ;
44 14.40 1 1 1 0 0 1 0  001 010 100 110 0000  00  ;OR $6 $2 $4
45 14.80 1 0 1 0 0 1 0  001 010 100 110 0000  00  ;
46 15.20 1 1 1 0 0 1 0  000 101 011 111 0000  00  ;AND $7 $5 $3
47 15.60 1 0 1 0 0 1 0  000 101 011 111 0000  00  ;
48 16.00 1 1 0 0 1 0 0  001 101 000 000 0000  02  ;STORE 02H $5
49 16.40 1 0 0 0 1 0 0  001 101 000 000 0000  02  ;
50 16.80 1 1 0 0 1 0 0  001 110 000 000 0000  03  ;STORE 03H $6
51 17.20 1 0 0 0 1 0 0  001 110 000 000 0000  03  ;
52 17.60 1 1 0 0 1 0 0  001 111 000 000 0000  04  ;STORE 04H $7
53 18.00 1 0 0 0 1 0 0  001 111 000 000 0000  04  ;
54 18.40 1 1 1 0 0 1 0  110 101 110 001 0000  00  ;MIN $1 $5 $6
55 18.80 1 0 1 0 0 1 0  110 101 110 001 0000  00  ;
56 19.20 1 1 1 0 0 1 0  110 110 101 111 0000  00  ;MIN $7 $6 $5
57 19.60 1 0 1 0 0 1 0  110 110 101 111 0000  00  ;
58 20.00 1 1 0 0 0 0 0  000 000 000 000 0000  00  ;NOP
59 20.40 1 0 0 0 0 0 0  000 000 000 000 0000  00  ;
```

```
58 20.00 1 1 0 0 0 0 0  000 000 000 000 0000  00  ;NOP
59 20.40 1 0 0 0 0 0 0  000 000 000 000 0000  00  ;
60 20.80 1 1 0 0 1 0 0  001 001 000 000 0000  05  ;STORE 05H $1
61 21.20 1 0 0 0 1 0 0  001 001 000 000 0000  05  ;
62 21.60 1 1 0 0 1 0 0  001 111 000 000 0000  06  ;STORE 06H $7
63 22.00 1 0 0 0 1 0 0  001 111 000 000 0000  06  ;
64 22.40 1 1 0 0 0 0 1  101 001 101 000 000A  00  ;MULI $5 $1 #000A
65 22.80 1 0 0 0 0 0 1  101 001 101 000 000A  00  ;
66 23.20 1 1 1 0 0 0 0  000 000 101 000 0000  00  ;Inserted Second MUL Write Inst
67 23.60 1 0 1 0 0 0 0  000 000 101 000 0000  00  ;
68 24.00 1 1 1 0 0 0 1  100 011 110 000 1234  00  ;ADDI $6 $3 #1234
69 24.40 1 0 1 0 0 0 1  100 011 110 000 1234  00  ;
70 24.80 1 1 0 0 0 0 0  000 000 000 000 0000  00  ;NOP
71 25.20 1 0 0 0 0 0 0  000 000 000 000 0000  00  ;
72 25.60 1 1 0 0 1 0 0  001 101 000 000 0000  07  ;STORE 07H $5
73 26.00 1 0 0 0 1 0 0  001 101 000 000 0000  07  ;
74 26.40 1 1 0 0 1 0 0  001 110 000 000 0000  08  ;STORE 08H $6
75 26.80 1 0 0 0 1 0 0  001 110 000 000 0000  08  ;
76 27.20 1 1 1 0 0 0 1  011 011 101 000 0003  00  ;SFR $5 $3 #0003
77 27.60 1 0 1 0 0 0 1  011 011 101 000 0003  00  ;
78 28.00 1 1 0 0 0 0 0  000 000 000 000 0000  00  ;Inserted NOP
79 28.40 1 0 0 0 0 0 0  000 000 000 000 0000  00  ;
80 28.80 1 1 1 0 0 0 1  001 010 110 000 000F  00  ;ORI $6 $2 #000F
81 29.20 1 0 1 0 0 0 1  001 010 110 000 000F  00  ;
82 29.60 1 1 1 0 0 0 1  000 101 111 000 FF00  00  ;ANDI $7 $5 #FF00
83 30.00 1 0 1 0 0 0 1  000 101 111 000 FF00  00  ;
84 30.40 1 1 0 0 1 0 0  001 101 000 000 0000  0C  ;STORE 0CH $5
85 30.80 1 0 0 0 1 0 0  001 101 000 000 0000  0C  ;
86 31.20 1 1 0 0 1 0 0  001 110 000 000 0000  0D  ;STORE 0DH $6
87 31.60 1 0 0 0 1 0 0  001 110 000 000 0000  0D  ;
```

```
88  32.00   1   1   0   0   1   0   0   001 111 000 000 0000    0E  ;STORE 0EH $7
89  32.40   1   0   0   0   1   0   0   001 111 000 000 0000    0E  ;
90  32.80   1   1   1   1   0   0   0   000 000 000 000 0000    00  ;LOAD $0 00H
91  33.20   1   0   1   1   0   0   0   000 000 000 000 0000    00  ;
92  33.60   1   1   1   1   0   0   0   000 000 000 000 0000    01  ;LOAD $0 01H
93  34.00   1   0   1   1   0   0   0   000 000 000 000 0000    01  ;
94  34.40   1   1   1   1   0   0   0   000 000 000 000 0000    02  ;LOAD $0 02H
95  34.80   1   0   1   1   0   0   0   000 000 000 000 0000    02  ;
96  35.20   1   1   1   1   0   0   0   000 000 000 000 0000    03  ;LOAD $0 03H
97  35.60   1   0   1   1   0   0   0   000 000 000 000 0000    03  ;
98  36.00   1   1   1   1   0   0   0   000 000 000 000 0000    04  ;LOAD $0 04H
99  36.40   1   0   1   1   0   0   0   000 000 000 000 0000    04  ;
00  36.80   1   1   1   1   0   0   0   000 000 000 000 0000    05  ;LOAD $0 05H
01  37.20   1   0   1   1   0   0   0   000 000 000 000 0000    05  ;
02  37.60   1   1   1   1   0   0   0   000 000 000 000 0000    06  ;LOAD $0 06H
03  38.00   1   0   1   1   0   0   0   000 000 000 000 0000    06  ;
04  38.40   1   1   1   1   0   0   0   000 000 000 000 0000    07  ;LOAD $0 07H
05  38.80   1   0   1   1   0   0   0   000 000 000 000 0000    07  ;
06  39.20   1   1   1   1   0   0   0   000 000 000 000 0000    08  ;LOAD $0 08H
07  39.60   1   0   1   1   0   0   0   000 000 000 000 0000    08  ;
08  40.00   1   1   1   1   0   0   0   000 000 000 000 0000    09  ;LOAD $0 09H
09  40.40   1   0   1   1   0   0   0   000 000 000 000 0000    09  ;
10  40.80   1   1   1   1   0   0   0   000 000 000 000 0000    0A  ;LOAD $0 0AH
11  41.20   1   0   1   1   0   0   0   000 000 000 000 0000    0A  ;
12  41.60   1   1   1   1   0   0   0   000 000 000 000 0000    0B  ;LOAD $0 0BH
13  42.00   1   0   1   1   0   0   0   000 000 000 000 0000    0B  ;
14  42.40   1   1   1   1   0   0   0   000 000 000 000 0000    0C  ;LOAD $0 0CH
15  42.80   1   0   1   1   0   0   0   000 000 000 000 0000    0C  ;
16  43.20   1   1   1   1   0   0   0   000 000 000 000 0000    0D  ;LOAD $0 0DH
17  43.60   1   0   1   1   0   0   0   000 000 000 000 0000    0D  ;
```

Demo Instruction Set

```
1  STOREI 0AH #ff0f
2  STOREI 0BH #0014
3  STOREI 2 18H #000B #00EE
4  LOAD $7 0AH
5  LOAD $2 0BH
6  LOAD $3 19H
7  LOAD $4 18H
8  MUL $5 $7 $2
9  MUL $6 $3 $5
10 MIN $1 $7 $2
11 STORE 1FH $5
12 STORE 00H $6
13 SFL $5 $3 #0005
14 OR $6 $5 $4
15 ANDI $6 $6 #00CC
16 ADD $6 $6 $4
17 STORE 09H $5
18 STORE 10H $6
19 LOAD $1 00H
20 LOAD $1 1FH
21 LOAD $1 09H
22 LOAD $1 10H
```

```
1  radix 1 1 1 1 1 1 1 111 111 111 111 4444 14
2  io i i i i i i i i i i i i i i i
3  vname Reset clk WB MEMRead MEMWrite Reg_Dst ALU_Src ALU_Op<[2:0]> rs<[2:0]> rt<[2:0]> rd<[2:0]> imme_num<[15:0]> addr<[4:0]>
4  tunit ns
5  slope 0.01
6  vih 1
7  vil 0
8
9     0    0   0   0   0   0   0   000 000 000 000 0000    00
10 0.80    1   1   0   0   1   0   1   001 000 000 000 FF0F    0A  ;STOREI 0AH #ff0f
11 1.20    1   0   0   0   1   0   1   001 000 000 000 FF0F    0A  ;
12 1.60    1   1   0   0   1   0   1   001 000 000 000 0014    0B  ;STOREI 0BH #0014
13 2.00    1   0   0   0   1   0   1   001 000 000 000 0014    0B  ;
14 2.40    1   1   0   0   1   0   1   001 000 000 000 000B    18  ;STOREI 2 18H #000B #00EE
15 2.80    1   0   0   0   1   0   1   001 000 000 000 000B    18  ;
16 3.20    1   1   0   0   1   0   1   001 000 000 000 00EE    19  ;Inserted Burst STOREI Instruction
17 3.60    1   0   0   0   1   0   1   001 000 000 000 00EE    19  ;
18 4.00    1   1   1   1   0   0   0   000 000 111 000 0000    0A  ;LOAD $7 0AH
19 4.40    1   0   1   1   0   0   0   000 000 111 000 0000    0A  ;
20 4.80    1   1   1   1   0   0   0   000 000 010 000 0000    0B  ;LOAD $2 0BH
21 5.20    1   0   1   1   0   0   0   000 000 010 000 0000    0B  ;
22 5.60    1   1   1   1   0   0   0   000 000 011 000 0000    19  ;LOAD $3 19H
23 6.00    1   0   1   1   0   0   0   000 000 011 000 0000    19  ;
24 6.40    1   1   1   1   0   0   0   000 000 100 000 0000    18  ;LOAD $4 18H
25 6.80    1   0   1   1   0   0   0   000 000 100 000 0000    18  ;
26 7.20    1   1   0   0   0   1   0   101 111 010 101 0000    00  ;MUL $5 $7 $2
27 7.60    1   0   0   0   0   1   0   101 111 010 101 0000    00  ;
28 8.00    1   1   1   0   0   0   0   000 000 101 000 0000    00  ;Inserted Second MUL Write Inst
29 8.40    1   0   1   0   0   0   0   000 000 101 000 0000    00  ;
30 8.80    1   1   0   0   0   0   0   000 000 000 000 0000    00  ;Inserted NOP
31 9.20    1   0   0   0   0   0   0   000 000 000 000 0000    00  ;
32 9.60    1   1   0   0   0   0   0   000 000 000 000 0000    00  ;Inserted NOP
33 10.00   1   0   0   0   0   0   0   000 000 000 000 0000    00  ;
34 10.40   1   1   0   0   0   1   0   101 011 101 110 0000    00  ;MUL $6 $3 $5
35 10.80   1   0   0   0   0   1   0   101 011 101 110 0000    00  ;
36 11.20   1   1   1   0   0   0   0   000 000 110 000 0000    00  ;Inserted Second MUL Write Inst
37 11.60   1   0   1   0   0   0   0   000 000 110 000 0000    00  ;
38 12.00   1   1   1   0   0   1   0   110 111 010 001 0000    00  ;MIN $1 $7 $2
39 12.40   1   0   1   0   0   1   0   110 111 010 001 0000    00  ;
40 12.80   1   1   0   0   1   0   0   001 101 000 000 0000    1F  ;STORE 1FH $5
```

```
40 12.80   1  1  0  0  1  0  0   001 101 000 000 0000   1F  ;STORE 1FH $5
41 13.20   1  0  0  0  1  0  0   001 101 000 000 0000   1F  ;
42 13.60   1  1  0  0  1  0  0   001 110 000 000 0000   00  ;STORE 00H $6
43 14.00   1  0  0  0  1  0  0   001 110 000 000 0000   00  ;
44 14.40   1  1  1  0  0  0  1   010 011 101 000 0005   00  ;SFL $5 $3 #0005
45 14.80   1  0  1  0  0  0  1   010 011 101 000 0005   00  ;
46 15.20   1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
47 15.60   1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
48 16.00   1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
49 16.40   1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
50 16.80   1  1  1  0  0  1  0   001 101 100 110 0000   00  ;OR $6 $5 $4
51 17.20   1  0  1  0  0  1  0   001 101 100 110 0000   00  ;
52 17.60   1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
53 18.00   1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
54 18.40   1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
55 18.80   1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
56 19.20   1  1  1  0  0  0  1   000 110 110 000 00CC   00  ;ANDI $6 $6 #00CC
57 19.60   1  0  1  0  0  0  1   000 110 110 000 00CC   00  ;
58 20.00   1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
59 20.40   1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
60 20.80   1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
61 21.20   1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
62 21.60   1  1  1  0  0  1  0   100 110 100 110 0000   00  ;ADD $6 $6 $4
63 22.00   1  0  1  0  0  1  0   100 110 100 110 0000   00  ;
64 22.40   1  1  0  0  1  0  0   001 101 000 000 0000   09  ;STORE 09H $5
65 22.80   1  0  0  0  1  0  0   001 101 000 000 0000   09  ;
66 23.20   1  1  0  0  0  0  0   000 000 000 000 0000   00  ;Inserted NOP
67 23.60   1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
68 24.00   1  1  0  0  1  0  0   001 110 000 000 0000   10  ;STORE 10H $6
69 24.40   1  0  0  0  1  0  0   001 110 000 000 0000   10  ;
70 24.80   1  1  1  1  0  0  0   000 000 001 000 0000   00  ;LOAD $1 00H
71 25.20   1  0  1  1  0  0  0   000 000 001 000 0000   00  ;
72 25.60   1  1  1  1  0  0  0   000 000 001 000 0000   1F  ;LOAD $1 1FH
73 26.00   1  0  1  1  0  0  0   000 000 001 000 0000   1F  ;
74 26.40   1  1  1  1  0  0  0   000 000 001 000 0000   09  ;LOAD $1 09H
75 26.80   1  0  1  1  0  0  0   000 000 001 000 0000   09  ;
76 27.20   1  1  1  1  0  0  0   000 000 001 000 0000   10  ;LOAD $1 10H
77 27.60   1  0  1  1  0  0  0   000 000 001 000 0000   10  ;
78 28.00   1  1  0  0  0  0  0   000 000 000 000 0000   00  ;NOP
79 28.40   1  0  0  0  0  0  0   000 000 000 000 0000   00  ;
```

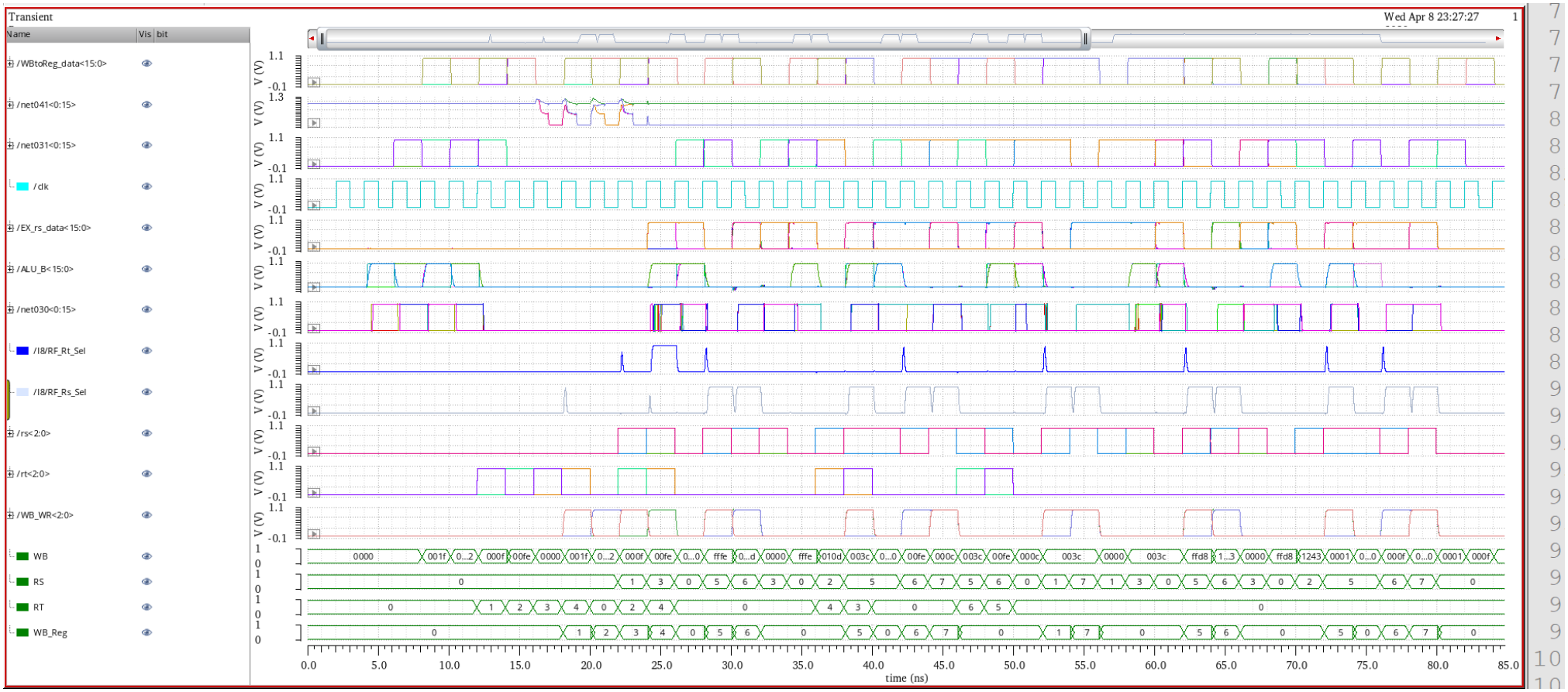
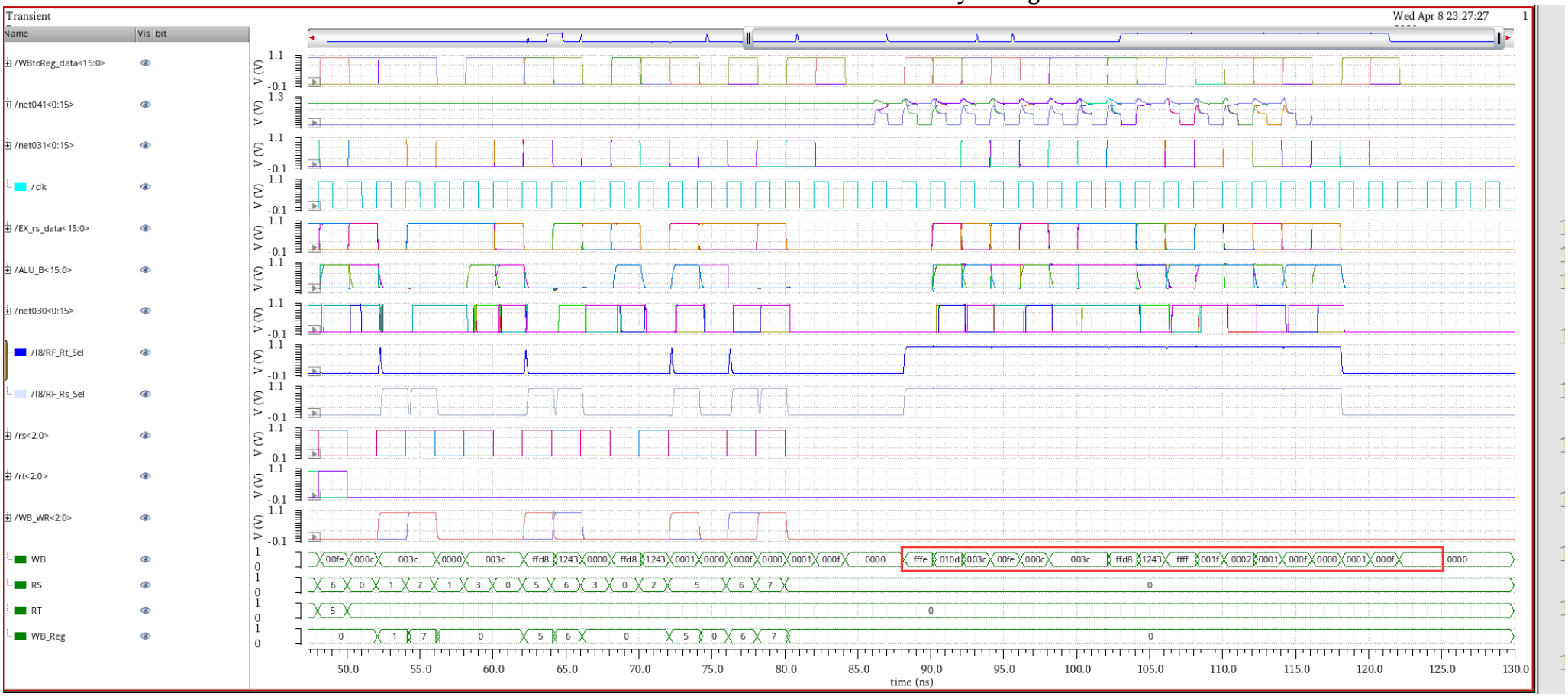


Figure 7.1: Output Waveform of CPU Schematic Design – 2ns clock period – DIY Instruction Set

The simulation schematic is shown below. We can check the 00H~0EH data value in Memory with golden result file.




```
#########################  MEM Content  #########################
   0H:Decimal:    -2     Hex: FFFE    |      1H:Decimal:  269     Hex: 010D    |
   2H:Decimal:    60     Hex: 003C    |      3H:Decimal:  254     Hex: 00FE    |
   4H:Decimal:    12     Hex: 000C    |      5H:Decimal:   60     Hex: 003C    |
   6H:Decimal:    60     Hex: 003C    |      7H:Decimal:  -40     Hex: FFD8    |
   8H:Decimal:  4675     Hex: 1243    |      9H:Decimal:    0     Hex: 0000    |
   AH:Decimal:    31     Hex: 001F    |      BH:Decimal:    2     Hex: 0002    |
   CH:Decimal:     1     Hex: 0001    |      DH:Decimal:   15     Hex: 000F    |
   EH:Decimal:     0     Hex: 0000    |      FH:Decimal:    0     Hex: 0000    |
```

Figure 7.2:2ns period CPU Schematic Design Simulation Compared with Golden Result File – DIY Instruction Set

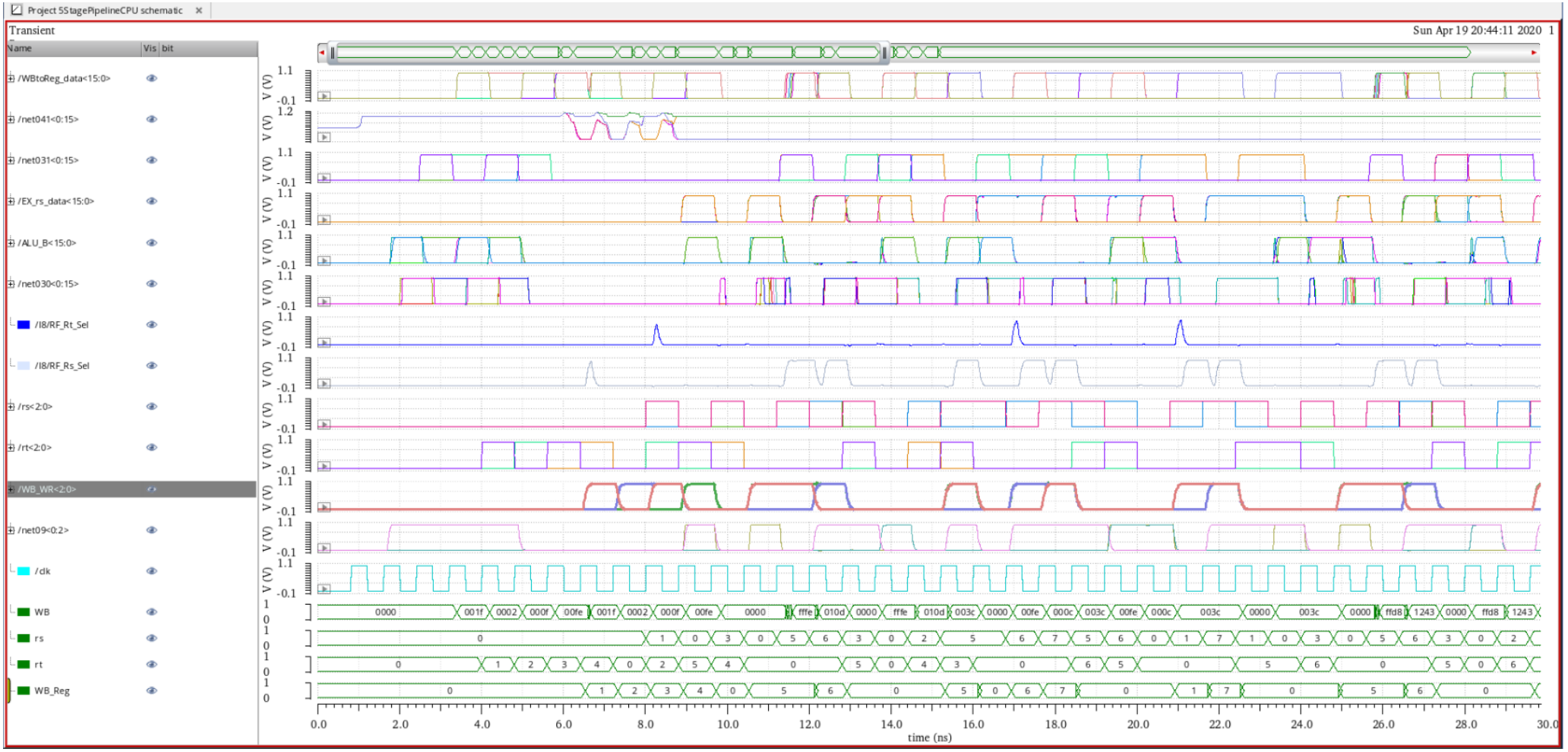
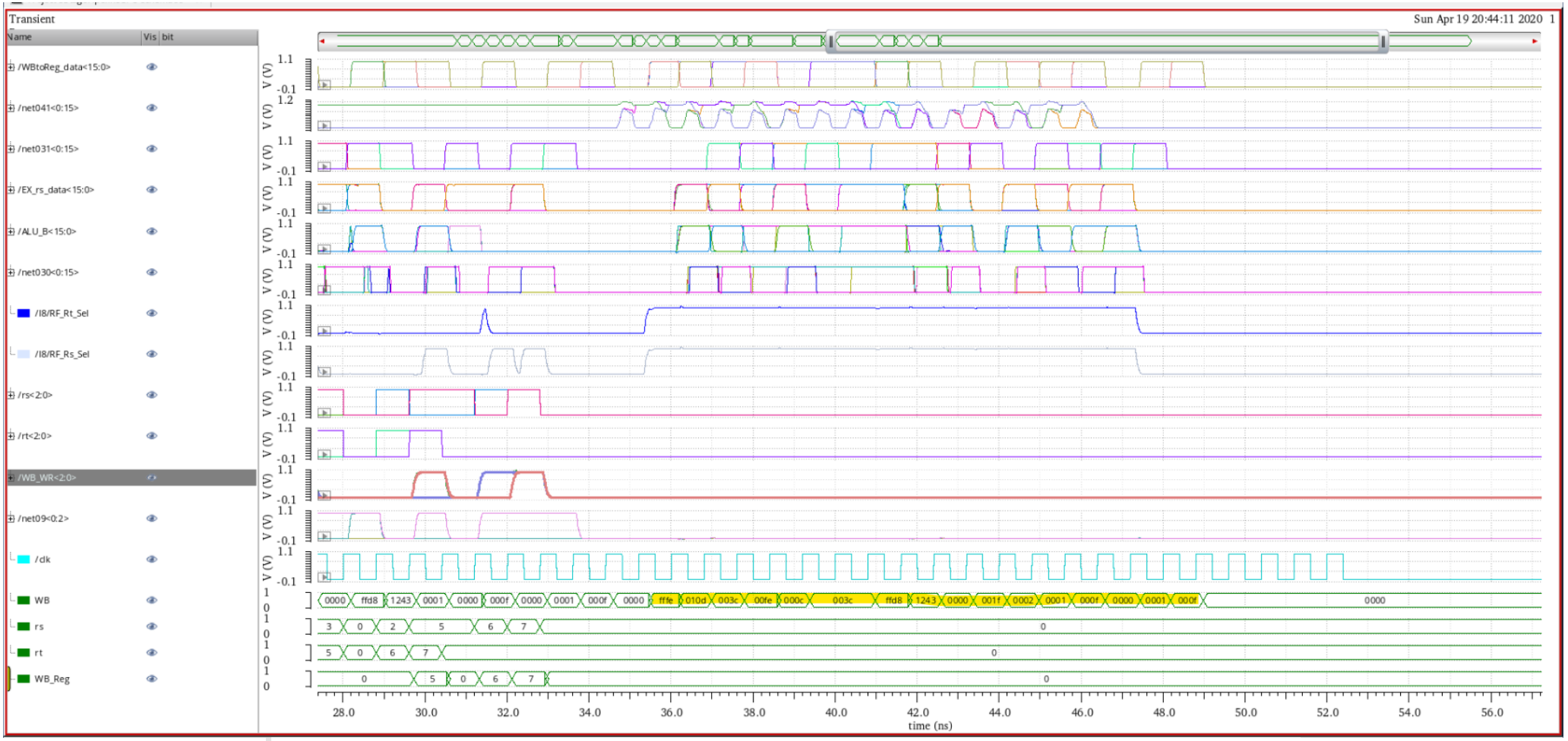
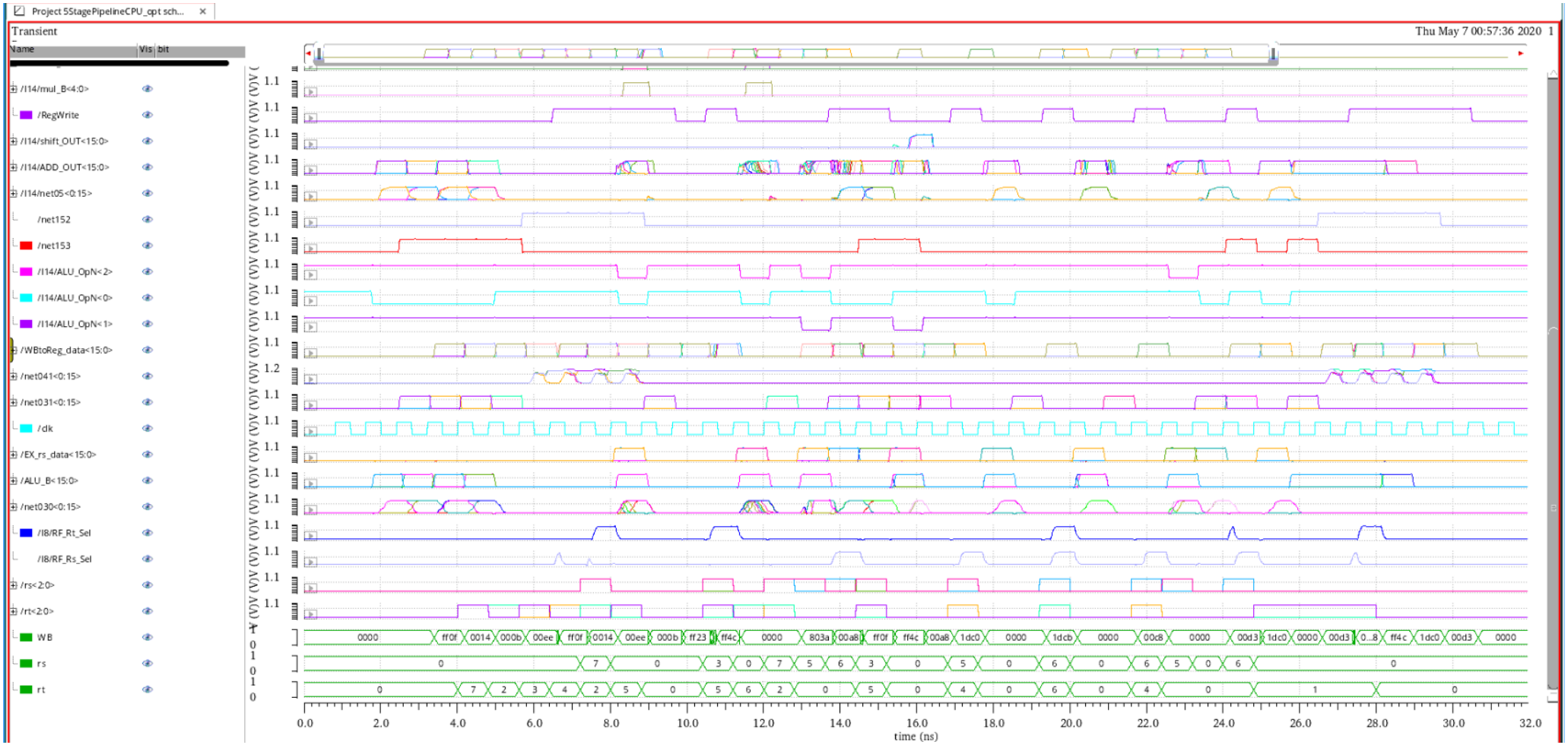


Figure 7.3: Output Waveform of CPU Schematic Design – 800ps clock period – DIY Instruction Set




```
##########################  MEM Content  ##########################
   0H:Decimal:    -2      Hex: FFFE     |      1H:Decimal:  269      Hex: 010D     |
   2H:Decimal:    60      Hex: 003C     |      3H:Decimal:  254      Hex: 00FE     |
   4H:Decimal:    12      Hex: 000C     |      5H:Decimal:   60      Hex: 003C     |
   6H:Decimal:    60      Hex: 003C     |      7H:Decimal:  -40      Hex: FFD8     |
   8H:Decimal:  4675      Hex: 1243     |      9H:Decimal:    0      Hex: 0000     |
   AH:Decimal:    31      Hex: 001F     |      BH:Decimal:    2      Hex: 0002     |
   CH:Decimal:     1      Hex: 0001     |      DH:Decimal:   15      Hex: 000F     |
   EH:Decimal:     0      Hex: 0000     |      FH:Decimal:    0      Hex: 0000     |
```

Figure 7.4:800ps period CPU Schematic Design Simulation Compared with Golden Result File – DIY Instruction Set


Project 5StagePipelineCPU_cpt sch...
Transient
Thu May 7 00:57:36 2020 1
Name
Vis
bit
/I14/mul_B<4:0>
/RegWrite
/I14/shift_OUT<15:0>
/I14/ADD_OUT<15:0>
/I14/net05<0:15>
/net152
/net153
/I14/ALU_OpN<2>
/I14/ALU_OpN<0>
/I14/ALU_OpN<1>
/WBtoReg_data<15:0>
/net041<0:15>
/net031<0:15>
/clk
/EX_rs_data<15:0>
/ALU_B<15:0>
/net030<0:15>
/I8/RF_Rt_Sel
/I8/RF_Rs_Sel
/rs<2:0>
/rt<2:0>
WB
rs
rt
0.0
2.0
4.0
6.0
8.0
10.0
12.0
14.0
16.0
18.0
20.0
22.0
24.0
26.0
28.0
30.0
32.0
time (ns)

```
#######################  RF Content  #######################
  $0:  Decimal:    0  Hex: 0000
  $1:  Decimal:  211  Hex: 00D3
  $2:  Decimal:   20  Hex: 0014
  $3:  Decimal:  238  Hex: 00EE
  $4:  Decimal:   11  Hex: 000B
  $5:  Decimal: 7616  Hex: 1DC0
  $6:  Decimal:  211  Hex: 00D3
  $7:  Decimal: -241  Hex: FF0F

#######################  MEM Content  #######################
   0H:Decimal:  168     Hex: 00A8    |      1H:Decimal:    0     Hex: 0000    |      2H:Decimal:    0     Hex: 0000    |      3H:Decimal:    0     Hex: 0000    |
   4H:Decimal:    0     Hex: 0000    |      5H:Decimal:    0     Hex: 0000    |      6H:Decimal:    0     Hex: 0000    |      7H:Decimal:    0     Hex: 0000    |
   8H:Decimal:    0     Hex: 0000    |      9H:Decimal: 7616     Hex: 1DC0    |      AH:Decimal: -241     Hex: FF0F    |      BH:Decimal:   20     Hex: 0014    |
   CH:Decimal:    0     Hex: 0000    |      DH:Decimal:    0     Hex: 0000    |      EH:Decimal:    0     Hex: 0000    |      FH:Decimal:    0     Hex: 0000    |
  10H:Decimal:  211     Hex: 00D3    |     11H:Decimal:    0     Hex: 0000    |     12H:Decimal:    0     Hex: 0000    |     13H:Decimal:    0     Hex: 0000    |
  14H:Decimal:    0     Hex: 0000    |     15H:Decimal:    0     Hex: 0000    |     16H:Decimal:    0     Hex: 0000    |     17H:Decimal:    0     Hex: 0000    |
  18H:Decimal:   11     Hex: 000B    |     19H:Decimal:  238     Hex: 00EE    |     1AH:Decimal:    0     Hex: 0000    |     1BH:Decimal:    0     Hex: 0000    |
  1CH:Decimal:    0     Hex: 0000    |     1DH:Decimal:    0     Hex: 0000    |     1EH:Decimal:    0     Hex: 0000    |     1FH:Decimal: -180     Hex: FF4C    |
```

Figure 7.5:800ps period CPU Schematic Design Simulation Compared with Golden Result File – Demo Instruction Set

7.2 Power Measurement

After the simulation, we send in the power output inside the calculator and use the average function to measure the average power for this CPU, get the result shown below in Figure 7.5, which is 1.134mW.

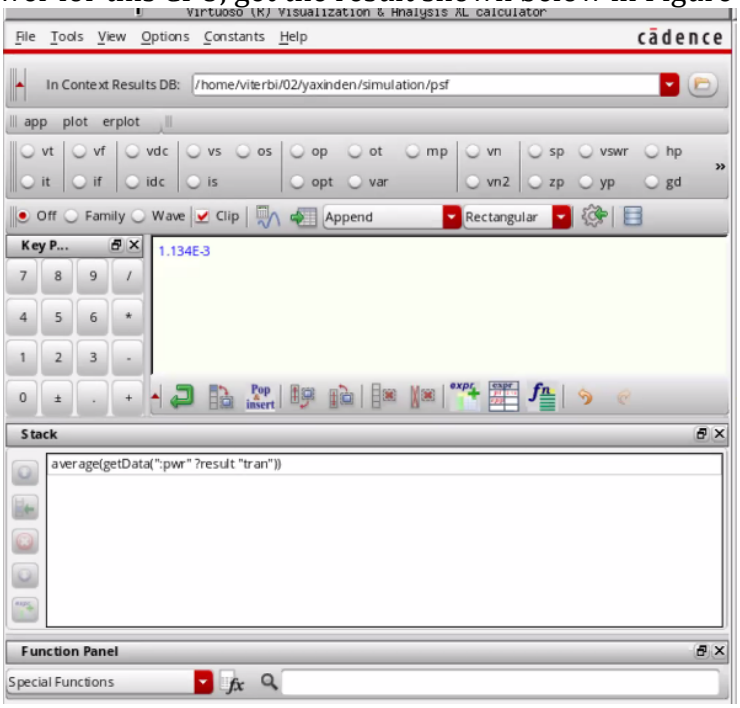


Figure 7.5: Average Power Measurement for CPU Schematic

Part 8 Layout

8.1 MEM Stage Layout

After added and optimized the control signal to SRAM, we reformate the layout for MEM stage shown below in Figure 8.1.1 below. This including the control signal circuit and SRAM for MEM stage. And the total area for MEM stage is 35.08 ×47.88 = 1679.6304um^2. LVS and DRC are check in Figure 8.1.2 and 8.1.3 below.

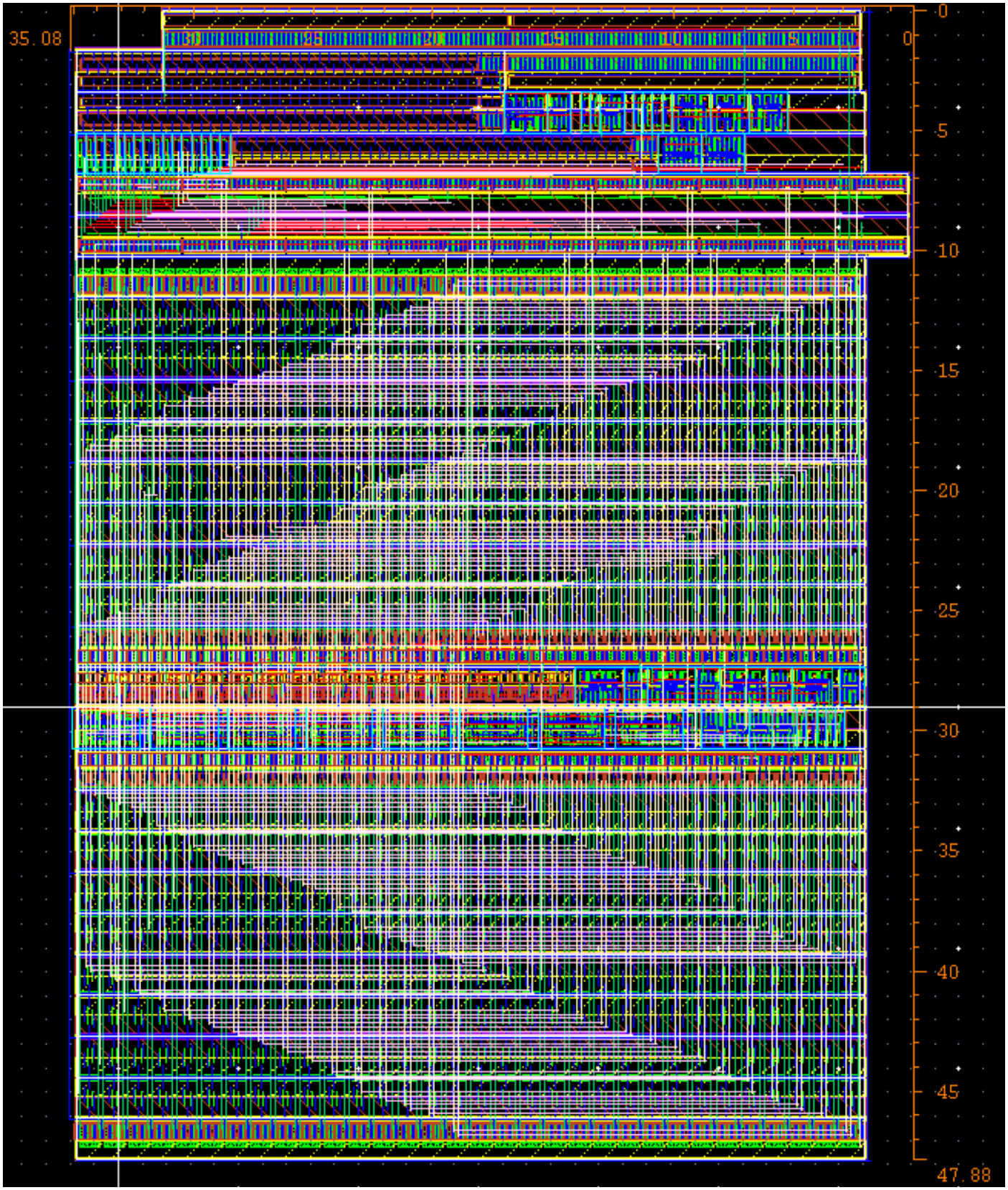


Figure 8.1.1: Total Layout for SRAM

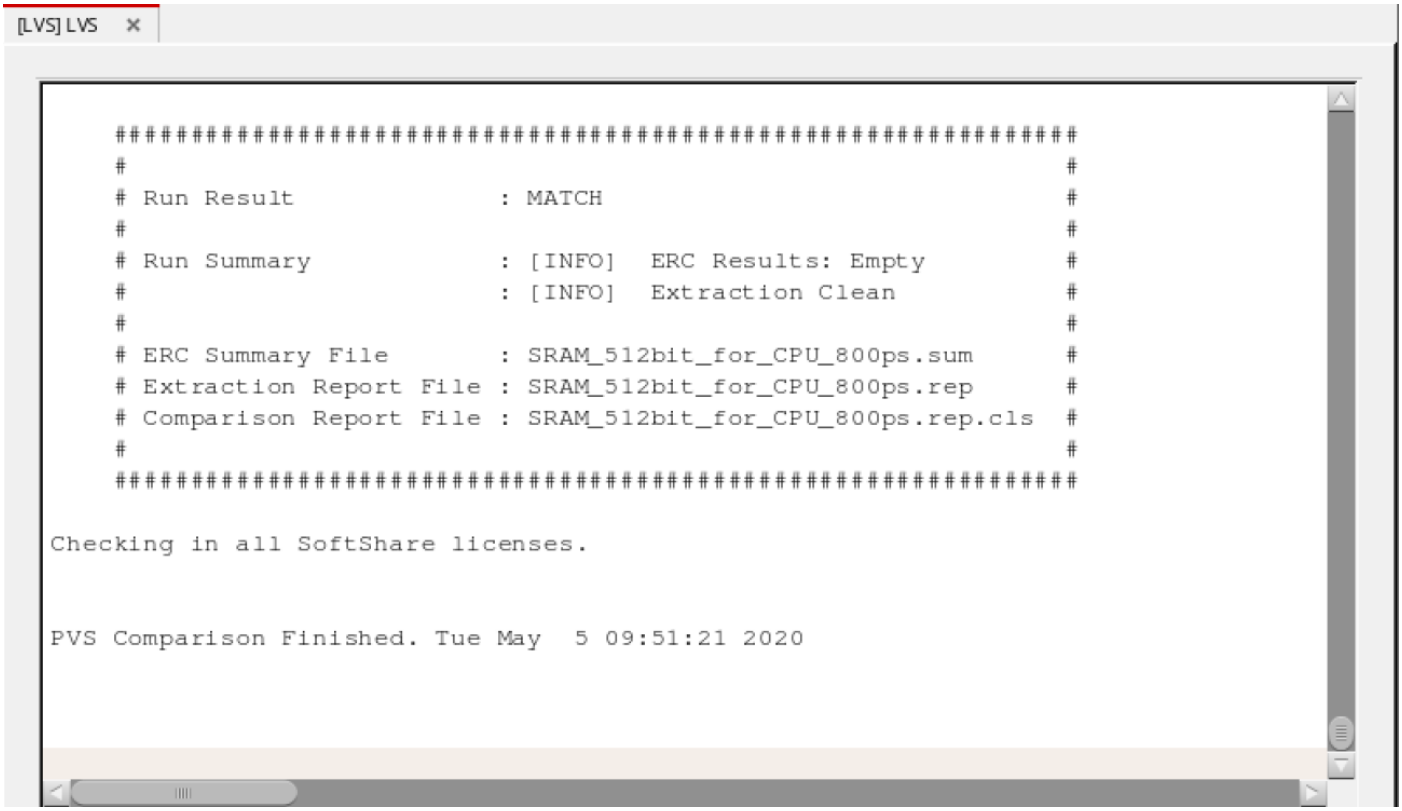


Figure 8.1.2: LVS for MEM stage

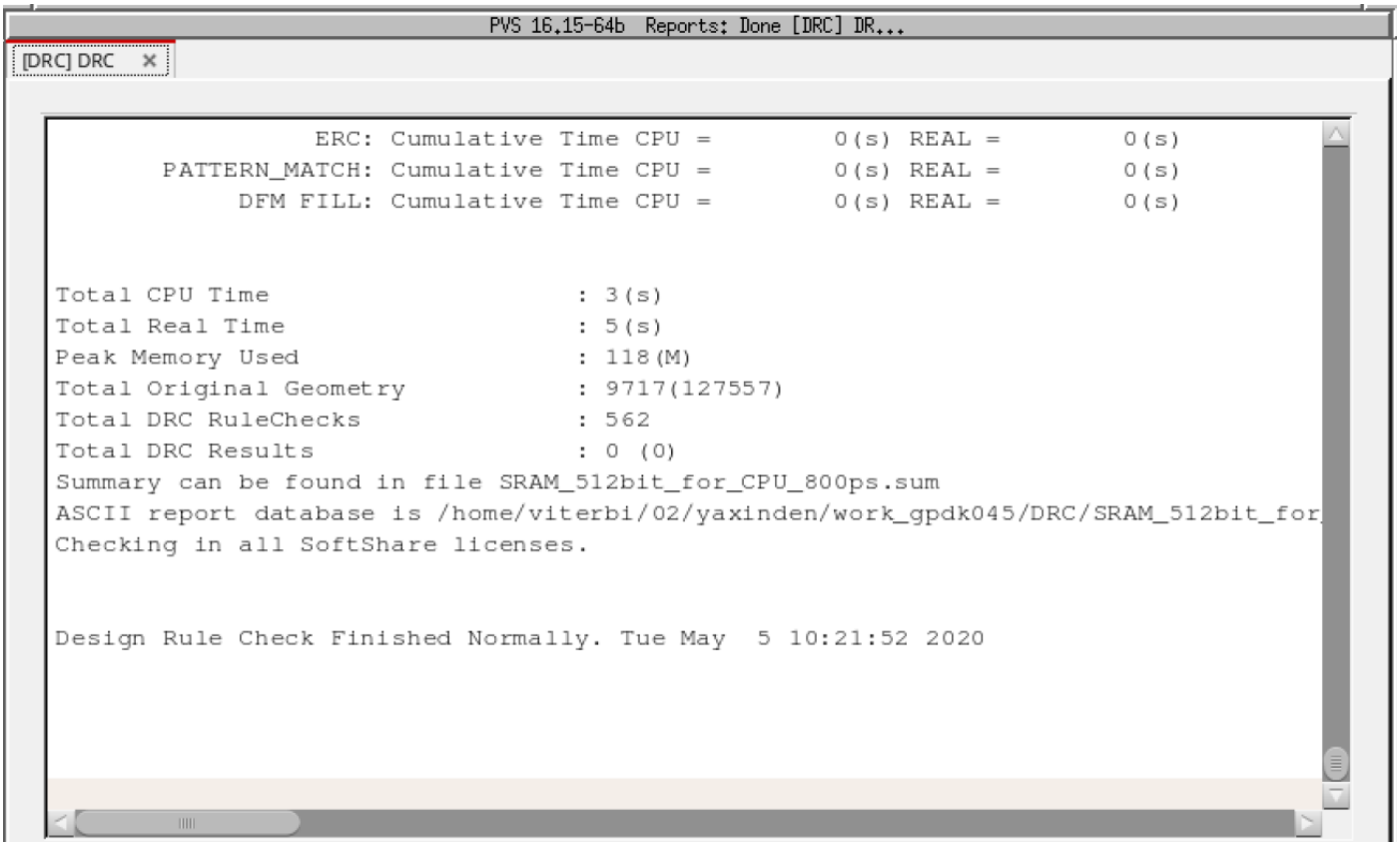


Figure 8.1.3: DRC for MEM stage

8.2 ALU Layout

8.2.1 Multiplier

For design our pipeline, we separate the Multiplier loop to the 4 stage pipeline design, and layout is shown below in Figure 8.2.1.

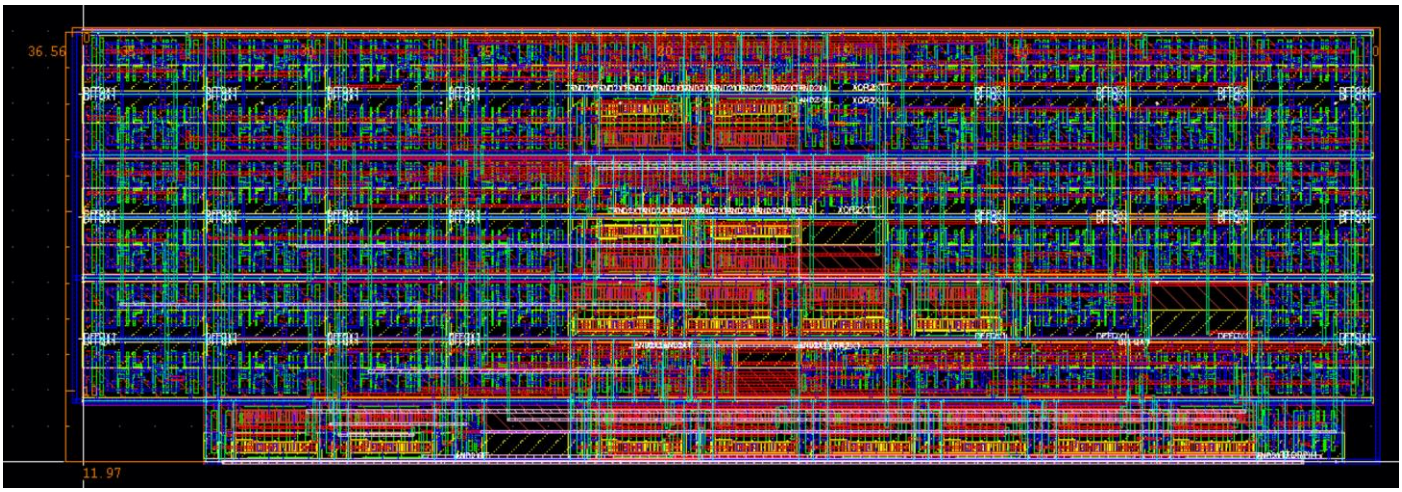


Figure 8.2.1: Multiplier Layout

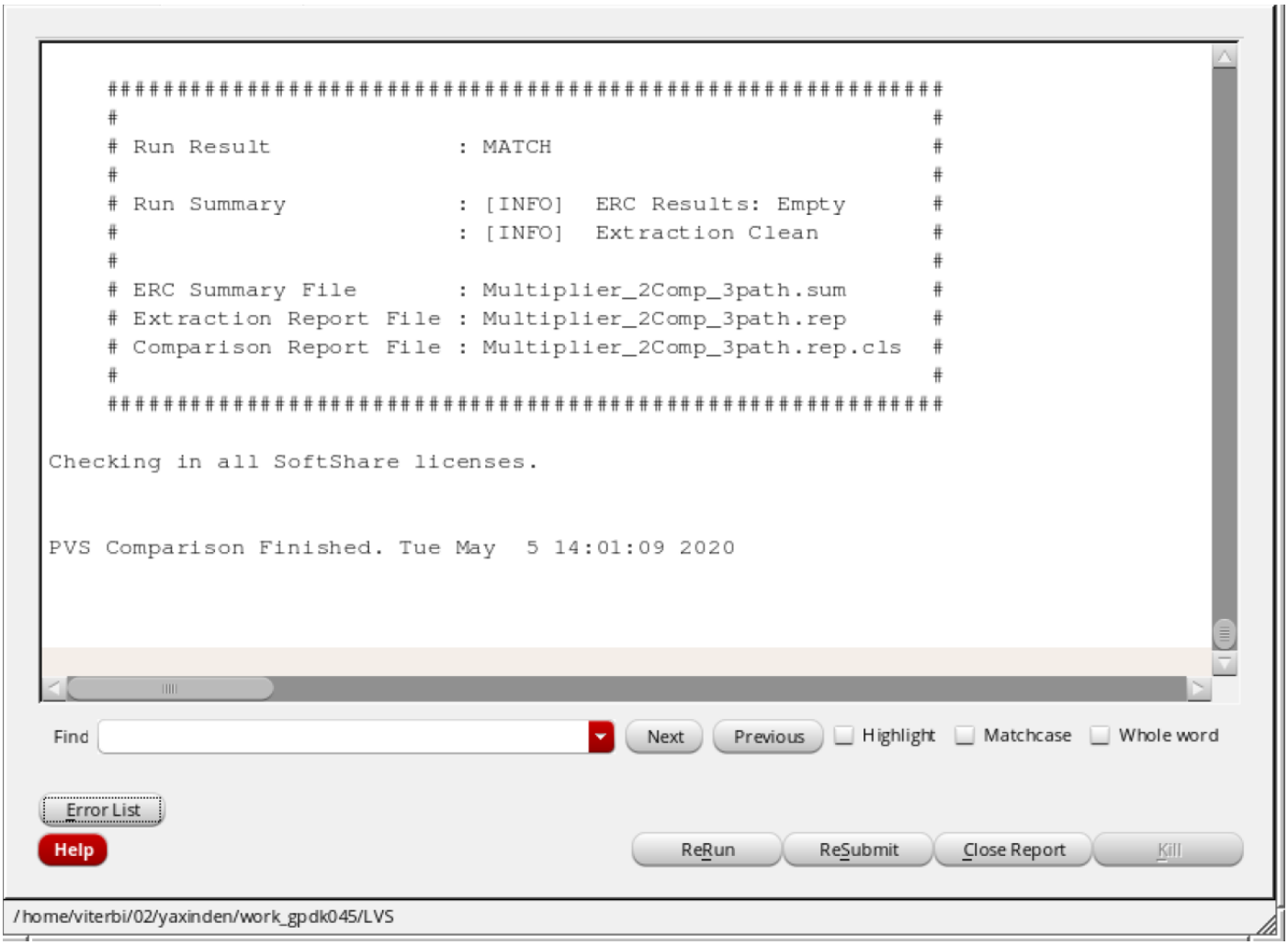


Figure 8.2.2: LVS for Multiplier

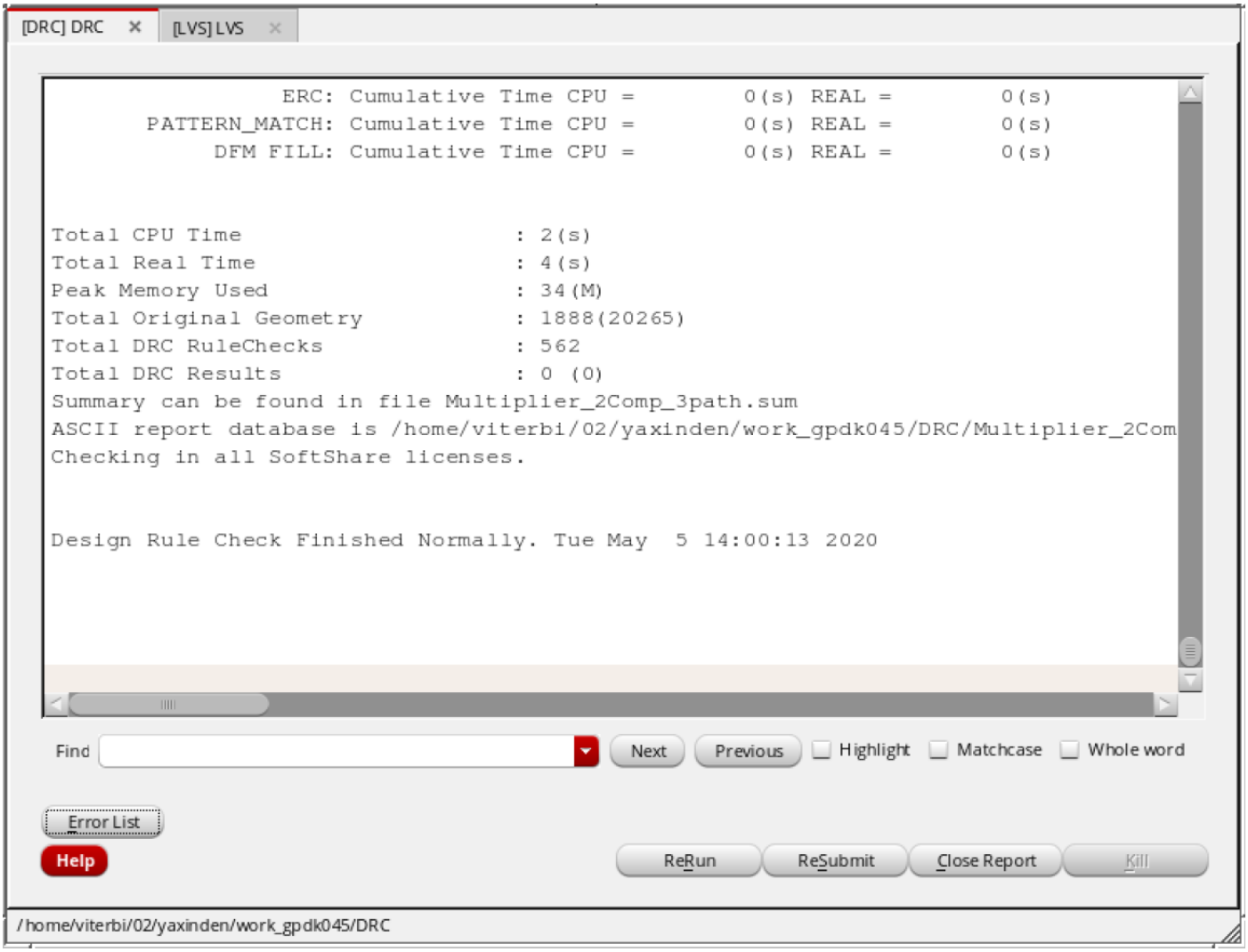
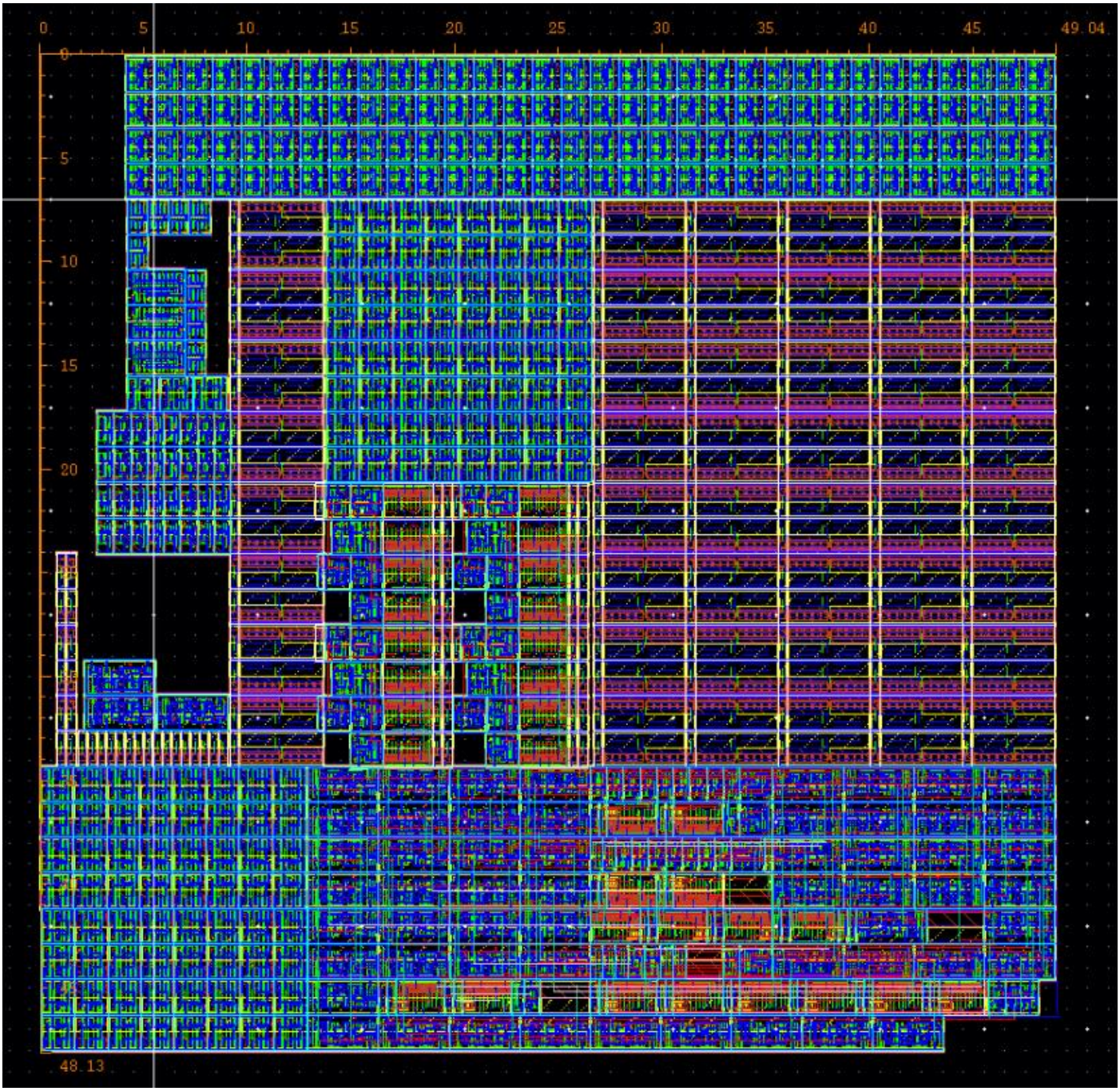
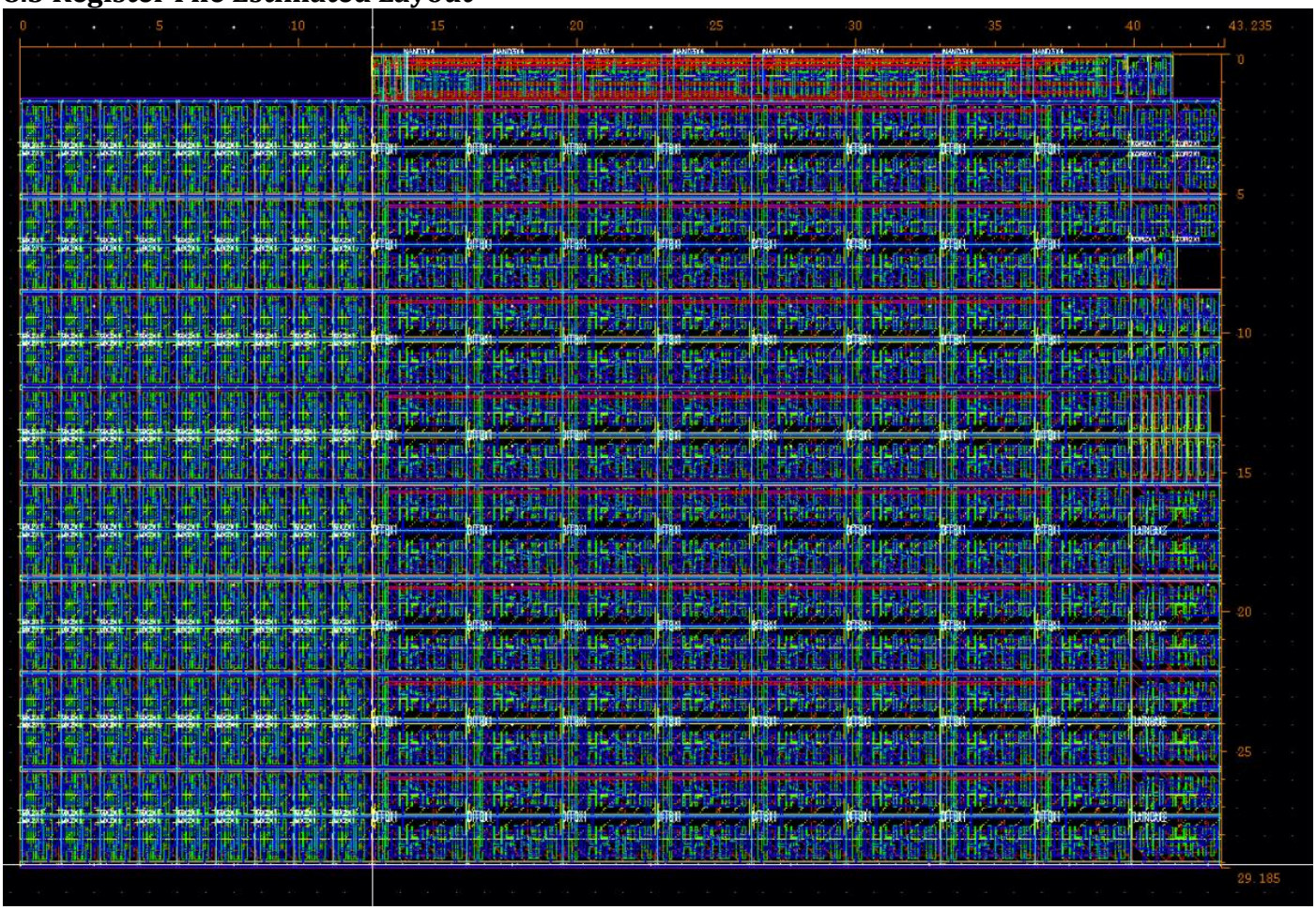


Figure 8.2.3: DRC for Multiplier

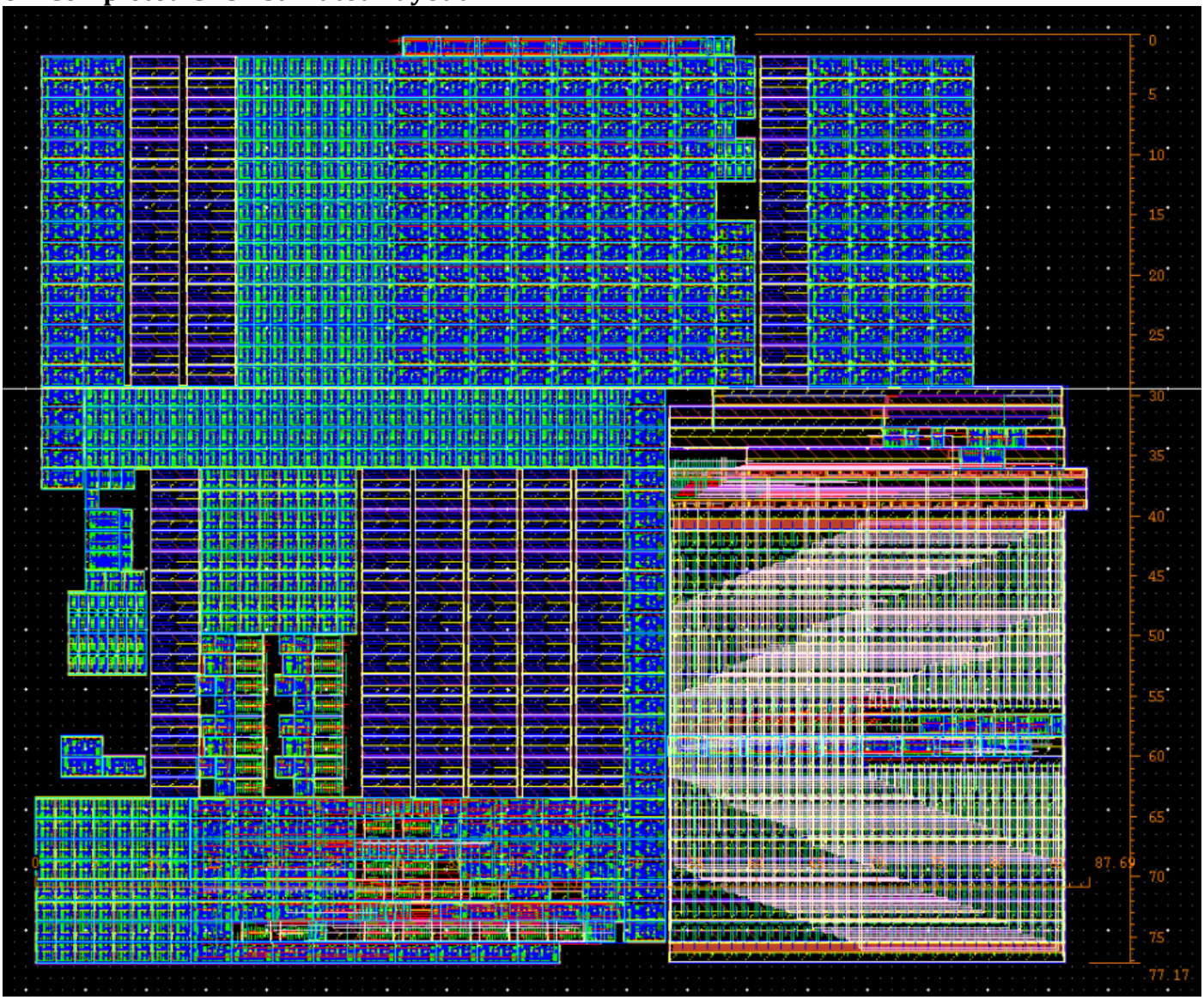
8.2.2 ALU estimated layout

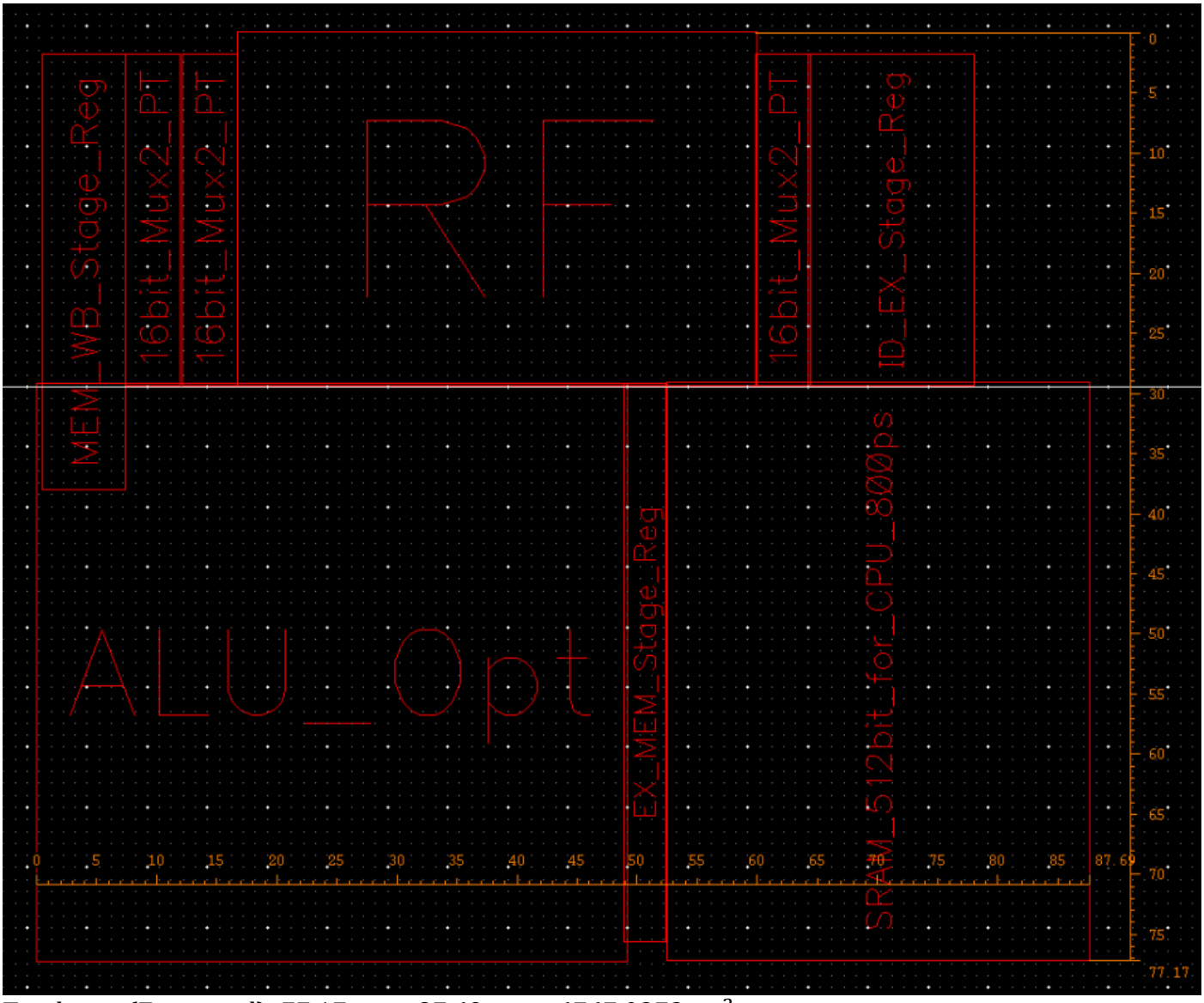

0
5
10
15
20
25
30
35
40
45
49.04
0
5
10
15
20
48.13

8.3 Register File Estimated Layout



8.4 Completed CPU Estimated Layout





Total area (Estimated): $77.17\mu m \times 87.69\mu m = 6767.0373\mu m^2$