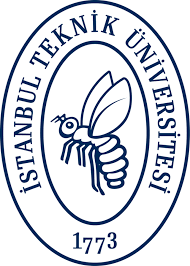
****

**VLSI Circuit Design II– EHB 425E**

**HOMEWORK V**

**Yiğit Bektaş GÜRSOY**

**040180063**

**Rana TİLKİ**

**040180741**

**Class Lecturer: Sıddıka Berna Örs Yalçın**

**Class Assistant:  
Yasin Fırat Kula**

1. **Instructions and Descriptions**

|  |  |  |
| --- | --- | --- |
| **Mnemonic** | **Name** | **Description** |
| **LUI** | Load Upper Immediate | rd 🡨 imm << 12 |
| **AUIPC** | Add Upper Immediate To Pc | rd 🡨 program counter(pc) + imm << 12 |
| **JAL** | Jump And Link | rd 🡨 {imm[20], imm[10:1], imm[11], imm[19:12]} |
| **JALR** | Jump and Link Register | rs1 🡨 imm[11:0] then rs1[0] 🡨 0  rd 🡨 pc + 4  pc 🡨 pc + imm |
| **BEQ** | Branch If Equal | if rs1 = rs2, pc 🡨 pc + imm  else pc 🡨 pc + 4 |
| **BNE** | Branch If Not Equal | if rs1 != rs2, pc 🡨 pc + imm  else pc = pc + 4 |
| **BLT** | Branch If Less Than | if rs1 < rs2, pc 🡨 pc + imm  else pc = pc + 4 |
| **BGE** | Branch If Greater or Equal | if rs1 >= rs2, pc 🡨 pc + imm  else pc = pc + 4 |
| **BLTU** | Branch If Less Than Unsigned | if rs1 < rs2, pc 🡨 pc + imm  else pc = pc + 4 |
| **BGEU** | Branch If Greater Than Unsigned | if rs1 >= rs2, pc 🡨 pc + imm  else pc = pc + 4 |
| **LB** | Load Byte | address 🡨 rs1 + imm[7:0]  byte\_value 🡨 Memory[address]  rd 🡨 sxt(byte\_value) |
| **LH** | Load Halfword | address 🡨 rs1 + imm[15:0]  byte\_value 🡨 Memory[address]  rd 🡨 sxt(byte\_value) |
| **LW** | Load Word | address 🡨 rs1 + imm[31:0]  byte\_value 🡨 Memory[address]  rd 🡨 sxt(byte\_value) |
| **LBU** | Load Byte Unsigned | address 🡨 rs1 + imm[7:0]  byte\_value 🡨 Memory[address]  rd 🡨 zxt(byte\_value) (zxt - unsigned extended |
| **LHU** | Load Halfword Unsigned | address 🡨 rs1 + imm[15:0]  byte\_value 🡨 Memory[address]  rd 🡨 zxt(byte\_value) |
| **SB** | Store Byte | address 🡨 rs1 + imm  Memory[address] 🡨rs2[7:0] |
| **SH** | Store Halfword | address 🡨 rs1 + imm  Memory[address] 🡨 rs2[15:0] |
| **SW** | Store Word | address 🡨rs1 + imm  Memory[address] 🡨rs2[31:0] |
| **ADDI** | Add Immediate | rd 🡨rs1 + sxt(imm) |
| **SLTI** | Set Less Than Immediate | if rs1<sxt(imm) then rd 🡨1  else rs 🡨 0 |
| **SLTIU** | Set Less Than Immediate Unsigned | if rs1 🡨 zxt(imm), rd 🡨1  else rd 🡨 0 |
| **XORI** | XOR Immediate | rd 🡨 rs1 ^ imm |
| **ORI** | OR Immediate | rd 🡨 rs1 | imm |
| **ANDI** | AND Immediate | rd 🡨 rs1 & imm |
| **SLLI** | Shift Left Logical Immediate | rd 🡨 rs1 << shamt |
| **SRLI** | Shift Right Logical Immediate | rd 🡨 rs1 >> shamt |
| **SRAI** | Shift Right Arithmetic Immediate | rd 🡨 rs1 >>> shamt |
| **ADD** | Addition | rd 🡨 rs1 + rs2 |
| **SUB** | Substraction | rd 🡨 rs1 – rs2 |
| **SLL** | Shift Left Logical | rd 🡨 rs1 << rs2[4:0] |
| **SLT** | Set Less Than | if rs1 < rs2 then rd 🡨 1  else rd 🡨 0 |
| **SLTU** | Set Less Than Unsigned | if rs1[31:0] < rs2[31:0] then rd 🡨 1  else rd 🡨 0 |
| **XOR** | Exclusive OR | rd 🡨 rs1 ^ rs2 |
| **SRL** | Shift Right Logical | rd 🡨 rs1 >> rs2[4:0] |
| **SRA** | Shift Right Arithmetic | rd 🡨 rs1 >>> rs2[4:0] |
| **OR** | OR | rd 🡨 rs1 | rs2 |
| **AND** | AND | rd 🡨 rs1 & rs2 |

1. **Machine Codes**

1-)ADDI: 000000000000\_00000\_000\_00001\_0010011

I-TYPE imm[11:0] rs1 funct3 rd opcode

imm[11:0] = 000000000000, rs1 = 00000, funct3 = 000, rd = 00001, opcode = 0010011

Hexadecimal : imm[11:0] = 0, rs1 = 0, rd=1

2-)BEQ: 0000000\_00010\_00011\_000\_00010\_0001100

B-TYPE imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode

imm[12:1] = 000000000001

imm[12|10:5] = 0000000, rs2 = 00010, rs1 = 00011, funct3 = 000, imm[4:1|11] = 00010, opcode = 1100011

Hexadecimal : imm[12:1] = 1, rs2 = 2, rs1=3

3-)LW : 000000000010\_00100\_010\_00101\_0000011

I-TYPE imm[11:0] rs1 funct3 rd opcode

imm[11:0] = 000000000010, rs1 = 00100, funct3 = 010, rd = 00101, opcode = 0000011

Hexadecimal: imm[11:0] = 2, rs1=4, rd=5

4-)SW : 0000000\_00110\_00111\_010\_00011\_0100011

S-TYPE imm[11:5] rs2 rs1 funct3 imm[4:0] opcode

imm[11:0] = 000000000011

imm[11:5] = 0000000, rs2 = 00110, rs1 = 00111, funct3 = 010, imm[4:0] = 00011, opcode = 0100011

Hexadecimal: imm[11:0] = 3, rs2=6, rs1=7

5-)BNE: 0000000\_01000\_01001\_001\_01000\_1100011

B-TYPE imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode

imm[12:1] = 000000000100

imm[12|10:5] = 0000000, rs2 = 01000, rs1 = 01001, funct3 = 001, imm[4:1|11] = 01000, opcode = 1100011

Hexadecimal : imm[12:1] = 4, rs2 = 8, rs1=9

6-)BLT: 0000000\_01010\_01011\_100\_01010\_1100011

B-TYPE imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode

imm[12:1] = 000000000101

imm[12|10:5] = 0000000, rs2 = 01010, rs1 = 01011, funct3 = 100, imm[4:1|11] = 01010, opcode = 1100011

Hexadecimal: imm[12:1] = 5, rs2 = 10, rs1=11

7-)BGE: 0000000\_01100\_01101\_101\_01100\_1100011

B-TYPE imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode

imm[12:1] = 000000000110

imm[12|10:5] = 0000000, rs2 = 01100, rs1 = 01101, funct3 = 101, imm[4:1|11] = 01100, opcode = 1100011

Hexadecimal: imm[12:1] = 6, rs2 = 12, rs1=13

8-)BLTU: 0000000\_01110\_01111\_110\_01110\_1100011

B-TYPE imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode

imm[12:1] = 000000000111

imm[12|10:5] = 0000000, rs2 = 01110, rs1 = 01111, funct3 = 110, imm[4:1|11] = 01110, opcode = 1100011

Hexadecimal: imm[12:1] = 7, rs2 = 14, rs1=15

9-)BGEU: 0000000\_10000\_10001\_111\_10000\_1100011

B-TYPE imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode

imm[12:1] = 000000001000

imm[12|10:5] = 0000000, rs2 = 10000, rs1 = 10001, funct3 = 111, imm[4:1|11] = 10000, opcode = 1100011

Hexadecimal: imm[12:1] = 8, rs2 = 16, rs1=17

10-)XOR: 0000000\_10010\_10011\_100\_10100\_0110011

R-TYPE func7 rs2 rs1 func3 rd opcode

func7 = 0000000, rs2 = 10010, rs1 = 10011, func3 = 100, rd = 10100, opcode = 0110011

Hexadecimal: rs2 = 18, rs1=19, rd=20

11-)OR: 0000000\_10101\_10110\_110\_10111\_0110011

R-TYPE func7 rs2 rs1 func3 rd opcode

func7 = 0000000, rs2 = 10101, rs1 = 10110, func3 = 110, rd = 10111, opcode = 0110011

Hexadecimal: rs2 = 21, rs1=22, rd=23

12-)AND: 0000000\_11000\_11001\_111\_11010\_0110011

R-TYPE func7 rs2 rs1 func3 rd opcode

func7 = 0000000, rs2 = 11000, rs1 = 11001, func3 = 111, rd = 11010, opcode = 0110011

Hexadecimal: rs2 = 24, rs1=25, rd=26

13-)SUB: 0100000\_11011\_11100\_000\_11101\_0110011

R-TYPE func7 rs2 rs1 func3 rd opcode

func7 = 0100000, rs2 = 11011, rs1 = 11100, func3 = 000, rd = 11101, opcode = 0110011

Hexadecimal: rs2 = 27, rs1=28, rd=29

14-)JAL: 00000001001000000000\_11110\_1101111

J-TYPE imm[20:1], rd, opcode

imm[20:1]= 00000000000000001001, rd = 11110, opcode = 1101111

Hexadecimal: imm[20:1]=9, rd=30

15-)AUIPC: 00000000000000001010\_11111\_0010111

U-TYPE imm[20:1], rd, opcode

imm[31:12]= 00000000000000001010, rd = 11111, opcode = 0010111

Hexadecimal: imm[31:12]=10, rd=31

16-)SRA: 0100000\_00000\_00001\_101\_00010\_0110011

R-TYPE func7 rs2 rs1 func3 rd opcode

func7 = 0100000, rs2 = 00000, rs1 = 00001, func3 = 101, rd = 00010, opcode = 0110011

Hexadecimal: rs2 = 0, rs1=1, rd=2

In RISC-V, the NOP (no operation) instruction is performed using the ADDI (add immediate) instruction with both source operands set to the zero register (x0). Since adding zero to a register doesn't change its value, the ADDI instruction effectively performs no operation.

NOP: 00000000000000000000000000010011

imm: 0000000000000000, rd: 00000, funct3: 000, rs1: 00000, opcode:0010011

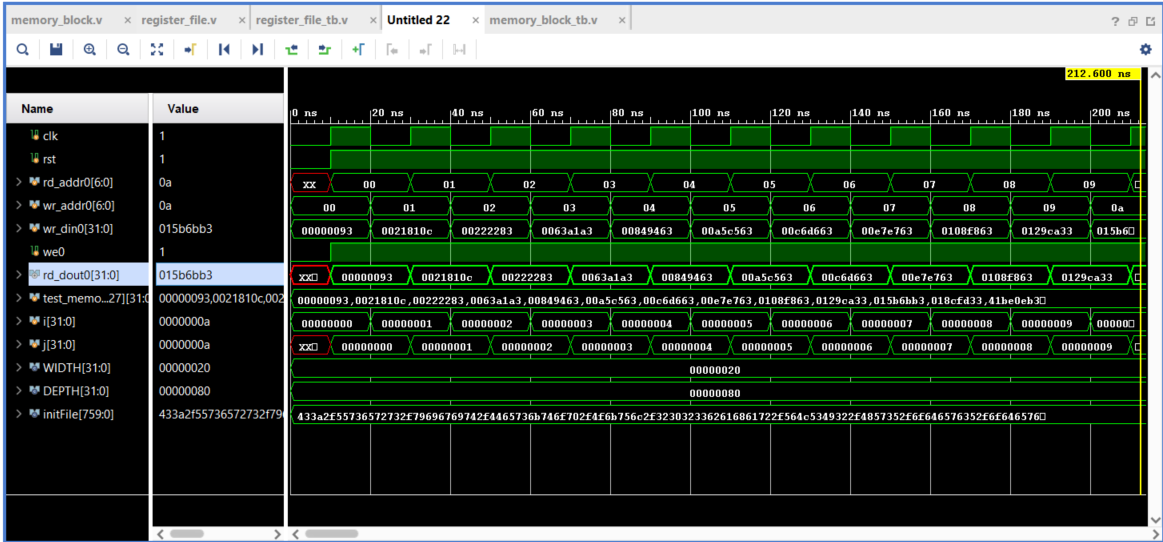
In this encoding, the opcode field is `0010011`, which corresponds to the ADDI instruction. The rs1 and rs2 fields are both set to the zero register (x0), and the immediate field is set to zero. Since adding zero to a register doesn't change its value, this instruction effectively performs no operation.

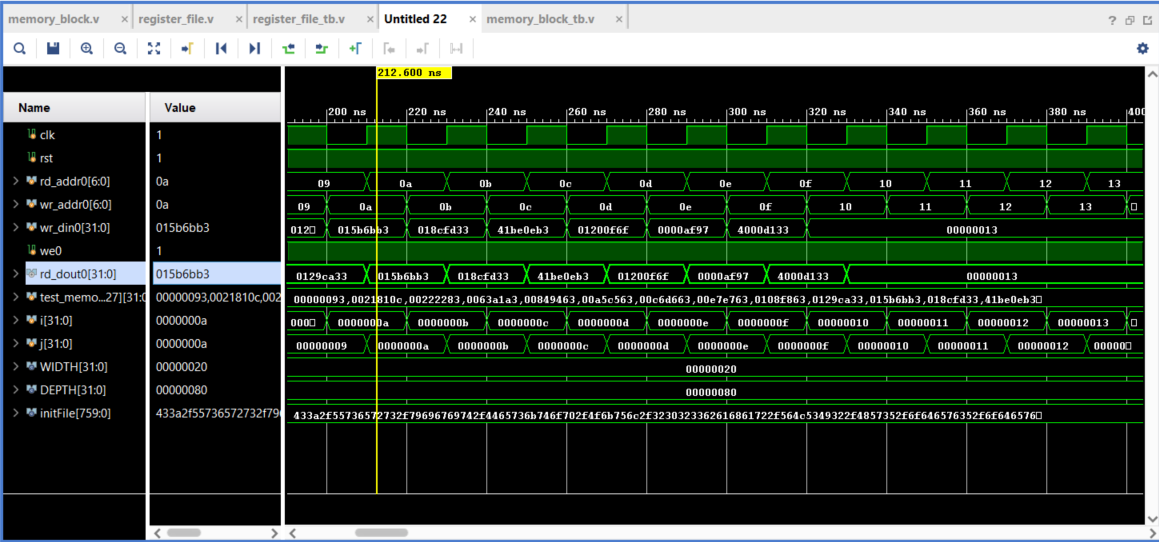
1. **Behavioral Simulation Results**

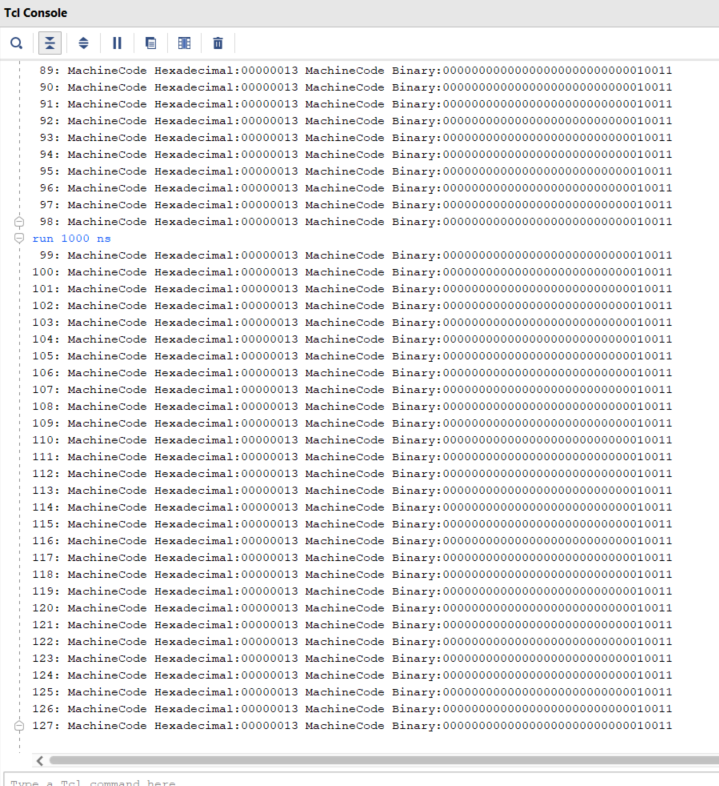
The codes are in the zip file. Only simulation results and comments are available here.

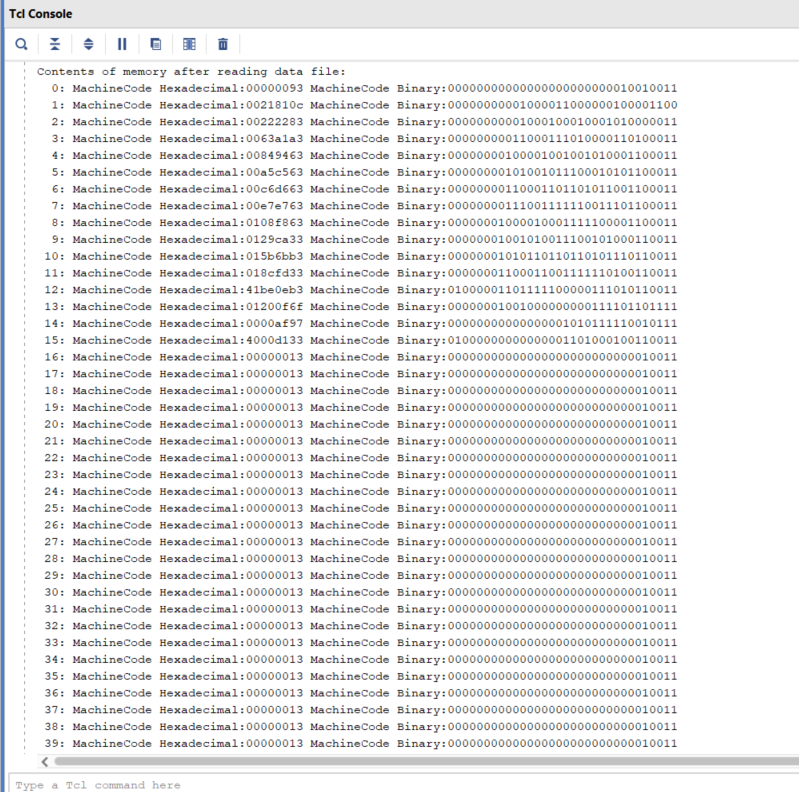
* **Memory Block**

The simulation results and TCL console outputs are shown below. 32bit data consisting of 128 lines in the "machine\_code.txt" file is read through the memory\_block.v file. With our "we0" signal being 1, the data starts to be written to the memory block. When wr\_din0 and rd\_dout signals are followed, it appears to be read successfully. After the first 16 lines are read, the remaining 112 lines are filled with "no action" machine code. The results are verified when comparing the memory written with the Tcl console outputs.

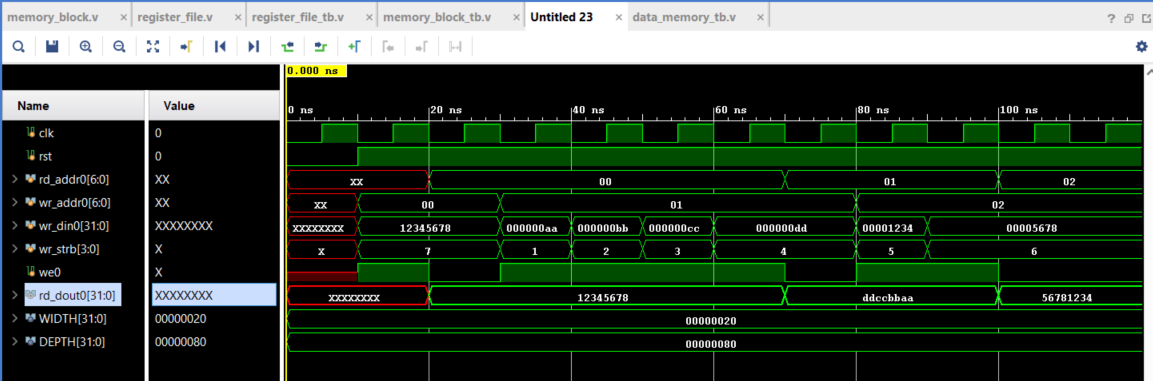




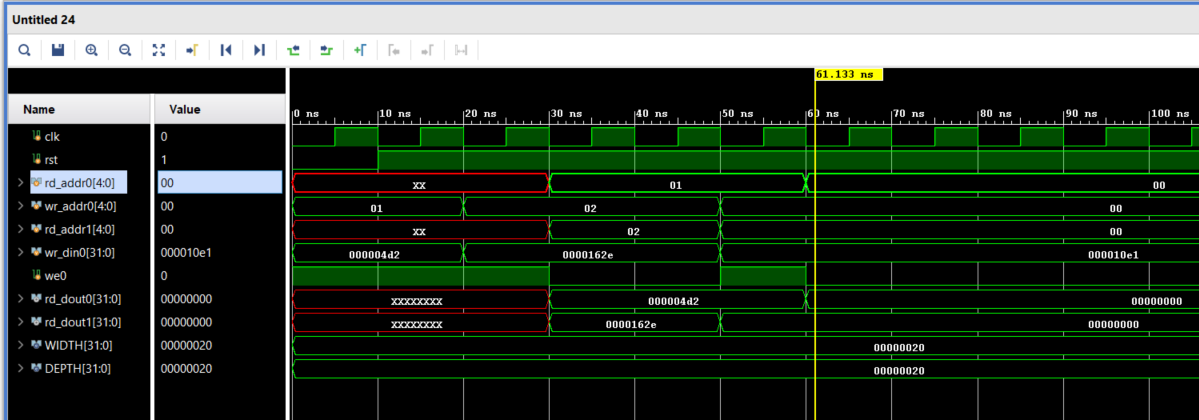




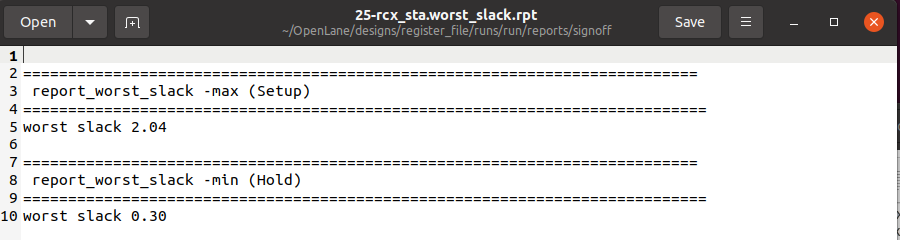
* **Data Memory**

In this section, it is aimed to show 8bit, 16bit and 32bit values. An 8-bit (1 byte) value is displayed in 4 ranges, a 16-bit (half word) value in 2 ranges, and a 32-bit (word) value in 1 range. This part is coded in the upper module with the case structure. Then a testbench was designed for this module. Word, 1byte and halfword are specified in the simulation, respectively. The wr\_strb signal represents the case states, there are 7 states in total. After the wr\_din0 signal reaches 32bit and the we01 signal becomes 1, the output appears in the rd\_dout0 signal.

* **Register File**

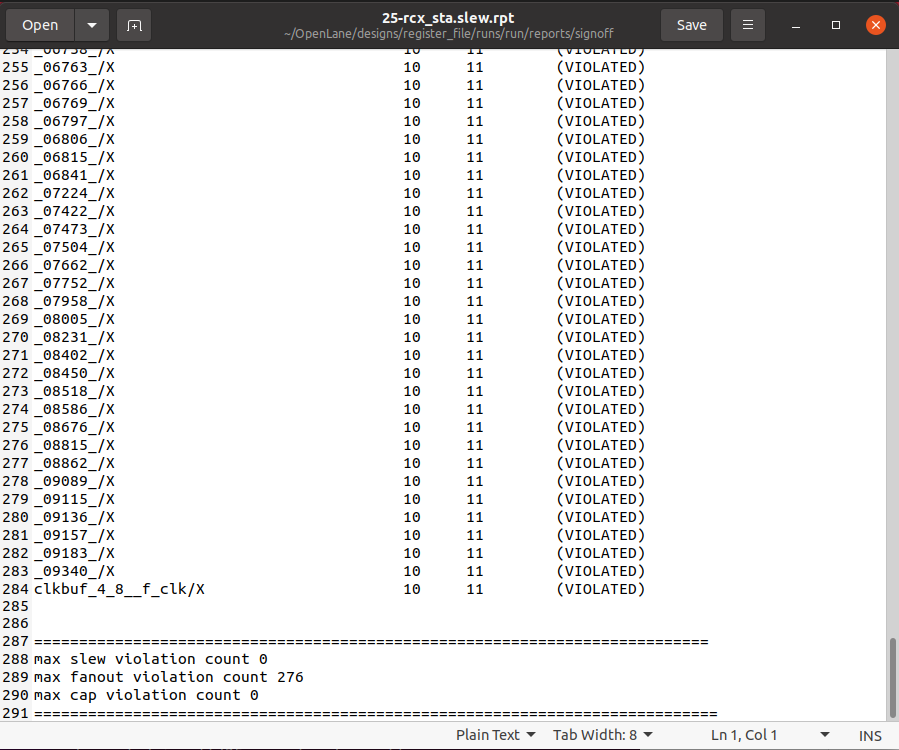
Below is the simulation result of the register file module created using the memory\_block.v top module. In this module, 2 inputs named rd\_addr0 and rd\_addr1 and two outputs named rd\_dout0 and rd\_dout1 have been added. The values given here appear in the relevant outputs, respectively. It is said that when rd\_addr0 and rd\_addr1 are given 0 as stated in the assignment, 0 should be output. It is stated that when both inputs return 0, 0 is read at the output. Incoming values other than 0 are shown to be output directly. The circuit is working correctly.

* Setup and Hold Violations

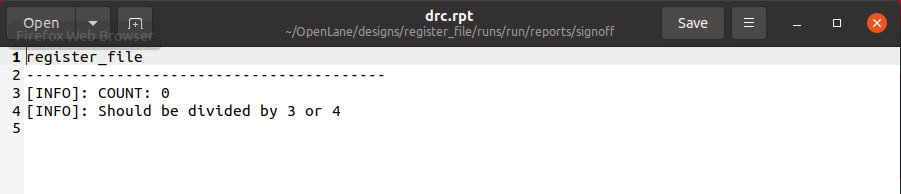


Maximum frequency 1/2.04\*10^-9 = 490.196 Mhz

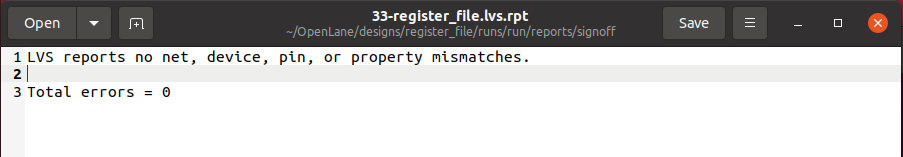
* DRV



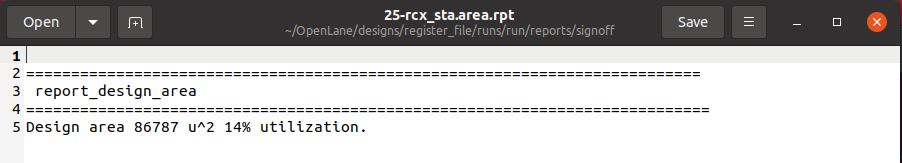
* DRC

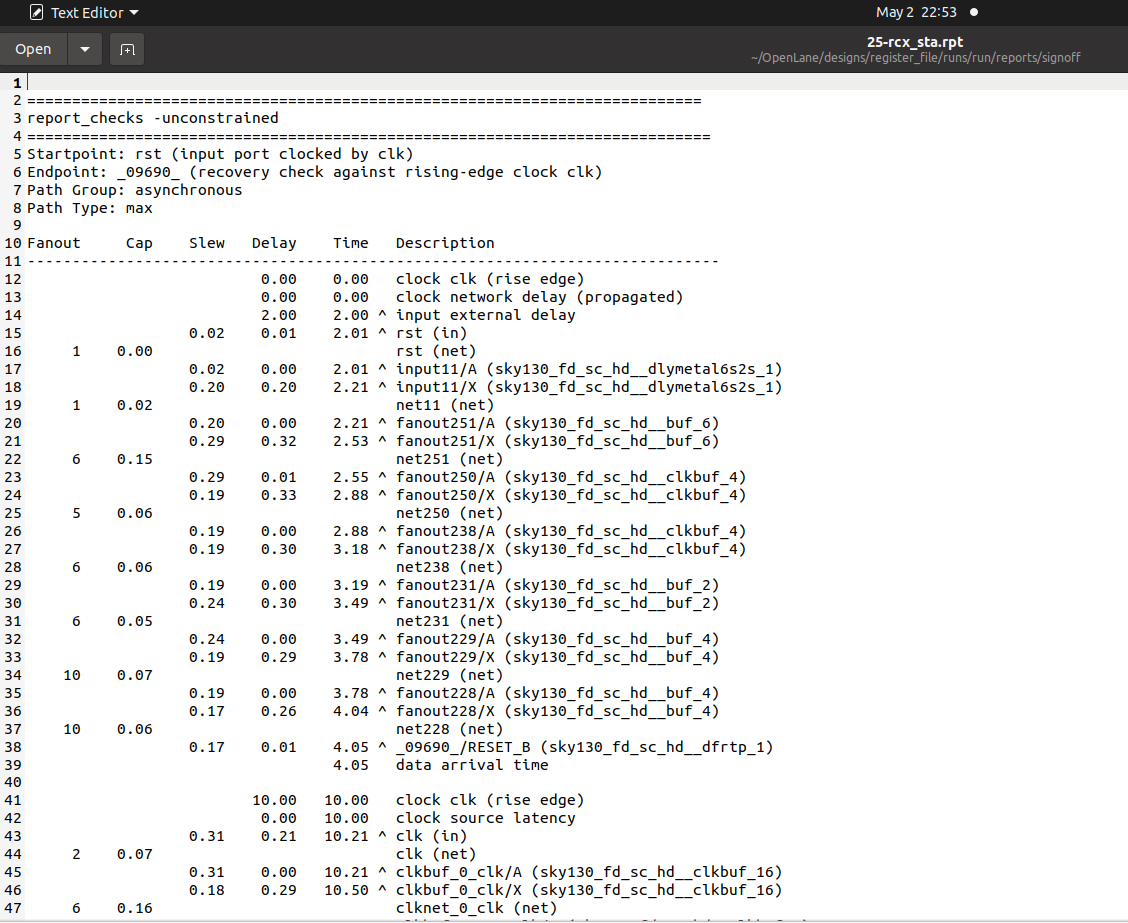


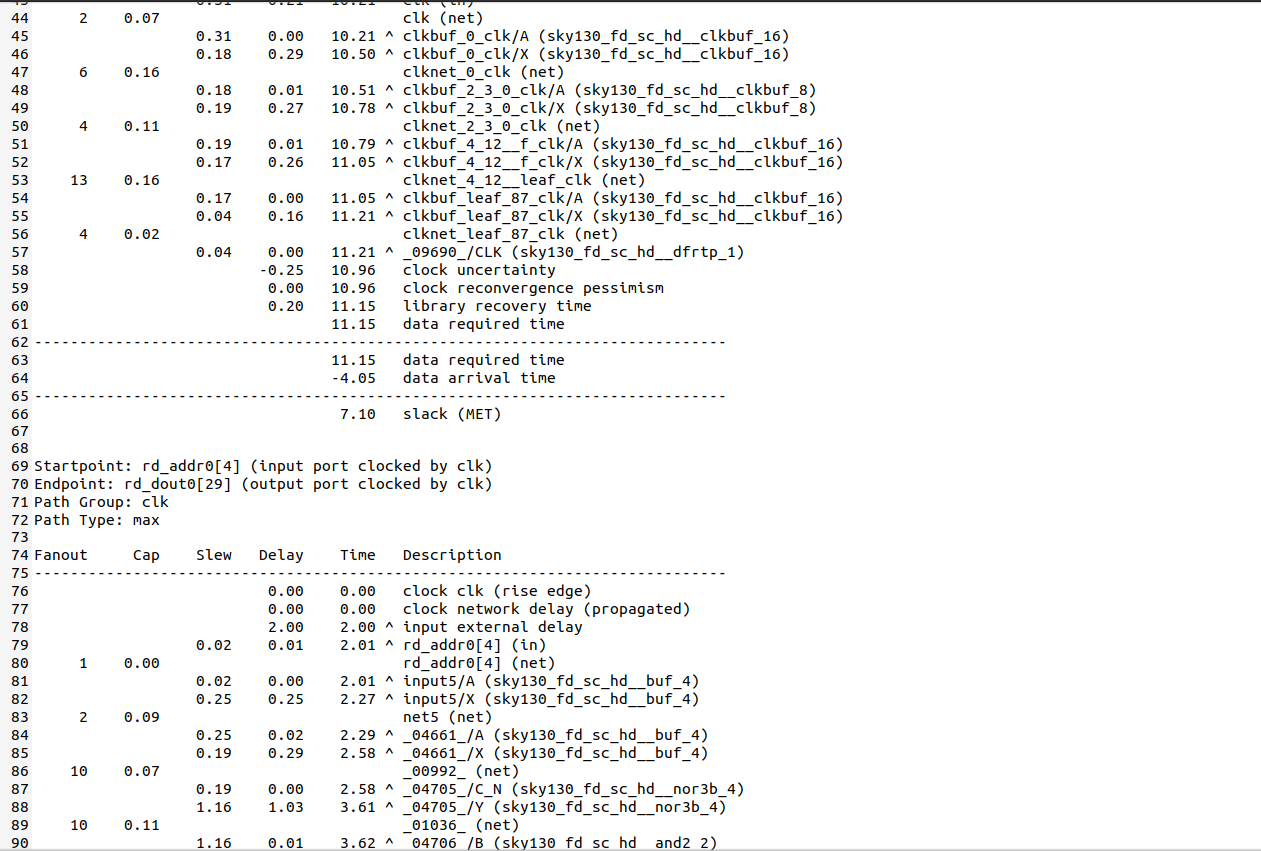
* LVS

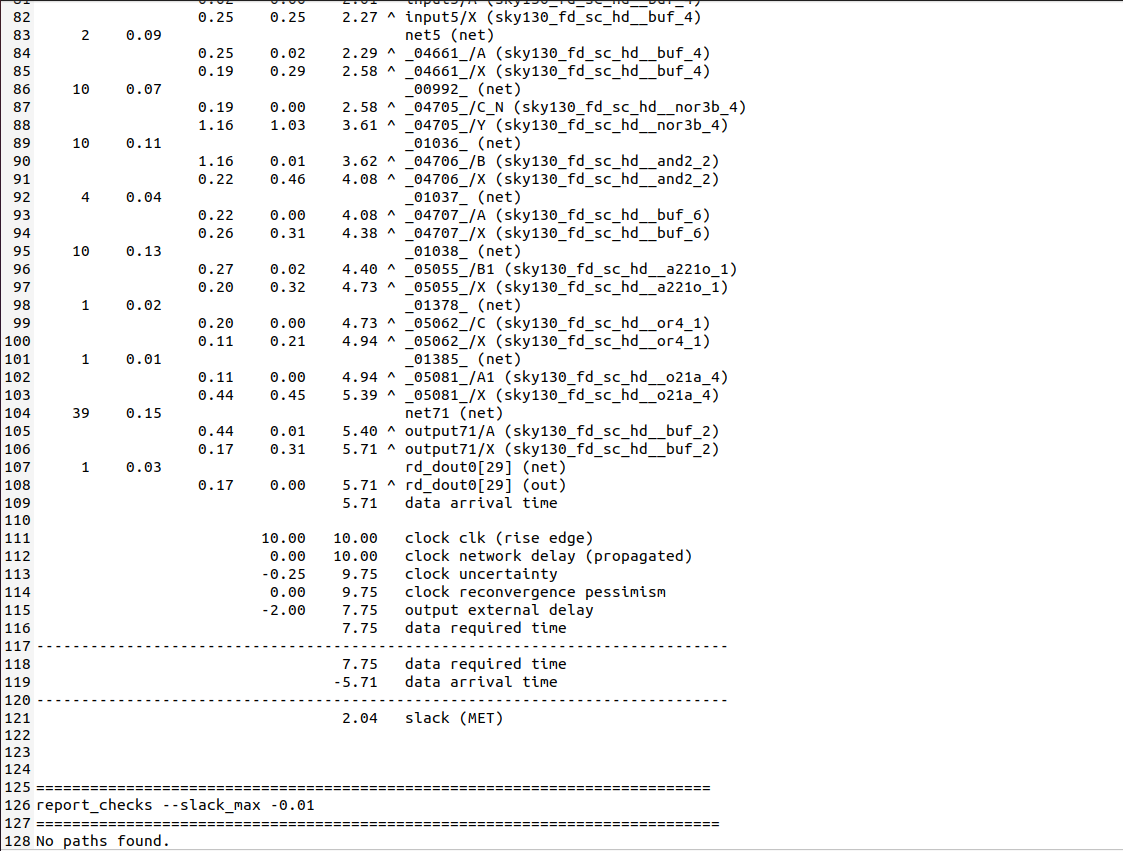


* Antennas outputs are in .zip file. It is not included here because it is too long.
* Final Area



* Path





**For asynchronous group**

Startpoint: rst (input port clocked by clk)

Endponit: \_09690\_ (recovery check against rising-edge clock clk)

Worst path is 7.1 ns (slack (MET)).

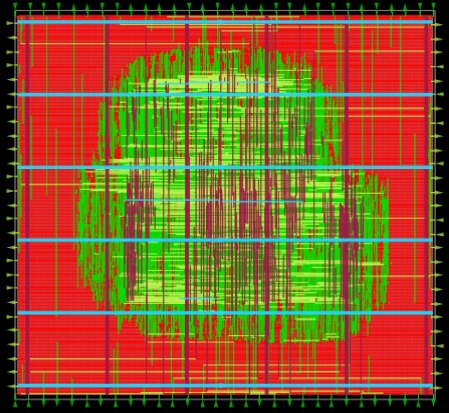
**For clk group**

Startpoint: rd\_addr0[4] (input port clocked by clk)

Endpoint: rd\_dout[29] (output port clocked by clk)

Worst path is 2.04 ns ( slack (MET) ).

The number whose maximum gives the value of the clock frequency is the number in the critical path. This is how the critical path is found. Based on this, the critical path can be found. The above results are written according to these inferences.

* Layout