VLSI-2

Project Raport-4

Deniz Zakir EROĞLU- 040200249

Şeyma ÇALIŞKAN- intern at GSTL

Deniz Zakir Eroğlu – 040200249 / Şeyma Çalışkan

Introduction

The design and verification of custom memory components play a crucial role in modern processor architecture, especially in RISC-based microprocessor systems. This report documents the development of configurable memory structures and register file components, All modules are designed using Verilog HDL and tested through simulatin.

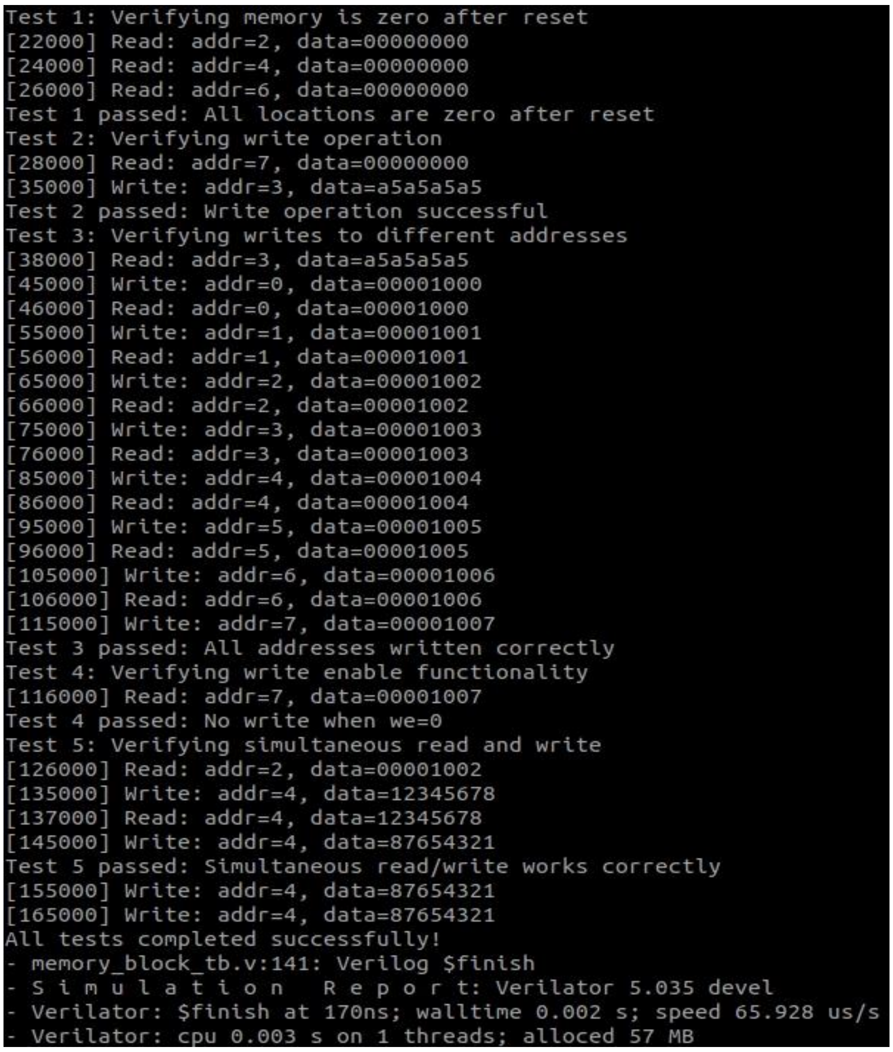
The project starts with the implementation of a 1-dimensional flip-flop based single-read/single-write (1R1W) memory block. This memory module supports parameterized width and depth, making it scalable for different applications. It features standard inputs and outputs including asynchronous reset, synchronous write operation, and asynchronous read access. This basic design forms the foundation for all subsequent memory modules in the project.

Basic Single Port RAM

The HDL code for designing a single-ported RAM, is provided below:

```verilog
module memory_block #(
    parameter DATA_WIDTH = 32,
    parameter DEPTH = 32,
    parameter INIT_FILE = ""
)(
    input clk,
    input rstn,
    input [$clog2(DEPTH)-1:0] rd_addr0,
    input [$clog2(DEPTH)-1:0] wr_addr0,
    output reg [DATA_WIDTH-1:0] rd_dout0,
    input [DATA_WIDTH-1:0] wr_din0,
    input we0);

    reg [DATA_WIDTH-1:0] mem [0:DEPTH-1];

    assign rd_dout0 = mem[rd_addr0];

    always @(posedge clk or negedge rstn) begin
        integer i;
        if (~rstn) begin
            if (INIT_FILE != "") begin
                $readmemh(INIT_FILE, mem);
            end
            else begin
                for (i = 0; i < DEPTH; i = i + 1) begin
                    mem[i] <= {DATA_WIDTH{1'b0}};
                end
            end
        end else begin
            if (we0) begin
                mem[wr_addr0] <= wr_din0;
            end
        end
    end
endmodule
```

Verification of Basic RAM:



```
Test 1: Verifying memory is zero after reset
[22000] Read: addr=2, data=00000000
[24000] Read: addr=4, data=00000000
[26000] Read: addr=6, data=00000000
Test 1 passed: All locations are zero after reset
Test 2: Verifying write operation
[28000] Read: addr=7, data=00000000
[35000] Write: addr=3, data=a5a5a5a5
Test 2 passed: Write operation successful
Test 3: Verifying writes to different addresses
[38000] Read: addr=3, data=a5a5a5a5
[45000] Write: addr=0, data=00001000
[46000] Read: addr=0, data=00001000
[55000] Write: addr=1, data=00001001
[56000] Read: addr=1, data=00001001
[65000] Write: addr=2, data=00001002
[66000] Read: addr=2, data=00001002
[75000] Write: addr=3, data=00001003
[76000] Read: addr=3, data=00001003
[85000] Write: addr=4, data=00001004
[86000] Read: addr=4, data=00001004
[95000] Write: addr=5, data=00001005
[96000] Read: addr=5, data=00001005
[105000] Write: addr=6, data=00001006
[106000] Read: addr=6, data=00001006
[115000] Write: addr=7, data=00001007
Test 3 passed: All addresses written correctly
Test 4: Verifying write enable functionality
[116000] Read: addr=7, data=00001007
Test 4 passed: No write when we=0
Test 5: Verifying simultaneous read and write
[126000] Read: addr=2, data=00001002
[135000] Write: addr=4, data=12345678
[137000] Read: addr=4, data=12345678
[145000] Write: addr=4, data=87654321
Test 5 passed: Simultaneous read/write works correctly
[155000] Write: addr=4, data=87654321
[165000] Write: addr=4, data=87654321
All tests completed successfully!
- memory_block_tb.v:141: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.035 devel
- Verilator: $finish at 170ns; walltime 0.002 s; speed 65.928 us/s
- Verilator: cpu 0.003 s on 1 threads; alloced 57 MB
```

Figure 1 Basic Memory Simulation Results

Deniz Zakir Eroğlu – 040200249 / Şeyma Çalışkan

RV32I Instruction Set

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

Table 1: Risc-V ISA

In the second section, focus shifts to the RISC-V RV32I instruction set. A representative set of 16 instructions from various instruction formats (R-type, I-type, S-type, etc.) is selected. Each instruction is described with a functional explanation and its binary-encoded machine code is derived. These instructions are written into a memory initialization file in hexadecimal format. To complete the 128-line memory file, NOP (No Operation) instructions—encoded as addi x0, x0, 0 with a machine code of 00000013—are used as padding.

Risc-V Program Machine Codes:

```
10000537 lui a0, 0x10000            00010000000000000000\_01010\_0110111
                                     _________IMM________\__RD__\_OPCODE
```

```
02a00593 addi a1, zero, 42          000000101010\_00000\_000\_01011\_0010011
                                    _____IMM____\__RS1__\_F3_\__RD__\_OPCODE
```

```
00452603 lw a2, 4(a0)               000000000100\_01010\_010\_01100\_0000011
                                    _____IMM____\__RS1__\_F3_\__RD__\_OPCODE
```

```
00c586b3 add a3, a1, a2             0000000\_01100\_01011\_000\_01101\_0110011
                                    ___F7__\__RS2__\_RS1__\_F3_\__RD__\_OPCODE
```

```
40b68733 sub a4, a3, a1             0100000\_01011\_01101\_000\_01110\_0110011
                                    ___F7___\__RS2__\__RS1_\_F3_\__RD__\_OPCODE
```

Deniz Zakir Eroğlu – 040200249 / Şeyma Çalışkan

```
00e6f7b3 and a5, a3, a4          0000000\_01110\_01101\_111\_01111\_0110011
                                  ___F7___\__RS2__\__RS1_\_F3_\__RD_\_OPCODE
```

```
00d52423 sw a3, 8(a0)            0000000\_01101\_01010\_010\_01000\_0100011
                                  __IMM__\_RS2__\__RS1_\_F3_\__IMM_\_OPCODE
```

```
00c58863 beq a1, a2, skip        0000000\_01100\_01011\_000\_10000\_1100011
                                 __IMM__\__RS2_\__RS1_\_F3_\__IMM_\_OPCODE
```

```
00c5c663 blt a1, a2, skip        0000000\_01100\_01011\_100\_01100\_1100011
                                 ___IMM__\__RS2_\__RS1_\_F3_\__IMM_\_OPCODE
```

```
00259813 slli a6, a1, 2          000000000010\_01011\_001\_10000\_0010011
                                 _____IMM_____\__RS1_\_F3_\__RD__\_OPCODE
```

```
010000ef jal ra, func            00000001000000000000\_00001\_1101111
                                 ________IMM________\__RD___\_OPCODE
```

```
0ff5c893 xori a7, a1, 0xFF       000011111111\_01011\_100\_10001\_0010011
                                 _____IMM____\__RS1_\_F3_\__RD__\_OPCODE
```

```
00000297 auipc t0, 0x0           00000000000000000000\_00101\_0010111
                                 ________IMM________\__RD__\_OPCODE
```

```
00008067 jalr zero, 0(ra)        000000000000\_00001\_000\_00000\_1100111
                                 _____IMM____\__RS1__\_F3_\__RD__\_OPCODE
```

```
00c5e333 or t1, a1, a2           0000000\_01100\_01011\_110\_00110\_0110011
                                ____F7___\__RS2_\__RS1_\_F3_\__RD__\_OPCODE
```

```
00c5c3b3 xor t2, a1, a2          0000000\_01100\_01011\_100\_00111\_0110011
                                 ___F7___\__RS2__\__RS1_\_F3_\__RD__\_OPCODE
```

NOP (No Operation) Instruction in RISC-V

In certain scenarios, a processor may need to remain idle for a clock cycle without performing any meaningful operation. Such situations can occur due to synchronization requirements, pipeline alignment, timing delays, or even user-defined wait cycles. However, modern processors cannot simply "pause" in the literal sense. Instead, they execute a special type of instruction known as a **NOP (No Operation)**, which effectively consumes a clock cycle without modifying the processor's state.

NOP: 00000000000000000000000000010011

IMM: 0000000000000000, RD: 00000, funct3: 000, RS1: 00000, opcode:0010011

In the RISC-V architecture, there is no dedicated opcode for NOP. Instead, the NOP behavior is achieved through the following instruction:

```
addi x0, x0, 0
```

This is an **ADDI (add immediate)** instruction where both the destination (RD) and the source register (RS1) are set to x0, which is a hardwired zero register in RISC-V. The immediate value is also set to zero. Since x0 is immutable and always holds the value zero, the result of this operation has no effect:

Instruction Memory

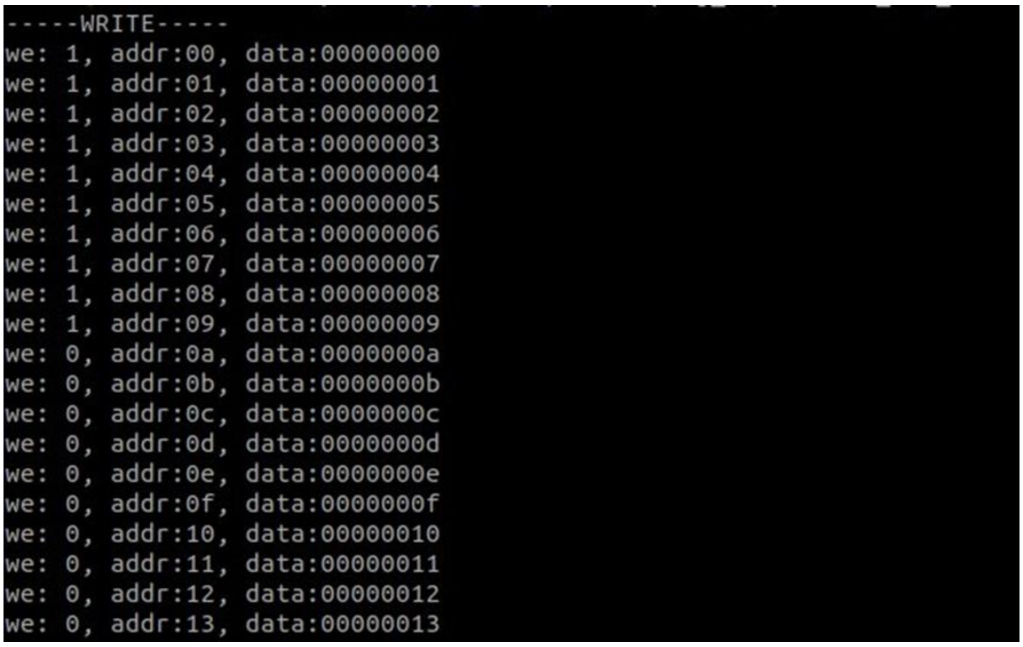
Following this, an instruction memory testbench is built by instantiating the 1R1W memory module. The instruction set file is loaded using Verilog's $readmemh function. The design is then tested using Verilator, where memory write and read operations are simulated and monitored to ensure correctness. We performed functional verification using basic testbenches. Although we attempted to implement the UVM environment, we encountered several issues and were unable to complete it successfully.

Deniz Zakir Eroğlu – 040200249 / Şeyma Çalışkan

The HDL code for designing a instruction memory, is provided below:

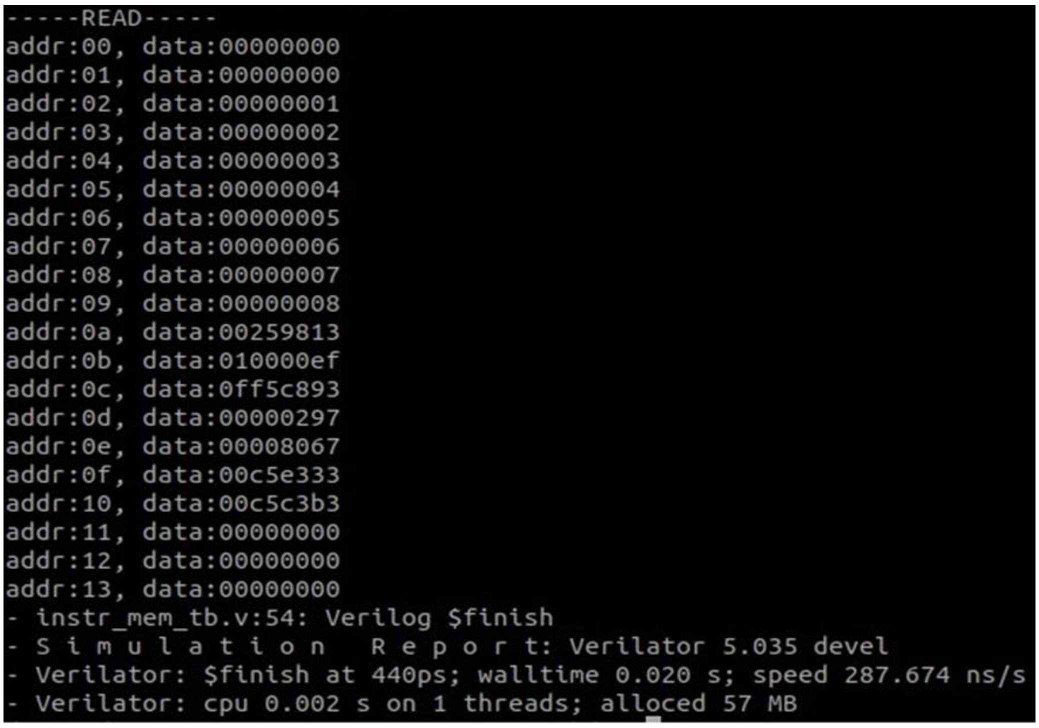
```
module instr_mem_tb();
    reg clk, rstn;
    reg [$clog2(DEPTH)-1:0] rd_addr0, wr_addr0;
    reg [DATA_WIDTH-1:0] wr_din0;
    wire [DATA_WIDTH-1:0] rd_dout0;
    reg we0;
    integer i = 0;
    parameter DEPTH=128;
    parameter DATA_WIDTH=32;
    memory_block
#(.DEPTH(DEPTH),.DATA_WIDTH(DATA_WIDTH),.INIT_FILE("instr_mem_tb.data"))uut(
        .clk(clk),
        .rstn(rstn),
        .rd_addr0(rd_addr0),
        .wr_addr0(wr_addr0),
        .wr_din0(wr_din0),
        .rd_dout0(rd_dout0),
        .we0(we0));

    initial begin
        clk = 0;
        rstn = 0;
        rd_addr0 = 0;
        #10;
        rstn = 1;

        $display("-----WRITE-----");
        for(i = 0; i < 10; i++) begin
            @(posedge clk);
            we0=1;
            {25'd0, wr_addr0} = i;
            wr_din0=i;
            $display("we: %h, addr:%h, data:%h", we0, wr_addr0, wr_din0);
        end

        for(i = 10; i < 20; i++) begin
            @(posedge clk);
            we0=0;
            {25'd0, wr_addr0} = i;
            wr_din0=i;
            $display("we: %h, addr:%h, data:%h", we0,wr_addr0, wr_din0);
        end

        $display("-----READ-----");
        for(i = 0; i < 20; i++) begin
            @(posedge clk);
            i = i + 1;
            if(i < 20) begin
                rd_addr0 = rd_addr0 + 1;
                $display("addr:%h, data:%h", rd_addr0, rd_dout0);
            end
        end
        #10;
        $finish;
    end
    always #5 clk=~clk;

endmodule
```

Results can be seen in below (Figure 2-3):



```
-----WRITE-----
we: 1, addr:00, data:00000000
we: 1, addr:01, data:00000001
we: 1, addr:02, data:00000002
we: 1, addr:03, data:00000003
we: 1, addr:04, data:00000004
we: 1, addr:05, data:00000005
we: 1, addr:06, data:00000006
we: 1, addr:07, data:00000007
we: 1, addr:08, data:00000008
we: 1, addr:09, data:00000009
we: 0, addr:0a, data:0000000a
we: 0, addr:0b, data:0000000b
we: 0, addr:0c, data:0000000c
we: 0, addr:0d, data:0000000d
we: 0, addr:0e, data:0000000e
we: 0, addr:0f, data:0000000f
we: 0, addr:10, data:00000010
we: 0, addr:11, data:00000011
we: 0, addr:12, data:00000012
we: 0, addr:13, data:00000013
```

Figure 2



```
-----READ-----
addr:00, data:00000000
addr:01, data:00000000
addr:02, data:00000001
addr:03, data:00000002
addr:04, data:00000003
addr:05, data:00000004
addr:06, data:00000005
addr:07, data:00000006
addr:08, data:00000007
addr:09, data:00000008
addr:0a, data:00259813
addr:0b, data:010000ef
addr:0c, data:0ff5c893
addr:0d, data:00000297
addr:0e, data:00008067
addr:0f, data:00c5e333
addr:10, data:00c5c3b3
addr:11, data:00000000
addr:12, data:00000000
addr:13, data:00000000
- instr_mem_tb.v:54: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.035 devel
- Verilator: $finish at 440ps; walltime 0.020 s; speed 287.674 ns/s
- Verilator: cpu 0.002 s on 1 threads; alloced 57 MB
```

Figure 3

Data Memory

Subsequently, a data memory modüle is created by extending the original 1R1W memory design with a write strobe (wr_strb) signal. This addition enables partial write operations to support RISC-V's SB, SH, and SW instructions. The correctness of byte-wise and half-word writes is verified through waveform inspection and console outputs using Verilator-based simulations.

The HDL code for designing a data memory, is provided below:

```
module data_mem#(
    parameter DEPTH=32,
    parameter DATA_WIDTH=32,
    parameter INIT_FILE=""
)(
    input clk, rstn,
    input [2:0] wr_strb,
    input [$clog2(DEPTH)-1:0] rd_addr0,wr_addr0,
    input [DATA_WIDTH-1:0] wr_din0,
    output [DATA_WIDTH-1:0] rd_dout0,
    input we0);

    reg [DATA_WIDTH-1:0] mem [DEPTH-1:0];
    assign rd_dout0 = mem[rd_addr0];
    integer i;

    always @(posedge clk or negedge rstn) begin
        if(~rstn) begin
            for (i = 0; i <= DEPTH; i = i + 1) begin
                mem[i]<=0;
            end
        end
        else begin
            if(we0) begin
                casez(wr_strb)
                    3'b?00 : mem[wr_addr0][7:0] <= wr_din0[7:0];
                    3'b?01 : mem[wr_addr0][15:0] <= wr_din0[15:0];
                    3'b?10 : mem[wr_addr0] <= wr_din0;
                    default : mem[wr_addr0] <= wr_din0;
                endcase
            end
        end
    end
endmodule
```

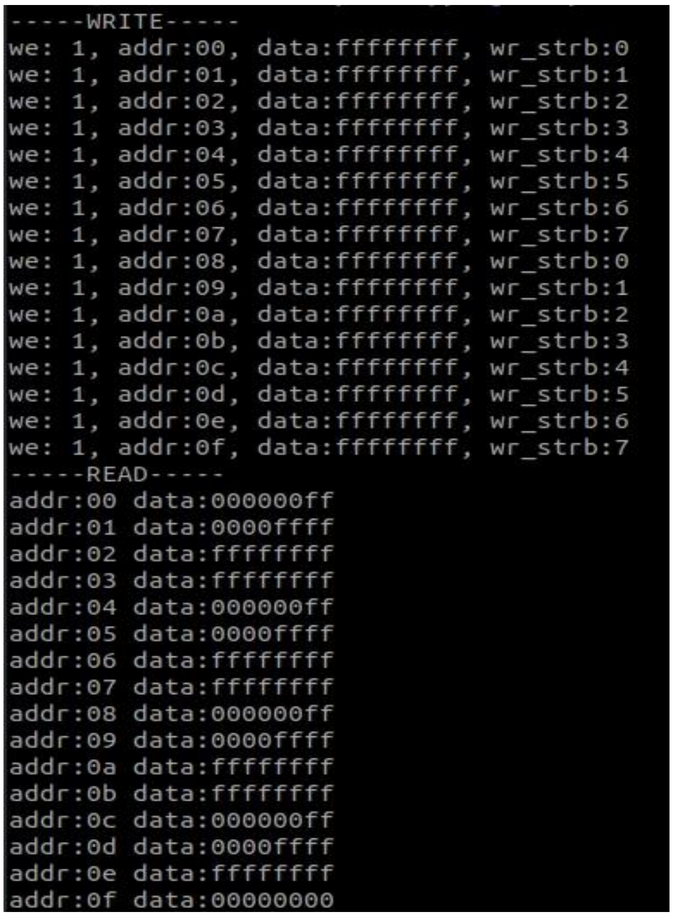
Deniz Zakir Eroğlu – 040200249 / Şeyma Çalışkan

The testbench code can be seen below:

```
module data_mem_tb();
    parameter DEPTH=128,DATA_WIDTH=32;
    reg clk,rstn;
    reg [2:0] wr_strb;
    reg [$clog2(DEPTH)-1:0] rd_addr0,wr_addr0;
    reg [DATA_WIDTH-1:0]wr_din0;
    wire [DATA_WIDTH-1:0] rd_dout0;
    reg we0;

    data_mem#(
        .DEPTH(DEPTH),
        .DATA_WIDTH(DATA_WIDTH),
        .INIT_FILE(""))
        uut(
        .clk(clk),
        .rstn(rstn),
        .wr_strb(wr_strb),
        .rd_addr0(rd_addr0),
        .wr_addr0(wr_addr0),
        .wr_din0(wr_din0),
        .rd_dout0(rd_dout0),
        .we0(we0));

    always #5 clk=~clk;
    integer i=1;

    initial begin
        wr_strb=0;
        wr_addr0 = 0;
        clk=0;
        rstn = 1;
        rd_addr0 = 0;
        rstn = 0;
        #10;
        rstn = 1;
        #10;
        wr_din0='hFFFFFFFF;
        we0 = 1;

        $display("-----WRITE-----");
        for (i = 0; i < 16; i++) begin
            @(negedge clk);
            {25'd0, wr_addr0} = i;
            wr_strb = i[2:0];
            rd_addr0 = wr_addr0;
            @(posedge clk);
            $display("we: %h, addr:%h, data:%h, wr_strb:%h", we0, wr_addr0,
wr_din0, wr_strb);
        end

        we0 = 0;
        #20;
```

```
 $display("-----READ-----");
        for (i = 0; i < 16; i++)begin
            {25'd0, rd_addr0} = i;
            @(posedge clk);
            $display("addr:%h data:%h", rd_addr0, rd_dout0);
        end

        #20;
        $finish;
    end
    initial begin
        $dumpfile("data_mem_tb.vcd");
        $dumpvars(0, data_mem_tb);
    end
endmodule
```

Results can be seen in below:



```
-----WRITE-----
we: 1, addr:00, data:ffffffff, wr_strb:0
we: 1, addr:01, data:ffffffff, wr_strb:1
we: 1, addr:02, data:ffffffff, wr_strb:2
we: 1, addr:03, data:ffffffff, wr_strb:3
we: 1, addr:04, data:ffffffff, wr_strb:4
we: 1, addr:05, data:ffffffff, wr_strb:5
we: 1, addr:06, data:ffffffff, wr_strb:6
we: 1, addr:07, data:ffffffff, wr_strb:7
we: 1, addr:08, data:ffffffff, wr_strb:0
we: 1, addr:09, data:ffffffff, wr_strb:1
we: 1, addr:0a, data:ffffffff, wr_strb:2
we: 1, addr:0b, data:ffffffff, wr_strb:3
we: 1, addr:0c, data:ffffffff, wr_strb:4
we: 1, addr:0d, data:ffffffff, wr_strb:5
we: 1, addr:0e, data:ffffffff, wr_strb:6
we: 1, addr:0f, data:ffffffff, wr_strb:7
-----READ-----
addr:00 data:000000ff
addr:01 data:0000ffff
addr:02 data:ffffffff
addr:03 data:ffffffff
addr:04 data:000000ff
addr:05 data:0000ffff
addr:06 data:ffffffff
addr:07 data:ffffffff
addr:08 data:000000ff
addr:09 data:0000ffff
addr:0a data:ffffffff
addr:0b data:ffffffff
addr:0c data:000000ff
addr:0d data:0000ffff
addr:0e data:ffffffff
addr:0f data:00000000
```

Figure 4: Data Memory Simulation Results

Register File

Lastly, a 32-register RISC-V register file is developed with dual read ports. Special care is taken to ensure that register x0 always reads as zero and ignores write attempts, as dictated by the RISC-V specification. The register file undergoes full physical design flow using OpenLane, including synthesis, floorplanning, placement, routing, and signoff. The final layout is generated, and the design is analyzed for timing violations, DRC/LVS compliance, and antenna effects. The post-layout reports and layout view are included in the results section.

The HDL code for designing a register file, is provided below:

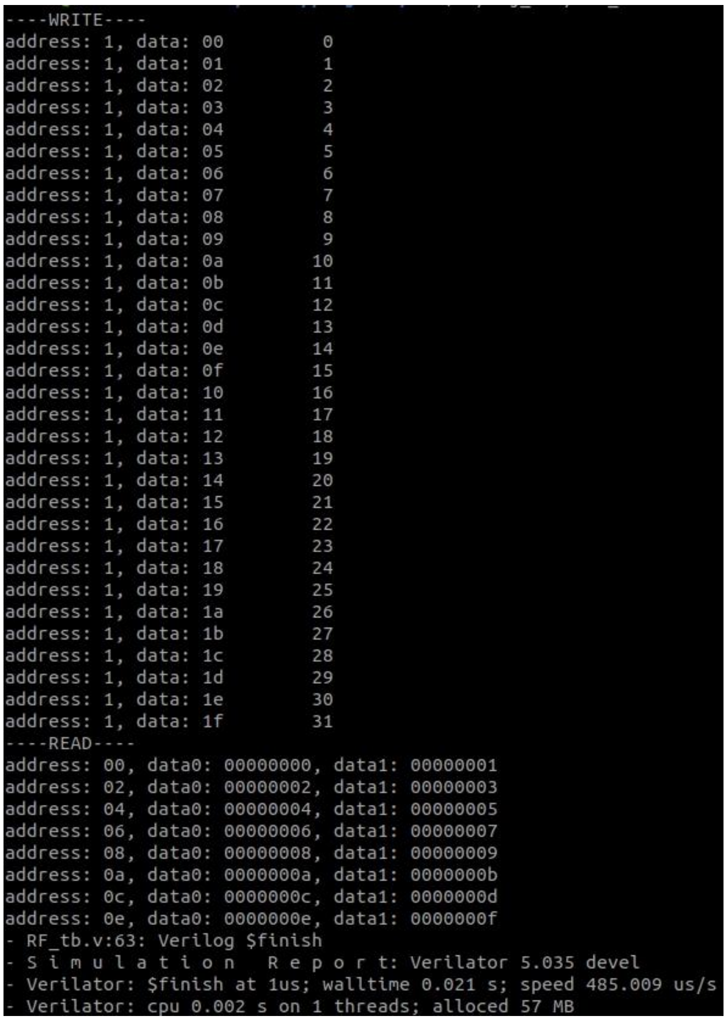
```
module RF#(
    parameter DEPTH = 32,
    parameter DATA_WIDTH = 32,
    parameter INIT_FILE = "")(
    input clk, rstn,
    input [$clog2(DEPTH)-1:0] rd_addr0, wr_addr0, rd_addr1,
    input [DATA_WIDTH-1:0] wr_din0,
    output [DATA_WIDTH-1:0] rd_dout0, rd_dout1,
    input we0
    );

    reg [DATA_WIDTH-1:0] mem [DEPTH-1:0];
    assign rd_dout0 = mem[rd_addr0];
    assign rd_dout1 = mem[rd_addr1];
    integer i;

    always @(posedge clk or negedge rstn) begin
        if(~rstn) begin
            for (i=0;i<=DEPTH;i=i+1) begin
                mem[i]<=0;
            end
        end
        else begin
            if((we0==1) & (|wr_addr0)!=0) begin
                mem[wr_addr0] <= wr_din0;
            end
        end
    end
endmodule
```

Deniz Zakir Eroğlu – 040200249 / Şeyma Çalışkan

The testbench code is below:

```
`timescale 1ns / 1ps
module RF_tb();
    reg clk, rstn;
    reg [4:0] rd_addr0, wr_addr0, rd_addr1;
    reg [31:0] wr_din0;
    wire [31:0] rd_dout0, rd_dout1;
    reg we0;

    RF uut(
        .clk(clk),
        .rstn(rstn),
        .we0(we0),
        .rd_addr0(rd_addr0),
        .rd_addr1(rd_addr1),
        .rd_dout0(rd_dout0),
        .rd_dout1(rd_dout1),
        .wr_addr0(wr_addr0),
        .wr_din0(wr_din0));

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

    integer i = 0;

    initial begin
        wr_addr0 = 0;
        rd_addr0 = 0;
        rd_addr1 = 0;
        we0 = 1;
        rstn = 1;
        clk = 0;
        #100;
        rstn = 0;
        #100;
        rstn = 1;
        wr_din0 = 32'hFFFFFFFF;
        #20;
        $display("-----WRITE-----");
        $display("we: %h addr:%h data:%h", we0, wr_addr0, wr_din0);
        wr_addr0 = 1;
        #40;
        $display("we: %h addr:%h data:%h", we0, wr_addr0, wr_din0);
        #20;
        we0 = 0;
        #20;
        $display("-----READ-----");
        $display("addr:%h data0:%h, data1:%h",rd_addr0, rd_out0, rd_out1);
        rd_addr0<=1;
        rd_addr1<=3;
        #40;
        $display("addr:%h data0:%h, data1:%h",rd_addr0, rd_out0, rd_out1);
    end
    always #10 clk=~clk;

endmodule
```

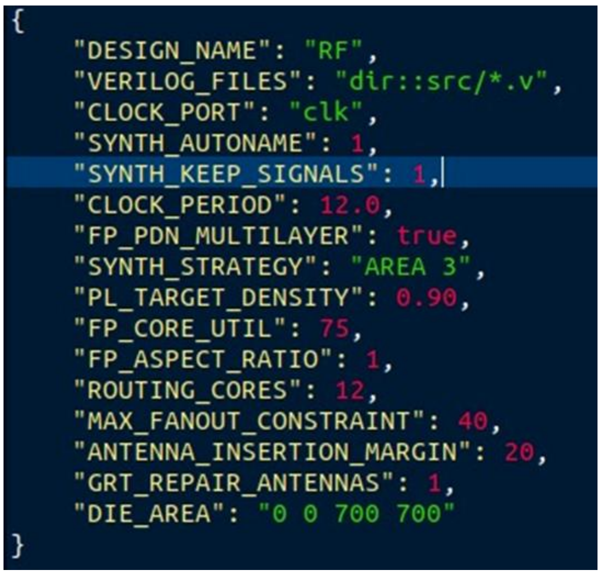
Results can be seen in below:



```
----WRITE----
address: 1, data: 00         0
address: 1, data: 01         1
address: 1, data: 02         2
address: 1, data: 03         3
address: 1, data: 04         4
address: 1, data: 05         5
address: 1, data: 06         6
address: 1, data: 07         7
address: 1, data: 08         8
address: 1, data: 09         9
address: 1, data: 0a        10
address: 1, data: 0b        11
address: 1, data: 0c        12
address: 1, data: 0d        13
address: 1, data: 0e        14
address: 1, data: 0f        15
address: 1, data: 10        16
address: 1, data: 11        17
address: 1, data: 12        18
address: 1, data: 13        19
address: 1, data: 14        20
address: 1, data: 15        21
address: 1, data: 16        22
address: 1, data: 17        23
address: 1, data: 18        24
address: 1, data: 19        25
address: 1, data: 1a        26
address: 1, data: 1b        27
address: 1, data: 1c        28
address: 1, data: 1d        29
address: 1, data: 1e        30
address: 1, data: 1f        31
----READ----
address: 00, data0: 00000000, data1: 00000001
address: 02, data0: 00000002, data1: 00000003
address: 04, data0: 00000004, data1: 00000005
address: 06, data0: 00000006, data1: 00000007
address: 08, data0: 00000008, data1: 00000009
address: 0a, data0: 0000000a, data1: 0000000b
address: 0c, data0: 0000000c, data1: 0000000d
address: 0e, data0: 0000000e, data1: 0000000f
- RF_tb.v:63: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.035 devel
- Verilator: $finish at 1us; walltime 0.021 s; speed 485.009 us/s
- Verilator: cpu 0.002 s on 1 threads; alloced 57 MB
```

Figure 5: Register file simulation results

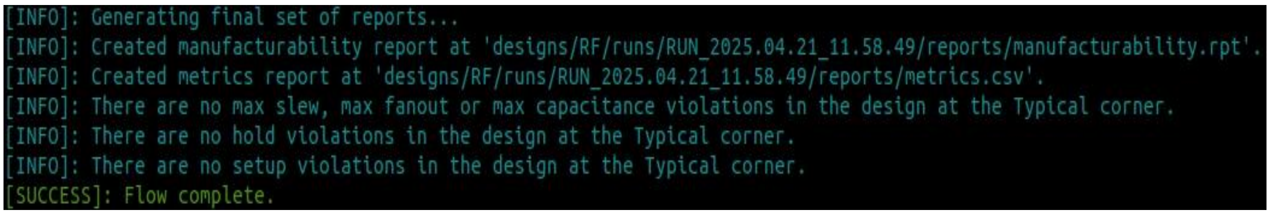
To start Openlane Flow, the config file can be seen below:



```
{
    "DESIGN_NAME": "RF",
    "VERILOG_FILES": "dir::src/*.v",
    "CLOCK_PORT": "clk",
    "SYNTH_AUTONAME": 1,
    "SYNTH_KEEP_SIGNALS": 1,
    "CLOCK_PERIOD": 12.0,
    "FP_PDN_MULTILAYER": true,
    "SYNTH_STRATEGY": "AREA 3",
    "PL_TARGET_DENSITY": 0.90,
    "FP_CORE_UTIL": 75,
    "FP_ASPECT_RATIO": 1,
    "ROUTING_CORES": 12,
    "MAX_FANOUT_CONSTRAINT": 40,
    "ANTENNA_INSERTION_MARGIN": 20,
    "GRT_REPAIR_ANTENNAS": 1,
    "DIE_AREA": "0 0 700 700"
}
```

Figure 6: Config File for RF

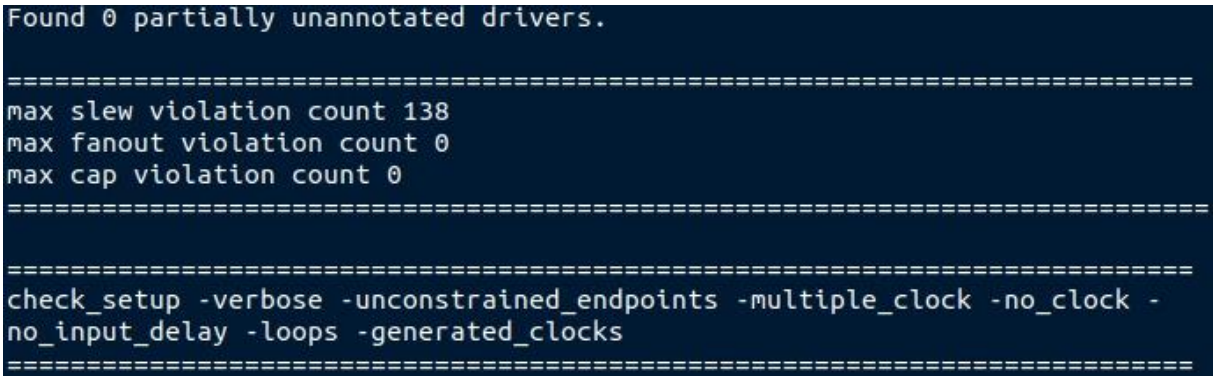
The OpenLane stage has been successfully completed. The output can be seen below:



```
[INFO]: Generating final set of reports...
[INFO]: Created manufacturability report at 'designs/RF/runs/RUN_2025.04.21_11.58.49/reports/manufacturability.rpt'.
[INFO]: Created metrics report at 'designs/RF/runs/RUN_2025.04.21_11.58.49/reports/metrics.csv'.
[INFO]: There are no max slew, max fanout or max capacitance violations in the design at the Typical corner.
[INFO]: There are no hold violations in the design at the Typical corner.
[INFO]: There are no setup violations in the design at the Typical corner.
[SUCCESS]: Flow complete.
```

Figure 7: OpenLane result

After preparing the config file, the run was started.

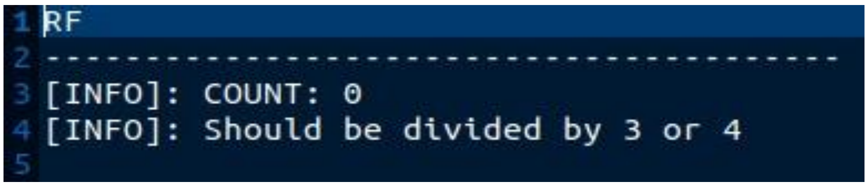


```
Found 0 partially unannotated drivers.

===========================================================================
max slew violation count 138
max fanout violation count 0
max cap violation count 0
============================================================================

===========================================================================
check_setup -verbose -unconstrained_endpoints -multiple_clock -no_clock -
no_input_delay -loops -generated_clocks
===========================================================================
```

Figure 8: Signoff/multi_corner_sta.checks.rpt file

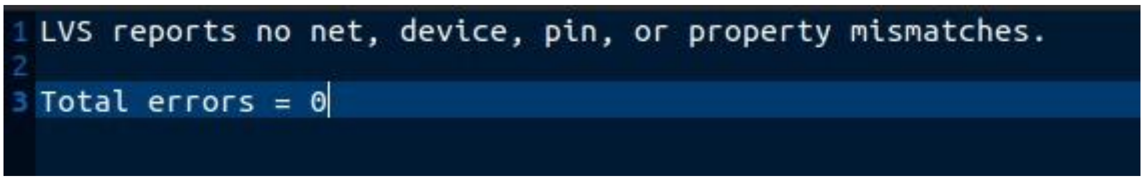
According to the signoff check.rpt 138 **slew violations**, while there are **no fanout** or **capacitance violations**. The check_setup command was run with verbose and multiple constraints-related options, indicating a detailed analysis across clocks, unconstrained endpoints, loops, and generated clocks.

To check DRC and LVS errors, results can be seen below:



```
RF
----------------------------------------
[INFO]: COUNT: 0
[INFO]: Should be divided by 3 or 4

```

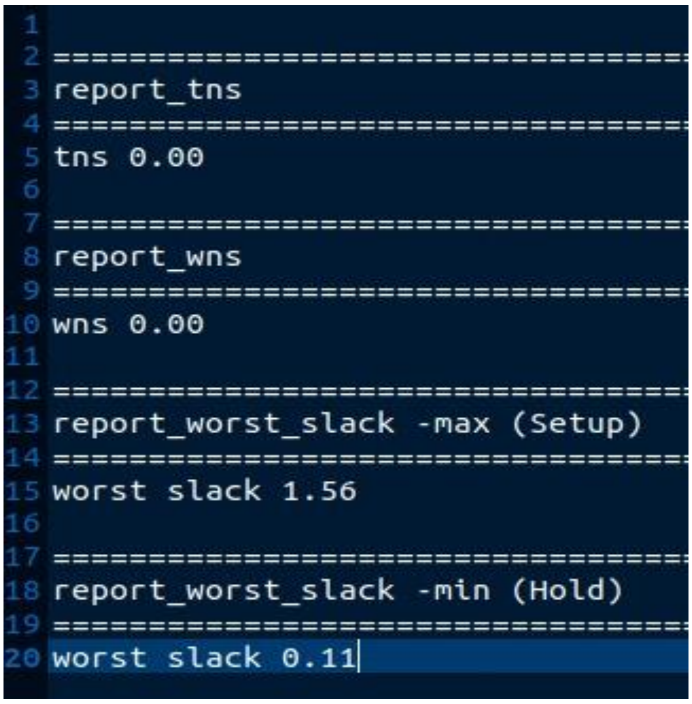
Figure 9: DRC reports



```
LVS reports no net, device, pin, or property mismatches.

Total errors = 0
```

Figure 10: *LVS reports*

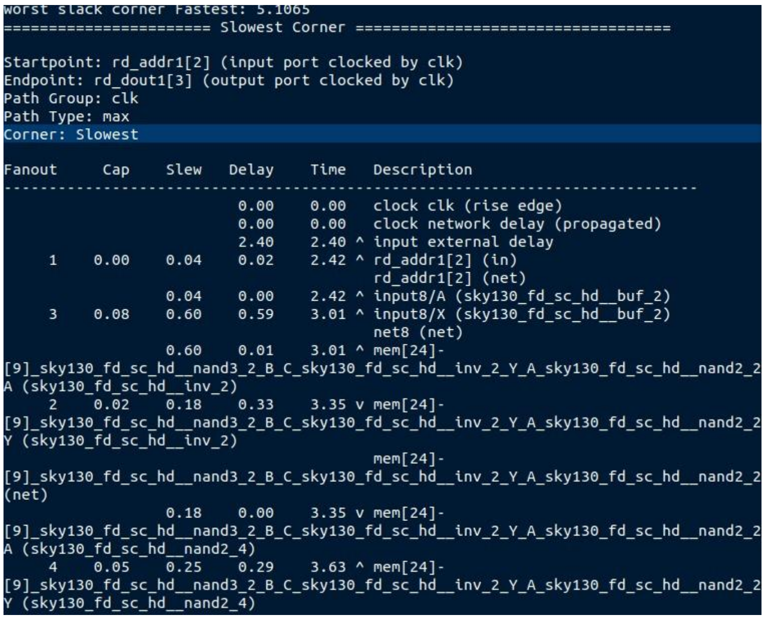
Timing analysis reports show that the design passes all setup and hold checks. Total Negative Slack (TNS) and Worst Negative Slack (WNS) are both zero, indicating no timing violations. The worst setup slack is 1.56 ns, and the worst hold slack is 0.11 ns, both of which are within acceptable limits



```

===================================
report_tns
===================================
tns 0.00

===================================
report_wns
===================================
wns 0.00

===================================
report_worst_slack -max (Setup)
===================================
worst slack 1.56

===================================
report_worst_slack -min (Hold)
===================================
worst slack 0.11
```

Figure 11: *Signoff/Clock Summary*

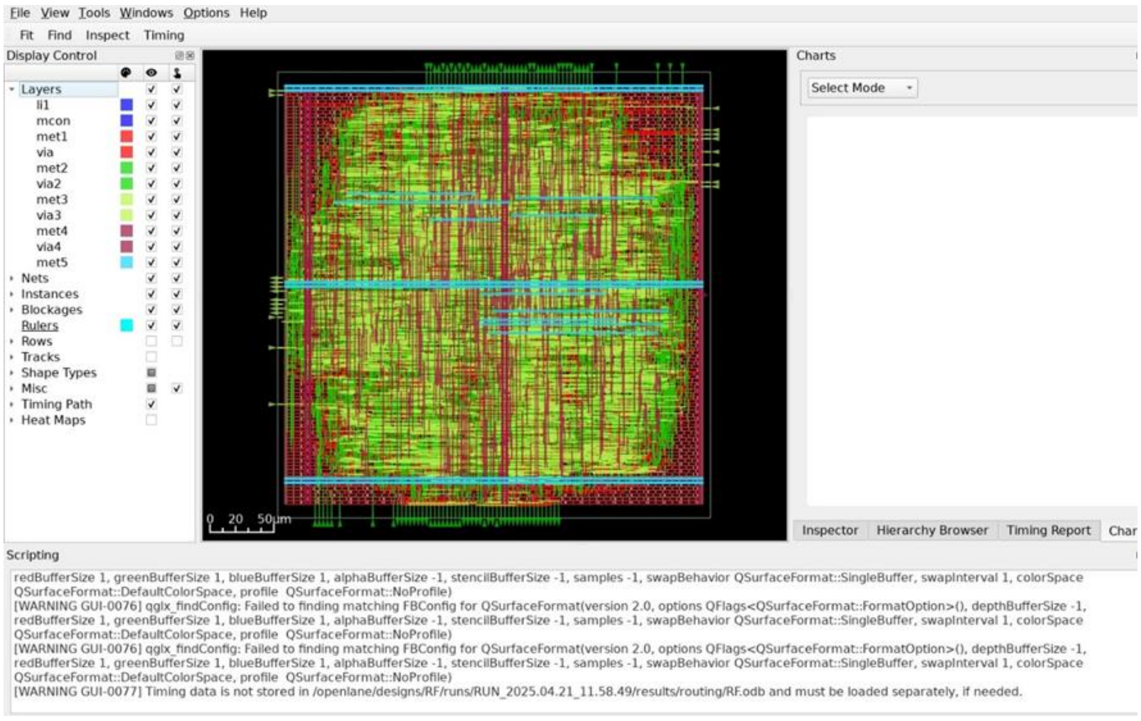
Also according to the signoff max corner timing results, the critical path from rd_addr1[2] to rd_dout1[3] has a worst-case delay of 5.106 ns in the slowest corner. There are no negative slacks, indicating setup timing is met for this path.



```
worst slack corner Fastest: 5.1065
====================== Slowest Corner ==================================

Startpoint: rd_addr1[2] (input port clocked by clk)
Endpoint: rd_dout1[3] (output port clocked by clk)
Path Group: clk
Path Type: max
Corner: Slowest

Fanout     Cap    Slew   Delay    Time   Description
-----------------------------------------------------------------------------
                          0.00    0.00   clock clk (rise edge)
                          0.00    0.00   clock network delay (propagated)
                          2.40    2.40 ^ input external delay
     1    0.00    0.04    0.02    2.42 ^ rd_addr1[2] (in)
                                         rd_addr1[2] (net)
                  0.04    0.00    2.42 ^ input8/A (sky130_fd_sc_hd__buf_2)
     3    0.08    0.60    0.59    3.01 ^ input8/X (sky130_fd_sc_hd__buf_2)
                                         net8 (net)
                  0.60    0.01    3.01 ^ mem[24]-
[9]_sky130_fd_sc_hd__nand3_2_B_C_sky130_fd_sc_hd__inv_2_Y_A_sky130_fd_sc_hd__nand2_2
A (sky130_fd_sc_hd__inv_2)
     2    0.02    0.18    0.33    3.35 v mem[24]-
[9]_sky130_fd_sc_hd__nand3_2_B_C_sky130_fd_sc_hd__inv_2_Y_A_sky130_fd_sc_hd__nand2_2
Y (sky130_fd_sc_hd__inv_2)
                                         mem[24]-
[9]_sky130_fd_sc_hd__nand3_2_B_C_sky130_fd_sc_hd__inv_2_Y_A_sky130_fd_sc_hd__nand2_2
(net)
                  0.18    0.00    3.35 v mem[24]-
[9]_sky130_fd_sc_hd__nand3_2_B_C_sky130_fd_sc_hd__inv_2_Y_A_sky130_fd_sc_hd__nand2_2
A (sky130_fd_sc_hd__nand2_4)
     4    0.05    0.25    0.29    3.63 ^ mem[24]-
[9]_sky130_fd_sc_hd__nand3_2_B_C_sky130_fd_sc_hd__inv_2_Y_A_sky130_fd_sc_hd__nand2_2
Y (sky130_fd_sc_hd__nand2_4)
```

Figure 12: multi_corner_sta.max.rpt

Post-Route Physical Design View: The final layout of the chip has been successfully visualized using the post-route .odb database. The metal routing layers from **met1 to met5** are visible, along with standard cell placements and net connections.



```
redBufferSize 1, greenBufferSize 1, blueBufferSize 1, alphaBufferSize -1, stencilBufferSize -1, samples -1, swapBehavior QSurfaceFormat::SingleBuffer, swapInterval 1, colorSpace
QSurfaceFormat::DefaultColorSpace, profile  QSurfaceFormat::NoProfile)
[WARNING GUI-0076] qglx_findConfig: Failed to finding matching FBConfig for QSurfaceFormat(version 2.0, options QFlags<QSurfaceFormat::FormatOption>(), depthBufferSize -1,
redBufferSize 1, greenBufferSize 1, blueBufferSize 1, alphaBufferSize -1, stencilBufferSize -1, samples -1, swapBehavior QSurfaceFormat::SingleBuffer, swapInterval 1, colorSpace
QSurfaceFormat::DefaultColorSpace, profile  QSurfaceFormat::NoProfile)
[WARNING GUI-0076] qglx_findConfig: Failed to finding matching FBConfig for QSurfaceFormat(version 2.0, options QFlags<QSurfaceFormat::FormatOption>(), depthBufferSize -1,
redBufferSize 1, greenBufferSize 1, blueBufferSize 1, alphaBufferSize -1, stencilBufferSize -1, samples -1, swapBehavior QSurfaceFormat::SingleBuffer, swapInterval 1, colorSpace
QSurfaceFormat::DefaultColorSpace, profile  QSurfaceFormat::NoProfile)
[WARNING GUI-0077] Timing data is not stored in /openlane/designs/RF/runs/RUN_2025.04.21_11.58.49/results/routing/RF.odb and must be loaded separately, if needed.
```

Figure 13: Openroad GUI Chip Viewer

Final layout, in KLayout can be seen in below:

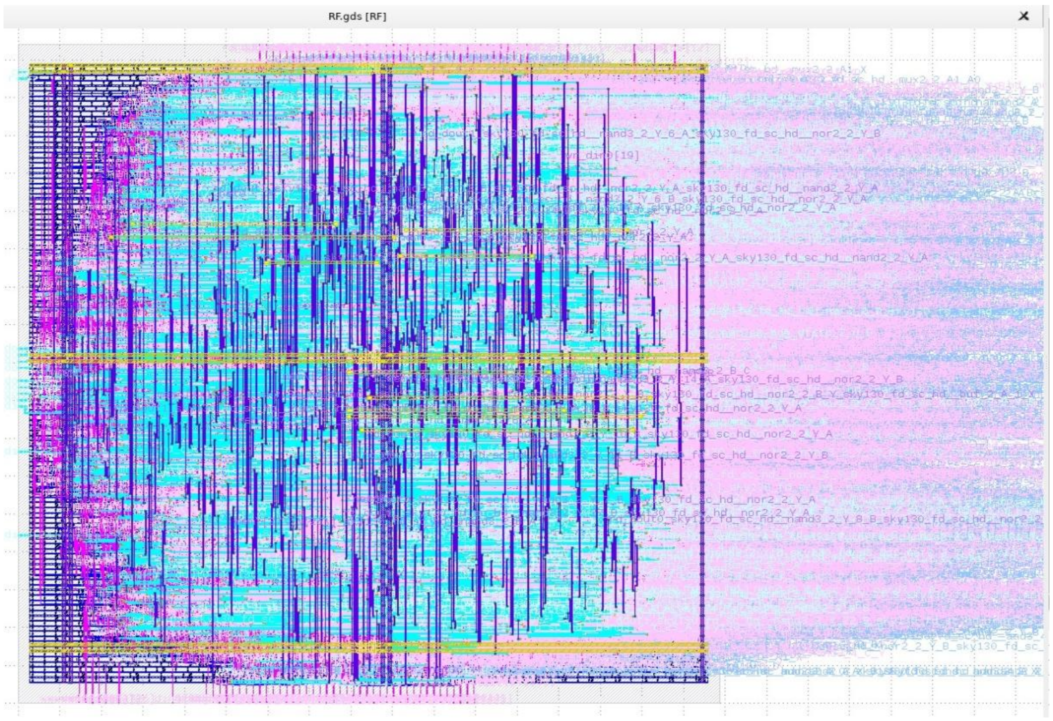
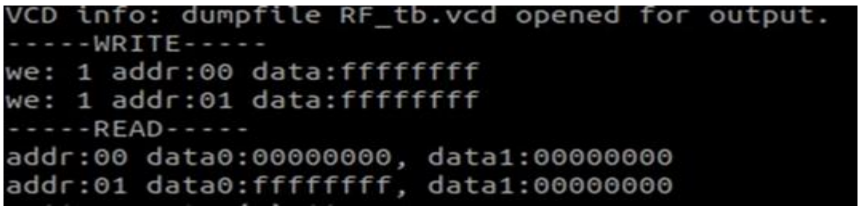


Figure 14: GDS in Klayout

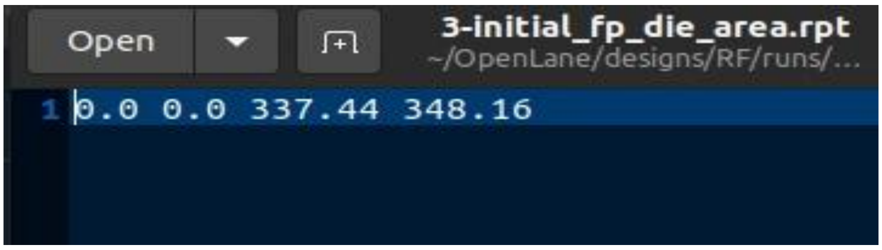
The post-route zero register simulation result is shown below. At this stage no trimming has been applied:



```
VCD info: dumpfile RF_tb.vcd opened for output.
-----WRITE-----
we: 1 addr:00 data:ffffffff
we: 1 addr:01 data:ffffffff
-----READ-----
addr:00 data0:00000000, data1:00000000
addr:01 data0:ffffffff, data1:00000000
```

Figure 15: Zero Register Simulation Results

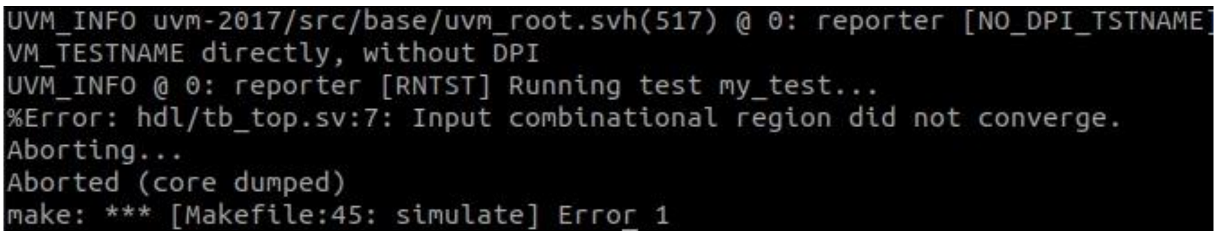
The result area has been determined as shown:



```
0.0 0.0 337.44 348.16
```

Figure 16: Die area

UVM Part:



```
UVM_INFO uvm-2017/src/base/uvm_root.svh(517) @ 0: reporter [NO_DPI_TSTNAME]
VM_TESTNAME directly, without DPI
UVM_INFO @ 0: reporter [RNTST] Running test my_test...
%Error: hdl/tb_top.sv:7: Input combinational region did not converge.
Aborting...
Aborted (core dumped)
make: *** [Makefile:45: simulate] Error 1
```

Figure 17: UVM result

As it can be seen in the Figure, the UVM testbenchs are unable to be run because of the verilator giving the error: Input combinational region did not converge. This might be because of a hardware loop existing in the test. We are unable to find a solution for this error right now, the UVM code will be uploaded and can be checked.