RISC-V Processor Design

Daeyoung Kim

April 22, 2025

1 Modules

To implement a soft RISC-V processor, I created several modules that work together to form the complete system. Each module is responsible for a specific part of the processor's functionality. In this section, I will describe each module briefly, including its purpose and how it interacts with other modules.

1.1 Types

The `rtl/types.sv` file contains various enumerations and constants used throughout the design. These include the instruction types, CPU states, RV32I op codes, funct3 bits, funct7 bits, and ALU control signals. Although SystemVerilog supports a convenient way to export this module as a package, since `yosys` does not support this feature, I had to use a workaround by converting every `.sv` file to a `.v` file using `sv2v`.

1.2 Arithmetic Logic Unit (ALU)

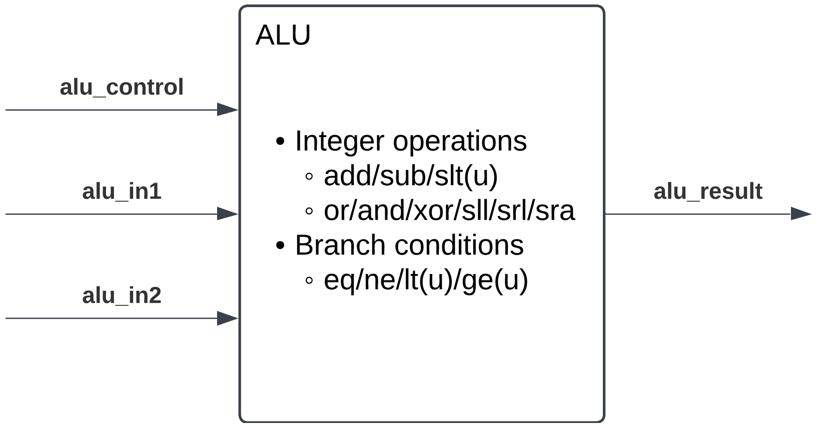

Figure 1: ALU

The ALU performs arithmetic and logical operations such as addition, subtraction, bitwise operations, and comparisons for R-type, I-type, and B-type instructions. It takes two operands and a control signal to determine which operation to execute.

1.3 Control Unit

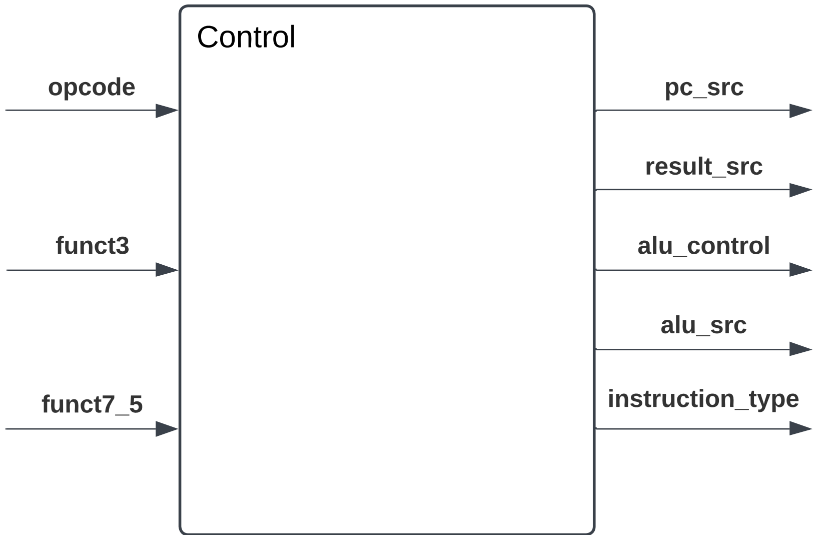

Figure 2: Control unit

The control unit decodes the instruction opcode and generates all necessary control signals for various multiplexers in the system – ALU operation, program counter updates, and register file write data source.

1.4 Immediate Extender

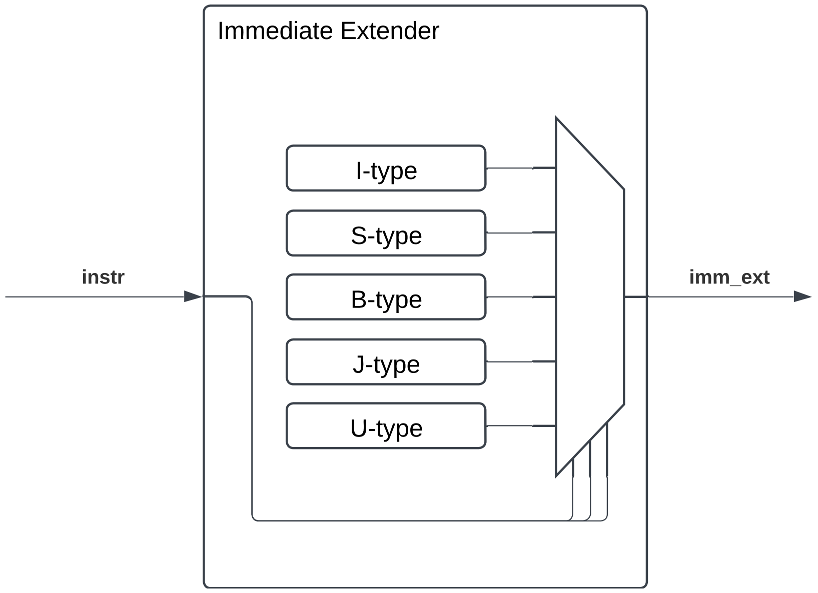

Figure 3: Immediate extender

The immediate extender takes the instruction and generates the immediate value based on opcode (the instruction type). The extended immediate value is used for address calculations, branch offsets, and immediate values for ALU operations, etc.

1.5 Register File

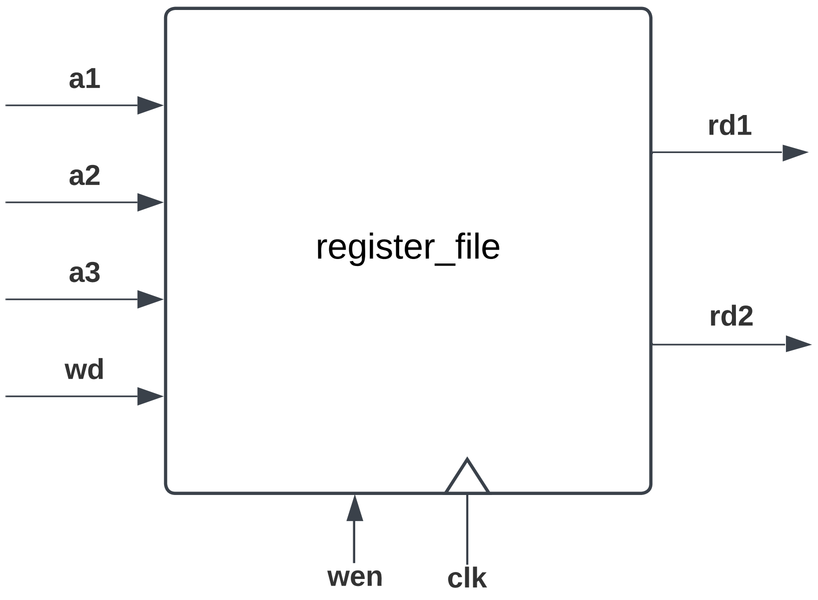

Figure 4: Register file

The register file contains 32 registers, where the 0th register is hardcoded to zero. It supports two simultaneous reads and one write operation. Although reads occur combinationally, writes occur synchronously at the rising edge of the clock.

1.6 Program Counter Selector

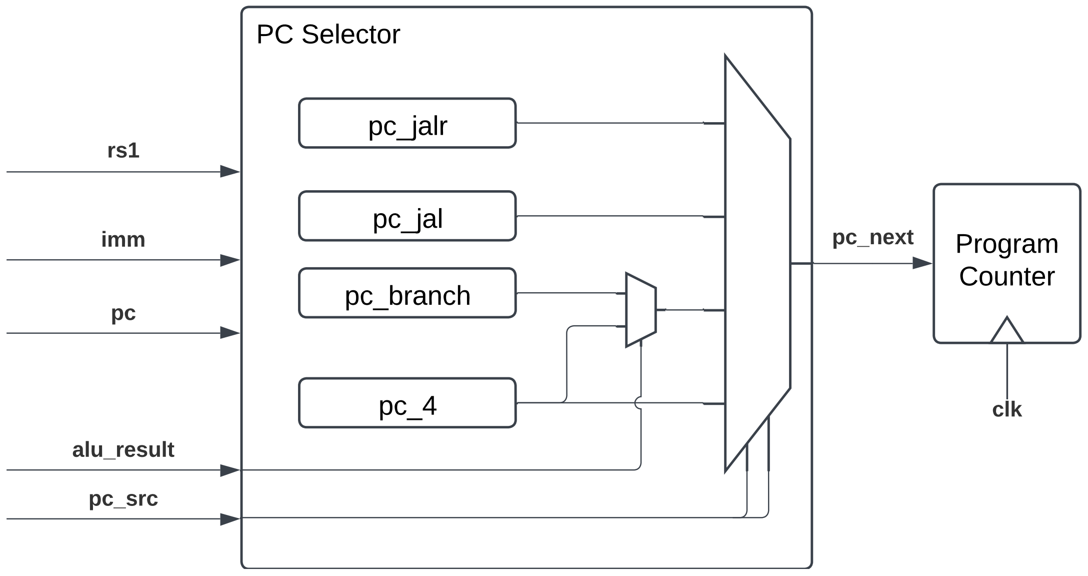

Figure 5: Program counter selector

The program counter selector computes the all four possible next addresses, and outputs the selected one based on the control signal. The four possible cases are:

- rs1 + immediate (JALR)

- PC + immediate (JAL)
- PC + immediate (branch when taken)
- PC + 4 (default case)

2 RISC-V

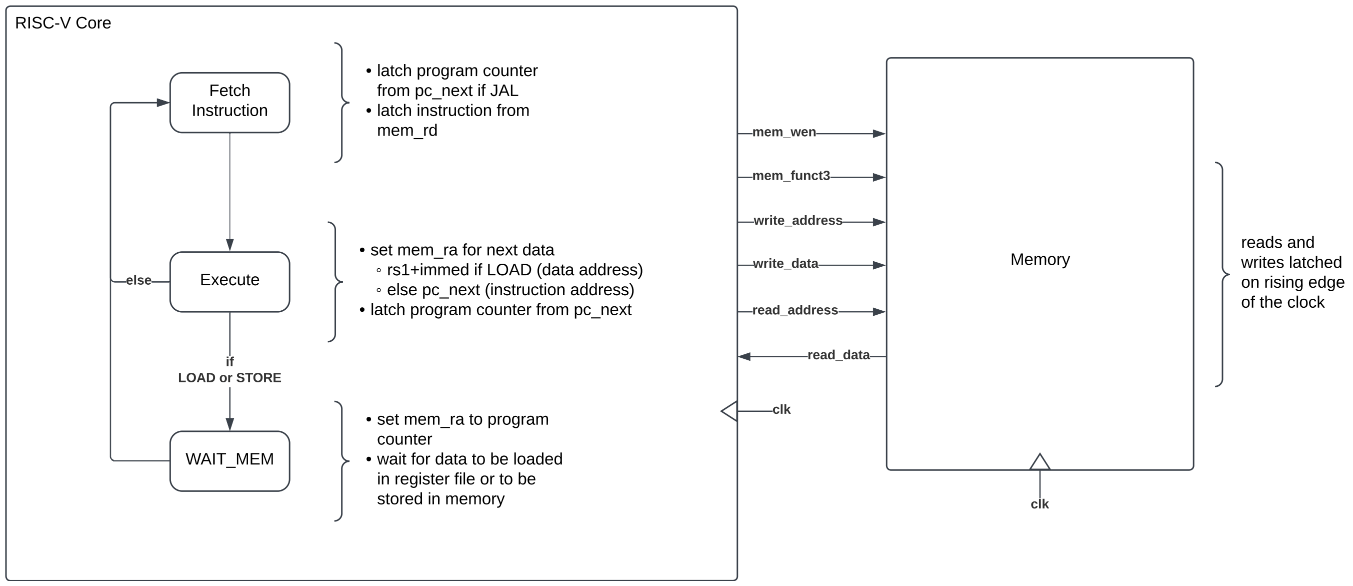
RISC-V Core
Fetch Instruction
Execute
WAIT_MEM
else
if LOAD or STORE
• latch program counter from pc_next if JAL
• latch instruction from mem_rd
• set mem_ra for next data
◦ rs1+immed if LOAD (data address)
◦ else pc_next (instruction address)
• latch program counter from pc_next
• set mem_ra to program counter
• wait for data to be loaded in register file or to be stored in memory
mem_wen
mem_funct3
write_address
write_data
read_address
read_data
clk
Memory
clk
reads and writes latched on rising edge of the clock

Figure 6: RISC-V processor connected to data and instruction memory

2.1 State Machine

The RISC-V processor operates as a finite state machine with three main states:

- `FETCH_INSTR`: The processor fetches the next instruction from memory. If the previous instruction was a jump (JAL), the program counter is updated here. Otherwise, it proceeds to the execute phase.
- `EXECUTE`: The processor performs computation or determines the next step based on the instruction type. If it's a load or store instruction, the processor transitions to the `WAIT_MEM` state to access memory. Otherwise, it immediately moves to the next instruction, usually after writing the results to the register file.
- `WAIT_MEM`: This state allows the processor to wait one cycle for memory to respond with the requested data (load) or acknowledge a store. Afterward, the state returns to `FETCH_INSTR`.

These states control instruction sequencing, memory access timing, and ensure correct operation for instructions involving memory. For all instructions except load and store, the processor completes the operation in two cycles, and three cycles otherwise. This is possible because the processor latches the next instruction at the end of `EXECUTE` state and most signals are calculated combinationally.

2.2 Top-Level Module

The top module connects the RISC-V core to the memory module in Von Neumann architecture, where data and instructions live in the same place. A system clock is generated either through simulation (`clk`) or by configuring a hardware oscillator and PLL to produce a 12 MHz clock for the physical FPGA. The RISC-V core sends memory read and write requests to the memory module based on the instruction. The memory module responds to read and write requests synchronously, using the funct3 control signal. The memory module also maps onboard LEDs (`LED`, `RGB_R`, `RGB_G`, `RGB_B`), and micros and millis counters.

3 Testbench

For testbenches, I used `cocotb` to write Python scripts that will interface with `iverilog` for simulation. Initially, I used module-level testbenches to verify the functionality of individual components. Then, I created an integrated testbench to test the entire RISC-V processor with a simple assembly programs written in the RV32I set.

3.1 Module-Level Testbenches

```python
import cocotb
from cocotb.triggers import Timer

@cocotb.test()
async def test_u_imm(dut):
    dut.instr.value = 0b00000000000000001110000010110111  # lui x1, 14
    imm_ext = 0b00000000000000001110000000000000  # {instr[31:12], {12{'0}}}
    await Timer(1, units="ns")
    assert dut.imm_ext.value == imm_ext
```

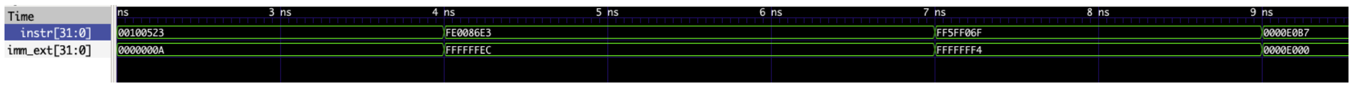

Figure 7: Waveform of the immediate extender testbench. The first value maps to the code snippet above, and the rest tests the other immediate types.

The above code is an example of a module-level testbench for the immediate extender module. It simply injects an I-type instruction (`lui x1, 14`) into `dut` (immediate extender) checks if it produces the expected immediate value. The results are shown in Figure 7. Similarly, I wrote testbenches for other modules to verify their functionality independent of the CPU.

3.2 Integrated Testbench

```python
import cocotb
from cocotb.triggers import ClockCycles
from utils import (
    init_clock,
    write_program_to_memory,
    reset_risc_v,
    get_register_value,
)
from functools import partial

@cocotb.test()
async def test_integer_register_immediate(dut):
    """
    Test integer register-immediate instructions (I-type).

    addi x1, x0, 0x80F    # pc = 0x00, x1 = 0xFFFF_F80F = -2033
    slti x2, x1, -2032    # pc = 0x04, x2 = 0x00000001; x1 < -2032
    sltiu x3, x1, 10   # pc = 0x08, x3 = 0x00000000; x1 > 10 (unsigned)
    sltiu x4, x0, 1      # pc = 0x0C, x4 = 0x00000001; x0 < 0x0000_0001 (unsigned) -- only possible when rs
    andi x5, x1, 0x0FF    # pc = 0x10, x5 = 0x0000_000F
    ori x6, x5, 0xFCF    # pc = 0x14, x6 = 0xFFFF_FFCF
    xori x7, x6, 0xFF0    # pc = 0x18, x7 = 0x0000_003F
    xori x8, x6, -1      # pc = 0x1C, x8 = 0x0000_0030
    slli x9, x7, 24       # pc = 0x1C, x9 = 0x3F00_0000
    srli x10, x9, 2       # pc = 0x20, x10 = 0x0FC0_0000
    srai x11, x10, 2      # pc = 0x24, x11 = 0x003F_0000
    """
    registers = partial(get_register_value, dut.u_risc_v.u_register_file)
    init_clock(dut)
    reset_risc_v(dut.u_risc_v)
    data = [
        0x80F00093,  # addi x1 x0 -2033
        0x8100A113,  # slti x2 x1 -2032
        0x00A0B193,  # sltiu x3 x1 10
        0x00103213,  # sltiu x4 x0 1
        0x0FF0F293,  # andi x5 x1 255
        0xFCF2E313,  # ori x6 x5 -49
        0xFF034393,  # xori x7 x6 -16
        0xFFF34413,  # xori x8 x6 -1
        0x01839493,  # slli x9 x7 24
        0x0024D513,  # srli x10 x9 2
        0x40255593,  # srai x11 x10 2
    ]
    write_program_to_memory(dut.u_memory, data)
    await ClockCycles(dut.clk, 2)

    await ClockCycles(dut.clk, 22)  # 11 instructions, 2 cycles each
    assert registers(1) == 0xFFFFF80F
    assert registers(2) == 0x00000001
    assert registers(3) == 0x00000000
    assert registers(4) == 0x00000001
    assert registers(5) == 0x0000000F
    assert registers(6) == 0xFFFFFFCF
    assert registers(7) == 0x0000003F
    assert registers(8) == 0x00000030
    assert registers(9) == 0x3F000000
    assert registers(10) == 0x0FC00000
    assert registers(11) == 0x03F00000
```

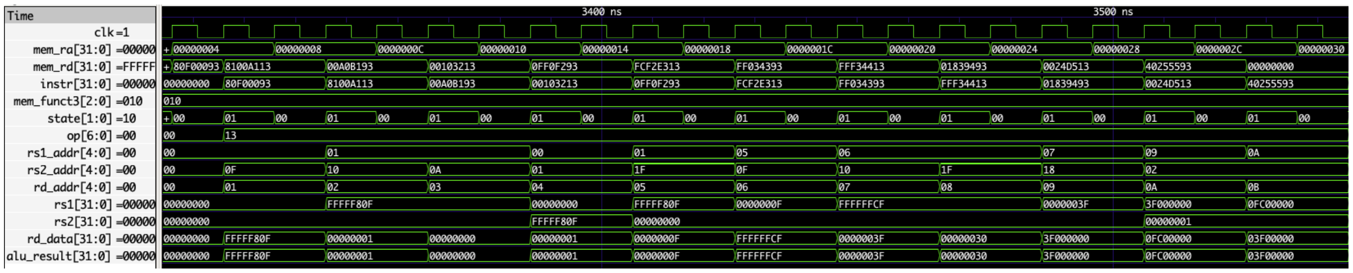

Figure 8: Waveform of the testbench for I-type instructions. The destination register data matches the expected values for each instruction.

Above is an example of an integrated testbench for the RISC-V processor. The `dut` object represents the top-level module of the RISC-V processor, which includes the core and memory module. Each tested instruction targets integer operation of I-type instructions, including immediate arithmetic operations and logical operations. These instructions are manually encoded into machine code and loaded into the memory model connected to the RISC-V processor. The test runs the simulation for 22 clock cycles (since the target instructions take 2 cycles each) and checks the values of registers `x1` through `x11` to ensure that each instruction executed as expected. Refer to Figure 8 for the waveform of the testbench.

```
lui x1, 0xFEDCC           # pc = 0x00, x1 = 0xFEDCC000
addi x1, x1, 0xA98        # pc = 0x04, x1 = 0xFEDCBA98
srli x2, x1, 4            # pc = 0x08, x2 = 0x0FEDCBA9
srai x3, x1, 4            # pc = 0x0C, x3 = 0xFFEDCBA9
xori x4, x3, -1           # pc = 0x10, x4 = 0x00123456
addi x5, x0, 2            # pc = 0x14, x5 = 0x00000002
add x6, x5, x4            # pc = 0x18, x6 = 0x00123458
sub x7, x6, x4            # pc = 0x1C, x7 = 0x00000002
sll x8, x4, x5            # pc = 0x20, x8 = 0x0048D158
ori x9, x8, 7             # pc = 0x24, x9 = 0x0048D15F
auipc x10, 0x12345        # pc = 0x28, x10 = 0x12345028
sw x1, 98(x5)             # pc = 0x2C, mem[0x00000002 + 98] = 0xFEDCBA98
lw x11, 98(x5)            # pc = 0x30, x11 = 0xFEDCBA98
```

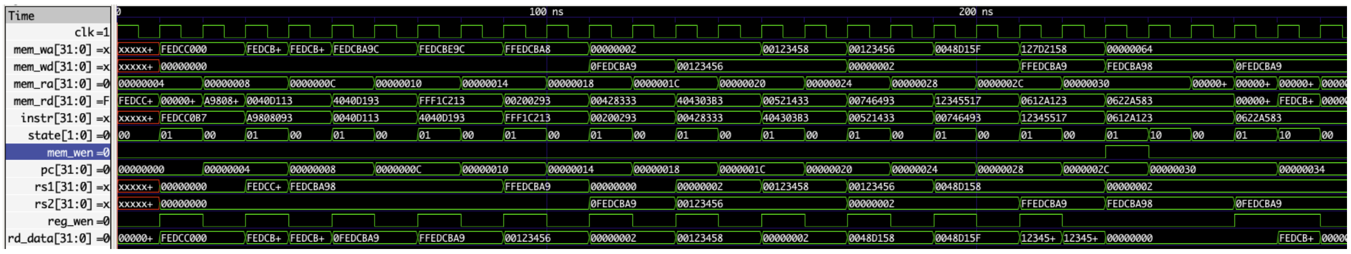

Figure 9: Waveform of the testbench for integer register-register and immediate instructions. The destination register data matches the expected values for each instruction.

Here's another sample waveform for an assembly program that tests R, I, U, and S instructions, including a store, load, and `AUIPC` instructions. Although Figure 9 shows the waveform of the testbench, it might be difficult to see the values of the registers. Please refer to the `test_r_i_u_s_instructions` function in the `tb/test_top.py` for more detail.

```
lui x1, 0xFF000   # lui for led bits; x1 = 0xFF00_0000
srli x2, x1, 8    # srai for red bits; x2 = 0x00FF_0000
xori x2, x2, -1    # xor for red bits; x2 = 0xFF00_FFFF
srli x3, x1, 16   # srli for green bits; x3 = 0x0000_FF00
xori x3, x3, -1    # xor for green bits; x3 = 0xFFFF_00FF
srli x4, x1, 24   # srli for blue bits; x4 = 0x0000_00FF
xori x4, x4, -1    # xor for blue bits; x4 = 0xFFFF_FF00
sw x2, -4(x0)     # store red bits at address 0xFFFF_FFFC
srli x6, x2, 28   # srli for counter; x6 = 0x0000_000F (very small delay for simulation)
addi x7, x7, 1    # increment counter x7
blt x7, x6, -4    # branch back up to increment if less than 0x000F_FFF0
addi x7, x0, 0    # reset counter x7 to 0
sw x3, -4(x0)     # store green bits at address 0xFFFF_FFFC
addi x7, x7, 1    # increment counter x7
blt x7, x6, -4    # branch back up to increment if less than 0x0000_000F
addi x7, x0, 0    # reset counter x7 to 0
sw x4, -4(x0)     # store blue bits at address 0xFFFF_FFFC
addi x7, x7, 1    # increment counter x7
blt x7, x6, -4    # branch back up to increment if less than 0x0000_000F
addi x7, x0, 0    # reset counter x7 to 0
jal x0, -52       # jump back up to storing red bits
```

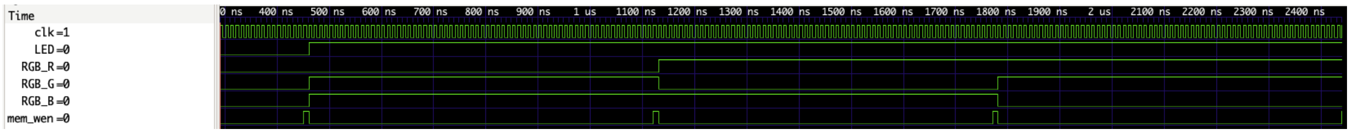

Figure 10: Waveform of the testbench for RGB LED. The waveform shows the RGB LEDs changing colors for a single loop.

Lastly, I wrote a simple assembly program to test the LED and RGB functionality of the RISC-V processor. The program uses the `sw` instruction to store the RGB values in the memory address mapped to the RGB LEDs. It uses `x7` register as a counter to create a delay between each color change by looping with branch and jump instructions. As expected, the waveform in Figure 10 shows the RGB LEDs changing colors in a loop.

4 RISC-V Compilation

The final step of the project was to compile a simple RISC-V program and run it on the soft RISC-V processor on FPGA. The program was written in C and compiled using the RISC-V GCC toolchain. I had to create a startup file to initialize the stack pointer to the top of the memory module (2 kB), but other than that, the program was straightforward. I tried two different programs: a simple program that loops the RGB LED and toggles the LED cyclically, and a more complex program that uses the micros peripheral for doing faded PWM transitions in RGB cycling. There is only the `main` function in both the programs and no other functions – with a function call, the program would not work as intended.

The first program is shown below with the oscilloscope screen capture in Figure 11.

```c
#include "peripherals.h"

int main(void) {
```

```
volatile char *led = (volatile char *)LED_ADDR;
volatile char *rgb_r = (volatile char *)RGB_R_ADDR;
volatile char *rgb_g = (volatile char *)RGB_G_ADDR;
volatile char *rgb_b = (volatile char *)RGB_B_ADDR;

*rgb_r = 0xFF;
*rgb_g = 0xFF;
*rgb_b = 0xFF;

while (1) {
*led = 0xFF;
*rgb_b = 0xFF;
*rgb_r = 0x00;
int count = CLK_HZ >> 5;
for (volatile int i = 0; i < count; i++);
*led = 0x00;
count = CLK_HZ >> 5;
for (volatile int i = 0; i < count; i++);
*led = 0xFF;
*rgb_r = 0xFF;
*rgb_g = 0x00;
count = CLK_HZ >> 5;
for (volatile int i = 0; i < count; i++);
*led = 0x00;
*rgb_g = 0xFF;
*rgb_b = 0x00;
count = CLK_HZ >> 5;
for (volatile int i = 0; i < count; i++);
}
return 0;
}
```

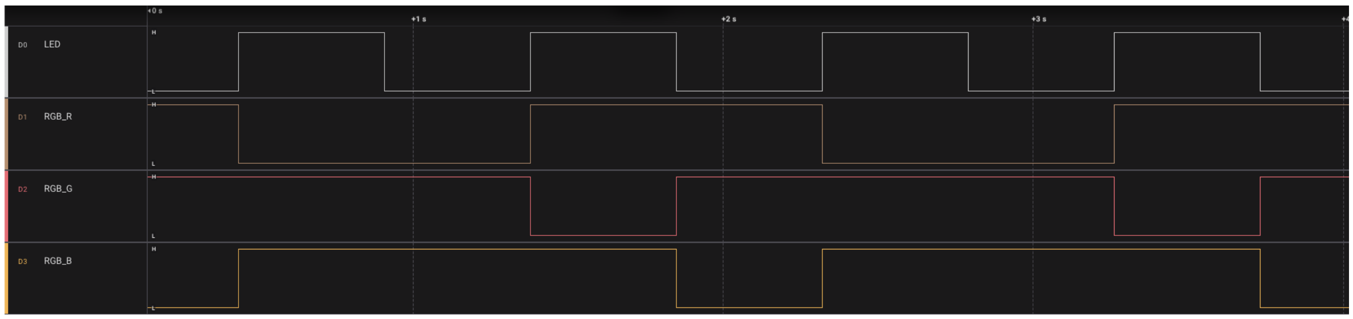

Figure 11: Oscilloscope result for RGB LED and LED. The waveform shows the LEDs changing colors for a single loop. The LEDs were mapped to GPIO pins for debugging. The oscilloscope has glitch filters enabled for 85 ns to avoid the single-clock-period (∼83 ns) glitch when the PWM counter overflows for the sake of clarity.