VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Computer Science and Engineering

LAB 3
INTRODUCTION TO SYSTEM ON CHIP
LABORATORY REPORT

Instructor: Pham Kieu Nhat Anh

Class: CC01
Name: Hoang Thuy Tram
Student ID: 2353202

Ho Chi Minh city, December 2025

Contents

1 Github

The Lab 3 project files (RTL source code, testbenches, simulation scripts) are located in the `Lab03` folder. They are available at the following repository: https://github.com/trxmhoang/SoC.git

2 Overview

The `DatapathSingleCycle.v` file details the design and implementation of a 32-bit single-cycle RISC-V processor using Verilog. The processor implements the RV32I base integer instruction set. The architecture is single-cycle, meaning every instruction completes its execution within one clock cycle, whether it is a simple addition or a complex memory access. The design consists of four main modules:

- RegFile: The register file contains 32 general-purpose registers.
  - The RISC-V architecture specifies 32 registers (x0 to x31), each 32 bits wide. `RegFile` module is used to implement this storage.
  - In RISC-V, register x0 is hardwired to zero. Therefore, the module also checks if the destination register is x0 to ensures that any instruction trying to write to x0 effectively does nothing.
  - Writing to registers happens on the `posedge clk`, ensuring state updates are synchronized.
  - Reading is combination, which happens immediately, allowing the datapath to use register values within the same clock cycle.
- DatapathSingleCycle: The core logic that handles instruction decoding, ALU operations and control flow. It is the "brain" of the processor.
  - The 32-bit instruction received from instruction memory is sliced into specific fields. This splitting allows the processor to immediately identify the source registers (rs1, rs2), the destination register (rd) and the operation codes (opcode, funct3, funct7).
  - RISC-V instructions pack immediate values (constants) differently depending on the instruction type (I, S, B, J). The code extracts these bits and performs sign extension, replicates the most significant bit (MSB) to fill the upper bits of the 32-bit word, allowing the processor to handle negative numbers correctly in arithmetic operations.
  - The core logic is implemented in a large combinatorial `always @(*)` block. This block acts as the control unit. Based on the `inst_opcode`, the processor determines ALU operations (add, subtract, XOR, etc), memory access (read/write to memory via `OpLoad, OpStore`) and branching (whether to change the Program Counter (`pcNext`) based on the comparisons).
- MemorySingleCycle: A unified memory module for both instructions and data.
  - The memory module simulates a Von Neumann architecture physically with one memory array, but behaves like a Harvard architecture with separate ports for instruction and data to support single-cycle execution.
  - While the memory array is 32-bits wide, RISC-V addresses memory by byte. The code handles this using `store_we_dmem`, a 4-bit write enable mask. This allows writing individual bytes (SB), half-word (SH) or full words (SW) without corrupting neighbors.

- Processor: The top-level module connecting the datapath and memory.

In this design, we use `cla` module from Lab 2 to implement the `addi, add,` and `sub` instructions for fast addition and subtraction. Other situations where we need to add things, such as incrementing the PC or computing branch targets, we will use the `'+'` operator. Module `divu_1iter, divider_unsigned` from Lab 1 are also used to replace `'/', '%'` operators that handle division and remainder operations.

The logic is designed so that `pcCurrent` updates exactly once per clock edge. In the time between clock edges, the signal propagates through:

- Instruction memory fetch
- Register read
- ALU execution
- Data memory access
- Register write back

This design is conceptually simple and easy to debug. The CPI (Cycles Per Instruction) is exactly 1. However, the clock frequency is limited by the longest possible path (usually a Load instruction), making it slower than pipelined architectures in terms of absolute time.

3 RTL implementation

The provided Verilog code successfully implements a functional subset of the RISC-V 32-bit instruction set. By modularizing the design into Datapath, Control (embedded in Datapath) and Memory, the system achieves a clear separation of concerns. The use of combinatorial logic for instruction decoding ensures that the processor can react to new instructions immediately within the single clock cycle, satisfying the requirements of the architecture.

```verilog
`timescale 1ns / 1ns

// registers are 32 bits in RV32
`define REG_SIZE 31

// RV opcodes are 7 bits
`define OPCODE_SIZE 6

// Don't forget your previous ALUs
/* `include "divider_unsigned.v"
`include "cla.v" */

module RegFile (
    input       [         4:0] rd,
    input       [`REG_SIZE:0] rd_data,
    input       [         4:0] rs1,
    output reg [`REG_SIZE:0] rs1_data,
    input       [         4:0] rs2,
    output reg [`REG_SIZE:0] rs2_data,
    input                      clk,
    input                     we,
    input                     rst
);
  localparam NumRegs = 32;
  reg [`REG_SIZE:0] regs[0:NumRegs-1];
  integer i;

// TODO: your code here
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      for (i = 0; i < NumRegs; i = i + 1) begin
        regs[i] <= 0;
      end
    end else if (we && (rd != 0)) begin
      regs[rd] <= rd_data;
    end
  end

  always @(*) begin
    rs1_data = (rs1 == 0) ? 32'd0 : regs[rs1];
    rs2_data = (rs2 == 0) ? 32'd0 : regs[rs2];
  end
endmodule

module DatapathSingleCycle (
    input                     clk,
    input                     rst,
    output reg                halt,
    output      [`REG_SIZE:0] pc_to_imem,
    input       [`REG_SIZE:0] inst_from_imem,
    // addr_to_dmem is a read-write port
    output reg [`REG_SIZE:0] addr_to_dmem,
    input       [`REG_SIZE:0] load_data_from_dmem,
    output reg [`REG_SIZE:0] store_data_to_dmem,
    output reg [         3:0] store_we_to_dmem
);

  // components of the instruction
  wire [           6:0] inst_funct7;
  wire [           4:0] inst_rs2;
  wire [           4:0] inst_rs1;
  wire [           2:0] inst_funct3;
  wire [           4:0] inst_rd;
  wire [`OPCODE_SIZE:0] inst_opcode;

  // split R-type instruction - see section 2.2 of RiscV spec
  assign {inst_funct7, inst_rs2, inst_rs1, inst_funct3, inst_rd,
     inst_opcode} = inst_from_imem;

  // setup for I, S, B & J type instructions
  // I - short immediates and loads
  wire [11:0] imm_i;
  assign imm_i = inst_from_imem[31:20];
```

```
wire [ 4:0] imm_shamt = inst_from_imem[24:20];

// S - stores
wire [11:0] imm_s;
assign imm_s = {inst_funct7, inst_rd};

// B - conditionals
wire [12:0] imm_b;
assign {imm_b[12], imm_b[10:1], imm_b[11], imm_b[0]} = {inst_funct7,
    inst_rd, 1'b0};

// J - unconditional jumps
wire [20:0] imm_j;
assign {imm_j[20], imm_j[10:1], imm_j[11], imm_j[19:12], imm_j[0]} =
    {inst_from_imem[31:12], 1'b0};

wire [`REG_SIZE:0] imm_i_sext = {{20{imm_i[11]}}, imm_i[11:0]};
wire [`REG_SIZE:0] imm_s_sext = {{20{imm_s[11]}}, imm_s[11:0]};
wire [`REG_SIZE:0] imm_b_sext = {{19{imm_b[12]}}, imm_b[12:0]};
wire [`REG_SIZE:0] imm_j_sext = {{11{imm_j[20]}}, imm_j[20:0]};

// opcodes - see section 19 of RiscV spec
localparam [`OPCODE_SIZE:0] OpLoad    = 7'b00_000_11;
localparam [`OPCODE_SIZE:0] OpStore   = 7'b01_000_11;
localparam [`OPCODE_SIZE:0] OpBranch  = 7'b11_000_11;
localparam [`OPCODE_SIZE:0] OpJalr    = 7'b11_001_11;
localparam [`OPCODE_SIZE:0] OpMiscMem = 7'b00_011_11;
localparam [`OPCODE_SIZE:0] OpJal     = 7'b11_011_11;

localparam [`OPCODE_SIZE:0] OpRegImm  = 7'b00_100_11;
localparam [`OPCODE_SIZE:0] OpRegReg  = 7'b01_100_11;
localparam [`OPCODE_SIZE:0] OpEnviron = 7'b11_100_11;

localparam [`OPCODE_SIZE:0] OpAuipc   = 7'b00_101_11;
localparam [`OPCODE_SIZE:0] OpLui     = 7'b01_101_11;

wire inst_lui    = (inst_opcode == OpLui    );
wire inst_auipc  = (inst_opcode == OpAuipc  );
wire inst_jal    = (inst_opcode == OpJal    );
wire inst_jalr   = (inst_opcode == OpJalr   );

wire inst_beq    = (inst_opcode == OpBranch ) & (inst_from_imem
   [14:12] == 3'b000);
wire inst_bne    = (inst_opcode == OpBranch ) & (inst_from_imem
   [14:12] == 3'b001);
wire inst_blt    = (inst_opcode == OpBranch ) & (inst_from_imem
   [14:12] == 3'b100);
wire inst_bge    = (inst_opcode == OpBranch ) & (inst_from_imem
   [14:12] == 3'b101);
wire inst_bltu   = (inst_opcode == OpBranch ) & (inst_from_imem
   [14:12] == 3'b110);
wire inst_bgeu   = (inst_opcode == OpBranch ) & (inst_from_imem
   [14:12] == 3'b111);
```

```verilog

wire inst_lb     = (inst_opcode == OpLoad   ) & (inst_from_imem[14:12] == 3'b000);
wire inst_lh     = (inst_opcode == OpLoad   ) & (inst_from_imem[14:12] == 3'b001);
wire inst_lw     = (inst_opcode == OpLoad   ) & (inst_from_imem[14:12] == 3'b010);
wire inst_lbu    = (inst_opcode == OpLoad   ) & (inst_from_imem[14:12] == 3'b100);
wire inst_lhu    = (inst_opcode == OpLoad   ) & (inst_from_imem[14:12] == 3'b101);

wire inst_sb     = (inst_opcode == OpStore  ) & (inst_from_imem[14:12] == 3'b000);
wire inst_sh     = (inst_opcode == OpStore  ) & (inst_from_imem[14:12] == 3'b001);
wire inst_sw     = (inst_opcode == OpStore  ) & (inst_from_imem[14:12] == 3'b010);

wire inst_addi   = (inst_opcode == OpRegImm ) & (inst_from_imem[14:12] == 3'b000);
wire inst_slti   = (inst_opcode == OpRegImm ) & (inst_from_imem[14:12] == 3'b010);
wire inst_sltiu  = (inst_opcode == OpRegImm ) & (inst_from_imem[14:12] == 3'b011);
wire inst_xori   = (inst_opcode == OpRegImm ) & (inst_from_imem[14:12] == 3'b100);
wire inst_ori    = (inst_opcode == OpRegImm ) & (inst_from_imem[14:12] == 3'b110);
wire inst_andi   = (inst_opcode == OpRegImm ) & (inst_from_imem[14:12] == 3'b111);

wire inst_slli   = (inst_opcode == OpRegImm ) & (inst_from_imem[14:12] == 3'b001) & (inst_from_imem[31:25] == 7'd0      );
wire inst_srli   = (inst_opcode == OpRegImm ) & (inst_from_imem[14:12] == 3'b101) & (inst_from_imem[31:25] == 7'd0      );
wire inst_srai   = (inst_opcode == OpRegImm ) & (inst_from_imem[14:12] == 3'b101) & (inst_from_imem[31:25] == 7'b0100000);

wire inst_add    = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b000) & (inst_from_imem[31:25] == 7'd0      );
wire inst_sub    = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b000) & (inst_from_imem[31:25] == 7'b0100000);
wire inst_sll    = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b001) & (inst_from_imem[31:25] == 7'd0      );
wire inst_slt    = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b010) & (inst_from_imem[31:25] == 7'd0      );
wire inst_sltu   = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b011) & (inst_from_imem[31:25] == 7'd0      );
wire inst_xor    = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b100) & (inst_from_imem[31:25] == 7'd0      );
wire inst_srl    = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b101) & (inst_from_imem[31:25] == 7'd0      );
wire inst_sra    = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b101) & (inst_from_imem[31:25] == 7'b0100000);
wire inst_or     = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b110) & (inst_from_imem[31:25] == 7'd0      );
wire inst_and    = (inst_opcode == OpRegReg ) & (inst_from_imem[14:12] == 3'b111) & (inst_from_imem[31:25] == 7'd0      );

wire inst_mul    = (inst_opcode == OpRegReg ) & (inst_from_imem[31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b000    );
wire inst_mulh   = (inst_opcode == OpRegReg ) & (inst_from_imem[31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b001    );
wire inst_mulhsu = (inst_opcode == OpRegReg ) & (inst_from_imem[31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b010    );
wire inst_mulhu  = (inst_opcode == OpRegReg ) & (inst_from_imem[31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b011    );
wire inst_div    = (inst_opcode == OpRegReg ) & (inst_from_imem[31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b100    );
wire inst_divu   = (inst_opcode == OpRegReg ) & (inst_from_imem[31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b101    );
wire inst_rem    = (inst_opcode == OpRegReg ) & (inst_from_imem[31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b110    );
wire inst_remu   = (inst_opcode == OpRegReg ) & (inst_from_imem[31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b111    );

wire inst_ecall  = (inst_opcode == OpEnviron) & (inst_from_imem[31:7] == 25'd0  );
wire inst_fence  = (inst_opcode == OpMiscMem);

// program counter
reg [`REG_SIZE:0] pcNext, pcCurrent;
always @(posedge clk) begin
  if (rst) begin
    pcCurrent <= 32'd0;
  end else begin
    pcCurrent <= pcNext;
  end
end
assign pc_to_imem = pcCurrent;

// cycle/inst._from_imem counters
reg [`REG_SIZE:0] cycles_current, num_inst_current;
always @(posedge clk) begin
  if (rst) begin
    cycles_current <= 0;
    num_inst_current <= 0;
  end else begin
    cycles_current <= cycles_current + 1;
    if (!rst) begin
      num_inst_current <= num_inst_current + 1;
    end
  end
end

```

```
// NOTE: don't rename your RegFile instance as the tests expect it
   to be `rf`
// TODO: you will need to edit the port connections, however.
wire [`REG_SIZE:0] rs1_data;
wire [`REG_SIZE:0] rs2_data;
reg [`REG_SIZE:0] rd_data;
reg rf_we;

RegFile rf (
  .clk      (clk),
  .rst      (rst),
  .we       (rf_we),
  .rd       (inst_rd),
  .rd_data  (rd_data),
  .rs1      (inst_rs1),
  .rs2      (inst_rs2),
  .rs1_data (rs1_data),
  .rs2_data (rs2_data)
);

wire [31:0] cla_res, cla_op_b, cla_op_b_raw;
wire cla_cin;

assign cla_op_b_raw = (inst_opcode == OpRegImm) ? imm_i_sext :
   rs2_data;
assign cla_op_b = inst_sub ? ~cla_op_b_raw : cla_op_b_raw;
assign cla_cin = inst_sub ? 1'b1 : 1'b0;

cla u_cla (
  .a   (rs1_data),
  .b   (cla_op_b),
  .cin (cla_cin),
  .sum (cla_res)
);

wire [31:0] div_op_a, div_op_b;
wire is_signed_div;
assign is_signed_div = inst_div | inst_rem;
assign div_op_a = (is_signed_div & rs1_data[31]) ? (~rs1_data + 1) :
    rs1_data;
assign div_op_b = (is_signed_div & rs2_data[31]) ? (~rs2_data + 1) :
    rs2_data;

wire [31:0] div_quot_raw, div_rem_raw;
divider_unsigned u_div (
  .dividend  (div_op_a),
  .divisor   (div_op_b),
  .quotient  (div_quot_raw),
  .remainder (div_rem_raw)
);

reg [31:0] div_res;
always @(*) begin
```

```
    if (inst_div || inst_divu) begin
      if (rs2_data == 0) begin
        div_res = 32'hFFFF_FFFF;
      end else if (inst_div && (rs1_data == 32'h8000_0000) && (rs2_data == 32'hFFFF_FFFF)) begin
        div_res = 32'h8000_0000;
      end else if (is_signed_div && (rs1_data[31] ^ rs2_data[31])) begin
        div_res = ~div_quot_raw + 32'd1;
      end else begin
        div_res = div_quot_raw;
      end
    end else begin // rem or remu
      if (rs2_data == 0) begin
        div_res = rs1_data;
      end else if (inst_rem && (rs1_data == 32'h8000_0000) && (rs2_data == 32'hFFFF_FFFF)) begin
        div_res = 32'd0;
      end else if (is_signed_div && (rs1_data[31])) begin
        div_res = ~div_rem_raw + 32'd1;
      end else begin
        div_res = div_rem_raw;
      end
    end
  end

  wire [63:0] mul_u, mul_s, mul_su;
  assign mul_u = {32'b0, rs1_data} * {32'b0, rs2_data};
  assign mul_s = $signed(rs1_data) * $signed(rs2_data);
  assign mul_su = $signed(rs1_data) * $signed({1'b0, rs2_data});

  reg take_branch;
  always @(*) begin
    case (inst_funct3)
      3'b000: take_branch = (rs1_data == rs2_data);                     // beq
      3'b001: take_branch = (rs1_data != rs2_data);                     // bne
      3'b100: take_branch = ($signed(rs1_data) < $signed(rs2_data));    // blt
      3'b101: take_branch = ($signed(rs1_data) >= $signed(rs2_data));   // bge
      3'b110: take_branch = (rs1_data < rs2_data);                      // bltu
      3'b111: take_branch = (rs1_data >= rs2_data);                     // bgeu
      default: take_branch = 1'b0;
    endcase
  end

  reg [31:0] load_data;
  always @(*) begin
    case (addr_to_dmem[1:0])
```

```
      2'b00: load_data = load_data_from_dmem;
      2'b01: load_data = load_data_from_dmem >> 8;
      2'b10: load_data = load_data_from_dmem >> 16;
      2'b11: load_data = load_data_from_dmem >> 24;
    endcase
  end

  reg illegal_inst;

  always @(*) begin
    illegal_inst = 1'b0;
    pcNext = pcCurrent + 4;
    rf_we = 0;
    rd_data = 0;
    halt = 0;
    addr_to_dmem = 0;
    store_data_to_dmem = 0;
    store_we_to_dmem = 0;

    case (inst_opcode)
      OpLui: begin
        // TODO: start here by implementing lui
        rf_we = 1;
        rd_data = {inst_from_imem[31:12], 12'b0};
      end

      OpAuipc: begin
        rf_we = 1;
        rd_data = pcCurrent + {inst_from_imem[31:12], 12'b0};
      end

      OpJal: begin
        rf_we = 1;
        rd_data = pcCurrent + 4;
        pcNext = pcCurrent + imm_j_sext;
      end

      OpJalr: begin
        rf_we = 1;
        rd_data = pcCurrent + 4;
        pcNext = (rs1_data + imm_i_sext) & ~32'd1;
      end

      OpBranch: begin
        if (take_branch) begin
          pcNext = pcCurrent + imm_b_sext;
        end
      end

      OpLoad: begin
        rf_we = 1;
        addr_to_dmem = rs1_data + imm_i_sext;

        case (inst_funct3)
          3'b000: rd_data = {{24{load_data[7]}}, load_data[7:0]};   // lb
          3'b001: rd_data = {{16{load_data[15]}}, load_data[15:0]}; // lh
          3'b010: rd_data = load_data;                              // lw
          3'b100: rd_data = {24'b0, load_data[7:0]};                // lbu
          3'b101: rd_data = {16'b0, load_data[15:0]};               // lhu
        endcase
      end

      OpStore: begin
        addr_to_dmem = rs1_data + imm_s_sext;
        store_data_to_dmem = rs2_data << (addr_to_dmem[1:0] * 8);

        case (inst_funct3)
          3'b000: store_we_to_dmem = 4'b0001 << addr_to_dmem[1:0]; // sb
          3'b001: store_we_to_dmem = 4'b0011 << addr_to_dmem[1:0]; // sh
          3'b010: store_we_to_dmem = 4'b1111;                      // sw
        endcase
      end

      OpRegImm: begin
        rf_we = 1;
        case (inst_funct3)
          3'b000: rd_data = cla_res; // addi
          3'b010: rd_data = ($signed(rs1_data) < $signed(imm_i_sext)) ? 32'd1 : 32'd0; // slti
          3'b011: rd_data = (rs1_data < imm_i_sext) ? 32'd1 : 32'd0;                   // sltiu
          3'b100: rd_data = rs1_data ^ imm_i_sext;                                     // xori
          3'b110: rd_data = rs1_data | imm_i_sext;                                     // ori
          3'b111: rd_data = rs1_data & imm_i_sext;                                     // andi
          3'b001: rd_data = rs1_data << imm_shamt;                                     // slli
          3'b101: begin
            if (inst_funct7 == 7'd0) begin
              rd_data = rs1_data >> imm_shamt; // srli
            end else begin
              rd_data = $signed(rs1_data) >>> imm_shamt; // srai
            end
          end
        endcase
      end

      OpRegReg: begin
        rf_we = 1;
        if (inst_funct7 == 7'd1) begin
          case (inst_funct3)
            3'b000: rd_data = rs1_data * rs2_data;            // mul
            3'b001: rd_data = mul_s[63:32];                   // mulh
            3'b010: rd_data = mul_su[63:32];                  // mulhsu
            3'b011: rd_data = mul_u[63:32];                   // mulhu
            3'b100, 3'b101, 3'b110, 3'b111: rd_data = div_res; // div, divu, rem, remu
          endcase
        end else begin
          case (inst_funct3)
            3'b000: rd_data = cla_res;                                        // add, sub
            3'b001: rd_data = rs1_data << rs2_data[4:0];                      // sll
            3'b010: rd_data = ($signed(rs1_data) < $signed(rs2_data)) ? 32'd1 : 32'd0; // slt
            3'b011: rd_data = (rs1_data < rs2_data) ? 32'd1 : 32'd0;          // sltu
            3'b100: rd_data = rs1_data ^ rs2_data;                            // xor
            3'b101: begin
              if (inst_funct7[5]) rd_data = $signed(rs1_data) >>> rs2_data[4:0];        // sra
              else rd_data = rs1_data >> rs2_data[4:0];                                 // srl
            end
            3'b110: rd_data = rs1_data | rs2_data;                            // or
            3'b111: rd_data = rs1_data & rs2_data;                            // and
          endcase
        end
      end

      OpEnviron: begin
        if (inst_from_imem[31:7] == 25'd0) halt = 1'b1; // ecall
      end

      default: begin
        illegal_inst = 1'b1;
      end
    endcase
  end
endmodule

/* A memory module that supports 1-cycle reads and writes, with one
```

```
 read-only port
 * and one read+write port.
 */
module MemorySingleCycle #(
    parameter NUM_WORDS = 512
) (
  input                       rst,                 // rst for both imem and dmem
  input                       clock_mem,           // clock for both imem and dmem
  input      [`REG_SIZE:0] pc_to_imem,           // must always be aligned to a 4B boundary
  output reg [`REG_SIZE:0] inst_from_imem,       // the value at memory location pc_to_imem
  input      [`REG_SIZE:0] addr_to_dmem,         // must always be aligned to a 4B boundary
  output reg [`REG_SIZE:0] load_data_from_dmem, // the value at memory location addr_to_dmem
  input      [`REG_SIZE:0] store_data_to_dmem,  // the value to be written to addr_to_dmem, controlled by store_we_to_dmem
  // Each bit determines whether to write the corresponding byte of store_data_to_dmem to memory location addr_to_dmem.
  // E.g., 4'b1111 will write 4 bytes. 4'b0001 will write only the least-significant byte.
  input      [        3:0] store_we_to_dmem
);

  // memory is arranged as an array of 4B words
  reg [`REG_SIZE:0] mem_array[0:NUM_WORDS-1];

  // preload instructions to mem_array
  always @(posedge rst) begin
    $readmemh("mem_initial_contents.hex", mem_array);
  end

  localparam AddrMsb = $clog2(NUM_WORDS) + 1;
  localparam AddrLsb = 2;

  always @(posedge clock_mem) begin
    inst_from_imem <= mem_array[{pc_to_imem[AddrMsb:AddrLsb]}];
  end

  always @(negedge clock_mem) begin
   if (store_we_to_dmem[0]) begin
     mem_array[addr_to_dmem[AddrMsb:AddrLsb]][7:0] <= store_data_to_dmem[7:0];
   end
   if (store_we_to_dmem[1]) begin
     mem_array[addr_to_dmem[AddrMsb:AddrLsb]][15:8] <= store_data_to_dmem[15:8];
   end
   if (store_we_to_dmem[2]) begin
     mem_array[addr_to_dmem[AddrMsb:AddrLsb]][23:16] <=
        store_data_to_dmem[23:16];
   end
   if (store_we_to_dmem[3]) begin
     mem_array[addr_to_dmem[AddrMsb:AddrLsb]][31:24] <= store_data_to_dmem[31:24];
   end
   // dmem is "read-first": read returns value before the write
   load_data_from_dmem <= mem_array[{addr_to_dmem[AddrMsb:AddrLsb]}];
  end
endmodule

/*
This shows the relationship between clock_proc and clock_mem. The clock_mem is
phase-shifted 90   from clock_proc. You could think of one proc cycle being
broken down into 3 parts. During part 1 (which starts @posedge clock_proc)
the current PC is sent to the imem. In part 2 (starting @posedge clock_mem) we
read from imem. In part 3 (starting @negedge clock_mem) we read/write memory and
prepare register/PC updates, which occur at @posedge clock_proc.

        ----
 proc: |    |______
           ----
 mem:  ___|    |___
*/
module Processor (
    input  clock_proc,
    input  clock_mem,
    input  rst,
    output halt
);

  wire [`REG_SIZE:0] pc_to_imem, inst_from_imem, mem_data_addr, mem_data_loaded_value, mem_data_to_write;
  wire [        3:0] mem_data_we;

  // This wire is set by cocotb to the name of the currently-running test, to make it easier
  // to see what is going on in the waveforms.
  wire [(8*32)-1:0] test_case;

  MemorySingleCycle #(
      .NUM_WORDS(8192)
  ) memory (
    .rst                 (rst),
    .clock_mem           (clock_mem),
    // imem is read-only
    .pc_to_imem          (pc_to_imem),
    .inst_from_imem      (inst_from_imem),
    // dmem is read-write
    .addr_to_dmem         (mem_data_addr),
    .load_data_from_dmem  (mem_data_loaded_value),
    .store_data_to_dmem   (mem_data_to_write),
    .store_we_to_dmem     (mem_data_we)
  );

  DatapathSingleCycle datapath (
    .clk                  (clock_proc),
    .rst                  (rst),
    .pc_to_imem           (pc_to_imem),
    .inst_from_imem       (inst_from_imem),
    .addr_to_dmem         (mem_data_addr),
    .store_data_to_dmem   (mem_data_to_write),
    .store_we_to_dmem     (mem_data_we),
    .load_data_from_dmem  (mem_data_loaded_value),
    .halt                 (halt)
  );
endmodule
```

Program 1: RTL implementation of the DatapathSingleCycle module.

4 Testbench

4.1 RegFile testbench

This test bench is used to verify the correctness of Register File. It uses two main tasks:

- Task `wr` for simulates a synchronous write operation by driving `we`, `rd` and `rd_data` aligned with the positive clock edge.
- Task `check` to sample the read ports (`rs1_data`, `rs2_data`) on the negative clock edge. This ensures the data from the DUT is stable before comparison.

The test bench provides four distinct tests to validate the correctness of the module:

- Test 1 (x0 verification): To confirm that register x0 is read-only and hardwired to zero.
- Test 2 (Basic read/write sanity check): To confirm basic storage capability.
- Test 3 (Iterative register access): To verify address decoding and reset behavior for all registers. The test generates a random 32-bit integer, which is then written to the register and immediately verified.
- Test 4 (Write all and read back): To ensure that writing to one register does not corrupt data in others. The test filling all registers then use a loop to read them back sequentially.

```
`timescale 1ns / 1ps
module tb_reg;
reg clk, rst, we;
reg [4:0] rs1, rs2, rd;
reg [31:0] rd_data;
wire [31:0] rs1_data, rs2_data;
```

```

integer i, pass, fail;
reg [31:0] wr_val [0:31];
reg [31:0] exp_val;

RegFile dut (
    .clk (clk),
    .rst (rst),
    .we  (we),
    .rs1 (rs1),
    .rs2 (rs2),
    .rd  (rd),
    .rd_data (rd_data),
    .rs1_data (rs1_data),
    .rs2_data (rs2_data)
);

task divi;
    $display ("%0s", {80{"-"}});
endtask

task br;
    $display ("%0s", {80{"="}});
endtask

task msg (input [700:0] txt);
    begin
        br();
        $display ("%0s", txt);
        br();
    end
endtask

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

task setup;
    begin
        rst = 1;
        repeat (2) @(posedge clk);
        #1;
        rst = 0;
        we = 0;
        @(posedge clk);
    end
endtask

task wr (input [4:0] reg_num, input [31:0] val);
    begin
        divi();
        $display ("[WRITE] Time = %0t | rd  = 0x%2h, Data = 0x%2h",
              $time, reg_num, val);
        divi();

        @(posedge clk);
        #1;
        we = 1;
        rd = reg_num;
        rd_data = val;

        @(posedge clk);
        #1;
        we = 0;
    end
endtask

task check (input [4:0] reg_num, input [31:0] exp);
    begin
        rs1 = reg_num;
        @(negedge clk);
        #1;
        if (rs1_data === exp) begin
            $display ("[PASS]  Time = %0t | rs1 = 0x%2h, Expected = 0x
               %8h, Output = 0x%8h", $time, reg_num, exp, rs1_data);
            pass = pass + 1;
        end else begin
            $display ("[FAIL]  Time = %0t | rs1 = 0x%2h, Expected = 0x
               %8h, Output = 0x%8h", $time, reg_num, exp, rs1_data);
            fail = fail + 1;
        end
    end
endtask

task check_all (input [4:0] reg1, input [31:0] exp1, input [4:0] reg2,
    input [31:0] exp2);
    begin
        rs1 = reg1;
        rs2 = reg2;
        @(negedge clk);
        #1;

        if (rs1_data === exp1) begin
            $display ("[PASS]  Time = %0t | rs1 = 0x%2h, Expected = 0x
               %8h, Output = 0x%8h", $time, reg1, exp1, rs1_data);
            pass = pass + 1;
        end else begin
            $display ("[FAIL]  Time = %0t | rs1 = 0x%2h, Expected = 0x
               %8h, Output = 0x%8h", $time, reg1, exp1, rs1_data);
            fail = fail + 1;
        end

        if (rs2_data === exp2) begin
            $display ("[PASS]  Time = %0t | rs2 = 0x%2h, Expected = 0x
               %8h, Output = 0x%8h", $time, reg2, exp2, rs2_data);
            pass = pass + 1;
        end else begin
            $display ("[FAIL]  Time = %0t | rs2 = 0x%2h, Expected = 0x
                %8h, Output = 0x%8h", $time, reg2, exp2, rs2_data);
            fail = fail + 1;
        end
    end
endtask

task init (input [4:0] reg1, input [4:0] reg2, input [31:0] exp);
    begin
        @(negedge clk);
        #1;
        if (rs1_data != 0 || rs2_data != 0) begin
            $display ("[FAIL]  Time = %0t | rs1 = 0x%2h and rs2 = 0x%2
                h should be initialized to 0", $time, reg1, reg2);
            fail = fail + 1;
        end else begin
            $display ("[PASS]  Time = %0t | rs1 = 0x%2h and rs2 = 0x%2
                h initialized to 0", $time, reg1, reg2);
            pass = pass + 1;
        end
    end
endtask

initial begin
    pass = 0;
    fail = 0;

    msg ("TEST 1: WRITE AND READ x0");
    setup();
    check (5'd0, 32'd0);

    wr (5'd0, 32'hFFFF_FFFF);
    check (5'd0, 32'd0);

    wr (5'd0, 32'h1234_5678);
    check (5'd0, 32'd0);

    msg ("TEST 2: WRITE AND READ x1");
    setup();
    wr (5'd1, 32'h1234_5678);
    check (5'd1, 32'h1234_5678);


    msg ("TEST 3: CHECK ALL REGISTERS");
    setup();
    for (i = 1; i < 32; i = i + 1) begin
        rs1 = i[4:0];
        rs2 = i[4:0];
        init (rs1, rs2, 32'd0);

        exp_val = $urandom();
```

```
            wr (i[4:0], exp_val);
            check_all (i[4:0], exp_val, i[4:0], exp_val);
        end

        msg ("TEST 4: WRITE ALL AND READ BACK");
        setup();
        wr_val[0] = 32'd0; // x0

        for (i = 1; i < 32; i = i + 1) begin
            wr_val[i] = $urandom();
            wr (i[4:0], wr_val[i]);
        end

        for (i = 1; i < 31; i = i + 1) begin
            check_all (i[4:0], wr_val[i], i + 1, wr_val[i + 1]);
        end

        msg ("SUMMARY");
        $display ("TOTAL TESTS : %0d", pass + fail);
        $display ("TOTAL PASSED: %0d", pass);
        $display ("TOTAL FAILED: %0d", fail);

        if (fail == 0)
            $display ("======> ALL TESTS PASSED");
        else if (pass == 0)
            $display ("======> ALL TESTS FAILED");
        else
            $display ("======> SOME TESTS FAILED");

        #100;
        $finish;
    end
    endmodule
```

Program 2: Test bench implementation for the RegFile module.

4.2 Processor testbench

The `tb_single` module serves as the top level integration test, validating the interaction between the Datapath, Control unit and Memory. It executes actual RISC-V machine code to ensure the processor complies with the ISA specification.
The testbench generates two clock signals to simulate the timing constraints of a single-cycle architecture:

- clock_proc: The main system clock. Register writes and PC updates occur on the rising edge.
- clock_mem: A phase-shifted 90 degrees from clock_proc clock.

The testbench uses the `load_inst` task to write 32-bit machine code directly into the `mem_array` of the instantiated Memory module. The task `check` uses hierarchical referencing to peek directly into the internal register file in order to validating a CPU requires checking the state of internal general-purpose registers (x1-x31). Since the registers are limited by 32 with one is hardwired to zero values, there is only 31 registers that can store data. Therefore, the testbench breaks into two tests, Test 1 focuses on ALU operations such as ADD, XOR, SRAI; multiplication and division; use LW and SW to verify the load/store path and executes a JAL to skip instructions, verifying the PC updates non-sequentially. Meanwhile, Test 2 focuses on implement signed and unsigned operations like MULH, MULHU; sub-word access that stores data using SB and reads back using LB and LBU to verifying that the memory masking logic correctly handles sign-extension for 8 and 16-bit values; and a JALR instruction is used to jump to an address calculated at runtime, verifying the register-based jump logic.

```
`timescale 1ns / 1ps
module tb_single;
reg clock_proc, clock_mem, rst;
wire halt;
integer i, test, timeout, pass, fail;

Processor dut (
    .clock_proc (clock_proc),
    .clock_mem  (clock_mem),
    .rst        (rst),
    .halt       (halt)
);

task br;
    $display ("%0s", {80{"="}});
endtask

task msg (input [700:0] txt);
    begin
        br();
        $display ("%0s", txt);
        br();
    end
endtask

task load_inst (input [31:0] addr, input [31:0] inst);
    begin
        dut.memory.mem_array[addr >> 2] = inst;
    end
endtask

task check (input [200:0] txt, input [4:0] reg_num, input [31:0] exp);
    begin
        if (dut.datapath.rf.regs[reg_num] === exp) begin
            $display ("[PASS] Time = %0t | %s | Output x%0d = 0x%8h",
               $time, txt, reg_num, dut.datapath.rf.regs[reg_num]);
            pass = pass + 1;
        end else begin
            $display ("[FAIL] Time = %0t | %s | Expect x%0d = 0x%8h |
               Output x%0d = 0x%8h", $time, txt, reg_num, exp, reg_num
               , dut.datapath.rf.regs[reg_num]);
            fail = fail + 1;
        end
    end
endtask
```

```

initial begin
    clock_proc = 0;
    forever #5 clock_proc = ~clock_proc;
end

initial begin
    #2.5 clock_mem = 0;
    forever #5 clock_mem = ~clock_mem;
end

initial begin
    msg ("TEST BEGIN");
    rst = 1;
    timeout = 0;
    pass = 0;
    fail = 0;
    test = 0;
    #20;

    msg ("TEST 1");
    load_inst (32'h00, 32'h00a00093); // ADDI x1, x0, 10
    load_inst (32'h04, 32'hffb00113); // ADDI x2, x0, -5
    load_inst (32'h08, 32'h002081b3); // ADD x3, x1, x2 (5)
    load_inst (32'h0C, 32'h40208233); // SUB x4, x1, x2 (15)
    load_inst (32'h10, 32'h0020f2b3); // AND x5, x1, x2 (10)
    load_inst (32'h14, 32'h0020e333); // OR x6, x1, x2 (-5)
    load_inst (32'h18, 32'h0020c3b3); // XOR x7, x1, x2 (-15)
    load_inst (32'h1C, 32'h00209413); // SLLI x8, x1, 2 (40)
    load_inst (32'h20, 32'h40115493); // SRAI x9, x2, 1 (-3)
    load_inst (32'h24, 32'h00115513); // SRLI x10, x2, 1
    load_inst (32'h28, 32'h001125b3); // SLT x11, x2, x1 (1)
    load_inst (32'h2C, 32'h00113633); // SLTU x12, x2, x1 (0)
    load_inst (32'h30, 32'h021086b3); // MUL x13, x1, x1 (100)
    load_inst (32'h34, 32'h0220c733); // DIV x14, x1, x2 (-2)
    load_inst (32'h38, 32'h0220e7b3); // REM x15, x1, x2 (0)
    load_inst (32'h3C, 32'h02115833); // DIVU x16, x2, x1
    load_inst (32'h40, 32'h06302223); // SW x3, 100(x0)
    load_inst (32'h44, 32'h06402903); // LW x18, 100(x0)
    load_inst (32'h48, 32'h00108463); // BEQ x1, x1, 8
    load_inst (32'h4C, 32'h06300993); // (Skipped)
    load_inst (32'h50, 32'h00100993); // ADDI x19, x0, 1
    load_inst (32'h54, 32'h00800a6f); // JAL x20, 8
    load_inst (32'h58, 32'h06300a93); // (Skipped)
    load_inst (32'h5C, 32'h00200a93); // ADDI x21, x0, 2
    load_inst (32'h60, 32'h12345b37); // LUI x22, 0x12345
    load_inst(32'h68, 32'h0050cb93);  // XORI x23, x1, 5
    load_inst (32'h6C, 32'h00f0fc13); // ANDI x24, x1, 0xF
    load_inst (32'h70, 32'hf006ec93); // ORI x25, x13, 0xFF00
    load_inst (32'h74, 32'h0283ad13); // SLTI x26, x7, 0x28
    load_inst (32'h78, 32'hff143d93); // SLTIU x27, x8, 0xfffffff1
    load_inst (32'h7C, 32'h01309e33); // SLL x28, x1, x19
    load_inst (32'h80, 32'h01335eb3); // SRL x29, x6, x19
    load_inst (32'h84, 32'h41335f33); // SRA x30, x6, x19
    load_inst (32'h8C, 32'h02337fb3); // REMU x31, x6, x3
    load_inst(32'h90, 32'h00000073); // HALT

    #10 rst = 0;
    #1;

    while (!halt && timeout < 1000) begin
        #10 timeout = timeout + 10;
    end

    if (test == 0) begin
        check ("ADDI x1, x0, 10", 1, 10);
        check ("ADDI x2, x0, -5", 2, -5);
        check ("ADD x3, x1, x2", 3, 5);
        check ("SUB x4, x1, x2", 4, 15);
        check ("AND x5, x1, x2", 5, 10);
        check ("OR x6, x1, x2", 6, -5);
        check ("XOR x7, x1, x2", 7, -15);
        check ("SLLI x8, x1, 2", 8, 40);
        check ("SRAI x9, x2, 1", 9, -3);
        check ("SRLI x10, x2, 1", 10, 32'h7fff_fffd);
        check ("SLT x11, x2, x1", 11, 1);
        check ("SLTU x12, x2, x1", 12, 0);
        check ("MUL x13, x1, x1", 13, 100);
        check ("DIV x14, x1, x2", 14, -2);
        check ("REM x15, x1, x2", 15, 0);
        check ("DIVU x16, x2, x1", 16, 32'h1999_9999);
        check ("LW x18, 100(x0)", 18, 5);
        check ("ADDI x19, x0, 1", 19, 1);
        check ("JAL x20, 8", 20, 32'h58); // PC of next instruction
        check ("ADDI x21, x0, 2", 21, 2);
        check ("LUI x22, 0x12345", 22, 32'h1234_5000);
        check ("XORI x23, x1, 5", 23, 32'hf);
        check ("ANDI x24, x1, 0xF", 24, 32'ha);
        check ("ORI x25, x13, 0xFF00", 25, 32'hffff_ff64);
        check ("SLTI x26, x7, 0x28", 26, 1);
        check ("SLTIU x27, x8, 0xFFFF_FFF1", 27, 1);
        check ("SLL x28, x1, x19", 28, 20);
        check ("SRL x29, x6, x19", 29, 32'h7FFF_FFFD);
        check ("SRA x30, x6, x19", 30, 32'hFFFF_FFFD);
        check ("REMU x31, x6, x3", 31, 1);

        if (halt) begin
            $display ("[PASS] HALT asserted.");
            pass = pass + 1;
        end else begin
            $display ("[FAIL] HALT is missing.");
            fail = fail + 1;
        end
    end

    msg ("TEST 2 (REUSED REGISTERS)");
    @(negedge clock_proc);
    rst = 1;
    test = 1;
    timeout = 0;
    #20;

    load_inst (32'h0, 32'hffb00093); // ADDI x1, x0, 0xFFFF_FFFB
    load_inst (32'h4, 32'h00300113); // ADDI x2, x0, 3
    load_inst (32'h8, 32'h022091b3); // MULH x3, x1, x2
    load_inst (32'hC, 32'h0220a233); // MULHSU x4, x1, x2
    load_inst (32'h10, 32'h0220b2b3); // MULHU x5, x1, x2
    load_inst (32'h14, 32'he9800313); // ADDI x6, x0, 0xABCD_FE98
    load_inst (32'h18, 32'h06600223); // SB x6, 100(x0)
    load_inst (32'h1C, 32'h06601423); // SH x6, 104(x0)
    load_inst (32'h20, 32'h06400383); // LB x7, 100(x0)
    load_inst (32'h24, 32'h06801403); // LH x8, 104(x0)
    load_inst (32'h28, 32'h00831463); // BNE x6, x8, 8
    load_inst (32'h2C, 32'h06904483); // LBU x9, 105(x0)
    load_inst (32'h30, 32'h06805503); // LHU x10, 104(x0)
    load_inst (32'h34, 32'h00a4c463); // BLT x9, x10, 12
    load_inst (32'h38, 32'h00a4c5b3); // XOR x11, x9, x10 //skipped
    load_inst (32'h3C, 32'h00a4e5b3); // OR x11, x9, x10 // skipped
    load_inst (32'h40, 32'h00a485b3); // ADD x11, x9, x10
    load_inst (32'h44, 32'h00535463); // BGE x6, x5, 8
    load_inst (32'h48, 32'h0062e663); // BLTU x5, x6, 12
    load_inst (32'h4C, 32'h00a4e633); // OR x12, x9, x10 // skip then
       turned back by bgeu
    load_inst (32'h50, 32'h08200767); // JALR x14, 130(x0)
    load_inst (32'h54, 32'h0004e633); // OR x12, x9, x0
    load_inst (32'h58, 32'h06c02c23); // SW x12, 120(x0)
    load_inst (32'h5C, 32'h07802683); // LW x13, 120(x0)
    load_inst (32'h60, 32'hfe9576e3); // BGEU x10, x9, -20
    load_inst (32'h82, 32'h00000073); // HALT

    #10 rst = 0;
    #1;
    while (!halt && timeout < 1000) begin
        #10 timeout = timeout + 10;
    end

    if (test) begin
        check ("ADDI x1, x0, 0xFFFF_FFFB", 1, 32'hFFFF_FFFB);
        check ("ADDI x2, x0, 3", 2, 32'h3);
        check ("MULH x3, x1, x2", 3, 32'hFFFF_FFFF);
        check ("MULHSU x4, x1, x2", 4, 32'hFFFF_FFFF);
        check ("MULHU x5, x1, x2", 5, 32'h2);
        check ("ADDI x6, x0, 0xABCD_FE98", 6, 32'hFFFF_FE98);
        check ("LB x7, 100(x0)", 7, 32'hFFFF_FF98);
        check ("LH x8, 104(x0)", 8, 32'hFFFF_FE98);
        check ("LBU x9, 105(x0)", 9, 32'h0000_00FE);
        check ("LHU x10, 104(x0)", 10, 32'h0000_FE98);
        check ("ADD x11, x9, x10", 11, 32'h0000_FF96);
        check ("OR x12, x9, x10", 12, 32'h0000_FEFE);
        check ("LW x13, 120(x0)", 13, 32'h0000_00FE);
        check ("JALR x14, 130(x0)", 14, 32'h54);

        if (halt) begin
            $display ("[PASS] HALT asserted.");
            pass = pass + 1;
        end else begin
            $display ("[FAIL] HALT is missing.");
            fail = fail + 1;
        end
    end

    msg ("SUMMARY");
    $display ("TOTAL TESTS : %0d", pass + fail);
    $display ("TOTAL PASSED: %0d", pass);
    $display ("TOTAL FAILED: %0d", fail);

    if (fail == 0)
        $display ("======> ALL TESTS PASSED");
    else if (pass == 0)
        $display ("======> ALL TESTS FAILED");
    else
        $display ("======> SOME TESTS FAILED");

    msg ("TEST END");

    #100;
    $finish;
end
endmodule
```

Program 3: Test bench implementation for the Processor module.

5 Testbench results

All test cases executed successfully for all tested modules, with each marked as `PASSED`. This outcome confirms that the test modules behave as intended and that the implemented RTL designs meet the expected functionality.

5.1 RegFile testbench results

```
# ================================================================
# TEST 1: WRITE AND READ x0
# ================================================================
# [PASS]  Time = 31000 | rs1 = 0x00, Expected = 0x00000000, Output = 0
   x00000000
# ----------------------------------------------------------------
# [WRITE] Time = 31000 | rd  = 0x00, Data = 0xffffffff
# ----------------------------------------------------------------
# [PASS]  Time = 51000 | rs1 = 0x00, Expected = 0x00000000, Output = 0
   x00000000
# ----------------------------------------------------------------
# [WRITE] Time = 51000 | rd  = 0x00, Data = 0x12345678
# ----------------------------------------------------------------
# [PASS]  Time = 71000 | rs1 = 0x00, Expected = 0x00000000, Output = 0
    x00000000
# ================================================================
# TEST 2: WRITE AND READ x1
# ================================================================
# ----------------------------------------------------------------
# [WRITE] Time = 95000 | rd  = 0x01, Data = 0x12345678
# ----------------------------------------------------------------
# [PASS]  Time = 121000 | rs1 = 0x01, Expected = 0x12345678, Output =
    0x12345678
# ================================================================
# TEST 3: CHECK ALL REGISTERS
# ================================================================
# [PASS]  Time = 151000 | rs1 = 0x01 and rs2 = 0x01 initialized to 0
# ----------------------------------------------------------------
# [WRITE] Time = 151000 | rd  = 0x01, Data = 0x369d4b49
# ----------------------------------------------------------------
# [PASS]  Time = 171000 | rs1 = 0x01, Expected = 0x369d4b49, Output =
    0x369d4b49
# [PASS]  Time = 171000 | rs2 = 0x01, Expected = 0x369d4b49, Output =
    0x369d4b49
# [PASS]  Time = 181000 | rs1 = 0x02 and rs2 = 0x02 initialized to 0
# ----------------------------------------------------------------
# [WRITE] Time = 181000 | rd  = 0x02, Data = 0x2e53207a
# ----------------------------------------------------------------
# [PASS]  Time = 201000 | rs1 = 0x02, Expected = 0x2e53207a, Output =
    0x2e53207a
# [PASS]  Time = 201000 | rs2 = 0x02, Expected = 0x2e53207a, Output =
    0x2e53207a
# [PASS]  Time = 211000 | rs1 = 0x03 and rs2 = 0x03 initialized to 0
# ----------------------------------------------------------------
# [WRITE] Time = 211000 | rd  = 0x03, Data = 0x7cb0da7e
# ----------------------------------------------------------------
# [PASS]  Time = 231000 | rs1 = 0x03, Expected = 0x7cb0da7e, Output =
    0x7cb0da7e
# [PASS]  Time = 231000 | rs2 = 0x03, Expected = 0x7cb0da7e, Output =
    0x7cb0da7e
# [PASS]  Time = 241000 | rs1 = 0x04 and rs2 = 0x04 initialized to 0
# ----------------------------------------------------------------
# [WRITE] Time = 241000 | rd  = 0x04, Data = 0x6a8defda
# ----------------------------------------------------------------
# [PASS]  Time = 261000 | rs1 = 0x04, Expected = 0x6a8defda, Output =
    0x6a8defda
# [PASS]  Time = 261000 | rs2 = 0x04, Expected = 0x6a8defda, Output =
    0x6a8defda
# [PASS]  Time = 271000 | rs1 = 0x05 and rs2 = 0x05 initialized to 0
# ----------------------------------------------------------------
# [WRITE] Time = 271000 | rd  = 0x05, Data = 0xc7119e4e
# ----------------------------------------------------------------
# [PASS]  Time = 291000 | rs1 = 0x05, Expected = 0xc7119e4e, Output =
    0xc7119e4e
```

```
# [PASS]  Time = 291000 | rs2 = 0x05, Expected = 0xc7119e4e, Output =
   0xc7119e4e
# [PASS]  Time = 301000 | rs1 = 0x06 and rs2 = 0x06 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 301000 | rd  = 0x06, Data = 0xf0b39a02
# ------------------------------------------------------------------
# [PASS]  Time = 321000 | rs1 = 0x06, Expected = 0xf0b39a02, Output =
   0xf0b39a02
# [PASS]  Time = 321000 | rs2 = 0x06, Expected = 0xf0b39a02, Output =
   0xf0b39a02
# [PASS]  Time = 331000 | rs1 = 0x07 and rs2 = 0x07 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 331000 | rd  = 0x07, Data = 0xb81d179b
# ------------------------------------------------------------------
# [PASS]  Time = 351000 | rs1 = 0x07, Expected = 0xb81d179b, Output =
   0xb81d179b
# [PASS]  Time = 351000 | rs2 = 0x07, Expected = 0xb81d179b, Output =
   0xb81d179b
# [PASS]  Time = 361000 | rs1 = 0x08 and rs2 = 0x08 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 361000 | rd  = 0x08, Data = 0xcaf481d9
# ------------------------------------------------------------------
# [PASS]  Time = 381000 | rs1 = 0x08, Expected = 0xcaf481d9, Output =
   0xcaf481d9
# [PASS]  Time = 381000 | rs2 = 0x08, Expected = 0xcaf481d9, Output =
   0xcaf481d9
# [PASS]  Time = 391000 | rs1 = 0x09 and rs2 = 0x09 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 391000 | rd  = 0x09, Data = 0x4aae874c
# ------------------------------------------------------------------
# [PASS]  Time = 411000 | rs1 = 0x09, Expected = 0x4aae874c, Output =
   0x4aae874c
# [PASS]  Time = 411000 | rs2 = 0x09, Expected = 0x4aae874c, Output =
   0x4aae874c
# [PASS]  Time = 421000 | rs1 = 0x0a and rs2 = 0x0a initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 421000 | rd  = 0x0a, Data = 0xd0bc7957
# ------------------------------------------------------------------
# [PASS]  Time = 441000 | rs1 = 0x0a, Expected = 0xd0bc7957, Output =
   0xd0bc7957
# [PASS]  Time = 441000 | rs2 = 0x0a, Expected = 0xd0bc7957, Output =
   0xd0bc7957
# [PASS]  Time = 451000 | rs1 = 0x0b and rs2 = 0x0b initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 451000 | rd  = 0x0b, Data = 0x6aa8df9b
# ------------------------------------------------------------------
# [PASS]  Time = 471000 | rs1 = 0x0b, Expected = 0x6aa8df9b, Output =
   0x6aa8df9b
# [PASS]  Time = 471000 | rs2 = 0x0b, Expected = 0x6aa8df9b, Output =
   0x6aa8df9b
# [PASS]  Time = 481000 | rs1 = 0x0c and rs2 = 0x0c initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 481000 | rd  = 0x0c, Data = 0xfd6562cf
# -------------------------------------------------------------------
# [PASS]  Time = 501000 | rs1 = 0x0c, Expected = 0xfd6562cf, Output =
   0xfd6562cf
# [PASS]  Time = 501000 | rs2 = 0x0c, Expected = 0xfd6562cf, Output =
   0xfd6562cf
# [PASS]  Time = 511000 | rs1 = 0x0d and rs2 = 0x0d initialized to 0
# -------------------------------------------------------------------
# [WRITE] Time = 511000 | rd  = 0x0d, Data = 0x25568ff3
# -------------------------------------------------------------------
# [PASS]  Time = 531000 | rs1 = 0x0d, Expected = 0x25568ff3, Output =
   0x25568ff3
# [PASS]  Time = 531000 | rs2 = 0x0d, Expected = 0x25568ff3, Output =
   0x25568ff3
# [PASS]  Time = 541000 | rs1 = 0x0e and rs2 = 0x0e initialized to 0
# -------------------------------------------------------------------
# [WRITE] Time = 541000 | rd  = 0x0e, Data = 0x66fa2046
# -------------------------------------------------------------------
# [PASS]  Time = 561000 | rs1 = 0x0e, Expected = 0x66fa2046, Output =
   0x66fa2046
# [PASS]  Time = 561000 | rs2 = 0x0e, Expected = 0x66fa2046, Output =
   0x66fa2046
# [PASS]  Time = 571000 | rs1 = 0x0f and rs2 = 0x0f initialized to 0
# -------------------------------------------------------------------
# [WRITE] Time = 571000 | rd  = 0x0f, Data = 0x41766910
# -------------------------------------------------------------------
# [PASS]  Time = 591000 | rs1 = 0x0f, Expected = 0x41766910, Output =
   0x41766910
# [PASS]  Time = 591000 | rs2 = 0x0f, Expected = 0x41766910, Output =
   0x41766910
# [PASS]  Time = 601000 | rs1 = 0x10 and rs2 = 0x10 initialized to 0
# -------------------------------------------------------------------
# [WRITE] Time = 601000 | rd  = 0x10, Data = 0x81103480
# -------------------------------------------------------------------
# [PASS]  Time = 621000 | rs1 = 0x10, Expected = 0x81103480, Output =
   0x81103480
# [PASS]  Time = 621000 | rs2 = 0x10, Expected = 0x81103480, Output =
   0x81103480
# [PASS]  Time = 631000 | rs1 = 0x11 and rs2 = 0x11 initialized to 0
# -------------------------------------------------------------------
# [WRITE] Time = 631000 | rd  = 0x11, Data = 0x2942e4d8
# -------------------------------------------------------------------
# [PASS]  Time = 651000 | rs1 = 0x11, Expected = 0x2942e4d8, Output =
   0x2942e4d8
# [PASS]  Time = 651000 | rs2 = 0x11, Expected = 0x2942e4d8, Output =
   0x2942e4d8
# [PASS]  Time = 661000 | rs1 = 0x12 and rs2 = 0x12 initialized to 0
# -------------------------------------------------------------------
# [WRITE] Time = 661000 | rd  = 0x12, Data = 0x546a052c
# -------------------------------------------------------------------
# [PASS]  Time = 681000 | rs1 = 0x12, Expected = 0x546a052c, Output =
   0x546a052c
# [PASS]  Time = 681000 | rs2 = 0x12, Expected = 0x546a052c, Output =
   0x546a052c
```

```
# [PASS]  Time = 691000 | rs1 = 0x13 and rs2 = 0x13 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 691000 | rd  = 0x13, Data = 0xe34c5b88
# ------------------------------------------------------------------
# [PASS]  Time = 711000 | rs1 = 0x13, Expected = 0xe34c5b88, Output =
   0xe34c5b88
# [PASS]  Time = 711000 | rs2 = 0x13, Expected = 0xe34c5b88, Output =
   0xe34c5b88
# [PASS]  Time = 721000 | rs1 = 0x14 and rs2 = 0x14 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 721000 | rd  = 0x14, Data = 0x3635d07
# ------------------------------------------------------------------
# [PASS]  Time = 741000 | rs1 = 0x14, Expected = 0x03635d07, Output =
   0x03635d07
# [PASS]  Time = 741000 | rs2 = 0x14, Expected = 0x03635d07, Output =
   0x03635d07
# [PASS]  Time = 751000 | rs1 = 0x15 and rs2 = 0x15 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 751000 | rd  = 0x15, Data = 0xc691eee3
# ------------------------------------------------------------------
# [PASS]  Time = 771000 | rs1 = 0x15, Expected = 0xc691eee3, Output =
   0xc691eee3
# [PASS]  Time = 771000 | rs2 = 0x15, Expected = 0xc691eee3, Output =
   0xc691eee3
# [PASS]  Time = 781000 | rs1 = 0x16 and rs2 = 0x16 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 781000 | rd  = 0x16, Data = 0x6e6773f3
# ------------------------------------------------------------------
# [PASS]  Time = 801000 | rs1 = 0x16, Expected = 0x6e6773f3, Output =
   0x6e6773f3
# [PASS]  Time = 801000 | rs2 = 0x16, Expected = 0x6e6773f3, Output =
   0x6e6773f3
# [PASS]  Time = 811000 | rs1 = 0x17 and rs2 = 0x17 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 811000 | rd  = 0x17, Data = 0xd0cd0050
# ------------------------------------------------------------------
# [PASS]  Time = 831000 | rs1 = 0x17, Expected = 0xd0cd0050, Output =
   0xd0cd0050
# [PASS]  Time = 831000 | rs2 = 0x17, Expected = 0xd0cd0050, Output =
   0xd0cd0050
# [PASS]  Time = 841000 | rs1 = 0x18 and rs2 = 0x18 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 841000 | rd  = 0x18, Data = 0x43c3b3e2
# ------------------------------------------------------------------
# [PASS]  Time = 861000 | rs1 = 0x18, Expected = 0x43c3b3e2, Output =
   0x43c3b3e2
# [PASS]  Time = 861000 | rs2 = 0x18, Expected = 0x43c3b3e2, Output =
   0x43c3b3e2
# [PASS]  Time = 871000 | rs1 = 0x19 and rs2 = 0x19 initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 871000 | rd  = 0x19, Data = 0x77622873
# ------------------------------------------------------------------
# [PASS]  Time = 891000 | rs1 = 0x19, Expected = 0x77622873, Output =
  0x77622873
# [PASS]  Time = 891000 | rs2 = 0x19, Expected = 0x77622873, Output =
  0x77622873
# [PASS]  Time = 901000 | rs1 = 0x1a and rs2 = 0x1a initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 901000 | rd  = 0x1a, Data = 0x9ef912bb
# ------------------------------------------------------------------
# [PASS]  Time = 921000 | rs1 = 0x1a, Expected = 0x9ef912bb, Output =
  0x9ef912bb
# [PASS]  Time = 921000 | rs2 = 0x1a, Expected = 0x9ef912bb, Output =
  0x9ef912bb
# [PASS]  Time = 931000 | rs1 = 0x1b and rs2 = 0x1b initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 931000 | rd  = 0x1b, Data = 0x34c281fc
# ------------------------------------------------------------------
# [PASS]  Time = 951000 | rs1 = 0x1b, Expected = 0x34c281fc, Output =
  0x34c281fc
# [PASS]  Time = 951000 | rs2 = 0x1b, Expected = 0x34c281fc, Output =
  0x34c281fc
# [PASS]  Time = 961000 | rs1 = 0x1c and rs2 = 0x1c initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 961000 | rd  = 0x1c, Data = 0xe659f7ae
# ------------------------------------------------------------------
# [PASS]  Time = 981000 | rs1 = 0x1c, Expected = 0xe659f7ae, Output =
  0xe659f7ae
# [PASS]  Time = 981000 | rs2 = 0x1c, Expected = 0xe659f7ae, Output =
  0xe659f7ae
# [PASS]  Time = 991000 | rs1 = 0x1d and rs2 = 0x1d initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 991000 | rd  = 0x1d, Data = 0x713c11cf
# ------------------------------------------------------------------
# [PASS]  Time = 1011000 | rs1 = 0x1d, Expected = 0x713c11cf, Output =
   0x713c11cf
# [PASS]  Time = 1011000 | rs2 = 0x1d, Expected = 0x713c11cf, Output =
   0x713c11cf
# [PASS]  Time = 1021000 | rs1 = 0x1e and rs2 = 0x1e initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 1021000 | rd  = 0x1e, Data = 0x87401cb6
# ------------------------------------------------------------------
# [PASS]  Time = 1041000 | rs1 = 0x1e, Expected = 0x87401cb6, Output =
   0x87401cb6
# [PASS]  Time = 1041000 | rs2 = 0x1e, Expected = 0x87401cb6, Output =
   0x87401cb6
# [PASS]  Time = 1051000 | rs1 = 0x1f and rs2 = 0x1f initialized to 0
# ------------------------------------------------------------------
# [WRITE] Time = 1051000 | rd  = 0x1f, Data = 0xa24d13c7
# ------------------------------------------------------------------
# [PASS]  Time = 1071000 | rs1 = 0x1f, Expected = 0xa24d13c7, Output =
   0xa24d13c7
# [PASS]  Time = 1071000 | rs2 = 0x1f, Expected = 0xa24d13c7, Output =
   0xa24d13c7
# ==================================================================
# TEST 4: WRITE ALL AND READ BACK
# ==================================================================
# ------------------------------------------------------------------
# [WRITE] Time = 1095000 | rd  = 0x01, Data = 0x4c8e7537
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1116000 | rd  = 0x02, Data = 0xc969e31a
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1136000 | rd  = 0x03, Data = 0xd837cd88
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1156000 | rd  = 0x04, Data = 0x9f28cd7b
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1176000 | rd  = 0x05, Data = 0x87f27d84
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1196000 | rd  = 0x06, Data = 0x9162bd0c
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1216000 | rd  = 0x07, Data = 0x613fb4ba
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1236000 | rd  = 0x08, Data = 0xa75a26b9
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1256000 | rd  = 0x09, Data = 0x97befd48
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1276000 | rd  = 0x0a, Data = 0x74871bf9
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1296000 | rd  = 0x0b, Data = 0x9b731590
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1316000 | rd  = 0x0c, Data = 0x22e8e590
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1336000 | rd  = 0x0d, Data = 0xd66099fd
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1356000 | rd  = 0x0e, Data = 0xba542fdb
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1376000 | rd  = 0x0f, Data = 0x8bb8376e
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1396000 | rd  = 0x10, Data = 0x1603d7b9
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1416000 | rd  = 0x11, Data = 0x781c5ba3
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1436000 | rd  = 0x12, Data = 0x2381cb4e
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1456000 | rd  = 0x13, Data = 0x671e6b6f
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1476000 | rd  = 0x14, Data = 0x2ebaab9a
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1496000 | rd  = 0x15, Data = 0xc3391889
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1516000 | rd  = 0x16, Data = 0xcfe6f7b5
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1536000 | rd  = 0x17, Data = 0x6be10d7b
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1556000 | rd  = 0x18, Data = 0x720de158
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1576000 | rd  = 0x19, Data = 0xdcdd928e
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1596000 | rd  = 0x1a, Data = 0x327638a8
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1616000 | rd  = 0x1b, Data = 0xdb062489
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1636000 | rd  = 0x1c, Data = 0x879f10ac
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1656000 | rd  = 0x1d, Data = 0x38674ea5
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1676000 | rd  = 0x1e, Data = 0x5a2a5c21
# ------------------------------------------------------------------
# ------------------------------------------------------------------
# [WRITE] Time = 1696000 | rd  = 0x1f, Data = 0xd5f6a9d
# ------------------------------------------------------------------
# [PASS]  Time = 1721000 | rs1 = 0x01, Expected = 0x4c8e7537, Output = 0x4c8e7537
# [PASS]  Time = 1721000 | rs2 = 0x02, Expected = 0xc969e31a, Output = 0xc969e31a
# [PASS]  Time = 1731000 | rs1 = 0x02, Expected = 0xc969e31a, Output = 0xc969e31a
# [PASS]  Time = 1731000 | rs2 = 0x03, Expected = 0xd837cd88, Output = 0xd837cd88
# [PASS]  Time = 1741000 | rs1 = 0x03, Expected = 0xd837cd88, Output = 0xd837cd88
# [PASS]  Time = 1741000 | rs2 = 0x04, Expected = 0x9f28cd7b, Output =
    0x9f28cd7b
# [PASS]  Time = 1751000 | rs1 = 0x04, Expected = 0x9f28cd7b, Output =
    0x9f28cd7b
# [PASS]  Time = 1751000 | rs2 = 0x05, Expected = 0x87f27d84, Output =
    0x87f27d84
# [PASS]  Time = 1761000 | rs1 = 0x05, Expected = 0x87f27d84, Output =
    0x87f27d84
# [PASS]  Time = 1761000 | rs2 = 0x06, Expected = 0x9162bd0c, Output =
    0x9162bd0c
# [PASS]  Time = 1771000 | rs1 = 0x06, Expected = 0x9162bd0c, Output =
    0x9162bd0c
# [PASS]  Time = 1771000 | rs2 = 0x07, Expected = 0x613fb4ba, Output =
    0x613fb4ba
# [PASS]  Time = 1781000 | rs1 = 0x07, Expected = 0x613fb4ba, Output =
    0x613fb4ba
# [PASS]  Time = 1781000 | rs2 = 0x08, Expected = 0xa75a26b9, Output =
    0xa75a26b9
# [PASS]  Time = 1791000 | rs1 = 0x08, Expected = 0xa75a26b9, Output =
    0xa75a26b9
# [PASS]  Time = 1791000 | rs2 = 0x09, Expected = 0x97befd48, Output =
    0x97befd48
# [PASS]  Time = 1801000 | rs1 = 0x09, Expected = 0x97befd48, Output =
    0x97befd48
# [PASS]  Time = 1801000 | rs2 = 0x0a, Expected = 0x74871bf9, Output =
    0x74871bf9
# [PASS]  Time = 1811000 | rs1 = 0x0a, Expected = 0x74871bf9, Output =
    0x74871bf9
# [PASS]  Time = 1811000 | rs2 = 0x0b, Expected = 0x9b731590, Output =
    0x9b731590
# [PASS]  Time = 1821000 | rs1 = 0x0b, Expected = 0x9b731590, Output =
    0x9b731590
# [PASS]  Time = 1821000 | rs2 = 0x0c, Expected = 0x22e8e590, Output =
    0x22e8e590
# [PASS]  Time = 1831000 | rs1 = 0x0c, Expected = 0x22e8e590, Output =
    0x22e8e590
# [PASS]  Time = 1831000 | rs2 = 0x0d, Expected = 0xd66099fd, Output =
    0xd66099fd
# [PASS]  Time = 1841000 | rs1 = 0x0d, Expected = 0xd66099fd, Output =
    0xd66099fd
# [PASS]  Time = 1841000 | rs2 = 0x0e, Expected = 0xba542fdb, Output =
    0xba542fdb
# [PASS]  Time = 1851000 | rs1 = 0x0e, Expected = 0xba542fdb, Output =
    0xba542fdb
# [PASS]  Time = 1851000 | rs2 = 0x0f, Expected = 0x8bb8376e, Output =
    0x8bb8376e
# [PASS]  Time = 1861000 | rs1 = 0x0f, Expected = 0x8bb8376e, Output =
    0x8bb8376e
# [PASS]  Time = 1861000 | rs2 = 0x10, Expected = 0x1603d7b9, Output =
    0x1603d7b9
# [PASS]  Time = 1871000 | rs1 = 0x10, Expected = 0x1603d7b9, Output =
    0x1603d7b9
```

```
# [PASS]  Time = 1871000 | rs2 = 0x11, Expected = 0x781c5ba3, Output =
    0x781c5ba3
# [PASS]  Time = 1881000 | rs1 = 0x11, Expected = 0x781c5ba3, Output =
    0x781c5ba3
# [PASS]  Time = 1881000 | rs2 = 0x12, Expected = 0x2381cb4e, Output =
    0x2381cb4e
# [PASS]  Time = 1891000 | rs1 = 0x12, Expected = 0x2381cb4e, Output =
    0x2381cb4e
# [PASS]  Time = 1891000 | rs2 = 0x13, Expected = 0x671e6b6f, Output =
    0x671e6b6f
# [PASS]  Time = 1901000 | rs1 = 0x13, Expected = 0x671e6b6f, Output =
    0x671e6b6f
# [PASS]  Time = 1901000 | rs2 = 0x14, Expected = 0x2ebaab9a, Output =
    0x2ebaab9a
# [PASS]  Time = 1911000 | rs1 = 0x14, Expected = 0x2ebaab9a, Output =
    0x2ebaab9a
# [PASS]  Time = 1911000 | rs2 = 0x15, Expected = 0xc3391889, Output =
    0xc3391889
# [PASS]  Time = 1921000 | rs1 = 0x15, Expected = 0xc3391889, Output =
    0xc3391889
# [PASS]  Time = 1921000 | rs2 = 0x16, Expected = 0xcfe6f7b5, Output =
    0xcfe6f7b5
# [PASS]  Time = 1931000 | rs1 = 0x16, Expected = 0xcfe6f7b5, Output =
    0xcfe6f7b5
# [PASS]  Time = 1931000 | rs2 = 0x17, Expected = 0x6be10d7b, Output =
    0x6be10d7b
# [PASS]  Time = 1941000 | rs1 = 0x17, Expected = 0x6be10d7b, Output =
    0x6be10d7b
# [PASS]  Time = 1941000 | rs2 = 0x18, Expected = 0x720de158, Output =
    0x720de158
# [PASS]  Time = 1951000 | rs1 = 0x18, Expected = 0x720de158, Output =
    0x720de158
# [PASS]  Time = 1951000 | rs2 = 0x19, Expected = 0xdcdd928e, Output =
    0xdcdd928e
# [PASS]  Time = 1961000 | rs1 = 0x19, Expected = 0xdcdd928e, Output =
    0xdcdd928e
# [PASS]  Time = 1961000 | rs2 = 0x1a, Expected = 0x327638a8, Output =
    0x327638a8
# [PASS]  Time = 1971000 | rs1 = 0x1a, Expected = 0x327638a8, Output =
    0x327638a8
# [PASS]  Time = 1971000 | rs2 = 0x1b, Expected = 0xdb062489, Output =
    0xdb062489
# [PASS]  Time = 1981000 | rs1 = 0x1b, Expected = 0xdb062489, Output =
    0xdb062489
# [PASS]  Time = 1981000 | rs2 = 0x1c, Expected = 0x879f10ac, Output =
    0x879f10ac
# [PASS]  Time = 1991000 | rs1 = 0x1c, Expected = 0x879f10ac, Output =
    0x879f10ac
# [PASS]  Time = 1991000 | rs2 = 0x1d, Expected = 0x38674ea5, Output =
    0x38674ea5
# [PASS]  Time = 2001000 | rs1 = 0x1d, Expected = 0x38674ea5, Output =
    0x38674ea5
```

```
# [PASS]  Time = 2001000 | rs2 = 0x1e, Expected = 0x5a2a5c21, Output =
    0x5a2a5c21
# [PASS]  Time = 2011000 | rs1 = 0x1e, Expected = 0x5a2a5c21, Output =
    0x5a2a5c21
# [PASS]  Time = 2011000 | rs2 = 0x1f, Expected = 0x0d5f6a9d, Output =
    0x0d5f6a9d
# ==================================================================
# SUMMARY
# ==================================================================
# TOTAL TESTS : 157
# TOTAL PASSED: 157
# TOTAL FAILED: 0
# ======> ALL TESTS PASSED
# ** Note: $finish    : ../../tb/tb_reg.v(186)
#    Time: 2111 ns  Iteration: 0  Instance: /tb_reg
```

Simulation 1: Simulation results of the RegFile module.

5.2 Processor testbench results

```
# ==================================================================
# ==================================================================
# TEST BEGIN
# ==================================================================
# ==================================================================
# TEST 1
# ==================================================================
# [PASS] Time = 371 |            ADDI x1, x0, 10 | Output x1 = 0
   x0000000a
# [PASS] Time = 371 |            ADDI x2, x0, -5 | Output x2 = 0
   xfffffffb
# [PASS] Time = 371 |             ADD x3, x1, x2 | Output x3 = 0
   x00000005
# [PASS] Time = 371 |             SUB x4, x1, x2 | Output x4 = 0
   x0000000f
# [PASS] Time = 371 |             AND x5, x1, x2 | Output x5 = 0
   x0000000a
# [PASS] Time = 371 |              OR x6, x1, x2 | Output x6 = 0
   xfffffffb
# [PASS] Time = 371 |             XOR x7, x1, x2 | Output x7 = 0
   xfffffff1
# [PASS] Time = 371 |             SLLI x8, x1, 2 | Output x8 = 0
   x00000028
# [PASS] Time = 371 |             SRAI x9, x2, 1 | Output x9 = 0
   xfffffffd
# [PASS] Time = 371 |            SRLI x10, x2, 1 | Output x10 = 0
   x7ffffffd
# [PASS] Time = 371 |            SLT x11, x2, x1 | Output x11 = 0
   x00000001
# [PASS] Time = 371 |           SLTU x12, x2, x1 | Output x12 = 0
   x00000000
# [PASS] Time = 371 |            MUL x13, x1, x1 | Output x13 = 0
   x00000064
```

```
# [PASS] Time = 371 |             DIV x14, x1, x2 | Output x14 = 0
   xfffffffe
# [PASS] Time = 371 |             REM x15, x1, x2 | Output x15 = 0
   x00000000
# [PASS] Time = 371 |            DIVU x16, x2, x1 | Output x16 = 0
   x19999999
# [PASS] Time = 371 |             LW x18, 100(x0) | Output x18 = 0
   x00000005
# [PASS] Time = 371 |             ADDI x19, x0, 1 | Output x19 = 0
   x00000001
# [PASS] Time = 371 |                  JAL x20, 8 | Output x20 = 0
   x00000058
# [PASS] Time = 371 |             ADDI x21, x0, 2 | Output x21 = 0
   x00000002
# [PASS] Time = 371 |            LUI x22, 0x12345 | Output x22 = 0
   x12345000
# [PASS] Time = 371 |             XORI x23, x1, 5 | Output x23 = 0
   x0000000f
# [PASS] Time = 371 |          ANDI x24, x1, 0xF | Output x24 = 0
   x0000000a
# [PASS] Time = 371 |       ORI x25, x13, 0xFF00 | Output x25 = 0
   xffffff64
# [PASS] Time = 371 |          SLTI x26, x7, 0x28 | Output x26 = 0
   x00000001
# [PASS] Time = 371 | LTIU x27, x8, 0xFFFF_FFF1 | Output x27 = 0
   x00000001
# [PASS] Time = 371 |            SLL x28, x1, x19 | Output x28 = 0
   x00000014
# [PASS] Time = 371 |            SRL x29, x6, x19 | Output x29 = 0
   x7ffffffd
# [PASS] Time = 371 |            SRA x30, x6, x19 | Output x30 = 0
   xfffffffd
# [PASS] Time = 371 |            REMU x31, x6, x3 | Output x31 = 0
   x00000001
# [PASS] HALT asserted.
# ============================================================
# TEST 2 (REUSED REGISTERS)
# ============================================================
# [PASS] Time = 651 |   ADDI x1, x0, 0xFFFF_FFFB | Output x1 = 0
   xfffffffb
# [PASS] Time = 651 |              ADDI x2, x0, 3 | Output x2 = 0
   x00000003
# [PASS] Time = 651 |             MULH x3, x1, x2 | Output x3 = 0
   xffffffff
# [PASS] Time = 651 |           MULHSU x4, x1, x2 | Output x4 = 0
   xffffffff
# [PASS] Time = 651 |            MULHU x5, x1, x2 | Output x5 = 0
   x00000002
# [PASS] Time = 651 |   ADDI x6, x0, 0xABCD_FE98 | Output x6 = 0
   xfffffe98
# [PASS] Time = 651 |              LB x7, 100(x0) | Output x7 = 0
   xffffff98
```

```
# [PASS] Time = 651 |             LH x8, 104(x0) | Output x8 = 0
    xffffe98
# [PASS] Time = 651 |            LBU x9, 105(x0) | Output x9 = 0
    x000000fe
# [PASS] Time = 651 |           LHU x10, 104(x0) | Output x10 = 0
    x0000fe98
# [PASS] Time = 651 |           ADD x11, x9, x10 | Output x11 = 0
    x0000ff96
# [PASS] Time = 651 |            OR x12, x9, x10 | Output x12 = 0
    x0000fefe
# [PASS] Time = 651 |            LW x13, 120(x0) | Output x13 = 0
    x000000fe
# [PASS] Time = 651 |          JALR x14, 130(x0) | Output x14 = 0
    x00000054
# [PASS] HALT asserted.
# ==================================================================
# SUMMARY
# ==================================================================
# TOTAL TESTS : 46
# TOTAL PASSED: 46
# TOTAL FAILED: 0
# ======> ALL TESTS PASSED
# ==================================================================
# TEST END
# ==================================================================
# ** Note: $finish    : ../../tb/tb_single.v(230)
#    Time: 751 ns  Iteration: 0  Instance: /tb_single
```

Simulate 2: Testbench results of the Processor module.

6 Simulation waveform

6.1 RegFile simulation waveform

The screenshot in Figure 1 presents a portion of the waveform from Test 3. In this test, a random value is written to the destination register. The waveform confirms that the same value can be correctly read from both source registers, rs1 and rs2.

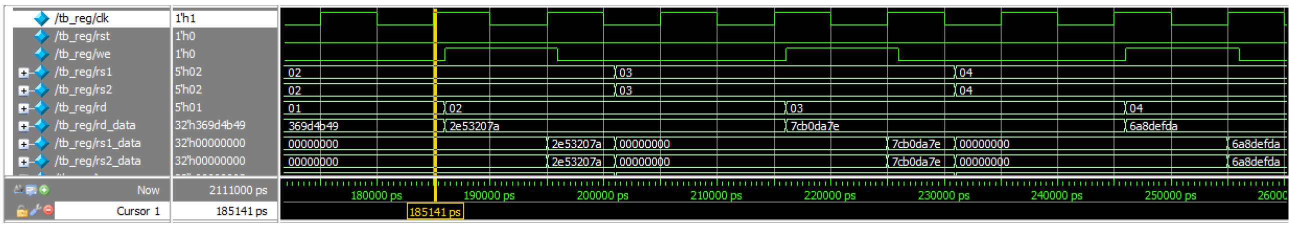


Figure 1

Figure 1 shows that at the positive clock edge occurring at 185141 ps, the write enable (`we`) signal is asserted. At this moment, the destination register `rd` is set to 0x2, and the written data `rd_data` is 32'h2e53207a. Because the storage is updated synchronously with the positive clock edge, while rs1 and rs2 can be read immediately after the new value is written, both `rs1_data, rs2_data` are updated to 32'h2e53207a on the next clock edge. This behavior confirms the correct store and read operations of the Register File module.

6.2 Processor simulation waveform

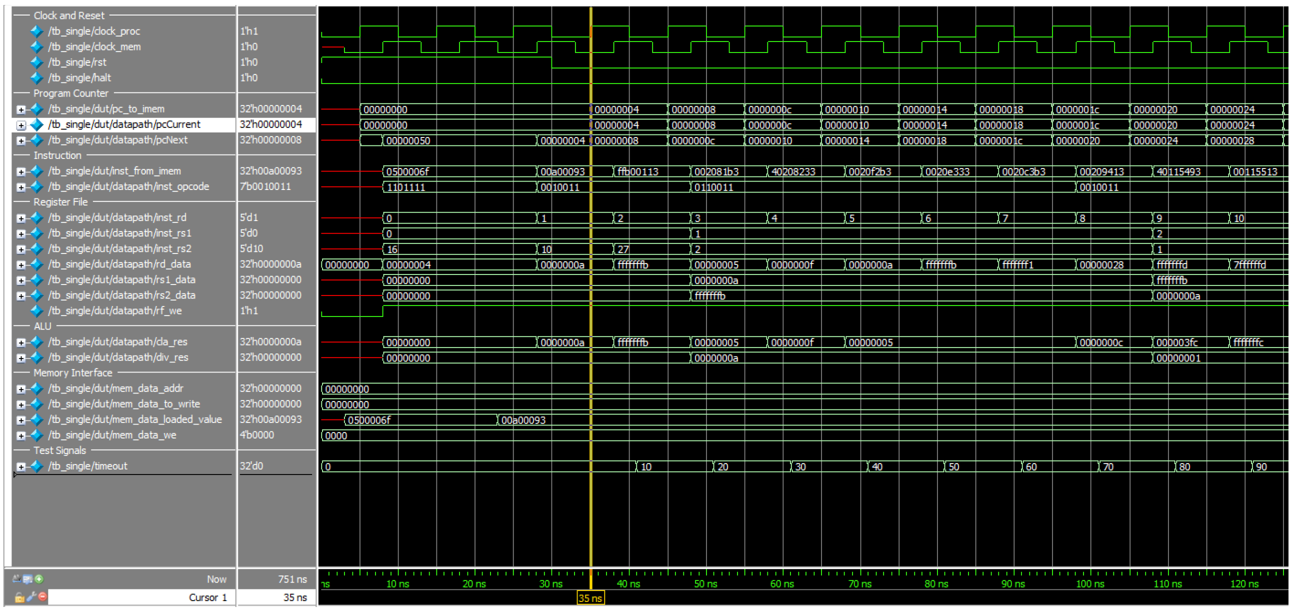

Clock and Reset
/tb_single/clock_proc
/tb_single/clock_mem
/tb_single/rst
/tb_single/halt
Program Counter
/tb_single/dut/pc_to_imem
/tb_single/dut/datapath/pcCurrent
/tb_single/dut/datapath/pcNext
Instruction
/tb_single/dut/inst_from_imem
/tb_single/dut/datapath/inst_opcode
Register File
/tb_single/dut/datapath/inst_rd
/tb_single/dut/datapath/inst_rs1
/tb_single/dut/datapath/inst_rs2
/tb_single/dut/datapath/rd_data
/tb_single/dut/datapath/rs1_data
/tb_single/dut/datapath/rs2_data
/tb_single/dut/datapath/rf_we
ALU
/tb_single/dut/datapath/cla_res
/tb_single/dut/datapath/div_res
Memory Interface
/tb_single/dut/mem_data_addr
/tb_single/dut/mem_data_to_write
/tb_single/dut/mem_data_loaded_value
/tb_single/dut/mem_data_we
Test Signals
/tb_single/timeout
Now
751 ns
Cursor 1
35 ns

Figure 2

Looking at Figure 2, we can see that the first 30 ns is used to load instructions into the memory array while reset is asserted. Once reset is released at t = 30ns, instruction execution begins starting from PC = 0x00.

The first instruction is 0x00a00093 (ADDI x1, x0, 10). The instruction fetch occurs on the positive edge of `clock_mem`, where the instruction is read from instruction memory into the `inst_from_imem` signal. The processor then decodes this instruction combinationally: `inst_rd` becomes 1 (x1), `inst_rs1` becomes 0 (x0), and the immediate value is extracted as 10. Since ADDI is an I-type instruction, `inst_rs2` is not used by the operation, but the decoder still extracts bits [24:20] from the instruction encoding. The register file performs a combinational read, so `rs1_data` immediately reflects the value of x0 (which is 0). On the next positive edge of `clock_proc`, the result (0 + 10 = 10) is written to register x1, and the PC increments to 0x04.

Looking at a later instruction 0x002081b3 at PC = 0x08 (ADD x3, x1, x2), we observe more complex datapath behavior. The decoder extracts: `inst_rd` = 3 (x3), `inst_rs1` = 1 (x1), `inst_rs2` = 2 (x2). The register file performs combinational reads, outputting `rs1_data` = 0xA (10 decimal) and `rs2_data` = 0xFFFFFFFB (-5 in two's complement). These values are fed into the CLA (Carry Lookahead Adder) module, which computes 10 + (-5) = 5. The `cla_res` signal shows 0x00000005, and this value propagates to `rd_data`. On the next positive edge of `clock_proc`, the value 5 is written to register x3.

The key timing relationship is: instruction fetch happens on the positive edge of `clock_mem`, combinational decoding and ALU computation occur immediately, and register writeback happens on the next positive edge of `clock_proc`. Memory operations (loads/stores) use the negative edge of `clock_mem` for data access to avoid conflicts with instruction fetch.

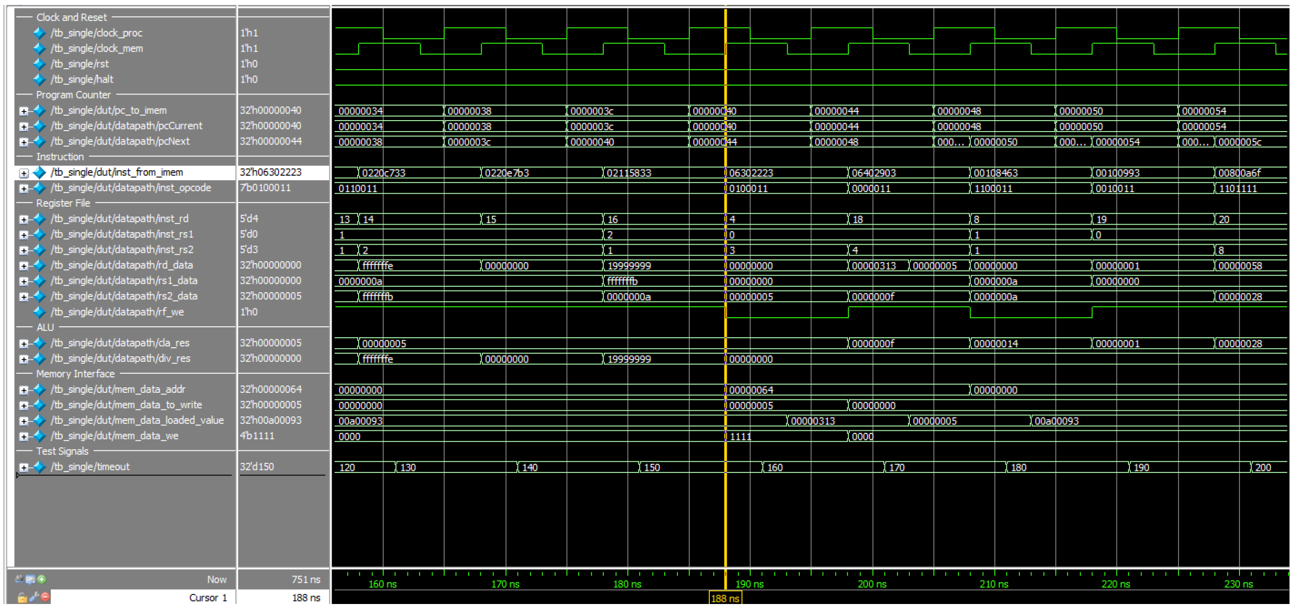
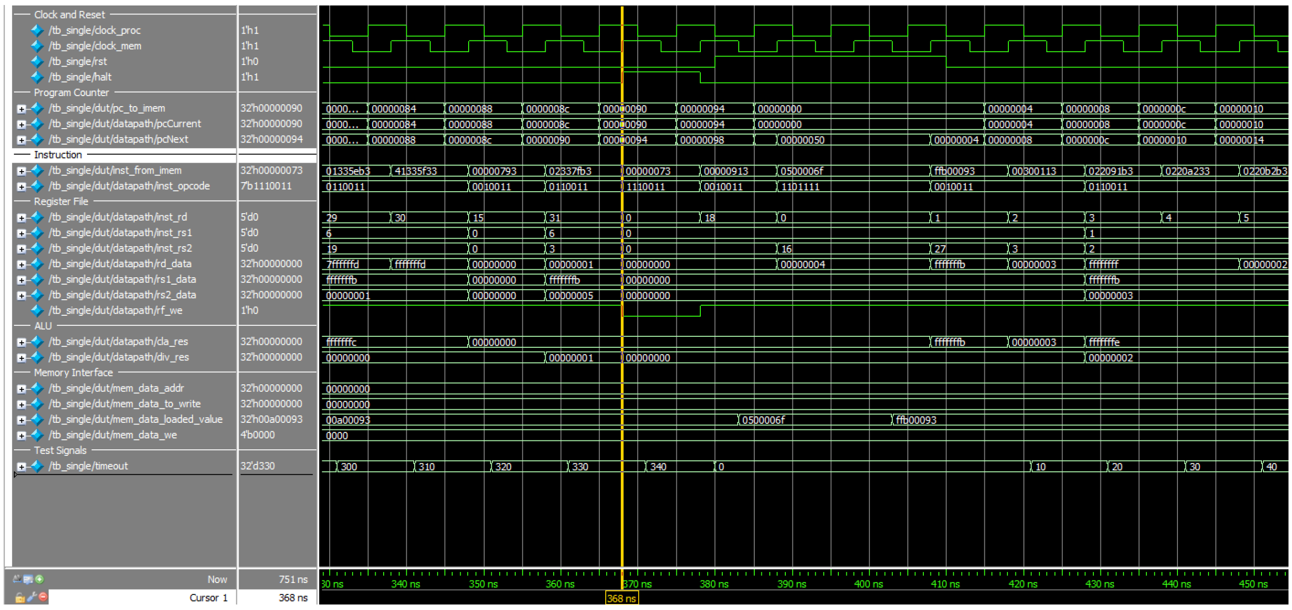

Clock and Reset
/tb_single/clock_proc
/tb_single/clock_mem
/tb_single/rst
/tb_single/halt
Program Counter
/tb_single/dut/pc_to_imem
/tb_single/dut/datapath/pcCurrent
/tb_single/dut/datapath/pcNext
Instruction
/tb_single/dut/inst_from_imem
/tb_single/dut/datapath/inst_opcode
Register File
/tb_single/dut/datapath/inst_rd
/tb_single/dut/datapath/inst_rs1
/tb_single/dut/datapath/inst_rs2
/tb_single/dut/datapath/rd_data
/tb_single/dut/datapath/rs1_data
/tb_single/dut/datapath/rs2_data
/tb_single/dut/datapath/rf_we
ALU
/tb_single/dut/datapath/cla_res
/tb_single/dut/datapath/div_res
Memory Interface
/tb_single/dut/mem_data_addr
/tb_single/dut/mem_data_to_write
/tb_single/dut/mem_data_loaded_value
/tb_single/dut/mem_data_we
Test Signals
/tb_single/timeout
Now 751 ns
Cursor 1 188 ns

Figure 3

Looking at the instruction 0x02115833 (DIVU x16, x2, x1) executing around 180 ns in Figure 3, we observe the unsigned division operation. The `rs2_data` signal shows 0x0000000A (10 decimal), which is the value from register x1, while `rs1_data` shows 0xFFFFFFFB (4,294,967,291 unsigned), which is the value from register x2. Note that DIVU follows the format DIVU rd, rs1, rs2, which computes rd = rs1 / rs2. Therefore, this instruction calculates x2 / x1 = 4,294,967,291 / 10 = 429,496,729 = 0x19999999. This unsigned division is executed by the `divider_unsigned` module, and the result appears on the `div_res` signal as 0x19999999. The `rd_data` signal also reflects this value, confirming that 0x19999999 will be written to register x16 on the next positive edge of `clock_proc`.

At approximately 188 ns, a store instruction 0x06302223 (SW x3, 100(x0)) executes. The `rf_we` (register file write enable) signal is deasserted because store instructions do not write to any register, unlike arithmetic or load instructions which require register writeback. For the SW instruction format (S-type), rs1 provides the base address, rs2 provides the source data, and the immediate provides the offset. In this case, `inst_rs2` = 3 (x3) with `rs2_data` = 0x00000005, `inst_rs1` = 0 (x0) with `rs1_data` = 0x00000000, and the offset is 100. The effective address is calculated as 0 + 100 = 100 (decimal) = 0x00000064 (hex), which appears on `mem_data_addr`. The value 0x00000005 from x3 appears on `mem_data_to_write`, and this data is written to memory address 100 on the negative edge of `clock_mem`.

Subsequently, when instruction 0x06402903 (LW x18, 100(x0)) executes, the processor reads from the same memory location. The effective address calculation uses `rs1_data` = 0x00000000 (from x0) plus offset 100, resulting in address 0x00000064 on `mem_data_addr`. The `mem_data_loaded_value` signal shows 0x00000005, which is the data previously stored. The `inst_rd` field specifies x18 as the destination register, so this loaded value will be written to register x18 on the next `clock_proc` positive edge.



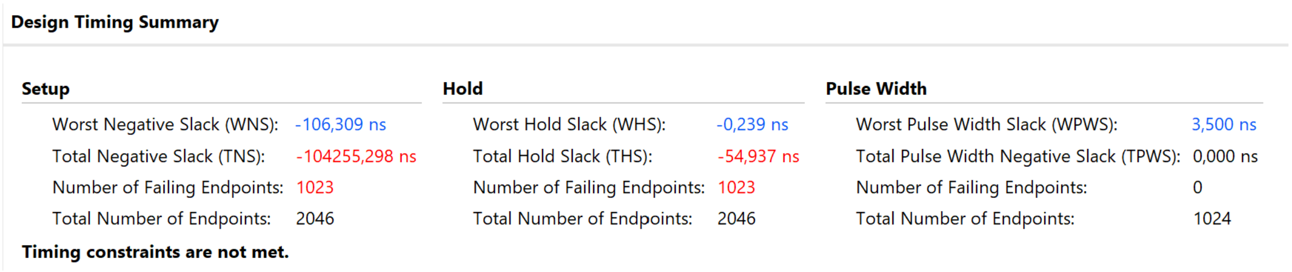
Clock and Reset
/tb_single/clock_proc 1'h1
/tb_single/clock_mem 1'h1
/tb_single/rst 1'h0
/tb_single/halt 1'h1
Program Counter
/tb_single/dut/pc_to_imem 32'h00000090
/tb_single/dut/datapath/pcCurrent 32'h00000090
/tb_single/dut/datapath/pcNext 32'h00000094
Instruction
/tb_single/dut/inst_from_imem 32'h00000073
/tb_single/dut/datapath/inst_opcode 7'b1110011
Register File
/tb_single/dut/datapath/inst_rd 5'd0
/tb_single/dut/datapath/inst_rs1 5'd0
/tb_single/dut/datapath/inst_rs2 5'd0
/tb_single/dut/datapath/rd_data 32'h00000000
/tb_single/dut/datapath/rs1_data 32'h00000000
/tb_single/dut/datapath/rs2_data 32'h00000000
/tb_single/dut/datapath/rf_we 1'h0
ALU
/tb_single/dut/datapath/cla_res 32'h00000000
/tb_single/dut/datapath/div_res 32'h00000000
Memory Interface
/tb_single/dut/mem_data_addr 32'h00000000
/tb_single/dut/mem_data_to_write 32'h00000000
/tb_single/dut/mem_data_loaded_value 32'h00a00093
/tb_single/dut/mem_data_we 4'b0000
Test Signals
/tb_single/timeout 32'd330
Now 751 ns
Cursor 1 368 ns
368 ns

Figure 4

In Figure 4, at approximately 368 ns, the HALT instruction (0x00000073) is executed at PC = 0x90. This causes the halt signal to assert high, indicating successful completion of Test 1. The processor stops fetching and executing further instructions when halt is asserted. At 380 ns, the testbench reasserts the rst (reset) signal to reinitialize the processor state for Test 2. During this reset phase, the program counter returns to 0x0, and all 32 registers in the register file are cleared to 0x0. The testbench then loads a new sequence of instructions into instruction memory before releasing reset to begin Test 2 execution.

7 Timing closure and Resource utilization

7.1 Timing closure



Design Timing Summary

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	-106,309 ns	Worst Hold Slack (WHS):	-0,239 ns	Worst Pulse Width Slack (WPWS):	3,500 ns
Total Negative Slack (TNS):	-104255,298 ns	Total Hold Slack (THS):	-54,937 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	1023	Number of Failing Endpoints:	1023	Number of Failing Endpoints:	0
Total Number of Endpoints:	2046	Total Number of Endpoints:	2046	Total Number of Endpoints:	1024

Timing constraints are not met.

Figure 5: Timing summary.

Based on the implementation results from Vivado, the timing summary is as follows:

- Clock (clock_proc) constraint: 125 MHz (Period: 8.000 ns)
- Worst negative slack (WNS): -106.309 ns
- Conclusion: The design has not achieved Timing closure (Timing constraints are not met).

The negative WNS indicates that the signal propagation delay exceeds the clock period constraint. This is expected for a single-cycle processor architecture. In a single-cycle design, the critical path is significantly long as it must traverse multiple stages within one clock cycle, including instruction memory fetch, register file read, ALU/divider execution, data memory

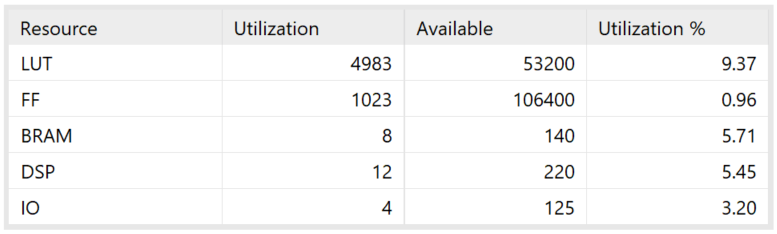
access and write-back.
Based on the WNS, the minimum clock period required for the design to function correctly is:

$$T_{min} = T_{constraint} - WNS = 8.000 - (-106.309) \approx 114.31ns$$

The estimated maximum operating frequency (Fmax) is:

$$F_{max} = \frac{1}{T\_min} \approx 8.75MHz$$

7.2 Resource utilization



Resource	Utilization	Available	Utilization %
LUT	4983	53200	9.37
FF	1023	106400	0.96
BRAM	8	140	5.71
DSP	12	220	5.45
IO	4	125	3.20

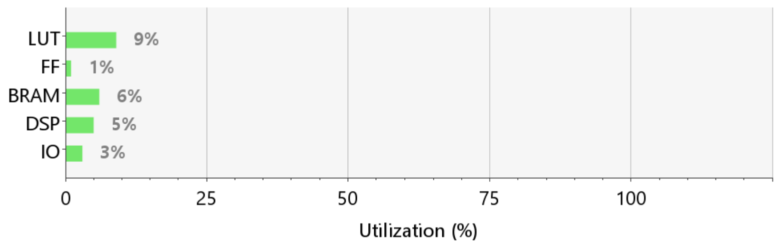


Figure 6: Utilization summary.

The design fits comfortably within the available resources of the Arty Z7-20 board. The utilization is dominated by Slice LUTs (Look-up Tables) rather than Flip-Flops. This is consistent with a single-cycle architecture, which relies heavily on large combinational logic blocks (such as the ALU, multiplier, and especially the divider) to complete instructions in a single cycle, rather than pipelining registers.