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

INTRODUCTION TO SYSTEM ON CHIP
ASSIGNMENT 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 Assignment project files (RTL source code, testbenches, simulation scripts) are located in the `Assignment` folder.
They are available at the following repository: https://github.com/trxmhoang/SoC.git

2 Overview

2.1 Introduction

The objective of this assignment was to upgrade a multi-cycle RISC-V datapath into a 5-stage pipelined processor. Pipelining increases instruction throughput by overlapping the execution of multiple instructions. This implement supports the RV32I instruction set and handles various hazards inherent in pipelined designs, including data hazards, control hazards and structural hazards introduced by the multi-cycle divider. The design was implemented in Verilog, simulated to verify functionality and synthesized to analyze resource usage and timing performance.

2.2 System design and Implementation

The datapath is divided into five standard stages: Fetch (D), Decode (D), Execute (X), Memory (M) and Writeback (W).

2.2.1 Pipeline stages

- Fetch (F): The Program Counter (PC) addresses the instruction memory. The PC is updated every cycle, either incrementally by 4 (PC + 4) or to a branch/jump target.
- Decode (D): Instructions are decoded and register operands are read from the Register File. Hazard detection logic for load-use and divider hazards operates in this stage to trigger stalls.
- Execute (X): The ALU performs arithmetic/logic operations. Branch outcomes are also resolved here. The 8-stage pipelined divider is also initiated in this stage.
- Memory (M): Data memory access (load/store) occurs here. The memory operates on the negative edge of the clock to ensure stability.
- Writeback (W): Results from the ALU, memory or divider are written back to the destination register.

2.2.2 Hazard handling

To ensure correct execution without flushing the pipeline unnecessarily, several hazard mitigation techniques were implemented.

- Data hazards (Forwarding): Forwarding (Bypassing) was implemented to supply the most recent register values to the Execute stage without waiting for Writeback.
  - MX bypass (Memory-to-Execute): If an instruction in the Memory stage writes to a register needed by the instruction needed in Execute, the value is forwarded from `m_alu_res`.
  - WX bypass (Writeback-to-Execute): If an instruction in the Writeback stage writes to a register needed in Execute, the value is forwarded from `w_final_data`.

  - WD bypass (Writeback-to-Decode): To handle scenarios where an instruction reads a register currently being written in the same cycle, internal forwarding is used in the Decode stage logic.

- Load-use hazards (Stalling): If a Load instruction is in the Execute stage and the instruction in Decode depends on the load result, a stall is required because the data is not yet available.
  - Detection: `x_mem_read` is true AND `x_rd` matches `d_rs1` or `d_rs2`.
  - Action: The PC and Fetch/Decode registers are frozen, and a NOP is inserted into the Execute stage bubble.

- Control hazards (Flushing): Branch decisions are made in the Execute stage. We assume "Branch not taken" by default.
  - Misprediction: If a branch is taken (`pc_src` is high), the instructions currently in Fetch and Decode are invalid.
  - The pipeline flushes these stages by synchronously resetting the `d_inst` and `x_inst` registers to NOPs.

2.2.3 Pipeline integration

This module reuses the `DividerUnsignedPipelined` (8-stage) and `cla` (Carry Lookahead Adder) from the previous labs.

- Wrapper logic: Since the divider is unsigned, the wrapper converts signed operands to absolute values, feeds them to the divider and restores the correct sign for the quotient/remainder based on the RISC-V specification.

- FIFO buffer: To manage the long latency of division (8 cycles), a FIFO (`fifo_div_...`) is used. It stores the destination register (`rd`), PC and other data alongside the divider stages. When the divider finshes (`div_done`), the result and the stored `rd` emerge and are selected for the Memory stage.

- Structural hazards and Stalling:
  - Raw hazards: If an instruction in Decode needs a register currently pending in the divider pipeline, `div_raw_hazard` triggers a stall.
  - Writeback order: To maintain safety, non-divide instructions that write to registers are stalled if the divider is busy (`div_struct_hazard`). This ensures that a fast ADD instruction does not overwrite a register before a slow DIV instruction finishes, preserving data integrity.

2.3 Implemention details

The `DatapathPipelined.v` module integrates all stages.

- Memory interface: The module interacts with `MemorySingleCycle`. Reads and writes to data memory are synchronized on the negative edge (`negedge`) of the clock.

- ALU and Divider selection: A multiplexer in the Memory stage logic selects bettween a standard ALU results (`alu_res`) and the divider result (`div_res`) based on the `div_done` signal.

3 RTL implementation

```
`timescale 1ns / 1ns
`define REG_SIZE 31
`define INST_SIZE 31
`define OPCODE_SIZE 6
`define DIVIDER_STAGES 8

// Don't forget your old codes
//`include "cla.v"
//`include "DividerUnsignedPipelined.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;

  always @(negedge clk) begin
    if (rst)
        for (i = 0; i < NumRegs; i = i + 1) regs[i] <= 0;
    else if (we && (rd != 0))
        regs[rd] <= rd_data;
  end

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

module DatapathPipelined (
  input                     clk,
  input                     rst,
  output     [`REG_SIZE:0] pc_to_imem,
  input      [`INST_SIZE:0] inst_from_imem,
  // dmem is read/write
  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,
  output reg               halt,
  // The PC of the inst currently in Writeback. 0 if not a valid inst.
```

```
52    output reg [ `REG_SIZE:0] trace_writeback_pc,
53    // The bits of the inst currently in Writeback. 0 if not a valid
         inst.
54    output reg [`INST_SIZE:0] trace_writeback_inst
55  );
56
57    localparam [`OPCODE_SIZE:0] OpcodeLoad    = 7'b00_000_11;
58    localparam [`OPCODE_SIZE:0] OpcodeStore   = 7'b01_000_11;
59    localparam [`OPCODE_SIZE:0] OpcodeBranch  = 7'b11_000_11;
60    localparam [`OPCODE_SIZE:0] OpcodeJalr    = 7'b11_001_11;
61    localparam [`OPCODE_SIZE:0] OpcodeMiscMem = 7'b00_011_11;
62    localparam [`OPCODE_SIZE:0] OpcodeJal     = 7'b11_011_11;
63
64    localparam [`OPCODE_SIZE:0] OpcodeRegImm  = 7'b00_100_11;
65    localparam [`OPCODE_SIZE:0] OpcodeRegReg  = 7'b01_100_11;
66    localparam [`OPCODE_SIZE:0] OpcodeEnviron = 7'b11_100_11;
67
68    localparam [`OPCODE_SIZE:0] OpcodeAuipc   = 7'b00_101_11;
69    localparam [`OPCODE_SIZE:0] OpcodeLui     = 7'b01_101_11;
70
71    localparam [`INST_SIZE:0] NOP = 32'h00000013;
72
73    reg [`REG_SIZE:0] cycles_current;
74    always @(posedge clk) begin
75      if (rst) begin
76        cycles_current <= 0;
77      end else begin
78        cycles_current <= cycles_current + 1;
79      end
80    end
81
82    reg  [`REG_SIZE:0] f_pc;
83    wire [`REG_SIZE:0] f_pc_next;
84    wire stall;
85    wire pc_src;
86    wire [`REG_SIZE:0] x_target_pc;
87
88    reg [`REG_SIZE:0] d_pc;
89    reg [`INST_SIZE:0] d_inst;
90
91    reg [`REG_SIZE:0] x_pc, x_rs1_data, x_rs2_data, x_imm;
92    reg [`INST_SIZE:0] x_inst;
93    reg [4:0] x_rd, x_rs1, x_rs2;
94    reg [6:0] x_opcode, x_funct7;
95    reg [2:0] x_funct3;
96    reg x_reg_write, x_mem_read, x_mem_write, x_branch, x_jal, x_jalr,
         x_halt;
97
98    reg [`REG_SIZE:0] m_pc, m_alu_res, m_rs2_data;
99    reg [`INST_SIZE:0] m_inst;
100   reg [4:0] m_rd;
101   reg [2:0] m_funct3;
102   reg m_reg_write, m_mem_read, m_mem_write, m_halt;
```

```
103
104   reg [`REG_SIZE:0] w_pc, w_alu_res, w_mem_read_data;
105   reg [`INST_SIZE:0] w_inst;
106   reg [4:0] w_rd;
107   reg [2:0] w_funct3;
108   reg w_reg_write, w_mem_read, w_halt;
109
110   wire [`REG_SIZE:0] w_final_data;
111
112   /***************/
113   /* FETCH STAGE */
114   /***************/
115   assign f_pc_next = pc_src ? x_target_pc : (f_pc + 4);
116   assign pc_to_imem = f_pc;
117
118   always @(posedge clk) begin
119     if (rst)
120       f_pc <= 0;
121     else if (!stall)
122       f_pc <= f_pc_next;
123   end
124
125   always @(posedge clk) begin
126     if (rst) begin
127         d_pc <= 0;
128         d_inst <= NOP;
129     end else if (pc_src) begin
130         d_pc <= 0;
131         d_inst <= NOP;
132     end else if (!stall) begin
133         d_pc <= f_pc;
134         d_inst <= inst_from_imem;
135     end
136   end
137
138   /****************/
139   /* DECODE STAGE */
140   /****************/
141
142   wire [6:0] d_funct7, d_opcode;
143   wire [2:0] d_funct3;
144   wire [4:0] d_rs1, d_rs2, d_rd;
145   wire [`REG_SIZE:0] rf_rs1_data, rf_rs2_data;
146   assign {d_funct7, d_rs2, d_rs1, d_funct3, d_rd, d_opcode} = d_inst;
147
148   RegFile rf (
149     .rd(w_rd),
150     .rd_data(w_final_data),
151     .we(w_reg_write),
152     .rs1(d_rs1),
153     .rs1_data(rf_rs1_data),
154     .rs2(d_rs2),
155     .rs2_data(rf_rs2_data),
```

```
156     .clk(clk),
157     .rst(rst)
158   );
159
160   wire [`REG_SIZE:0] d_rs1_val = (w_reg_write && w_rd != 0 && w_rd ==
          d_rs1) ? w_final_data : rf_rs1_data;
161   wire [`REG_SIZE:0] d_rs2_val = (w_reg_write && w_rd != 0 && w_rd ==
          d_rs2) ? w_final_data : rf_rs2_data;
162
163   reg [`REG_SIZE:0] d_imm;
164   always @(*) begin
165     case (d_opcode)
166       OpcodeStore:  d_imm = { {20{d_inst[31]}}, d_inst[31:25], d_inst
              [11:7] };
167       OpcodeBranch: d_imm = { {20{d_inst[31]}}, d_inst[7], d_inst
              [30:25], d_inst[11:8], 1'b0 };
168       OpcodeJal:    d_imm = { {12{d_inst[31]}}, d_inst[19:12], d_inst
              [20], d_inst[30:21], 1'b0 };
169       OpcodeLui, OpcodeAuipc: d_imm = { d_inst[31:12], 12'b0 };
170       default:      d_imm = { {20{d_inst[31]}}, d_inst[31:20] };
171     endcase
172   end
173
174   wire d_use_rs1 = (d_opcode != OpcodeLui && d_opcode != OpcodeAuipc
          && d_opcode != OpcodeJal);
175   wire d_use_rs2 = (d_opcode == OpcodeRegReg || d_opcode ==
          OpcodeStore || d_opcode == OpcodeBranch);
176
177   wire load_use_hazard = (x_mem_read && x_rd != 0 && ((d_use_rs1 &&
          d_rs1 == x_rd) || (d_use_rs2 && d_rs2 == x_rd)));
178
179   reg [4:0] fifo_div_rd [0:`DIVIDER_STAGES-1];
180   reg fifo_div_valid [0:`DIVIDER_STAGES-1];
181   reg fifo_div_rem [0:`DIVIDER_STAGES-1];
182   reg [31:0] fifo_div_pc [0:`DIVIDER_STAGES-1];
183   reg [31:0] fifo_div_inst [0:`DIVIDER_STAGES-1];
184   reg fifo_div_a_neg [0:`DIVIDER_STAGES-1];
185   reg fifo_div_b_neg [0:`DIVIDER_STAGES-1];
186   reg fifo_div_by_zero [0:`DIVIDER_STAGES-1];
187   reg [31:0] fifo_div_dividend [0:`DIVIDER_STAGES-1];
188
189   wire x_is_div = (x_opcode == OpcodeRegReg) && (x_funct7 == 7'
          b0000001) && x_funct3[2];
190   wire div_in_ex = x_is_div && x_reg_write && (x_rd != 0);
191
192   wire [4:0] div_rd_in_fifo [0:7];
193   assign div_rd_in_fifo[0] = fifo_div_valid[0] ? fifo_div_rd[0] : 5'b0
          ;
194   assign div_rd_in_fifo[1] = fifo_div_valid[1] ? fifo_div_rd[1] : 5'b0
          ;
195   assign div_rd_in_fifo[2] = fifo_div_valid[2] ? fifo_div_rd[2] : 5'b0
          ;
```

```
196   assign div_rd_in_fifo[3] = fifo_div_valid[3] ? fifo_div_rd[3] : 5'b0
      ;
197   assign div_rd_in_fifo[4] = fifo_div_valid[4] ? fifo_div_rd[4] : 5'b0
      ;
198   assign div_rd_in_fifo[5] = fifo_div_valid[5] ? fifo_div_rd[5] : 5'b0
      ;
199   assign div_rd_in_fifo[6] = fifo_div_valid[6] ? fifo_div_rd[6] : 5'b0
      ;
200   assign div_rd_in_fifo[7] = fifo_div_valid[7] ? fifo_div_rd[7] : 5'b0
      ;
201
202   wire div_raw_hazard =
203                     (d_use_rs1 && (d_rs1 != 0) &&
204                       ( (div_in_ex && (d_rs1 == x_rd)) ||
205                         (d_rs1 == div_rd_in_fifo[0]) ||
206                         (d_rs1 == div_rd_in_fifo[1]) ||
207                         (d_rs1 == div_rd_in_fifo[2]) ||
208                         (d_rs1 == div_rd_in_fifo[3]) ||
209                         (d_rs1 == div_rd_in_fifo[4]) ||
210                         (d_rs1 == div_rd_in_fifo[5]) ||
211                         (d_rs1 == div_rd_in_fifo[6])
212                       )
213                     ) ||
214                     (d_use_rs2 && (d_rs2 != 0) &&
215                       ( (div_in_ex && (d_rs2 == x_rd)) ||
216                         (d_rs2 == div_rd_in_fifo[0]) ||
217                         (d_rs2 == div_rd_in_fifo[1]) ||
218                         (d_rs2 == div_rd_in_fifo[2]) ||
219                         (d_rs2 == div_rd_in_fifo[3]) ||
220                         (d_rs2 == div_rd_in_fifo[4]) ||
221                         (d_rs2 == div_rd_in_fifo[5]) ||
222                         (d_rs2 == div_rd_in_fifo[6])
223                       )
224                     );
225
226
227   wire d_writes_reg = (d_opcode != OpcodeStore && d_opcode !=
      OpcodeBranch) && (d_rd != 0);
228   wire d_is_div = (d_opcode == OpcodeRegReg) && (d_funct7 == 7'
      b0000001) && d_funct3[2];
229
230   wire div_busy = div_in_ex || fifo_div_valid[0] || fifo_div_valid[1]
      || fifo_div_valid[2] || fifo_div_valid[3] || fifo_div_valid[4] ||
       fifo_div_valid[5] || fifo_div_valid[6];
231
232   wire div_struct_hazard = div_busy && d_writes_reg && !d_is_div;
233   assign stall = load_use_hazard || div_raw_hazard ||
      div_struct_hazard;
234
235   always @(posedge clk) begin
236     if (rst || stall || pc_src) begin
237       x_pc <= 0;
238       x_inst <= NOP;
```

```
239         x_rd <= 0;
240         x_rs1 <= 0;
241         x_rs2 <= 0;
242         x_rs1_data <= 0;
243         x_rs2_data <= 0;
244         x_imm <= 0;
245         x_opcode <= 0;
246         x_funct3 <= 0;
247         x_funct7 <= 0;
248         x_reg_write <= 0;
249         x_mem_read <= 0;
250         x_mem_write <= 0;
251         x_branch <= 0;
252         x_jal <= 0;
253         x_jalr <= 0;
254         x_halt <= 0;
255       end else begin
256         x_pc <= d_pc;
257         x_inst <= d_inst;
258         x_rd <= d_rd;
259         x_rs1 <= d_rs1;
260         x_rs2 <= d_rs2;
261         x_rs1_data <= d_rs1_val;
262         x_rs2_data <= d_rs2_val;
263         x_imm <= d_imm;
264         x_opcode <= d_opcode;
265         x_funct3 <= d_funct3;
266         x_funct7 <= d_funct7;
267         x_reg_write <= (d_opcode != OpcodeStore && d_opcode !=
              OpcodeBranch);
268         x_mem_read <= (d_opcode == OpcodeLoad);
269         x_mem_write <= (d_opcode == OpcodeStore);
270         x_branch <= (d_opcode == OpcodeBranch);
271         x_jal <= (d_opcode == OpcodeJal);
272         x_jalr <= (d_opcode == OpcodeJalr);
273         x_halt <= (d_inst == 32'h00000073) || (d_inst == 32'h00100073);
274       end
275     end
276
277     /****************/
278     /* EXECUTE STAGE */
279     /****************/
280
281     wire match_mem_rs1 = (m_reg_write && m_rd != 0 && m_rd == x_rs1);
282     wire match_wb_rs1  = (w_reg_write && w_rd != 0 && w_rd == x_rs1);
283
284     wire match_mem_rs2 = (m_reg_write && m_rd != 0 && m_rd == x_rs2);
285     wire match_wb_rs2  = (w_reg_write && w_rd != 0 && w_rd == x_rs2);
286
287     wire [`REG_SIZE:0] fwd_a_val = match_mem_rs1 ? m_alu_res :
288                                    (match_wb_rs1 ? w_final_data :
                                       x_rs1_data);
289     wire [`REG_SIZE:0] fwd_b_val = match_mem_rs2 ? m_alu_res :
```

```
290                                 (match_wb_rs2 ? w_final_data :
                                       x_rs2_data);
291
292     wire x_is_imm = (x_opcode == OpcodeRegImm) || (x_opcode ==
           OpcodeLoad) || (x_opcode == OpcodeJalr) || (x_opcode ==
           OpcodeAuipc) || (x_opcode == OpcodeLui) || (x_opcode ==
           OpcodeStore);
293     wire [`REG_SIZE:0] x_alu_op1 = fwd_a_val;
294     wire [`REG_SIZE:0] x_alu_op2 = x_is_imm ? x_imm : fwd_b_val;
295     wire x_is_sub = (x_opcode == OpcodeRegReg) && (x_funct7[5] &&
           x_funct3 == 3'b000);
296
297     wire div_start = x_is_div && !pc_src;
298     wire div_signed = !x_funct3[0];
299     wire current_div_a_neg = div_signed && fwd_a_val[31];
300     wire current_div_b_neg = div_signed && fwd_b_val[31];
301     wire current_div_by_zero = (fwd_b_val == 0);
302     wire [31:0] current_div_a = current_div_a_neg ? (~fwd_a_val + 1) :
           fwd_a_val;
303     wire [31:0] current_div_b = current_div_b_neg ? (~fwd_b_val + 1) :
           fwd_b_val;
304
305     integer k;
306     always @(posedge clk) begin
307       if (rst) begin
308         for (k = 0; k < `DIVIDER_STAGES; k = k + 1) begin
309           fifo_div_valid[k] <= 0;
310           fifo_div_rd[k] <= 0;
311           fifo_div_rem[k] <= 0;
312           fifo_div_pc[k] <= 0;
313           fifo_div_inst[k] <= 0;
314           fifo_div_a_neg[k] <= 0;
315           fifo_div_b_neg[k] <= 0;
316           fifo_div_by_zero[k] <= 0;
317           fifo_div_dividend[k] <= 0;
318         end
319       end else begin
320         for (k = `DIVIDER_STAGES-1; k > 0; k = k - 1) begin
321           fifo_div_valid[k] <= fifo_div_valid[k - 1];
322           fifo_div_rd[k] <= fifo_div_rd[k - 1];
323           fifo_div_rem[k] <= fifo_div_rem[k - 1];
324           fifo_div_pc[k] <= fifo_div_pc[k - 1];
325           fifo_div_inst[k] <= fifo_div_inst[k - 1];
326           fifo_div_a_neg[k] <= fifo_div_a_neg[k - 1];
327           fifo_div_b_neg[k] <= fifo_div_b_neg[k - 1];
328           fifo_div_by_zero[k] <= fifo_div_by_zero[k - 1];
329           fifo_div_dividend[k] <= fifo_div_dividend[k - 1];
330         end
331
332         fifo_div_valid[0] <= div_start;
333         fifo_div_rd[0] <= x_rd;
334         fifo_div_rem[0] <= x_funct3[1];
335         fifo_div_pc[0] <= x_pc;
```

```
336         fifo_div_inst[0] <= x_inst;
337         fifo_div_a_neg[0] <= current_div_a_neg;
338         fifo_div_b_neg[0] <= current_div_b_neg;
339         fifo_div_by_zero[0] <= current_div_by_zero;
340         fifo_div_dividend[0] <= fwd_a_val;
341       end
342     end
343
344     wire [31:0] div_quot_raw, div_rem_raw;
345     DividerUnsignedPipelined u_div (
346       .clk(clk),
347       .rst(rst),
348       .stall(1'b0),
349       .i_dividend(current_div_a),
350       .i_divisor(current_div_b),
351       .o_quotient(div_quot_raw),
352       .o_remainder(div_rem_raw)
353     );
354
355     wire [31:0] div_quot_mid = (fifo_div_a_neg[`DIVIDER_STAGES-1] ^
          fifo_div_b_neg[`DIVIDER_STAGES-1]) ? (~div_quot_raw + 1) :
          div_quot_raw;
356     wire [31:0] div_rem_mid = fifo_div_a_neg[`DIVIDER_STAGES-1] ? (~
          div_rem_raw + 1) : div_rem_raw;
357     wire [31:0] div_quot_final = fifo_div_by_zero[`DIVIDER_STAGES-1] ?
          32'hFFFFFFFF : div_quot_mid;
358     wire [31:0] div_rem_final = fifo_div_by_zero[`DIVIDER_STAGES-1] ?
          fifo_div_dividend[`DIVIDER_STAGES-1] : div_rem_mid;
359     wire [31:0] div_res = fifo_div_rem[`DIVIDER_STAGES-1] ?
          div_rem_final : div_quot_final;
360
361     wire div_done = fifo_div_valid[`DIVIDER_STAGES-1];
362
363     wire [31:0] cla_res;
364     wire [31:0] cla_op_b = x_is_sub ? ~x_alu_op2 : x_alu_op2;
365     wire cla_cin = x_is_sub ? 1'b1 : 1'b0;
366
367     cla u_cla (
368       .a(x_alu_op1),
369       .b(cla_op_b),
370       .cin(cla_cin),
371       .sum(cla_res)
372     );
373
374     wire [63:0] mul_u, mul_s, mul_su;
375     assign mul_u = x_alu_op1 * x_alu_op2;
376     assign mul_s = $signed(x_alu_op1) * $signed(x_alu_op2);
377     assign mul_su = $signed(x_alu_op1) * $signed({1'b0, x_alu_op2});
378
379     reg [`REG_SIZE:0] alu_res;
380     always @(*) begin
381       case (x_opcode)
382         OpcodeLui: alu_res = x_alu_op2;
```

```
383       OpcodeAuipc: alu_res = x_pc + x_alu_op2;
384       OpcodeJal, OpcodeJalr: alu_res = x_pc + 4;
385       OpcodeLoad, OpcodeStore: alu_res = cla_res;
386       OpcodeRegImm, OpcodeRegReg:
387         if ((x_opcode == OpcodeRegReg) && (x_funct7 == 7'b0000001))
              begin
388           case (x_funct3)
389             3'b000: alu_res = mul_s[31:0];
390             3'b001: alu_res = mul_s[63:32];
391             3'b010: alu_res = mul_su[63:32];
392             3'b011: alu_res = mul_u[63:32];
393             default: alu_res = div_res;
394           endcase
395         end else begin
396           case (x_funct3)
397             3'b000: alu_res = cla_res;
398             3'b001: alu_res = x_alu_op1 << x_alu_op2[4:0];
399             3'b010: alu_res = ($signed(x_alu_op1) < $signed(x_alu_op2)
                    ) ? 32'd1 : 32'd0;
400             3'b011: alu_res = (x_alu_op1 < x_alu_op2) ? 32'd1 : 32'd0;
401             3'b100: alu_res = x_alu_op1 ^ x_alu_op2;
402             3'b101: begin
403               if (x_funct7[5]) begin
404                 alu_res = $signed(x_alu_op1) >>> x_alu_op2[4:0];
405               end else begin
406                 alu_res = x_alu_op1 >> x_alu_op2[4:0];
407               end
408             end
409             3'b110: alu_res = x_alu_op1 | x_alu_op2;
410             3'b111: alu_res = x_alu_op1 & x_alu_op2;
411             default: alu_res = 32'd0;
412           endcase
413         end
414         default: alu_res = 32'd0;
415     endcase
416   end
417
418   // Branch
419   reg taken;
420   always @(*) begin
421     case (x_funct3)
422       3'b000: taken = (fwd_a_val == fwd_b_val);
423       3'b001: taken = (fwd_a_val != fwd_b_val);
424       3'b100: taken = ($signed(fwd_a_val) < $signed(fwd_b_val));
425       3'b101: taken = ($signed(fwd_a_val) >= $signed(fwd_b_val));
426       3'b110: taken = (fwd_a_val < fwd_b_val);
427       3'b111: taken = (fwd_a_val >= fwd_b_val);
428       default: taken = 0;
429     endcase
430   end
431
432   assign pc_src = (x_branch && taken) || x_jal || x_jalr;
```

```
433   assign x_target_pc = (x_opcode == OpcodeJalr) ? ((fwd_a_val + x_imm)
          & ~32'd1) : (x_pc + x_imm);
434
435   always @(posedge clk) begin
436     if (rst) begin
437       m_pc <= 0;
438       m_inst <= NOP;
439       m_alu_res <= 0;
440       m_rs2_data <= 0;
441       m_rd <= 0;
442       m_funct3 <= 0;
443       m_reg_write <= 0;
444       m_mem_read <= 0;
445       m_mem_write <= 0;
446       m_halt <= 0;
447     end else begin
448       m_pc <= x_pc;
449       m_inst <= x_inst;
450       m_rs2_data <= fwd_b_val;
451       m_funct3 <= x_funct3;
452       m_mem_read <= x_mem_read;
453       m_mem_write <= x_mem_write;
454       m_halt <= x_halt;
455
456       if (div_done) begin
457         m_alu_res <= div_res;
458         m_rd <= fifo_div_rd[`DIVIDER_STAGES-1];
459         m_reg_write <= 1'b1;
460       end else begin
461         m_alu_res <= alu_res;
462         m_rd <= x_rd;
463         m_reg_write <= x_reg_write && !x_is_div;
464       end
465     end
466   end
467
468   /*****************/
469   /* MEMORY STAGE */
470   /*****************/
471
472   always @(*) begin
473     // default values
474     addr_to_dmem = m_alu_res;
475     store_data_to_dmem = m_rs2_data;
476     store_we_to_dmem = 4'b0000;
477
478     if (m_mem_write) begin
479       case (m_funct3)
480         2'b00: begin // SB
481             case (m_alu_res[1:0])
482                 2'b00: begin store_data_to_dmem = {24'b0, m_rs2_data
                        [7:0]};       store_we_to_dmem = 4'b0001; end
```

```
483                 2'b01: begin store_data_to_dmem = {16'b0, m_rs2_data
                        [7:0], 8'b0}; store_we_to_dmem = 4'b0010; end
484                 2'b10: begin store_data_to_dmem = {8'b0, m_rs2_data
                        [7:0], 16'b0}; store_we_to_dmem = 4'b0100; end
485                 2'b11: begin store_data_to_dmem = {m_rs2_data[7:0],
                        24'b0};       store_we_to_dmem = 4'b1000; end
486             endcase
487         end
488         2'b01: begin // SH
489             case (m_alu_res[1])
490                 1'b0: begin store_data_to_dmem = {16'b0, m_rs2_data
                        [15:0]}; store_we_to_dmem = 4'b0011; end
491                 1'b1: begin store_data_to_dmem = {m_rs2_data[15:0],
                        16'b0}; store_we_to_dmem = 4'b1100; end
492             endcase
493         end
494         default: begin // SW
495             store_data_to_dmem = m_rs2_data;
496             store_we_to_dmem = 4'b1111;
497         end
498       endcase
499     end
500   end
501
502   always @(posedge clk) begin
503     if (rst) begin
504       w_reg_write <= 0;
505       w_rd <= 0;
506       w_pc <= 0;
507       w_inst <= NOP;
508       w_halt <= 0;
509       w_alu_res <= 0;
510       w_mem_read_data <= 0;
511       w_funct3 <= 0;
512       w_mem_read <= 0;
513     end else begin
514       w_reg_write <= m_reg_write;
515       w_rd <= m_rd;
516       w_alu_res <= m_alu_res;
517       w_mem_read_data <= load_data_from_dmem;
518       w_mem_read <= m_mem_read;
519       w_funct3 <= m_funct3;
520       w_pc <= m_pc;
521       w_inst <= m_inst;
522       w_halt <= m_halt;
523     end
524   end
525
526   /****************/
527   /*WRITEBACK STAGE*/
528   /****************/
529
530   wire [`REG_SIZE:0] w_load_process;
```

```
531   reg [`REG_SIZE:0] w_load_shift;
532   wire [1:0] w_byte_offset = w_alu_res[1:0];
533
534   wire [7:0] byte0, byte1, byte2, byte3;
535   assign {byte3, byte2, byte1, byte0} = w_mem_read_data;
536   wire [15:0] half0, half1;
537   assign {half1, half0} = w_mem_read_data;
538
539   wire [31:0] lb0 = {{24{byte0[7]}}, byte0};
540   wire [31:0] lb1 = {{24{byte1[7]}}, byte1};
541   wire [31:0] lb2 = {{24{byte2[7]}}, byte2};
542   wire [31:0] lb3 = {{24{byte3[7]}}, byte3};
543   wire [31:0] lh0 = {{16{half0[15]}}, half0};
544   wire [31:0] lh1 = {{16{half1[15]}}, half1};
545
546   assign w_load_process = (w_funct3 == 3'b000) ? // LB
547                             ( w_byte_offset == 2'b00 ? lb0 :
548                               w_byte_offset == 2'b01 ? lb1 :
549                               w_byte_offset == 2'b10 ? lb2 : lb3) :
550                           (w_funct3 == 3'b001) ? // LH
551                             ( w_byte_offset == 2'b00 ? lh0 : lh1) :
552                           (w_funct3 == 3'b100) ? // LBU
553                             (w_byte_offset == 2'b00 ? {24'd0, byte0} :
554                             w_byte_offset == 2'b01 ? {24'd0, byte1} :
555                             w_byte_offset == 2'b10 ? {24'd0, byte2} :
                                {24'd0, byte3}) :
556                           (w_funct3 == 3'b101) ? // LHU
557                             (w_byte_offset == 2'b00 ? {16'd0, half0} :
                                 {16'd0, half1}) :
558                           w_mem_read_data; // LW
559
560 assign w_final_data = w_mem_read ? w_load_process : w_alu_res;
561
562   always @(posedge clk) begin
563     if (rst) begin
564       halt <= 0;
565       trace_writeback_pc <= 0;
566       trace_writeback_inst <= NOP;
567     end else begin
568       halt <= w_halt;
569       trace_writeback_pc <= w_pc;
570       trace_writeback_inst <= w_inst;
571     end
572   end
573 endmodule
574
575 module MemorySingleCycle #(
576     parameter NUM_WORDS = 512
577 ) (
578     input                    rst,                // rst for both imem
        and dmem
579     input                    clk,                // clock for both
       imem and dmem
```

```
580                                                    // The memory reads/
                                                          writes on @(
                                                          negedge clk)
581     input       [`REG_SIZE:0] pc_to_imem,          // must always be
           aligned to a 4B boundary
582     output reg  [`REG_SIZE:0] inst_from_imem,      // the value at
           memory location pc_to_imem
583     input       [`REG_SIZE:0] addr_to_dmem,        // must always be
           aligned to a 4B boundary
584     output reg  [`REG_SIZE:0] load_data_from_dmem, // the value at
           memory location addr_to_dmem
585     input       [`REG_SIZE:0] store_data_to_dmem,  // the value to be
           written to addr_to_dmem, controlled by store_we_to_dmem
586     // Each bit determines whether to write the corresponding byte of
           store_data_to_dmem to memory location addr_to_dmem.
587     // E.g., 4'b1111 will write 4 bytes. 4'b0001 will write only the
           least-significant byte.
588     input       [         3:0] store_we_to_dmem
589 );
590
591   reg [`REG_SIZE:0] mem_array[0:NUM_WORDS-1];
592
593   always @(negedge rst) begin
594     if (rst == 0)
595       $readmemh("mem_initial_contents.hex", mem_array, 0, NUM_WORDS-1)
             ;
596   end
597
598   localparam AddrMsb = $clog2(NUM_WORDS) + 1;
599   localparam AddrLsb = 2;
600
601   always @(negedge clk) begin
602     inst_from_imem <= mem_array[{pc_to_imem[AddrMsb:AddrLsb]}];
603   end
604
605   always @(negedge clk) begin
606     if (store_we_to_dmem[0]) begin
607       mem_array[addr_to_dmem[AddrMsb:AddrLsb]][7:0] <=
             store_data_to_dmem[7:0];
608     end
609     if (store_we_to_dmem[1]) begin
610       mem_array[addr_to_dmem[AddrMsb:AddrLsb]][15:8] <=
             store_data_to_dmem[15:8];
611     end
612     if (store_we_to_dmem[2]) begin
613       mem_array[addr_to_dmem[AddrMsb:AddrLsb]][23:16] <=
             store_data_to_dmem[23:16];
614     end
615     if (store_we_to_dmem[3]) begin
616       mem_array[addr_to_dmem[AddrMsb:AddrLsb]][31:24] <=
             store_data_to_dmem[31:24];
617     end
618     // dmem is "read-first": read returns value before the write
```

```
619     load_data_from_dmem <= mem_array[{addr_to_dmem[AddrMsb:AddrLsb]}];
620   end
621 endmodule
622
623 module Processor (
624     input                  clk,
625     input                  rst,
626     output                 halt,
627     output [ `REG_SIZE:0] trace_writeback_pc,
628     output [`INST_SIZE:0] trace_writeback_inst
629 );
630
631   wire [`INST_SIZE:0] inst_from_imem;
632   wire [ `REG_SIZE:0] pc_to_imem, mem_data_addr, mem_data_loaded_value
        , mem_data_to_write;
633   wire [         3:0] mem_data_we;
634
635   // This wire is set by cocotb to the name of the currently-running
        test, to make it easier
636   // to see what is going on in the waveforms.
637   wire [(8*32)-1:0] test_case;
638
639   MemorySingleCycle #(
640       .NUM_WORDS(8192)
641   ) memory (
642     .rst                 (rst),
643     .clk                 (clk),
644     // imem is read-only
645     .pc_to_imem          (pc_to_imem),
646     .inst_from_imem      (inst_from_imem),
647     // dmem is read-write
648     .addr_to_dmem        (mem_data_addr),
649     .load_data_from_dmem (mem_data_loaded_value),
650     .store_data_to_dmem  (mem_data_to_write),
651     .store_we_to_dmem    (mem_data_we)
652   );
653
654   DatapathPipelined datapath (
655     .clk                 (clk),
656     .rst                 (rst),
657     .pc_to_imem          (pc_to_imem),
658     .inst_from_imem      (inst_from_imem),
659     .addr_to_dmem        (mem_data_addr),
660     .store_data_to_dmem  (mem_data_to_write),
661     .store_we_to_dmem    (mem_data_we),
662     .load_data_from_dmem (mem_data_loaded_value),
663     .halt                (halt),
664     .trace_writeback_pc  (trace_writeback_pc),
665     .trace_writeback_inst (trace_writeback_inst)
666   );
667 endmodule
```

Program 1: RTL implementation of the DatapathPipelined file.

4 Testbench

The functionality of the pipelined processor was verified using a custom self-checking testbench (`tb_pipe.v`). This testbench is designed to validate the execution of the RV32I instruction set, including arithmetic, logic, branching, memory access and division operations.

- Task `load_inst`: The testbench writes machine code instruction directly into the DUT's instruction memory array (`dut.memory.mem_array`). This allows for rapid setup of specific instruciton sequences for testing.
- Task `check`: To verify correctness, the testbench reads the values directly from the Processor's Register File (`dut.datapath.rf.regs`). It compares the actual value in a specific register against the expected theoretical value.

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.

This testbench primarily serves as a sanity check to verify the basic functionality of RISC-V instructions. It ensures that operations behave as expected (for example, add performs addition rather than subtraction) and that register values are stored correctly. However, it does not provide detailed verification of pipeline timing or hazard handling. Instead, correctness is assessed manually by inspecting the simulation waveforms.

```
1  `timescale 1ns / 1ps
2  module tb_pipelined;
3  reg clk, rst;
4  wire halt;
5  wire [31:0] trace_pc, trace_inst;
6  integer i, test, timeout, pass, fail;
7
8  Processor dut (
9      .clk  (clk),
10     .rst  (rst),
11     .halt (halt),
12     .trace_writeback_pc (trace_pc),
13     .trace_writeback_inst (trace_inst)
14 );
15
16 task br;
17     $display ("%0s", {80{"="}});
18 endtask
19
20 task msg (input [700:0] txt);
21     begin
22         br();
23         $display ("%0s", txt);
```

```
24          br();
25      end
26  endtask
27
28  task load_inst (input [31:0] addr, input [31:0] inst);
29      begin
30          dut.memory.mem_array[addr >> 2] = inst;
31      end
32  endtask
33
34  task check (input [200:0] txt, input [4:0] reg_num, input [31:0] exp);
35      begin
36          if (dut.datapath.rf.regs[reg_num] === exp) begin
37              $display ("[PASS] Time = %0t | %s | Output x%0d = 0x%8h",
                    $time, txt, reg_num, dut.datapath.rf.regs[reg_num]);
38              pass = pass + 1;
39          end else begin
40              $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]);
41              fail = fail + 1;
42          end
43      end
44  endtask
45
46  initial begin
47      clk = 0;
48      forever #5 clk = ~clk;
49  end
50
51  initial begin
52      msg ("TEST BEGIN");
53      rst = 1;
54      timeout = 0;
55      pass = 0;
56      fail = 0;
57      test = 0;
58      repeat (5) @(posedge clk);
59      #1;
60      rst = 0;
61      #1;
62
63      msg ("TEST 1");
64      load_inst (32'h00, 32'h00a00093); // ADDI x1, x0, 10
65      load_inst (32'h04, 32'hffb00113); // ADDI x2, x0, -5
66      load_inst (32'h08, 32'h002081b3); // ADD x3, x1, x2 (5)
67      load_inst (32'h0C, 32'h40208233); // SUB x4, x1, x2 (15)
68      load_inst (32'h10, 32'h0020f2b3); // AND x5, x1, x2 (10)
69      load_inst (32'h14, 32'h0020e333); // OR x6, x1, x2 (-5)
70      load_inst (32'h18, 32'h0020c3b3); // XOR x7, x1, x2 (-15)
71      load_inst (32'h1C, 32'h00209413); // SLLI x8, x1, 2 (40)
72      load_inst (32'h20, 32'h40115493); // SRAI x9, x2, 1 (-3)
73      load_inst (32'h24, 32'h00115513); // SRLI x10, x2, 1
```

```
74      load_inst (32'h28, 32'h001125b3); // SLT x11, x2, x1 (1)
75      load_inst (32'h2C, 32'h00113633); // SLTU x12, x2, x1 (0)
76      load_inst (32'h30, 32'h021086b3); // MUL x13, x1, x1 (100)
77      load_inst (32'h34, 32'h0220c733); // DIV x14, x1, x2 (-2)
78      load_inst (32'h38, 32'h0220e7b3); // REM x15, x1, x2 (0)
79      load_inst (32'h3C, 32'h02115833); // DIVU x16, x2, x1
80      load_inst (32'h40, 32'h06302223); // SW x3, 100(x0)
81      load_inst (32'h44, 32'h06402903); // LW x18, 100(x0)
82      load_inst (32'h48, 32'h00108463); // BEQ x1, x1, 8
83      load_inst (32'h4C, 32'h06300993); // (Skipped)
84      load_inst (32'h50, 32'h00100993); // ADDI x19, x0, 1
85      load_inst (32'h54, 32'h00800a6f); // JAL x20, 8
86      load_inst (32'h58, 32'h06300a93); // (Skipped)
87      load_inst (32'h5C, 32'h00200a93); // ADDI x21, x0, 2
88      load_inst (32'h60, 32'h12345b37); // LUI x22, 0x12345
89      load_inst(32'h68, 32'h0050cb93);  // XORI x23, x1, 5
90      load_inst (32'h6C, 32'h00f0fc13); // ANDI x24, x1, 0xF
91      load_inst (32'h70, 32'hf006ec93); // ORI x25, x13, 0xFF00
92      load_inst (32'h74, 32'h0283ad13); // SLTI x26, x7, 0x28
93      load_inst (32'h78, 32'hff143d93); // SLTIU x27, x8, 0xfffffff1
94      load_inst (32'h7C, 32'h01309e33); // SLL x28, x1, x19
95      load_inst (32'h80, 32'h01335eb3); // SRL x29, x6, x19
96      load_inst (32'h84, 32'h41335f33); // SRA x30, x6, x19
97      load_inst (32'h8C, 32'h03b77fb3); // REMU x31, x14, x27
98      load_inst(32'h90, 32'h00000073); // HALT
99
100     timeout = 0;
101     pass = 0;
102     fail = 0;
103
104     while (!halt && timeout < 1000) begin
105         #10 timeout = timeout + 10;
106     end
107
108     if (test == 0) begin
109         check ("ADDI x1, x0, 10", 1, 10);
110         check ("ADDI x2, x0, -5", 2, -5);
111         check ("ADD x3, x1, x2", 3, 5);
112         check ("SUB x4, x1, x2", 4, 15);
113         check ("AND x5, x1, x2", 5, 10);
114         check ("OR x6, x1, x2", 6, -5);
115         check ("XOR x7, x1, x2", 7, -15);
116         check ("SLLI x8, x1, 2", 8, 40);
117         check ("SRAI x9, x2, 1", 9, -3);
118         check ("SRLI x10, x2, 1", 10, 32'h7fff_fffd);
119         check ("SLT x11, x2, x1", 11, 1);
120         check ("SLTU x12, x2, x1", 12, 0);
121         check ("MUL x13, x1, x1", 13, 100);
122         check ("DIV x14, x1, x2", 14, -2);
123         check ("REM x15, x1, x2", 15, 0);
124         check ("DIVU x16, x2, x1", 16, 32'h1999_9999);
125         check ("LW x18, 100(x0)", 18, 5);
126         check ("ADDI x19, x0, 1", 19, 1);
```

```
127         check ("JAL x20, 8", 20, 32'h58); // PC of next instruction
128         check ("ADDI x21, x0, 2", 21, 2);
129         check ("LUI x22, 0x12345", 22, 32'h1234_5000);
130         check ("XORI x23, x1, 5", 23, 32'hf);
131         check ("ANDI x24, x1, 0xF", 24, 32'ha);
132         check ("ORI x25, x13, 0xFF00", 25, 32'hffff_ff64);
133         check ("SLTI x26, x7, 0x28", 26, 1);
134         check ("SLTIU x27, x8, 0xFFFF_FFF1", 27, 1);
135         check ("SLL x28, x1, x19", 28, 20);
136         check ("SRL x29, x6, x19", 29, 32'h7FFF_FFFD);
137         check ("SRA x30, x6, x19", 30, 32'hFFFF_FFFD);
138         check ("REMU x31, x14, x27", 31, 0);
139
140         if (halt) begin
141             $display ("[PASS] HALT asserted.");
142             pass = pass + 1;
143         end else begin
144             $display ("[FAIL] HALT is missing.");
145             fail = fail + 1;
146         end
147     end
148
149     msg ("TEST 2 (REUSED REGISTERS)");
150     rst = 1;
151     repeat (5) @(posedge clk);
152     #1;
153     rst = 0;
154     #1;
155
156     load_inst (32'h0, 32'hffb00093); // ADDI x1, x0, 0xFFFF_FFFB
157     load_inst (32'h4, 32'h00300113); // ADDI x2, x0, 3
158     load_inst (32'h8, 32'h022091b3); // MULH x3, x1, x2
159     load_inst (32'hC, 32'h0220a233); // MULHSU x4, x1, x2
160     load_inst (32'h10, 32'h0220b2b3); // MULHU x5, x1, x2
161     load_inst (32'h14, 32'he9800313); // ADDI x6, x0, 0xABCD_FE98
162     load_inst (32'h18, 32'h06600223); // SB x6, 100(x0)
163     load_inst (32'h1C, 32'h06601423); // SH x6, 104(x0)
164     load_inst (32'h20, 32'h06400383); // LB x7, 100(x0)
165     load_inst (32'h24, 32'h06801403); // LH x8, 104(x0)
166     load_inst (32'h28, 32'h00831463); // BNE x6, x8, 8
167     load_inst (32'h2C, 32'h06904483); // LBU x9, 105(x0)
168     load_inst (32'h30, 32'h06805503); // LHU x10, 104(x0)
169     load_inst (32'h34, 32'h00a4c463); // BLT x9, x10, 12
170     load_inst (32'h38, 32'h00a4c5b3); // XOR x11, x9, x10 //skipped
171     load_inst (32'h3C, 32'h00a4e5b3); // OR x11, x9, x10 // skipped
172     load_inst (32'h40, 32'h00a485b3); // ADD x11, x9, x10
173     load_inst (32'h44, 32'h00535463); // BGE x6, x5, 8
174     load_inst (32'h48, 32'h0062e663); // BLTU x5, x6, 12
175     load_inst (32'h4C, 32'h00a4e633); // OR x12, x9, x10 // skip then
        turned back by bgeu
176     load_inst (32'h50, 32'h08200767); // JALR x14, 130(x0)
177     load_inst (32'h54, 32'h0004e633); // OR x12, x9, x0
178     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

    test = 1;
    timeout = 0;

    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
```

```
232 endmodule
```

Program 2: Testbench implementation for the Processor module.

5 Testbench results

All test cases executed successfully, with each marked as `PASSED`. This outcome confirms that the test module behaves as intended and that the implemented RTL design meets the expected functionality.

```
# ================================================================
# TEST BEGIN
# ================================================================
# ================================================================
# TEST 1
# ================================================================
# [PASS] Time = 547000 |            ADDI x1, x0, 10 | Output x1 = 0
   x0000000a
# [PASS] Time = 547000 |            ADDI x2, x0, -5 | Output x2 = 0
   xfffffffb
# [PASS] Time = 547000 |             ADD x3, x1, x2 | Output x3 = 0
   x00000005
# [PASS] Time = 547000 |             SUB x4, x1, x2 | Output x4 = 0
   x0000000f
# [PASS] Time = 547000 |             AND x5, x1, x2 | Output x5 = 0
   x0000000a
# [PASS] Time = 547000 |              OR x6, x1, x2 | Output x6 = 0
   xfffffffb
# [PASS] Time = 547000 |             XOR x7, x1, x2 | Output x7 = 0
   xfffffff1
# [PASS] Time = 547000 |            SLLI x8, x1, 2 | Output x8 = 0
   x00000028
# [PASS] Time = 547000 |            SRAI x9, x2, 1 | Output x9 = 0
   xfffffffd
# [PASS] Time = 547000 |           SRLI x10, x2, 1 | Output x10 = 0
   x7fffffd
# [PASS] Time = 547000 |           SLT x11, x2, x1 | Output x11 = 0
   x00000001
# [PASS] Time = 547000 |          SLTU x12, x2, x1 | Output x12 = 0
   x00000000
# [PASS] Time = 547000 |           MUL x13, x1, x1 | Output x13 = 0
   x00000064
# [PASS] Time = 547000 |           DIV x14, x1, x2 | Output x14 = 0
   xfffffffe
# [PASS] Time = 547000 |           REM x15, x1, x2 | Output x15 = 0
   x00000000
# [PASS] Time = 547000 |          DIVU x16, x2, x1 | Output x16 = 0
   x19999999
# [PASS] Time = 547000 |           LW x18, 100(x0) | Output x18 = 0
   x00000005
# [PASS] Time = 547000 |           ADDI x19, x0, 1 | Output x19 = 0
   x00000001
# [PASS] Time = 547000 |                JAL x20, 8 | Output x20 = 0
   x00000058
# [PASS] Time = 547000 |           ADDI x21, x0, 2 | Output x21 = 0
   x00000002
# [PASS] Time = 547000 |          LUI x22, 0x12345 | Output x22 = 0
   x12345000
```

```
# [PASS] Time = 547000 |            XORI x23, x1, 5 | Output x23 = 0
   x0000000f
# [PASS] Time = 547000 |          ANDI x24, x1, 0xF | Output x24 = 0
   x0000000a
# [PASS] Time = 547000 |       ORI x25, x13, 0xFF00 | Output x25 = 0
   xffffff64
# [PASS] Time = 547000 |         SLTI x26, x7, 0x28 | Output x26 = 0
   x00000001
# [PASS] Time = 547000 |  LTIU x27, x8, 0xFFFF_FFF1 | Output x27 = 0
   x00000001
# [PASS] Time = 547000 |           SLL x28, x1, x19 | Output x28 = 0
   x00000014
# [PASS] Time = 547000 |           SRL x29, x6, x19 | Output x29 = 0
   x7fffffd
# [PASS] Time = 547000 |           SRA x30, x6, x19 | Output x30 = 0
   xfffffffd
# [PASS] Time = 547000 |         REMU x31, x14, x27 | Output x31 = 0
   x00000000
# [PASS] HALT asserted.
# ================================================================
# TEST 2 (REUSED REGISTERS)
# ================================================================
# [PASS] Time = 987000 |   ADDI x1, x0, 0xFFFF_FFFB | Output x1 = 0
   xfffffffb
# [PASS] Time = 987000 |             ADDI x2, x0, 3 | Output x2 = 0
   x00000003
# [PASS] Time = 987000 |            MULH x3, x1, x2 | Output x3 = 0
   xffffffff
# [PASS] Time = 987000 |          MULHSU x4, x1, x2 | Output x4 = 0
   xffffffff
# [PASS] Time = 987000 |           MULHU x5, x1, x2 | Output x5 = 0
   x00000002
# [PASS] Time = 987000 |   ADDI x6, x0, 0xABCD_FE98 | Output x6 = 0
   xfffffe98
# [PASS] Time = 987000 |             LB x7, 100(x0) | Output x7 = 0
   xffffff98
# [PASS] Time = 987000 |             LH x8, 104(x0) | Output x8 = 0
   xfffffe98
# [PASS] Time = 987000 |            LBU x9, 105(x0) | Output x9 = 0
   x000000fe
# [PASS] Time = 987000 |           LHU x10, 104(x0) | Output x10 = 0
   x0000fe98
# [PASS] Time = 987000 |           ADD x11, x9, x10 | Output x11 = 0
   x0000ff96
# [PASS] Time = 987000 |            OR x12, x9, x10 | Output x12 = 0
   x0000fefe
# [PASS] Time = 987000 |            LW x13, 120(x0) | Output x13 = 0
   x000000fe
# [PASS] Time = 987000 |          JALR x14, 130(x0) | Output x14 = 0
   x00000054
# [PASS] HALT asserted.
# ================================================================
# SUMMARY
```

```
58 # ================================================================
59 # TOTAL TESTS : 46
60 # TOTAL PASSED: 46
61 # TOTAL FAILED: 0
62 # ======> ALL TESTS PASSED
63 # ================================================================
64 # TEST END
65 # ================================================================
66 # ** Note: $finish    : ../../tb/tb_pipe.v(232)
67 #    Time: 1087 ns  Iteration: 0  Instance: /tb_pipelined
```

6 Simulation waveform

To demonstrate the correct operation of the 5-stage pipeline, critical moments during the simulation was captured and explain in details.

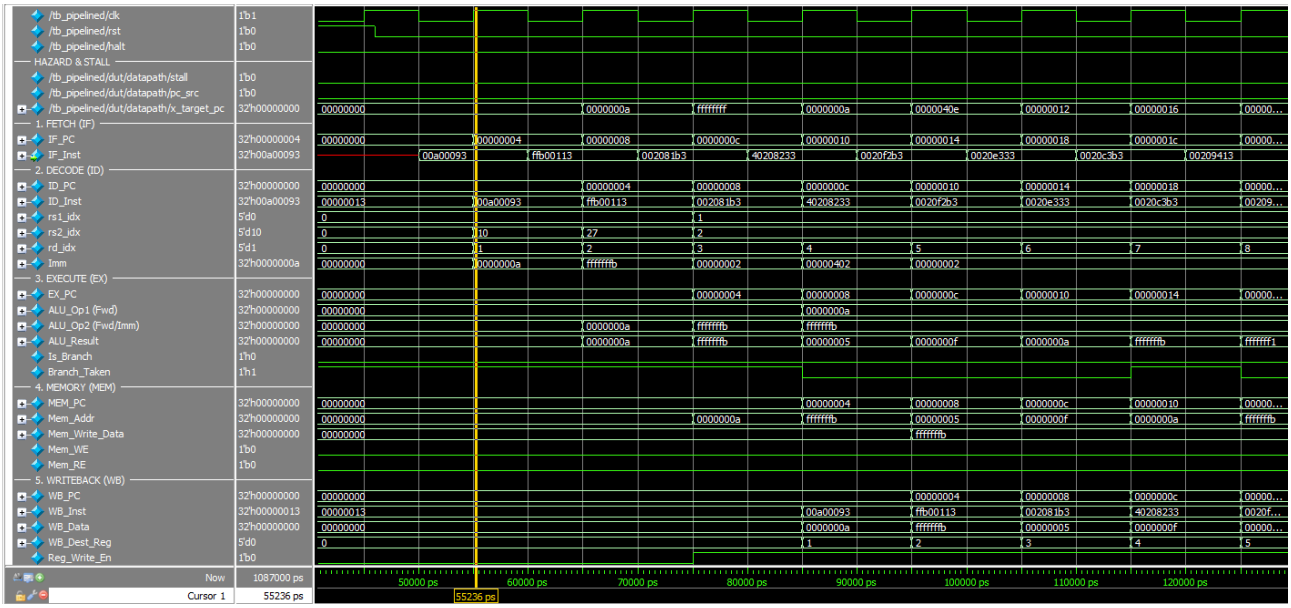

Figure 1

Figure 1 demonstrates the pipeline's ability to resolve multiple Read-After-Write (RAW) data hazards simultaneously using both Memory-to-Execute (MX) and Writeback-to-Execute (WX) bypassing mechanisms. The simulation runs a sequence of instructions where ADDI x1, x0, 10 (PC 0x00) and ADDI x2, x0, -5 (PC 0x04) are immediately followed by ADD x3, x1, x2 (PC 0x08), which requires reading both x1 and x2.

At approximately 85,000 ps, the ADD instruction has successfully entered the Execute stage (EX_PC reads 0x08). Simultaneously, the ADDI x2 instruction, which produces the second operand, has moved to the Memory stage (MEM_PC reads 0x04), while the ADDI x1 instruction, which produces the first operand, has reached the Writeback stage (WB_PC reads 0x00). Crucially, because the ADDI x2 instruction has not yet reached the Writeback stage, the value of register x2 inside the physical Register File is technically outdated. However, the waveform confirms that the hazard logic functions correctly: the signal ALU_Op2 (Fwd/Imm) displays the correct value 0xFFFFFFFB (-5) instead of the old register value. This proves that the forwarding multiplexers successfully detected the dependency (matching rs2 in Execute with rd in Memory) and bypassed the data directly from the Memory stage via the MX Bypass. Additionally, ALU_Op1 correctly reads 0x0000000A (10), confirming the WX Bypass from the Writeback stage. Consequently, the ALU_Result is correctly computed as 0x5 (10 + -5) in the same clock cycle, allowing the pipeline to maintain optimal throughput without inserting stall cycles.

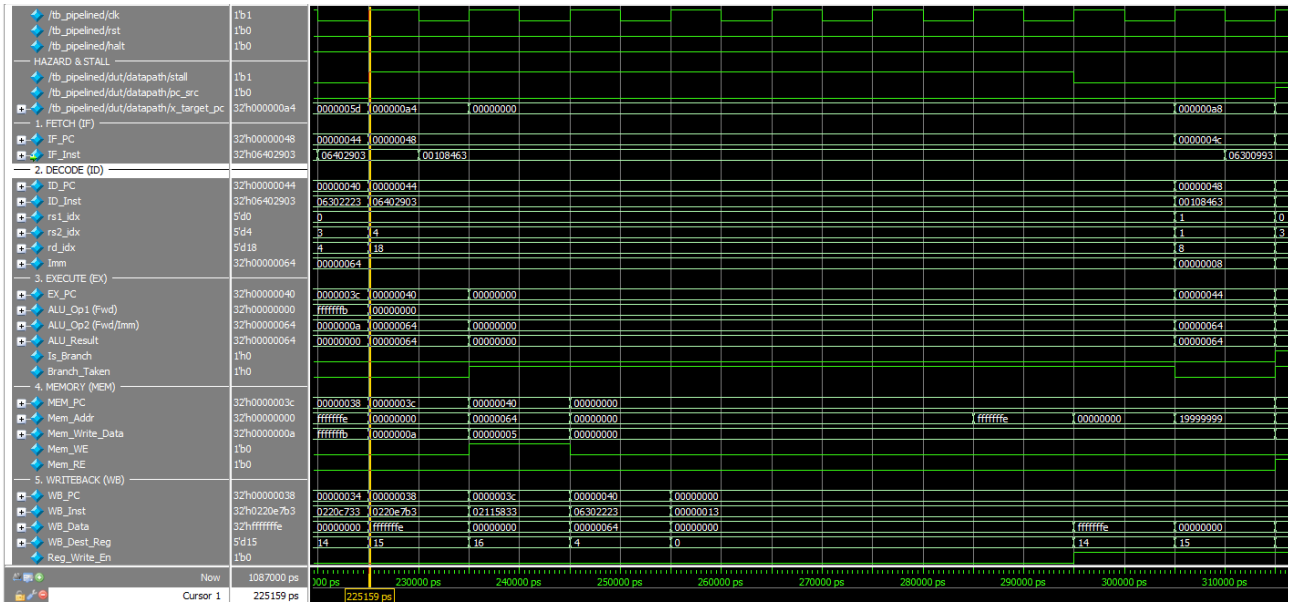

Figure 2

Figure 2 demonstrates the pipeline's handling of a structural hazard and the resulting "ripple effect" delay passed from a LW instruction to a subsequent BEQ instruction. The simulation executes a sequence where LW x18, 100(x0) (instruction 0x06402903 at PC 0x44) is immediately followed by BEQ x1, x1, 8 (instruction 0x00108463 at PC 0x48).

At approximately 225,159 ps, the LW instruction attempts to pass through the Decode stage. However, the logic detects a structural conflict: the pipelined divider is currently busy processing a previous DIVU instruction, and the design rules prevent any new instruction that writes to a register (like LW) from proceeding to prevent potential write-back conflicts. Consequently, the stall signal is asserted High.

Crucially, this stall creates a direct temporal relationship between the two instructions:

- The blocker (LW): Because of the stall, the LW instruction is frozen in the Decode stage (ID_PC remains 0x44). It cannot advance to Execute.
- The victim (BEQ): The BEQ instruction, which has already been fetched and is waiting in the Fetch stage (IF_PC is 0x48), is forcibly blocked from entering the Decode stage.

The waveform confirms this inter-instruction dependency: the BEQ instruction is held in the Fetch buffer solely because its predecessor LW was stalled. Once the hazard resolves (DIVU finishes executing in 8 cycles), the stall signal drops, allowing LW to advance to Execute and finally permitting BEQ to enter the Decode stage, demonstrating how hazards on a single instruction propagate delays to all subsequent instructions in the pipeline.

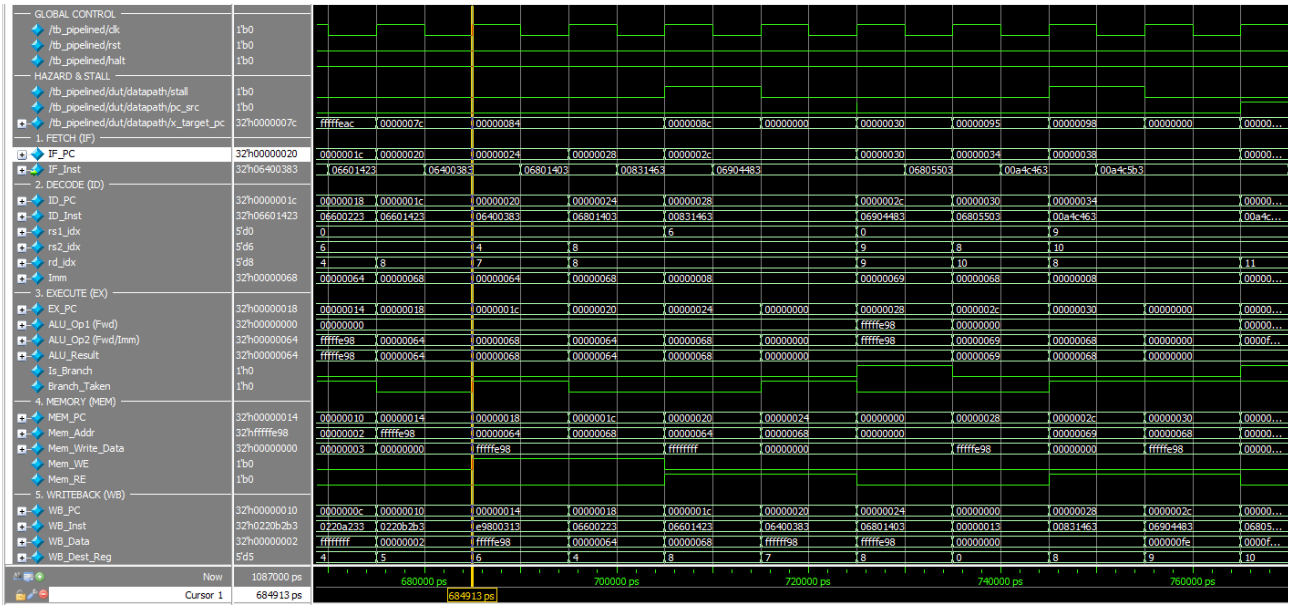

Figure 3

Figure 3 demonstrates the pipeline's ability to resolve a Load-Use data hazard by inserting a stall cycle. The simulation executes a dependency sequence starting with LH x8, 104(x0) (instruction 0x06801403 at PC 0x24), which loads a value into register x8. This is immediately followed by BNE x6, x8, 8 (instruction 0x00831463 at PC 0x28), which attempts to read x8 to determine the branch outcome.

At approximately 704,000 ps, the LH instruction successfully moves to the Execute (EX) stage, and the BNE instruction enters the Decode (ID) stage (ID_PC becomes 0x28). Crucially, the hazard detection unit identifies a conflict: the instruction in Execute (LH) is a memory load (Mem_Read is active) and its destination (rd = 8) matches the source operand (rs2 = 8) of the instruction in Decode. Since the memory data is not yet available for forwarding, the pipeline cannot proceed. The waveform confirms the correct hazard resolution: the stall signal asserts High for exactly one clock cycle. Consequently, the Program Counter (ID_PC) freezes at 0x28 and the ID_Inst holds the BNE instruction constant, effectively pausing the Decode stage. Simultaneously, a NOP (bubble) is inserted into the Execute stage for the next cycle (visible where EX_PC would normally update). Once the LH instruction advances to the Memory stage (where forwarding becomes possible), the stall signal de-asserts, allowing the BNE instruction to proceed to Execute at approximately 724,000 ps, thereby preserving data integrity.

7 Timing closure and Resource utilization

7.1 Timing closure

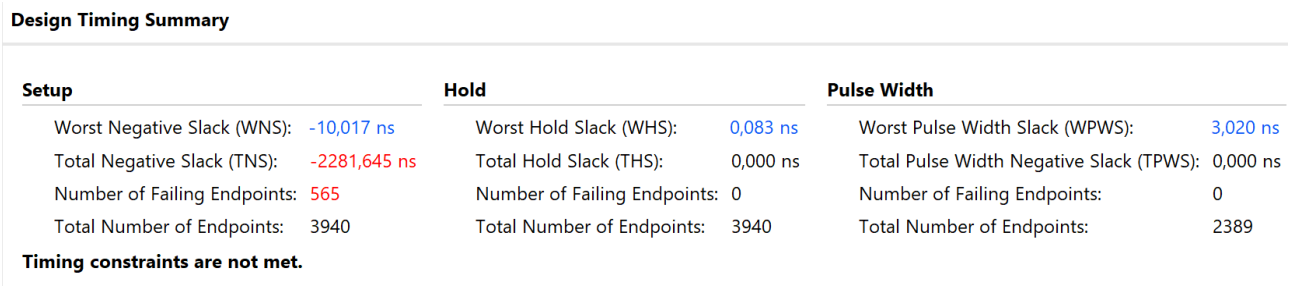

Figure : Timing summary.

After synthesizing and implementing the pipelined datapath design on the Arty Z7-20 FPGA, we obtained the following timing results:

- Target clock frequency: 125 MHz (Period: 8.000 ns)
- Worst Negative Slack (WNS): -10.017 ns.
- Conclusion: The design has not achieved Timing closure (Timing constraints are not met).

Based on the WNS, the maximum operating frequency (F_max) is calculated as follows:

$$T\_min = T\_target - WNS = 8.000 - (-10.017) = 18.017ns$$

$$F_{max} = \frac{1}{T\_min} = \frac{1}{18.017} \approx 55.5MHz$$

Based on the calculation, the processor cannot run at the requested 125 MHz, but it can safely operate at approximately 55 MHz. Ideally, pipelining increases frequency by dividing a large combination block (like the entire CPU) into smaller stages. However, our results show a decrease in slack (worse timing) compared to the multi-cycle design (-9.451 ns). This is caused by hazard management overhead.

In the multi-cycle design, the ALU operands are read directly from the register file or temporary registers. However, in the pipelined design, the ALU inputs depend on complex comparison logic (Forwarding) that must settle within the same clock cycles as the ALU operations. Moreover, the Forwarding (bypassing) logic is inserted directly into the critical path of the Execute stage. This added logic depth exceeds the time savings gained by splitting the stages, resulting in a lower F_max, even though the Instructions Per Cycle (IPC) has increased by 5 times.

In the Execution stage, the critical path has been lengthened by:

- Comparators: The logic must first compare souce registers (rs1, rs2) with destination registers in Memory (`m_rd`) and Writeback (`w_rd`) stages.
- Forwarding muxes: Large multiplexers select between the Register File value, the Memory Forwarded value or the Writeback Forwarded value.
- ALU operation: Only after the forwarding decision is made can the ALU (CLA adder/-Multiplier) begin its calculation.

$$Delay_{Pipe\_Ex} = Delay_{Fowarding\_Mux} + Delay_{ALU}$$

This Forwarding Mux delay exists only in the pipelined version. Since the ALU itself (especially the Adder/Multiplier) is already a deep combinational circuit, adding the forwarding logic in series with it increases the total propagation delay of the Execute stage beyond the delay of the multi-cycle's Execute state.

While the Pipelined design likely achieves higher instruction throughput compared to the multi-cycle design, the stage latency (and thus the maximum clock frequency) is slightly reduced due to the complexity of the data forwarding paths required to resolve hazards.

7.2 Utilization

Resource	Utilization	Available	Utilization %
LUT	4625	53200	8.69
FF	2300	106400	2.16
BRAM	8	140	5.71
DSP	12	220	5.45
IO	67	125	53.60

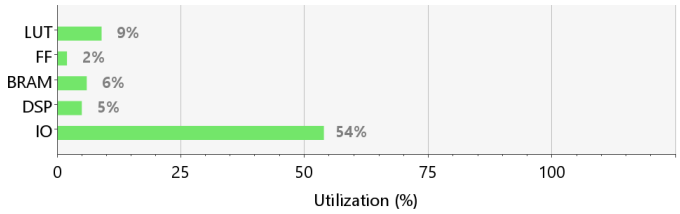

Figure : Utilization summary.

Compared to the multi-cycle datapath, the results for LUT, Flip-Flop, BRAM and DSP of the pipelined datapath are nearly similar, by rounding up they are both 9%, 2%, 6% and 5% respectively. Meanwhile, the IO significantly increased to approximately 54%. The reason behind this number is because the `Processor` module outputs `trace_writeback_pc` and `trace_writeback_inst` (32 bits), plus `halt, clk,` and `rst`. As a total we have 67 pins. If we eliminates these debugging signals, the design fits easily within the Arty Z7-20 FPGA.

8 Conclusion

8.1 Functional verification

The primary objective of this Assignment is to design and implement a 5-stage pipelined RISC-V processor, which has been successfully met. The design was rigorously verified using the provided testbenches and self-test testbench (`tb_pipe.v`), covering a comprehensive set of instruction types including arithmetic, logic, memory access and branching. The simulation results confirm that the hazard detection unit and forwarding logic function correctly.

8.2 Performance analysis and Architecture limitations

While the design is functionally robust, the timing analysis indicates a Maximum Frequency (F_max) of approximately 55 MHz, which falls short of the theoretical target of 125 MHz, which often desired for high-performance FPGA designs.

This frequency limitation is not due to a coding error, but rather a fundamental constraint of the standard 5-stage architecture on this FPGA fabric:

- Critical path depth: The Execute stage currently contains the longest critical path. In a single clock cycle, signals must propagate through the Forwarding Multiplexers, into the ALU (which includes wide adders and shifters), and finally to the pipeline register.
- Forwarding complexity: As shown in the synthesis results, the addition of forwarding logic places complex combinational gates in series with the ALU. This effectively increases the logic depth compared to a non-pipelined or non-forwarded design, reducing the available timing slack.

8.3 Future optimization strategy

To achieve a target frequency of 125 MHz or higher, simply optimizing the Verilog code is insufficient. A significant architectural shift would be required, which was beyond the scope of the current timeline:

- Super-pipelining: The Execute stage could be decomposed into two or more sub-stages. For example, Execute_1 for operand selection/fowarding and Execute_2 for ALU computation. This would cut the critical path in half, potentially doubling the frequency.
- Retiming: Moving logic across pipeline boundaries to balance the delay between stagges.

In summary, this project demonstrates a fully functional, hazard-safe pipelined processor. While the current 5-stage depth limits the clock speed to approximately 55 MHz.