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

LAB 4
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 4 project files (RTL source code, testbenches, simulation scripts) are located in the `Lab04` folder. They are available at the following repository: https://github.com/trxmhoang/SoC.git

2 RTL implementation

2.1 DividerUnsignedPipelined file

The `DividerUnsignedPipelined` module implements a high-throughtput divider using an 8-stage pipeline. Instead of calculating the full 32-bit division in a single cycle, which would require a long critical path, the workload is distributed across 8 clock cycles. Each stage processes 4 bits of the quotient using a chain of `divu_1iter` sub-modules. Registers (`r_dividend`, `r_divisor`, etc) are placed between each stage to store intermediate results, allowing the divider to accept new inputs every clock cycle while producing a result with a latency of 8 cycles.

```
`timescale 1ns / 1ns

// quotient = dividend / divisor

module DividerUnsignedPipelined (
    input              clk, rst, stall,
    input      [31:0] i_dividend,
    input      [31:0] i_divisor,
    output reg [31:0] o_remainder,
    output reg [31:0] o_quotient
);

  // TODO: your code here
reg [31:0] r_dividend [0:6];
reg [31:0] r_divisor [0:6];
reg [31:0] r_quotient [0:6];
reg [31:0] r_remainder [0:6];

// STAGE 0
wire [31:0] s0_dividend [0:4];
wire [31:0] s0_quotient [0:4];
wire [31:0] s0_remainder [0:4];

assign s0_dividend[0] = i_dividend;
assign s0_quotient[0] = 32'b0;
assign s0_remainder[0] = 32'b0;

genvar i, j;
generate
    for (j = 0; j < 4; j = j + 1) begin : stage0_loop
        divu_1iter u_div0 (
            .i_dividend  (s0_dividend[j]),
            .i_divisor   (i_divisor),
            .i_remainder (s0_remainder[j]),
            .i_quotient  (s0_quotient[j]),
            .o_dividend  (s0_dividend[j + 1]),
```

```
            .o_remainder (s0_remainder[j + 1]),
            .o_quotient  (s0_quotient[j + 1])
        );
    end
endgenerate

always @(posedge clk) begin
    if (rst) begin
        r_dividend[0] <= 32'b0;
        r_divisor[0] <= 32'b0;
        r_quotient[0] <= 32'b0;
        r_remainder[0] <= 32'b0;
    end else if (!stall) begin
        r_dividend[0] <= s0_dividend[4];
        r_divisor[0] <= i_divisor;
        r_quotient[0] <= s0_quotient[4];
        r_remainder[0] <= s0_remainder[4];
    end
end

// STAGE 1 - STAGE 6
generate
    for (i = 1; i < 7; i = i + 1) begin : pipe_loop
        wire [31:0] chain_dividend [0:4];
        wire [31:0] chain_quotient [0:4];
        wire [31:0] chain_remainder [0:4];
        wire [31:0] stage_divisor;

        assign chain_dividend[0] = r_dividend[i - 1];
        assign chain_quotient[0] = r_quotient[i - 1];
        assign chain_remainder[0] = r_remainder[i - 1];
        assign stage_divisor = r_divisor[i - 1];

        for (j = 0; j < 4; j = j + 1) begin : iter_loop
            divu_1iter u_div (
                .i_dividend  (chain_dividend[j]),
                .i_divisor   (stage_divisor),
                .i_remainder (chain_remainder[j]),
                .i_quotient  (chain_quotient[j]),
                .o_dividend  (chain_dividend[j + 1]),
                .o_remainder (chain_remainder[j + 1]),
                .o_quotient  (chain_quotient[j + 1])
            );
        end

        always @(posedge clk) begin
            if (rst) begin
                r_dividend[i] <= 32'b0;
                r_divisor[i] <= 32'b0;
                r_quotient[i] <= 32'b0;
                r_remainder[i] <= 32'b0;
            end else if (!stall) begin
                r_dividend[i] <= chain_dividend[4];
```

```
                r_divisor[i] <= stage_divisor;
                r_quotient[i] <= chain_quotient[4];
                r_remainder[i] <= chain_remainder[4];
            end
        end
    end
endgenerate

// STAGE 7
wire [31:0] s7_dividend [0:4];
wire [31:0] s7_quotient [0:4];
wire [31:0] s7_remainder [0:4];

assign s7_dividend[0] = r_dividend[6];
assign s7_quotient[0] = r_quotient[6];
assign s7_remainder[0] = r_remainder[6];

generate
    for (j = 0; j < 4; j = j + 1) begin : stage7_loop
        divu_1iter u_div7 (
            .i_dividend  (s7_dividend[j]),
            .i_divisor   (r_divisor[6]),
            .i_remainder (s7_remainder[j]),
            .i_quotient  (s7_quotient[j]),
            .o_dividend  (s7_dividend[j + 1]),
            .o_remainder (s7_remainder[j + 1]),
            .o_quotient  (s7_quotient[j + 1])
        );
    end
endgenerate

always @(posedge clk) begin
    if (rst) begin
        o_quotient <= 32'b0;
        o_remainder <= 32'b0;
    end else if (!stall) begin
        o_quotient <= s7_quotient[4];
        o_remainder <= s7_remainder[4];
    end
end
endmodule

module divu_1iter (
    input       [31:0] i_dividend,
    input       [31:0] i_divisor,
    input       [31:0] i_remainder,
    input       [31:0] i_quotient,
    output reg  [31:0] o_dividend,
    output reg  [31:0] o_remainder,
    output reg  [31:0] o_quotient
);

wire [31:0] new_remainder, subtract;
```

```
assign new_remainder = {i_remainder[30:0], i_dividend[31]};
assign subtract = new_remainder - i_divisor;

always @(*) begin
  o_dividend = i_dividend << 1;
  if (new_remainder >= i_divisor) begin
    o_remainder = subtract;
    o_quotient = {i_quotient[30:0], 1'b1};
  end else begin
    o_remainder = new_remainder;
    o_quotient = {i_quotient[30:0], 1'b0};
  end
end
endmodule
```

Program 1: RTL implementation of the DividerUnsignedPipelined file.

2.2 DatapathMultiCycle file

The `DatapathMulticycle` file is an enhanced version of the previous `DatapathSingleCycle` file. While the core instruction decoding and execution logic remain similar, this module integrates the 8-stage pipeplined divider to handle division operations (DIV, DIVU, REM, REMU). This change allows the processor to manage long-latency arithmetic operations without halting the entire system permanently, although in this initial integration, the processor stalls to wait for the result.

Since the new divider takes 8 clock cycles to compute a result, the processor cannot proceed to the next instruction immediately when a division opcode is detected. A stall counter mechanism was added to freeze the Program Counter (PC) and the instruction fetch process.

- Logic: When a division-type instruction (`is_div_op`) is detected, a counter increments from 0 to 8.
- Stall signal (`stall_en`): This signal remains high while `stall_counter` < 8, effectively pausing the PC update and instruction fetch.
- Resume: Once the counter reaches 8, the divider's output is valid. The stall signal goes low, allowing the result to be written to the register file (`rf_we` enables) and the PC to advance to the next instruction.

```
/* INSERT NAME AND PENNKEY HERE */

`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 CLA and Divider
/* `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
);

  // TODO: copy your homework #3 code here
  localparam NumRegs = 32;
  reg [`REG_SIZE:0] regs[0:NumRegs-1];
  integer i;

  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 DatapathMultiCycle (
    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
);

  // TODO: your code here (largely based on homework #3)
  // 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;
```

```
68  wire [`OPCODE_SIZE:0] inst_opcode;
69
70  // split R-type instruction - see section 2.2 of RiscV spec
71  assign {inst_funct7, inst_rs2, inst_rs1, inst_funct3, inst_rd,
        inst_opcode} = inst_from_imem;
72
73  // setup for I, S, B & J type instructions
74  // I - short immediates and loads
75  wire [11:0] imm_i;
76  assign imm_i = inst_from_imem[31:20];
77  wire [ 4:0] imm_shamt = inst_from_imem[24:20];
78
79  // S - stores
80  wire [11:0] imm_s;
81  assign imm_s = {inst_funct7, inst_rd};
82
83  // B - conditionals
84  wire [12:0] imm_b;
85  assign {imm_b[12], imm_b[10:1], imm_b[11], imm_b[0]} = {inst_funct7,
        inst_rd, 1'b0};
86
87  // J - unconditional jumps
88  wire [20:0] imm_j;
89  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};
90
91  wire [`REG_SIZE:0] imm_i_sext = {{20{imm_i[11]}}, imm_i[11:0]};
92  wire [`REG_SIZE:0] imm_s_sext = {{20{imm_s[11]}}, imm_s[11:0]};
93  wire [`REG_SIZE:0] imm_b_sext = {{19{imm_b[12]}}, imm_b[12:0]};
94  wire [`REG_SIZE:0] imm_j_sext = {{11{imm_j[20]}}, imm_j[20:0]};
95
96  // opcodes - see section 19 of RiscV spec
97  localparam [`OPCODE_SIZE:0] OpLoad    = 7'b00_000_11;
98  localparam [`OPCODE_SIZE:0] OpStore   = 7'b01_000_11;
99  localparam [`OPCODE_SIZE:0] OpBranch  = 7'b11_000_11;
100 localparam [`OPCODE_SIZE:0] OpJalr    = 7'b11_001_11;
101 localparam [`OPCODE_SIZE:0] OpMiscMem = 7'b00_011_11;
102 localparam [`OPCODE_SIZE:0] OpJal     = 7'b11_011_11;
103
104 localparam [`OPCODE_SIZE:0] OpRegImm  = 7'b00_100_11;
105 localparam [`OPCODE_SIZE:0] OpRegReg  = 7'b01_100_11;
106 localparam [`OPCODE_SIZE:0] OpEnviron = 7'b11_100_11;
107
108 localparam [`OPCODE_SIZE:0] OpAuipc   = 7'b00_101_11;
109 localparam [`OPCODE_SIZE:0] OpLui     = 7'b01_101_11;
110
111 wire inst_lui    = (inst_opcode == OpLui    );
112 wire inst_auipc  = (inst_opcode == OpAuipc  );
113 wire inst_jal    = (inst_opcode == OpJal    );
114 wire inst_jalr   = (inst_opcode == OpJalr   );
115
116 wire inst_beq    = (inst_opcode == OpBranch ) & (inst_from_imem
        [14:12] == 3'b000);
```

```
117   wire inst_bne    = (inst_opcode == OpBranch ) & (inst_from_imem
          [14:12] == 3'b001);
118   wire inst_blt    = (inst_opcode == OpBranch ) & (inst_from_imem
          [14:12] == 3'b100);
119   wire inst_bge    = (inst_opcode == OpBranch ) & (inst_from_imem
          [14:12] == 3'b101);
120   wire inst_bltu   = (inst_opcode == OpBranch ) & (inst_from_imem
          [14:12] == 3'b110);
121   wire inst_bgeu   = (inst_opcode == OpBranch ) & (inst_from_imem
          [14:12] == 3'b111);
122
123   wire inst_lb     = (inst_opcode == OpLoad   ) & (inst_from_imem
          [14:12] == 3'b000);
124   wire inst_lh     = (inst_opcode == OpLoad   ) & (inst_from_imem
          [14:12] == 3'b001);
125   wire inst_lw     = (inst_opcode == OpLoad   ) & (inst_from_imem
          [14:12] == 3'b010);
126   wire inst_lbu    = (inst_opcode == OpLoad   ) & (inst_from_imem
          [14:12] == 3'b100);
127   wire inst_lhu    = (inst_opcode == OpLoad   ) & (inst_from_imem
          [14:12] == 3'b101);
128
129   wire inst_sb     = (inst_opcode == OpStore  ) & (inst_from_imem
          [14:12] == 3'b000);
130   wire inst_sh     = (inst_opcode == OpStore  ) & (inst_from_imem
          [14:12] == 3'b001);
131   wire inst_sw     = (inst_opcode == OpStore  ) & (inst_from_imem
          [14:12] == 3'b010);
132
133   wire inst_addi   = (inst_opcode == OpRegImm ) & (inst_from_imem
          [14:12] == 3'b000);
134   wire inst_slti   = (inst_opcode == OpRegImm ) & (inst_from_imem
          [14:12] == 3'b010);
135   wire inst_sltiu  = (inst_opcode == OpRegImm ) & (inst_from_imem
          [14:12] == 3'b011);
136   wire inst_xori   = (inst_opcode == OpRegImm ) & (inst_from_imem
          [14:12] == 3'b100);
137   wire inst_ori    = (inst_opcode == OpRegImm ) & (inst_from_imem
          [14:12] == 3'b110);
138   wire inst_andi   = (inst_opcode == OpRegImm ) & (inst_from_imem
          [14:12] == 3'b111);
139
140   wire inst_slli   = (inst_opcode == OpRegImm ) & (inst_from_imem
          [14:12] == 3'b001) & (inst_from_imem[31:25] == 7'd0      );
141   wire inst_srli   = (inst_opcode == OpRegImm ) & (inst_from_imem
          [14:12] == 3'b101) & (inst_from_imem[31:25] == 7'd0      );
142   wire inst_srai   = (inst_opcode == OpRegImm ) & (inst_from_imem
          [14:12] == 3'b101) & (inst_from_imem[31:25] == 7'b0100000);
143
144   wire inst_add    = (inst_opcode == OpRegReg ) & (inst_from_imem
          [14:12] == 3'b000) & (inst_from_imem[31:25] == 7'd0      );
145   wire inst_sub    = (inst_opcode == OpRegReg ) & (inst_from_imem
          [14:12] == 3'b000) & (inst_from_imem[31:25] == 7'b0100000);
```

```
146   wire inst_sll    = (inst_opcode == OpRegReg ) & (inst_from_imem
        [14:12] == 3'b001) & (inst_from_imem[31:25] == 7'd0      );
147   wire inst_slt    = (inst_opcode == OpRegReg ) & (inst_from_imem
        [14:12] == 3'b010) & (inst_from_imem[31:25] == 7'd0      );
148   wire inst_sltu   = (inst_opcode == OpRegReg ) & (inst_from_imem
        [14:12] == 3'b011) & (inst_from_imem[31:25] == 7'd0      );
149   wire inst_xor    = (inst_opcode == OpRegReg ) & (inst_from_imem
        [14:12] == 3'b100) & (inst_from_imem[31:25] == 7'd0      );
150   wire inst_srl    = (inst_opcode == OpRegReg ) & (inst_from_imem
        [14:12] == 3'b101) & (inst_from_imem[31:25] == 7'd0      );
151   wire inst_sra    = (inst_opcode == OpRegReg ) & (inst_from_imem
        [14:12] == 3'b101) & (inst_from_imem[31:25] == 7'b0100000);
152   wire inst_or     = (inst_opcode == OpRegReg ) & (inst_from_imem
        [14:12] == 3'b110) & (inst_from_imem[31:25] == 7'd0      );
153   wire inst_and    = (inst_opcode == OpRegReg ) & (inst_from_imem
        [14:12] == 3'b111) & (inst_from_imem[31:25] == 7'd0      );
154
155   wire inst_mul    = (inst_opcode == OpRegReg ) & (inst_from_imem
        [31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b000    );
156   wire inst_mulh   = (inst_opcode == OpRegReg ) & (inst_from_imem
        [31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b001    );
157   wire inst_mulhsu = (inst_opcode == OpRegReg ) & (inst_from_imem
        [31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b010    );
158   wire inst_mulhu  = (inst_opcode == OpRegReg ) & (inst_from_imem
        [31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b011    );
159   wire inst_div    = (inst_opcode == OpRegReg ) & (inst_from_imem
        [31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b100    );
160   wire inst_divu   = (inst_opcode == OpRegReg ) & (inst_from_imem
        [31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b101    );
161   wire inst_rem    = (inst_opcode == OpRegReg ) & (inst_from_imem
        [31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b110    );
162   wire inst_remu   = (inst_opcode == OpRegReg ) & (inst_from_imem
        [31:25] == 7'd1  ) & (inst_from_imem[14:12] == 3'b111    );
163
164   wire inst_ecall  = (inst_opcode == OpEnviron) & (inst_from_imem
        [31:7] == 25'd0  );
165   wire inst_fence  = (inst_opcode == OpMiscMem);
166
167   // stall
168   reg [3:0] stall_counter;
169   wire stall_en;
170   wire is_div_op = inst_div | inst_divu | inst_rem | inst_remu;
171
172   always @(posedge clk) begin
173     if (rst) begin
174       stall_counter <= 0;
175     end else begin
176       if (is_div_op) begin
177         if (stall_counter < 8) begin
178           stall_counter <= stall_counter + 1;
179         end else begin
180           stall_counter <= 0;
181         end
```

```
182       end else begin
183         stall_counter <= 0;
184       end
185     end
186   end
187
188   assign stall_en = is_div_op & (stall_counter < 8);
189
190   // program counter
191   reg [`REG_SIZE:0] pcNext, pcCurrent;
192   always @(posedge clk) begin
193     if (rst) begin
194       pcCurrent <= 32'd0;
195     end else if (!stall_en) begin
196       pcCurrent <= pcNext;
197     end
198   end
199   assign pc_to_imem = pcCurrent;
200
201   // cycle/inst._from_imem counters
202   reg [`REG_SIZE:0] cycles_current, num_inst_current;
203   always @(posedge clk) begin
204     if (rst) begin
205       cycles_current <= 0;
206       num_inst_current <= 0;
207     end else begin
208       cycles_current <= cycles_current + 1;
209       if (!rst && !stall_en) begin
210         num_inst_current <= num_inst_current + 1;
211       end
212     end
213   end
214
215   // NOTE: don't rename your RegFile instance as the tests expect it
        to be `rf`
216   // TODO: you will need to edit the port connections, however.
217   wire [`REG_SIZE:0] rs1_data;
218   wire [`REG_SIZE:0] rs2_data;
219   reg [`REG_SIZE:0] rd_data;
220   reg rf_we;
221
222   RegFile rf (
223     .clk      (clk),
224     .rst      (rst),
225     .we       (rf_we),
226     .rd       (inst_rd),
227     .rd_data  (rd_data),
228     .rs1      (inst_rs1),
229     .rs2      (inst_rs2),
230     .rs1_data (rs1_data),
231     .rs2_data (rs2_data)
232   );
233
```

```
234   wire [31:0] cla_res, cla_op_b, cla_op_b_raw;
235   wire cla_cin;
236
237   assign cla_op_b_raw = (inst_opcode == OpRegImm) ? imm_i_sext :
          rs2_data;
238   assign cla_op_b = inst_sub ? ~cla_op_b_raw : cla_op_b_raw;
239   assign cla_cin = inst_sub ? 1'b1 : 1'b0;
240
241   cla u_cla (
242     .a      (rs1_data),
243     .b      (cla_op_b),
244     .cin    (cla_cin),
245     .sum    (cla_res)
246   );
247
248   wire [31:0] div_op_a, div_op_b;
249   wire is_signed_div;
250
251   assign is_signed_div = inst_div | inst_rem;
252   assign div_op_a = (is_signed_div & rs1_data[31]) ? (~rs1_data + 1) :
          rs1_data;
253   assign div_op_b = (is_signed_div & rs2_data[31]) ? (~rs2_data + 1) :
          rs2_data;
254
255   wire [31:0] div_quot_raw, div_rem_raw;
256   DividerUnsignedPipelined u_div (
257     .clk (clk),
258     .rst (rst),
259     .stall (1'b0),
260     .i_dividend (div_op_a),
261     .i_divisor  (div_op_b),
262     .o_quotient (div_quot_raw),
263     .o_remainder(div_rem_raw)
264   );
265
266   reg [31:0] div_res;
267   always @(*) begin
268     if (inst_div || inst_divu) begin
269       if (rs2_data == 0) begin
270         div_res = 32'hFFFF_FFFF;
271       end else if (inst_div && (rs1_data == 32'h8000_0000) && (
              rs2_data == 32'hFFFF_FFFF)) begin
272         div_res = 32'h8000_0000;
273       end else if (is_signed_div && (rs1_data[31] ^ rs2_data[31]))
              begin
274         div_res = ~div_quot_raw + 32'd1;
275       end else begin
276         div_res = div_quot_raw;
277       end
278     end else begin // rem or remu
279       if (rs2_data == 0) begin
280         div_res = rs1_data;
281       end else if (inst_rem && (rs1_data == 32'h8000_0000) && (
```

```
        rs2_data == 32'hFFFF_FFFF)) begin
282       div_res = 32'd0;
283     end else if (is_signed_div && (rs1_data[31])) begin
284       div_res = ~div_rem_raw + 32'd1;
285     end else begin
286       div_res = div_rem_raw;
287     end
288   end
289 end
290
291 wire [63:0] mul_u, mul_s, mul_su;
292 assign mul_u = {32'b0, rs1_data} * {32'b0, rs2_data};
293 assign mul_s = $signed(rs1_data) * $signed(rs2_data);
294 assign mul_su = $signed(rs1_data) * $signed({1'b0, rs2_data});
295
296 reg take_branch;
297 always @(*) begin
298   case (inst_funct3)
299     3'b000: take_branch = (rs1_data == rs2_data);
          // beq
300     3'b001: take_branch = (rs1_data != rs2_data);
          // bne
301     3'b100: take_branch = ($signed(rs1_data) < $signed(rs2_data));
          // blt
302     3'b101: take_branch = ($signed(rs1_data) >= $signed(rs2_data));
          // bge
303     3'b110: take_branch = (rs1_data < rs2_data);
          // bltu
304     3'b111: take_branch = (rs1_data >= rs2_data);
          // bgeu
305     default: take_branch = 1'b0;
306   endcase
307 end
308
309 reg [31:0] load_data;
310 always @(*) begin
311   case (addr_to_dmem[1:0])
312     2'b00: load_data = load_data_from_dmem;
313     2'b01: load_data = load_data_from_dmem >> 8;
314     2'b10: load_data = load_data_from_dmem >> 16;
315     2'b11: load_data = load_data_from_dmem >> 24;
316   endcase
317 end
318
319 reg illegal_inst;
320
321 always @(*) begin
322   illegal_inst = 1'b0;
323   pcNext = pcCurrent + 4;
324   rf_we = 0;
325   rd_data = 0;
326   halt = 0;
327   addr_to_dmem = 0;
```

```
328     store_data_to_dmem = 0;
329     store_we_to_dmem = 0;
330
331     case (inst_opcode)
332       OpLui: begin
333         // TODO: start here by implementing lui
334         rf_we = 1;
335         rd_data = {inst_from_imem[31:12], 12'b0};
336       end
337
338       OpAuipc: begin
339         rf_we = 1;
340         rd_data = pcCurrent + {inst_from_imem[31:12], 12'b0};
341       end
342
343       OpJal: begin
344         rf_we = 1;
345         rd_data = pcCurrent + 4;
346         pcNext = pcCurrent + imm_j_sext;
347       end
348
349       OpJalr: begin
350         rf_we = 1;
351         rd_data = pcCurrent + 4;
352         pcNext = (rs1_data + imm_i_sext) & ~32'd1;
353       end
354
355       OpBranch: begin
356         if (take_branch) begin
357           pcNext = pcCurrent + imm_b_sext;
358         end
359       end
360
361       OpLoad: begin
362         rf_we = 1;
363         addr_to_dmem = rs1_data + imm_i_sext;
364
365         case (inst_funct3)
366           3'b000: rd_data = {{24{load_data[7]}}, load_data[7:0]};   //
                lb
367           3'b001: rd_data = {{16{load_data[15]}}, load_data[15:0]}; //
                lh
368           3'b010: rd_data = load_data;                              //
                lw
369           3'b100: rd_data = {24'b0, load_data[7:0]};                //
                lbu
370           3'b101: rd_data = {16'b0, load_data[15:0]};               //
                lhu
371         endcase
372       end
373
374       OpStore: begin
375         addr_to_dmem = rs1_data + imm_s_sext;
```

```
376         store_data_to_dmem = rs2_data << (addr_to_dmem[1:0] * 8);
377
378         case (inst_funct3)
379           3'b000: store_we_to_dmem = 4'b0001 << addr_to_dmem[1:0]; //
                sb
380           3'b001: store_we_to_dmem = 4'b0011 << addr_to_dmem[1:0]; //
                sh
381           3'b010: store_we_to_dmem = 4'b1111;                      //
                sw
382         endcase
383       end
384
385       OpRegImm: begin
386         rf_we = 1;
387         case (inst_funct3)
388           3'b000: rd_data = cla_res;
                                                                  // addi
389           3'b010: rd_data = ($signed(rs1_data) < $signed(imm_i_sext))
                ? 32'd1 : 32'd0; // slti
390           3'b011: rd_data = (rs1_data < imm_i_sext) ? 32'd1 : 32'd0;
                              // sltiu
391           3'b100: rd_data = rs1_data ^ imm_i_sext;
                                                    // xori
392           3'b110: rd_data = rs1_data | imm_i_sext;
                                                    // ori
393           3'b111: rd_data = rs1_data & imm_i_sext;
                                                    // andi
394           3'b001: rd_data = rs1_data << imm_shamt;
                                                    // slli
395           3'b101: begin
396             if (inst_funct7 == 7'd0) begin
397               rd_data = rs1_data >> imm_shamt;
                                                          // srli
398             end else begin
399               rd_data = $signed(rs1_data) >>> imm_shamt;
                                                 // srai
400             end
401           end
402         endcase
403       end
404
405       OpRegReg: begin
406         if (is_div_op) begin
407           rf_we = !stall_en;
408         end else begin
409           rf_we = 1;
410         end
411
412         if (inst_funct7 == 7'd1) begin
413           case (inst_funct3)
414             3'b000: rd_data = rs1_data * rs2_data;                 // mul
415             3'b001: rd_data = mul_s[63:32];                        // mulh
416             3'b010: rd_data = mul_su[63:32];                       //
```

```
              mulhsu
417           3'b011: rd_data = mul_u[63:32];                    //
              mulhu
418           3'b100, 3'b101, 3'b110, 3'b111: rd_data = div_res; // div,
               divu, rem, remu
419         endcase
420       end else begin
421         case (inst_funct3)
422           3'b000: rd_data = cla_res;
                                                                 // add,
               sub
423           3'b001: rd_data = rs1_data << rs2_data[4:0];
                                                 // sll
424           3'b010: rd_data = ($signed(rs1_data) < $signed(rs2_data))
              ? 32'd1 : 32'd0; // slt
425           3'b011: rd_data = (rs1_data < rs2_data) ? 32'd1 : 32'd0;
                                 // sltu
426           3'b100: rd_data = rs1_data ^ rs2_data;
                                                       // xor
427           3'b101: begin
428             if (inst_funct7[5]) rd_data = $signed(rs1_data) >>>
                rs2_data[4:0];       // sra
429             else rd_data = rs1_data >> rs2_data[4:0];
                                                    // srl
430           end
431           3'b110: rd_data = rs1_data | rs2_data;
                                                       // or
432           3'b111: rd_data = rs1_data & rs2_data;
                                                       // and
433         endcase
434       end
435     end
436
437     OpEnviron: begin
438       if (inst_from_imem[31:7] == 25'd0) halt = 1'b1; // ecall
439     end
440
441     default: begin
442       illegal_inst = 1'b1;
443     end
444   endcase
445 end
446 endmodule
447
448 module MemorySingleCycle #(
449     parameter NUM_WORDS = 512
450 ) (
451   input                    rst,                 // rst for both imem
        and dmem
452   input                    clock_mem,           // clock for both imem
       and dmem
453   input      [`REG_SIZE:0] pc_to_imem,          // must always be
        aligned to a 4B boundary
```

```
454   output reg [`REG_SIZE:0] inst_from_imem,      // the value at memory
          location pc_to_imem
455   input      [`REG_SIZE:0] addr_to_dmem,        // must always be
          aligned to a 4B boundary
456   output reg [`REG_SIZE:0] load_data_from_dmem, // the value at memory
          location addr_to_dmem
457   input      [`REG_SIZE:0] store_data_to_dmem,  // the value to be
          written to addr_to_dmem, controlled by store_we_to_dmem
458   // Each bit determines whether to write the corresponding byte of
          store_data_to_dmem to memory location addr_to_dmem.
459   // E.g., 4'b1111 will write 4 bytes. 4'b0001 will write only the
          least-significant byte.
460   input      [        3:0] store_we_to_dmem
461 );
462
463   // memory is arranged as an array of 4B words
464   reg [`REG_SIZE:0] mem_array[0:NUM_WORDS-1];
465
466   // preload instructions to mem_array
467   always @(posedge rst) begin
468     $readmemh("mem_initial_contents.hex", mem_array);
469   end
470
471   localparam AddrMsb = $clog2(NUM_WORDS) + 1;
472   localparam AddrLsb = 2;
473
474   always @(posedge clock_mem) begin
475     inst_from_imem <= mem_array[{pc_to_imem[AddrMsb:AddrLsb]}];
476   end
477
478   always @(negedge clock_mem) begin
479     if (store_we_to_dmem[0]) begin
480       mem_array[addr_to_dmem[AddrMsb:AddrLsb]][7:0] <=
            store_data_to_dmem[7:0];
481     end
482     if (store_we_to_dmem[1]) begin
483       mem_array[addr_to_dmem[AddrMsb:AddrLsb]][15:8] <=
            store_data_to_dmem[15:8];
484     end
485     if (store_we_to_dmem[2]) begin
486       mem_array[addr_to_dmem[AddrMsb:AddrLsb]][23:16] <=
            store_data_to_dmem[23:16];
487     end
488     if (store_we_to_dmem[3]) begin
489       mem_array[addr_to_dmem[AddrMsb:AddrLsb]][31:24] <=
            store_data_to_dmem[31:24];
490     end
491     // dmem is "read-first": read returns value before the write
492     load_data_from_dmem <= mem_array[{addr_to_dmem[AddrMsb:AddrLsb]}];
493   end
494 endmodule
495
496 /*
```

```
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)
  );

  DatapathMultiCycle 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 2: RTL implementation of the DatapathMultiCycle file.

3 Testbench

3.1 Divider testbench

This testbench is used to verify the functional correctness and the latency of the 8-stage Pipelined Unsigned Divider. Unlike standard combinational logic tests, this testbench must account for the 8-cycle latency between input stimulus and valid output.

The testbench uses two separate `initial` blocks running in parallel to handle the pipelined nature of the device:

- Stimulus generator: Feeds input data into the divider on every clock cycle.
- Output checker: Waits for the pipeline latency (roughly 10 cycles for reset and pipeline stages) before beginning to check the output against the expected values.

The testbench generates 15 random and directed test cases:

- Test 0: The first iteration test has exact division.
- Test 1: The second iteration test has remainder.
- Test 2: The third iteration test has dividend smaller than divisor.
- Test 3-14: The remaining cases use `$random` to generate arbitrary 32-bit values to ensure robust coverage.

```
`timescale 1ns / 1ps
module tb_div;
reg clk, rst, stall;
reg [31:0] i_dividend, i_divisor;
wire [31:0] o_remainder, o_quotient;

integer i, j;
integer pass, fail;
parameter TEST_CNT = 15;

reg [31:0] i_divd [0:TEST_CNT-1];
reg [31:0] i_divs [0:TEST_CNT-1];
reg [31:0] exp_quot [0:TEST_CNT-1];
reg [31:0] exp_rem  [0:TEST_CNT-1];

DividerUnsignedPipelined dut (
    .clk(clk),
```

```
18      .rst(rst),
19      .stall(stall),
20      .i_dividend(i_dividend),
21      .i_divisor(i_divisor),
22      .o_remainder(o_remainder),
23      .o_quotient(o_quotient)
24  );
25
26  task divi;
27      $display ("%0s", {80{"-"}});
28  endtask
29
30  task br;
31      $display ("%0s", {100{"="}});
32  endtask
33
34  task msg (input [700:0] txt);
35      begin
36          br();
37          $display ("%0s", txt);
38          br();
39      end
40  endtask
41
42  initial begin
43      clk = 0;
44      forever #5 clk = ~clk;
45  end
46
47  initial begin
48      $dumpfile("wave_div.vcd");
49      $dumpvars(0, tb_div);
50      msg ("TEST BEGIN");
51
52      rst = 1;
53      stall = 0;
54      i_dividend = 0;
55      i_divisor = 1;
56      pass = 0;
57      fail = 0;
58      #20;
59
60      @(negedge clk);
61      rst = 0;
62      repeat (2) @(posedge clk);
63
64      for (i = 0; i < TEST_CNT; i = i + 1) begin
65          @(negedge clk);
66          if (i == 0) begin
67              i_divd[i] = 100;
68              i_divs[i] = 10; // chia het
69          end else if (i == 1) begin
70              i_divd[i] = 100;
```

```
            i_divs[i] = 30; // co du
        end else if (i == 2) begin
            i_divd[i] = 5;
            i_divs[i] = 10; // bi chia < chia
        end else begin
            i_divd[i] = $random;
            i_divs[i] = ($random % 100) + 1;
        end

        i_dividend = i_divd[i];
        i_divisor = i_divs[i];

        exp_quot[i] = i_divd[i] / i_divs[i];
        exp_rem[i]  = i_divd[i] % i_divs[i];
    end

    @(negedge clk);
    i_dividend = 0;
    i_divisor = 1;
end

initial begin
    wait (rst == 0);
    repeat (10) @(posedge clk); // 2 delay + 8 latency = 10

    for (j = 0; j < TEST_CNT; j = j + 1) begin
        @(negedge clk);
        if ((o_quotient === exp_quot[j]) && (o_remainder === exp_rem[j
           ])) begin
            $display("[PASS] Time = %0t | Test #%2d: %d / %d | Output
               Q = %d, R = %d", $time, j, i_divd[j], i_divs[j],
               o_quotient, o_remainder);
            pass = pass + 1;
        end else begin
            $display("[FAIL] Time = %0t | Test #%2d: %d / %d | Expect
               Q = %d, R = %d | Output Q = %d, R = %d", $time, j,
               i_divd[j], i_divs[j], exp_quot[j], exp_rem[j],
               o_quotient, o_remainder);
            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");
```

```
118
119     msg ("TEST END");
120
121     #100;
122     $finish;
123 end
124 endmodule
```

Program 3: Testbench implementation of the DividerUnsignedPipelined module.

3.2 Processor testbench

To verify the `DatapathMultiCycle` file, we reused the testbench originally designed for the `DatapathSingleCycle` file (`tb_single.v`), can be viewed again in Section 4.2 in Lab 3 report. This approach is valid because the Instruction Set Architecture (ISA) interface remains identical between the two designs. Both processors:

- Read 32-bit instruction from memory.
- Execute standard RV32I instructions (ADD, SUB, BEQ, etc).
- Perform division operations (DIV, DIVU, REM, REMU), although with different internal latencies.
- Ultimately update the register file and memory with the same correct values.

Since the testbench verifies correctness based on the final state rather than internal details, it serves as an effective regression test to ensure that the new multi-cycle features did not break existing functionality.

While the test logic remains the same, the execution time for division-heavy programs increase significantly dues to the 8-cycle latency per division operation.

4 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.

4.1 Divider testbench results

```
1 # ================================================================
2 # TEST BEGIN
3 # ================================================================
4 # [PASS] Time = 120000 | Test # 0:         100 /          10 | Output Q
    =          10, R =           0
5 # [PASS] Time = 130000 | Test # 1:         100 /          30 | Output Q
    =           3, R =          10
6 # [PASS] Time = 140000 | Test # 2:           5 /          10 | Output Q
    =           0, R =           5
7 # [PASS] Time = 150000 | Test # 3:   303379748 / 4294967198 | Output Q
    =           0, R =   303379748
```

```
# [PASS] Time = 160000 | Test # 4: 2223298057 / 4294967288 | Output Q
    =          0, R = 2223298057
# [PASS] Time = 170000 | Test # 5:  112818957 /         58 | Output Q
    =    1945154, R =         25
# [PASS] Time = 180000 | Test # 6: 2999092325 / 4294967283 | Output Q
    =          0, R = 2999092325
# [PASS] Time = 190000 | Test # 7:   15983361 /         30 | Output Q
    =     532778, R =         21
# [PASS] Time = 200000 | Test # 8:  992211318 /         98 | Output Q
    =   10124605, R =         28
# [PASS] Time = 210000 | Test # 9: 1993627629 /         13 | Output Q
    =  153355971, R =          6
# [PASS] Time = 220000 | Test #10: 2097015289 / 4294967227 | Output Q
    =          0, R = 2097015289
# [PASS] Time = 230000 | Test #11: 3807872197 / 4294967223 | Output Q
    =          0, R = 3807872197
# [PASS] Time = 240000 | Test #12: 1924134885 / 4294967256 | Output Q
    =          0, R = 1924134885
# [PASS] Time = 250000 | Test #13: 2301810194 /         40 | Output Q
    =   57545254, R =         34
# [PASS] Time = 260000 | Test #14: 2033215986 / 4294967251 | Output Q
    =          0, R = 2033215986
# ================================================================
# SUMMARY
# ================================================================
# TOTAL TESTS : 15
# TOTAL PASSED: 15
# TOTAL FAILED: 0
# ======> ALL TESTS PASSED
# ================================================================
# TEST END
# ================================================================
# ** Note: $finish    : ../../tb/tb_div.v(122)
#    Time: 360 ns  Iteration: 0  Instance: /tb_div
```

Simulation 1: Testbench results of the DividerUnsignedPipelined module.

4.2 Processor testbench results

```
# ================================================================
# TEST BEGIN
# ================================================================
# ================================================================
# TEST 1
# ================================================================
# [PASS] Time = 691 |            ADDI x1, x0, 10 | Output x1 = 0
    x0000000a
# [PASS] Time = 691 |            ADDI x2, x0, -5 | Output x2 = 0
    xfffffffb
# [PASS] Time = 691 |             ADD x3, x1, x2 | Output x3 = 0
    x00000005
# [PASS] Time = 691 |             SUB x4, x1, x2 | Output x4 = 0
    x0000000f
```

```
# [PASS] Time = 691 |              AND x5, x1, x2 | Output x5 = 0
    x0000000a
# [PASS] Time = 691 |               OR x6, x1, x2 | Output x6 = 0
    xfffffffb
# [PASS] Time = 691 |              XOR x7, x1, x2 | Output x7 = 0
    xfffffff1
# [PASS] Time = 691 |             SLLI x8, x1, 2 | Output x8 = 0
    x00000028
# [PASS] Time = 691 |             SRAI x9, x2, 1 | Output x9 = 0
    xfffffffd
# [PASS] Time = 691 |            SRLI x10, x2, 1 | Output x10 = 0
    x7fffffd
# [PASS] Time = 691 |            SLT x11, x2, x1 | Output x11 = 0
    x00000001
# [PASS] Time = 691 |           SLTU x12, x2, x1 | Output x12 = 0
    x00000000
# [PASS] Time = 691 |            MUL x13, x1, x1 | Output x13 = 0
    x00000064
# [PASS] Time = 691 |            DIV x14, x1, x2 | Output x14 = 0
    xfffffffe
# [PASS] Time = 691 |            REM x15, x1, x2 | Output x15 = 0
    x00000000
# [PASS] Time = 691 |           DIVU x16, x2, x1 | Output x16 = 0
    x19999999
# [PASS] Time = 691 |            LW x18, 100(x0) | Output x18 = 0
    x00000005
# [PASS] Time = 691 |            ADDI x19, x0, 1 | Output x19 = 0
    x00000001
# [PASS] Time = 691 |                 JAL x20, 8 | Output x20 = 0
    x00000058
# [PASS] Time = 691 |            ADDI x21, x0, 2 | Output x21 = 0
    x00000002
# [PASS] Time = 691 |           LUI x22, 0x12345 | Output x22 = 0
    x12345000
# [PASS] Time = 691 |            XORI x23, x1, 5 | Output x23 = 0
    x0000000f
# [PASS] Time = 691 |          ANDI x24, x1, 0xF | Output x24 = 0
    x0000000a
# [PASS] Time = 691 |       ORI x25, x13, 0xFF00 | Output x25 = 0
    xffffff64
# [PASS] Time = 691 |         SLTI x26, x7, 0x28 | Output x26 = 0
    x00000001
# [PASS] Time = 691 |  LTIU x27, x8, 0xFFFF_FFF1 | Output x27 = 0
    x00000001
# [PASS] Time = 691 |           SLL x28, x1, x19 | Output x28 = 0
    x00000014
# [PASS] Time = 691 |           SRL x29, x6, x19 | Output x29 = 0
    x7fffffd
# [PASS] Time = 691 |           SRA x30, x6, x19 | Output x30 = 0
    xfffffffd
# [PASS] Time = 691 |           REMU x31, x6, x3 | Output x31 = 0
    x00000001
# [PASS] HALT asserted.
```

```
# ================================================================
# TEST 2 (REUSED REGISTERS)
# ================================================================
# [PASS] Time = 971 |   ADDI x1, x0, 0xFFFF_FFFB | Output x1 = 0
   xfffffffb
# [PASS] Time = 971 |             ADDI x2, x0, 3 | Output x2 = 0
   x00000003
# [PASS] Time = 971 |            MULH x3, x1, x2 | Output x3 = 0
   xffffffff
# [PASS] Time = 971 |          MULHSU x4, x1, x2 | Output x4 = 0
   xffffffff
# [PASS] Time = 971 |           MULHU x5, x1, x2 | Output x5 = 0
   x00000002
# [PASS] Time = 971 |   ADDI x6, x0, 0xABCD_FE98 | Output x6 = 0
   xfffffe98
# [PASS] Time = 971 |             LB x7, 100(x0) | Output x7 = 0
   xffffff98
# [PASS] Time = 971 |             LH x8, 104(x0) | Output x8 = 0
   xfffffe98
# [PASS] Time = 971 |            LBU x9, 105(x0) | Output x9 = 0
   x000000fe
# [PASS] Time = 971 |           LHU x10, 104(x0) | Output x10 = 0
   x0000fe98
# [PASS] Time = 971 |           ADD x11, x9, x10 | Output x11 = 0
   x0000ff96
# [PASS] Time = 971 |            OR x12, x9, x10 | Output x12 = 0
   x0000fefe
# [PASS] Time = 971 |            LW x13, 120(x0) | Output x13 = 0
   x000000fe
# [PASS] Time = 971 |          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: 1071 ns  Iteration: 0  Instance: /tb_single
```

Simulation 2: Testbench results of the Processor module.

5 Simulation waveform

5.1 Divider simulation waveform

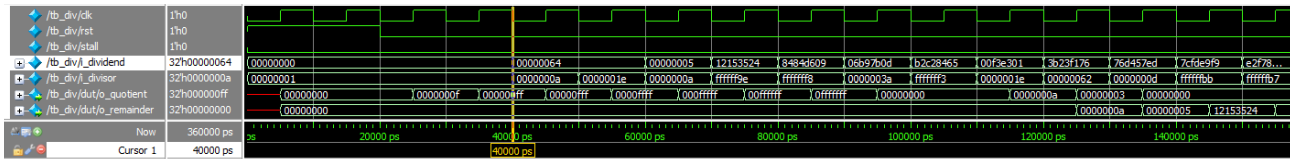

Figure 1

In Figure 1, the testbench updates the dividend and divisor values on each negative clock edge to initiate new division operations. At 40,000 ps, the first test case loads dividend = 100 (0x64) and divisor = 10 (0x0a). The `DividerUnsignedPipelined` module has an 8-stage pipeline latency, meaning the result appears after 8 positive clock edges. Consequently, at 120,000 ps (8 cycles later), the outputs are updated: quotient = 10 (0x0a) and remainder = 0.

The second test case loads dividend = 100 (0x64) and divisor = 30 (0x1e) at 50,000 ps (one clock cycle after the first). Due to the pipelined architecture, this operation also requires 8 clock cycles to complete. At 130,000 ps, the second result emerges: quotient = 3 and remainder = 10 (0x0a). This pattern continues for subsequent test cases, with each new operation initiated on consecutive negative edges.

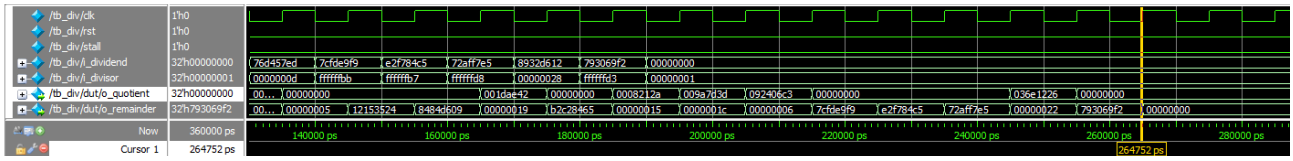

Figure 2

As shown in Figure 2, once the pipeline is filled (after the initial 8-cycle latency), the divider produces one result per clock cycle in steady-state operation. The testbench loads 15 test cases sequentially, feeding new operands every cycle. After all valid test cases complete, the testbench sets dividend = 0 and divisor = 1 as safe default values to avoid division-by-zero scenarios, which are not handled in this design. Once the pipeline is full, we get one result per cycle, although each operation needs 8 cycles.

5.2 Processor simulation waveform

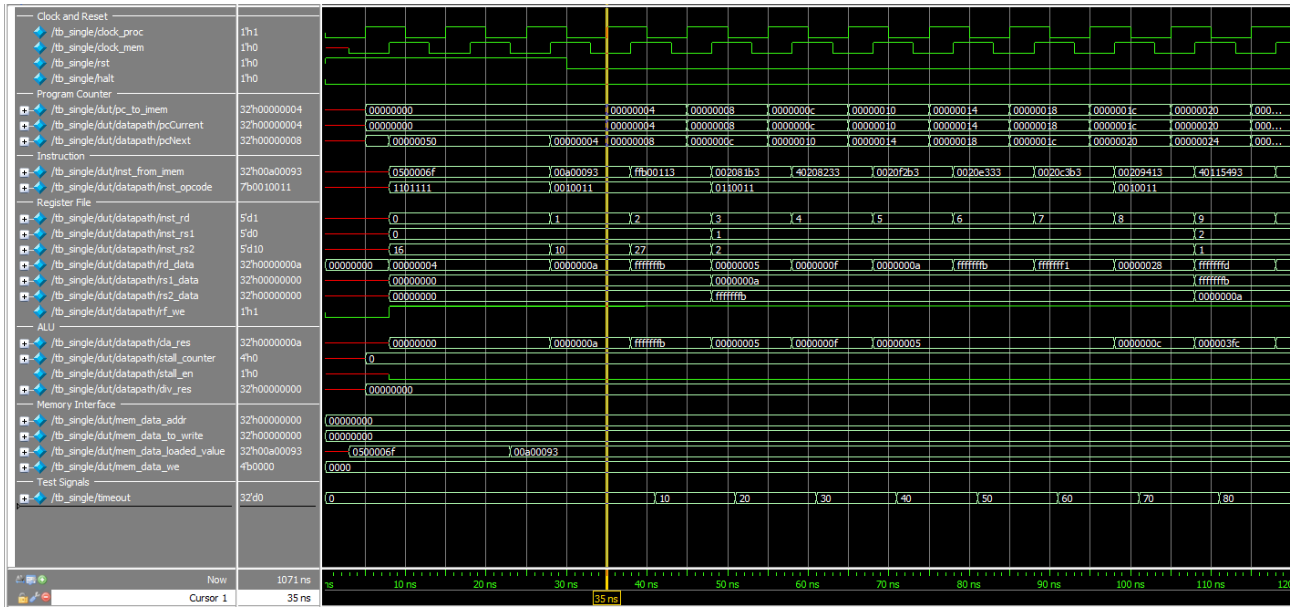

Figure 3

Both processor implementations were tested using the same testbench file `tb_single.v` to ensure they produce identical functional results. As shown in Figure 3, the multi-cycle processor executes most instructions with the same behavior as the single-cycle implementation. The register file values, ALU results and memory interface signals match between both designs, confirming that both processors correctly implement the RISC-V instruction set. For instructions like ADD, SUB, LW, SW, and branches, both processors complete execution in one clock cycle and advance the program counter by 4 bytes to fetch the next instruction.

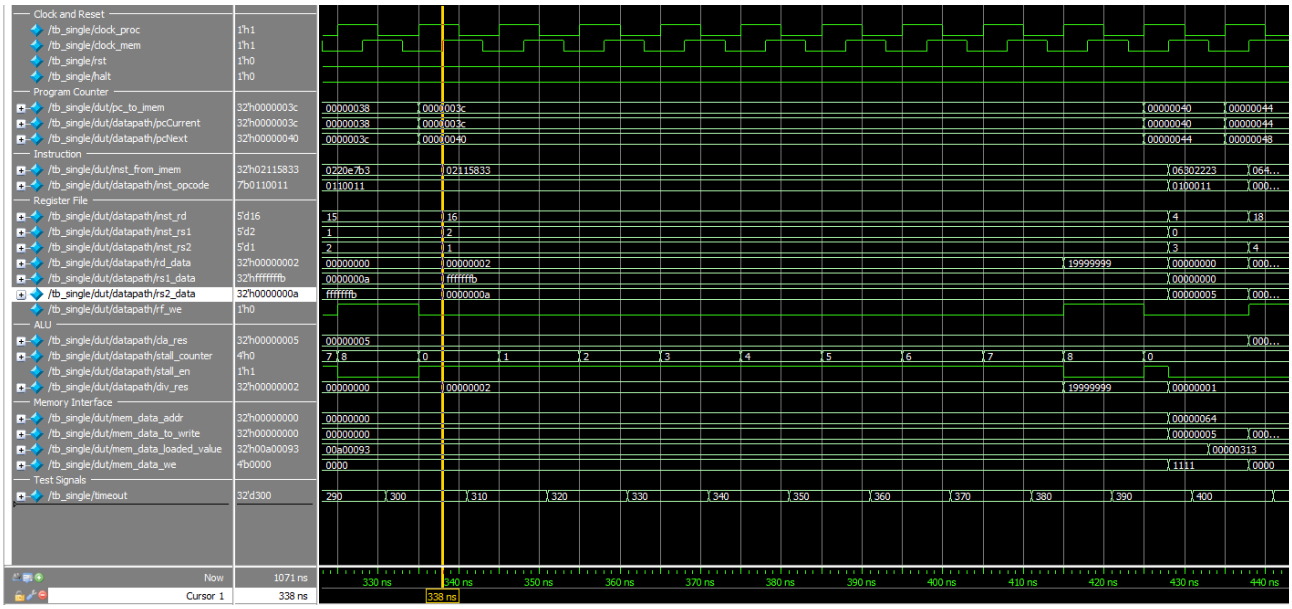

Figure 4

However, a significant difference appears when executing division instructions. The single-cycle processor uses a combinational divider that completes the entire division operation within a single clock cycle. In contrast, the multi-cycle processor employs a pipelined divider which has an 8-stage pipeline architecture. As shown in Figure 4 at approximately 338 ns, when the DIVU instruction is executed, the processor detects that this is a division operation and activates the stall mechanism. The `stall_counter` begins counting from 0 to 8 over eight consecutive clock cycles, during which the `stall_en` signal remains high. While `stall_en` is active, the program counter freezes at its current value (0x3C in this case) and does not advance to the next instruction. The register file write enable signal `rf_we` is also disabled during the stall period, preventing any premature register updates. After finishing the 8-stages pipelines in 8 clock cyles, on the next postive edge `clock_proc`, the program counter finally advanced to the next instruction. Therefore, each division instruction takes 9 clock cycles total: 8 for computation and 1 for safe control handoff, which is standard practice in pipelined processor design where the write-back stage occurs one cycle after the execution stage completes.

Although each individual division takes 9 cycles to complete in the multi-cycle design compared to just 1 cycle in the single-cycle design, the multi-cycle processor can run at a higher clock frequency. For programs that do not heavily rely on division operations, the multi-cycle processor achieves better overall performance through its higher clock frequency despite having a variable CPI (Cycles Per Instruction) of 1 for most instructions and 9 for divisions.

6 Timing closure and Resource utilization

6.1 Timing closure

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-9,451 ns	Worst Hold Slack (WHS):	-0,239 ns	Worst Pulse Width Slack (WPWS):	3,020 ns
Total Negative Slack (TNS):	-8431,276 ns	Total Hold Slack (THS):	-189,649 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	1290	Number of Failing Endpoints:	1587	Number of Failing Endpoints:	0
Total Number of Endpoints:	4276	Total Number of Endpoints:	4276	Total Number of Endpoints:	1664

Timing constraints are not met.

Figure 5: Timing summary.

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

- Constraint clock (clock_proc) frequency: 125 MHz (Period: 8.000 ns)
- Worst Negative Slack (WNS): -9.451 ns
- Conclusion: The design has not achieved Timing closure (Timing constraints are not met).

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

$$T\_min = T\_constraint - WNS = 8.000 - (-9.451) = 17.451ns$$

$$F\_max = \frac{1}{T\_min} \approx 57.30MHz$$

The previous results we get from the single-cycle datapath design in Lab 3 (can be viewed in Section 7.1) had a significantly lower F_max (8.75 MHz). Meanwhile, the introduction of the 8-stage pipelined divider has successfully broken the critical path. The new F_max of 57.30 MHz represents a massive improvement, approximately 6.5 times faster, compared to the single-cycle version. Although the design still does not meet the aggressive 125 MHz constraint, the pipelining strategy has proven effective in reducing the critical path delay, validating the design goal of the multi-cycle architecture.

6.2 Resouce utilization

Resource	Utilization	Available	Utilization %
LUT	4941	53200	9.29
FF	1627	106400	1.53
BRAM	8	140	5.71
DSP	12	220	5.45
IO	4	125	3.20

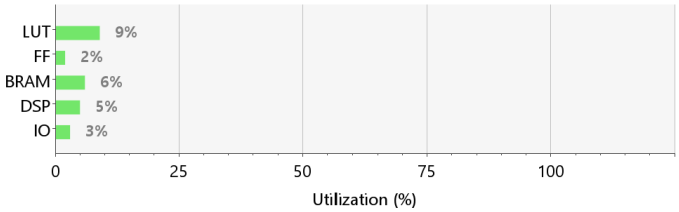

Figure 6: Utilization summary.

Both single-cycle datapath and multi-cycle datapath resulted in 9% LUT usage, indicating that the combinational logic complexity is similar. However, there is a noticeable increase in Flip-Flop utilization, 1.53% vs 0.96% in single-cycle). This increase is directly attributed to the pipeline registers added between the 8 stages of the divider and the new state registers (stall counter) in the datapath. In conclusion, the design fits easily within the Arty Z7-20 FPGA.