Implementation of a Two Stage Pipelined Processor using RISC-V Instruction Set Architecture

Zhanhui Chen
500131284

***Abstract*—The basic RISC-V instruction consisted of mainly six different formats. The RISC-V micro-processor is capable to pipelining, which provides a considerable speedup to the processor in general when implemented. However, the pipelining strategy might also introduce the risk of data, control and structural hazards during instruction flows. Therefore, additional method like Forwarding and Interlock need to be implemented to handle the hazards. The methods are generally straightforward to implement, since the RISC-V architecture is rather simple.**

I. Introduction

RISC-V, where the number V (five) refers to the number of generations of RISC architecture that were developed at the University of California since 1981. It is an open standard instruction set architecture (ISA) based on established RISC principles. Some renowned ISA that are vastly used these days include RISC-V, advanced RISC machine (ARM) and x86. While ARM and x86 are incredibility difficult to study and code, hence RISC-V is implemented here due to its simplicity. RISC-V is open-sourced and is very flexibly with the ability of pipelining. In this report, a general workflow of a single cycle and two-stage pipelined processor will be discussed. Interlock and Forwarding strategies will also be implemented and visualisation of their effects on the processor such as performance will be provided in the appendix. All implementations used the RISC-V 32bit instruction set, were written in System Verilog and simulated in ModelSim.

II. Overview and Backbround

A. Overview

An RISC-V instruction can be broken down into five successive steps: instruction fetch (IF), instruction decode and register file fetch (ID), execution (EX), memory allocation (MA) and write back (WB). In a single cycle processor, these steps will be done within one cycle, as the name suggested. In a pipelined processor, these steps can also be done over multiple cycles.

B. Specifications

The IF step involves a program counter (PC) that increments by 4 each cycle and provides the address of the current instruction at the positive clock edge of the concurrent cycle. The address is then stored in the instruction memory (imem). Therefore, on the system rising clock edge, the PC will increment and point to the next instruction in memory. The value of PC can sometimes increment or decrement by more than 4 if a jump or branch instruction is called, based on the offset value provided. These instructions can change the PC value to any location in imem within a positive 32-bit address range.

The ID step will determine the instruction that the ALU is going to compute in this cycle using the func3, func7 and the opcode of the instruction. The RISC-V 32-bit ISA has six different instruction types, R, I, S, B, U and J. These instructions can be distinguished based on the format and the opcode. Figure II-B-1 below shows the formatting of each instruction. In addition to determining the type of the instruction, the decode stage also specific which two values will input to the ALU, some of the values will also be fetched from the register files base on the address given from the instruction prior to the computation.

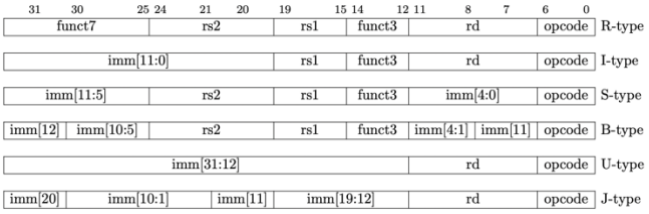

Figure II-B-1 RISC-V 32-bit Instruction Format [1]

The EX step is when the two values from previous step is computed. Apart from the general arithmetic calculation like an ADD or ADDI, a branch instruction compares between the two values and alter the PC base on the condition. Furthermore, a jump and link instruction simply jump to the specific location in the program without comparison. Nevertheless, an output value will be produced in this step.

The MA step is where the data is loaded from or stored to the data memory (dmem) if required by the instruction.

The WB stage is for the computed result from the ALU output to write back to the destination register, specified from the 32-bit instruction, in the register file.

III. Architechture

In this section, a general picture of single cycle processor and an example use of pipeline register will be discussed. Also, there will be a comparison between the two.

A. Single Cycle Processor

The simplest implementation of the processor is called a single cycle processor, because it only processes one instruction at each cycle. This ensure that no hazard will be presented during the instruction flow and hence reduce the complexity in hardwiring and structuring the program. Unfortunately, there is one drawback from this implementation, which is the clock frequency of such processor is slower than that of pipelined processor. This disadvantage is because the processor must wait until the current instruction is finished before reading and execute the next instruction. As a result, the overall speed is limited by the propagation delay in the processor.

Referring to the simple single cycle processor data path, figure III-A-1, the instruction fetch stage is to fetch the next instruction from the instruction memory at the beginning of the current clock cycle. First, the program counter will push its value to the instruction memory, the instruction memory will fetch the 32 bits instruction according to the address specified by the program counter. Meanwhile, the program counter will also update its value by 4 by default.

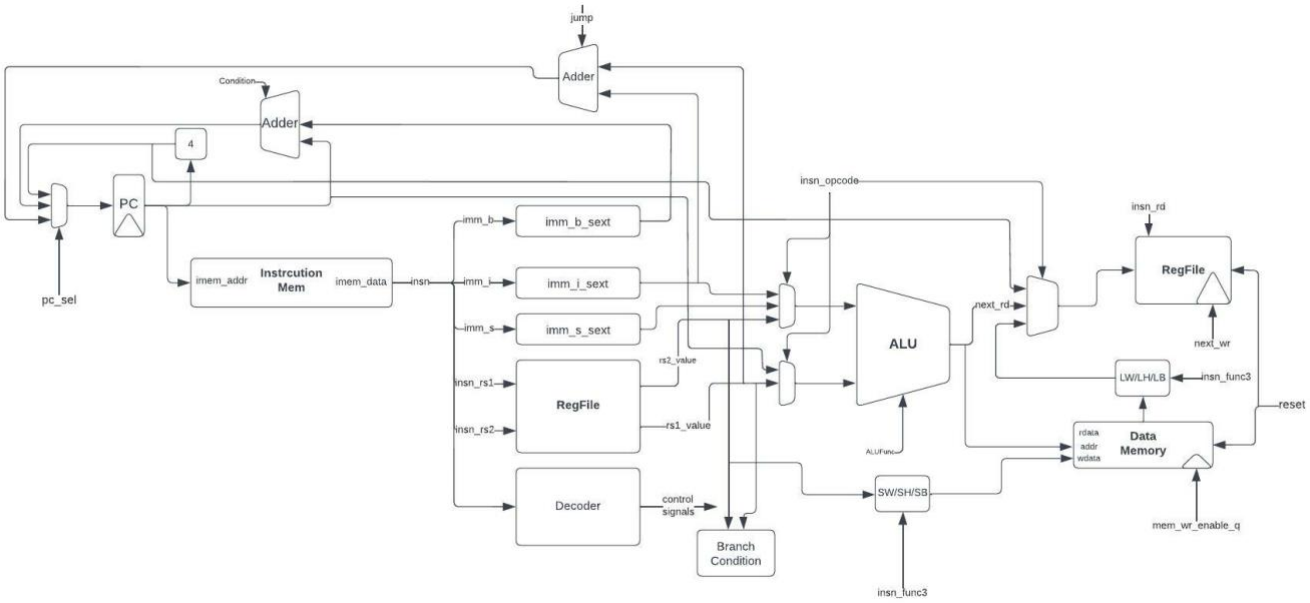

Figure III-A-1 Simple Single Cycle Processor Datapath

Then in the decode and register fetch stage, the 32 bits instruction will be broken down into several pieces in order to satisfy different operation needs. By referring to the instruction format listed in figure II-B-1, one can see how an instruction is separated into different values. Because these values are coded using assignments, the values will be updated as soon as the instruction is updated. Same applies to the rs1 and rs2 values. The register file will first receive the register locations (insn_rs1 and insn_rs2) the instruction would like to access, then the register file will output the values stored inside those addresses. The immediate values from the instruction will also be sign extended. The control signals generated in this step, for example the opcode, will be used in upcoming stages to determine the operations to execute and data value to proceed with.

The next stage in the instruction flow is the execution, here the two multiplexers before the ALU will determine the values that can proceed to the ALU. The multiplexers will look at the opcode from the control signal and select the values from the last step. Then the ALU will look at the func3 and func7 from the control signal to select the operation that it needs to execute. If the current instruction is R or I type, the ALU will just computed the two values and store the result in next_rd. If the operation is a branch, the rs1_value and rs2_value will have a comparison, if the condition is true, the next program counter will update by an offset taken from the imm_b_sext and the multiplexer before the PC will enable this value to proceeed next cycle. Or, if the branch condition is not satisfied, the next program counter will still be PC + 4. If the instruction is a jump instruction, there will be no condition need to be satisfied and the next program counter will modify by the combination of rs1_value and imm_i_sext. The return address will be stored in next_rd (the figure has separated next_rd and next program counter for clear reading). If the operation is a load instruction, the address to read from the data memory is the value of rs1 plus imm_i_sext and the value will be stored in register insn_rd. Similarly, for store instruction, the write address is the value of rs1 plus imm_s_sext, and the data to write is rs2_value.

Once the computation has completed, it is time to modify the data memory if necessary. If the current instruction is load or store, there will be some interactions between the data memory and the control signals. Otherwise, the data flow will simply proceed to the final step. In this RISC-V processor, both the instruction and data memory are 32 bits in each length. The value that is going to store in the data memory will be assigned to the correct address inside the data memory, only if the control signal mem_wr_enable is high. Because how the processor is designed in System Verilog, the computed result writes at the positive edge of each cycle, hence the load instruction will take a total of two cycle to complete and write to register file in the next cycle. To avoid the instruction fetch continues at the next cycle, while the current load instruction has not completed yet, the control signal mem_rd_enable_q allows the program counter to freeze and wait an additional cycle.

The final writes back step contains a multiplexer which controls the data value that is going to feed back to the register file if the control signal next_wr is enabled.

B. Two-Stage Pipeline Processor

A two-stage pipeline processor allows two instructions are executed in parallel. It involves breaking the execution of an instruction across multiple clock cycles, in this case two clock cycles. With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing arithmetic operations, holding them in a buffer close to the processor until each instruction operation can be performed. The staging of instruction fetching is continuous. The result is an increase in the number of instructions that can be performed during a given time period. [2] As a result, the overall frequency is increased because the critical path of the processor has been divided. In a single cycle processor, the critical path is 5 because an instruction needs to complete all 5 steps before the next instruction starts. In an optimised pipeline processor, the pipeline registers are placed before or after the ALU, because the data path has been spilt rather evenly.

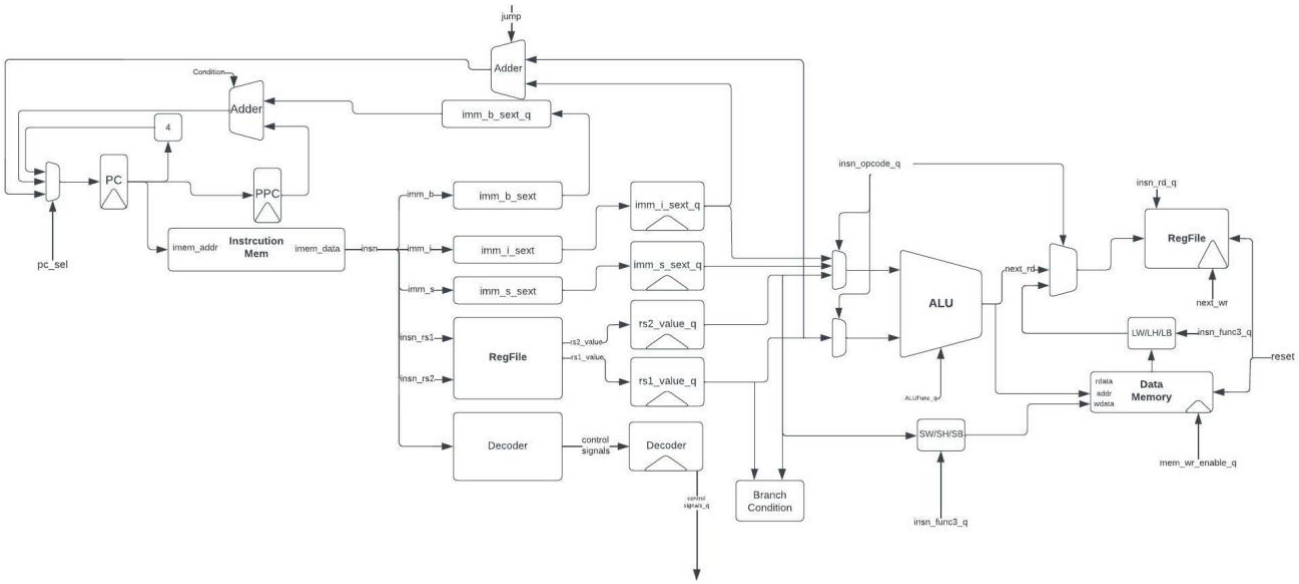

Figure III-B-1 Two-Stage Processor Datapath

In this implemented pipeline processor, the pipeline registers are placed before the ALU. By doing this, the data path is split into 2 parts, the first part includes IF and ID step, and the second part includes EX, MA and WB step. Hence, the critical data path is now 3, which is the second part, and theoretically the frequency of such pipeline processor should be faster than the single cycle processor.

The new processor data path is modified as shown in figure III-B-1. At each positive clock edge, the data values computed in ID stage will be fetch to the pipeline registers indicated using "_q". Hence, all the data values will now be delay by one cycle before they are executed. In addition, the program counter needs also a pipeline register called PPC to incorporate the branch instruction. This is because the execution of these instructions requires the program counter value when they are fetched from the instruction memory. And at the next cycle, the data that are stored inside the pipeline registers will proceed to the next step, EX. Meanwhile, the new 32-bit instruction is fetched from the instruction memory at the same time, and the new values after broken down the instruction will again occupy the pipeline register.

This does seem more efficient than single cycle processor but implementing pipelining can also introduced many hazards. Structural hazard arises when the same physical hardware is accessing by more than 1 pipeline stages. A data hazard occurs if an instruction reads a register that a previous instruction overwrites in the future cycle. Because the processor can only write the data to register file at the beginning of the cycle, the read instruction will fetch the value in the register file before the write starts in the current cycle. As a result, an old value will be read, and the computation is therefore incorrect. A good choice to use a Forwarding strategy. Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers. This is also called bypassing.[3] A simple structure of Forwarding is displayed below in figure III-B-2.

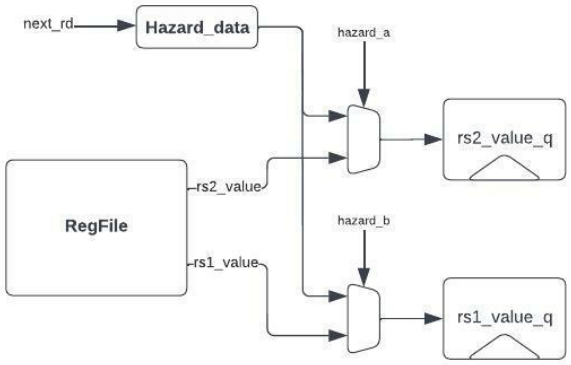

Figure III-B-2 Forwarding strategy

Here the hazard_data will take the computed result, next_rd, at the positive edge of the cycle and points to the multiplexers inserted before the pipelined rs values. The hazard control signals are computed by comparing the current write back register address and the register read addresses that is going to execute in the next cycle. If the address is the same, then there is a data hazard. And if there is hazard, then the hazard_data will be assigned to the particular rs value(s), since the hazard_data is the updated data value from the last computation, that just has not been overwrite in the register file. However, this strategy does not deal with control hazard in this processor. And because the natural property of a branch instruction, it not only involves an offset to the program counter, but also need to compare two values, the branch instruction in pipeline design has potential data and control hazards. Hence a new method needs also to resolve the hazard problem.

Control hazard occurs whenever the pipeline makes incorrect branch prediction decisions, resulting in instructions entering the pipeline that must be discarded. In the pipeline register, before any strategies is used to resolve hazard, if the current instruction is a branch or jump, the next instruction has been fetched and it is already waiting in the pipeline register.

In that case, the next instruction will be executed and overwrite regardless the branch condition. This is not wanted because in the case of a branch or jump, the next program counter is not necessarily PC + 4, so a PC + 4 instruction should not be executed sometimes. A solution to this problem is via stalling.

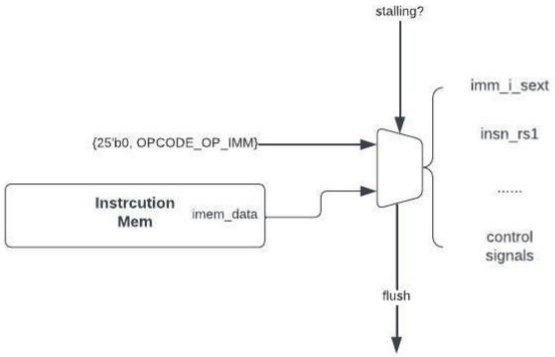

Figure III-B-3 Stalling strategy

Figure III-B-3 shows the stalling implementation of this two-stage pipeline processor. Like a 'NOP', if a control hazard happened, the next cycle will be filled with a meaningless instruction and the program counter will freeze. The instruction {25'b0, OPCODE_OP_IMM} here will write 0 to the zero register, and because the zero register is hardwired to 0, it would not conflict with the other instructions. This strategy has been used to deal with the branch and unconditional jump instructions. Because the property of the pipeline registers, while executing the current instruction, the next instruction is already waiting in the pipeline registers. Hence stalling here is to replace the next instruction with a meaningless instruction to accomplish a 'NOP'. One drawback from the implementation is that this cannot reduce the number of cycles for the program. In the pipeline design, Forwarding is used instead of Branch Prediction, this is due to the consideration that Forwarding can result in a greater reduction in the number of cycles compared to branch prediction, as a prediction is not always true.

IV. Result and Discussion

```
Info: Max frequency for clock 'clock$SB_IO_IN_$glb_clk': 37.02 MHz (PASS at 12.00 MHz)

Test summary
period: 2.701242571582928e-08
test1.result: x10=55 cycles=18 extime=4.86223662884927e-07 normextime=1.0
test2.result: x10=3790353928 cycles=15 extime=4.051863857374392e-07 normextime=1.0
test3.result: x10=45 cycles=40 extime=1.0804970286331712e-06 normextime=1.0
test4.result: x10=210 cycles=130 extime=3.5116153430578064e-06 normextime=1.0
test5.result: x10=21 cycles=942 extime=2.5445705024311184e-05 normextime=1.0
Geometric mean=1.0
```

Figure IV-1 Performance of single cycle processor

```
Info: Max frequency for clock 'clock$SB_IO_IN_$glb_clk': 40.04 MHz (PASS at 12.00 MHz)

Test summary
period: 2.8473804100227792e-08
test1.result: x10=55 cycles=19 extime=5.41002277904328e-07 normextime=1.112661351556568
test2.result: x10=3790353928 cycles=16 extime=4.5558086560364467e-07 normextime=1.124373576309795
test3.result: x10=45 cycles=50 extime=1.4236902050113896e-06 normextime=1.3176252847380412
test4.result: x10=210 cycles=191 extime=5.438496583143508e-06 normextime=1.5487164885228668
test5.result: x10=21 cycles=1215 extime=3.4595671981776765e-05 normextime=1.359587873423966
Geometric mean=1.282593255014273 3
```

Figure IV-2 Performance of pipelining processor

In comparison, the maximum frequency of the pipelining processor has a slightly increase by 3MHz. The number of cycles of the two simulation is different, the pipelining processor has more cycles than the single cycle processor. This is due to the use of stalling. Because stalling is to add 'NOP' to the instructions and freeze the program counter, the number of cycles it takes to complete the whole program is then increased. The rate of increasing is really depending on the number of stalling used, since load and branch and jump instruction will all insert stalling to the cycle after them every time.

Through additional simulation, if all the hazards are resolved using only stalling, the number of cycles for each test are 30, 20, 70, 215 and 1282. This is because the 'NOP' is inserted in a dummy way, so regardless the branch condition or if there is data hazard in a load and store instruction, the program will still freeze the PC and wait for an additional cycle. Hence, by using Forwarding strategy, we could see a meaningful drop on the number of cycles. This is due to the removal of several 'NOP's in the program via feeding the updated values to resolve some data hazards. Apart from the number of cycles, unfortunately, the period and execution time of the pipeline processor shows a negative impact. A potential reason could be that there are now more assignments and data to manipulate in one cycle, so it requires more time to complete them. Another reason for the negative improvement is because of bad coding in System Verilog.

V. Conclusion

Although the result for this pipelining seems to have a negative impact on the processor's performance overall, due the increase in execution time, the benefit of pipelining lies in the improvement in the maximum frequency of the design. Theoretically, the pipeline register can reduce the critical path for the processor, and hence reduce the cycle time.

Improvements can be made to the pipeline design in the future. The number of variables and assignments can be reduced by slightly change the strategy plan. The problem of the design is that too many assignments has increased the execution time, so change how stalling and Forwarding can be implemented using less variables could help with the execution time. Another potential improvement could be changing the strategy to resolve hazard, a branch prediction might also be a good idea to replace stalling as well, since it adds 'NOP' only when the branch condition is true, otherwise the instruction will proceed. This way can further reduce the cycle time, I expected. The stalling method can still be optimized in the design. As mentioned earlier, sometimes when the branch is not taken, there is no needed for a 'NOP', so here a branch prediction method could reduce the number of cycles even more. Also, the current design guarantees a 'NOP' after each load instruction, but in fact load instruction does not always have hazard problems. One can improve the method by considering the load instruction a little bit more by comparing the addresses of the successive instructions, so only insert 'NOP' after load instruction when a data hazard will happen.

References

[1] Andrew WaterMan; Krste Asanovic; SiFive Inc, CS Division, EECS Department, University of California, Berkeley, "***The RISC-V Instruction Set Manual***", https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[2] Whatls.com, "***pipelining***", https://www.techtarget.com/whatis/definition/pipelining

[3] DR A.P.SHANTHI, Computer Architecture, "***Handling Data Hazards***", https://www.cs.umd.edu/~meesh/411/CA-online/chapter/handling-data-hazards/index.html

APPENDIX A

SIMULATION RESULTS

```
Info: Max frequency for clock 'clock$SB_IO_IN_$glb_clk': 40.04 MHz (PASS at 12.00 MHz)
Test summary
period: 2.8473804100227792e-08
test1.result: x10=55 cycles=19 extime=5.41002277904328e-07 normextime=1.112661351556568
test2.result: x10=3790353928 cycles=16 extime=4.5558086560364467e-07 normextime=1.124373576309795
test3.result: x10=45 cycles=50 extime=1.4236902050113896e-06 normextime=1.3176252847380412
test4.result: x10=210 cycles=191 extime=5.438496583143508e-06 normextime=1.5487164885228668
test5.result: x10=21 cycles=1215 extime=3.4595671981776765e-05 normextime=1.359587873423966
Geometric mean=1.2825932550142733
```

TWO-STAGES PIPELINE PROCESSOR CODE

```
module nerv #(
    parameter [31:0] RESET_ADDR = 32'h 0000_0000,
    parameter integer NUMREGS = 32
) (
    input clock,
    input reset,
    output trap,
    output logic [31:0] x10,

    // we have 2 external memories
    // one is instruction memory
    output [31:0] imem_addr,
    input  [31:0] imem_data,

    // the other is data memory
    output        dmem_valid,
    output [31:0] dmem_addr,
    output [ 3:0] dmem_wstrb,
    output [31:0] dmem_wdata,
    input  [31:0] dmem_rdata
);
    logic mem_wr_enable;
    logic [31:0] mem_wr_addr;
    logic [31:0] mem_wr_data;
    logic [3:0] mem_wr_strb;

    logic mem_rd_enable;
    logic [31:0] mem_rd_addr;
    logic [4:0] mem_rd_reg;
    logic [4:0] mem_rd_func;

    logic mem_rd_enable_q;
    logic [4:0] mem_rd_reg_q;
    logic [4:0] mem_rd_func_q;
```

```
// delayed copies of mem_rd
always @(posedge clock) begin
    mem_rd_enable_q <= mem_rd_enable;
    mem_rd_reg_q <= mem_rd_reg;
    mem_rd_func_q <= mem_rd_func;
    if (reset) begin
        mem_rd_enable_q <= 0;
    end
end

// memory signals
assign dmem_valid = mem_wr_enable || mem_rd_enable;
assign dmem_addr  = mem_wr_enable ? mem_wr_addr : mem_rd_enable ? mem_rd_addr : 32'h x;
assign dmem_wstrb = mem_wr_enable ? mem_wr_strb : mem_rd_enable ? 4'h 0 : 4'h x;
assign dmem_wdata = mem_wr_enable ? mem_wr_data : 32'h x;

// registers, instruction reg, program counter, next pc
logic [31:0] regfile [0:NUMREGS-1];
logic [31:0] insn;
logic [31:0] insn_q;
logic [31:0] npc;
logic [31:0] pc;
logic [31:0] ppc;

logic [31:0] imem_addr_q;

// instruction memory pointer
assign imem_addr = (trap || mem_rd_enable_q) ? imem_addr_q : npc;
assign insn = imem_data;

// rs1 and rs2 are source for the instruction
logic [31:0] rs1_value;
logic [31:0] rs2_value;
logic [31:0] rs1_value_q;
logic [31:0] rs2_value_q;


always @(posedge clock) begin
    imem_addr_q <= imem_addr;
    ppc <= pc;

    if((npc != pc +4) || (insn_opcode_q == OPCODE_JAL) || insn_opcode_q == OPCODE_JALR) begin
        insn_q <= {25'b0, OPCODE_OP_IMM};
        flush <= 1;
    end else begin
        rs1_value <= !insn_rs1 ? 0 : regfile[insn_rs1];
        rs2_value <= !insn_rs2 ? 0 : regfile[insn_rs2];
        imm_i_sext <= $signed(imm_i);
        imm_s_sext <= $signed(imm_s);
        imm_b_sext <= $signed(imm_b);
        imm_j_sext <= $signed(imm_j);
        insn_q <= insn;
        flush <= 0;
    end
end
assign rs1_value_q = hazard_a? hazard_data : rs1_value;
assign rs2_value_q = hazard_b? hazard_data : rs2_value;

// components of the instruction
logic [6:0] insn_funct7;
logic [4:0] insn_rs2;
logic [4:0] insn_rs1;
logic [2:0] insn_funct3;
logic [4:0] insn_rd;
logic [6:0] insn_opcode;

// split R-type instruction - see section 2.2 of RiscV spec
assign {insn_funct7, insn_rs2, insn_rs1, insn_funct3, insn_rd, insn_opcode} = insn;

logic [6:0] insn_funct7_q;
logic [4:0] insn_rs2_q;
logic [4:0] insn_rs1_q;
logic [2:0] insn_funct3_q;
logic [4:0] insn_rd_q;
logic [6:0] insn_opcode_q;

// split R-type instruction - see section 2.2 of RiscV spec
assign {insn_funct7_q, insn_rs2_q, insn_rs1_q, insn_funct3_q, insn_rd_q, insn_opcode_q} = insn_q;

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

// S - stores
wire [11:0] imm_s;
assign imm_s[11:5] = insn_funct7, imm_s[4:0] = insn_rd;

// B - conditionals
wire [12:0] imm_b;
assign {imm_b[12], imm_b[10:5]} = insn_funct7, {imm_b[4:1], imm_b[11]} = insn_rd, imm_b[0] = 1'b0;

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

logic [31:0] imm_i_sext_q;
logic [31:0] imm_s_sext_q;
logic [31:0] imm_b_sext_q;
logic [31:0] imm_j_sext_q;
logic [31:0] imm_i_sext;
logic [31:0] imm_s_sext;
logic [31:0] imm_b_sext;
logic [31:0] imm_j_sext;
```

```
assign imm_i_sext_q = imm_i_sext;
assign imm_s_sext_q = imm_s_sext;
assign imm_b_sext_q = imm_b_sext;
assign imm_j_sext_q = imm_j_sext;

// opcodes - see section 19 of RiscV spec
localparam OPCODE_LOAD       = 7'b 00_000_11;
localparam OPCODE_STORE      = 7'b 01_000_11;
localparam OPCODE_MADD       = 7'b 10_000_11;
localparam OPCODE_BRANCH     = 7'b 11_000_11;

localparam OPCODE_LOAD_FP    = 7'b 00_001_11;
localparam OPCODE_STORE_FP   = 7'b 01_001_11;
localparam OPCODE_MSUB       = 7'b 10_001_11;
localparam OPCODE_JALR       = 7'b 11_001_11;

localparam OPCODE_CUSTOM_0   = 7'b 00_010_11;
localparam OPCODE_CUSTOM_1   = 7'b 01_010_11;
localparam OPCODE_NMSUB      = 7'b 10_010_11;
localparam OPCODE_RESERVED_0 = 7'b 11_010_11;

localparam OPCODE_MISC_MEM   = 7'b 00_011_11;
localparam OPCODE_AMO        = 7'b 01_011_11;
localparam OPCODE_NMADD      = 7'b 10_011_11;
localparam OPCODE_JAL        = 7'b 11_011_11;

localparam OPCODE_OP_IMM     = 7'b 00_100_11;
localparam OPCODE_OP         = 7'b 01_100_11;
localparam OPCODE_OP_FP      = 7'b 10_100_11;
localparam OPCODE_SYSTEM     = 7'b 11_100_11;

localparam OPCODE_AUIPC      = 7'b 00_101_11;
localparam OPCODE_LUI        = 7'b 01_101_11;
localparam OPCODE_RESERVED_1 = 7'b 10_101_11;
localparam OPCODE_RESERVED_2 = 7'b 11_101_11;
```

```
localparam OPCODE_OP_IMM_32  = 7'b 00_110_11;
localparam OPCODE_OP_32      = 7'b 01_110_11;
localparam OPCODE_CUSTOM_2   = 7'b 10_110_11;
localparam OPCODE_CUSTOM_3   = 7'b 11_110_11;

// next write, next destination (rd), illegal instruction registers
logic next_wr;
logic [31:0] next_rd;
logic illinsn;

// logic [5:0] prev_rd;
logic [31:0] hazard_data;
logic hazard_a;
logic hazard_b;
logic flush;

logic trapped;
logic trapped_q;
assign trap = trapped;

always_comb begin

    npc = pc + 4;

    // defaults for read, write
    next_wr = 0;
    next_rd = 0;
    illinsn = 0;

    mem_wr_enable = 0;
    mem_wr_addr = 'hx;
    mem_wr_data = 'hx;
    mem_wr_strb = 'hx;


mem_rd_enable = 0;
mem_rd_addr = 'hx;
mem_rd_reg = 'hx;
mem_rd_func = 'hx;

// act on opcodes
case (insn_opcode_q)
    // Load Upper Immediate
    OPCODE_LUI: begin
        next_wr = 1;
        next_rd = insn_q[31:12] << 12;
    end
    // Add Upper Immediate to Program Counter
    OPCODE_AUIPC: begin
        next_wr = 1;
        next_rd = (insn_q[31:12] << 12) + pc;
    end
    // Jump And Link (unconditional jump)
    OPCODE_JAL: begin
        next_wr = 1;
        next_rd = npc - 4;
        npc = ppc + imm_j_sext_q;
        if (npc & 32'b 11) begin
            illinsn = 1;
            npc = npc & ~32'b 11;
        end
    end
```

```
OPCODE_BRANCH: begin
    case (insn_funct3_q)
        3'b 000 /* BEQ  */: begin if (rs1_value_q == rs2_value_q) npc = ppc + imm_b_sext_q; end
        3'b 001 /* BNE  */: begin if (rs1_value_q != rs2_value_q) npc = ppc + imm_b_sext_q; end
        3'b 100 /* BLT  */: begin if ($signed(rs1_value_q) < $signed(rs2_value_q)) npc = ppc + imm_b_sext_q; end
        3'b 101 /* BGE  */: begin if ($signed(rs1_value_q) >= $signed(rs2_value_q)) npc = ppc + imm_b_sext_q; end
        3'b 110 /* BLTU */: begin if (rs1_value_q < rs2_value_q) npc = ppc + imm_b_sext_q; end
        3'b 111 /* BGEU */: begin if (rs1_value_q >= rs2_value_q) npc = ppc + imm_b_sext_q; end
        default: illinsn = 1;
    endcase
    if (npc & 32'b 11) begin
        illinsn = 1;
        npc = npc & ~32'b 11;
    end
end
// load from memory into rd: Load Byte, Load Halfword, Load Word, Load Byte Unsigned, Load Halfword Unsigned
OPCODE_LOAD: begin
    mem_rd_addr = rs1_value_q + imm_i_sext_q;
    casez ({insn_funct3_q, mem_rd_addr[1:0]})
        5'b 000_zz /* LB  */,
        5'b 001_z0 /* LH  */,
        5'b 010_00 /* LW  */,
        5'b 100_zz /* LBU */,
        5'b 101_z0 /* LHU */: begin
            mem_rd_enable = 1;
            mem_rd_reg = insn_rd_q;
            mem_rd_func = {mem_rd_addr[1:0], insn_funct3_q};
            mem_rd_addr = {mem_rd_addr[31:2], 2'b 00};
        end
        default: illinsn = 1;
    endcase
    npc = pc;
end
OPCODE_STORE: begin
    mem_wr_addr = rs1_value_q + imm_s_sext_q;
    casez ({insn_funct3_q, mem_wr_addr[1:0]})
        5'b 000_zz /* SB */,
        5'b 001_z0 /* SH */,
        5'b 010_00 /* SW */: begin
            mem_wr_enable = 1;
            mem_wr_data = rs2_value_q;
            mem_wr_strb = 4'b 1111;
            case (insn_funct3_q)
                3'b 000 /* SB  */: begin mem_wr_strb = 4'b 0001; end
                3'b 001 /* SH  */: begin mem_wr_strb = 4'b 0011; end
                3'b 010 /* SW  */: begin mem_wr_strb = 4'b 1111; end
            endcase
            mem_wr_data = mem_wr_data << (8*mem_wr_addr[1:0]);
            mem_wr_strb = mem_wr_strb << mem_wr_addr[1:0];
            mem_wr_addr = {mem_wr_addr[31:2], 2'b 00};
        end
        default: illinsn = 1;
    endcase
end

OPCODE_OP_IMM: begin
    casez ({insn_funct7_q, insn_funct3_q})
        10'b zzzzzzz_000 /* ADDI  */: begin next_wr = 1; next_rd = rs1_value_q + imm_i_sext_q; end
        10'b zzzzzzz_010 /* SLTI  */: begin next_wr = 1; next_rd = $signed(rs1_value_q) < $signed(imm_i_sext_q); end
        10'b zzzzzzz_011 /* SLTIU */: begin next_wr = 1; next_rd = rs1_value_q < imm_i_sext_q; end
        10'b zzzzzzz_100 /* XORI  */: begin next_wr = 1; next_rd = rs1_value_q ^ imm_i_sext_q; end
        10'b zzzzzzz_110 /* ORI   */: begin next_wr = 1; next_rd = rs1_value_q | imm_i_sext_q; end
        10'b zzzzzzz_111 /* ANDI  */: begin next_wr = 1; next_rd = rs1_value_q & imm_i_sext_q; end
        10'b 0000000_001 /* SLLI  */: begin next_wr = 1; next_rd = rs1_value_q << insn_q[24:20]; end
        10'b 0000000_101 /* SRLI  */: begin next_wr = 1; next_rd = rs1_value_q >> insn_q[24:20]; end
        10'b 0100000_101 /* SRAI  */: begin next_wr = 1; next_rd = $signed(rs1_value_q) >>> insn_q[24:20]; end
        default: illinsn = 1;
    endcase
end
    OPCODE_OP: begin
        case ({insn_funct7_q, insn_funct3_q})
            10'b 0000000_000 /* ADD  */: begin next_wr = 1; next_rd = rs1_value_q + rs2_value_q; end
            10'b 0100000_000 /* SUB  */: begin next_wr = 1; next_rd = rs1_value_q - rs2_value_q; end
            10'b 0000000_001 /* SLL  */: begin next_wr = 1; next_rd = rs1_value_q << rs2_value_q[4:0]; end
            10'b 0000000_010 /* SLT  */: begin next_wr = 1; next_rd = $signed(rs1_value_q) < $signed(rs2_value_q); end
            10'b 0000000_011 /* SLTU */: begin next_wr = 1; next_rd = rs1_value_q < rs2_value_q; end
            10'b 0000000_100 /* XOR  */: begin next_wr = 1; next_rd = rs1_value_q ^ rs2_value_q; end
            10'b 0000000_101 /* SRL  */: begin next_wr = 1; next_rd = rs1_value_q >> rs2_value_q[4:0]; end
            10'b 0100000_101 /* SRA  */: begin next_wr = 1; next_rd = $signed(rs1_value_q) >>> rs2_value_q[4:0]; end
            10'b 0000000_110 /* OR   */: begin next_wr = 1; next_rd = rs1_value_q | rs2_value_q; end
            10'b 0000000_111 /* AND  */: begin next_wr = 1; next_rd = rs1_value_q & rs2_value_q; end
            default: illinsn = 1;
        endcase
    end
    default: illinsn = 1;
endcase

// if last cycle was a memory read, then this cycle is the 2nd part of it and imem_data will not be a valid instruction
if (mem_rd_enable_q) begin
    npc = pc;
    next_wr = 0;
    illinsn = 0;
    mem_rd_enable = 0;
    mem_wr_enable = 0;
end
```

```
    // reset
    if (reset || reset_q) begin
        npc = RESET_ADDR;
        next_wr = 0;
        illinsn = 0;
        mem_rd_enable = 0;
        mem_wr_enable = 0;
    end
end

logic reset_q;
logic [31:0] mem_rdata;

// mem read functions: Lower and Upper Bytes, signed and unsigned
always_comb begin
    mem_rdata = dmem_rdata >> (8*mem_rd_func_q[4:3]);
    case (mem_rd_func_q[2:0])
        3'b 000 /* LB  */: begin mem_rdata = $signed(mem_rdata[7:0]); end
        3'b 001 /* LH  */: begin mem_rdata = $signed(mem_rdata[15:0]); end
        3'b 100 /* LBU */: begin mem_rdata = mem_rdata[7:0]; end
        3'b 101 /* LHU */: begin mem_rdata = mem_rdata[15:0]; end
    endcase
end

// every cycle
always @(posedge clock) begin

    reset_q <= reset;
    trapped_q <= trapped;

    // increment pc if possible
    if (!trapped && !reset && !reset_q) begin
        if (illinsn)
            trapped <= 1;

        pc <= npc;
```

```
        if(!flush && insn_rd_q == insn_rs1) begin
            hazard_a <= 1;
            hazard_data <= next_rd;
        end else if (!flush && insn_rd_q == insn_rs2 && insn_opcode != OPCODE_OP_IMM) begin
            hazard_b <= 1;
            hazard_data <= next_rd;
        end else begin
            hazard_a <= 0;
            hazard_b <= 0;
        end

        // update registers from memory or rd (destination)
        if (mem_rd_enable_q || next_wr) begin
            regfile[mem_rd_enable_q ? mem_rd_reg_q : insn_rd_q] <= mem_rd_enable_q ? mem_rdata : next_rd;
        end
        x10 <= regfile[10];
    end

    // reset
    if (reset || reset_q) begin
        pc <= RESET_ADDR - (reset ? 4 : 0);
        trapped <= 0;
    end
end

endmodule
```

DESIGN LOG

```
Info: Max frequency for clock 'clock$SB_IO_IN_$glb_clk': 40.04 MHz (PASS at 12.00 MHz)

Info: Max delay <async>                          -> <async>                          : 10.64 ns
Info: Max delay <async>                          -> posedge clock$SB_IO_IN_$glb_clk: 23.53 ns
Info: Max delay posedge clock$SB_IO_IN_$glb_clk -> <async>                          : 23.43 ns

Info: Slack histogram:
Info:  legend: * represents 13 endpoint(s)
Info:          + represents [1,13) endpoint(s)
Info: [ 58358,  59551) |****************+
Info: [ 59551,  60744) |*****+
Info: [ 60744,  61937) |******************+
Info: [ 61937,  63130) |************************************************************+
Info: [ 63130,  64323) |*****************************************************************+
Info: [ 64323,  65516) |******************+
Info: [ 65516,  66709) |*********************************+
Info: [ 66709,  67902) |************************************************+
Info: [ 67902,  69095) |***+
Info: [ 69095,  70288) |****************+
Info: [ 70288,  71481) |**********+
Info: [ 71481,  72674) |*************************************************+
Info: [ 72674,  73867) |*******************************+
Info: [ 73867,  75060) |************+
Info: [ 75060,  76253) |*********************+
Info: [ 76253,  77446) |************+
Info: [ 77446,  78639) |****+
Info: [ 78639,  79832) |*****+
Info: [ 79832,  81025) |****************+
Info: [ 81025,  82218) |*******+
201 warnings, 0 errors

Info: Program finished normally.
```