Computer Organization (H) Project Report

Mengxuan Wu Taojie Wang Sirui Wu

Friday 31st May, 2024

1 Developer Contribution

Name	Student ID	Contribution	Percentage
Mengxuan Wu	12212006	Pipeline Design and Hazard Handling	33.3%
Taojie Wang	12210519	Pipeline Design and Testing	33.3%
Sirui Wu	12210122	RISC-V 32 Assembly Code Writing	33.3%

2 CPU Design

2.1 CPU Feature

- ISA: RISC-V 32 Basic Instruction Set, Support 32 32-bit registers, No exception handling
- CPU Clock: 5-stage Pipeline, 23MHz, Handle hazard by forwarding and stalling
- Storage: Harvard architecture, 64KB each for instruction and data memory
- IO Design: MMIO, IO device address begins at `0xFFC0`

2.2 CPU Ports

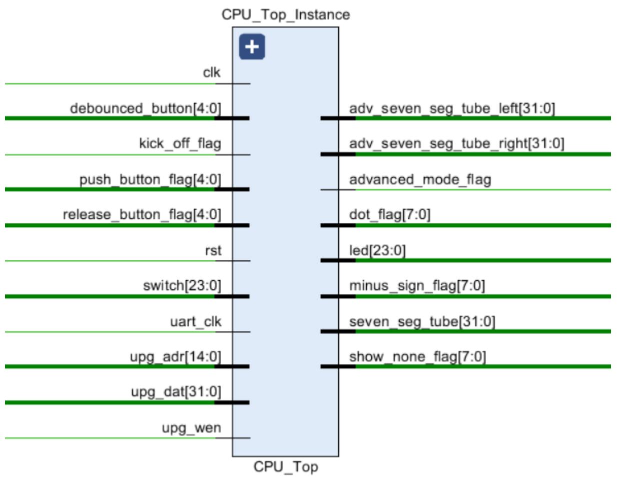


Figure 1: CPU

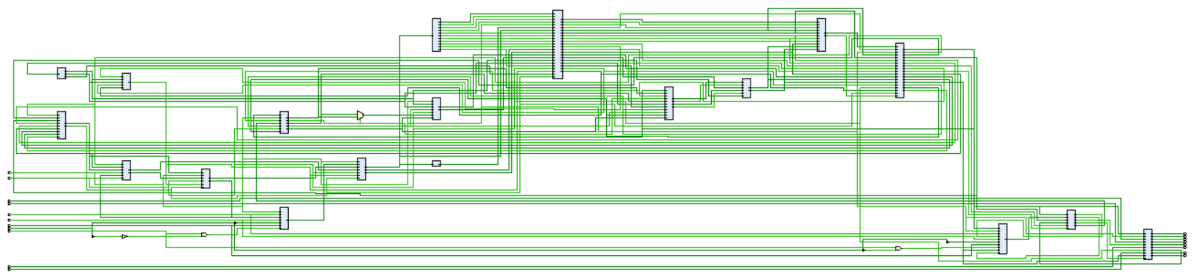
2.2.1 Input

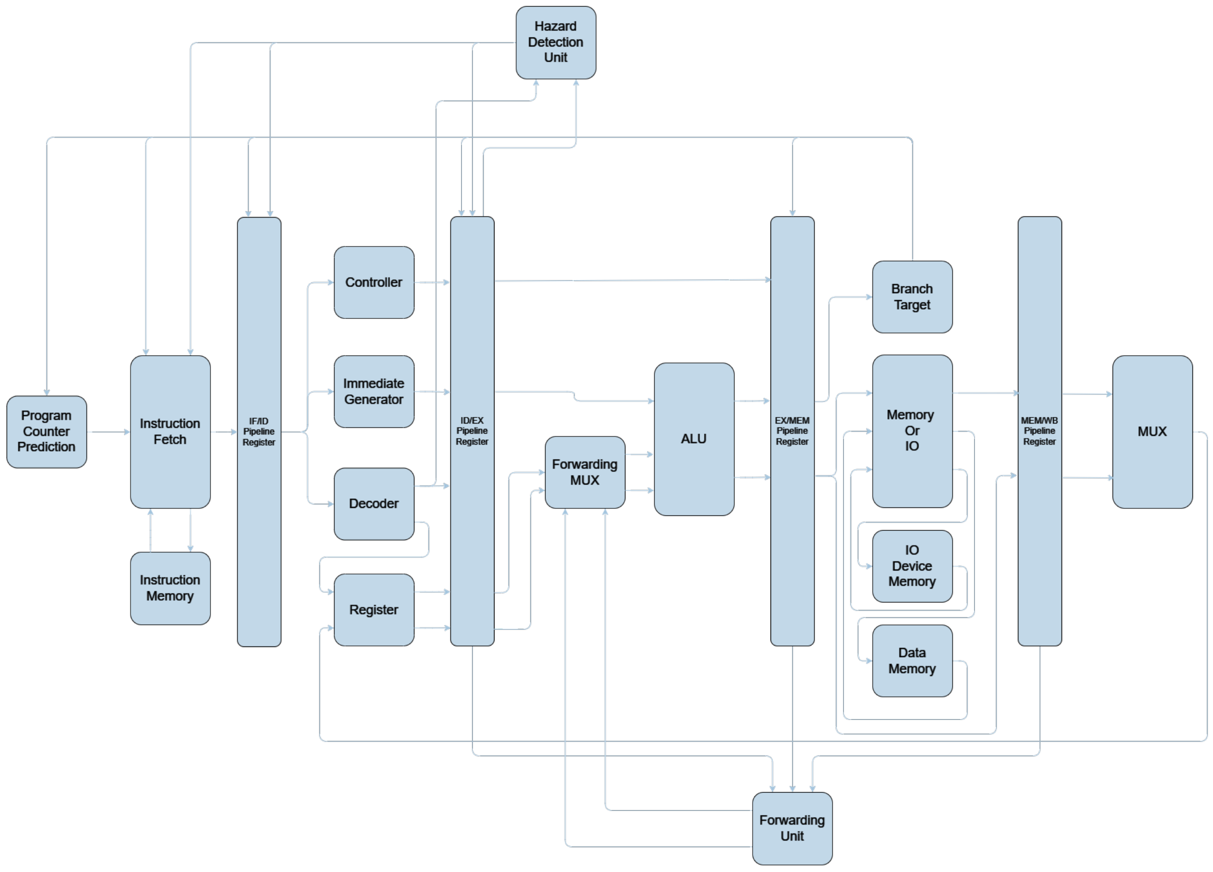
Related Module	Port Name	Port Width	Description
Clock	clk	1	Clock signal generated by IP core
Reset	rst	1	Reset signal (high-enable)
IO Device Input	debounced_button	5	Debounced button signal
	push_button_flag	5	Flag indicating a button push down event
	release_button_flag	5	Flag indicating a button release event
	switch	24	Switch signal
Uart Signal	kick_off_flag	1	Flag indicating communication mode or working mode
	uart_clk	1	Clock for UART module(23MHz)
	upg_adr	15	Address to write in memory used by UART mode
	upd_dat	32	Data to write into memory by UART mode
	upg_wen	1	Write enable in UART mode

2.2.2 Output

Related Module	Port Name	Width	Explanation
Led Output	led	24	LED output signal for displaying status or information
Seven Segment Tube Display Output	seven_seg_tube	32	Output signal for the seven segment tube display
	advanced_mode_flag	1	Flag indicating whether advanced mode is activated
	adv_seven_seg_tube_left	32	Output signal for the left part of the advanced seven segment tube display
	adv_seven_seg_tube_right	32	Output signal for the right part of the advanced seven segment tube display
	dot_flag	8	Flag indicating which dots should be illuminated on the display
	minus_sign_flag	8	Flag indicating whether a minus sign should be displayed
	show_none_flag	8	Flag indicating whether the display should be turned off

2.3 CPU Structure





2.3.1 Top

Port Name	Port Type	Port Width	Description
raw_clk	input	1	Raw clock signal
raw_rst	input	1	Raw reset signal
switch	input	24	Switch inputs
button	input	5	Button inputs
led	output	24	LED outputs
tube_select_onehot	output	8	One-hot selection for tube display
tube_shape	output	8	Shape data for tube display
rx	input	1	UART receive signal
tx	output	1	UART transmit signal

Description The `Top` module integrates various sub-modules to form the complete system. It includes the main clock, debouncer, UART, CPU, and seven-segment tube driver modules.

2.3.2 CPU_Main_Clock_ip

Port Name	Port Type	Port Width	Description
clk_in1	input	1	Raw clock input
clk_out1	output	1	CPU clock output (23MHz)
clk_out2	output	1	UART clock output (10MHz)

Description The `CPU_Main_Clock_ip` module generates two clock signals from the raw clock input: a 23MHz clock for the CPU and a 10MHz clock for the UART.

2.3.3 Debouncer

Port Name	Port Type	Port Width	Description
raw_clk	input	1	Raw clock signal
rst	input	1	Reset signal
button	input	5	Button inputs
debounced_button	output	5	Debounced button signals
push_button_flag	output	5	Push button flag signals
release_button_flag	output	5	Release button flag signals

Description The `Debouncer` module debounces the raw button inputs, providing stable button signals and flagging push and release events.

2.3.4 UART

Port Name	Port Type	Port Width	Description
uart_clk_in	input	1	UART clock input (10MHz)
raw_clk	input	1	Raw clock signal
raw_rst	input	1	Raw reset signal
start_pg	input	1	Start program signal from push button
kick_off_flag	output	1	Mode flag (1: working, 0: communication)
uart_clk_out	output	1	UART clock output
upg_wen	output	1	UART write enable
upg_adr	output	15	UART address output
upg_dat	output	32	UART data output
rst	output	1	Reset signal
rx	input	1	UART receive signal
tx	output	1	UART transmit signal

Description The `UART` module handles UART communication, providing interface signals for data transmission and reception, and managing write operations to memory.

2.3.5 CPU_Top

Port Name	Port Type	Port Width	Description
clk	input	1	CPU clock signal
rst	input	1	Reset signal
switch	input	24	Switch inputs
debounced_button	input	5	Debounced button inputs
push_button_flag	input	5	Push button flag signals
release_button_flag	input	5	Release button flag signals
led	output	24	LED outputs
seven_seg_tube	output	32	Seven-segment tube data
minus_sign_flag	output	8	Minus sign flag for tube display
dot_flag	output	8	Dot flag for tube display
show_none_flag	output	8	Show none flag for tube display
advanced_mode_flag	output	1	Advanced mode flag
adv_seven_seg_tube_left	output	32	Advanced mode left tube data
adv_seven_seg_tube_right	output	32	Advanced mode right tube data
kick_off_flag	input	1	Mode flag (1: working, 0: communication)
uart_clk	input	1	UART clock signal
upg_wen	input	1	UART write enable
upg_adr	input	15	UART address input
upg_dat	input	32	UART data input

Description The `CPU_Top` module is the main CPU module, interfacing with various inputs and outputs, including switches, debounced buttons, LEDs, and seven-segment tube displays. It also manages UART communication signals.

2.3.6 Seven_Seg_Tube_Driver

Port Name	Port Type	Port Width	Description
raw_clk	input	1	Raw clock signal
rst	input	1	Reset signal
data	input	32	Data for seven-segment tube display
minus_sign_flag	input	8	Minus sign flag
dot_flag	input	8	Dot flag
show_none_flag	input	8	Show none flag
advanced_mode_flag	input	1	Advanced mode flag
adv_seven_seg_tube_left	input	32	Advanced mode left tube data
adv_seven_seg_tube_right	input	32	Advanced mode right tube data
tube_select_onehot	output	8	One-hot selection for tube display
tube_shape	output	8	Shape data for tube display

Description The `Seven_Seg_Tube_Driver` module drives the seven-segment tube display, handling various display flags and data inputs to control the tube selection and shape output.

2.3.7 Instruction_Fetch

Port Name	Port Type	Port Width	Description
clk	input	1	Clock signal
rst	input	1	Reset signal
wrong_prediction_flag	input	1	Wrong branch prediction flag
branch_pc	input	32	Branch target address
stall_flag	input	1	Stall signal
program_counter_prediction	output	32	Predicted program counter
program_counter	output	32	Current program counter
prev_pc	output	32	Previous program counter

Description The `Instruction_Fetch` module is responsible for fetching instructions from the instruction memory. It handles the program counter logic, including branch prediction and stall conditions.

2.3.8 Instruction_Memory

Port Name	Port Type	Port Width	Description
clk	input	1	Clock signal
program_counter	input	32	Address of the current instruction
inst	output	32	Instruction data
kick_off_flag	input	1	UART kick-off flag
uart_clk	input	1	UART clock signal
upg_wen	input	1	UART write enable
upg_adr	input	14	UART write address
upg_dat	input	32	UART write data

Description The `Instruction_Memory` module fetches the instruction data from the instruction memory based on the program counter. It also supports UART-based updates for loading new instructions.

2.3.9 Program_Counter_Prediction

Port Name	Port Type	Port Width	Description
clk	input	1	Clock signal
rst	input	1	Reset signal
branch_from_pc	input	32	Branch origin address
branch_to_pc	input	32	Branch target address
program_counter	input	32	Current program counter
branch_flag	input	1	Branch flag
program_counter_prediction	output	32	Predicted program counter
prev_pcp	output	32	Previous program counter prediction

Description The `Program_Counter_Prediction` module predicts the next program counter value based on branch instructions. It updates the program counter prediction for the instruction fetch stage.

2.3.10 Controller

Port Name	Port Type	Port Width	Description
inst	input	32	Instruction data
branch_flag	output	1	Branch flag
ALU_operation	output	2	ALU operation code
ALU_src_flag	output	1	ALU source flag
mem_read_flag	output	1	Memory read flag
mem_write_flag	output	1	Memory write flag
mem_to_reg_flag	output	1	Memory to register flag
reg_write_flag	output	1	Register write flag
jal_flag	output	1	JAL instruction flag
jalr_flag	output	1	JALR instruction flag
lui_flag	output	1	LUI instruction flag
auipc_flag	output	1	AUIPC instruction flag

Description The `Controller` module decodes the instruction and generates control signals for the ALU, memory access, and register write operations. It also handles specific instruction flags for branching and immediate operations.

2.3.11 Immediate_Generator

Port Name	Port Type	Port Width	Description
inst	input	32	Instruction data
imme	output	32	Immediate value

Description The `Immediate_Generator` module extracts the immediate value from the instruction, which is used for ALU operations and memory addressing.

2.3.12 Decoder

Port Name	Port Type	Port Width	Description
inst	input	32	Instruction data
read_reg_idx_1	output	5	Read register index 1
read_reg_idx_2	output	5	Read register index 2
write_reg_idx	output	5	Write register index

Description The `Decoder` module decodes the instruction to determine the source and destination register indices for the register file operations.

2.3.13 Register

Port Name	Port Type	Port Width	Description
clk	input	1	Clock signal
rst	input	1	Reset signal
reg_write_flag	input	1	Register write flag
read_reg_idx_1	input	5	Read register index 1
read_reg_idx_2	input	5	Read register index 2
write_reg_idx	input	5	Write register index
write_data	input	32	Data to be written to the register
read_data_1	output	32	Data read from register 1
read_data_2	output	32	Data read from register 2

Description The `Register` module implements the register file, providing read and write access to the CPU registers based on the control signals and register indices.

2.3.14 Forwarding_Mux

Port Name	Port Type	Port Width	Description
read_data_1_raw	input	32	Raw data read from register 1
read_data_2_raw	input	32	Raw data read from register 2
read_data_1_forwarding	input	32	Forwarded data for register 1
read_data_2_forwarding	input	32	Forwarded data for register 2
read_data_1_forwarding_flag	input	1	Forwarding flag for register 1
read_data_2_forwarding_flag	input	1	Forwarding flag for register 2
read_data_1	output	32	Data for ALU input 1
read_data_2	output	32	Data for ALU input 2

Description The `Forwarding_Mux` module selects the appropriate data for the ALU inputs, either from the raw register data or from forwarded data, based on the forwarding flags.

2.3.15 ALU

Port Name	Port Type	Port Width	Description
read_data_1	input	32	Data for ALU input 1
read_data_2	input	32	Data for ALU input 2
imme	input	32	Immediate value
ALU_operation	input	2	ALU operation code
ALU_src_flag	input	1	ALU source flag
inst	input	32	Instruction data
program_counter	input	32	Program counter
jal_flag	input	1	JAL instruction flag
jalr_flag	input	1	JALR instruction flag
lui_flag	input	1	LUI instruction flag
auipc_flag	input	1	AUIPC instruction flag
ALU_result	output	32	ALU result
zero_flag	output	1	Zero result flag

Description The `ALU` module performs arithmetic and logical operations based on the control signals and input data. It produces the result and a zero flag indicating if the result is zero.

2.3.16 Branch_Target

Port Name	Port Type	Port Width	Description
jal_flag	input	1	JAL instruction flag
jalr_flag	input	1	JALR instruction flag
branch_flag	input	1	Branch flag
zero_flag	input	1	Zero result flag
read_data_1	input	32	Data read from register 1
imme	input	32	Immediate value
program_counter	input	32	Current program counter
inst	input	32	Instruction data
program_counter_prediction	input	32	Predicted program counter
wrong_prediction_flag	output	1	Wrong prediction flag
branch_pc	output	32	Branch target address

Description The `Branch_Target` module calculates the target address for branch instructions and checks for branch prediction correctness.

2.3.17 Memory_Or_IO

Port Name	Port Type	Port Width	Description
address_absolute	input	32	Absolute address for memory or IO access
inst	input	32	Instruction data
mem_read_flag	input	1	Memory read flag
mem_write_flag	input	1	Memory write flag
data_memory_read_data	input	32	Data read from data memory
io_device_read_data	input	32	Data read from IO device
read_data	output	32	Data read from memory or IO
data_memory_read_flag	output	1	Data memory read flag
data_memory_write_flag	output	1	Data memory write flag
io_device_read_flag	output	1	IO device read flag
io_device_write_flag	output	1	IO device write flag

Description The `Memory_Or_IO` module determines whether a memory or IO access should occur and manages the data transfer between the CPU and the respective memory or IO device.

2.3.18 Data_Memory

Port Name	Port Type	Port Width	Description
clk	input	1	Clock signal
address_absolute	input	32	Absolute address for memory access
write_data	input	32	Data to be written to memory
read_flag	input	1	Memory read flag
write_flag	input	1	Memory write flag
read_data	output	32	Data read from memory
kick_off_flag	input	1	UART kick-off flag
uart_clk	input	1	UART clock signal
upg_wen	input	1	UART write enable
upg_adr	input	14	UART write address
upg_dat	input	32	UART write data

Description The `Data_Memory` module handles data storage and retrieval operations. It supports UART-based updates for loading new data into memory.

2.3.19 IO_Device_Memory

Port Name	Port Type	Port Width	Description
clk	input	1	Clock signal
rst	input	1	Reset signal
address_absolute	input	32	Absolute address for IO access
write_data	input	32	Data to be written to IO device
read_flag	input	1	IO read flag
write_flag	input	1	IO write flag
switch	input	24	Switch inputs
debounced_button	input	5	Debounced button signals
push_button_flag	input	5	Push button flag signals
release_button_flag	input	5	Release button flag signals
read_data	output	32	Data read from IO device
led	output	24	LED outputs
seven_seg_tube	output	32	Seven-segment tube data
minus_sign_flag	output	8	Minus sign flag for tube display
dot_flag	output	8	Dot flag for tube display
show_none_flag	output	8	Show none flag for tube display
advanced_mode_flag	output	1	Advanced mode flag
adv_seven_seg_tube_left	output	32	Advanced mode left tube data
adv_seven_seg_tube_right	output	32	Advanced mode right tube data

Description The `IO_Device_Memory` module manages the interaction with IO devices, including switches, buttons, LEDs, and seven-segment displays. It handles both reading from and writing to IO devices.

2.3.20 Forwarding_Unit

Port Name	Port Type	Port Width	Description
ALU_result_MEM	input	32	ALU result from MEM stage
write_reg_idx_MEM	input	5	Write register index from MEM stage
write_reg_flag_MEM	input	1	Register write flag from MEM stage
mem_to_reg_flag_MEM	input	1	Memory to register flag from MEM stage
ALU_result_WB	input	32	ALU result from WB stage
read_data_WB	input	32	Data read from memory or IO in WB stage
write_reg_idx_WB	input	5	Write register index from WB stage
write_reg_flag_WB	input	1	Register write flag from WB stage
mem_to_reg_flag_WB	input	1	Memory to register flag from WB stage
read_reg_idx_1_EX	input	5	Read register index 1 from EX stage
read_reg_idx_2_EX	input	5	Read register index 2 from EX stage
read_data_1_forwarding	output	32	Forwarded data for register 1
read_data_2_forwarding	output	32	Forwarded data for register 2
read_data_1_forwarding_flag	output	1	Forwarding flag for register 1
read_data_2_forwarding_flag	output	1	Forwarding flag for register 2

Description The `Forwarding_Unit` module resolves data hazards by forwarding the appropriate data from MEM and WB stages to the EX stage, ensuring that the ALU has the correct operands.

2.3.21 Hazard_Detection_Unit

Port Name	Port Type	Port Width	Description
write_reg_idx_EX	input	5	Write register index from EX stage
write_reg_flag_EX	input	1	Register write flag from EX stage
mem_to_reg_flag_EX	input	1	Memory to register flag from EX stage
read_reg_idx_1_ID	input	5	Read register index 1 from ID stage
read_reg_idx_2_ID	input	5	Read register index 2 from ID stage
stall_flag	output	1	Stall signal

Description The `Hazard_Detection_Unit` module detects data hazards that require stalling the pipeline. It generates a stall signal when a hazard is detected, preventing incorrect data from being used.

2.3.22 Pipeline Registers

Description Pipeline registers are used to transmit the data between different stages. The ports of these pipeline registers are omitted here for simplicity. There are four pipeline registers in total. They are IF_ID, ID_EX, EX_MEM, MEM_WB.

3 Onboard instructions

3.1 Reset Operation

Reset Button: reset button is S6 on the board. The reset button on the development board is used to perform a system reset. When the reset button is pressed, it sends a signal to the system to clear all registers and set the program counter (PC) to 0.

- **Note**: when the board is switched into UART communication mode, then the board is automatically reset, until the communication mode ends.

3.2 Input Operations

The development board supports various input operations through switches and buttons:

- **Switches**: The switches on the board can be used to provide various input signals to the system. These switches are connected to the `switch` input of the main module.
- **Buttons**: Only two buttons are effectively used for our CPU: S6 to reset the board, P2 to enter UART communication mode.

3.3 Output Operations

The system outputs can be observed through the following regions on the development board:

- **LEDs**: The LEDs on the board display the output signals from the system. Each LED corresponds to a specific bit in the output signal, allowing for easy monitoring of the system's status.

- **Seven-Segment Display**: The seven-segment displays are used to show numerical data and are controlled by the display driver module. The display driver handles various display flags and data inputs to control the tube selection and shape output.

3.4 UART Communication

Press button P2 to enter communication mode. The UART module handles the communication between the development board and an external device, allowing for data exchange and programming operations.

- **Note**: when the board is switched into UART communication mode, then the board is automatically reset, until the communication mode ends.

4 Self-Testing Instructions

The tests include simulation and on-board testing, covering both unit and integration tests. The table below summarizes the test methods, test types, test case descriptions, and test results.

Test Method	**Test Type**	**Test Case Description**	**Test Result**	**Conclusion**
Simulation	Unit Test	ALU operation correctness	Pass	The ALU performs arithmetic and logical operations correctly.
Simulation	Unit Test	Register file read/write	Pass	Registers correctly store and provide data as expected.
Simulation	Unit Test	Instruction fetch and decode	Pass	Instructions are fetched and decoded accurately.
On-board Testing	Integration Test	Complete CPU functionality	Pass	The CPU executes a set of predefined instructions correctly on the hardware.
On-board Testing	Integration Test	UART communication	Pass	UART module successfully transmits and receives data.
Simulation	Integration Test	Memory read/write operations	Pass	Data memory performs read and write operations correctly.
Simulation	Integration Test	Pipeline register transfer	Pass	Data and control signals are correctly transferred between pipeline stages.
On-board Testing	Unit Test	Button debounce functionality	Pass	Buttons provide stable signals without glitches.
Simulation	Integration Test	Branch prediction accuracy	Pass	The branch prediction unit correctly predicts and handles branches.

5 Bonus

5.1 Forwarding Unit

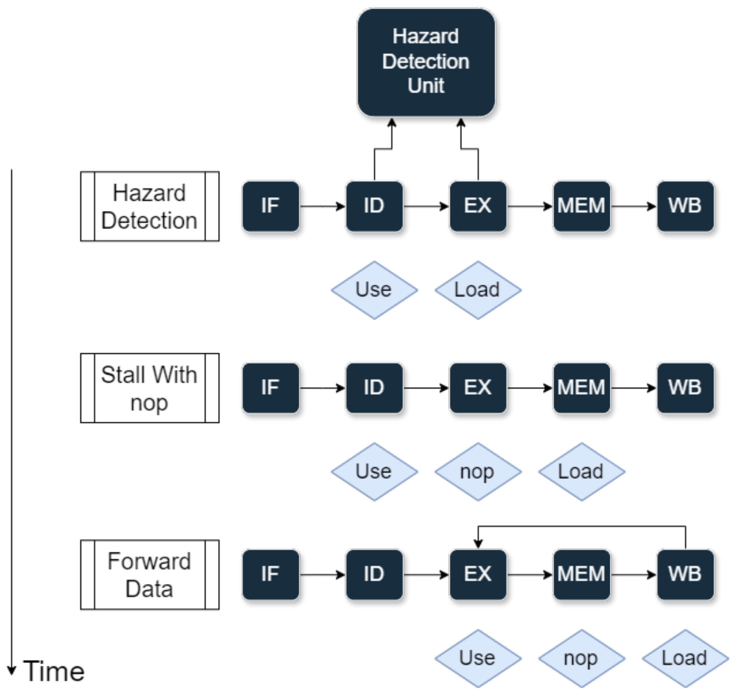
The forwarding Unit module follows the very same idea from textbook "Computer Organization and Design". This module forwards data to ALU if a hazard happens. However, it cannot resolve load-use hazard, which is handled by Hazard Detection Unit.
The forwarding unit receive the register index to be written back in MEM and WB stage. If the index matches any of source register 1 or source register 2, it will forward the corresponding data to ALU. There are also two special case to handle. First, when the destination register is written in two consecutive instructions, the data in MEM stage is forwarded (since it is more recently updated). Second, if destination register is `x0`, we should not forward any data to ensure ALU get all 0 from register `x0`.

```
module Forwarding_Unit(
    input [31:0] ALU_result_MEM,
    input [4:0] write_reg_idx_MEM,
    input write_reg_flag_MEM,
    input mem_to_reg_flag_MEM,
    input [31:0] ALU_result_WB,
    input [31:0] read_data_WB,
    input [4:0] write_reg_idx_WB,
    input write_reg_flag_WB,
    input mem_to_reg_flag_WB,
    input [4:0] read_reg_idx_1_EX,
    input [4:0] read_reg_idx_2_EX,

    output reg [31:0] read_data_1_forwarding,
    output reg [31:0] read_data_2_forwarding,
    output read_data_1_forwarding_flag,
    output read_data_2_forwarding_flag
);

wire MEM_hazard_1_flag;
wire MEM_hazard_2_flag;
wire WB_hazard_1_flag;
wire WB_hazard_2_flag;

assign MEM_hazard_1_flag = (write_reg_flag_MEM && (write_reg_idx_MEM ==
↪   read_reg_idx_1_EX) && (read_reg_idx_1_EX != 0) &&
↪   !mem_to_reg_flag_MEM);
assign MEM_hazard_2_flag = (write_reg_flag_MEM && (write_reg_idx_MEM ==
↪   read_reg_idx_2_EX) && (read_reg_idx_2_EX != 0) &&
↪   !mem_to_reg_flag_MEM);
assign WB_hazard_1_flag = (write_reg_flag_WB && (write_reg_idx_WB ==
↪   read_reg_idx_1_EX) && (read_reg_idx_1_EX != 0));
assign WB_hazard_2_flag = (write_reg_flag_WB && (write_reg_idx_WB ==
↪   read_reg_idx_2_EX) && (read_reg_idx_2_EX != 0));

```

```
assign read_data_1_forwarding_flag = MEM_hazard_1_flag ||
↪  WB_hazard_1_flag;
assign read_data_2_forwarding_flag = MEM_hazard_2_flag ||
↪  WB_hazard_2_flag;

always @* begin
    if (read_data_1_forwarding_flag)
        if (MEM_hazard_1_flag)
            read_data_1_forwarding = ALU_result_MEM;
        else
            if (mem_to_reg_flag_WB)
                read_data_1_forwarding = read_data_WB;
            else
                read_data_1_forwarding = ALU_result_WB;
    else
        read_data_1_forwarding = 32'b0;

    if (read_data_2_forwarding_flag)
        if (MEM_hazard_2_flag)
            read_data_2_forwarding = ALU_result_MEM;
        else
            if (mem_to_reg_flag_WB)
                read_data_2_forwarding = read_data_WB;
            else
                read_data_2_forwarding = ALU_result_WB;
    else
        read_data_2_forwarding = 32'b0;
end

endmodule
```

5.2 Hazard Detection Unit

This module detects load-use hazard and send corresponding signals to other modules. It detect load-use hazard early in ID and EX stage. First, it checks if the ID instruction (instruction in ID stage) tries to read from the same register being written in EX instruction. If so, it checks if the EX instruction is a load instruction. If this is also true, then we find a load-use hazard.
To resolve this, a stall command will be send to PC and IF/ID pipeline register, making them maintain their current value for one clock cycle (this will keep the instruction in IF and ID stage unchanged). At the same time, the ID/EX pipeline register will also receive a command that makes it output `nop`. This effectively let the instructions in the first two stage stay still, and the instructions in the other stages keep moving, and insert an `nop` into the gap.
Then, in the next cycle, the data is fetched from memory and moves into WB stage. Then the forwarding unit can handle the hazard by forwarding WB data to ALU.



```verilog
module Hazard_Detection_Unit(
    input [4:0] write_reg_idx_EX,
    input write_reg_flag_EX,
    input mem_to_reg_flag_EX,
    input [4:0] read_reg_idx_1_ID,
    input [4:0] read_reg_idx_2_ID,

    output stall_flag
);

assign stall_flag = (write_reg_flag_EX && mem_to_reg_flag_EX &&
↪   write_reg_idx_EX != 0 && (write_reg_idx_EX == read_reg_idx_1_ID ||
↪   write_reg_idx_EX == read_reg_idx_2_ID));
endmodule

```

5.3 Branch Prediction

For branch prediction, we implemented a module `Program_Counter_Prediction`. The module's core component is a 4 element 64 bit array, that serves as a LRU cache. For each entry of the cache, the first 32 bits store the program counter value of a branch instruction, and the last 32 bits store the program counter value of target instruction (where branch instruction goes to). Every clock cycle, the module checks if current program counter value matches any entry. If so, it will directly set the next value of program counter to the last 32 bits of that entry. Otherwise, it simply add 4 to current value.
The update of the cache is done when branching takes place. The `Branch_Target` module will send a 64 bit entry to `Program_Counter_Prediction` and a LRU replacement will be performed.

```verilog
module Program_Counter_Prediction(
    input                 clk,
    input                 rst,
    input      [31:0] branch_from_pc,                // where the branch instruction is
    input      [31:0] branch_to_pc,                  // where the branch instruction will go
    input      [31:0] program_counter,
    input                 branch_flag,

    output reg [31:0] program_counter_prediction,
    output reg [31:0] prev_pcp
);

// Each LRU_cache is divided into 2 parts
// The first half stores the branch_from_pc (where the branch instruction is)
// The second half stores the branch_to_pc (where the branch instruction will go)
// If the branch_from_pc is in the LRU_cache, the most recent branch_to_pc is the prediction
// If the branch_from_pc is not in the LRU_cache, we predict branch will not be taken

reg [63:0] LRU_cache [0:3];
reg current_pc_in_cache_flag;
reg [1:0] current_pc_in_cache_idx;
reg branch_from_pc_in_cache_flag;
reg [1:0] branch_from_pc_in_cache_idx;
reg [2:0] LRU_capacity;

always @* begin
    if (LRU_cache[0][31:0] == program_counter) begin
        current_pc_in_cache_flag = 1'b1;
        current_pc_in_cache_idx = 2'b00;
    end
    else if (LRU_cache[1][31:0] == program_counter) begin
        current_pc_in_cache_flag = 1'b1;
        current_pc_in_cache_idx = 2'b01;
    end
    else if (LRU_cache[2][31:0] == program_counter) begin
        current_pc_in_cache_flag = 1'b1;
        current_pc_in_cache_idx = 2'b10;
    end
    else if (LRU_cache[3][31:0] == program_counter) begin
        current_pc_in_cache_flag = 1'b1;
        current_pc_in_cache_idx = 2'b11;
    end
    else begin
        current_pc_in_cache_flag = 1'b0;
        current_pc_in_cache_idx = 2'b00;
    end

    if (LRU_cache[0][31:0] == branch_from_pc) begin
        branch_from_pc_in_cache_flag = 1'b1;
        branch_from_pc_in_cache_idx = 2'b00;
    end
    else if (LRU_cache[1][31:0] == branch_from_pc) begin
        branch_from_pc_in_cache_flag = 1'b1;
        branch_from_pc_in_cache_idx = 2'b01;
    end
    else if (LRU_cache[2][31:0] == branch_from_pc) begin
        branch_from_pc_in_cache_flag = 1'b1;
        branch_from_pc_in_cache_idx = 2'b10;
    end
    else if (LRU_cache[3][31:0] == branch_from_pc) begin
        branch_from_pc_in_cache_flag = 1'b1;
        branch_from_pc_in_cache_idx = 2'b11;
    end
    else begin
        branch_from_pc_in_cache_flag = 1'b0;
        branch_from_pc_in_cache_idx = 2'b00;
    end

    if (current_pc_in_cache_flag && current_pc_in_cache_idx < LRU_capacity) begin
        program_counter_prediction = LRU_cache[current_pc_in_cache_idx][63:32];
    end
    else begin
        program_counter_prediction = program_counter + 4;
    end
end

always @(negedge clk) begin
    if (rst) begin
        prev_pcp <= 32'h00000000;
    end
    else begin
        prev_pcp <= program_counter_prediction;
    end

    if (rst) begin
        LRU_cache[0] <= 64'h0000000000000000;
        LRU_cache[1] <= 64'h0000000000000000;
        LRU_cache[2] <= 64'h0000000000000000;
```

```verilog
            LRU_cache[3] <= 64'h0000000000000000;
            LRU_capacity <= 3'b000;
        end
        else if (branch_flag) begin
            if (branch_from_pc_in_cache_flag) begin
                case (branch_from_pc_in_cache_idx)
                    2'b00: begin
                        LRU_cache[0] <= {branch_to_pc, branch_from_pc};
                        LRU_cache[1] <= LRU_cache[1];
                        LRU_cache[2] <= LRU_cache[2];
                        LRU_cache[3] <= LRU_cache[3];
                    end
                    2'b01: begin
                        LRU_cache[0] <= {branch_to_pc, branch_from_pc};
                        LRU_cache[1] <= LRU_cache[0];
                        LRU_cache[2] <= LRU_cache[2];
                        LRU_cache[3] <= LRU_cache[3];
                    end
                    2'b10: begin
                        LRU_cache[0] <= {branch_to_pc, branch_from_pc};
                        LRU_cache[1] <= LRU_cache[0];
                        LRU_cache[2] <= LRU_cache[1];
                        LRU_cache[3] <= LRU_cache[3];
                    end
                    2'b11: begin
                        LRU_cache[0] <= {branch_to_pc, branch_from_pc};
                        LRU_cache[1] <= LRU_cache[0];
                        LRU_cache[2] <= LRU_cache[1];
                        LRU_cache[3] <= LRU_cache[2];
                    end
                endcase
                LRU_capacity <= LRU_capacity;
            end
            else begin
                LRU_cache[0] <= {branch_to_pc, branch_from_pc};
                LRU_cache[1] <= LRU_cache[0];
                LRU_cache[2] <= LRU_cache[1];
                LRU_cache[3] <= LRU_cache[2];
                if (LRU_capacity == 3'b100) begin
                    LRU_capacity <= 3'b100;
                end
                else begin
                    LRU_capacity <= LRU_capacity + 1;
                end
            end
        end
    end

```

```
endmodule
```

5.4 lui and auipc

The `lui` and `auipc` would require addition information, which is the value of program counter. We detect these two instructions in `ALU` module. If any is found, the ALU_result will be the corresponding value of U-type command, instead of normal arithmatic operation. This also works for `jal` and `jalr` instructions.
Below is part of code from `ALU` module.

```
always @* begin
    if (jal_flag || jalr_flag) begin
        ALU_result = program_counter + 4;
    end
    else if (lui_flag) begin
        ALU_result = imme;
    end
    else if (auipc_flag) begin
        ALU_result = program_counter + imme;
    end
    else
        case(ALU_control)
            4'b0000:
            begin
                ALU_result = operand_1 + operand_2;
            end
            4'b0001:
            begin
                ALU_result = operand_1 - operand_2;
            end
            4'b0010:
            begin
                ALU_result = operand_1 ^ operand_2;
            end
            4'b0011:
            begin
                ALU_result = operand_1 | operand_2;
            end
            4'b0100:
            begin
                ALU_result = operand_1 & operand_2;
            end
            4'b0101:
            begin
                ALU_result = operand_1 << operand_2[4:0];
            end
            4'b0110:
            begin
                ALU_result = operand_1 >> operand_2[4:0];
```

```verilog
            end
            4'b0111:
            begin
                ALU_result = operand_1 >>> operand_2[4:0];
            end
            4'b1000:
            begin
                ALU_result = ($signed(operand_1) < $signed(operand_2)) ? 32'b1 : 32'b0;
            end
            4'b1001:
            begin
                ALU_result = (operand_1 < operand_2) ? 32'b1 : 32'b0;
            end
            default:
            begin
                ALU_result = 32'b0;
            end
        endcase
end
```

6 Issues and Summary

6.1 Issues Encountered

During the development process, several issues were encountered that required careful analysis and resolution:

- **UART Communication:** When implementing UART communication mode, we found that unit test works properly, but the integrated test failed. It is because the MSB of UART address is used to distinguish data for instruction memory and for data memory, not an actual data address.
- **Memory Management:** There should be an address shifting when implementing memory IP core. Because memory IP core always start at address `0`, but in the memory layout of RARS, the address of memory usually isn't (in our case data memory starts at `0x2000`). This is because RARS is designed for Von Neumann architecture, where the data memory and instruction memory is stored in the same hardware (address before `0x2000` is reserved for instruction memory). So we implemented shifting when accessing memories (data memory address for IP core = raw data memory address generated by RARS - `0x2000`) to better utilized memory. When dumping the memories contents, we should properly configure the memory layout to obtain the correct result.
- **Latch:** When writing case statement in Verilog, we should consider all the possible cases, since unidentified cases may lead to latches.

6.2 Thoughts

Throughout the development process, several key considerations were highlighted:

- **Documentation:** We pay careful attention to the documentations and comments in the project. Each module is documented in detail, and some necessary notes to understand the code logic are also added. Thanks to this convention, each team member's code can be easily understood by others, which greatly boost the efficiency of our project.
- **Team Collaboration:** We use version control tool Git to facilitate out CPU development. Some conventions are followed, for example, we have several branches like dev, test, debug etc.
- **Incremental Development:** Our Pipeline CPU scheme is not devised from scratch. We adopted an incremental development approach during the project. We implemented a single-cycle CPU at first. We tackled some issues with it, ensuring that these modules were robust when later being reused in the pipeline schematic. More importantly, we realized the biggest drawback of single-cycle CPU - we don't have enough clock edges to trigger different events, but these events are supposed to be done in a defined sequence with more than 2 steps (hence only two edges in a single cycle is hardly enough). But this problem is perfectly handled by pipeline CPU.
- **Code Convention:** It's important to have a code convention, especially for pipeline development. This is because each stage needs to maintain its own flag and variables. For example each stage manages its current instruction, then there needs to be different naming for instruction variables in different stages.

6.3 Summary

In summary, the development process was marked by a series of challenges that were successfully overcome through careful planning, rigorous testing, and effective teamwork. We really enjoy this project, and appreciate our teacher.