REPUBLIC OF TURKEY
YILDIZ TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

RISC-V PROCESSOR DESIGN WITH ADVANCED HARDWARE FEATURES

20011068 — Ahmet Akib GÜLTEKİN
20011906 — Basel KELZİYE

SENIOR PROJECT

Advisor
Dr. Erkan USLU

January, 2024

ACKNOWLEDGEMENTS

We would like to acknowledge and give our warmest thanks to our supervisor DR. Erkan USLU who made this work possible. His guidance and advice carried us through all the stages of writing our project, and we would also thank Umut KAYA for his previous work on a Risc-V single core.

Ahmet Akib GÜLTEKİN
Basel KELZİYE

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
IF	Instruction Fetch
ID	Instruction Decode
EX	Execute
MEM	Memory Operation
WB	Write Back
CPI	Cycle Per Instruction
PWM	Pulse Width Modulation
UART	Universal Asynchronous Receiver Transmitter
SPI	Serial Peripheral Interface
ISA	Instruction Set Architecture

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

RISC-V PROCESSOR DESIGN WITH ADVANCED HARDWARE FEATURES

Ahmet Akib GÜLTEKİN
Basel KELZİYE

Department of Computer Engineering
Senior Project

Advisor: Dr. Erkan USLU

This study aims to delve into the design of a chip pipeline and cache system based on the RISC-V architecture, a leading open-source instruction set architecture (ISA). Central to the research is the conceptual development of a pipeline that is harmoniously aligned with the unique features and requirements of the RISC-V ISA. This involves meticulous examination of the stages of the pipeline and ensuring that the data flow is processed smoothly and effectively. Simultaneously, the work also pays significant attention to the creation of a cache design. This direction aims to establish a cache architecture that aims to strike an effective balance between the demands of fast data access and the power and area limitations of the RISC-V framework. The combination of these two design efforts aims to showcase the potential of the RISC-V architecture to address complex computational tasks. With this dual-focus approach, the research provides valuable insights into the field of computer architecture and design and a detailed roadmap for effectively and scalably building RISC-V-based systems.

Keywords: RISC-V architecture, chip pipeline design, cache system design, instruction set architecture, computer architecture, data flow, computational tasks.

ÖZET

GELİŞMİŞ DONANIM ÖZELLİKLERİNE SAHİP RISC-V İŞLEMCİ TASARIMI

Ahmet Akib GÜLTEKİN
Basel KELZİYE

Bilgisayar Mühendisliği Bölümü
Bitirme Projesi

Danışman: Dr. Erkan USLU

Bu kapsamlı çalışma, önde gelen açık kaynaklı bir komut seti mimarisi (ISA) olan RISC-V mimarisine dayalı bir çip hattı ve önbellek sisteminin tasarımını derinlemesine incelemeye yöneliktir. Araştırmanın merkezinde, RISC-V ISA'nın benzersiz özellikleri ve gereksinimleriyle uyumlu bir şekilde hizalanmış bir hattın kavramsal olarak geliştirilmesi bulunmaktadır. Bu, hattın aşamalarının titiz bir şekilde incelenmesini ve veri akışının sorunsuz ve etkili bir şekilde işlenmesini sağlamayı içerir. Eş zamanlı olarak, çalışma ayrıca bir önbellek tasarımının oluşturulmasına önemli ölçüde dikkat göstermektedir. Bu yön, hızlı veri erişimi talepleri ile RISC-V çerçevesinin güç ve alan sınırlamaları arasında etkin bir denge kurmayı hedefleyen bir önbellek mimarisi kurmayı amaçlamaktadır. Bu iki tasarım çabasının birleşimi, RISC-V mimarisinin karmaşık hesaplama görevlerini ele alma potansiyelini sergilemeyi amaçlamaktadır. Bu çift odaklı yaklaşımla, araştırma bilgisayar mimarisi ve tasarımı alanına değerli içgörüler ve RISC-V tabanlı sistemlerin etkili ve ölçeklenebilir bir şekilde inşa edilmesi için detaylı bir yol haritası sunmaktadır.

Anahtar Kelimeler: RISC-V mimarisi, çip hattı tasarımı, önbellek sistemi tasarımı, komut seti mimarisi, bilgisayar mimarisi, veri akışı, hesaplama görevleri.

1 Introduction

1.1 Instruction Set Architecture

an Instruction Set Architecture is the set of an instructions that A CPU is able to execute. The instruction set doesn't specify how the hardware should be designed to execute the instructions, rather it only responsible specifying the rules of an instruction.

Today the Instruction Set Architectures are divided by 2 groups.

- CISC (X86 Intel ISA)
- RISC (ARM, **RISC-V** ISA's)

1.1.1 CISC

CISC is short for **Complex Instruction Set Architecture**. The main idea behind this architecture is that each instruction may take different clock cycles, one drawback is that it get really challenging to decode the instructions [1].

1.1.2 RISC

RISC is short for **Reduced Instruction Set Architecture** which is based on the idea of each instruction should take the same amount of clock cycles to be done. This vision helps us predicting our energy consumption and our program's behaviour compared to the CISC architecture [2].

1.2 RISC-V Architecture

The problem with ISA's such as x86 and ARM that they are licensed and each time you have to use them in a product you must pay for the license. Apple is said to pay Arm 0.30$ for each device sold, which approximately makes ARM's %5 Revenue [3].

The RISC-V architecture is an open-source instruction set architecture developed in UC Berkeley [4], RISC-V is under the RISC Family and has a similar ISA as MIPS ISA. RISC-V ISA is developed to be Pipelined-friendly and easy to decode the instruction which makes the hardware faster by nature.

RISC-V is notable for its high scalability and low power consumption. This makes it ideal for use in various areas, including mobile devices, embedded systems, and high-performance computing applications [5]. The development and implementation of the RISC-V architecture allow for innovative steps in the field of computer architecture, opening the way for next-generation solutions that can meet the continuously changing needs of this field.

1.3 Project Objective

The aim of this project is to enhance A RV32IM compatible core with a 5 staged Pipeline with forwarding unit to increase throughput and Cache memories (instruction and memory) for faster execution time.

2
Preliminary Examination

In this section, we examine the current specifications of our processor and outline the anticipated enhancements.

2.1 Current Processor Specifications

Our current processor is a single-cycle design, supporting the RV32IM instruction set. It features interfaces for UART, SPI, and PWM. The processor includes a 1-level instruction cache of 8 KiB and a data memory of 64 KiB.

2.2 Final Processor Specifications

The goal is to transition to a 5-stage pipeline architecture, aiming to significantly increase the clock frequency. This change will mark the shift from a single-cycle to a multi-cycle operation. We plan to integrate a data cache mechanism and implement cryptographic instructions[6] as specified by the "Teknofest Chip Design Competition"[7].

3
Feasability

3.1 Technical Feasability

3.1.1 Software Feasability

Verilog HDL is used as a programming language, and Xilinx Vivado as development environment.

3.1.2 Hardware Feasability

Since Xilinx Vivado is only available on windows OS, a computer with windows OS is required.

3.2 Labor and Time Feasibility

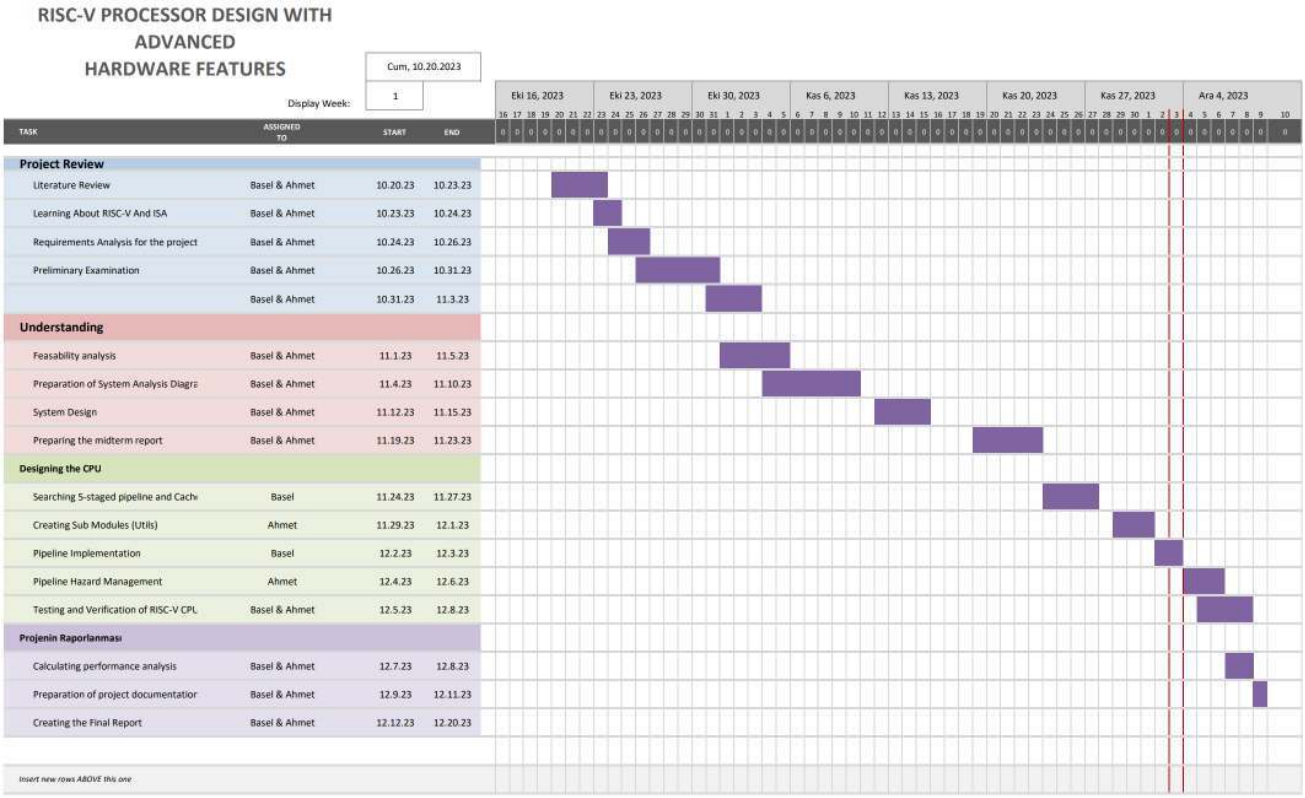


Figure 3.1 Gantt Diyagramı

3.3 Legal Feasability

RISC-V is open-sourced instruction set[8] , Xilinx Vivado Development Environmnet is free for Educational purposes we have no legal obligations.

3.4 Economic Feasability

Xilinx Vivado is freely available and Windows OS is freely provided by the university hence we have no economical expenses in the project.

4
System Analysis

In this section, the requirements, objectives and performance criteria of the design are examined.

4.1 Requirements

- *Architecture Compliance:* The design must be fully compatible with the RISC-V RV32IM instruction set architecture.
- *Processing Capabilities:* Support for integer and multiplication instructions as specified in RV32IM.
- A 5 stage pipeline that resolves all the hazards may occur within the pipeline.

4.2 Objectives

- *Optimal Performance:* Achieve efficient execution of RV32IM instructions with minimal latency.
- *Scalability:* Ensure the design is scalable for future enhancements and compatible with different RISC-V extensions.

4.3 Performance Criteria

- *Instruction Throughput:* Measure the number of instructions processed per cycle.
- *Resource Efficiency:* Evaluate memory and power consumption against processing output.
- *CPI:* Clock Cycle per instruction

5
System Design

5.1 What and why

In a traditional Single cycle design, The clock frequency should be as long as the longest instruction to execute referred as "critical path" (+some time for propagation), usually this instruction is "LOAD" instruction. because we get the instruction from instruction Cache (IF Stage), we decode the instruction (ID), we calculate the address to load value from (EX), we access the memory and read value from that address (MEM) and finally we write back the value to the desired register (WB). The data path of a Load instruction is shown in Figure 5.1

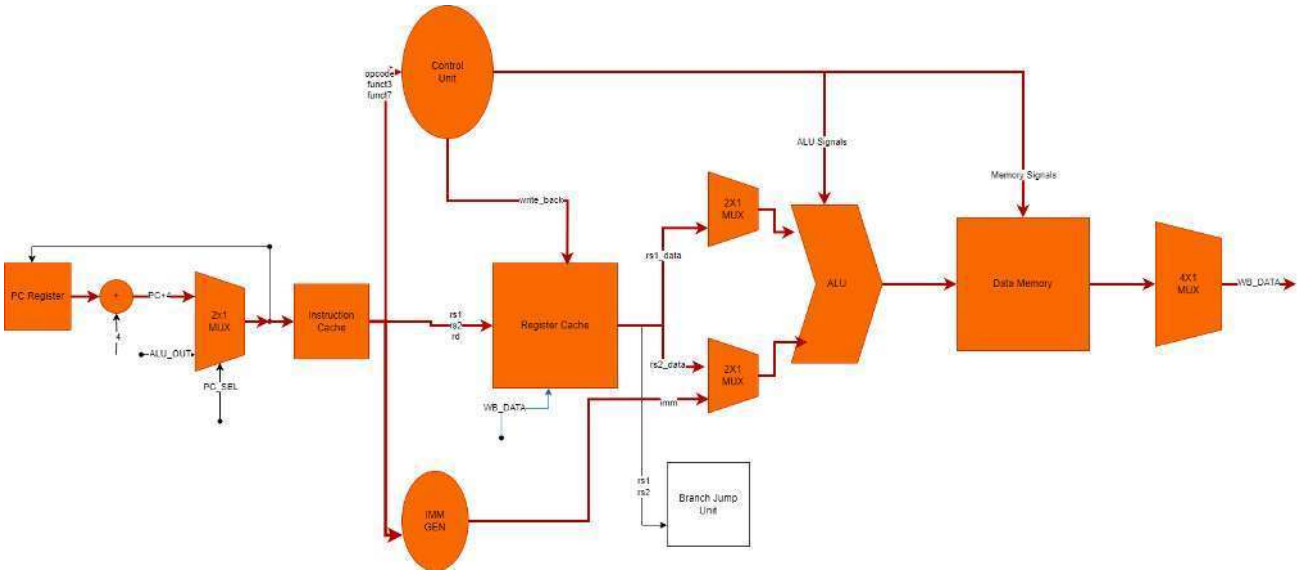


Figure 5.1 Load Instruction Data Path

So this instruction passes by and uses the 5 stages of our Processor so we should assure that our next clock will not occur until our critical path is safely executed. The problem with this design is that a basic instruction such as "NOP" instruction which basically just increment the Program counter will take as much as a "LOAD" instruction.

5.2 Pipeline

Pipelining is a mechanism that allows us to execute several instructions (in our design 5 at most) at the same time [9]. This is possible by dividing the 5 stages shown in Figure 5.2 it allows us to increase the clock frequency since our critical path is just the path between 2 stages. Thus pipeline enhance our throughput while increasing our clock frequency

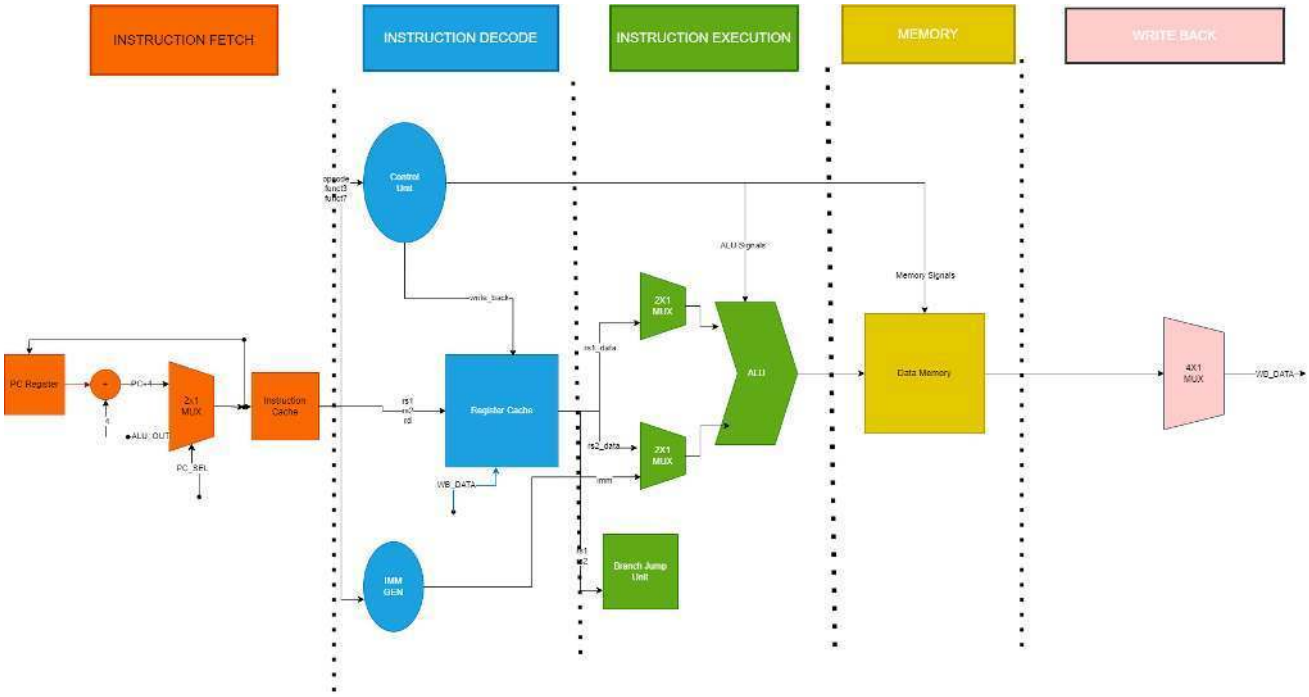


Figure 5.2 Dividing Our Data Path To 5 Stages

Our pipeline will have 5 stages. Those stages are (IF, ID, EX, MEM and WB) To implement this architecture we use 4 flip flops which helps us to forward the data from the previous stage, as shown in Figure 5.3.

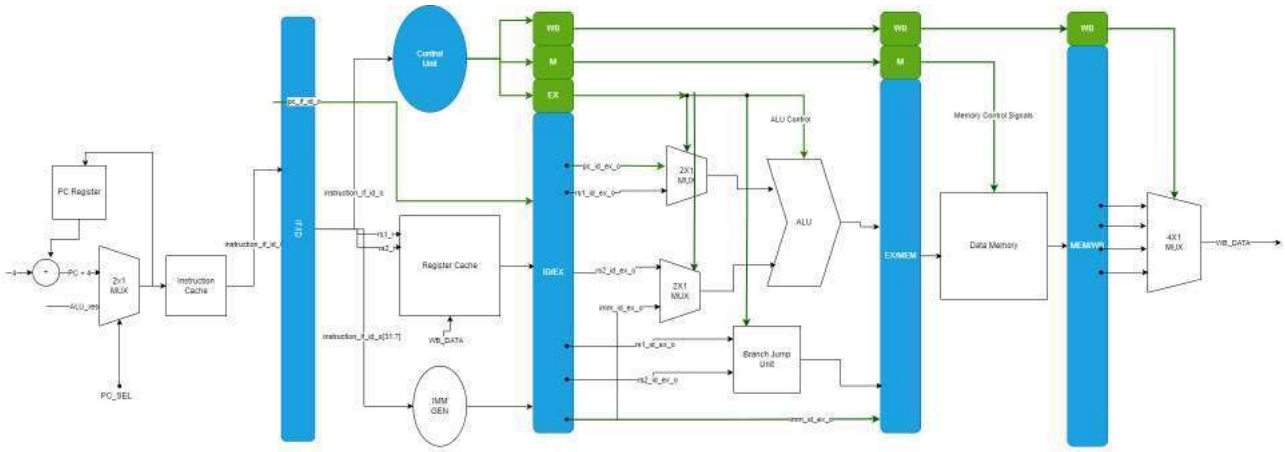


Figure 5.3 Pipeline Design

Now using this structure each stage can work on a different instruction, this is possible because our flip flops will be holding the required information related to that instruction for a valid execution.

Now let's try to understand how our pipeline will look like when executing the assembly code given in Listing 5.1

Listing 5.1 Sample RISC-V Assembly Code

```
1 add x14, x5, x6
2 ld x13, 48(x1)
3 add x12, x3, x4
4 sub x11, x2, x3
5 ld x10, 40(x1)
```

Now let's assume our CPU start fetching instructions at Clock cycle 1, Shown at Figure 5.4.

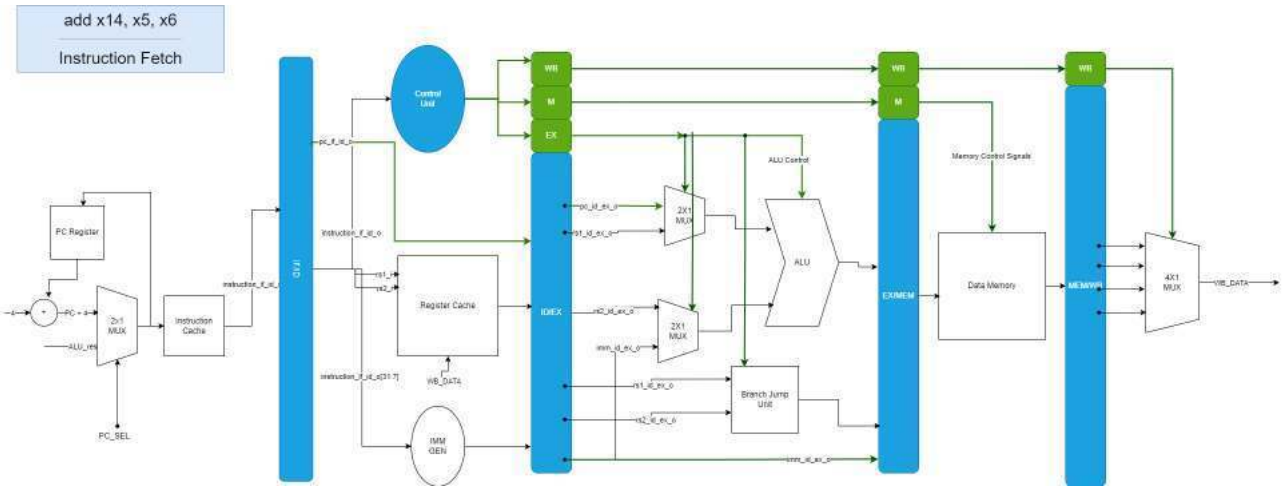


Figure 5.4 Pipeline Status At CC 1

and when our next clock edge occurs, our add instruction will be moved to the next stage and the next instruction which is "ld x13, 48(x1)" will arrive to the instruction fetch stage shown in Figure.

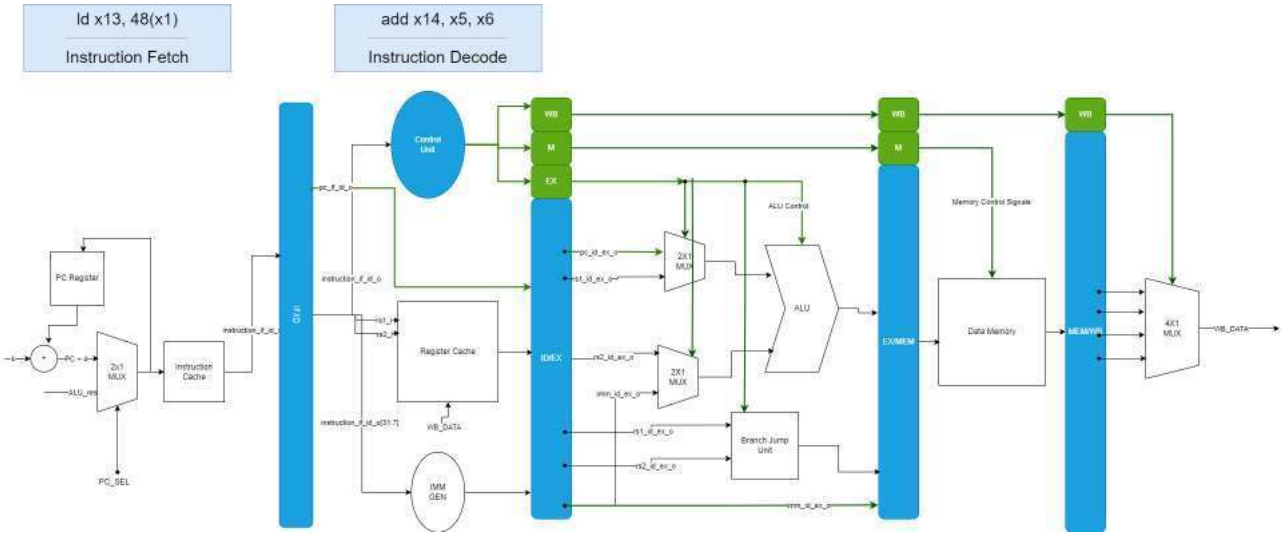


Figure 5.5 Pipeline Status At CC 2

and on each cycle a new instruction will be entering to the pipeline (sometimes we have to stall the pipeline, mentioned later in pipeline hazards section.) and if we continue like this we can see in Figure 5.6 that our pipeline reaches it max capacity with 5 instruction executing simultaneously.

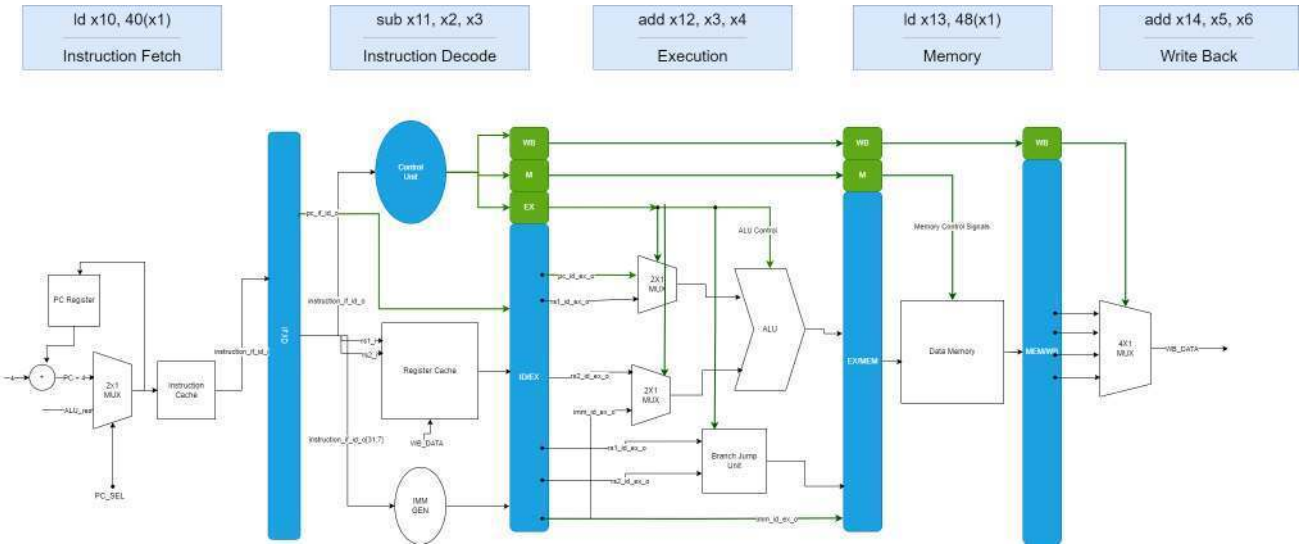


Figure 5.6 Pipeline Status At CC 5

Although pipelining is a way to increase performance of a CPU and throughput, it may not be that much easy at the end of the day. This is because of the nature of the pipeline, instructions can depend on each other and if these situations are not handled properly our CPU's registers will have inconsistent data and will lead to catastrophic results.

5.2.1 Pipelining in RISC-V

RISC-V ISA is designed with features that facilitate efficient pipelining [10]. Below is a list of its advantages:

- All RISC-V instructions are of uniform length. This consistency simplifies the instruction fetch process in the first pipeline stage and aids in their decoding in the second stage. In contrast, instruction sets like x86, where instruction lengths vary significantly (from 1 to 15 bytes), present greater challenges for pipelining.
- RISC-V employs a limited number of instruction formats. Additionally, the source and destination register fields are consistently located in the same positions across instructions. This uniformity streamlines the decode process, as the pipeline can handle these instructions more predictably and efficiently.
- In RISC-V, memory operands are exclusively found in load and store instructions. This design allows the execute stage of the pipeline to focus on calculating memory addresses, with the actual memory access occurring in a subsequent stage. This separation of concerns simplifies the design and operation of the pipeline stages.

5.3 Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called **hazards**, and there are three different types.

1. Structural Hazard
2. Data Hazard
3. Control Hazard

5.3.1 Structural Hazard

When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute [11]. Consider our design Having only one shared memory for instructions and data, this will lead to conflict in the IF and MEMORY stages in Figure 5.7.

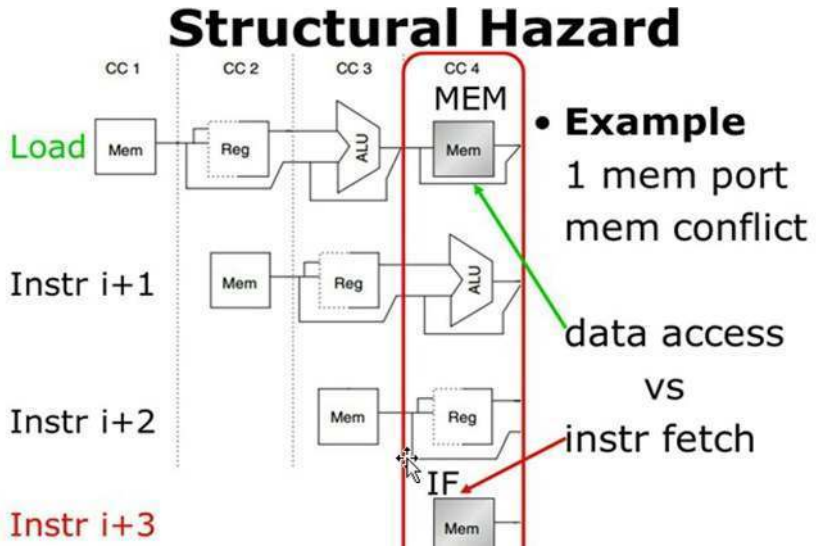


Figure 5.7 Structural Hazard In Shared Memory

How ever because we use a Seperated Memory for instruction and data (Harvard Architecture) our design is not prone to structural Hazards

5.3.2 Data Hazards

data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline [12]. For example, suppose we have an add instruction followed immediately by a subtract instruction that uses that sum

Listing 5.2 Data Dependency Code Example

```
1 add x1, x2, x3
2 sub x4, x1, x5
```

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.

solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called forwarding or bypassing.

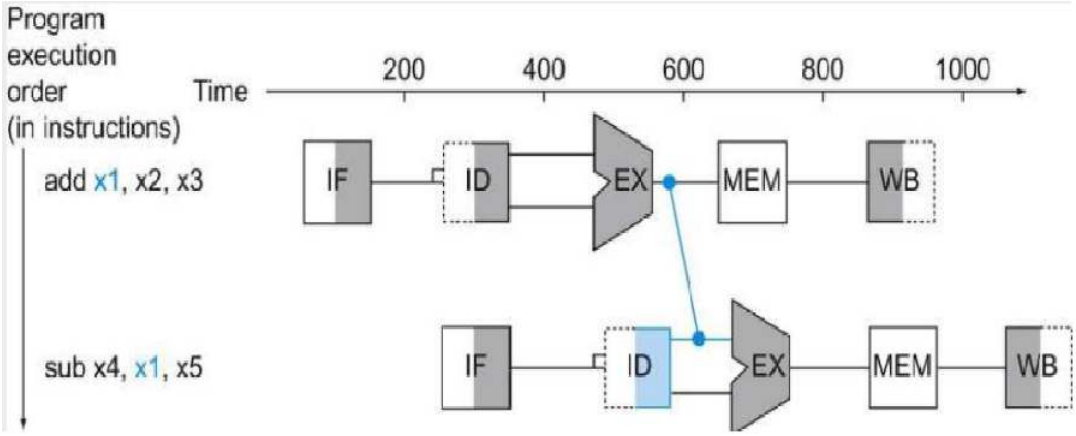


Figure 5.8 Data forwarding Graphical Representation

5.3.2.1 Load Data Hazards

Load Data hazards are a specific form of data hazards in which the data being loaded by a load instruction have not yet become available when they are needed by another instruction. In this situation data forwarding can't solve it and we must stall the pipeline and add a "NOP" instruction in between, note that if forwarding weren't available we would have to stall the pipeline for 2 Clock Cycles.

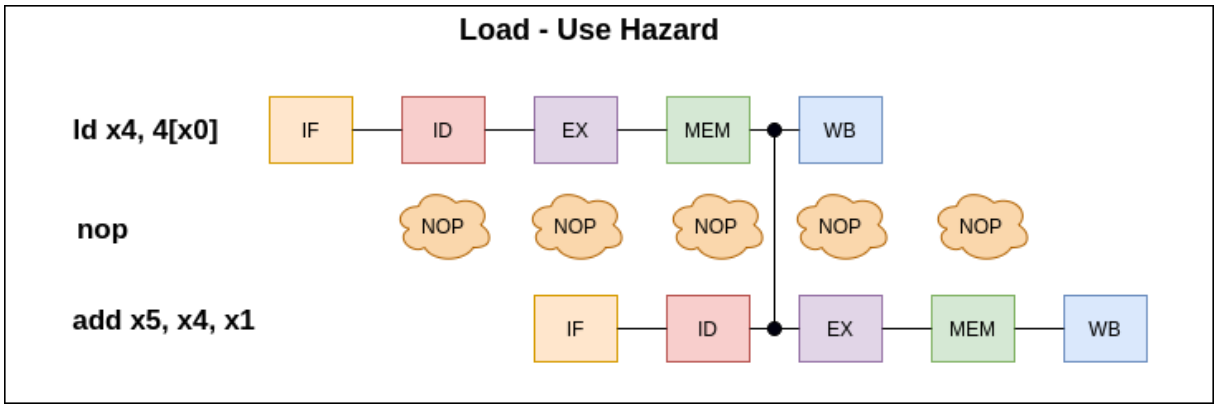


Figure 5.9 Example Of Load Data Hazard

5.3.3 Control Hazards

control hazard arises from the need to make a decision based on the results of one instruction while others are executing. Computers do indeed use prediction to handle conditional branches. One simple approach is to predict always that conditional branches will be untaken [13]. When you're right, the pipeline proceeds at full speed. Only when conditional branches are taken does the pipeline stall as in Figure

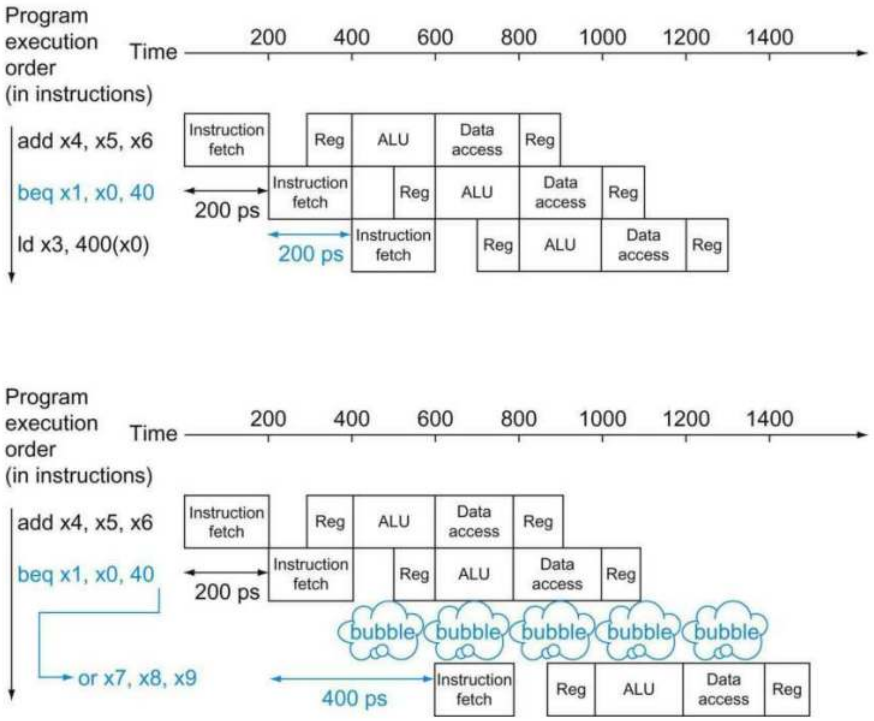


Figure 5.10 Predicting that branches are not taken as a solution to control hazard

5.4 Cache Memory Design

Implementing an efficient cache design is crucial for optimizing the performance of our RV32IM CPU [14]. In this section, we explore various cache design possibilities, considering factors such as size, associativity, replacement policies, and write policies [15].

5.4.1 Cache Size and Levels

- **L1 Cache:** A smaller, faster cache that is closer to the CPU core. We consider implementing separate instruction and data caches (I-cache and D-cache) to improve access efficiency.
- **L2 Cache:** A larger, but slower cache compared to L1. It acts as an intermediate store before accessing the main memory. The size and existence of an L2 cache will be determined based on performance simulations.

5.4.2 Cache Associativity

- **Direct-Mapped Cache:** The simplest form, where each block in main memory maps to exactly one block in the cache.
- **Set-Associative Cache:** A compromise between direct-mapped and fully associative caches, where each memory block can be cached in any of the N blocks in a specific set.
- **Fully Associative Cache:** Any block of memory can be placed in any block of the cache, offering the most flexibility but at a higher cost in terms of complexity and power consumption.

5.4.3 Replacement Policies

- **Least Recently Used (LRU):** Replaces the cache block that has not been used for the longest period.
- **First-In, First-Out (FIFO):** Replaces the oldest block in the cache.
- **Random Replacement:** Randomly selects a block to replace, offering a simple yet effective approach in certain scenarios.

5.4.4 Write Policies

- **Write-Through:** Updates are written to both the cache and the main memory simultaneously.
- **Write-Back:** Updates are initially made in the cache. The modified cache block is written back to main memory only when it is replaced.
- **Write Allocation:** Determines whether a block of main memory is loaded into the cache on a write miss.

This design exploration aims to balance performance, complexity, and power consumption for our RV32IM CPU.

Our Cache Design Structure

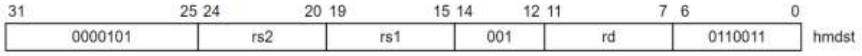
Our design is focused on an L1 cache and we chose it due to its simplicity and practicality. For cache associativity, we opted for Direct-Mapped Cache as it is straightforward, suitable for our design concept, and easy to implement. Additionally, we have incorporated FIFO (First-In, First-Out) Replacement Policies alongside Write-Through Write Policies to ensure efficiency and effectiveness in our cache strategy. These choices align well with our design requirements and use cases.

5.5 Cryptographic Instruction Designs

in Figures 5.11 and 5.12 are the cryptographical instructions with their explanations that our processor is expected to implement.

hmdst

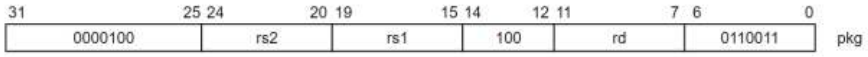
This instruction is used to calculate the Hamming distance of two 32-bit numbers. The instruction coding is as follows:



This instruction reads the rs1 and rs2 registers, it calculates the bit differences between them and writes it to the rd register.

pkg

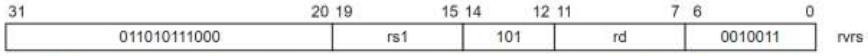
This command packs the least the significant half of the two registers and writes to another register. The instruction coding is as follows:



This instruction combines the least significant half of the values in the rs2 and rs1 registers and writes it to the rd register such that the half of rs2 remains in the most significant part.

rvrs

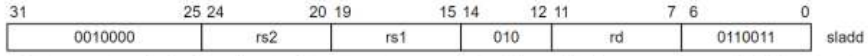
This instruction reverses the byte order of the value in a register. The instruction coding is as follows:



This instruction reads the rs register, reverses the byte order and writes it to the rd register.

sladd

This instruction shifts the value in one register by one to the left and adds the value in another register. The instruction coding is as follows:

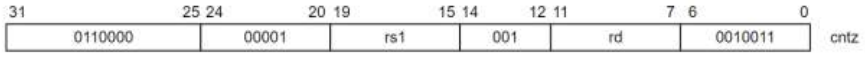


This instruction shifts the value in the rs1 register to the left by 1 bit, then sums it up with the value in rs2 and writes it to the rd register.

Figure 5.11 Cyrptographic Instructions Part 1

cntz

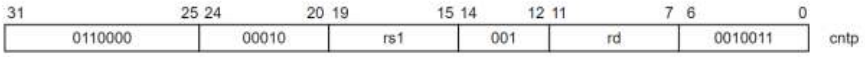
This instruction counts the trailing zeros of a value in a register. The instruction coding is as follows:



This instruction counts the number of 0s from the least significant bit to the most significant bit until a 1 and writes the result to the rd register. If the rs1 register is equal to 0, the result is XLEN, and if the least significant bit is 1, the result is equal to 0.

cntp

This instruction counts the number of ones in a register. The instruction coding is as follows:



It counts the number of ones in rs1 and writes the result to the rd register.

Figure 5.12 Cyrptographic Instructions Part 2

5.6 Input/Output Design

5.6.1 Input Design

- **Instruction Sets and Test Data:** Inputs are primarily provided through a testbench, simulating instruction sets and data inputs to the processor model.
- **Control Signals:** Simulated control signals, such as reset and clock, mimic real-world operational conditions, ensuring accurate behavior of the processor under various scenarios.

5.6.2 Output Design

- **Monitoring Processor States:** Outputs are observed as changes in the internal states, registers, and memory elements, reflecting the processor's responses to inputs.
- **Pipeline Output Analysis:** Each stage of the pipeline, including the results of instruction execution, is monitored for validation and debugging purposes.

- **Output Logging:** For detailed analysis, the simulation environment enables logging of outputs at each cycle or significant events, providing an in-depth view of the processor's operations.

6
Methods

This section is to describe how we implement the 5-staged pipeline to our RISC-V Core, and the required hardware to resolve the pipeline hazards.

6.1 Pipeline Stages

A pipeline stage is basically flip flop, when a clk edge occurs, it passes the input state to the output.

Below is a high level diagram of a the IF/ID stage:

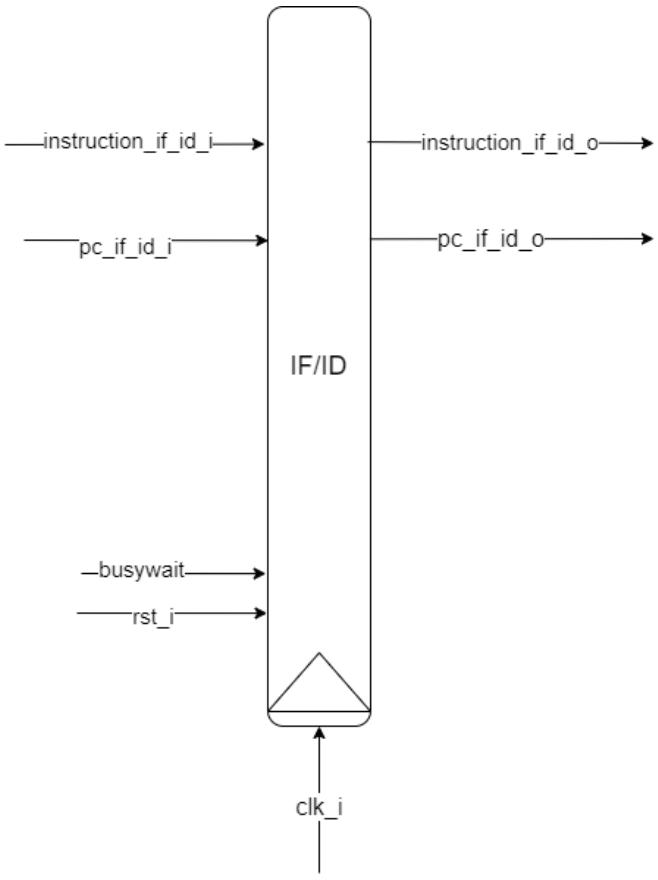


Figure 6.1 IF/ID Stage Pipeline

And from the Verilog perspective

```
module if_id_stage_reg(
    input clk_i,
    input rst_i,
    input busywait,
    input [31:0] instruction_if_id_i,
    input [31:0] pc_if_id_i,//jump birimi PC i kullandigi
       icin onu da iletmemiz gerekiyor boru hattinda
    input stall,
    output reg [31:0] instruction_if_id_o,
    output reg [31:0] pc_if_id_o
    );

    always @(*)
     begin
        if(rst_i)
        begin
        #0.1;
        pc_if_id_o = 32'd0;
        instruction_if_id_o = 32'dx;
        end
     end


     always @(posedge clk_i)
     begin
        #0;
        if(!busywait && !stall)//
            begin
            pc_if_id_o <= pc_if_id_i; //non-blocking
               statements.
            instruction_if_id_o <= instruction_if_id_i;
            end
     end
endmodule
```

Since our all pipeline stages function as the same only the input/output changes dependent on their stage and will not be showed here.

6.2 Forwarding Unit

The forwarding Unit takes the destination register from EX/MEM and MEM/WB, and takes the control signals with the register value. If the instruction will update the destination register (RD) e.g. if its a write instruction and the RD matches RS1 or RS2 in the execution stage. The value should be forwarded from the meant stages.

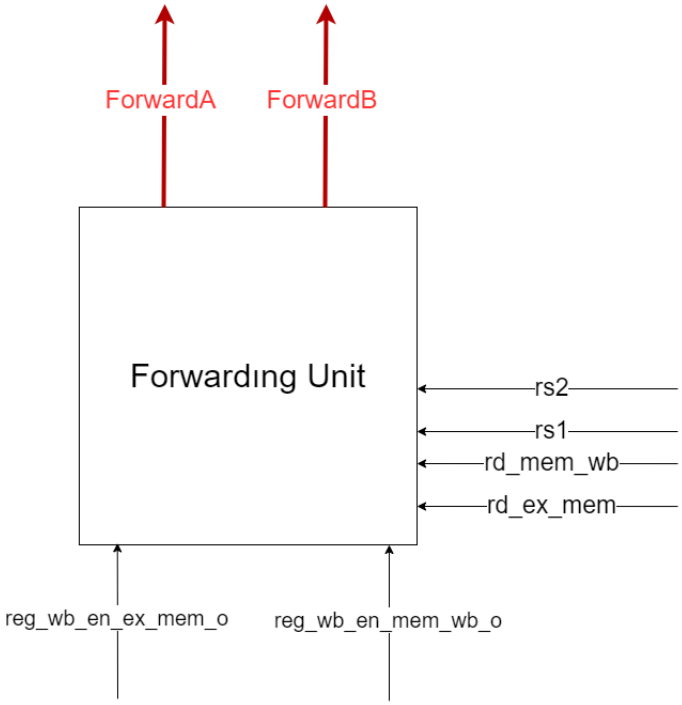


Figure 6.2 High View of Forwarding Unit

now we can implement this logic with simple if statements and logical conditions in verilog like:

```
1
2
3  module forwarding_unit(
4      input [4:0] rd_label_ex_mem_o,
5      input [4:0] rd_label_mem_wb_o,
6      input [4:0] rs1_label_id_ex_o,
7      input [4:0] rs2_label_id_ex_o,
8
9      input reg_wb_en_ex_mem_o,
10     input reg_wb_en_mem_wb_o,
11     input is_memory_instruction,
12     input is_memory_instruction_mem_wb_o,
13     input [6:0] opcode,
14     output reg [1:0]  forwardA,
```

```
15      output reg   [1:0]   forwardB
16      );
17
18      always @(rs1_label_id_ex_o, reg_wb_en_ex_mem_o,
           reg_wb_en_mem_wb_o, rd_label_ex_mem_o,
           rd_label_mem_wb_o,is_memory_instruction_mem_wb_o)
           begin
19
20      if (reg_wb_en_mem_wb_o && (rd_label_mem_wb_o != 0) &&
21              !(reg_wb_en_ex_mem_o && (rd_label_ex_mem_o !=
                   0) && (rd_label_ex_mem_o ==
                   rs1_label_id_ex_o)) &&
22              (rd_label_mem_wb_o == rs1_label_id_ex_o) &&
23            !is_memory_instruction &&
24            !is_memory_instruction_mem_wb_o ) begin
25             forwardA = 2'b01;
26         end
27       else  if(rd_label_mem_wb_o != 0  &&
28                  reg_wb_en_mem_wb_o &&
29                  is_memory_instruction_mem_wb_o &&  // bir
                       onceki islem memory islemi
30                  !is_memory_instruction &&//su an memory
                       islemi yok
31                  $signed(rd_label_mem_wb_o) == $signed(
                       rs1_label_id_ex_o))
32                  begin
33                   forwardA = 2'b11;
34                  end
35
36          else if(rd_label_ex_mem_o != 5'b0  &&
37             reg_wb_en_ex_mem_o &&
38             $signed(rd_label_ex_mem_o) == $signed(
                  rs1_label_id_ex_o) &&
39             !is_memory_instruction)
40             begin
41              forwardA = 2'b10;//RS1 = EX/MEM RD
42             end
43
44          else
45               begin
46               forwardA = 2'b00; //no forwarding
```

```
47              end
48      end
49
50          always @(rs2_label_id_ex_o, reg_wb_en_ex_mem_o,
               reg_wb_en_mem_wb_o, rd_label_ex_mem_o,
               rd_label_mem_wb_o,
               is_memory_instruction_mem_wb_o) begin
51
52          if(rd_label_ex_mem_o  &&
53             reg_wb_en_ex_mem_o &&
54             $signed(rd_label_ex_mem_o) == $signed(
                  rs2_label_id_ex_o) &&
55             !is_memory_instruction &&
56             !is_memory_instruction_mem_wb_o)
57              begin
58                      forwardB = 2'b10; // RS2 = EX/MEM  RD
59              end
60
61          else if(reg_wb_en_mem_wb_o && (rd_label_mem_wb_o
               != 0) &&
62          !(reg_wb_en_ex_mem_o && (rd_label_ex_mem_o != 0)
               && (rd_label_ex_mem_o == rs2_label_id_ex_o)) &&
63          (rd_label_mem_wb_o == rs2_label_id_ex_o) &&
64             !is_memory_instruction && !
                  is_memory_instruction_mem_wb_o) begin
65              forwardB = 2'b01;   //RS2 = MEM/WB RD
66          end
67       else  if(rd_label_mem_wb_o != 0  &&
68                   reg_wb_en_mem_wb_o &&
69                   is_memory_instruction_mem_wb_o &&  // bir
                        onceki islem memory islemi
70                   !is_memory_instruction &&//su an memory
                        islemi yok
71                   $signed(rd_label_mem_wb_o) == $signed(
                        rs2_label_id_ex_o))
72                   begin
73                    forwardB = 2'b11;
74                   end
75          else
76              begin
77              forwardB = 2'b00; //no forwarding
```

```
78                  end
79          end
80          endmodule
```

6.3 Hazard Detection Unit

Hazard Detection Unit is the simplest of all, if there an instruction that depends on a previous load instruction the pipeline must stall, and a new instruction shouldn't be fetched and a NOP instruction should be exectued. A Nop instruction can implemented such as in 6.1

Listing 6.1 NOP instruction in RISC-V Assembly

```
1 addi x0, x0, 0
```

Instead of making our design more complex for fetching specified instruction each time, we can mimic a NOP instruction by resetting the "Memory Write" and "Register Write" signals, thus the instruction won't change the CPU state.

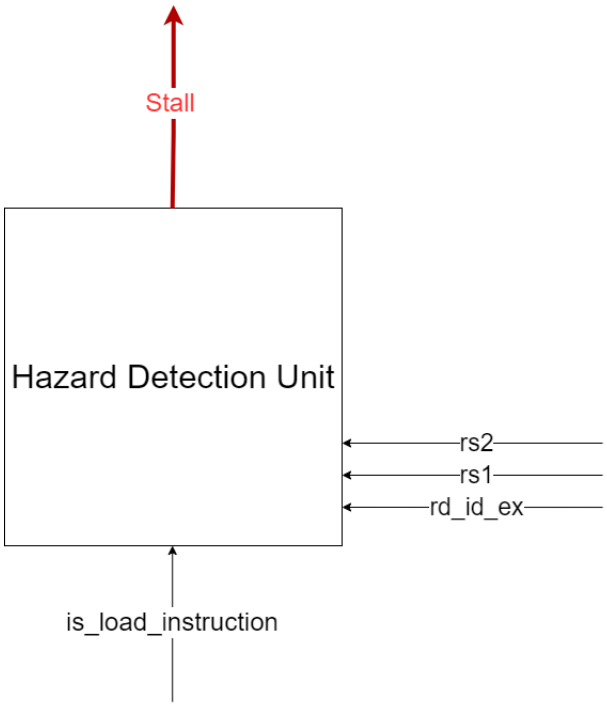


Figure 6.3 High View Of Hazard Detection Unit

and now we can implement the hazard detection unit in verilog like:

```
1 module hazard_detection_unit(
2     input clk_i,  // Clock input
```

```
    input is_load_instruction,
    input [4:0] rd_label_id_ex_o,
    input [4:0] rs1_label_if_id_o,
    input [4:0] rs2_label_if_id_o,

    output reg stall
);

// Temporary register to track hazard condition
reg hazard_detected;

initial begin
    stall <= 1'b0;
    hazard_detected <= 1'b0;
end

always @(posedge clk_i) begin
    if (is_load_instruction && ((rs1_label_if_id_o == rd_label_id_ex_o) || (rs2_label_if_id_o == rd_label_id_ex_o))) begin
        hazard_detected <= 1'b1;
    end
    else begin
        hazard_detected <= 1'b0;
    end

    // Set stall for one clock cycle when hazard is detected
    stall <= hazard_detected;
end

endmodule
```

6.4 Data Path Of Pipelined Architecture

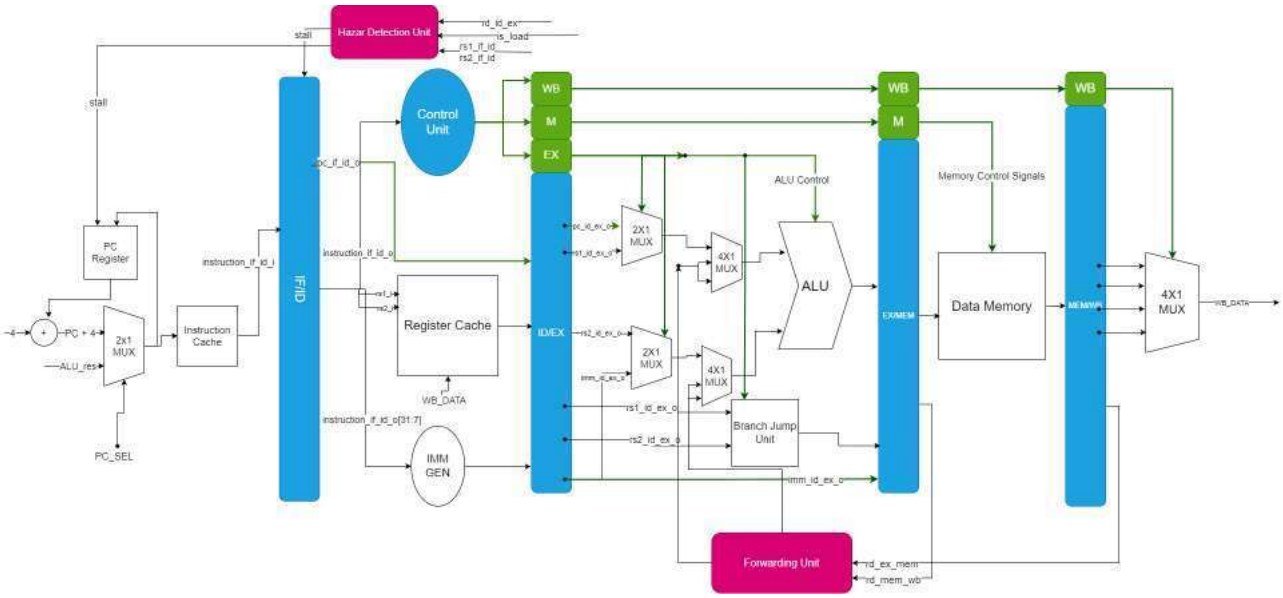


Figure 6.4 Our Design Enhanced With Forwarding And Hazard Units

6.4.1 Cache Design

Cache design is crucial in enhancing data access speed and minimizing latency in computing systems. A cache is a smaller, faster memory component located close to a processor, designed to temporarily hold frequently accessed data and instructions (Figure 6.5). This setup reduces the time it takes for the processor to access data, as retrieving information from cache memory is significantly faster than accessing it from main memory. In the context of this study, two primary cache methodologies are examined. These methodologies aim to optimize cache performance by effectively managing how data is stored and retrieved, thereby improving the overall efficiency of the computing system. By employing advanced cache design strategies, the study seeks to reduce latency and increase data throughput, which is critical for high-performance computing applications.

6.4.1.1 Mapped Cache

Mapped caching involves assigning fixed locations in the cache for data blocks. This direct mapping ensures quick access times but may increase the rate of collisions. The design challenge here is to effectively manage the trade-off between speed and the probability of cache misses.

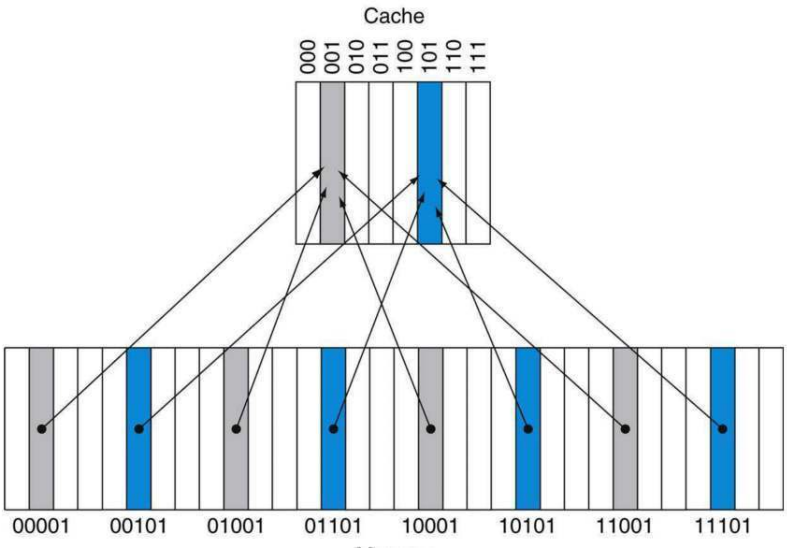


Figure 6.5 Direct Mapped Cache

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Figure 6.6 Direct Mapped Cache Structure

6.4.1.2 Address Resolution in Direct-Mapped Cache

Direct-mapped cache is a method of cache organization where each block of main memory maps to a single, specific cache line. The following sections break down the process of address resolution in such a cache.

Memory Address Breakdown In a direct-mapped cache system, each memory address is divided into four distinct parts(Figure 6.6):

- **Tag**: The upper bits of the address, identifying a unique block of memory. It is required for hit and miss check.
- **Index**: The middle part of the address, indicating the specific cache line to which the memory block maps.
- **Block Offset**: The lower bits of the address, determining the exact word or byte within the cache line that is being accessed.

- **Valid Bit**: A bit associated with each cache line indicating whether the contents of that cache line are valid and up-to-date.

Address Mapping Process The mapping process in a direct-mapped cache works as follows:

1. The cache uses the memory access *index* from the CPU-generated memory address to determine the cache line to inspect.
2. It checks the *valid bit* of the cache line. If the bit is not set (invalid), a cache miss is declared.
3. If the valid bit is set, the cache compares the *tag* stored in that cache line with the *tag* from the memory address. A match indicates a cache hit, implying the data is present in the cache.
4. In the case of a mismatch or an empty cache line, it results in a cache miss. The cache then retrieves the required data block from the main memory and stores it in the specified cache line, setting the valid bit accordingly.

6.4.1.3 First-In, First-Out (FIFO) Replacement Policy

The First-In, First-Out (FIFO) replacement policy is a cache management strategy used in our L1 cache design. This policy is characterized by its simplicity and straightforward approach to cache line replacement.

Under FIFO, when a new data block needs to be loaded into the cache and the cache is full, the cache line that was loaded first (and is the oldest) is replaced. This process is analogous to a queue, where the first element to enter is also the first to leave.

Operation of FIFO Policy: In FIFO policy, each cache line is tagged with a time marker or a similar mechanism that tracks when it was loaded into the cache. Upon a cache miss, the system identifies the cache line with the oldest time marker and replaces it with the new data.

Advantages: The primary advantage of the FIFO replacement policy lies in its simplicity. It does not require complex algorithms to determine which cache line to replace, making it relatively easy to implement and understand. This can lead to faster decision-making processes in cache management.

Disadvantages: However, FIFO can be suboptimal in terms of cache performance. It does not consider the frequency or recency of access to the cached data. As a result, frequently accessed data might be replaced simply because it was the first loaded into the cache, potentially leading to higher cache miss rates in certain scenarios.

Suitability for Our Design: FIFO is good for our RISC-V L1 cache design where simplicity and functionality are key. It's balanced and offers reasonable cache performance without complex algorithms.

6.4.1.4 Write-Through Policy

The Write-Through Policy is a strategy for managing data writing operations in cache memory. This policy is integral to our L1 cache design within the RISC-V architecture. This policy keeps cache sync with main memory

Operation of Write-Through Policy: In Write-Through cache, every write operation to the cache also writes data to the corresponding location in main memory to keep them in sync. This can affect write performance as the system waits for both operations to complete.

Data Integrity and Consistency: The Write-Through policy is highly advantageous in maintaining data consistency and integrity between the cache and main memory. It can be compared to a librarian who instantly logs a new book in both the library's local records (cache) and a central database (main memory) to avoid any discrepancies. This policy is essential in systems where even a slight mismatch in data can have significant consequences. For instance, in banking systems, it's crucial to update the account balance immediately and accurately in both the system's quick-access records (cache) and its core database (main memory) when processing a transaction. Any inaccuracies, even if temporary, may result in incorrect account balances being displayed, leading to confusion or further transaction errors.

Implications on Performance: Write-Through policy ensures data integrity but can slow down the system due to increased traffic on the memory bus. Write-Back policy may be faster as it initially writes data only in the cache.

Suitability for Our Design: The Write-Through policy is a reliable solution for maintaining data accuracy and consistency in the RISC-V architecture. It offers simplicity of implementation and is an appropriate choice for our L1 cache.

6.4.1.5 Example

Imagine a CPU requests data from a memory address, 0x1234ABCD, in a system with a 32-bit address and a direct-mapped cache of 64 lines. We use 6 bits for the index (since $2^6 = 64$), 4 bits for the block offset, and the remaining 22 bits for the tag.

Cache Access Process:

1. Index and Tag Extraction: From 0x1234ABCD, the cache extracts the index (middle 6 bits) and tag (first 22 bits).

2. Valid Bit Check: It checks the valid bit of the identified cache line.

Cache Hit Scenario: - If the valid bit is set and the tag matches, it's a cache hit. The data is quickly accessed from the cache.

Cache Miss Scenario: - If the valid bit is not set or the tag differs, it's a cache miss. The cache fetches and stores the required data from main memory, updating the tag and valid bit.

7
Experimental Outcomes

In this section we will try to test our forwarding unit, Hazard detection unit and the CPU as system.

7.1 Forwarding Unit

To test the forwarding unit we used the code below which have alot of dependencies

Listing 7.1 Sample RISC-V Assembly Code

```
addi x1, x0, 15   #x1 = 15
addi x3, x1, 10   #x3 = 25
add x5, x3, x1    #x5 = 40
addi x6, x5, 9    #x6 = 49
sub x2, x1, x3    #x2 = -10
and x12, x2, x5   #x12 = 32
or x13, x6, x2    #x13 = -9
add x14, x2, x2   #x14 = -20
```

here we can see the sequence of dependencies between the values, now let's examine the values in our waveform

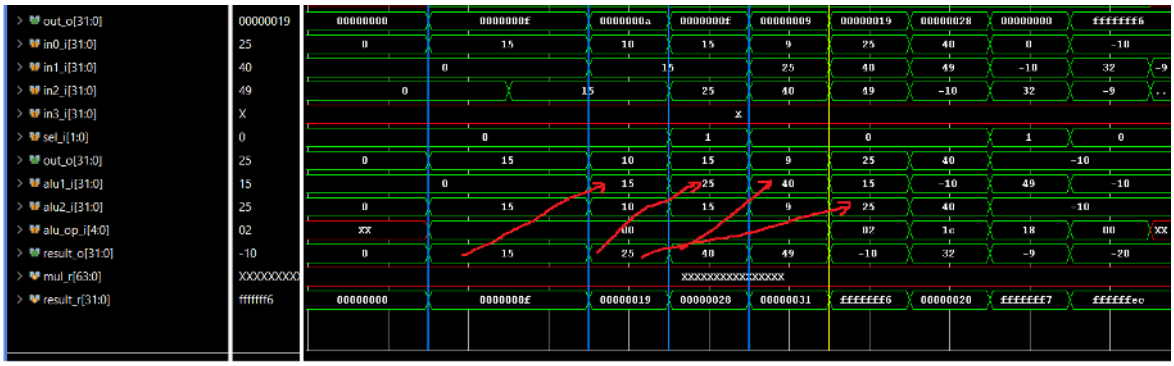


Figure 7.1 Data Forwarding Representation In Waveform

Here we can see how the data is being forwarded from EX/MEM and MEM/WB stages, Now let's check the final result of the registers in Figure 7.2.

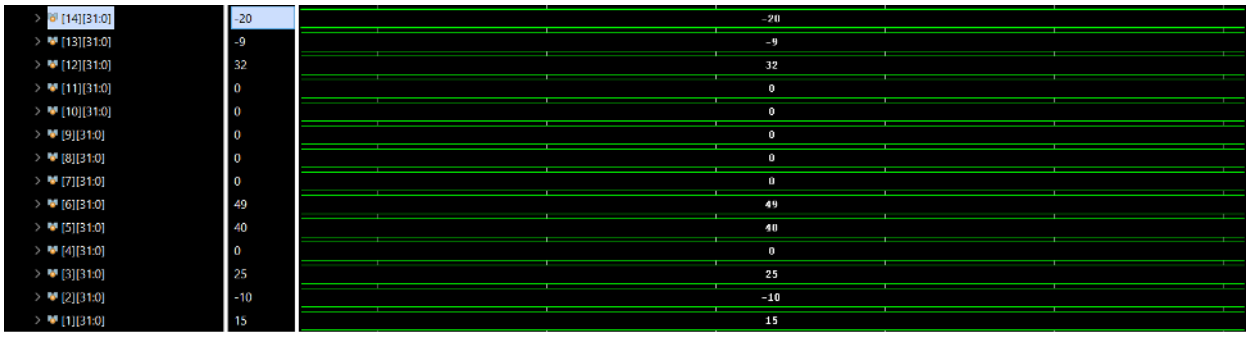


Figure 7.2 Final Result of Registers

7.2 Hazard Unit Detection

Now let's try to mix our forwarding unit with a hazard detection unit in the following example:

Listing 7.2 Sample RISC-V Assembly Code

```
addi x1, x0, 10; x1 = 0x000A
sb x1, 1(x0);    [0x01] = 0x000A
lb x2, 1(x0);    x2 = [0x01] --> 0x000A
```

```
lh x3, 1(x0);    x3 = [0x01][0x00] --> 0x0A00
add x4, x2, x3; X4 = 0x0A0A.
```

After the "lh" instruction, a stall signal is expected and the pipeline should stall from fetching a new instruction, this behavior can be observed in Figure 7.3

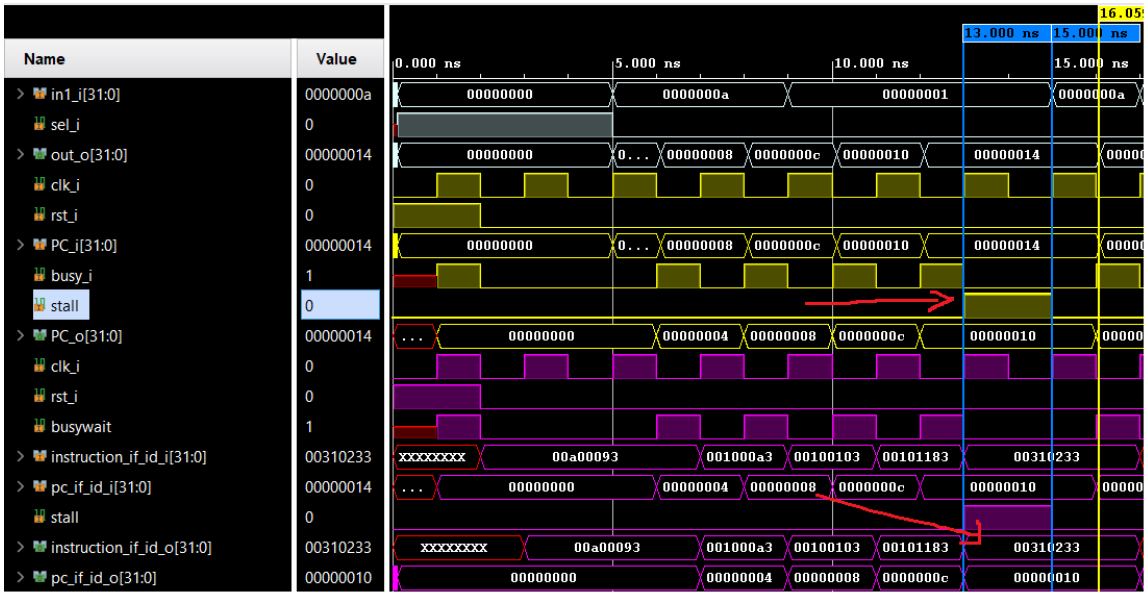


Figure 7.3 Stall Of Pipeline

8
Performance Analysis

8.1 Core Performance Analysis With Basic Commands

Comparing our new pipelined RISC-V Core would be best examined when compared with a single-cycled one, here we will run the following assembly code that contains basic commands and defines performance differences.

Listing 8.1 Sample RISC-V Assembly Code

```
addi x1, x0, 10; x1 = 0x000A
sb x1, 1(x0);    [0x01] = 0x000A
lb x2, 1(x0);    x2 = [0x01] --> 0x000A
lh x3, 1(x0);    x3 = [0x01][0x00] --> 0x0A00
add x4, x2, x3; X4 = 0x0A0A.
sub x5, x4, x1; x5 = 0x0A00
addi x31, x0, 255; x31 = 0x00FF
mul x6, x5, x4;  0x00646400
mulh x7, x6, x5; x7 = 0x0003
addi x30, x0, 1; x30 = 1
sub x8, x3, x30; x8 = 0x09FF
divu x9, x6, x8; x9 = 0x0A0B
remu x10, x6, x8; x10 = 0x000B
xor x29, x8, x8; x29 = 0x0000
divu x28, x6, x29; x27 = 0xFFFFFFFF
addi x27, x0, -255; x28 = 0xFFFFFF01
div x26, x6, x27; x26 = 0xFFFF9B38
lui x11, 1; x11 = 0x1000
auipc x12, 1; x12=0x1048
add x24, x0, x1; x24 = 0x000a
add x25, x0, x1; x25 = 0x000a
```

Now let's Compare our waveform results and see the last Writeback cycle, assigning 0x00a to register 25.

Figure 8.1 Single Cycle Core Waveform

Notice that register r25 is getting written at 123ns which means the end of execution, and now lets examine our performance in our pipelined design in Figure 8.2

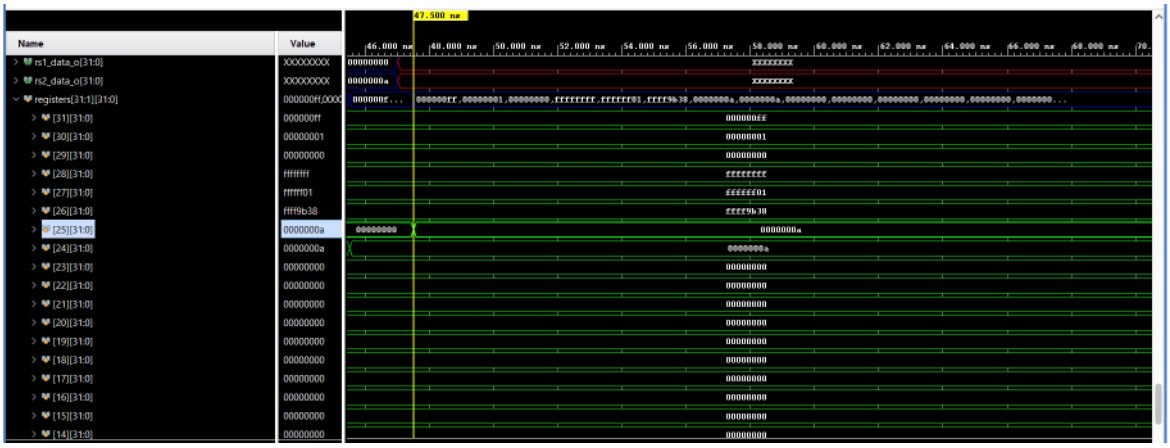


Figure 8.2 Pipelined Core Waveform

which finishes at 47.5ns so we can say approximately our pipelined is 3x faster thanks to our increased clock frequency.

8.1.1 Performance Analysis With Bubble Sort

Bubble Sort is a basic sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. This process is repeated until no swaps are needed, which means the list is sorted. We also tried this algorithm for our two cores with single-cycle and pipeline design.

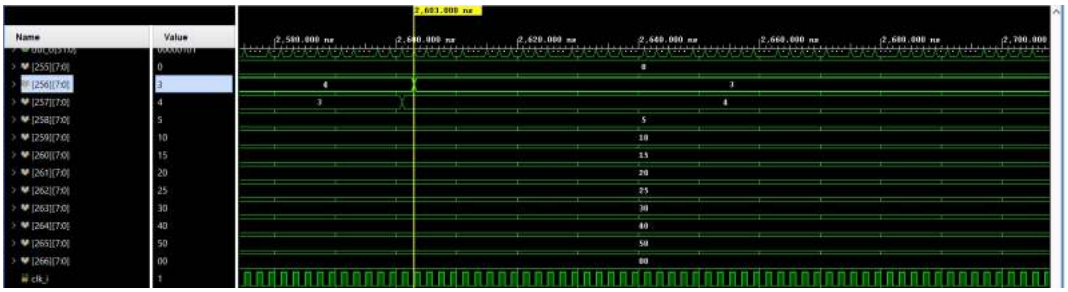


Figure 8.3 Bubble Sort Pipeline Supported Processor

Figure 8.4 Bubble Sort Non-Pipeline Processor

Table 8.1 Bubble Sort Algorithm Performance

	Pipeline Supported Processor (ns)	Non-Pipeline Supported Processor (ns)
Worst Case	2603	5236
Average Case	362	1324

As seen in Table 8.1, the design supporting pipelining offers significantly superior performance for bubble sort compared to the non-pipelined design. It is approximately twice as fast in the worst case and about four times faster in the average case. You can see Figure 8.4 and Figure 8.3 waveform of the bubble sort algorithm.

8.1.2 Performance Difference Between Branch Prediction Methods

Listing 8.2 Code snippet from the image

```
int i = 0;
while(i < 10){
    // rest of the code
    i++;
}
```

Compare the success of branch prediction for the code snippet shown in the figure.

Never branches:

- i = 0 does not branch (Correct prediction)
- i = 1 does not branch (Correct prediction)
- ...
- i = 10 branches (Wrong prediction)

10 Correct predictions, 1 wrong. accuracy ≈ 90.9%

Always branches:

- i = 0 does not branch (Wrong prediction)
- i = 1 does not branch (Wrong prediction)
- ...
- i = 10 branches (Correct prediction)

10 Wrong predictions, 1 Correct. accuracy ≈ 9.09%

8.2 Performance Benchmark for Cache

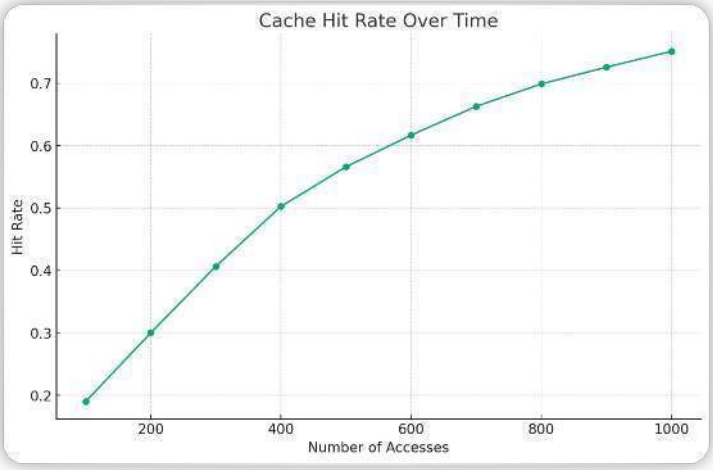


Figure 8.5 Cache Hit Rate Over Time

As shown in Figure 8.5, the cache hit rate increases with more accesses. Initially, at 100 accesses, the hit rate is 19%, which steadily climbs to 75.1% by the time it reaches 1000 accesses, demonstrating an improvement in cache efficiency over time.

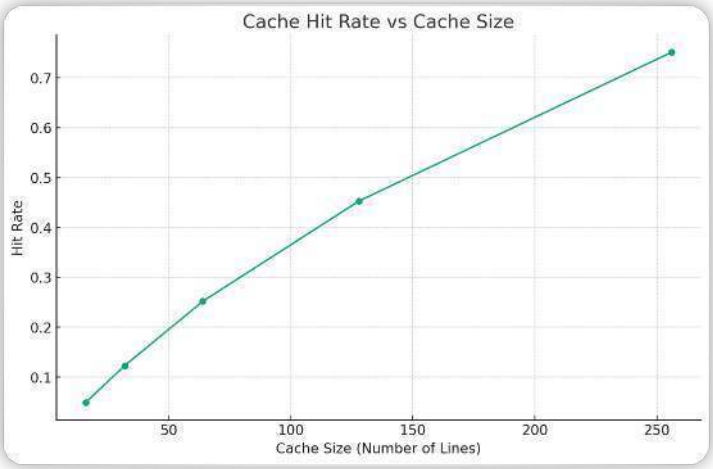


Figure 8.6 Cache Hit Rate Vs Cache Size

As shown in Figure 8.6 the cache size increases also the hit rate increases. However, cache size larger than 256 does not provide higher performance.

9
Conclusion

In Summary we have developed our 5-staged pipeline and enhanced it with Forwarding Unit to resolve data hazards, and a naive branch prediction for control hazards and bringing L1 Cache Layers for our instruction and data memories hierarchies reduce our execution time this is because memory access is considered the bottleneck in most Computer systems.

References

[1] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," *ACM SIGARCH Computer Architecture News*, vol. 8, no. 6, pp. 25–33, 1980.

[2] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2013, pp. 1–12.

[3] "Apple pays for arm license." Accessed: 2023-12-02. (2021), [Online]. Available: https://forums.appleinsider.com/discussion/234446/arm-wants-more-than-0-30-per-iphone-from-apple-but-wont-get-it.

[4] "Risc-v information." (2013), [Online]. Available: https://riscv.org/about/ (visited on 12/02/2023).

[5] E. Blem, J. Menon, T. Vijayaraghavan, and K. Sankaralingam, "Isa wars: Understanding the relevance of isa being risc or cisc to performance, power, and energy on modern architectures," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 1, pp. 1–34, 2015.

[6] "Teknofest chip design specification." Accessed: 2023-12-02. (2021), [Online]. Available: https://cdn.teknofest.org/media/upload/userFormUpload/Teknofest_Cip_Sartnamesi_2023_v8_paylasilan_naLIQ.pdf.

[7] "Teknofest chip design specification." (2021), [Online]. Available: https://www.teknofest.org/en/competitions/competition/121 (visited on 12/02/2023).

[8] RISCV. "Riscv-opensoruce." (2021), [Online]. Available: https://riscv.org/about/ (visited on 12/02/2023).

[9] A. Raveendran, V. B. Patil, D. Selvakumar, and V. Desalphine, "A risc-v instruction set processor-micro-architecture design and analysis," in *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*, IEEE, 2016, pp. 1–7.

[10] S. L. Harris and D. Harris, "Digital design and risc-v computer architecture textbook," in *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, 2021, pp. 300–315. DOI: 10.1109/WCAE53984.2021.9707615.

[11] M. Aagaard and M. Leeser, "Reasoning about pipelines with structural hazards," in *International Conference on Theorem Provers in Circuit Design*, Springer, 1994, pp. 13–32.

[12] W. P. Kiat, K. M. Mok, W. K. Lee, H. G. Goh, and I. Andonovic, "A comprehensive analysis on data hazard for risc32 5-stage pipeline processor," in *2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, IEEE, 2017, pp. 154–159.

[13] J. E. Smith, "A study of branch prediction strategies," in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 202–215.

[14] M. Kowarschik and C. Weiß, "An overview of cache optimization techniques and cache-aware numerical algorithms," *Algorithms for memory hierarchies: advanced lectures*, pp. 213–232, 2003.

[15] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

Curriculum Vitae

FIRST MEMBER

Name-Surname: Ahmet Akib GÜLTEKİN
Birthdate and Place of Birth: 16.07.2002, Elazığ
E-mail: akib.gultekin@std.yildiz.edu.tr
Phone: 0552 582 23 02
Practical Training: TUBITAK BILGEM

SECOND MEMBER

Name-Surname: Basel KELZİYE
Birthdate and Place of Birth: 21.06.2001, Halep
E-mail: basel.kelziye@std.yildiz.edu.tr
Phone: 0535 569 62 46
Practical Training: TÜBİTAK Bilgem

Project System Informations

System and Software: Windows İşletim Sistemi, Python, C++
Required RAM: 16GB
Required Disk: 4096MB