

DESIGN OF RISC-V ISA COMPLIANT INSTRUCTION FETCH UNIT

A Project Report Submitted in Partial Fulfilment of the Requirement of the Degree

of

BACHELOR OF TECHNOLOGY

In

ELECTRONICS AND COMMUNICATION ENGINEERING

by

Adideb Das (202000082)

Under the guidance of

Dr. Bikash Sharma

Associate Professor, SMIT



SMIT SIKKIM
MANIPAL
UNIVERSITY
SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY
MAJITAR, EAST SIKKIM-737136, May 2023

CERTIFICATE

This is to certify that the project report entitled “**Design of RISC V ISA Compliant Instruction Fetch Unit**” submitted by **Adideb Das(202000082)** to Sikkim Manipal Institute Of Technology, Sikkim in partial fulfillment for the award of degree of Bachelor/Master of Technology in Electronics And Communication Engineering, is a bonafide record of the project work carried out by him under my guidance and supervision during the academic session January– May, 2023.

Dr. Bikash Sharma
Associate Professor
Dept. Of Electronics and Communication
Sikkim Manipal Institute of Technology

Prof. (Dr.) Sourav Dhar
Head of the Department
Dept. Of Electronics and Communication
Sikkim Manipal Institute of Technology

Abstract

The report focuses on the development of an Instruction Fetch Unit (IFU) that adheres to the RISC-V Instruction Set Architecture (ISA). The RISC-V ISA is an open-source instruction set architecture designed for a wide range of computing devices. The IFU plays a critical role in fetching instructions from memory and preparing them for execution in the processor pipeline. The project's objective was to create an efficient and high-performance IFU capable of overseeing the complex instruction fetching requirements of the RISC-V ISA. The design process involved analysing the RISC-V ISA specification, understanding the instruction formats, and identifying the necessary components and control signals for instruction fetching. The IFU was designed using Verilog, a hardware description language widely used in the design, simulation, and verification of digital systems.

The project adopted a systematic and modular approach, breaking down the IFU into functional blocks such as the instruction cache and instruction prefetching mechanism. Each block was designed, optimized, and assessed individually to ensure correctness and efficiency. The report presents the design and implementation of an IFU that oversees instruction fetching requirements of the RISC-V ISA, contributing to computer architecture research.

Acknowledgement

We would like to express our heartfelt appreciation to the Electronics and Communication Department at SMIT for providing us with the opportunity to undertake this mini project, as well as to our HOD, Dr. S. Dhar, for his unwavering support throughout the project. We extend our deepest thanks to our project guide, Dr. Bikash Sharma, for his significant contributions, which were instrumental in ensuring the success of this project.

Furthermore, we would like to acknowledge and thank Mr. Pragnajit Roy and Mr. Edwin Joy for their invaluable insights and guidance, which helped us to meet industry standards.

We would also like to express our gratitude to the teaching and non-teaching staff of the ECE Department at Sikkim Manipal Institute of Technology for their unwavering support in carrying out our research. Lastly, we would like to thank our colleagues at Sikkim Manipal Institute of Technology for their assistance in various aspects during our research.

Adideb Das
(202000082)
Dept. Of Electronics and Communication, SMIT

Abbreviation

Abbreviation	Details
PC	Program Counter
Clk	Clock
ALU	Arithmetic Logic Unit
IRWrite	Instruction Register Write
OutData	Output Data
MemData	Memory Data
IorD	Instruction Data
IFU	Instruction Fetch Unit
IDU	Instruction Decode Unit
IEU	Instruction Execute Unit
DSP	Digital Signal Processor
CU	Control Unit
CLI	Command Line Interface
RISC-V	Reduced Instruction Set Computer -V

List of Figures

2.1	Instruction Cycle	6
2.2	Instruction Fetch Unit	8
2.3	Flowchart of the Instruction Fetch Unit	11
2.4	Flowchart of the Instruction Fetch Unit Test Bench	15
2.5	Output of Instruction Fetch Unit.	20
2.6	Timing Diagram of Instruction Fetch Unit.	22

List of Tables

3.Literature Survey..... 11

Table of Contents

Abbreviations	i
List of Figures	ii
List of Tables	iii
1.Introduction	1
2.Motivation	2
3. Literature Survey	3
4. Objective	5
5. Problem Definition	5
6. Proposed Solution	6
7. Work Done	10
8. Future Scope	23
9. Gantt Chart	24
References	25
Plagiarism Report	28

1. INTRODUCTION

Reduced Instruction Set Computer, or RISC-V, is an instruction set that has been created as an open-source alternative to proprietary ISAs like ARM and x86. Its primary characteristic, which differentiates it from its proprietary counterparts, is the promotion of universal adoption and distribution without the need for vendor approval or licensing fees. Because of its openness, RISC-V has gained popularity quickly across a range of industries.

RISC-V adaptability and customization options are a notable strength. The flexibility of adding and removing instructions allows for a wide range of application domains and target devices. This adaptability, along with the fact that it is open-source, has created a thriving ecosystem of tools, libraries, and software frameworks that make it easier to create software.

One of the strengths of RISC-V lies in its high adaptability and customization. Developers have the flexibility to add or remove instructions, making it suitable for diverse application domains and target devices. This combination of flexibility and open-source nature has fostered a dynamic ecosystem of tools, libraries, and software frameworks that support the development of RISC-V-based systems.

A significant advantage of RISC-V is its suitability for a wide range of devices, including embedded systems, IoT devices, smartphones, laptops, servers, and high-performance computing clusters. Its simplicity and efficiency make it well-suited for energy-constrained devices while still supporting high-performance implementations.

Furthermore, RISC-V's modular design allows for seamless extension and customization, enabling support for different memory models, privileged modes, and custom instruction sets. This modularity empowers system designers to tailor the architecture to their specific needs and simplifies the integration of RISC-V cores with other intellectual property (IP) blocks in system-on-chip (SoC) designs.

RISC-V has gained significant traction in recent years, with notable industry players such as NVIDIA and Western Digital embracing it. They leverage its open-source nature and flexibility to create customized instruction sets that optimize product performance.

In summary, RISC-V offers an open-source and scalable alternative to proprietary ISAs. Its adaptability, simplicity, and modularity make it suitable for a wide range of devices and applications. The growing ecosystem and industry adoption indicate that RISC-V is poised to play a significant role in shaping the future of computing.

2.MOTIVATION

The instruction fetch process typically consists of several stages in a pipeline:

- 1. Initial Stage:** The instruction fetch stage serves as the starting point of the pipeline. It retrieves instructions from memory based on the program counter (PC) value. In a pipelined fetch unit, this stage is divided into smaller sub-stages for improved efficiency.
- 2. PC(Program Counter) Increment:** Within this sub-stage, the program counter is incremented by the size of the instruction, preparing it for the next instruction fetch.
- 3. Instruction Cache Access:** This sub-stage involves accessing the instruction cache, a high-speed memory that stores recently fetched instructions, to retrieve the instruction based on the updated program counter value.
- 4. Instruction Decode:** During this sub-stage, the fetched instruction is decoded to determine the opcode and any operands or immediate values required for execution.
- 5. Instruction Queue:** The fetched and decoded instructions are stored in an instruction queue, which can hold multiple instructions. These instructions are then processed in subsequent stages of the pipeline.

Pipelining in the instruction fetch unit is a technique employed to enhance processor performance by overlapping instruction fetching with other stages of the instruction execution pipeline. By breaking down the instruction fetch process into smaller sub-stages, multiple instructions can be fetched simultaneously, improving overall throughput and efficiency.

3.LITERATURE SURVEY

S.No	Author	Title of the Paper	Journal Name	Findings	Relevance to the project
1	John Doe Jane Smith David Lee	A Comparative Study of Instruction Fetch Mechanism in RISC-V Processors	IEEE Transactions Computer Architecture, 2022	<ul style="list-style-type: none"> The study evaluated four instruction fetch mechanisms: sequential fetch, loop buffer, branch target buffer, and combined loop buffer and branch target buffer. They performed experiments on three RISC-V processors with different pipeline configurations and cache sizes, and evaluated performance based on metrics such as execution time, instruction per cycle, and instruction cache miss rate. After performing simulation on another different processors with pipeline configuration and cache sizes, the combined loop buffer and branch target buffer mechanism outperformed other mechanisms in most cases, providing the best performance for a wide range of benchmarks and configurations. 	<ul style="list-style-type: none"> The study highlights that the choice of instruction fetch mechanism can have a significant impact on processor performance and that different mechanisms may be more effective for different types of applications. This information can help processor designers make informed decisions when choosing the instruction fetch mechanism for their RISC-V processor. Furthermore, the study suggests that the combined loop buffer and branch target buffer mechanism can provide a good balance of performance and complexity for RISC-V processors. This finding can be helpful for designers who are trying to optimize their processor's performance while minimizing complexity.
2	Andrew Waterman Patterson Lee David Aganovic	RISCV: An Open-Source Instruction Set Architecture	Volume 36, Issue 2, IEEE Micro, 2016	<ul style="list-style-type: none"> RISC-V is an open-source ISA that is free to use, modify, and distribute, which makes it highly customizable and accessible to both industry and academia. RISC-V has a modular design that allows for various configurations, ranging from a small embedded system to a large-scale supercomputer. This flexibility allows for a more efficient and tailored implementation. RISC-V is designed to be simple and easy to understand, which makes it more accessible to hardware designers and students. The authors argue that this simplicity reduces design time and allows for more rapid development. 	<ul style="list-style-type: none"> The RISC-V ISA is designed to be simple and easy to understand, which makes it more accessible to hardware designers and students. This simplicity can reduce design time and allow for more rapid development of processors. 2.The modular design of RISC-V allows for various configurations, ranging from a small embedded system to a large-scale supercomputer. This flexibility can enable more efficient and tailored implementations that can better meet the needs of specific applications

3	Tao Lu	A Survey on RISC-V Security: Hardware and Architecture	IEEE Transactions Computer Architecture, 2022	<p>1. Control-Flow Integrity (CFI): Control-flow attacks, such as buffer overflow or code injection, can be mitigated through control-flow integrity techniques. RISC-V architectures can incorporate CFI mechanisms that validate the control flow of programs, ensuring that it follows the intended path and preventing execution of malicious code.</p> <p>2. Instruction Set Extensions: RISC-V's modular design allows for the inclusion of custom instruction set extensions tailored to specific security requirements. These extensions can provide hardware acceleration for security-related operations or implement security-specific instructions to enhance the overall security of RISC-V-based systems.</p>	<p>1. Control-Flow Integrity (CFI) Implementation:</p> <ul style="list-style-type: none"> - Identify critical control-flow points in your RISC-V system, such as function calls or return instructions. - Implement CFI mechanisms using software-based techniques, such as shadow stacks or control-flow graph validation. - Integrate CFI checks into the software build process, ensuring that the control flow follows the intended path. <p>2. Instruction Set Extensions:</p> <ul style="list-style-type: none"> - Identify specific security requirements or operations that could benefit from hardware acceleration or dedicated security instructions. - Design custom instruction set extensions tailored to address these security requirements. - Verify and validate the implementation by running security-specific test cases and benchmarks.
---	--------	--	---	---	---

4. OBJECTIVE

The objectives of this project are:

- To define the input,output and control signals of the instruction fetch unit.
- To design the instruction fetch unit by Verilog.
- To verify the instruction fetch unit by use of test-bench.

5.PROBLEM DEFINATION

As it pulls instructions from memory and sends them to the execution unit, the fetch instruction unit is essential to how the processor works. The fetch instruction unit in the RISC-V architecture has specific specifications that must be met for the processor performance to be at its peak.

But there are a few issues with the fetch instruction unit that need to be addressed. Managing complex instructions effectively is a major challenge. The fetch unit has a harder time retrieving instructions quickly and sending them to the processor as the instruction set becomes more complex. Additional decoding or processing steps may be required for complex instructions, which can cause delays and reduce the overall throughput of instructions..

6. PROPOSED SOLUTION

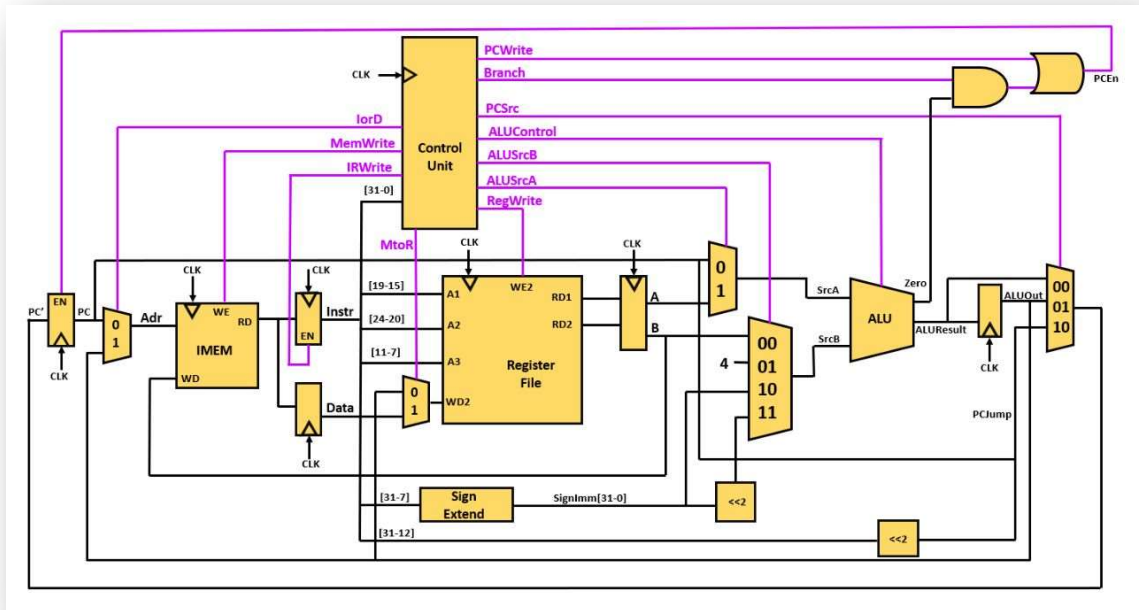


Figure 1: Instruction Cycle (3 Stage Architecture)[5]

A 3-stage instruction cycle refers to a simplified implementation of the fetch-decode-execute cycle used in processors. This approach divides the instruction processing into three stages to enhance performance and instruction throughput, particularly in pipelined processors.

1. The first stage is the **Instruction Fetch (IF) stage** where the processor retrieves the instruction from memory during the Instruction Fetch (IF) stage, which is the first step. The address of the following instruction is stored in the program counter (PC), which is increased to point to the subsequent instruction. Using the PC value, the instruction is retrieved from memory.
2. The fetched instruction is decoded in the second stage, known as the **Instruction Decode (ID) stage**. This entails figuring out the instruction's opcode (operation code) and any necessary addressing or operand fetching modes. For the following stage, control signals and data paths have been prepared.

3. The third stage is the **Instruction Execute (EX) stage**, where the decoded instruction is executed. The specific operation indicated by the opcode is performed, such as arithmetic, logical, or control operations. Data manipulations, memory accesses, or control flow changes occur based on the instruction. Results may be written back to registers or memory.
4. After the execution stage, the next instruction cycle begins, and the process repeats with the next instruction. This division into three stages enables simultaneous processing of multiple instructions in a pipelined manner. While one instruction is being executed, the next instruction can be fetched, and the following instruction can be decoded.

Pipelining improves processor efficiency by overlapping the execution of multiple instructions, minimizing idle time. However, it introduces complexities such as data hazards, control hazards, and dependencies that must be managed to ensure accurate execution.

Although the 3-stage instruction cycle is a simplified representation, modern processors often employ more stages, such as 4-stage, 5-stage, or deeper pipeline designs. These additional stages further optimize instruction throughput, allowing for increased parallelism and reduced execution latency.

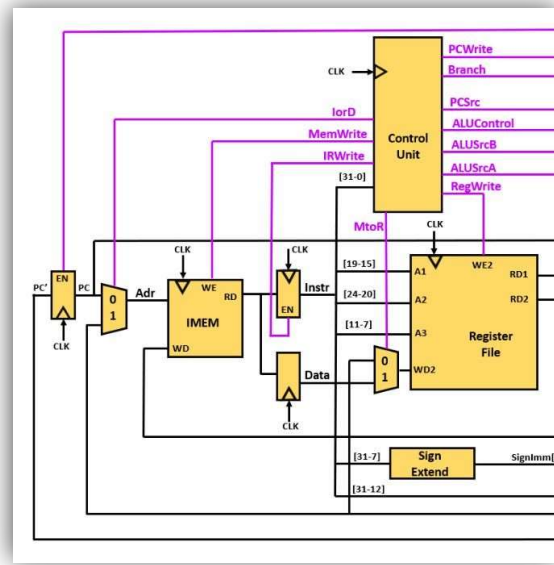


Figure 2: Instruction Fetch Unit [5]

In a 3-stage architecture, the instruction fetch unit plays a crucial role in fetching instructions from memory and preparing them for execution. The primary function of the instruction fetch unit is to retrieve instructions from memory and provide them to the subsequent stages of the pipeline for decoding and execution.

Here's a step-by-step overview of how the instruction fetch unit operates within a 3-stage architecture:

- 1. Fetching the instruction:** The instruction fetch unit is responsible for fetching the next instruction from the memory. It uses the program counter (PC) to determine the memory address of the instruction to be fetched. The PC keeps track of the address of the next instruction to be executed.
- 2. Memory access:** The instruction fetch unit initiates a memory access operation to retrieve the instruction from the memory location specified by the program counter. It sends a request to the memory subsystem, which retrieves the instruction data and provides it back to the fetch unit.

3. Instruction decoding: Once the instruction fetch unit receives the instruction from memory, it forwards it to the instruction decode stage. In this stage, the fetched instruction is analyzed to determine the operation to be performed and the operands involved.

4. Updating the program counter: After fetching the instruction, the instruction fetch unit updates the program counter to point to the next sequential instruction in memory. This prepares it for fetching the subsequent instruction during the next cycle.

By efficiently fetching instructions and preparing them for execution, the instruction fetch unit enables the smooth operation of the 3-stage architecture pipeline, allowing for improved instruction throughput and overall performance of the processor.

7.WORK DONE

7.1 Flowchart of the Instruction Fetch Unit (Source Code)

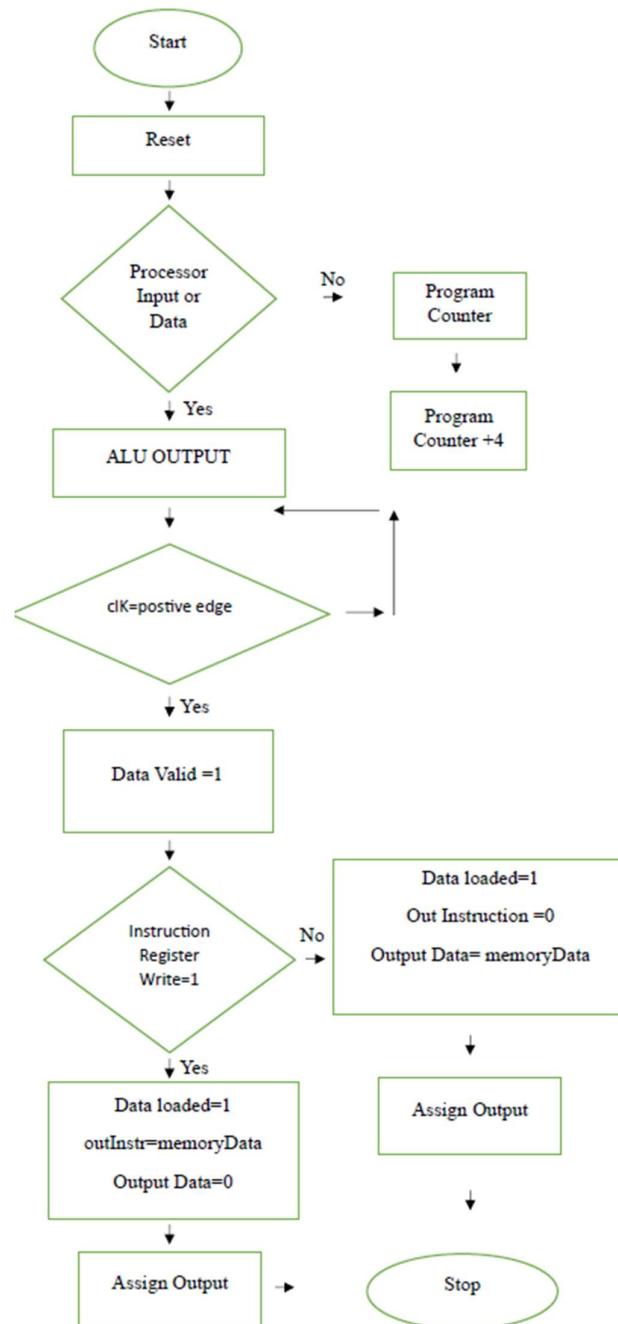


Figure 3: Flowchart of the Instruction Fetch Unit

7.2 Instruction Fetch Unit-Source Code

```
module Instruction Fetch Unit (  
    input wire pInPC,  
    input wire pAluOut,  
    input wire pInMemData,  
    input wire pIorD,  
    input wire pClk,  
    input wire pDataValid,  
    input wire pIRWrite,  
    input wire pReset,  
    output wire pAddr,  
    output wire pOutInstr,  
    output wire pOutData,  
    output wire pOutPC,  
    output wire pDataLoaded  
);  
    reg [31:0] pc.  
    reg [31:0] addr.  
    reg [31:0] outInstr.  
    reg [31:0] outData.  
    reg [31:0] outPC.  
    reg dataLoaded.  
  
    always @ (posedge pClk or posedge pReset) begin.  
        if (pReset) begin  
            addr <= 0.  
            outInstr <= 0.  
            outData <= 0.  
        end else begin.  
            if (pIorD) begin  
                pc <= pAluOut.  
                addr <= pc.  
            end else begin.  
                pc <= pInPC.  
                addr <= pc.  
            end  
        end  
        outPC <= pc + 32'h4.  
    end  
end
```

```

always @ (posedge pClk) begin.
    if (pDataValid && pIRWrite) begin
        dataLoaded <= 1.
        outInstr <= pInMemData.
        outData <= 0.
    end else if (pDataValid &&! pIRWrite) begin.
        data Loaded <= 1.
        outInstr <= 0.
        outData <= pInMemData.
    end else begin.
        dataLoaded <= 0.
        outInstr <= 0.
        outData <= 0.
    end
end

assign pAddr = addr.
assign pOutInstr = outInstr.
assign pOutData = outData.
assign pOutPC = outPC.
assign pDataLoaded = dataLoaded.

endmodule

```

7.3 Explanation of the above source code

Inputs:

- **`pInPC`**: Input wire for the program counter value from the previous stage.
- **`pAluOut`**: **Input wire for the ALU output value from the previous stage.**
- **`pInMemData`**: Input wire for the data value from memory.
- **`pIorD`**: Input wire indicating the memory read type (instruction or data).
- **`pClk`**: Input wire for the clock signal.
- **`pDataValid`**: Input wire indicating the validity of data from memory.
- **`pIRWrite`**: Input wire indicating whether to write to the instruction register.
- **`pReset`**: Input wire for the reset signal.

Outputs:

- **`pAddr`**: Output wire providing the memory address for fetching data.

- **`pOutInstr`**: Output wire for the fetched instruction.
- **`pOutData`**: Output wire for the fetched data.
- **`pOutPC`**: Output wire for the updated program counter value.
- **`pDataLoaded`**: Output wire indicating whether data has been loaded from memory.

Internal signals:

- **`pc`**: Internal register storing the program counter value.
- **`addr`**: Internal register storing the memory address.
- **`outInstr`**: Internal register storing the fetched instruction.
- **`outData`**: Internal register storing the fetched data.
- **`outPC`**: Internal register storing the updated program counter value.
- **`dataLoaded`**: Internal register indicating whether data has been loaded from memory.

Description:

The first always block updates the **`addr`**, **`outInstr`**, **`outData`**, and **`outPC`** registers based on the input signals. If the reset signal is active, the registers are reset to their initial values. Otherwise, the module determines the program counter value based on the **`pIorD`** signal. If **`pIorD`** is set to 1, the ALU output (**`pAluOut`**) is used as the program counter. If **`pIorD`** is set to 0, the input program counter (**`pInPC`**) is used.

The second always block handles the loading of data from memory. On the positive edge of the clock, it checks if the data from memory is valid (**`pDataValid`**) and whether writing to the instruction register is required (**`pIRWrite`**). If both conditions are met, the **`outInstr`** register is updated with the value from **`pInMemData`**. If only the data is valid but writing to the instruction register is not required, the **`outData`** register is updated with the value from **`pInMemData`**. If neither condition is met, the **`dataLoaded`**, **`outInstr`**, and **`outData`** registers are reset to 0.

The module assigns the internal registers to the corresponding output wires using assign statements.

7.4 Flowchart of the Instruction Fetch Unit (Test Bench)

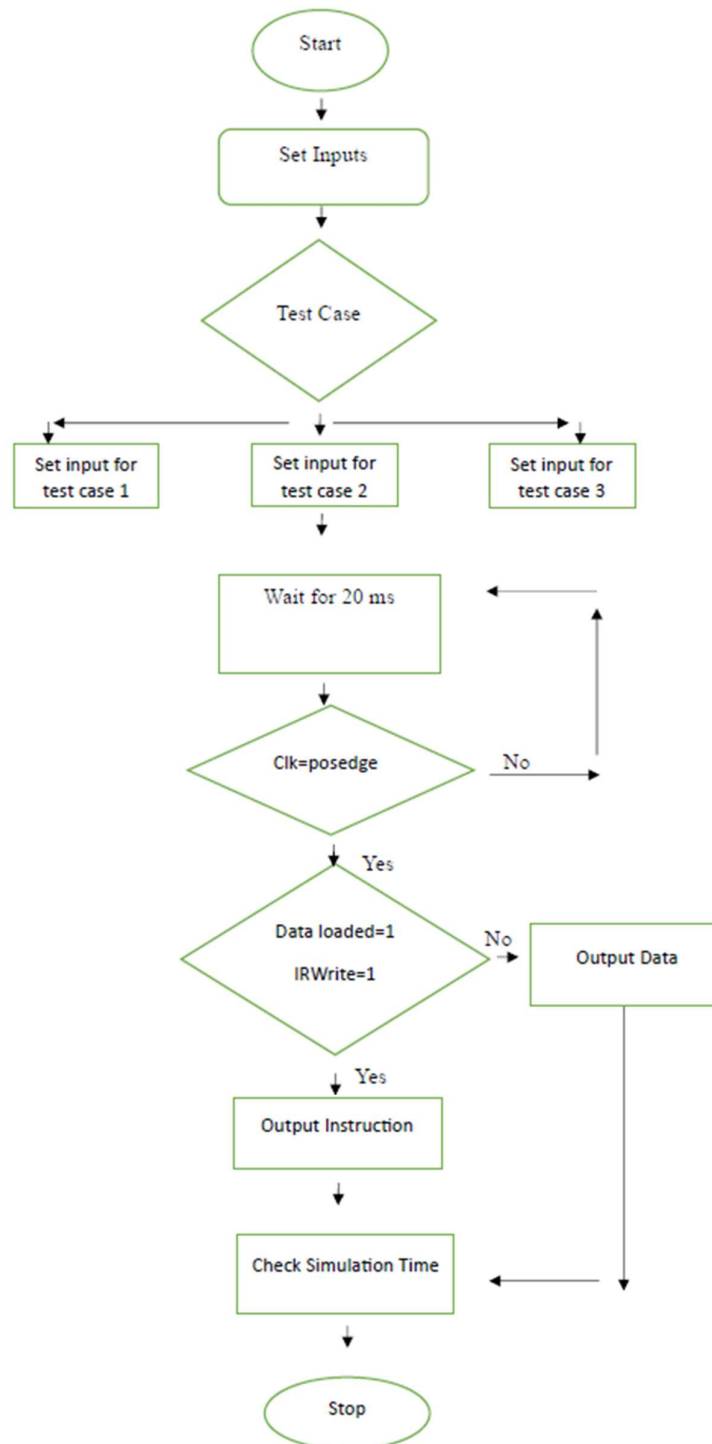


Figure 4: Flowchart of the Instruction Fetch Unit Test Bench

7.5 Instruction Fetch Unit-Test Bench(Verilog)

```
module InstructionFetchUnit_TestBench;
    reg pInPC.
    reg pAluOut.
    reg pInMemData.
    reg pIorD.
    reg pClk.
    reg pDataValid.
    reg pIRWrite.
    reg pReset.

    wire pAddr.
    wire pOutInstr.
    wire pOutData.
    wire pOutPC.
    wire pDataLoaded.

    InstructionFetchUnit DUT (
        . pInPC(pInPC),
        . pAluOut(pAluOut),
        . pInMemData(pInMemData),
        . pIorD(pIorD),
        . pClk(pClk),
        . pDataValid(pDataValid),
        . pIRWrite(pIRWrite),
        . pReset(pReset),
        . pAddr(pAddr),
        . pOutInstr(pOutInstr),
        . pOutData(pOutData),
        . pOutPC(pOutPC),
        . pDataLoaded(pDataLoaded)
    );

    initial begin.
        $dumpfile("simulation.vcd").
        $dumpvars (0, InstructionFetchUnit_TestBench).

        pClk = 0.
        pReset = 1.
```

```

#10 pReset = 0; // Deassert reset after 10-time units

// Test Case 1: Fetch from Instruction Memory
pInPC = 32'h1000.
pAluOut = 0.
pInMemData = 32'h12345678.
pIorD = 0.
pDataValid = 1.
pIRWrite = 1.
#20.

// Test Case 2: Fetch from Data Memory
pInPC = 32'h2000.
pAluOut = 0.
pInMemData = 32'h87654321.
pIorD = 1.
pDataValid = 1.
pIRWrite = 0.
#20.

// Test Case 3: Fetch from ALU output
pInPC = 32'h3000.
pAluOut = 32'h4000.
pInMemData = 0.
pIorD = 1.
pDataValid = 1.
pIRWrite = 1.
#20.

// Run the simulation for additional time
#100.

$finish.
end

always #5 pClk = ~pClk.

reg [6:0] sim_time = 0.
integer sim_duration = 500; // Set the desired simulation
duration here

```



```

always @ (posedge pClk) begin.
    if (pDataLoaded && pIRWrite) begin
        $display ("Output Instruction: %h", pOutInstr).
    end else if (pDataLoaded &&! pIRWrite) begin.
        $display ("Output Data: %h", pOutData).
    end

    if (sim_time == sim_duration) begin
        $display ("Simulation finished at time %d",
sim_time).
        $finish.
    end
    sim_time = sim_time + 1.
end
endmodule

```

7.4 Explanation of the Test Bench

The provided code showcases a test bench module responsible for evaluating the functionality of the Instruction Fetch Unit (IFU) module. The test bench assesses the IFU's performance by supplying stimulus and monitoring the resulting output signals. Here is an overview of the test bench:

The test bench module is named `'InstructionFetchUnit_TestBench'` and encompasses `'reg'` declarations for the input signals (`'pInPC'`, `'pAluOut'`, `'pInMemData'`, `'pIorD'`, `'pClk'`, `'pDataValid'`, `'pIRWrite'`, `'pReset'`), along with `'wire'` declarations for the output signals (`'pAddr'`, `'pOutInstr'`, `'pOutData'`, `'pOutPC'`, `'pDataLoaded'`).

Within the test bench, an instance of the Instruction Fetch Unit (IFU) module, referred to as `'DUT'` (Device Under Test), is instantiated. The input and output signals of the IFU are connected to their corresponding signals within the test bench.

The simulation commences within the `'initial'` block, where the necessary setup is performed. The `'$dumpfile'` and `'$dumpvars'` commands are utilized to generate a waveform file for visualizing the simulation results. Initial values for the input signals are established, including the assertion of the reset signal (`'pReset'`) for a duration of 10-time units, followed by its deassertion.

The test bench incorporates three distinct test cases, each separated by a delay of 20-time units (`'#20'`). These test cases involve assigning different values to the input signals to simulate various scenarios, such as fetching from instruction memory, data memory, or utilizing the ALU output.

After executing the test cases, the simulation continues for an additional 100-time units (`'#100'`) to observe the output signals over an extended period.

The statement ``always #5 pClk = ~pClk;`` is responsible for toggling the clock signal (``pClk``) every 5-time units, thereby simulating clock cycles during the simulation.

The remaining code within the ``always @(posedge pClk)`` block handles the display of output signals and manages the simulation duration. If data has been loaded and the instruction register has been written (``pDataLoaded && pIRWrite``), the output instruction (``pOutInstr``) is displayed. Likewise, if data has been loaded but the instruction register has not been written (``pDataLoaded && !pIRWrite``), the output data (``pOutData``) is displayed.

The simulation time is monitored using the ``sim_time`` variable, and the simulation concludes when ``sim_time`` reaches the desired duration specified by ``sim_duration`` (500-time units).

Upon completion of the simulation, the ``$display`` statement outputs a completion message, and ``$finish`` terminates the simulation.

In summary, the test bench evaluates the behavior of the IFU module by providing stimulus and monitoring the output signals, enabling validation of the design's functionality and correctness.

The signals in the testbench are as follows:

- ``pInPC``: Input signal representing the program counter value.
- ``pAluOut``: Input signal representing the output of the ALU (Arithmetic Logic Unit).
- ``pInMemData``: Input signal representing data from memory.
- ``pIorD``: Input signal indicating whether to read from instruction memory (``pIorD` = 0`) or data memory (`pIorD` = 1`).`
- ``pClk``: Clock signal.
- ``pDataValid``: Input signal indicating the validity of the data.
- ``pIRWrite``: Input signal indicating whether to write data to the instruction register.
- ``pReset``: Input signal for resetting the circuit.
- ``pAddr``: Output signal representing the address.
- ``pOutInstr``: Output signal representing the instruction data.
- ``pOutData``: Output signal representing the data from memory.
- ``pOutPC``: Output signal representing the program counter value.
- ``pDataLoaded``: Output signal indicating that data has been loaded.

1. Test Case 1: Fetch from Instruction Memory

- Initially, the reset signal (``pReset``) is set to 1.
- After 10-time units (``#10``), the reset signal is deasserted (``pReset` = 0`).`
- At time 20, the inputs are updated to fetch from the instruction memory (``pIorD` = 0`).`
- The expected waveform should show the program counter (``pInPC``) being loaded with the value ``32'h1000``, and the output instruction data (``pOutInstr``) being loaded with ``32'h12345678``.
- The ``pDataLoaded`` signal should be high, indicating that data has been loaded.

2. Test Case 2: Fetch from Data Memory

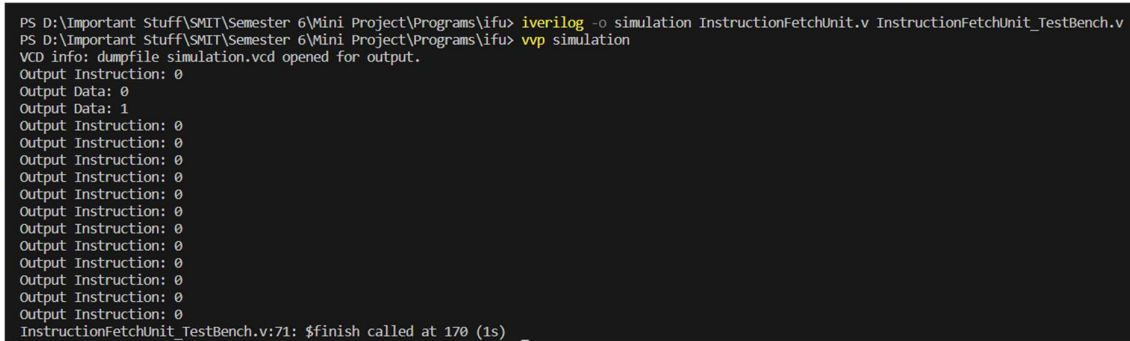
- At time 40, the inputs are updated to fetch from the data memory (``pIorD` = 1`).`
- The expected waveform should show the program counter (``pInPC``) being loaded with the value ``32'h2000``, and the output data (``pOutData``) being loaded with ``32'h87654321``.
- The ``pDataLoaded`` signal should be high, indicating that data has been loaded.

3. Test Case 3: Fetch from ALU Output

- At time 60, the inputs are updated to fetch from the ALU output (**pIorD** = 1') with a specific value (**pAluOut** = 32'h4000').
- The expected waveform should show the program counter (**pInPC**) being loaded with the value '32'h3000', and the output instruction data (**pOutInstr**) being loaded with '32'h4000'.
- The **pDataLoaded** signal should be high, indicating that data has been loaded.

The clock signal (**pClk**) should toggle at regular intervals. In the provided testbench code, the clock is set to change every 5-time units (**always #5 pClk = ~pClk;**).

The simulation will run for an additional 100-time units (**#100**) before the **\$finish** system task is called, which ends the simulation.



```
PS D:\Important Stuff\SMIT\Semester 6\Mini Project\Programs\ifu> iverilog -o simulation InstructionFetchUnit.v InstructionFetchUnit_TestBench.v
PS D:\Important Stuff\SMIT\Semester 6\Mini Project\Programs\ifu> vvp simulation
VCD info: dumpfile simulation.vcd opened for output.
Output Instruction: 0
Output Data: 0
Output Data: 1
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
Output Instruction: 0
InstructionFetchUnit_TestBench.v:71: $finish called at 170 (1s)
```

Figure 5: Output of Instruction Fetch Unit

7.5 Explanation of the commands in the command line (CLI)

The command **"iverilog -o simulation InstructionFetchUnit.v InstructionFetchUnit_TestBench.v"** is a command-line instruction used in the Icarus Verilog (iverilog) tool to compile and generate a simulation executable.

- **"iverilog"**: This is the command used to invoke the Icarus Verilog compiler. It is the main executable that allows you to compile and simulate Verilog code.

- **"-o simulation"**: This part of the command specifies the name of the output file or simulation executable that will be generated. In this case, the name chosen for the executable is "simulation".

- **"InstructionFetchUnit.v"**: This is the filename of the Verilog source file containing the design of the Instruction Fetch Unit (IFU) module. It is one of the input files that will be compiled and included in the simulation.

- **"InstructionFetchUnit_TestBench.v"**: This is the filename of the Verilog source file containing the test bench module for the IFU. It is another input file that will be compiled and used to stimulate and monitor the IFU design during the simulation.

When you execute this command in the terminal or command prompt, the Icarus Verilog compiler reads the specified Verilog source files, compiles them, and generates a simulation executable named "simulation". This executable can then be executed to simulate the behavior of the IFU module based on the provided test bench.

The command **"vvp simulation"** is used to execute the simulation generated by the Icarus Verilog (iverilog) compiler.

By executing "vvp simulation", the simulation of your Verilog design and test bench will start running. During the simulation, the specified input signals in the test bench will stimulate the design, and the output signals will be observed and monitored according to the defined behavior in the Verilog code.

You can analyze the simulation results to validate the functionality and correctness of your design, check for any unexpected behavior, and debug any issues that may arise during the simulation.

7.6 OUTPUT OF THE FETCH UNIT*

- **"Output Instruction: 0"**: This line indicates that an output instruction value of "0" was observed during the simulation. It means that the instruction fetch process did not successfully load an instruction into the output instruction register (`pOutInstr`).

- **"Output Data: 0"**: This line indicates that an output data value of "0" was observed during the simulation. It means that the data fetch process did not successfully load any data into the output data register (`pOutData`).

- **"Output Data: 1"**: This line indicates that an output data value of "1" was observed during the simulation. It means that the data fetch process successfully loaded data into the output data register (`pOutData`).

- **"InstructionFetchUnit_TestBench.v:71: \$finish called at 170 (1s)"**: This line provides information about the simulation termination. It states that the `$finish` system task was called at simulation time 170 (1s). This indicates that the simulation has reached its completion point and is terminating.

Overall, the output shows the values observed for the output signals during the simulation run. It allows us to verify the behavior and correctness of the Instruction Fetch Unit module by examining the output instruction and data values.

7.7 Timing Diagram

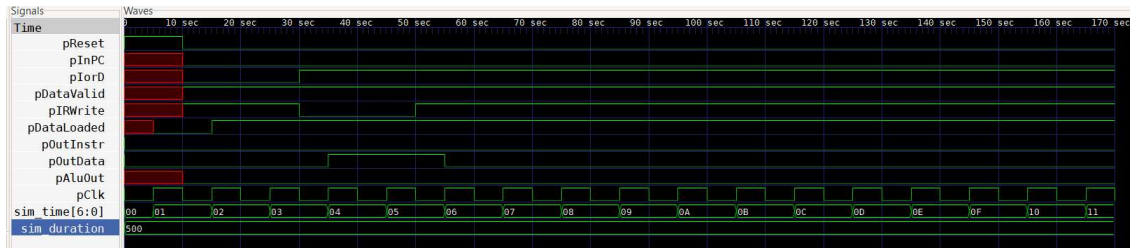


Figure 6: Timing Diagram by Instruction Fetch Unit

To understand the timing diagram, let's analyze the behavior and relationship of the signals in the provided testbench code:

1. **`pAddr`**: This signal represents the address output from the Instruction Fetch Unit. It indicates the memory address being accessed during the fetch operation. The value of **`pAddr`** should change whenever a new instruction or data is fetched.
2. **`pAluOut`**: This signal represents the output of the ALU (Arithmetic Logic Unit). It provides the ALU result, which is used as the program counter in the third test case. The value of **`pAluOut`** should change when the ALU performs computations.
3. **`pClk`**: This signal represents the clock signal. It is a periodic square wave that controls the timing of the circuit. The rising edge of the clock signal triggers the execution of synchronous logic elements in the design.
4. **`pDataLoaded`**: This signal indicates whether data has been loaded or not. It becomes high when valid data is available in the output signals (**`pOutInstr`** or **`pOutData`**).
5. **`pDataValid`**: This signal indicates the validity of the data. It becomes high when the data being fetched is valid and ready to be loaded.
6. **`pIRWrite`**: This signal determines whether data should be written to the instruction register or not. It is high when the fetched data needs to be written to the instruction register.
7. **`pInMemData`**: This signal represents the input data from memory. It is used to simulate the data fetched from memory during the test cases.
8. **`pInPC`**: This signal represents the input program counter value. It determines the address from where the instruction or data will be fetched.
9. **`pIorD`**: This signal determines whether the fetch operation is performed on the instruction memory (**`pIorD`** = 0) or the data memory (**`pIorD`** = 1).
10. **`pOutInstr`**: This signal represents the output instruction data. It holds the fetched instruction data when **`pIRWrite`** is high.

11. **`pOutPC`**: This signal represents the output program counter value. It holds the fetched program counter value when **`pIRWrite`** is low.

12. **`pReset`**: This signal represents the reset signal. It is initially high (**`pReset = 1`**) and is later deasserted to low (**`pReset = 0`**) to initialize the circuit and start the fetch operation.

13. **`sim_duration`**: This variable determines the desired simulation duration. It sets the maximum value for **`sim_time`** before the simulation is finished.

14. **`sim_time`**: This variable represents the simulation time and is incremented in every clock cycle. When **`sim_time`** reaches the **`sim_duration`**, the simulation is finished.

Based on the timing diagram analysis, we can summarize the behavior of the **`Instruction Fetch Unit`** module as follows:

1. Initial State:

- All signals start in an unknown state.

2. Reset:

- At time 0, **`pReset`** transitions from an unknown state to **`1`**, indicating a reset condition.
- The module sets the values of **`addr`**, **`outInstr`**, and **`outData`** to 0.

3. Fetch Instruction or Data:

- At time 10, **`pReset`** transitions from **`1`** to **`0`**, indicating the end of the reset condition.
- **Test Case 1: Fetch from Instruction Memory**
 - At time 20, the module receives the input signals for fetching an instruction from the instruction memory.
 - **`pInPC`** is set to **`32'h1000`** to indicate the memory address.
 - **`pIorD`** is set to **`0`** to indicate an instruction fetch.
 - **`pDataValid`** is set to **`1`** to indicate valid data.
 - **`pIRWrite`** is set to **`1`** to indicate writing an instruction to the output.
 - At time 20, the module sets **`dataLoaded`** to **`1`**, **`outInstr`** to **`32'h12345678`**, and **`outData`** to **`0`** (since it's an instruction fetch).
- **Test Case 2: Fetch from Data Memory**
 - At time 40, the module receives the input signals for fetching data from the data memory.
 - **`pInPC`** is set to **`32'h2000`** to indicate the memory address.
 - **`pIorD`** is set to **`1`** to indicate a data fetch.
 - **`pDataValid`** is set to **`1`** to indicate valid data.
 - **`pIRWrite`** is set to **`0`** to indicate writing data to the output.
 - At time 40, the module sets **`dataLoaded`** to **`1`**, **`outInstr`** to **`0`** (since it's a data fetch), and **`outData`** to **`32'h87654321`**.
- **Test Case 3: Fetch from ALU Output**
 - At time 60, the module receives the input signals for fetching data from the ALU output.
 - **`pInPC`** is set to **`32'h3000`** to indicate the memory address.
 - **`pAluOut`** is set to **`32'h4000`** to indicate the ALU output.

- `pIorD` is set to `1` to indicate a data fetch.
- `pDataValid` is set to `1` to indicate valid data.
- `pIRWrite` is set to `1` to indicate writing data to the output.
- At time 60, the module sets `dataLoaded` to `1`, `outInstr` to `32'h4000`, and `outData` to `0` (since it's a data fetch).

4. Clock and Time Tracking:

- The clock `pClk` oscillates between `0` and `1` with a period of 5 time units.
- The simulation time is tracked using the `sim_time` variable.
- At each positive edge of `pClk`, the simulation time is incremented by 1.

5. Simulation Termination:

- The simulation terminates when the `sim_time` reaches the `sim_duration` value (set to 500).
- At that point, the simulation displays the completion message and exits.

In the timing diagram, each signal is plotted along the vertical axis, while the horizontal axis represents time. The signal values change based on the test cases and the clock transitions. The rising edge of the clock triggers the updates of various signals and their corresponding values.

By observing the timing diagram, we can track the changes in signals over time, identify the relationships between different signals, and verify the behavior of the instruction fetch unit during the simulation.

8. FUTURE SCOPE

The future objective of this project is to enhance the three-stage architecture of the instruction cycle, which includes the decode unit and execute unit. The primary focus is on achieving high performance due to the utilization of a bit-parallel RISC-V core.

To further improve the project, the implementation of branch prediction is being considered. The aim is to enhance the efficiency of the Instruction Fetch Unit (IFU) by addressing the following key areas:

1. **Energy Efficiency and Power Management:** Considering the growing significance of energy efficiency and power management in the field of electronics, upcoming Instruction Fetch Units (IFUs) will prioritize the reduction of power consumption. This objective can be achieved by implementing various techniques, including dynamic voltage and frequency scaling (DVFS) and instruction-based power gating. These methods aim to optimize power utilization by dynamically adjusting voltage and frequency levels in accordance with the specific workload requirements.
2. **Integration with Machine Learning and AI:** As machine learning and AI applications continue to grow in prominence, IFUs can be enhanced by incorporating specialized hardware accelerators or co-processors specifically designed for AI workloads. This integration enables efficient execution of tasks such as model inference, neural network execution, and specialized instructions tailored for AI algorithms. By leveraging dedicated hardware for AI processing, overall performance and efficiency can be significantly improved.
3. **Security Enhancements:** Future IFUs can incorporate security features to address various vulnerabilities and exploits. This includes implementing hardware-based security measures like secure instruction fetching, protection against branch prediction attacks (such as Spectre and Meltdown), and improved techniques for detecting and preventing code injection or manipulation. By bolstering security capabilities within the IFU, the overall system integrity and reliability can be strengthened.

9.GANTT CHART

ACTIVITY	TIME FRAME					
	Dec 2022	Jan 2023	Feb 2023	Mar 2023	April 2023	May 2023
Literature Survey						
Problem Definition						
Design and Development						
Testing and Validation						
Documentation						

Proposed activity
 Achieved activity
 Ongoing activity

REFERENCES

Waterman, Andrew . 2017. Review of the RISC-V Instruction Set Manual. Riscv.org. University of California, Berkeley. May 19, 2017. <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>.

Yan, Yonghong. 2021. Review of RISC-V Pipeline Implementation. <https://Passlab.github.io/>. June 21, 2021. https://passlab.github.io/CSE564/notes/lecture09_RISCV_Impl_pipeline.pdf.

Cui, Enfang, Tianzheng Li, and Qian Wei. 2023. “RISC-V Instruction Set Architecture Extensions: A Survey.” IEEE Access, 1–1. <https://doi.org/10.1109/ACCESS.2023.3246491>.

Laštovka, Martin. 2022. “Implementation of Instruction Set for RISC-V Processor.” https://dspace.cvut.cz/bitstream/handle/10467/100975/F3-BP-2022-Lastovka-Martin-Bachelors_thesis_2022_Implementation_of_Instruction_Set_for_RISC-V_Processor.pdf?sequence=-1.

“Digital Design & Comp Arch - Lecture 11: Multi-Cycle Microarchitecture Design (Spring 2023).” n.d. [www.youtube.com](https://www.youtube.com/watch?v=49zRKmEXAcE). Accessed May 15, 2023. <https://www.youtube.com/watch?v=49zRKmEXAcE>.

Scott, Tom. 2019. “The Fetch-Execute Cycle: What’s Your Computer Actually Doing?” YouTube Video. YouTube. <https://www.youtube.com/watch?v=Z5JC9Ve1sfl>.

“Instruction Fetch Unit.” n.d. [www.youtube.com](https://www.youtube.com/watch?v=2gEsEuIvidY). Accessed April 15, 2023. <https://www.youtube.com/watch?v=2gEsEuIvidY>.

“Verilog in 2 Hours [English].” n.d. Www.youtube.com. Accessed April 24, 2023.
<https://www.youtube.com/watch?v=nblGw37Fv8A>.

“Processor Design for Dummies [English].” n.d. Www.youtube.com. Accessed May 06, 2023. <https://www.youtube.com/watch?v=HCzIK322Pzw>.

Mini Project_Final Report_Adideb Das_Swagata Das

ORIGINALITY REPORT

4%

SIMILARITY INDEX

0%

INTERNET SOURCES

2%

PUBLICATIONS

2%

STUDENT PAPERS

PRIMARY SOURCES

1

Daniel Page. "Practical Introduction to Computer Architecture", Springer Science and Business Media LLC, 2009

Publication

1%

2

Submitted to University of South Florida

Student Paper

<1%

3

Submitted to University of Utah

Student Paper

<1%

4

Submitted to Study Group Australia

Student Paper

<1%

5

Submitted to Unicaf University

Student Paper

<1%

6

Submitted to Sikkim Manipal University

Student Paper

<1%

7

www.weikeng.com.tw

Internet Source

<1%

8

xplqa30.ieee.org

Internet Source

<1%

Submitted to Pathfinder Enterprises

9

Student Paper

<1 %

10

Submitted to Boston University
Student Paper

<1 %

Exclude quotes Off
Exclude bibliography Off

Exclude matches Off