32 Bit Single Cycle RISC-V Processor using Verilog

1st Devanshi Jariwala
21bec050@nirmauni.ac.in
Institute Of Technology,Nirma University,
Ahmedabad

2nd Riya Jha
21bec051@nirmauni.ac.in
Institute Of Technology,Nirma University,
Ahmedabad

***Abstract*—Processors are an integral part of the computer and electronics industry. Every computational unit contains some sort of processing circuit, designed to perform multiple operations on a single device and can be categorized based on its speed, flexibility and adaptability.This paper presents a methodology for designing a single clock cycle MIPS RISC Processor using Verilog HDL.Verilog HDL facilitates the description, verification, simulation, and hardware implementation of the processor.The RISC processor includes 32-bit general-purpose registers and operates with a memory word size of 32 bits. The processor is structured into five stages: instruction fetch, instruction decode, execution, data memory, and write back, each controlled by a control unit. VHDL is chosen for its ability to handle concurrency, aligning with the parallel nature of digital hardware. All modules in the design are coded in VHDL, and a top-level module integrates these stages.**

***Index Terms*—single cycle, RISC V, datapath, verilog**

I. Introduction

RISC-V is an open-source instruction set architecture (ISA) based on the principles of Reduced Instruction Set Computing (RISC). It provides a framework for designing processors and computer systems, allowing for customization and innovation. RISC-V is characterized by its simplicity, modularity, and extensibility, making it suitable for a wide range of applications from embedded systems to supercomputers. One of its key features is its modular design, which allows users to choose and implement only the instructions and features they need for a particular application. Additionally, RISC-V is designed to be scalable, supporting both 32-bit and 64-bit address spaces.

In the field of computer architecture, the design of a processor's datapath is a critical factor that influences its performance, efficiency, and complexity. Within this set of design techniques, the single-cycle datapath stands out as a paradigm distinguished by its simplicity and deterministic execution model.

Fundamentally, the single-cycle datapath works on the premise of executing each instruction completely inside a single clock cycle. This means meticulously completing all necessary tasks—ranging from instruction fetch to write-back—within the time constraints of a single clock pulse.

This design approach is centered on the goal of simplicity and predictability. The single-cycle datapath streamlines data handling within the CPU by following a defined sequence of phases for each instruction. This predictability extends to the processor's temporal behavior, making it deterministic and susceptible to rigorous analysis—a feature widely valued in applications with strict timing limitations, such as real-time systems.

The single-cycle datapath's architectural elegance is enhanced by its reduced control logic. Control signals, which are critical in orchestrating the flow of data and instructions within the CPU, are obtained from a finite state machine using the opcode of the current instruction. This simple control mechanism allows for easy installation and testing, making it an appealing option for both educational and professional design projects.

Furthermore, the single-cycle datapath is an effective teaching tool, providing students and practitioners with a clear perspective from which to understand the fundamental principles of computer architecture. Its sequential execution paradigm and deterministic timing characteristics provide fertile ground for understanding the complexities of instruction execution, memory access, and control flow, demonstrating its continued significance in the field of computer science education.

For building a single-cycle datapath, RISC (Reduced Instruction Set Computing) architecture offers advantages over MIPS architecture. RISC simplifies things by using a smaller set of straightforward instructions, making it easier to execute tasks within a single clock cycle. Its instruction format is consistent and predictable, which streamlines the design process for the datapath and control logic. Additionally, RISC instructions are designed to be executed sequentially, which helps avoid complications and delays in single-cycle implementations. Moreover, RISC architecture benefits from compiler optimizations, allowing programs to efficiently schedule instructions for faster execution. Overall, RISC's simplicity, consistency, and optimization make it a preferred choice for single-cycle datapaths compared to MIPS architecture.

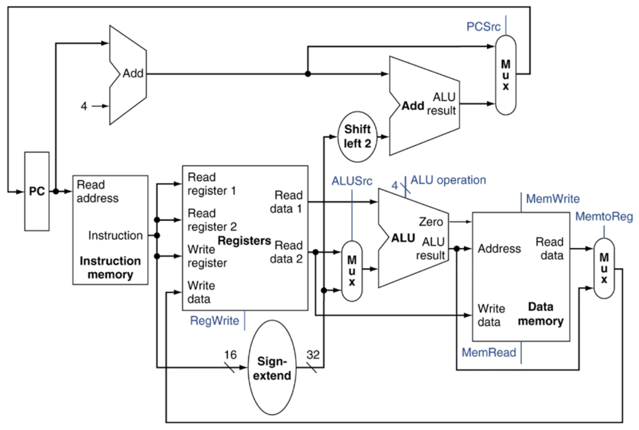


Fig. 1. Single cycle datapath

II. Stages of Datapath

The execution of instructions in a processor follows a structured pattern known as the instruction execution cycle as shown in Fig 2, which often consists of multiple stages. In the context of a single cycle processor datapath, this cycle is divided into five major stages, each of which plays an important part in the overall execution process. The stages are: Instruction Fetch (IF), Instruction Decode (ID), Execution, Memory Access (MEM), and Write Back (WB). In this explanation, we will delve into the complexities of each stage, outlining its function and relevance within the larger framework of instruction execution.

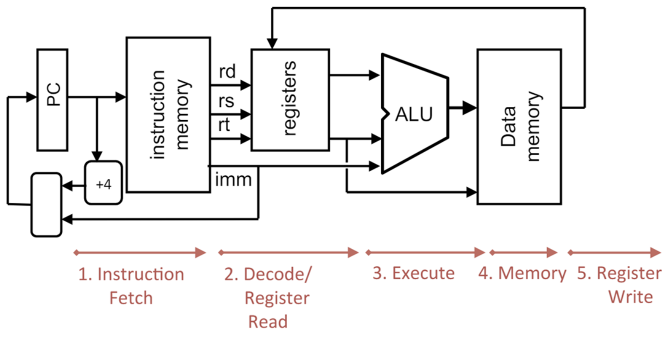


Fig. 2. Stages of datapath

A. *Instruction Fetch Stage*

The first stage of RISC is the instruction fetch (IF) stage. The IF step retrieves the needed instructions from memory. The operation of the IF stage begins when the program counter (PC) sends a 32-bit register to fetch instructions from memory into the instruction register (IR) and the computer is incremented by an adder to process the next sequential command. The IR is used to hold the necessary instructions in subsequent clock cycles. The block diagram of the IF stage is shown in the figure. 1.

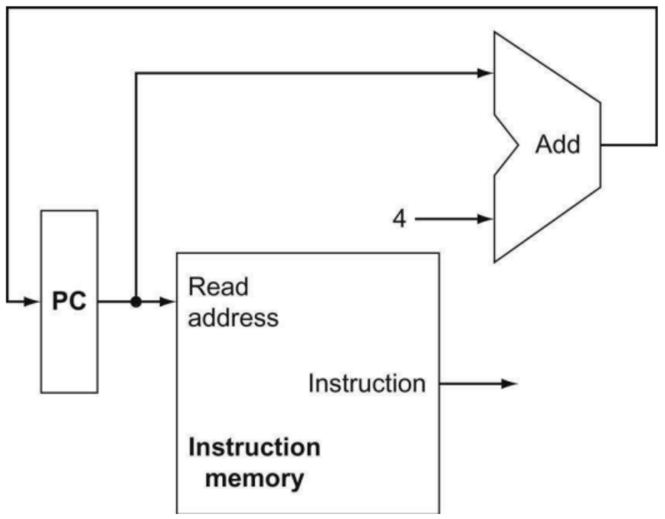


Fig. 3. Instruction Fetch Unit

B. *Instruction decode stage*

The control unit receives the instruction's opcode when it is fetched from the IF stage, and the ALU control unit receives the function code. The two-port register file is addressed using the register address fields in the instruction. In a single clock cycle, The processor's 32 general-purpose registers are stored

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

Fig. 4. RISC V base Instruction Formats

in a structure called a register file. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer. In addition, we will need an ALU to operate on the values read from the registers. R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge. The figure below shows the result; we need a total of three inputs (two for register numbers and one for data) and two outputs (both for data).

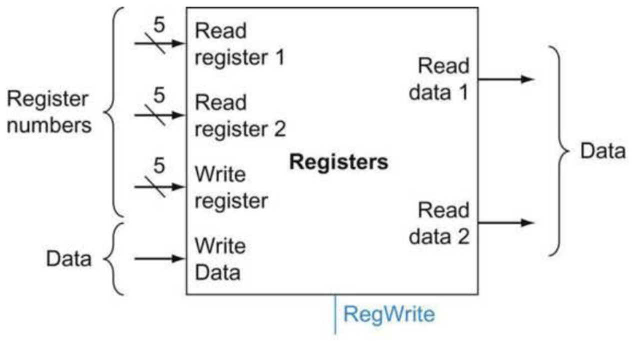


Fig. 5. Register Block

C. *Execution Unit*

The part in a single-cycle datapath that handles arithmetic and logical operations is commonly referred to as the execution unit. According to the processor's instruction set architecture (ISA), this unit carries out operations such as addition, subtraction, AND, OR, and so forth. Every instruction in a

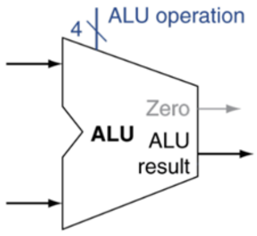
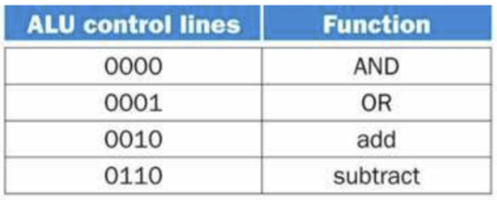


Fig. 6. ALU

single-cycle architecture executes in precisely one clock cycle. Consequently, the execution unit has one clock cycle to finish its task. This frequently means that the execution unit must be quick and easy to use in order to finish the intended task in the allocated amount of time.

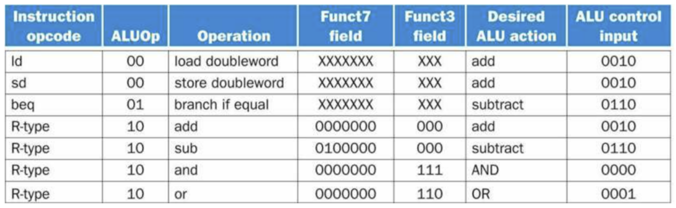
Depending on the instruction class, the ALU will need to perform one of these four functions. For load and store instructions, we use the ALU to compute the memory address by addition. For the Rtype instructions, the ALU needs to perform one of the four actions (AND, OR, add, or subtract), depending on the value of the 7-bit funct7 field (bits 31:25) and 3-bit funct3 field (bits 14:12) in the instruction . For the conditional branch if equal instruction, the ALU subtracts two operands and tests to see if the result is 0. We can generate the 4-bit ALU control input using a small control unit that has as inputs the funct7 and funct3 fields of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract and test if zero (01) for beq, or be determined by the operation encoded in the funct7 and funct3 fields (10). The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations shown previously. In Figure , we show how to set the ALU control inputs based on the 2-bit ALUOp control, funct7, and funct3 fields.



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Fig. 7. Truth Table for ALU control

This truth table shows how the 4-bit ALU control is set depending on these input fields. Since the full truth table is very large, and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value. Throughout this chapter, we will use this practice of showing only the truth table entries for outputs that must be asserted and not showing those that are all deasserted or don't care.



Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Fig. 8. Truth Table for ALU

D. *Data Memory Stage*

In the data memory step, values are loaded and stored in memory. This phase is in effect when the branch is being instructed. One output is used for reading from data memory, while two inputs are used for writing and addressing the data. Any particular clock has either MemWrite or MemRead asserted as two distinct read and write controls.

The read memory control loads an instruction.

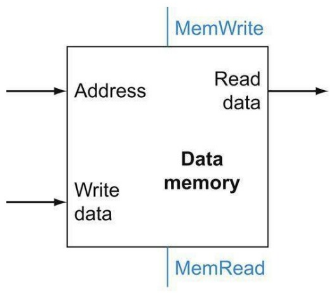


Fig. 9. Data Memory Unit

1.$MemoryRead$:

It is claimed that MemRead makes it possible to read data from the data memory. One of the inputs provides an address to the data memory. After that, the data is delivered from the data memory to the register file via its read data output. To facilitate error testing and simulation, the memory is initialised to a certain value while creating the VHDL codes for the Data Memory stage.

2.$MemoryWrite$:

An instruction can be stored via the write memory control It is claimed that MemWrite makes it possible for data to be written into data memory. One of the inputs provides an address to the data memory.

The input Write Data is used to send the data to be written into the data memory. In VHDL coding, the writing operation into the memory starts when the write control signal of the relevant memory is asserted. To accomplish this, place an AND gate between the address that identifies the memory to be written and the memWrite control signal. After that, the information on the write data bus is written to the relevant memory address.

E. Write Back stage

Unlike pipelined systems, which have a unique write-back stage, single-cycle datapath designs execute each instruction in a single clock cycle. Rather, depending on the particular architecture, the write-back process is usually integrated with other phases, such the execution stage or the memory stage.
Execution Stage: The outcome of the operation may be directly written back to the target register in the register file during the execution stage itself in certain designs, especially those where the stage entails arithmetic or logical operations.
Memory Stage: The result of an operation that retrieves data from memory, like a load instruction, may be written back to the register file's destination register at this point.
combination: The write-back procedure may comprise a mixture of phases in more intricate designs or instructions. For instance, the outcome of an ALU operation could be calculated during the execution phase and subsequently recorded back to the register file during the memory phase that follows.

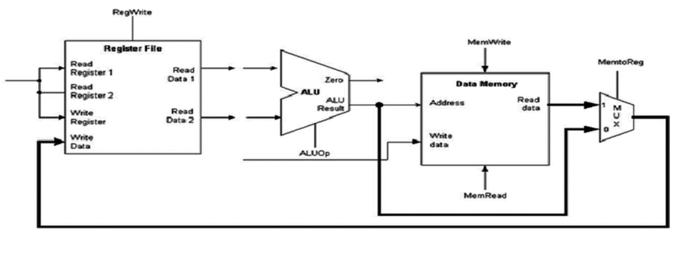


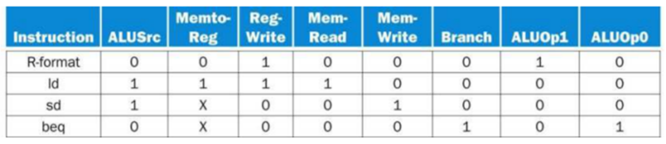
Fig. 10. Write Back Stage

The write-back operation is essentially integrated into the entire instruction execution process in a single-cycle datapath; it may happen in one or more stages, contingent upon the particular requirements of each instruction. The objective is to guarantee that, prior to the next instruction starting execution in the ensuing clock cycle, the result of each instruction is correctly recorded back to the register file.

F. Control unit

Control signals are essential in a single-cycle datapath design because they help the processor's multiple parts work together to execute instructions in a single clock cycle. The ALU, register file, multiplexers, and memory are examples of datapath elements whose actions are determined by these control signals.

1) Opcode Decoder Outputs: These control signals are produced in accordance with the operation code, or opcode, of the command that is now being carried out. They choose which particular operation—such as addition, subtraction, logical AND, logical OR, etc.—will be carried out by the ALU.
2) ALU Control Signals: These signals manage how the ALU functions. They define the kind of logical or arithmetic operation that needs to be carried out, including AND, OR, subtraction, addition, and so forth.
3) Read Register Signals: These signals specify which registers need to be read from the register file. They select the source operands for the instruction.
4) Write Register Signal: This signal specifies which register in the register file will receive the result of the instruction execution.
5) Memory Read/Write Signals: If the instruction involves memory access (e.g., load or store), these signals control the data memory unit. They indicate whether a memory read or write operation is required.
6) Memory Address Signals: These signals specify the memory address for memory access instructions. They determine which location in memory should be accessed.
7) Branch Control Signals: These signals control the behavior of the program counter (PC) and determine whether a branch should be taken based on the result of a comparison operation.
8) Jump Control Signals: If the instruction involves a jump or branch instruction, these signals control the update of the program counter to the target address specified in the instruction.



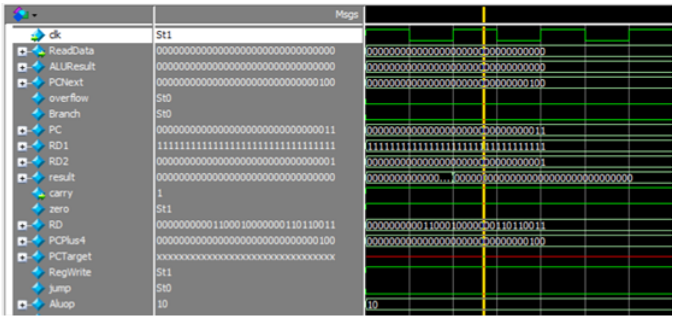
Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

The control unit generates these control signals in response to the command that is being carried out. In a single-cycle datapath design, they guarantee that the datapath components function correctly and in synchrony to execute the instruction inside a single clock cycle.

The control function for the simple single-cycle implementation is completely specified by this truth table.

III. Simulations And Results

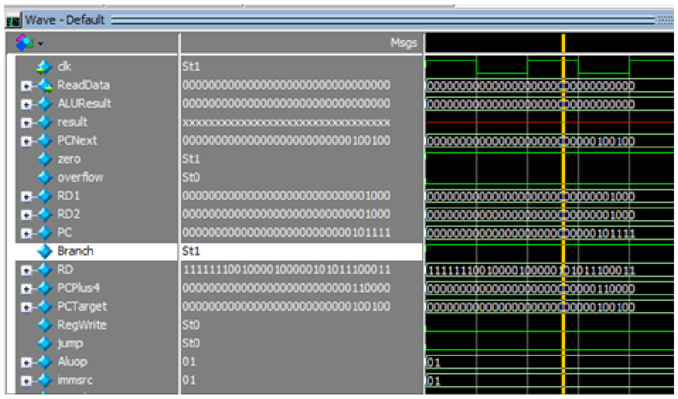
The Verilog code implementing the single-cycle datapath was synthesized and produced with the Quartus Prime software suite, which interfaces easily with ModelSim Altera Edition for simulation.



```
                Msgs
clk             St1
ReadData        00000000000000000000000000000000
ALUResult       00000000000000000000000000000000
PCNext          00000000000000000000000000000100
overflow        St0
Branch          St0
PC              00000000000000000000000000000011
RD1             11111111111111111111111111111111
RD2             00000000000000000000000000000001
result          00000000000000000000000000000000
carry           1
zero            St1
RD              00000000001100010000000110110011
PCPlus4         00000000000000000000000000000100
PCTarget        xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
RegWrite        St1
jump            St0
Aluop           10
```

Fig. 11. Simulation Result of addition operation

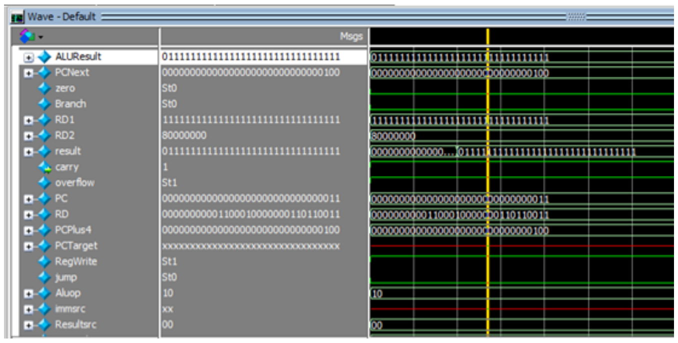
The initial simulation focused on examining the addition operation in the single-cycle datapath configuration. The simulation scenario involves adding two values stored in registers: one register with all 1s for 32 bits and another register with a value of one. This setup was chosen to test the behavior of the carry and zero flags produced during the addition procedure. After running the simulation, the waveform analysis validated the expected behavior of the single-cycle datapath during the addition operation. Specifically, the carry flag indicated the presence of a carry during addition, whereas the zero flag accurately detected circumstances when the addition resulted in zero. The second simulation focused on determining the behavior of the branch equal (BEQ) instruction in the single-cycle datapath configuration. The BEQ instruction compares two register values and branches to the target address if they are equal. The purpose of this simulation scenario was to ensure that the BEQ instruction and its related control signals functioned properly within the datapath.



```
Wave - Default
                Msgs
clk             St1
ReadData        00000000000000000000000000000000
ALUResult       00000000000000000000000000000000
result          xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
PCNext          00000000000000000000000000100100
zero            St1
overflow        St0
RD1             00000000000000000000000000001000
RD2             00000000000000000000000000001000
PC              00000000000000000000000000101111
Branch          St1
RD              11111110010000100000101011100011
PCPlus4         00000000000000000000000000110000
PCTarget        00000000000000000000000000100100
RegWrite        St0
jump            St0
Aluop           01
immsrc          01
```

Fig. 12. Simulation result for Beq

This simulation scenario required adding two register values: one with all 1s for 32 bits and another with 1 on the most significant bit (MSB) and 0s on all other bits. The objective was to assess the behavior of the overflow flag during an addition operation within the single-cycle datapath design.



```
Wave - Default
                Msgs
ALUResult       01111111111111111111111111111111
PCNext          00000000000000000000000000000100
zero            St0
Branch          St0
RD1             11111111111111111111111111111111
RD2             80000000
result          01111111111111111111111111111111
carry           1
overflow        St1
PC              00000000000000000000000000000011
RD              00000000001100010000000110110011
PCPlus4         00000000000000000000000000000100
PCTarget        xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
RegWrite        St1
jump            St0
Aluop           10
immsrc          xx
Resultsrc       00
```

Fig. 13. Simulation Result Of overflow

IV. Drawbacks of single cycle datapath

While the single-cycle design is functional, it is too inefficient to be applied in contemporary designs. Observe that in this single-cycle design, the clock cycle must have the same length for each instruction to understand why this is the case. The clock cycle is, of course, determined by the processor's longest path. This path is probably a load instruction, which makes use of the register file, data memory, instruction memory, ALU, and register file as five functional units in succession. A single-cycle implementation's overall performance is probably going to be subpar even though the CPI is 1. This is because the clock cycle is too long. Although there is a significant cost associated with adopting the single-cycle design with a fixed clock cycle, for this small instruction set it might be deemed acceptable. Historically, this implementation style was used by early computers with very basic instruction sets. However, this single-cycle design would not function at all if we attempted to construct the floating-point unit or an instruction set with more sophisticated instructions. It is pointless to explore implementation strategies that decrease common case delay but do not enhance worst-case cycle duration, as we have to believe that the clock cycle represents the worst-case delay for every instruction.

V. Conclusion

Exploring the five stages of a processor's instruction execution cycle reveals a carefully choreographed sequence critical for efficient instruction processing. This analysis yields numerous major conclusions, showing both the strengths and potential areas for improvement within current design paradigms.

For starters, the structured form of the instruction execution cycle ensures that instructions are processed in a systematic order, from retrieval to execution and, finally, result dissemination. Each stage, from Instruction Fetch to Write Back, serves a specific purpose, adding to the overall efficiency and efficacy of the processor's work.

Additionally, the division of labor among the stages allows for parallelism and pipelining, which improves throughput and overall speed. Processors can maximize available resources and reduce latency by processing several instructions concurrently at different phases of the execution cycle, resulting in increased computing throughput.

Despite the inherent virtues of the current design method, some restrictions remain. The most significant of these is the problem of balancing resource utilization with time restrictions. As processors grow to meet rising complexity and performance demands, maintaining efficient resource allocation while adhering to strict timing requirements remains a key engineering problem.

REFERENCES

[1] I.D.Sulik, M. Vasilko, D. Dmckova, P. .Fuchs, "Design ofa RISC Muocontroller Core in 48 Hours", hnp:l/ulxw.itpapers.com/cgilPSu"aryIT.pl ?papd=12747scid=108

[2] Charles E. Gimarc, Veljko M. Mhtinovic, "RISC Principles, Architecture, and Design", Computer Science Press Inc., 1989.

[3] Kalahari Journals,"Single Cycle 32-bit RISC-V ISA Implementation and Verification"

[4] robotics.shanghaitech.edu.cn/courses/ca/20s/disc/Discussion7.pdf

[5] J. S. Chen, B. Mulgrew, and P. M. Gran¡ "A clustering technique for digital communications channel equalization using radial basis function networks," EEE Trans. Neuml Nerworkr, vol. 4