Integrated Assembler-Processor Instruction Execution

1st Hetansh Shah
Institute of Technology
Nirma University
Ahmedabad, Gujarat
21bec045@nirmauni.ac.in

2nd Himangi Agrawal
Institute of Technology
Nirma University
Ahmedabad, Gujarat
21bec046@nirmauni.ac.in

***Abstract*—The implementation of a completely synthesized 32-bit processor built within the freely available RISC-V (RV32I) ISA. is described in this paper. With a single clock cycle to complete each instruction, the single-cycle processor architecture provides an easy way to execute instructions. This design makes it easier to understand how the processor operates and simplifies the control logic. The fetch, decode, execute, and access memory instructions, and write-back stages of the RISC-V single-cycle processor are among the main elements covered in this paper. To ensure the compliance with the RISC-V ISA semantics and instruction formats, the Verilog implementation covers the datapath components, control unit, register file, ALU (Arithmetic Logic Unit), and memory interface. The assembler is implemented in python. It generates the machine codes for the required assembly code. The assembly instructions are read from the external file and stored in the instruction memory in Verilog. The processor simulation is conducted in ModelSim. Section II describes the RISC-V instruction set followed by Section III which covers the architecture and implementation. Section IV briefs about the assembler and results obtained from the assembler. Section V includes the simulation results in ModelSim, followed by the conclusion.**

***Index Terms*—RISC-V, single cycle, RV321 ISA, Verilog, datapath, Modelsim, assembler.**

I. Introduction

With its openness, flexibility, and scalability, the RISC-V (Reduced Instruction Set Computing - V) instruction set architecture has become a key standard in the field of modern computer architecture. Of all the ways that RISC-V processors can be implemented, the single-cycle design is a fascinating one because of how straightforward and effective it is. RISC-V has been created to provide support 32-bit, 64-bit, and 128-bit address spaces in order to ensure its success and widespread adoption. The ISA is divided into a modest base number ISA and a large base integer ISA, which are both minimal sets of instructions sufficient to give operating systems, linkers, compilers, and assemblers a reasonable target [1]. The core ideas of RISC architecture are embodied in the RISC-V single-cycle processor, which places an emphasis on uniformity, simplicity, and efficient instruction execution. The single-cycle processor seeks to complete each instruction in a single clock cycle, in contrast to its multi-cycle counterparts, where instructionscould require multiple clock cycles to complete. Reduced Instruction Set Computer (RISC) based ARM architectures have already taken over mobile platforms like Apple and Android, while MIPS based architectures are the foundation of the majority of game consoles. A new open ISA called RISC-V is built on the RISC architecture and is intended to aid in research and teaching [2]. Modern gadgets like smartphones, cloud computers, and tiny embedded systems are made for RISC-V [3]. This work presents a hardware design architecture for the 32-bit address space, called the RV32I base integer instruction set. The implementation is an in-order, single-core, single-cycle architecture that is not bus-based and fully supports the RV32I base integer instruction set. Acoustic signal processing, real-time embedded systems, sensor technology, and numerous other domains are among the application domains. The processor is implemented in Verilog HDL. The project's main goal is to create a fully functional computer system, utilizing Verilog and an assembler to convert high-level source code into processor-executable instructions. This integration provides the framework for building a useful computing environment that can effectively carry out programs. The assembler converts high-level commands into binary machine language that the processor can understand, acting as a link between human-readable source code and machine-executable instructions. Its job is to parse, analyze, and translate the assembly code into a series of operands and opcodes that are specific to the architecture of the processor. The machine code is produced by the assembler and then fed into the Verilog implementation. A hardware description language called Verilog makes it easier to design and simulate digital circuits, such as processors. Here, Verilog acts as a specification for the logical architecture and behavior of the processor, acting as a blueprint for its hardware implementation. It builds the processor's registers, arithmetic logic units, and control units using the machine language output from the assembler and the opcodes that are produced. Then, using the predefined instruction set architecture, the synthesized processor runs instructions based on the supplied machine code. This smooth transition between Verilog-defined processor hardware and assembler-generated machine code makes it possible to realize a functional computing system that can accurately and efficiently execute a variety of programs.

II. RISC-V INSTRUCTION SET

The RISC-V ISA is an open standard instruction set architecture. It outlines a processor's architecture, including the registers, memory model, privilege levels, instruction set, and other architectural components. Because of its modular and extensible design, it is possible to implement the RISC-V ISA in a variety of ways to meet different needs. 32 registers, indexed 0 to 31, make up the RISC-V register file. Of these, 31 are GPRs (general purpose registers) (index 1 to 31).By default, register which is at index 0 is fix at 0.Since RISC-V does not specify a calling convention for software, this work adhered to the MIPS calling convention . In RISC-V, another user-visible register is the program counter [2]. The RV32I base integer instruction set has four primary instruction formats: R, I, S, and U. Five distinct immediate types (I/S/B/U/J) make up the immediate instruction type. The leftmost portion of the instruction is used for sign extension. Based on how immediate values are handled, the instruction formats are available in two additional variations (SB/UJ). RISC-V (RV32I) instructions follow a fixed-length format. The Instruction format is shown in Fig 1.

The opcode field is the initial field in the R-type instruction format. The purpose of this 7-bit (0–6) opcode is to indicate the kind of instruction format being used, such as R-type, I-type, U-type, etc. The R-type instruction format's next field is known as rd field. The Rd refers to the destination register. The location where the operation's result is stored is indicated by this rd field. The rd field is five bits (7–11) long. Following the destination register is the field funct3 (function - 3). This field, which spans bits 12 to 14, has a length of three bits. In addition, it offers more details about the operation, such as whether addition, subtraction, or a logical operation is being performed. Two source registers, rs1 and rs2, each with a bit length of five, are present.The bit lengths of rs1 and rs2 are 5 bits, ranging from 15 to 19, and 20 to 24 respectively. The final field in the R-type instruction format is called funct7.It provides details about the operation, such as whether it's a multiplication or shift operation. Depending on the input format provided to them, the operations are described by either funct3 or funct7. I-type format is used to carry out instructions that use register and immediate value operations [5]. Instructions that store a value from a register into memory are implemented using this s-type format [5]. The branch instructions that conditionally transfer control to a newly specified instruction address are the type of instructions that use the b-type format [5]. For performing instructions that load a 20- bit immediate value into a register, use the u-type format [5]. The j-type format is used to implement jump type instructions that unconditionally transfer control to a new instruction address [5].

III. ARCHITECTURE AND IMPLEMENTATION

Every step of processor pipeline in a single-cycle implementation requires precisely one clock cycle, regardless of it's complexity. For embedded systems where dependability, low power consumption, and predictable timing are essential,

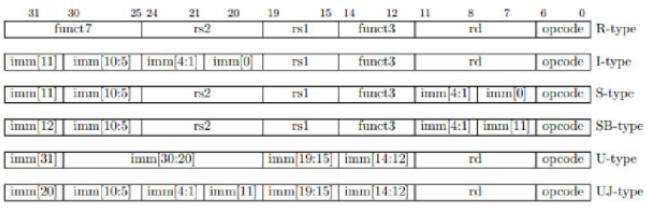

Fig. 1. INSTRUCTION FORMAT

TABLE I
INSTRUCTION CATEGORY AND OPCODE

CATEGORY	OPCODE	INSTRUCTION	EXAMPLE
Integer Register-Register Instruction(R-Type)	0110011	add, sub, sll, slt, sltu, xor, srl, sra, or, and	add rd,rs1,rs2 Register rd ¡– [rs1]+[rs2].
Integer Register-Immediate Instruction(I-Type)	0010011	addi, slti, sltiu, ori, xori, andi, slli, srli, srai	xori rd,rs1,imm Register rd ¡– [rs1] xor [imm]
Integer Computational Instruction(U-Type)	0110111	lui	lui rd,imm Register rd 20 bits MSB ¡– immed value, LSB 12 bits ¡– 0.
Unconditional Jumps (UJ-Type)	1101111	jal, jalr	jal rd,imm The jump target address = immed offest + PC. Stores the return address in register ra.
Conditional Branches (SB-Type)	1100011	beq, bne, blt, bltu, bge, bgeu	beq rs1,rs2,imm If rs1=rs2, then jump to PC+immed.
Load Instruction (I-Type)	0000011	lw, lh, lb, lhu, lbu	lw rd,rs1,imm Load the contents of [rs1]+immed into rd.
Store Instruction (S-Type)	0100011	sw, sh, sb	sw rs1,rs2,imm Store the contents of rs1 to [rs2+immed].

single-cycle RISC-V processors are a good fit. In applications like wearable technology, industrial automation, and Internet of Things devices, these processors can effectively handle tasks like sensor interfacing, control logic, and communication protocols.

A. Overview

Five logical modules make up the CPU core, as shown in Fig. 2. Based on the PC value, the CPU retrieves the instruction from the instruction memory during the instruction fetch stage. After the instruction has been decoded, the control unit receives the opcode and function class. After that, register values are read and sent to the ALU, which processes the data and returns the result to the register bank or data memory based on control signals. The instruction memory block, instruction decoder, control unit, program counter, and special adder for

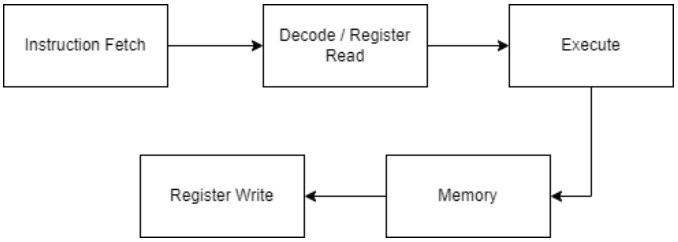

Fig. 2. Overview

PC incrementation are all included in the fetch, decode, and control block. It also includes signals for both internal and external exceptions, as well as a combinatorial control unit. 31 GPRs, register addressing, read/write logic, and an ALU for register-register and register-immediate arithmetic operations make up the register bank and ALU logic module. While the register bank supplies the second operand, the sign extension block guarantees proper sign extension and reordering of 32-bit immediate operands for the ALU in register-immediate operations. Calculating branch and jump target addresses does not involve the ALU. To decide whether to accept a branch, the control unit uses the ALU's computation results of branch instruction conditions. A precise reordering and sign extension of immediate values are required in the decoding step because different instruction formats have different encodings. Data is stored in the memory block so that it can be read or written to load and save instructions. Writing computed or obtained values from memory or the ALU is done by the register write block.

B. Microarchitecture

Address of the following processor instruction is stored in a unique register called the PC, which is located inside the processor. The data width of the PC register in the RV32I ISA is 32 bits. The instruction memory receives the address of the PC register in order to retrieve the instruction. The PC is typically multiplied by 4 when an instruction is being executed normally. The majority of CPUs are byte addressable, which accounts for the factor 4 increase. This work multiplies the PC by one to obtain all 32-bit instructions from the instruction memory, hence reducing the complexity of the memory design. The PC register records the computed value of the jump addresses when B-type (branch) and J-type (jump) instructions are executed. The modified PC value is assigned to the register to fetch the subsequent instruction in the next clock cycle. Furthermore, there is a specific adder for raising the value of the program counter.

The instruction memory houses the 32-bit computer code. The address of the PC register serves as the instruction memory's input, and the machine code for the instruction stored in the position indicated by that address serves as its output. Generally memory from which we read instruction is read-only, but in our work, we are reading the data from an external text file which contains the instructions. The text file is read, and the instructions are loaded into the instruction memory. The register file, ALU, data memory, and data modification module are among the modules that receive the 32-bit instruction from the instruction memory's output. The 32-bit output functions as a control signal that, according to the kind of instruction to be executed, activates the values modules.The address rs1, rs2, and rd is provided in the register file by the instruction. The control unit receives the opcode and function class in order to supply signals to every module. Opcode, func3, func7, and ALUop are inputs to the control unit. The control unit selects and executes instructions correctly by sending various control signals to various modules based on these inputs. When Reg WR is 1, the register file is placed in write mode; when it is 0, it is placed in read-only mode.ALU Ctrl (4-bit), for example, instructs ALU what action to take on the operands. In a similar manner, additional modules function [1].

The register file consists of 32 GPRs, each of 32 bits which holds the data. A register is a type of memory that helps the processor temporarily retain the information needed to process an instruction. Because there aren't many registers and they're directly connected to the ALU, intermediate data can be computed and stored very quickly [1]. The register file block takes 5 bit addresses of destination register, two source registers, and input data which is to be wriiten into the destination register(lw instruction) as inputs and provides the content of source registers as output. The processor consists of a sign extension block for I-Type, S-Type, branch and jal type instructions. It takes the machine code and a 2-bit control signal, Immsrc, as input and give 32 bit sign extended value as output depending upon Immsrc signal. The extension is done according to instruction format. Immsrc is kept as 00 for I-type, 01 for S-type, 10 for branch and 11 for jal.

The ALU performs logical, shifting, slt, and arithmetic operations. An essential building block of every processor is the ALU. Data must pass through a register in the ALU, it cannot store data on its own. Two 32-bit inputs, one for each operation, and a control signal to select which operation to execute are provided to the ALU. It has a 32-bit output. One input can either be from register file or an immediate value. The signals are routed through a multiplexer to the ALU's input. ALU can perform ADD, SUB, SLT, XOR, OR, AND, SLL, SRL and SRA. The output signal is allocated to the register file and data memory based on the kind of instruction.

The data memory is where information stored that isn't machine code or instructions. This data is typically generated when the processor is operating. Random-Access Memory (RAM) is the name for this kind of memory, which can read and write data. The Least Significant Byte (LSB) is kept in memory first when storing data because RISC-V is little endian. The memory segment of data has WriteEn, ALU output and address generated (for lw and sw) as inputs and a 32-bit output which is assigned to the mux which decides the result source. The ALUDecoder block is used to specify which operation, based on different inputs such ALUop, func3, func7, and other control signals, should be executed. The ALU block is designated to handle the output. The Quartus software's RTL

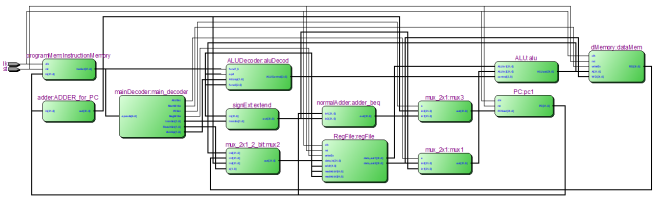

Fig. 3. RTL View of RISCV processor generated in Quartus

perspective of the processor architecture is shown in Fig. [3].

IV. ASSEMBLER

```
PS D:\Nirma\Sem 6\CA\riscprocessor> python -u "d:\Nirma\Sem 6\CA\riscp
rocessor\mainFile.py"
['addi', 'x1', 'x0', '4095']
['addi', 'x2', 'x1', '1']
['add', 'x6', 'x2', 'x1']

Machine Code succesfully written in file machineCodes.txt
```

Fig. 4. Code conversion using Python

Machine code, the binary form of instructions that a computer's CPU can directly execute, is translated from assembly language code using a specific type of translator called an assembler. An assembler is an essential tool in the process of translating instructions written in human-readable assembly language into a low-level language that computers can comprehend and use. The input file containing the assembly instructions are fed into the assembler & converted to machine codes. The processor receives the output file in Verilog HDL, which contains the necessary machine codes. The binary equivalent is written to the instruction memory in preparation for execution. R-type, I-type, load, store, B-type, and J-type instructions are all supported by the assembler. Python is used to implement it. The code and its respective machine codes generated are sown in fig[3], fig[4], fig[5] and fig[6] .

assemblyFile.txt | mainFile.py

assemblyFile.txt

```
1   addi x1 x0 4095
2   addi x2 x1 1
3   add x6 x2 x1
```

Fig. 5. User's Instruction

V. SIMULATION

The machine codes are loaded into the verilog after the assembly codes are written in python file. The processor operates in accordance with the machine code that it receives as input. Quartus II is the software that runs the processor. A particular kind of translator: multiple programs are sent

assemblyFile.txt | mainFile.py | machineCodeInHex.txt

machineCodeInHex.txt

```
1   0xfff00093
2   0x108113
3   0x110333
4
```

Fig. 6. Machine code in Hex

assemblyFile.txt | mainFile.py | machineCodes.txt

machineCodes.txt

```
1   11111111111100000000000010010011
2   00000000000100001000000100010011
3   00000000000100010000001100110011
4
```

Fig. 7. Machine code in Binary

as test cases to the assembler for translation and, later, to the instruction memory for testing and validation in order to accomplish testing at multiple levels. Now, assembly language code is converted into machine code—the binary form of instructions that a computer's CPU can directly execute—using an assembler. Both the program level and the instruction level correctness are thoroughly tested by the test cases. The simulations are done in ModelSim Altera. As indicated in the subsections below, simulations are performed for a variety of instruction categories.

A. R-Type and I-Type

The assembly code snippet for R-Type and I-Type instructions and it's waveforms are shown in Fig[7] and Fig[8] respectively.For simulations, instructions add and addi are taken. The other R-types (or, and, xor, sub, slt) and I-Type can be examined in a similar manner.

B. L-Type and S-Type

The assembly code snippet for L-Type and S-Type instructions and it's waveforms are shown in Fig[9] and Fig[10] respectively.

C. UJ-Type

The assembly code snippet for J-Type instruction and it's waveforms are shown in Fig[11] and Fig[12] respectively.

```
PS D:\Nirma\Sem 6\CA\riscprocessor> python -u "d:\Nirma\Sem 6\CA\riscp
rocessor\mainFile.py"
['addi', 'x1', 'x0', '45']
['addi', 'x2', 'x1', '-5']
['add', 'x6', 'x2', 'x1']

Machine Code succesfully written in file machineCodes.txt
```

Fig. 8. Code for R and I Type instruction

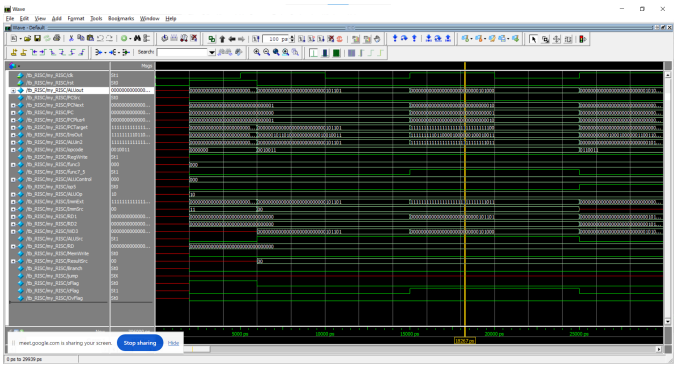

Fig. 9. Simulation of R and I Type instruction

```
PS D:\Nirma\Sem 6\CA\riscprocessor> python -u "d:\Nirma\Sem 6\CA\riscprocessor\
mainFile.py"
['addi', 'x1', 'x0', '100']
['addi', 'x3', 'x0', '255']
['sw', 'x3', '30', 'x1']
['lw', 'x4', '30', 'x1']

Machine Code succesfully written in file machineCodes.txt
```

Fig. 10. Code for 'lw' and 'sw' instructions

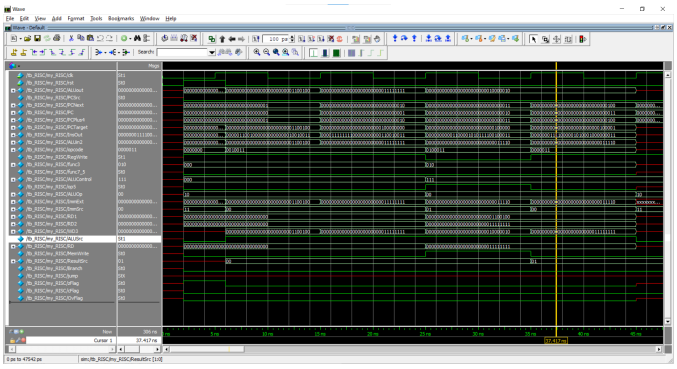

Fig. 11. Simulation for 'lw' and 'sw' instructions

```
PS D:\Nirma\Sem 6\CA\riscprocessor> python -u "d:\Nirma\Sem 6\CA\riscprocessor\
mainFile.py"
['addi', 'x1', 'x0', '100']
['jal', 'x1', '15']

Machine Code succesfully written in file machineCodes.txt
```

Fig. 12. Code for 'jal' instruction

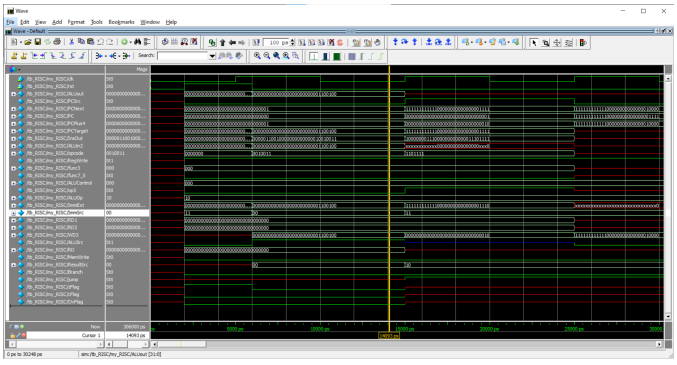

Fig. 13. Simulation for 'jal' instruction

```
PS D:\Nirma\Sem 6\CA\riscprocessor> python -u "d:\Nirma\Sem 6\CA\riscprocessor\
mainFile.py"
['addi', 'x1', 'x0', '100']
['addi', 'x2', 'x0', '100']
['beq', 'x2', 'x1', '300']

Machine Code succesfully written in file machineCodes.txt
```

Fig. 14. Code for 'beq' instruction

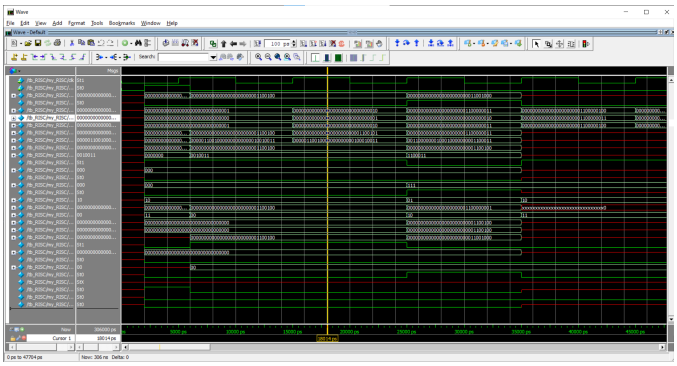

Fig. 15. Simulation for 'beq' instruction

D. SB-type

The assembly code snippet for SB-Type instruction and it's waveforms are shown in Fig[13] and Fig[14] respectively. For simulations, beq instruction is taken. The other SB-Type instructions can be examined in a similar manner.

VI. CONCLUSION

Verilog HDL is used to implement a completely synthesized 32-bit processor based on the open-source RISC-V (RV32I) ISA. A single clock cycle is used to execute each instruction, as demonstrated by the simulations. In comparison to pipelined or superscalar processors, the control logic of single-cycle processors is simpler because each instruction is carried out in a single cycle. Smaller chips and reduced power consumption can be the results of this simplicity. Every instruction has a predictable execution time as it completes in a fixed amount of time, or one clock cycle. This consistency makes timing analysis simpler and makes debugging and verification easier. The above implementation supports a total of 15 instructions from R-Type, I-Type, L-Type, S-Type and UJ-Type. The assembler is implemented in Python. The user can write the assembly code in a text file which is given as an input to the assembler. The assembler converts the instructions into corresponding machine codes. The processor takes the machine code text file as an input and performs the operation. The entire development environment is made simpler by the assembler and processor's seamless integration. Without having to worry about the complexities of machine code generation, users can write assembly code directly and watch it execute in simulation tools like ModelSim. To confirm the processor's functionality, the developer does not need to manually input the machine code into the simulation environment. This minimizes error and lowers the testing's complexity. Debugging becomes more effective when machine code and assembly code are closely

integrated within the development environment. If developers see the assembly code and watch how it executes in a simulation, they can find and fix problems fast. The machine code that is generated is specifically designed for the target processor architecture because the assembler is integrated into the development environment. This guarantees the code's portability and compatibility across various platforms without the need for manual adjustments.

ACKNOWLEDGMENT

We would like to sincerely thank each and every one of our fellow classmates who helped us to successfully complete the project, "Integrated assember-processor instruction execution." Without Dr. Dhaval Shah's steadfast support and direction, this project would not have been feasible. His knowledge and encouragement were crucial in helping to shape the project.

REFERENCES

[1] D. K. Dennis et al., "Single cycle RISC-V micro architecture processor and its FPGA prototype," 2017 7th International Symposium on Embedded Computing and System Design (ISED), Durgapur, India, 2017, pp. 1-5, doi: 10.1109/ISED.2017.8303926. keywords: Registers;Computer architecture;Field programmable gate arrays;Hardware;Instruction sets;Hardware design languages;Decoding;RISC-V;RV32I ISA;Micro-architecture;FPGA prototype,

[2] Gur, E., Sataner, Z. E., Durkaya, Y. H., Bayar, S. (2018). FPGA Implementation of 32-bit RISC-V Processor with Web-Based Assembler-Disassembler. 2018 International Symposium on Fundamentals of Electrical Engineering (ISFEE).

[3] Single Cycle 32-bit RISC-V ISA Implementation and Verification

[4] https://en.wikipedia.org/wiki/RISC-V

[5] https://www.ijraset.com/research-paper/basic-risc-v-instruction-set-architecture

[6] https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[7] A. Waterman, Y. Lee, D. A. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson, "The risc-v instruction set manual," 2014.