Exploring CPU Microarchitecture Design with RV32IM Instruction Set Architecture

Krishbin Paudel[1], *Bibek Pariyar*[2]

krishbinp@outlook.com, 076bei012.bibek@pcampus.edu.np

Abstract

The RV32I processor that we are going to create here is a variant of the RISC-V architecture that implements the integer instruction set. It is a 32-bit processor with a simple and elegant design that emphasizes modularity, extensibility, and flexibility. The RV32I processor supports a wide range of applications, including embedded systems, microcontrollers, and low-power devices.

The RV32I processor implements a subset of the RISC-V instruction set that includes basic integer arithmetic and logical operations, as well as memory access and control flow instructions. It also supports a number of extensions that can be added to the core instructions.

Verilog was the hardware language of choice here to study and develop the architecture. Simulations of verilog code were done in Icarus Verilog toolchain. An assembler was written by us to translate the assembly code to machine code.

Unlike other popular CPU micro-architecture, RISC-V is open source. Its Instruction Set Architecture is freely available to read and implement. The main focus of this article is to recreate a RV32IM architecture and run some programs on it. The architecture design and the components that builds it up will be discussed.

The conclusion reflects on the current state of RV32IM microarchitecture design and its potential future developments. It emphasizes the impact of open-source hardware on democratizing access to advanced computing technologies and the role of the RV32IM ISA in driving this paradigm shift

Index Terms: RV32I processor, RISC-V architecture, integer instruction set, 32-bit processor, modularity, extensibility, flexibility, embedded systems, microcontrollers, low-power devices, arithmetic operations, logical operations, memory access

1. Introduction

RISC-V dates back to 2010 which was spun up as a research project in UC, Berkeley. Since then it has gain popularity in both academic and industry circles. RISC-V is an open-source Instruction Set Architecture(ISA) and stands as ”Reduced Instruction Set Architecture Five”. Its name is a general architecture shift RISC has gone through the years and V is a representation its fifth iteration. The open source nature of RISC-V allows anyone to design, implement and sell RISC-V based processors without the need to pay royalties or license fees. With more than a billion devices already in the market, adoption of RISC-V has surged an all time high. RISC-V is increasingly being used in various applications, including mobile devices, embedded systems, and high-performance computing. The flexibility of the architecture and the open-source nature of its development make it an attractive option for many developers and companies.

1.1. RISC-V ISA

RISC-V ISA is an Instruction Set Architecture (ISA) that defines the set of instructions that a processor can execute. It is a type of Reduced Instruction Set Computing (RISC) architecture that is designed to be simple, modular, and extensible. The RISC-V ISA includes a small core set of instructions that are common to all implementations, as well as optional instruction extensions that can be added to support specific applications or workloads. This modular design allows RISC-V to be customized for different use cases, without sacrificing compatibility with the core instruction set. RISC-V also supports a range of addressing modes, from simple load-store operations to more complex operations for managing memory and accessing I/O devices. It also provides support for virtual memory and exception handling, which are essential for operating systems and other software that runs on top of the hardware.

1.2. RV32IM

RV32I is a subset of the RISC-V Instruction Set Architecture (ISA) that includes the core integer instructions for 32-bit RISC-V processors. The ”RV32” part of the name indicates that this subset is designed for 32-bit processors, while the ”I” indicates that it includes only integer instructions. [1]

The RV32I subset includes a total of 47 instructions, which cover basic arithmetic and logical operations, memory access, control flow, and other common operations.
Some of the key instructions included in RV32I are:

- add, sub, and, or, xor: basic arithmetic and logical operations
- load and store: memory access operations
- branch and jump: control flow instructions for branching and jumping to different parts of a program
- lui and auipc: instructions for loading immediate values into registers
- csr and fence: instructions for single stepping and control registers

1.2.1. RV32I Instruction Encoding

The RISC-V Instruction Set Architecture (ISA) uses a fixed-length instruction encoding format. For RV32I, each instruction is encoded using 32 bits, which is the standard word size for 32-bit RISC-V processors. [1]

The 32-bit instruction word is divided into several fields, each of which specifies a different aspect of the instruction. The exact layout of the fields varies depending on the instruction,

but there are some common fields that are used by many instructions in the RV32I subset.

In RISC-V, there are six primary types of instruction encodings that are used to specify different types of operations and operand.

1. R-Type
2. I-Type
3. S-Type
4. B-Type
5. U-Type
6. J-Type

1.2.2. Registers

Table 1: *RV32I Register Names and Indices*

Register	Abbreviation	Index
x0	zero	0
x1	ra	1
x2	sp	2
x3	gp	3
x4	tp	4
x5-x7	t0-t2	5-7
x8	s0/fp	8
x9	s1	9
x10-x11	a0-a1	10-11
x12-x17	a2-a7	12-17
x18-x27	s2-s11	18-27
x28	t3	28
x29	t4	29
x30	t5	30
x31	t6	31

1. PC: Program Counter
2. SP: Stack Pointer
3. LR: Link Register
4. GP: Global Pointers

Register are used for storing data, addresses and control information during program execution.

1.2.3. Control and Status Registers

CSR (Control and Status Registers) are a type of special-purpose register in RISC-V processors, including those based on the RV32I ISA. These registers are used to store control and status information related to the processor and its operation. They are accessible only via special instructions called ”CSR instructions”.[1] In RV32I, there are 4096 CSR registers defined by the RISC-V specification. Each CSR is identified by a unique 12-bit address, ranging from 0x000 to 0xFFF. These addresses are used as operands for the CSR instructions.

2. ApiCore-RISCV RV32I Design

ApiCore is the name of the CPU microarchitecture we developed using RISC-V ISA.

2.0.1. Features

- 32 bit RISCV CPU
- 32-bit word length
- 32-bit address bus
- Based on RV32I
- 32 32-bit registers
- Harvard Architecture
- In order execution
- 8KB ROM, 64KB RAM
- 100 MHz clock speed

2.1. Architecture

The architecture of our CPU includes a Arithmetics and Logic Unit, Load and Store Unit, Memory Interface Unit, Instruction Fetch Unit, Instruction Decode and Control Unit, Execution Unit.

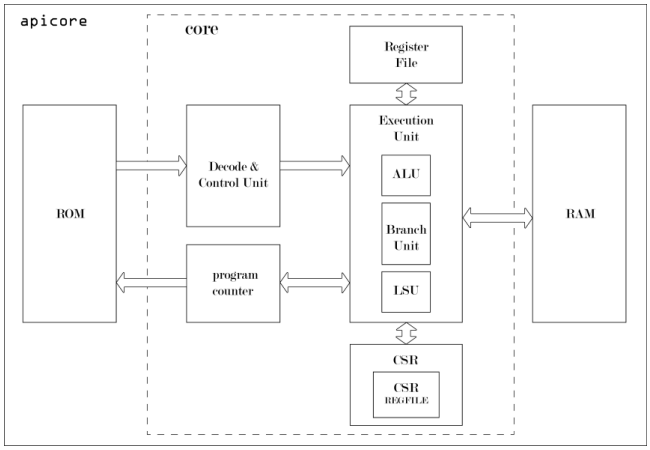

Figure 1: *ApiCore*

2.2. Execution Unit

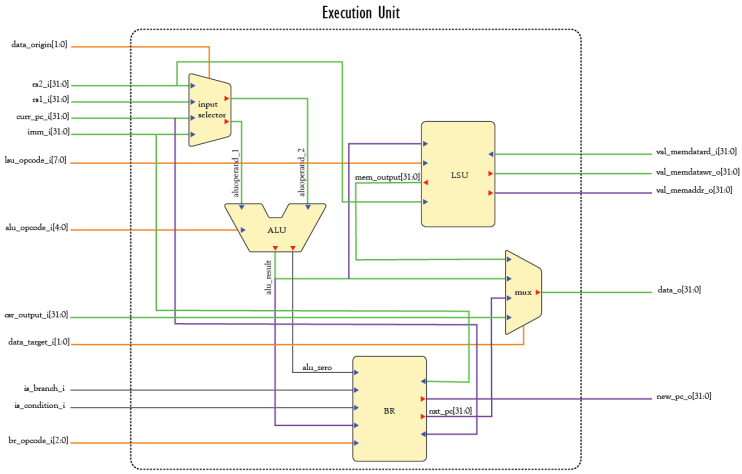

Figure 2: *RV32I Execution Unit*

In RV32I, the Execution Unit (EXU) is a functional unit that performs the actual computation specified by the instruction. The EXU contains several subunits that work together to execute instructions:

- Arithmetic and Logic Unit (ALU)
- Branch Unit
- Multiplication/Division Unit

• Shift Unit

The EXU is responsible for executing all types of instructions that require computational operations, including arithmetic, logic, shift, and branch/jump instructions. It receives control signals from the Instruction Decode and Control Unit (IDCU) and generates the appropriate results based on the operands and operation specified by the instruction.

2.3. Arithmetics and Logic Unit

In RV32I, the Arithmetic and Logic Unit (ALU) is a subunit of the Execution Unit (EXU) that performs arithmetic and logic operations on data. The ALU is responsible for executing most arithmetic and logic instructions in the RV32I instruction set architecture. The ALU can perform operations such as addition, subtraction, logical AND/OR/XOR, and bit shifting.

The opcodes are necessary to determine the type of instruction. This can be different on other architecture but is defined as such for ApiCore. We will later see that it is the job of Instruction decoder to feed this opcode to ALU.

2.3.1. ALU Opcode

Instruction Type	Instruction	d4: d3 : d2-d1
Addition/Subtraction	ADD	0:0:000
Addition/Subtraction	SUB	0:1:000
Logical AND	AND	0:0:111
Logical OR	OR	0:0:110
Logical XOR	XOR	0:0:100
Shift Left Logical	SLL	0:0:001
Set Less Than	SLT	0:0:010
Set Less Than Unsigned	SLTU	0:0:011
Shift Right Logical	SRL	0:0:101
Shift Right Arithmetic	SRA	0:1:101

Table 2: *Opcode of ALU instructions in RV32I*

2.4. Load and Store Unit

In RV32I, the Load and Store Unit (LSU) is a subunit of the processor that is responsible for handling memory operations. The LSU is designed to manage the data transfer between the processor's registers and the main memory. The LSU can perform two types of memory operations:

• Load Operations

• Store Operations

In RV32I, the LSU is also responsible for managing the alignment of data when transferring it between the registers and memory. This is important because some memory architectures require that data be aligned on specific byte boundaries. The LSU works in conjunction with the Data Memory Interface Unit (DMIU), which provides the LSU with access to the main memory. The DMIU handles the actual data transfer between the processor and memory, while the LSU manages the memory addresses and data alignment.

2.4.1. LSU Opcode

Instruction Type	Instruction	func3 : opcode(d6:d4)
Load Byte	LB	000:00000
Load Half Word	LH	001:00000
Load Word	LW	010:00000
Load Byte Unsigned	LBU	100:00000
Load Half Word Unsigned	LHU	101:00000
Store Byte	SB	000:01000
Store Half Word	SH	001:01000
Store Word	SW	010:01000

Table 3: *Opcode of LSU instructions in RV32I*

2.5. Branch Unit

In RV32I, the Branch Unit (BRU) is a subunit of the processor that is responsible for handling branch instructions. The BRU is designed to calculate the target address of a branch instruction and to determine whether the branch should be taken or not. The BRU takes the instruction opcode and the operands as input, and based on the instruction type, it can calculate the target address for the branch. The target address is typically calculated by adding a signed offset to the current program counter (PC) value. Once the target address is calculated, the BRU compares it with the current PC value and determines whether the branch should be taken or not. If the branch is taken, the BRU updates the PC value with the target address, causing the program to jump to the specified location in memory. In RV32I, the BRU can handle several types of branch instructions, including:

• Conditional Branch Instructions

• Unconditional Jump Instructions

• Call Instructions

• Return Instructions

The BRU works in conjunction with the Instruction Fetch Unit (IFU) and the Instruction Decode and Control Unit (IDCU), which fetch and decode the branch instructions, respectively. The BRU is a critical component of the processor in RV32I, as it allows the processor to implement control flow in software, which is essential for many computing tasks.

2.5.1. BR Opcode

Instruction Type	Instruction	bropcode:: func3
If Equal	BEQ	000
If Not Equal	BNE	001
If Less Than	BLT	100
If Greater or Equal	BGE	101
If Less Than (U)	BLTU	110
If Greater or Equal (U)	BGEU	111

Table 4: *Opcode of BR instructions in RV32I*

2.6. Decoder and Control Unit

In RV32I, the Instruction Decode and Control Unit (IDCU) is a subunit of the processor that is responsible for decoding instructions and generating control signals for the other subunits of the processor. The IDCU takes the instruction opcode and operands as input and decodes the instruction to determine the operation that needs to be performed. It then generates the control signals required by the Arithmetic and Logic Unit (ALU), Load and Store Unit (LSU), Execution Unit (EXU), and other subunits of the processor.

Mainly there is a control signal to control which data to direct to exu and which data to get from exu. Other control signals include to enable read and write from registers, enable

read and write from csr. The decoded instruction are matched and required data are provided to the Execution Unit.

2.6.1. *Block Diagram of Decoder and Control Unit*

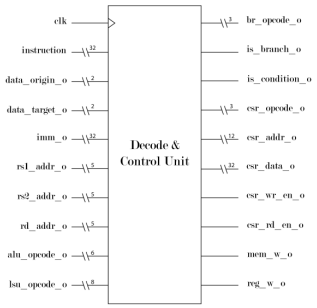

Figure 3: *Block Diagram of Decoder and Control Unit*

2.7. Register File

In RV32I, the Register File is a subunit of the processor that is responsible for storing the processor's general-purpose registers. The Register File consists of a set of 32 registers, each of which is 32 bits wide. These registers are used for storing data and intermediate values during the execution of programs. The Register File has two read ports and one write port. The two read ports allow two operands to be read from the Register File simultaneously, while the write port allows one operand to be written to the Register File.

3. Results

All the architecture is written in Verilog which is available at `https://github.com/krishbin/apicore`.

3.1. A Fibonacci Series Assembly Program for RV32IM

```
addi x4, x0, 0x0                    // count = 0
addi x5, x0, 0x0d                   // n = 10
addi x2, x0, 0x00                   // a = 0
addi x3, x0, 0x01                   // b = 1
start:  sb x2, 0(x4)                // store a
        add x1, x2, x3              // temp = a + b
        add x2, x0, x3              // a = b
        add x3, x0, x1              // b = temp
        addi x4, x4, 0x1            // count++
        bne x5, x4, start
addi x0, x0, 0x0            // nop
addi x0, x0, 0x0            // nop
addi x0, x0, 0x0            // nop
addi x0, x0, 0x0            // nop
addi x4, x0, 0x0            // count = 0
print:  lbu x3, 0(x4)
        addi x4, x4, 0x1
        bne x5, x4, print
        beq x5, x4, end
addi x0, x0, 0x0 // nop
end: addi x0, x0, 0x0 // nop
```

3.1.1. *RAM Dump*

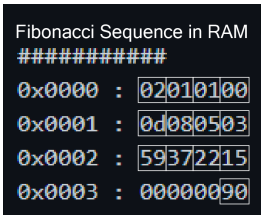

Figure 4: *Fibonacci RAM Dump*

3.1.2. *Simulation Waveform*

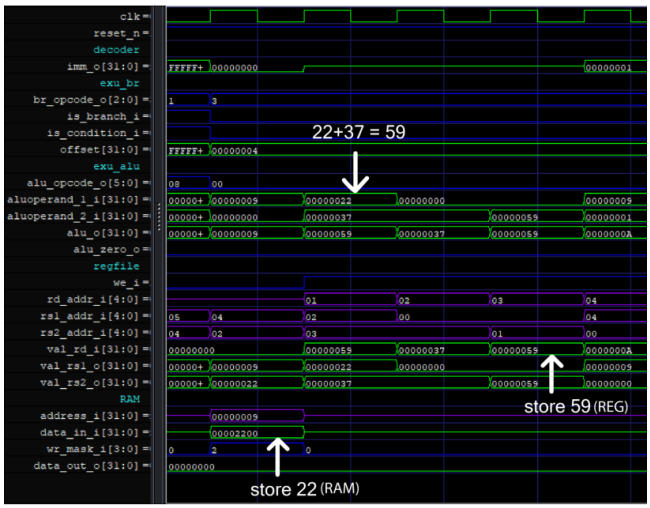

Figure 5: *Fibonacci Simulation Waveform*

4. Discussion

Implementing RV32I in Verilog is a complex task that requires a deep understanding of both the RISC-V architecture and Verilog hardware description language. However, the process of implementing RV32I in Verilog can provide several valuable insights into processor design and computer architecture. First and foremost, implementing RV32I in Verilog can provide a better understanding of the RISC-V architecture itself. By implementing the different components of the RV32I processor in Verilog, one can gain a deeper understanding of how the various components of the processor work together to execute instructions. This can include understanding how the instruction fetch, decode, and execute stages work, as well as how the various types of instructions are implemented in hardware.

5. References

[1] "Risc-v instruction set manual, volume i: Risc-v user-level isa," Dec 2019. [Online]. Available: https://riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf