Custom CPU Architecture: ARM Cortex M0 implementation

David Ortiz Cota[a], **Sebastián Pulido Espinoza**[a], **Alfredo Romo**[a], **Jorge González Díaz**[a]

[a]*Student*

José Ignacio Parra Vilchis

Abstract—*This paper delves into the intricate design and implementation of a complete microprocessor based on the ARM Cortex M0 Harvard architecture. Moving beyond a single instruction, it explores the fundamental building blocks that orchestrate the complex process of fetching, decoding, and executing instructions. The paper outlines the core architecture of the microprocessor, including the key functional units like the Control Unit, Arithmetic Logic Unit (ALU), Program Memory, register file and RAM. It delves into the instruction set architecture (ISA), detailing the instruction formats, addressing modes, and their impact on program execution. The implementation details encompass the logic design principles employed for each unit, exploring the data flow and control signals that govern their operation. Furthermore, the paper discusses the memory hierarchy and its interaction with the processor.*
Finally, the paper touches upon the challenges encountered during implementation, potential optimization strategies, and the verification methodologies used to ensure the microprocessor's functionality. This comprehensive exploration provides valuable insights into the intricate world of microprocessor design, serving as a valuable resource for computer architecture students and professionals.

keywords—*Arm Cortex M0, Harvard, Microcontroller, Microprocessor, Verilog, FPGA board, Hardware Description Language, Soft processor*

Contents

1. Introduction

Microprocessors, the brains of modern computing devices, orchestrate the complex dance of data processing that powers our digital world. This paper embarks on a journey to unravel the intricacies of microprocessor design and implementation, delving into the fundamental principles that govern these remarkable machines.

A microprocessor is a computer processor for which the data processing logic and control is included on a single integrated circuit (IC), or a small number of ICs. The microprocessor contains the arithmetic, logic, and control circuitry required to perform the functions of a computer's central processing unit (CPU). The IC is capable of interpreting and executing program instructions and performing arithmetic operations. The microprocessor is a multipurpose, clock-driven, register-based, digital integrated circuit that accepts binary data as input, processes it according to instructions stored in its memory, and provides results (also in binary form) as output. Microprocessors contain both combinational logic and sequential digital logic, and operate on numbers and symbols represented in the binary number system.

1.1. Brief History of Microprocessors.

It all began with the invention of the transistor by Bell Labs in the 1950s. These tiny, solid-state devices replaced bulky and power-hungry vacuum tubes, paving the way for smaller and more efficient electronic circuits. This technological leap laid the foundation for the integrated circuit (IC), a revolutionary concept conceived by Jack Kilby at Texas Instruments (TI) in 1958. The IC miniaturized electronics even further by integrating multiple transistors onto a single silicon chip, dramatically reducing size and complexity.
The year 1971 witnessed the birth of the first commercially available microcontrollers. There's some debate about which device holds this title. The Texas Instruments TMS 1000, developed by Gary Boone and Michael Cochran, is widely recognized as a pioneer. This 4-bit microcontroller had a simple instruction set, built-in memory, and rudimentary input/output (I/O) capabilities, making it suitable for

basic control tasks. Around the same time, Intel released its 4-bit 4004. Initially designed for calculators, the 4004's versatility and low cost led to broader applications in control systems. These early microcontrollers marked the beginning of a new era in embedded computing.

1.2. Existing Architectures

There are 2 main existing architectures used nowadays, they are Von Neumann Architecture and Harvard Architecture.

1.2.1. Hardvard Architecture:

The Harvard architecture is a computer architecture with separate storage and signal pathways for instructions and data. It is often contrasted with the von Neumann architecture, where program instructions and data share the same memory and pathways. This architecture is often used in real-time processing or low-power applications. There is no need to make the two memories share characteristics. In particular, the word width, timing, implementation technology, and memory address structure can differ. In some systems, instructions for pre-programmed tasks can be stored in read-only memory while data memory generally requires read-write memory. In some systems, there is much more instruction memory than data memory so instruction addresses are wider than data addresses.

1.2.2. Von Neumann Architecture

The von Neumann architecture is a foundational design concept for computers. It features a unified memory space that stores both program instructions and data. The CPU fetches both from this single location, simplifying the overall design and control logic. This architecture typically involves sequential access, where instructions and data are retrieved one after another. While this can be efficient for applications where instruction and data access are balanced, it can create bottlenecks for tasks requiring frequent data manipulation, as the CPU might need to wait for data retrieval before continuing program execution. Despite this potential drawback, the von Neumann architecture remains popular due to its simplicity, ease of programming, and cost-effectiveness, making it a suitable choice for general-purpose computing.

1.2.3. Dataflow Architecture

Dataflow architecture is a dataflow-based computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. Dataflow architectures have no program counter, in concept: the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution may be hard to predict.

Although no commercially successful general-purpose computer hardware has used a dataflow architecture, it has been successfully implemented in specialized hardware such as in digital signal processing, network routing, graphics processing, telemetry, and more recently in data warehousing, and artificial intelligence (as: polymorphic dataflow Convolution Engine, structure-driven, dataflow scheduling). It is also very relevant in many software architectures today including database engine designs and parallel computing frameworks.

Synchronous dataflow architectures tune to match the workload presented by real-time data path applications such as wire speed packet forwarding. Dataflow architectures that are deterministic in nature enable programmers to manage complex tasks such as processor load balancing, synchronization and accesses to common resources.

1.2.4. Vector Architecture

A vector processor or array processor is a central processing unit (CPU) that implements an instruction set where its instructions are designed to operate efficiently and effectively on large one-dimensional arrays of data called vectors. This is in contrast to scalar processors, whose instructions operate on single data items only, and in contrast to some of those same scalar processors having additional single instruction, multiple data (SIMD) or SWAR Arithmetic Units. Vector processors can greatly improve performance on certain workloads, notably numerical simulation and similar tasks. Vector processing techniques also operate in video-game console hardware and in graphics accelerators.

1.3. Existing implementations

There also 2 main existing implementations, they are the Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). An instruction set, also known as an Instruction Set Architecture (ISA), is essentially the language a processor understands. It defines a set of commands that the processor can execute to perform various operations. These operations can be simple, like adding two numbers, or more complex, like moving data between memory locations or controlling external devices.

1.3.1. RISC

RISC processors focus on a smaller set of simpler instructions. These instructions typically perform a single, well-defined operation. While they might require more instructions to achieve the same result as a single CISC instruction, their simplicity allows for faster execution and optimization. RISC processors often excel in performance due to their simpler instructions. These instructions can be decoded and executed more efficiently, leading to faster processing speeds. Well-known RISC architectures include ARM (used in most mobile devices) and SPARC (developed by Oracle).

1.3.2. CISC

CISC processors boast a rich set of complex instructions. These instructions can perform multiple operations in a single step, making programming potentially easier and more intuitive for developers. For example, a single CISC instruction might add two numbers and store the result in a specific memory location. While CISC instructions can be efficient for some tasks, their complexity can sometimes lead to slower execution compared to RISC. Decoding these complex instructions can take time, and they might not always be perfectly optimized for the hardware. Popular CISC architectures include the x86 family (used by Intel and AMD) and the POWER architecture (developed by IBM).

1.4. Our Implementation

The implementation we well be looking into in the following steps of this paper is an ARM Cortex M0 with a custom *Instuction Set Architecture* (ISA). This implementation is far simlpler …

2. Design and Implementation

Our architecture is based on an Arm Cortex M0 3-stage architecture. The 3-stage pipeline is a fundamental concept in computer architecture that optimizes the instruction processing speed within a CPU. It breaks down the instruction execution process into three distinct stages, allowing the CPU to work on multiple instructions simultaneously.

2.1. Pipeline overview

- 1. Fetch (Instruction Fetch): In this stage, the CPU fetches an instruction from memory based on the address provided by the program counter (PC). Think of it like a librarian retrieving a book (instruction) from the shelf (memory) based on the call number (PC).
- 2. Decode (Instruction Decode): Once fetched, the instruction is passed to the decode stage. Here, the control unit deciphers

the instruction by analyzing its opcode (operation code) and operands. Imagine the librarian decoding the Dewey Decimal System (opcode) and identifying the section (data to be processed) and category (operation) of the book (instruction).

- 3. Execute (Instruction Execution): In the final stage, the decoded instruction is executed. The control unit sends control signals to the relevant ALU (Arithmetic Logic Unit) or other execution units based on the instruction type (addition, data transfer, etc.). The ALU performs the designated operation and stores the result. This stage is like the librarian following the instructions (data and operation) in the book (instruction) to perform a specific task (calculation, data movement).

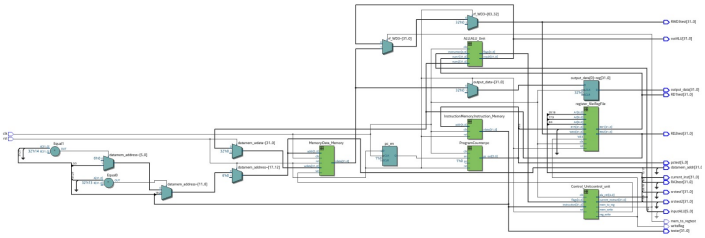

Figure 1. RTL Overview

2.2. Modules

There are 5 main modules that form the CPU, the *Program Counter*, *Instruction Memory*, *ALU*, *Data Memory*, *Register File*.
Each one of them will be explained in detail, as well as there individual inputs and outputs:

2.2.1. PC

The program counter (PC), also known as instruction pointer (IP), is a critical register within the CPU of a microprocessor. It acts like a conductor's baton, keeping track of the memory address from which the next instruction will be fetched. In essence, the PC dictates the execution flow of a program by specifying the sequence of instructions to be processed by the CPU.

The value of the program counter can be modified to allow for *jumps* and *branches* (which have yet to be implemented in our architecture). In our PC implementation, the program counter starts and resets at 0, and count up by 1 every clock cicle.

Inputs:

- Clock and Reset: the value PC value increments by 1 every clock cicle and resets to 0 when the reset signal is enabled

Outputs:

- PC Value: the actual value of the PC, will be the address of the instruction retrieved from the *Instruction Memory*.

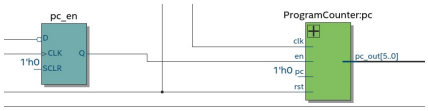

Figure 2. Program Counter RTL

2.2.2. Instruction Memory

The instruction memory module, often abbreviated as IM, serves as the brain's storage room for a program's instructions within a microprocessor. It's a read-only memory (ROM) specifically designed to hold the sequence of instructions that make up the software being executed.

The main functions of the Instruction Memory are the following:

- **Storage**: The IM stores the program code in the form of binary machine code instructions. These instructions are pre-loaded before program execution and remain fixed throughout.
- **Retrieval**: When the program counter sends a memory address, the instruction memory acts like a lookup table. It retrieves the instruction located at that specific address.
- **Feeding the Pipeline**: The retrieved instruction is then fed into the processor's pipeline for decoding and execution. This fetch-decode-execute cycle continues until the program terminates.

Inputs:

- Address: Program counter output (interpreted as the address to retrieve from memory).

Outputs:

- Instruction: 32-bit instruction retrieved from memory.

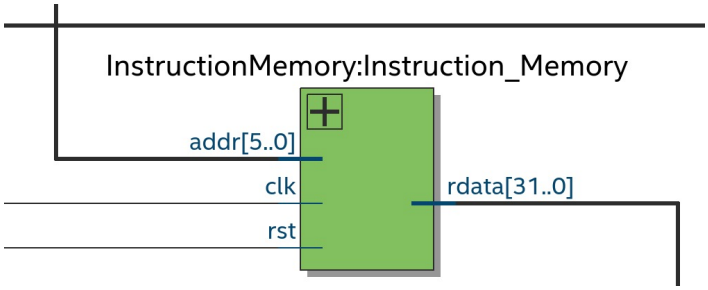

Figure 3. Instruction Memory RTL

2.2.3. Control Unit

The control unit (CU) acts as the conductor of the orchestra within a microprocessor's central processing unit (CPU). It's the central brain that interprets instructions, coordinates execution, and ensures all the other parts work together seamlessly. The details of its operation are the following:

- **Instruction Decoding**: Once fetched, the CU decodes the instruction to understand the operation it represents. This involves deciphering the binary code of the instruction and identifying what needs to be done, it takes the first 5 bits of the instruction and interprets it as an OPCODE, the remaining bits are interpreted as *destination* for the selected operation (9-bits), and operands 1 & 2 (9-bits and 9-bits).
- **Issuing Control Signals**: Based on the decoded instruction, the CU generates control signals that act like instructions for other CPU components. These signals direct the arithmetic logic unit (ALU) to perform calculations, tell registers where to store data, and coordinate communication with memory and input/output devices.
- **Execution Flow Management**: The CU doesn't just issue commands; it also manages the overall flow of program execution. It determines the sequence of steps involved in executing an instruction and ensures they happen in the correct order. This might involve fetching operands (data to be manipulated), sending them to the ALU, and then storing the results.

Inputs:

- Clk and Rst: Clock and reset signals.
- Instruction: 32-bit instruction received from Instruction Memory.
- Flags: flags recieved from a previous ALU operation.

Outputs:

- Memory to Register: Enables whether the memory output should be written to registers.
- Memory Write Enable: Write enable signal for Data Memory
- Alu Control: 5 bit OPCODE for ALU in case the instruction requires it

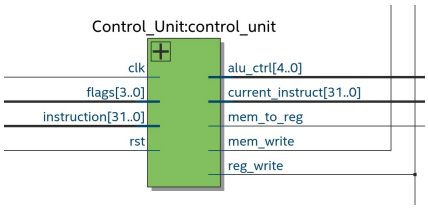

Figure 4. Control Unit RTL

2.2.4. ALU

The ALU, which stands for Arithmetic Logic Unit, is the workhorse of the CPU (Central Processing Unit) when it comes to performing calculations and making logical decisions. It can do arithmetic operation like addition and subtraction, Logical operations like comparing 2 numbers, and bitwise operations like shifts

Inputs:

- Instruction: 5-bit OPCODE that gets interpreted as a certain operation via a demultiplexer.
- Num1 & Num2: Operands for operations

Outputs:

- Flags: Flags raised by certain operations. Our architecture includes flags for *Negative, Zero, Carry & oVerflow*
- Result: 32-bit result from whatever operation was performed by the Alu.

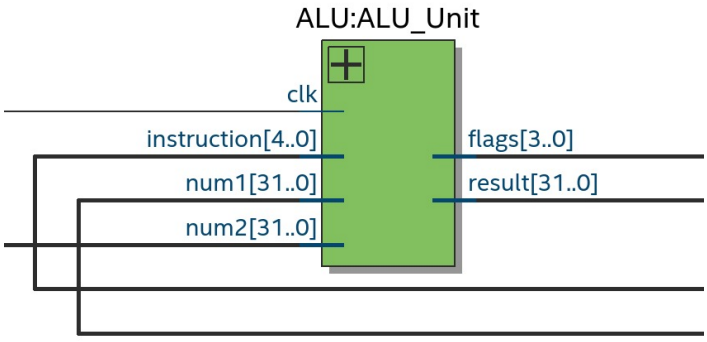

Figure 5. ALU RTL

2.2.5. Data Memory

Data memory, also commonly referred to as RAM (Random Access Memory), is the work area of a microprocessor where it stores and retrieves the temporary data that's actively used during program execution. Unlike instruction memory, which holds the program instructions permanently, data memory is constantly changing as the program progresses. It serves as the temporary storage space for various types of data, including: Variables, Intermediate Results, Function Arguments, Return Values and Heap Memory.
Data memory is volatile, meaning it loses its content when the power is turned off. This is because RAM uses capacitors to store data, and these capacitors discharge over time without a constant power supply.

Inputs:

- Clk and Rst: When the rst signal is active, the memory resets to all 0's.
- we: Write enable to determin read or write operation.
- wdata: Data to be written to a certain memory address (in case write enable is active).
- Address: Address to be writen or retrieved from memory.

Outputs:

- Return Data: If write enable is disabled, the data fetched will be returned via this bus.

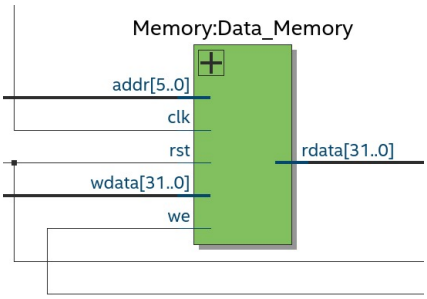

Figure 6. Data Memory RTL

2.2.6. Register File

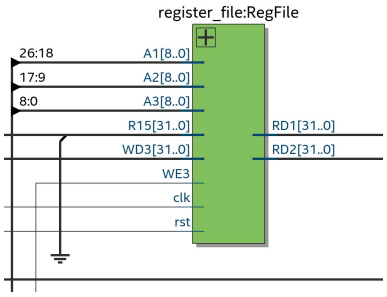

Figure 7. Register File RTL

The register file is a set of high-speed internal memory locations within the CPU (Central Processing Unit). They act as the CPU's scratchpad or workbench, holding frequently accessed data and intermediate results for quick manipulation. Compared to main memory (data memory), registers offer significantly faster access times, making them critical for efficient program execution.
This implementation consist of a 14+1 register file. The first 14 registers act general function registers, while the 15th register gets its value from the program counter + 2, the purpose of this offset is to compensate for the 2 clock cicles that have elapsed (Fetch/Decode) from the previous clock cicles.

Inputs:

- Clk and Rst: When the rst signal is active, the memory resets to all 0's.
- A1 & A2: Addresses to be acted upon, received from the instruction.
- A3: Writeback address.
- WE3: Write enable from address 3 (recieved from the Control Unit).

- R15: Value recieved from the program counter (+2 for reasons mentioned above).

Outputs:

- RD1 & RD2: Return data from A1 and A2, used as the operands for the ALU.

3. Instruction List (ISA)

We implemented a total of 24 instructions that run inside the ALU (18) and on their own (6 managed souly by the Control Unit). These instructions span from simple logical operations like an AND bitwise instruction operation, to arithmetic operations like adding of two integers. We *do not implement floating point arithmetic* natively. The implementation of a Floating Point unit goes beyond the scope of this paper, but implementing it would require doing *Soft-Floating Point*, which is a floatin point implementation that uses the CPU integer registers to store and do bit manipulation.

3.1. ANDS

This operation performs a bitwise AND on two 32-bit operands, where each bit of the output is the logical AND of the corresponding bits of the operands. This means that if both corresponding bits are 1, then the output bit is 1, otherwise it is 0.

3.2. ORRS

This operation performs a bitwise OR on two 32-bit operands, where each bit of the output is the logical OR of the corresponding bits of the operands. This means that if either corresponding bit is 1, then the output bit is 1, otherwise it is 0.

3.3. MVNS

This operation performs a bitwise NOT on a 32-bit operand, which means that it flips all the bits of the operand. This is also known as a bitwise complement.

3.4. EORS

This operation performs a bitwise XOR on two 32-bit operands, where each bit of the output is the logical XOR of the corresponding bits of the operands. This means that if the corresponding bits are not equal, then the output bit is 1, otherwise it is 0.

3.5. ADCS

This operation performs an addition with carry on two 32-bit operands, where the carry flag is taken into account. This means that if the carry flag is set, then it is added to the result of the addition. This operation is useful for implementing multi-word arithmetic.

3.6. ADDS

This operation performs a normal addition on two 32-bit operands, without taking the carry flag into account. This operation is useful for adding two numbers together.

3.7. SBCS

This operation performs a subtraction with carry on two 32-bit operands, where the carry flag is taken into account. This means that if the carry flag is set, then it is subtracted from the result of the subtraction. This operation is useful for implementing multi-word arithmetic.

3.8. SUB

This operation performs a normal subtraction on two 32-bit operands, without taking the carry flag into account. This operation is useful for subtracting two numbers together.

3.9. MULS

This operation performs a multiplication on two 32-bit operands, where the result is a 32-bit value. This operation is useful for multiplying two numbers together.

3.10. LSRS

This operation performs a logic shift right on a 32-bit operand, where the rightmost bit is shifted out, and the leftmost bit is filled with 0. This operation is useful for dividing a number by a power of 2.

3.11. LSLS

This operation performs a logic shift left on a 32-bit operand, where the leftmost bit is shifted out, and the rightmost bit is filled with 0. This operation is useful for multiplying a number by a power of 2.

3.12. ROR

This operation performs a rotate right on a 32-bit operand, where the rightmost bit is shifted into the leftmost bit and all other bits are shifted to the right. This operation is useful for circularly shifting a number.

3.13. UXTB

This operation performs an unsigned extend byte on a 32-bit operand, where the lower 8 bits of the operand are sign-extended to 32 bits. This operation is useful for converting a byte value to a 32-bit value.

3.14. UXTH

This operation performs an unsigned extend halfword on a 32-bit operand, where the lower 16 bits of the operand are sign-extended to 32 bits. This operation is useful for converting a halfword value to a 32-bit value.

3.15. SXTB

This operation performs a signed extend byte on a 32-bit operand, where the lower 8 bits of the operand are sign-extended to 32 bits. This operation is useful for converting a byte value to a 32-bit value.

3.16. SXTH

This operation performs a signed extend halfword on a 32-bit operand, where the lower 16 bits of the operand are sign-extended to 32 bits. This operation is useful for converting a halfword value to a 32-bit value.

3.17. CMP

This operation performs a comparison between two 32-bit operands, where the result is determined by subtracting one operand from the other and setting the status flags based on the result. This operation is useful for comparing two numbers.

3.18. NOP

This operation performs a no-operation, which means that it does nothing. This operation is useful for adding a delay or for synchronizing operations.

4. Memory instructions

4.1. LOADI

This instruction loads data from memory (RAM) into a register. It is useful for accessing data stored in memory.

4.2. STORE

This instruction stores data from a register into memory (RAM). It is useful for saving data to be used later.

4.3. Move (MOV)

This instruction moves data between registers or from a register to memory. It is useful for manipulating data within the processor.

5. Jump instructions

5.1. Jump (J)

This instruction allows the CPU to jump to a specific memory address to continue execution. It is useful for changing the flow of execution in a program.

5.2. Branch Equal (BEQ)

This instruction enables the CPU to branch to a specific memory address if two registers are equal. It is useful for conditional branching in a program.

5.3. Halt (HLT)

This instruction stops the CPU from executing further instructions, effectively halting the processor. It is useful for terminating a program or stopping execution when necessary.

6. Instruction Decoding Process

The design we will be looking into considers 32 bit long instructions which contain the necessary information to execute various operations within our RISC ARM-v6 based micro controller. Out of these 32 bits, the first 5 (MSB) are intended to be used as the instruction opcode which identifies the type of operation that will be executed, leaving us with 27 free bits which will be distributed equally for specifying destination and source memory addresses for instruction execution.

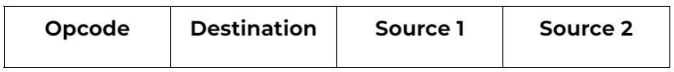

Figure 8. Instruction Set Format

6.1. Opcode

The first 5 bits ([31:27]) are used as the Opcode. When the control unit recieves an instruction, it separates this bits from the rest of the outputs and decides (via a multiplexer implemented as a verilog switch/case) wether the instruction should be performed by the ALU, or in the case of other instruction (like LOAD or STR) just raise flags to allow the low of data

Opcode	Operation
00000	RESERVED
00001	ANDS
00010	ORRS
00011	MVNS
00100	EORS
00101	ADCS
00110	ADDS
00111	SBCS
01000	SUB
01001	MULS
01010	LSRS
01011	LSLS
01100	ROR
01101	UXTB
01110	UXTH
01111	SXTB
10000	SXTH
10001	CMP
10010	NOP
10011	LOADI
10100	STORE
10101	MOV
10110	J
10111	BEQ
11000	HLT
11001-11111	RESERVED

The main consideration for designing and understanding the opcode of our proposed architecture is knowing how many instructions there are to be represented. Adding up 18 operations from the ALU, 3 from the memory interactions, and 3 from jump instructions (24 in total), we come out to needing 5 bits of opcode, given by the formula:

$$\text{Bits} = \lceil \log_2 x \rceil \tag{1}$$

6.2. Destination

The destination is given in the following 9 bits of the instruction after the OPCODE ([26:18])

6.3. Operands 1 & 2

The 2 operands (op1 and op2) are used as the operands for the ALU, or valuese for other instructions, they are encoded in the last 18 bits of the opcode (OP1 on [17:9]) (OP2 on [8:0])

7. Instructions Testbench

This section is dedicated to explaining the testing and operation of some relevant instruction

7.1. ADD

The ADD instruction is beign tested by sending the instruction 00110000000001000000010000000111 (show in the Testbench as current instruction). By decoding it we get the opcode to be 00110 which is the add instruction, Then the next nine bits are the destination where we want to store the result (in this case address 1), them the next 9 bits form operand 1 and the last 9 operand 2. This instruction would be similar to writing C code like x = 1 + 2

7.2. AND

The AND instruction is beign tested by sending the instruction 00001000000001000000011000000010 (show in the Testbench as current instruction). By decoding it we get the opcode to be 00001 which is the AND instruction, Then the next nine bits are the destination where we want to store the result (in this case address 1), them the next 9 bits form operand 1 and the last 9 operand 2. This instruction would be similar to writing C code like x = (1 | 3)

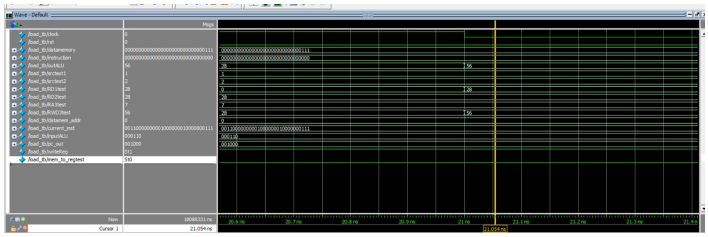

Figure 9. ADD Testbench

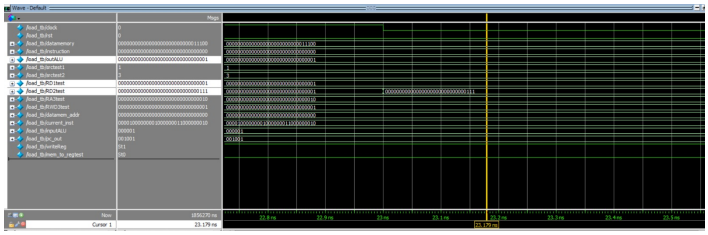

Figure 10. AND Testbench

8. Case of Study:

8.1. Program description and Instructions

8.2. C like code

8.3. Data Flow

8.4. Testbench & results

9. Conclusion

This exploration has unveiled the fundamental building blocks of a Central Processing Unit (CPU). We've journeyed through the Program Counter (PC), the conductor keeping track of instruction flow. We've peeked into the Instruction Memory (IM), the program's instruction storehouse. We've witnessed the Control Unit (CU) acting as the maestro, interpreting instructions and directing the CPU's operations. The Arithmetic Logic Unit (ALU) emerged as the workhorse, performing calculations and making logical decisions. Data Memory, often referred to as RAM, served as the CPU's ever-changing workspace. Finally, the Register File proved to be the CPU's scratchpad, holding frequently accessed data for quick manipulation.

By understanding these core components, we gain a deeper appreciation for the intricate dance that unfolds within a CPU. Each component plays a vital role, transforming a sequence of instructions into actions that power our digital world.

However, this journey is just the beginning. The world of CPUs is vast, with advanced features like pipelining, caching, floating poing arithmetic and multi-core architectures waiting to be explored. As we delve deeper, we'll uncover how these components work together at high speeds, optimizing program execution and driving ever-increasing processing power. This newfound understanding can empower us to make informed decisions about the technology we use and appreciate the remarkable engineering marvels that bring our computers to life.

10. Sources

1. *Shirriff, K. (2022, March 9). The surprising story of the first microprocessors. IEEE Spectrum. https://spectrum.ieee.org/the-surprising-story-of-the-first-microprocessors*
2. *The Editors of Encyclopaedia Britannica. (n.d.). transistor summary. Encyclopedia Britannica. https://www.britannica.com/summary/transistor*
3. *Cass, S. (2024, March 15). Chip Hall of Fame: Intel 4004 microprocessor. IEEE Spectrum. https://spectrum.ieee.org/chip-hall-of-fame-intel-4004-microprocessor*
4. *Arm Ltd. (n.d.). What is RISC? Arm | the Architecture for the Digital World. https://www.arm.com/glossary/risc*
5. *RISC vs. CISC. (n.d.). https://cs.stanford.edu/people/eroberts/courses*
6. *Cortex-M0. (n.d.). https://developer.arm.com/Processors/Cortex-M0*
7. *https://course.ece.cmu.edu/ ece740/f13/lib/exe/fetch.php%3Fmedia=seth-740-fall13-module5.1-simd-vector-gpu.pdf*