# MIPS32 Simulator Proposal

> **Name:** Ekansh Somani | **Roll:** 20110065
>
> **Course:** CS 215 - Computer Organization and Architecture
>
> **GitHub Link:** MIPS32-Simulator | **Team Name:** Lone-wolf

## My Understanding of the problem statement

Design a simulator for 32 bits MIPS/ARM processor (I choose MIPS). It must take in all the instructions from a file containing a sequence of 32 bits instruction codes, and then execute them. It also needs to have memory mapped IO and the output needs to be displayed on the screen.

## MIPS32 Processor and ISA Basic Details

- "MIPS architecture is based on a fixed length, regularly encoded instruction set. ... [where] ... All the operations are performed on operands in processor registers, and main memory is accessed only by load and store instructions." [1.a]
- The MIPS32 Architecture has had 6 releases of ISAs' of which Release-2 and Release-6 had major revisions.[1.a]
- The instructions for main MIPS32 processor can be divided into four major categories: Load and Store, Computational, Jump and Branch, and Address computation and large constant Instructions. There are some miscellaneous instructions which don't fall into any of the above categories. Miscellaneous instruction include: Exception, Conditional Move, Prefetch, NOP and Serialization Instructions.[1.b]
- MIPS32 Processor has 32 general purpose registers, a `PC` (program counter), `$count` (cycle count) and `HI` and `LO` registers.
- MIPS32 ISA also support upto 4 coprocessors named as:

  - `cp0`: System Control Coprocessor. Provides control, memory management, and exception handling functions.
  - `cp1`: Floating point Coprocessor. Load and store instructions (move to and from coprocessor), reserved in the main opcode space. Coprocessor-specific operations, defined entirely by the coprocessor.
  - `cp2` and `cp3` (Custom Implementation)
- The original MIPS architecture not just supports but rather requires delayed branches.[1.c] This creates an ambiguity for our simulator because delay slot would depend on the number of stages we have in the pipeline. This (thankfully) is resolved by MIPS32 Release-6 ISA.
- MIPS stands for Microprocessor without Interlocked Pipeline Stages.

For my simulator, (**Tools and Technology**)

- I have decided to use Release-6 ISA for the processor. The simulator support all the instructions given in the ISA.
- `cp0` System Control Coprocessor is an essential requirement for exception handling, memory management and kernel control. It would thus be included in the simulator. We wouldn't support floating point operations though as I have decided to not include `cp1`.
- I intended the Pipeline to have three stages fetch, decode, execute. Mem and Write Back could be performed as required in the instruction itself while executing. But in order to later be able to implement the concept of clock speed, and variable cycles (for memory and cache

access), I have decided to go with a four stage pipeline - fetch, decode, execute, write back (Mem would still happen during execution) - instead.

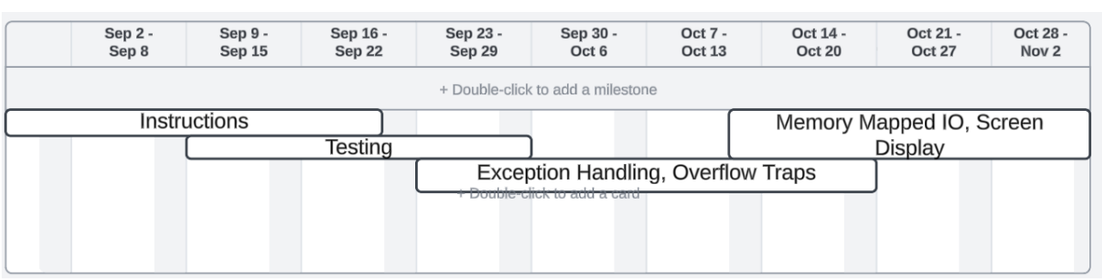
- My memory segment would be of 8 MiB size. (1024*1024 containers of 32 bits width).
- I am slightly confused about memory mapped IO and have come up with two options:
  - import all the input (that would be required while the simulator would run) from a file onto memory and take input from there. It writes output onto memory, that we export into another file at the end.
  - Run a separate program, that's constantly running in parallel with the simulator, that takes input from keyboard and immediately loads it onto a section of memory that our simulator can read. And there is another section of memory where our simulator can write the output, and the program will be constantly reading that memory and displaying that output on the screen.
- Language: C++20 for simulator. Maybe Python 3.9 for generating binary instructions and memory mapped IO.
- Code Editor: VS Code. OS: Windows 11 (ubuntu with wsl-2 if needed). Version Control: Git. Compiler: g++13.2.0.

Things I can do if the implementation goes really well (and fast):

- **Clock:** The notion of clock is only included in the simulator in the way that there is a cycle counter. Initially, I only intend to parse an instruction (irrespective of its complexity) through one stage every cycle. If things do end-up being easier than I think to implement, then I could try giving variable number of cycles to things. Something like loading and storing data from memory would take 300 cycles, whereas from cache it would only take 2-3 cycles. Execute stage might take variable number of cycles for different instructions based on their complexity. This would complicate the project though and would require the processor to have multiple ALUs to allow for simultaneous execution and not clog the pipeline. But it could lead to a much better and realistic understanding of how a processor works.
- **Cache:** How cache works isn't very clear to me beyond the fact that it allows for faster access of frequently accessed objects. But I can try to implement a small 4-8 KiB cache at the end, and understand how it really works in the process.

## Key Milestones/Task List

- Coprocessors, Registers, Memory objects created. A basic process of fetch, decode, and execute ready. (31 Aug)
- Instructions Complete without pipelining, overflow traps and exception handling. (20 Sep)
- Testing Complete: Prepare a few binary programs either directly or by using an assembler (without exceptions). And test the code to see how it performs. (29 Sep)
- Pipelining, Exception Handling, Overflow Traps Complete. (19 Oct)
- Memory Mapped IO, Screen Display systems. (2 Nov)




## Learning Outcome

- I didn't know much about processors at the start of this project, and I have already learned a lot. I have learned how the RAM is managed for a program, how instructions are fetched and

executed. How registers store data, and the many ways instructions can interpret it. I know what kinds of instructions are presented at the assembly and binary level now.

- I expect to have better understanding of how a processor functions, how interrupts, and handling of exception work, and what do coprocessors handle by the end of this project. I also expect to learn more about Memory Mapped IO, and learn and implement exactly how heaps and stacks work in memory.

## Reference


1. MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture Revision 6.01
    a. Chapter 2: Overview Of the MIPS Architecture. 2.1 Historical Perspective [Pg 21]
    b. Chapter 5: CPU Instruction Set.
    c. Chapter 5: CPU Instruction Set. 5.3 Jump and Branch Instructions [Pg 61]