Interim Project Report

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

From past feedback

One of the Key questions posed in the feedback for the proposal was the concept of the clock in my simulator. So let's first address that,

Clock Cycle

The simulator employs a synchronous clock to regulate the execution of instructions. Each clock cycle represents a discrete unit of time.

Pipeline Stages:

The pipeline is divided into three stages:

1. **Fetch and Decode:** During this stage, the next instruction is fetched from memory and decoded to determine its operation and operands.
2. **Execute:** In this stage, the decoded instruction is executed. This may involve arithmetic operations, data transfers, or control flow decisions. The register read or write also occurs in this stage as the instruction requires.
3. **Memory:** If the instruction requires memory access (e.g., load or store), this stage performs the necessary read or write operation.

Pipeline Synchronization:

In a processor, a clock signal would ensure that all stages operate synchronously. At the rising edge of the clock, each stage completes its current operation and prepares to receive data from the preceding stage. This pipelining allows for efficient instruction execution by overlapping the stages of different instructions.

Fixed-Cycle Execution:

A key characteristic of this simulator is that each pipeline stage takes exactly one clock cycle to complete, regardless of the complexity of the instruction.

This simplifies the design and implementation of the control logic.

This description raises a question regarding how the data would flow from one stage to another.

Datapath:

The Datapath consists of three channels:

- **Decode Channel:**
  - Data bus: Carries 32-bit instructions.
  - Control bus: 2 bits
    - **Reserved Instruction:** Indicates if the previous instruction was a branch. This allows decode to signal a reserved instruction exception while decoding the current instruction if required.
    - **Actually Branching:** Indicates if the execution of branch instruction led to a successful change in program counter in this cycle. Decode will discard the current instruction instead of writing it to the execution channel if yes.
- **Execute Channel:**
  - Includes a function pointer for instruction execution.
  - Has an 8-bit control bus (6 bits used) specifying line usage.
  - Contains a 64-bit data bus divided into:
    - Four 8-bit lines. These are the least significant bits of the bus.
    - One 16-bit line. This isn't a line. We combine the data from the last two 4-bit lines if indicated yes by control.
    - One 32-bit line. These are the most significant bits of the bus.
  - Note: This data be carried by a 32-bit bus as well if encoded by the control bus efficiently. But for simplicity's sake, I have currently implemented only a 64-bit bus.
- **Memory Channel:**
  - Data Bus: Holds a 32-bit memory address.
  - Control Bus: Contains an 8-bit control bus and an additional one-bit data line:

- **Read or Write (1 bit):** Indicates read or write operation.
- **Width (2 bits):** Specifies the data size (1, 2, 4 bytes).
- **Register Address (5 bits):** Specifies the register for reading or writing.
- **Sign Extend (1 bit):** Indicates whether to sign-extend the read data.

Control:

The control operates in a pipelined fashion, executing instructions in three stages: fetch and decode, Execute, and Memory. Each stage takes a single clock cycle, regardless of instruction complexity.

The control section follows these steps during each cycle:

1. **Memory Stage:** If the memory channel is non-empty, perform the specified store or load operation.
2. **Execute:** If the execute channel is non-empty:
    - Execute the instruction using the function pointer.
    - If the instruction is a store or load, push as required onto the memory channel.
    - If the instruction is a branch that needs to be taken, set the "actually branching" flag in the decode channel to indicate that the next decoded instruction should be discarded.
3. **Fetch & Decode:**
    - Fetch instructions from the memory.
    - Decode the instruction.
    - If the instruction is a branch, set the "reserved instruction" flag to indicate that the next instruction will be in the forbidden slot.
    - If the reserved instruction flag is up
        - Signal exception if the instruction happens to be one of the forbidden ones.
        - If the actually branching flag is up. Discard the decoded instruction. Don’t push it into the execution channel. Instead Modify the PC value to be as indicated by the execution channel, and fetch and decode again.

- End the cycle.

Flushing: To ensure proper pipeline operation, stages are flushed in reverse order, clearing channels for the previous stage.

The above detail also addresses the recommendation to not have a single-stage pipeline. However, this led to a change in objectives and deadlines than what was planned in the original proposal. So let's head to first see a brief overview of the progress and then we will see the effects on the timeline.

Progress Overview

Writing functions for instructions described in release-6 ISA are essentially complete. I have not yet implemented the instructions that require system control or cache clearing. Exception handling, Overflow traps, and other such things have also been left out of the functions for now.

It should also be noted that the instruction list that I obtained, while being vast might not have been exhaustive and might not completely cover the release-6 ISA. I have only covered the main instructions as described in MIPS Reference Manual Volume-I A. An Exhaustive list of the instructions however is given in Volume II A. I might try to crosscheck to see if I am missing any instructions later.

I have mostly completed the implementation of the Datapath, Control flow with the Pipeline as described above. The implementation of the Fetch & Decode stage however is not fully complete.

Objectives Achieved

The objective as laid out in the proposal for 20 September has been achieved:

- Instructions Complete without pipelining, overflow traps, and exception handling. (20 Sep)

The reworking of the first objective before this is ninety percent complete. It should be complete by Sunday.

- Coprocessors, Registers, and Memory objects created. A basic process of fetch, decode and execute ready. (31 Aug)

Plan for Next Stage

As of yet, I am not changing the plan for the next stage. As one week should be more than enough to test my code.

- 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)

The next deadline set on 19th October however is slightly shaky.

- Pipelining, Exception Handling, Overflow Traps, Implementation of Coprocessor 0 for System, Memory and Kernel Control. (19 Oct)

This brings us to the next point given in the feedback. I was advised to reconsider my plan to include a Coprocessor in the simulator. My focus until now has primarily been on the main MIPS32-Processor and implementing it in code. Thereby I do not completely understand the details of how `cp0` works, and how to implement it.

I still very much want to implement `cp0` as it seems to be an essential part of the functionality of the MIPS32 processor. But if I do not have enough time at the end, or I find its implementation to be a little too complex, I won't implement it. As for `cp1`, I don't think is that essential for the processor, and my simulator should work just as well without floating-point operations. So I will not be implementing that.

Lastly, the timeline for memory-mapped IO remains as it is.

- Memory Mapped IO, Screen Display systems. (2 Nov)

References

1. MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture Revision 6.01
2. MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set Manual Revision 6.01