MIPS1 CPU Coursework Data Sheet – Team 09
ELEC50010 Instruction Architectures & Compilers

1. Overview of the Architecture and Design

The MIPS1 CPU implemented in this project is a RISC (Reduced Instruction Set Computer) processor with a load-store architecture (instructions are divided based on memory access and ALU operations) using the Harvard interface (separate memory units for instructions and program data) targeting the instruction set encoded in 32-bit little-endian MIPS1, as defined by the MIPS ISA Specification Revision 3.2 with naming conventions defined by the MIPS O32 ABI.

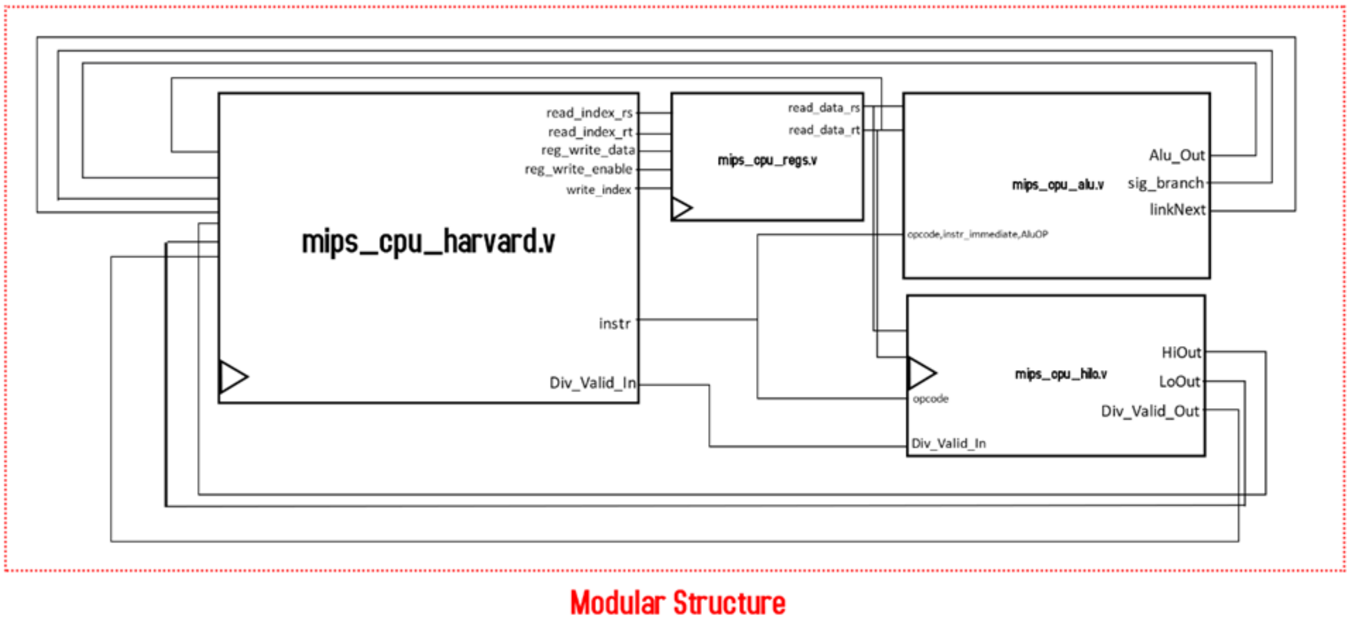


Figure 1: CPU RTL Modules

The CPU deliverable comprises the following modules as shown in Figure 1:

a. <u>mips_harvard_cpu:</u>
The top-level entity for the CPU. It handles inputs and outputs, decodes instructions and controls other modules.

b. <u>mips_cpu_ALU:</u>
The Arithmetic and Logic Unit (ALU), which combinatorically performs addition, subtraction, shifting logic operations as well as comparisons for conditional branch operations.

c. <u>mips_cpu_regs:</u>
A general-purpose register file, comprising thirty-two 32-bit general purpose registers.

d. <u>mips_cpu_hilo:</u>
The module in charge of sequential division and combinatorial multiplication, using two separate 'HI' and 'LO' registers, which allow for numerical results longer than 32-bits to be obtained.

The Harvard architecture was chosen as the target interface because of its explicit architectural advantages such separate instruction and data memory units and buses, increasing predictability of the memory bandwidth (no scheduling involved), therefore explicitly presenting a solution to structural hazards if the CPU was to be pipelined (as a future extension) to improve performance by increasing instruction throughput by achieving a higher clock rate. Figure 2 shows the I/O interface connected to the Harvard CPU.

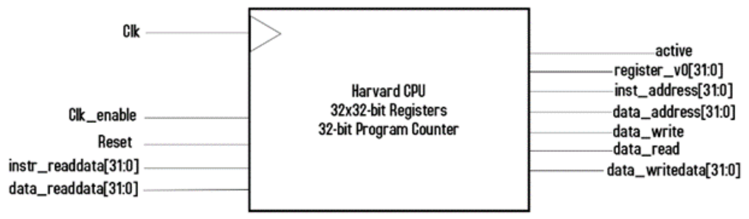


Figure 2: Harvard I/O Interface

2. Design Decisions

a. Multi-cycle approach - 3 stage approach involving FETCH, DECODE, EXECUTE:

Unlike single-cycle MIPS CPUs, this CPU uses 3 states – FETCH, DECODE, EXECUTE (for instructions other than DIV and DIVU) to have a sufficiently high clock rate while maintaining a reasonable CPI. The control signals and registered immediate are implemented in a way to promote pipelining (as an extension task), ensuring all control signals are ready at each clock cycle and reducing the risk of structural and data hazards. The multi-cycle approach also benefits the implementation of the byte-enabled store instructions, which loads the value from memory at one cycle and on the next cycle, bit-masking is used to isolate and modify certain parts of the loaded word with values from a register.

b. Combinatorial multiplication and sequential division implementation:

The multiplier in this CPU uses combinatorial logic to perform single cycle signed and unsigned multiplication using the System Verilog multiplication operator (*) which is synthesisable and generates the optimal hardware implementation in the FPGA. However, since the division operator is not always synthesisable and often has an inefficient hardware implementation, division in this CPU is performed through consecutive shifts and subtraction. The division unit (part of mips_cpu_hilo) uses sequential logic to perform signed and unsigned division on two 32-bit integers and executes for 33 consecutive cycles, in addition to a single cycle setup phase where the values are initially loaded into the division unit.

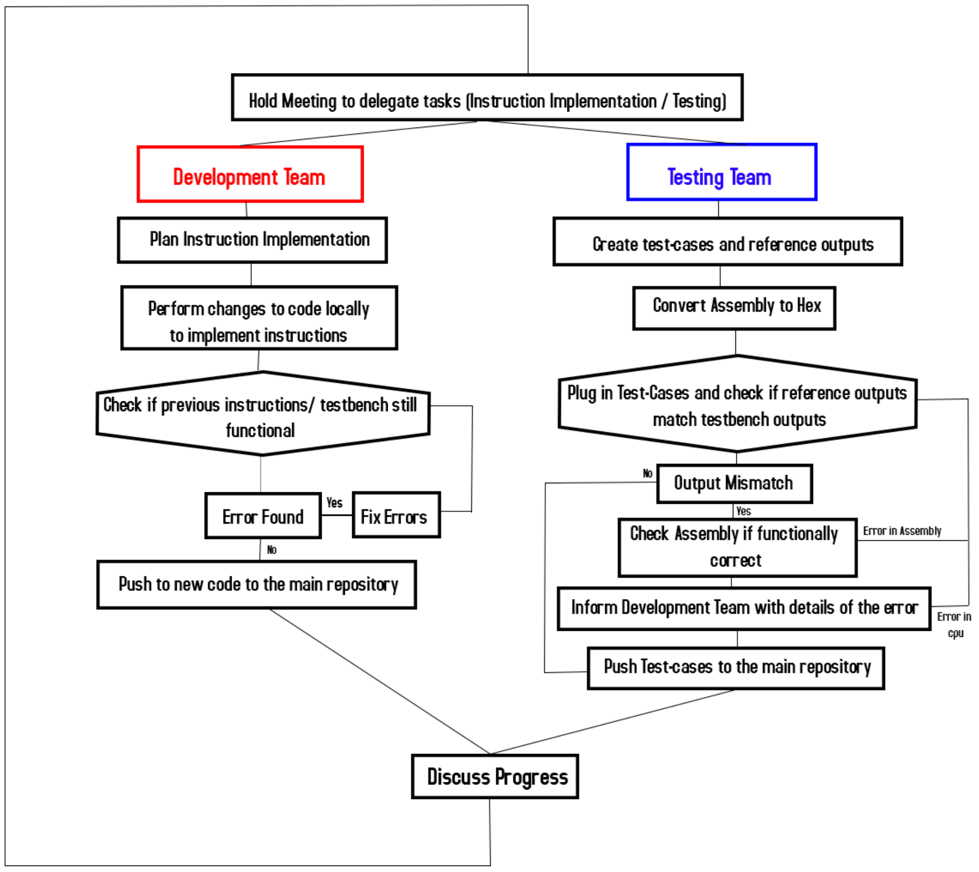
c. Branch delay slot implementation using control flags:

The branch delay slot is implemented by using two flags. The first flag is to check whether the current instruction is a jump/branch instruction and store the branch address if the conditions are fulfilled. On the next instruction, the second flag checks whether the first flag is asserted and then the Program Counter jumps to the branch address.

d. Handling branch instructions in the ALU:

The ALU unit also contains functionality for logical operations and thus is used to perform logical comparison to evaluate the branch conditions for branch instructions. The ALU relays a branch signal, which indicates if the branch condition is fulfilled, to the main CPU and this determines whether a branch occurs or not.

3. Testing Approach



Hold Meeting to delegate tasks (Instruction Implementation / Testing)
Development Team
Testing Team
Plan Instruction Implementation
Perform changes to code locally to implement instructions
Check if previous instructions/ testbench still functional
Error Found
Yes
Fix Errors
No
Push to new code to the main repository
Create test-cases and reference outputs
Convert Assembly to Hex
Plug in Test-Cases and check if reference outputs match testbench outputs
No
Output Mismatch
Yes
Check Assembly if functionally correct
Error in Assembly
Inform Development Team with details of the error
Error in cpu
Push Test-cases to the main repository
Discuss Progress

Figure 3: Development and testing approach flowchart

a. <u>Testing the CPU module:</u>

While attempting to fix erroneous test-cases, the first step is to check whether the test-case is functionally correct. The testing team traces the execution of this test-case by executing the assembly in the VISUAL-MIPS Emulator[1] to generate an accurate reference output. If the test-case is proven to be functionally correct, the hex files are checked for correct conversion from assembly to little-endian hexadecimal values. If no issues are found within the test-case, the testing team informs the development team to inspect the CPU modules.

Debugging the CPU is accomplished by tracing the instructions executed by monitoring the Program Counter value and control signals. The first check to be performed when implementing new instructions, is to verify whether the instruction has been decoded properly by monitoring the opcode part of the instruction and the opcode case section of the code. For branch and jump instructions, this identifies improper jumps or unfulfilled branch conditions. For arithmetic instructions, debugging takes place in the ALU, where the value of intermediates is sent to the output. Division uses a more complex debug mechanism as it is the only sequential operation

[1] *Visual MIPS | Online MIPS Emulator [online] Available at: <https://visualmips.github.io/>*

taking multiple cycles, and so it is tested by checking the remainder and quotient values at each iteration to see if it performs as expected.

b. Testbench and test scripts:

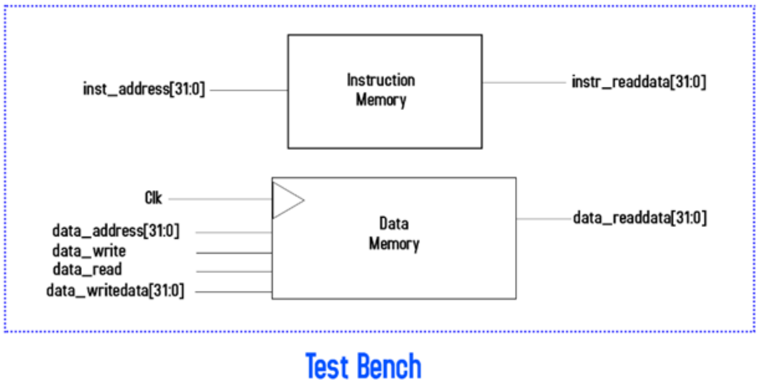
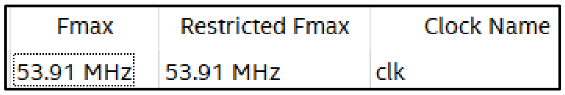


Figure 4: Test bench memory interface

The testbench has multiple checks before the CPU begins execution to affirm that the initial state and I/O of the module behaves as expected. These include checking for the ACTIVE signal and ensuring no memory transactions have been issued by the CPU while it is being reset. While the test-bench is running, it checks for unusual behaviour such as if the CPU is trying to read and write from memory simultaneously, or if the CPU had entered an infinite loop and has timed out. After execution, the test-bench compares the outputs of the CPU, in this case the value of register_v0, to the reference outputs provided. If any of the checks fail, the testbench halts the execution of the CPU and reports the failed test-case with appropriate comments to describe the issue (where applicable).

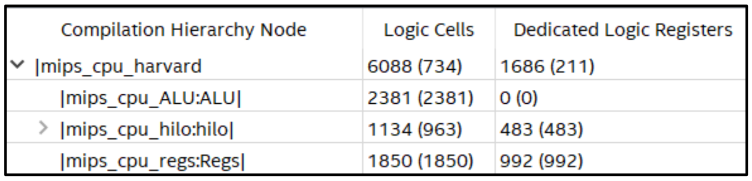
4. Area and Timing Summary

With respect to the worst-case working conditions, defined to be 1200mV at 85C, the CPU reaches a maximum clock frequency of 53.91 MHz with the most significant delay, that is, the critical path arising from the 'hilo' unit, which implements multiplication and division, which is expected as the multiplier has the longest propagation delay with an area-time product of $O(N^2)$. The total number of logic elements present in the CPU is 6088, with 4402 being combinatorial with no register, 478 being register only and 1208 being combinatorial with a register.



Fmax	Restricted Fmax	Clock Name
53.91 MHz	53.91 MHz	clk

Figure 5: Timing summary (clock rate)



Compilation Hierarchy Node	Logic Cells	Dedicated Logic Registers
|mips_cpu_harvard	6088 (734)	1686 (211)
|mips_cpu_ALU:ALU|	2381 (2381)	0 (0)
|mips_cpu_hilo:hilo|	1134 (963)	483 (483)
|mips_cpu_regs:Regs|	1850 (1850)	992 (992)

Figure 6: Area summary (logic cells for each module)