Design and Architecture

The overall design decisions made to develop this CPU are based on the following guidelines:

1) A client can easily understand the purpose of the CPU
2) The ability for further adaptation of the CPU by a different developer
3) A compartmentalised CPU allows developers to pinpoint the bugs and logic errors.

5-cycle Design

The MIPS CPU has been designed to execute these five cycles + Halt:

Fetch (S_FETCH)	The CPU fetches the next instruction from memory (RAM) at the address pointed by the program counter (PC), and it stores it inside the instruction register.
Decode (S_DECODE)	The instruction word is decoded, and any operands required are read to execute the instruction.
Execute (S_EXECUTE)	The CPU begins to execute its instructions in this cycle. Furthermore, the ALU operations are computed, including multiplication, division, branches etc.
Memory (S_MEMORY)	Data in the memory is accessed in this stage. Load instructions will read data from memory, and store instructions will write data into the memory.
Write Back (S_WRITEBACK)	This cycle will write the result of the instruction into the destination register.
Halted (S_HALTED)	Indicates a definitive endpoint to the CPU to know that all the instructions have been executed.

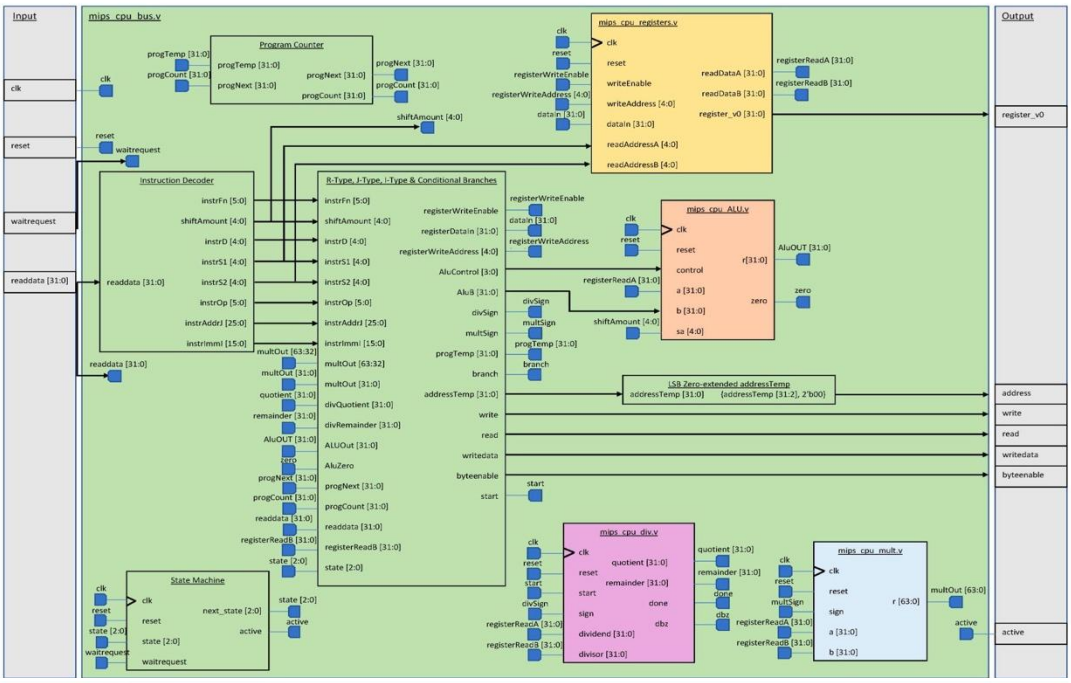

Figure 1: CPU Diagram

mips_cpu_bus.v

The *bus.v* file is designed to handle the MIPS instruction set architecture and manage communications with other blocks. One of our focus points when designing it was readability. Therefore, all the states and instructions are annotated in the *bus.v* file for the reader's understanding. Besides that, we also wanted our CPU to be scalable and extensible. So, instructions are processed on a case-to-case basis, with explicit partitioning on each instruction type (R, I, or J). As a result, this eases the modification or addition to the current code, making any development on our design possible.

mips_cpu_registers.v

The register file is designed to write and read from 32 registers of size 32 bits. When reset is high, all the registers will have their values set to zero. The register file also operates on the positive edge of the clock, and the value stored in the register will be changed if the write enable is high and the write address is non-zero. The register_v0 has been fixed to the second register for testing purposes, and the register [0] is always set to zero; values written to it are ignored.

mips_cpu_ALU.v

After reading through the specification sheet and looking at all the instructions needed, we have decided that these operations are sufficient to carry out all the necessary instructions.
These operations are as shown:

Operations	**Definition**
Bitwise AND	Output = A & B
Bitwise OR	Output = A | B
Bitwise XOR	Output = A ^ B
LUI	Output = B[15:0] + 16 Zero Extended Bits
Addition	Output = A + B
Subtraction	Output = A - B
Set Less Than	Output = (Signed (A < B)) ? 1 : 0
Set Less Than Unsigned	Output = (Unsigned (A < B)) ? 1 : 0
Shift Left	Output = B << ShiftAmount
Shift Right	Output = B >> ShiftAmount
Shift Left Variable	Output = B << ShiftAmountVariable
Shift Right Variable	Output = B >> ShiftAmountVariable
Shift Right Arithmetic	Output = Signed(B) >>> ShiftAmount
Shift Right Arithmetic Variable	Output = Signed(B) >>> ShiftAmountVariable

mips_cpu_div.v & mips_cpu_mult.v

These two blocks are designed to handle signed and unsigned division and multiplication efficiently. To check for a signed integer, we look at the value of the variable sign and the 31st bit of the inputs. If they are both 1, the input is a negative number.
The division block has been designed to execute its instruction on the worst case of 32 cycles. When we divide by zero in Verilog, an error will be produced. We created a particular case to account for division by zero to fix this. This will set the quotient and remainder to zero. The 32-cycle circuit also reduces circuit complexity and critical path length.

Testing

In order to run our tests efficiently, we coded an assembler in python that converts the assembly code we write in a text file into a hex RAM file which is what the testbench reads as the RAM for any particular instruction. We also calculated by hand what the outputs of each test case would be and included in the assembler a function that read that value and created a reference file for us to compare against the last few lines in the testing output so we can confirm whether each instruction either passes or fails.

We have decided to split our testing approach into two ways: the CPU and its instructions. We tested the CPU to ensure that it functions as expected because all the instructions depend on the CPU. We did this by making the testbench write to our output files when there was an error so it didn't stop the program from running and allowed us to write our own additional outputs to help locate what is causing the errors. For example, after resetting the CPU, the active signal should be high. Furthermore, the CPU can either be reading or writing at a given point in time, not both. The CPU should also write/read from the correct address in RAM. If the CPU fails to halt within 10,000 cycles, the testbench will abort and exit with failure code 2. These functionalities and more have been thoroughly tested in the files *mips_cpu_bus_tb.v* and *mips_cpu_bus_tb_mem.v*.

Following the specification, the CPU's testbench should ideally test the functional correctness of every instruction that is included in the CPU. As such, we determined that the most efficient method was to use a standard testbench for all instructions, which compares the output of the CPU and a reference output. This allows us to test all instructions case-by-case in a relatively short amount of time.

To assert functional correctness of the instruction, the two CPU outputs are checked:

1) The register v0 output
2) Active signal

When creating test cases, we ensured at least 2 for each instruction to make sure to test any edge cases that could cause errors. In conjunction, the output value of register_v0 is always set to a non-zero value to ensure that data is always changed. Furthermore, there were more test cases for big/little-endian instructions like Load Byte because of the number of various implementations available, and we made sure to check all the possible cases for functional correctness. Finally, to aid in debugging instructions, there are outputs in every state of the *bus.v* file showing the corresponding data in each state. As a result, this allowed us to identify where our errors were quickly.

The test script utilises 185 pre-assembled test cases and reference outputs, each testing a variation of an instruction loaded into the RAM that the CPU interacts with as memory. As a result, this produces our testing workflow, which looks like the diagram below:

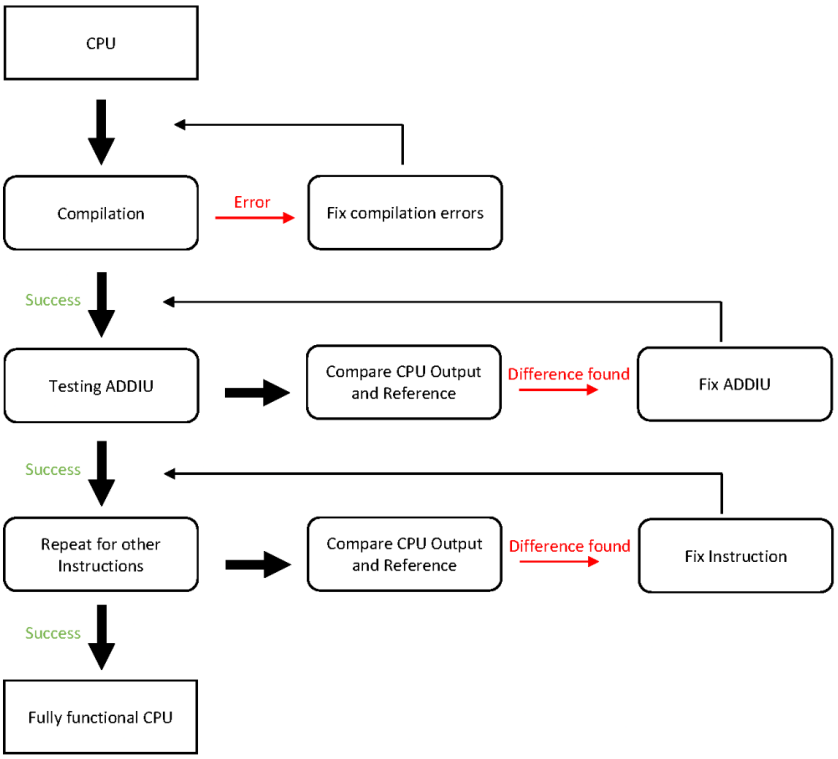

Figure 2: Testing Flow Chart

When the test script ends, each test case will produce a result indicating whether it has passed or failed. When a test case has failed, we begin to debug the instruction and repeat for all other failed test cases.

Area Summary

Estimated Total logic elements	5,147
Total combinational functions	4155
Dedicated logic registers	1464
I/O pins	138
Embedded Multiplier 9-bit elements	16
Maximum fan-out node	clk~input
Maximum fan-out	1464
Total fan-out	19893
Average fan-out	3.37

Timing Summary

Slow 1200mV 85C Model	63.19MHz
Slow 1200mV 0C Model	71.19 MHz
Fast 1200mV 0C Model	108.34 MHz

Of the timing summary, the most realistic clock timings are the two slow models due to them accounting for the worst case time and therefore allowing the clock to run with all instructions and of the two, the one at 85C will be more accurate still as computers heat up naturally so will always be slower than the 0C model. These timings are the best effort time results based on the available information.