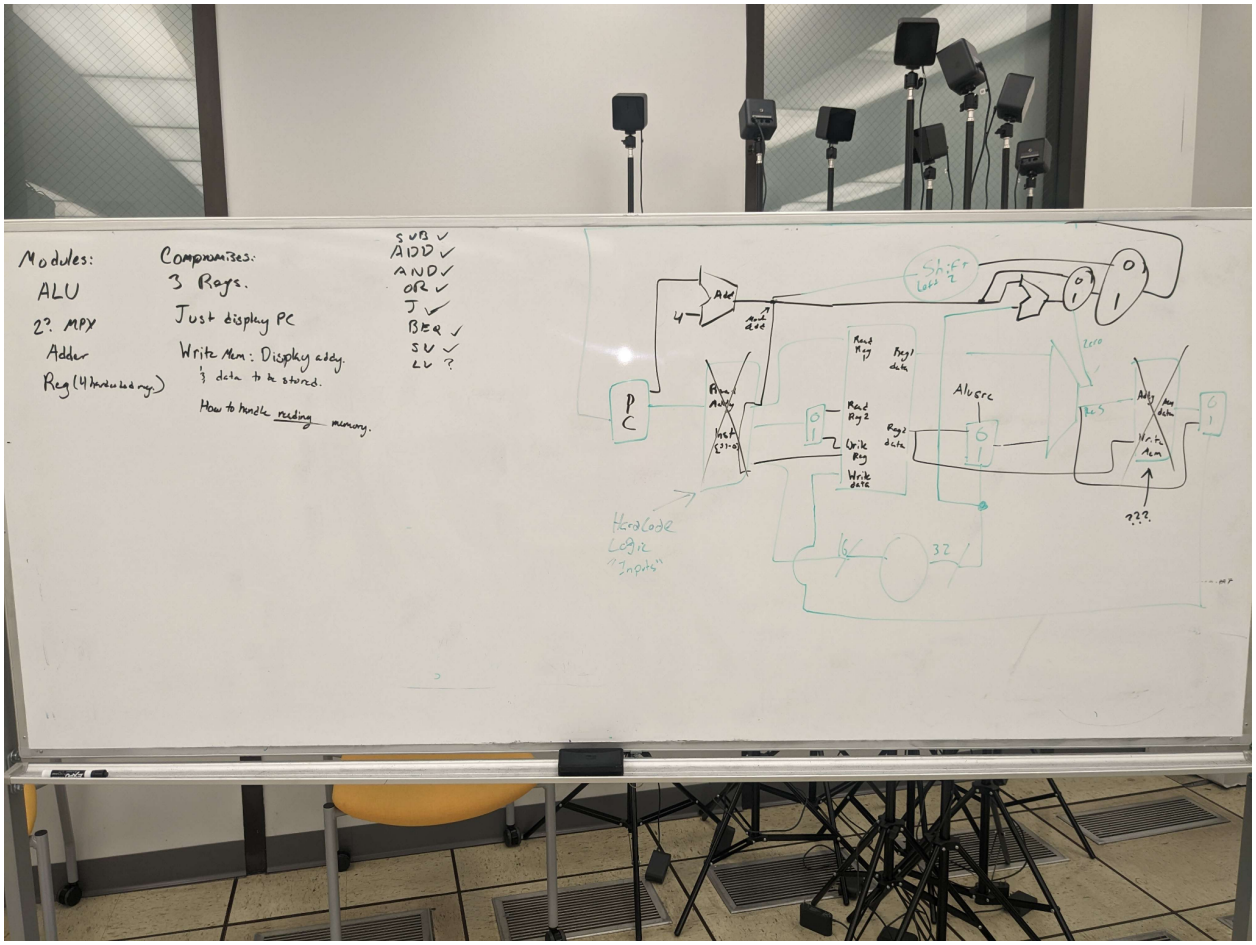
Vincent Lee

Brandon Chao

Brenden Larios

Project 1: Certified PreOwned Processor

Project Planning:



Before using someone else's source code, we wanted to try to think of the implications of implementing MIPS architecture in HDL. We knew from last time, that we would need some sort of top module to supply inputs to all other sub modules, and it was clear that that was the control. Control would need an instruction supplied from the instruction memory, however at the time we were unaware that Verilog could access any sort of memory, so to circumvent this problem, we

assumed that control would be sent an entire instruction via the top module, and that it would supply the initial values for certain sub modules in addition to the necessary boolean flags.

For the sub modules, we considered making the ALU, Multiplexors, Adder, and Register File their own modules. The ALU was the simplest of them all consider we had covered that subject in Project 0. The multiplexors were rendered obsolete after we learned about the existence of ternary assignment and if statements in Verilog. An adder would be the simplest module to implement, but upon later consideration, we realized that addition can be handled in Verilog with a simple line of code after which we can conditionally set PC based on the instruction. The register file implementation was something that we pondered for a reasonable amount of time. We didn't want to implement every single MIPS register, we decided to cut the amount of registers down to exactly three because of the regularity of MIPS instructions needing at most 3 registers.

The major setback that we were unable to solve was the access of memory and how to simulate that in Verilog. Again, at this time we were unaware of the ability to access ram memory, so this problem seemed insurmountable unless we considered an "address" to be a line of an external file and we'd just read from that file. In a similar vein, store word instructions would be impossible to implement without memory, but our solution was to simply add a testbench output declaring what value was stored in which memory address, there would be no need to actually store it. No similar work around would work for load word unless we also wrote that value to a file with an associated address, but it was too convoluted of an option to consider.

Now that we attempted to ponder the problem ourselves, we turned to online source code to see how they implemented the single cycle instruction set. Our findings are as follows…

Findings:

- We learned that Verilog can access memory and write to it
- We understand the control unit of a single cycle implementation and which bits to set to acquire a specific operation.
- We obtained significant practice converting hex to binary and vice versa
- By looking at the GTKwave output, we now definitively understand that many values are calculated but are never used, depending on the operation.
    - For example, if we do a simple add instruction, the jump address is still calculated, but never used
- We now know how to adjust the *timescale* on verilog, and the precision that GTKwave can read the waveforms.
- We learned that despite files being in the same directory, the keyword *include* was necessary for them to reference each other.
- We learned that you can use ternary assignment for *assign* statements in Verilog