CompArch Lab 3: Single Cycle CPU

Allison Busa
Jordan Crawford-O'Banner
Shashank Swaminathan

November 6, 2019

Processor Architecture

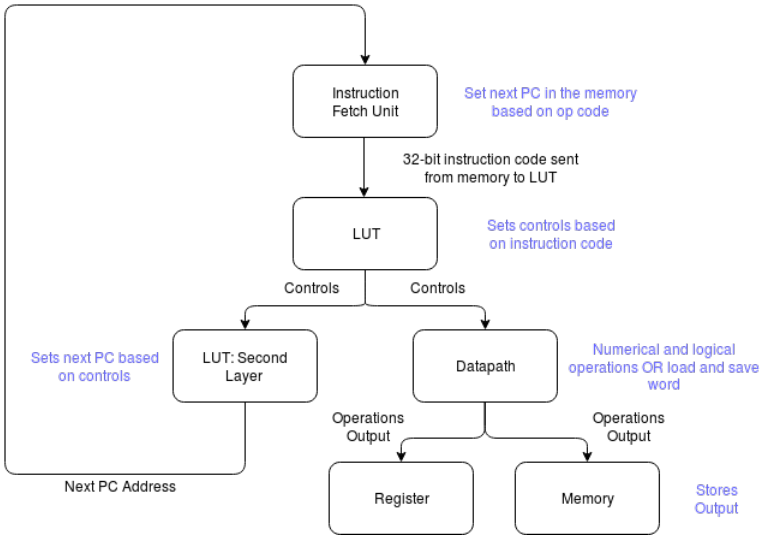


Figure 1: Block Diagram

The processor is a single-cycle CPU, meaning the processor's main functions all occur within a cycle - between two successive positive edges of the clock. Here is a breakdown of the operations happening within such a cycle:

1. The Instruction Fetch Unit (IFU) provides the address of the current instruction in memory via the Program Counter (PC).
2. The 32-bit instruction code is retrieved from the memory, and is sent to the Lookup Table (LUT) to be parsed.
3. The LUT parses the instruction code, and sets all the relevant control inputs in the data path.
4. The data path executes the operations specified by the LUT's controls. The IFU determines the next PC - the address of the next instruction.
5. The output of the data path's operation is written to either the memory or the register. The values within the register/memory are only updated on the next positive clock edge.
6. On the positive edge of the next clock signal, the register/memory is updated, as is the PC. The cycle repeats.

The IFU also takes a reset signal input; when the signal is on, the PC is set to 0 (the beginning of the instruction list).

Testing

Test plan

For our Single-Cycle CPU, we first created individual component tests. By testing each component, such as the regfile, ALU and IFU, we were able to catch design flaws much earlier, and easily debug the root causes of errors. Had we decided to simply put the CPU together and run the final tests, the errors would have been near impossible to decode, due to the sheer complexity of the implementation.

Some of the modules we used, we imported from previous assignments. Specifically, our ALU, n-bit registers and multiplexer, adders, and 32-by-32 bit regfile, were all from previous labs and homeworks. Since we had tested these modules successfully in the past, we felt that we did not need to test them again. However, along with the actual module code, we also imported their associated test benches.

On the other hand, we created the other modules. We created the IFU, LUT and the whole CPU assembled together. Since the IFU was an isolated component, we could not have tested it with the assembly code, and instead created a Verilog test bench. The test bench consisted of six targeted tests, instead of a suite of exhaustive tests, because it is more time effective and less computationally expensive. We tested for a normal operation, in which the program counter just increments by 4. This would happen for instructions with an immediate, for example. We also tested for instruction controls that interact uniquely with the IFU. Specifically, we tested for `JAL`, `J`, `JR`, `BEQ`, and `BNE`. We did not test the LUT, because it just equates one command to a series of other commands; there is no computation being done. Our final tests, in which we tested the overall performance of the CPU, were based on assembly code. We had a series of tests which were based on individual operations, and then we had two larger tests which were more involved. A description of the assembly tests can be found below:

Assembly Code Test Benches

We had three suites of assembly test benches used to test the single cycle CPU. They were as follows:

- Basic Op Tests
  - Here, we tested each individual operation of the CPU, in the following order:
    1. `ADDI`
    2. `ADD`
    3. `SUB`
    4. `J`
    5. `BEQ`
    6. `BNE`
    7. `SLT`
    8. `SW`
    9. `LW`
    10. `XORI`
    11. `JAL/JR`
        (a) Due to the nature of the JAL and JR instructions, and their use cases, the tests for these two operations were combined into one overall test.
  - The tests were run in this order, since many instructions later in the list required previously tested operations during the setup (ex. in order to check equivalence between registers `$a0` and `$a1`, they need to be initialized via `ADDI`).

  - These tests were not expansive - apart from the `JAL/JR` test, they did not test many other operations apart from the desired op. However, they did let us check if the specific operation was working as intended in the CPU. Because the tests were so small and focused, it also reduced the debugging workload.
  - These tests exposed many minute issues in our hardware implementations, both in modules imported from previous assignments and from modules created solely for the CPU. As there were many of these issues, they are discussed separately in the next sub-section.

- Fairly complex code test - finding the GCD between two numbers
  - This was a more expansive test that checked how the CPU worked with multiple operations. The test involved implementing a technique of finding the greatest common denominator of two numbers by continually subtracting the smaller number from the largest number until result of the subtraction is 0. This test uses `ADDI`, `SUB`, `J`, and `BEQ`. The test functionally tests all of the different types of MIPS commands and many of the commands that can be used most often in functions, which makes it very effective for testing the functionality of our CPU.
  - This test passed with flying colors (after all of our basic ops worked)!

- Very complex code test - recursive in-place insertion sort of an array in memory
  - The purpose of this test was to cover ground on multiple fronts:
    * Using all the operations in one test.
    * Using loops, recursion, and the stack pointer.
    * Multiple loads and saves to and from memory.
    * Writing a fun and involved assembly function!
  - As indicated by the test name, the test was an insertion sort algorithm that sorted an array in the memory in place, recursively. It used a loop to initialize the array, and then used recursion to move a pointer through the array and sort bottom up (from the end of the array to the front). At the end, it looped through the array once more to ensure that the elements were in ascending order. Because the recursion algorithm didn't actually end up using the `XORI` operation, the final checker used it to verify if the algorithm worked.
  - As we were already checking the CPU's behavior with each operation in the basic op tests, and how the CPU behaved for relatively complex algorithms in the GCD test, this test served really to confirm that our CPU scaled well from our tests to actual full-blown algorithms. It turned out that this test also covered some additional ground - it used multiple `SW` commands. Through that, we realized that we had RegWrite and MemWrite enabled at the same time - something we couldn't catch with a simple one-op `SW` command.

Errors found during testing

- ALU
  - We had imported our code for the ALU from a previous lab as is. Because the deliverables the ALU was then addressing was slightly different than the desired output for the CPU, a few errors sprouted from the mismatch.
  - For example, our outputs were still all `X`s or incorrect outputs. When we looked at GTK wave, we saw that the Evaluation stage was taking longer than it should. That's when we realized, that our ALU and its subcomponents had delays on the logic gates, which were messing with our design. We took the delays out, and our timing issue was resolved.
  - Our SLT implementation only set one bit of the 32-bit ALU output, since we only checked the LSB of the ALU output during the SLT tests. However, the CPU looked at the entire 32-bits, and the unset bits gave us `X`s on the output. To fix that, we set the other 31 bits to zero - the standard output of an SLT implementation.

- Aside from the errors above, we also got some pedantic Verilog errors. Our CPU testbench failed to compile initially, and when we looked into the Verilog error statements, we saw that IVerilog no longer treated the code we used to implement the ALU as perfectly valid, for unknown reasons. Updating our code structure to match IVerilog's requirements solved that error.

- Instruction Fetch Unit (IFU)
  - At first, the `JAL` test which we created in our IFU testbench was evaluating to false. We figured out that this was happening because we were running the testbench in a way that wasn't representative of the actual behavior. Before, we had set all the inputs on the negative edge of the clock, and evaluated the outputs on the positive edge of the clock. Hence, the `JAL` testbench wasn't working. We reconvened and determined that the instructions should be set on the positive input of the clock, and then the PC would be set on the next positive edge of the clock. Here is a diagram of the new structure of the testbench:

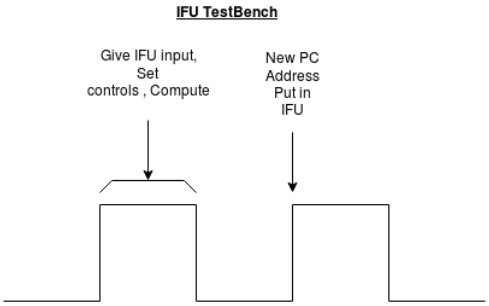
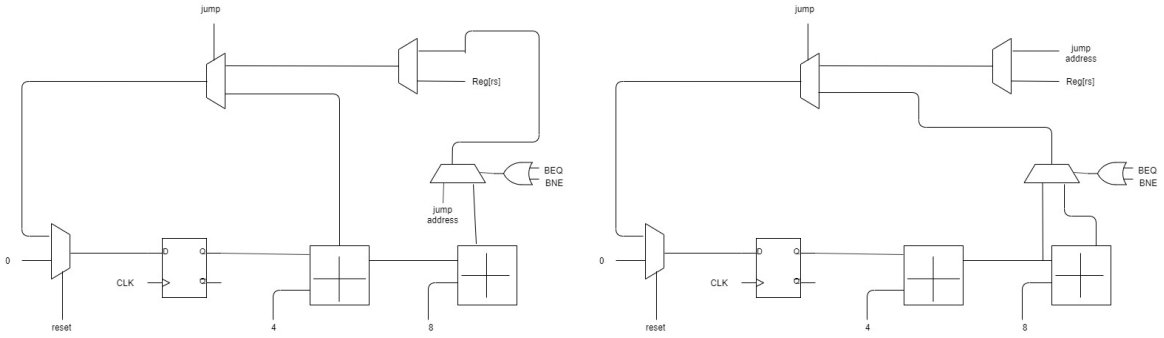


Figure 2: Block Diagram

  - For our `JAL`, `JR`, `BEQ` and `BNE` testbenches in our overall CPU design, we found that the PC was jumping to the wrong address. This is because MIPS increments by 1 byte, not by 4 (the length of a word). So, with the way we were implementing the code before, the PC was incrementing instructions inside of a word, instead of going from word to word. The way that we solved this is that we left-shifted our branch and jump address by two. This is equivalent to multiplying by 4 in binary. Thus, when we left-shifted, the instructions were incrementing by word, and our bug was fixed.
  - When we were testing the `BNE` and `BEQ` commands in the assembly tests, we came upon a scenario that we hadn't considered. Before, we had a multiplexer to decide whether or not to let the jump address pass, or to let branch address + 4 + PC pass. We were controlling the multiplexer with the output of OR between the input `BNE` and `BEQ` signals. What these signals were supposed to represent is if `BNE` and `BEQ` were the current commands and relevant. Before they were input into the PC, they were created as the `AND` between the zero output and the `BNE/BEQ` signal from the LUT. What we hadn't considered was if `BNE` or `BEQ` were true, but the output of `zero` wasn't correct. For example, if you are checking if two values are equal, and they're not- then the `zero` output is not true and the output of the `BEQ` signal is true. `AND`ed together, these signals are false, and thus the input signal into the multiplexer controlling the inputs in the IFU are set that they should be outputting the jump address (instead of branch address + PC + 4, as he should be). The way that we fixed this is that we switched the wires and moved the multiplexer. The old and new designs can be seen below.

- Regfile
  - When we first started testing the overall CPU, we tried to run the `ADDI` command. This initially failed due to the PC counter incrementing issue discussed in the IFU errors section above. However, after we fixed that error, we realized that the output of the test was still `X`s. After looking at the signal traces in GTKWave, we realized it was because the zero register was outputting `X`s. Further inspection showed that our `zero` register was initially unset, and would only set the



(a) Old IFU Design

(b) Current IFU Design

Figure 3: Simulation Results before Optimization: Simple and Complex Models

output to zeros after being sent a write enable signal - which happened in our test bench, but not the CPU. We then fixed it to always output zero, regardless of write enable, and that fixed the issue.

- Memory/Datapath
  - When we were testing our CPU, the testbench would continuously produce an error because the memory unit was receiving odd values for the address. This was caused by the fact that the ALU was directly connected to the memory unit. We did not believe that this would be an issue because we had set the write enable for the memory unit to 0, but we had used the memory unit that was provided to us, which would automatically fail the test when any inappropriate address was passed to the memory unit. In order to avoid this, we added a multiplexer to decide whether the memory address should receive the output of the ALU or a dummy address for when nothing is being stored in the memory unit.
- LUT
  - We had a few issues with our LUT for decoding instructions. Most of these amounted to small errors where some of the controls were accidentally set to the wrong value, or where we had not realized that a certain control had an effect on a command. We fixed these by simply switching values within the LUT based upon the output we expected from the GTKwave.

Performance

For the performance analysis of our circuit, we decided to analyze the area and delay of the IFU. In exploring the area of the IFU, we decided to use the model that determines area by the number of inputs into each of the individual gates.

By using the area analysis in previous labs, we found that the area of each multiplexer is 320, each adder is 483 and the PC counter has an area of 416. Altogether the area of our IFU comes to be 2665 units. We realized after we finished our implementation that the area could be lowered if we replaced the second adder with another multiplexer.

When analyzing performance, we did not take into account the timing delay of each of the individual gates or components. Instead we chose to analyze which of the MIPS commands would cause the longest timing delay. The BNE and BEQ commands would have the largest timing delay out of all of the possible commands. This is because the BNE and BEQ commands are the only commands that require that the ALU does a computation, which causes a much longer time delay than in any other command that has to deal with the IFU such as jump or jump register.