# Lab 4: Single Cycle CPU with Multiplier Functionality

Abby Fry, Grace Montagnino, Vienna Scheyer

November 19, 2019

## Introduction

To start out, we worked on updating our CPU to handle more MIPS instructions and to be more functional. In addition to this, we upgraded our 32-bit single cycle CPU to handle to MUL instruction from the MIPS architecture. To do this, we modified our Lab 2 unsigned sequential multiplier to be able to handle 32-bit by 32-bit multiplication. To add a multiplier to our CPU architecture, we need 2 clocks. This is because our 32-bit unsigned sequential multiplier takes many cycles to complete the operation, while our CPU is still single cycle. This led to an update in our system timing where there is a relatively slower clock that dictates the overall CPU behavior, and there is a faster clock that specifically handles the multiplier sequential operations.

## Architecture

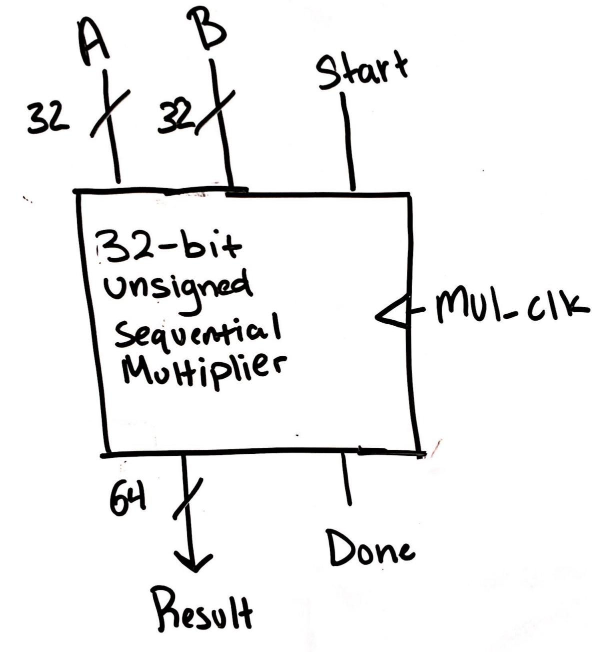


Figure 1: Top level schematic of our 32-bit multiplier.

To add multiplier functionality to our CPU, we first began by creating a multiplier component that could do 32 by 32-bit multiplication. Then we used the MIPS instruction set to begin understanding the data path need to incorporate the multiplier functionality.

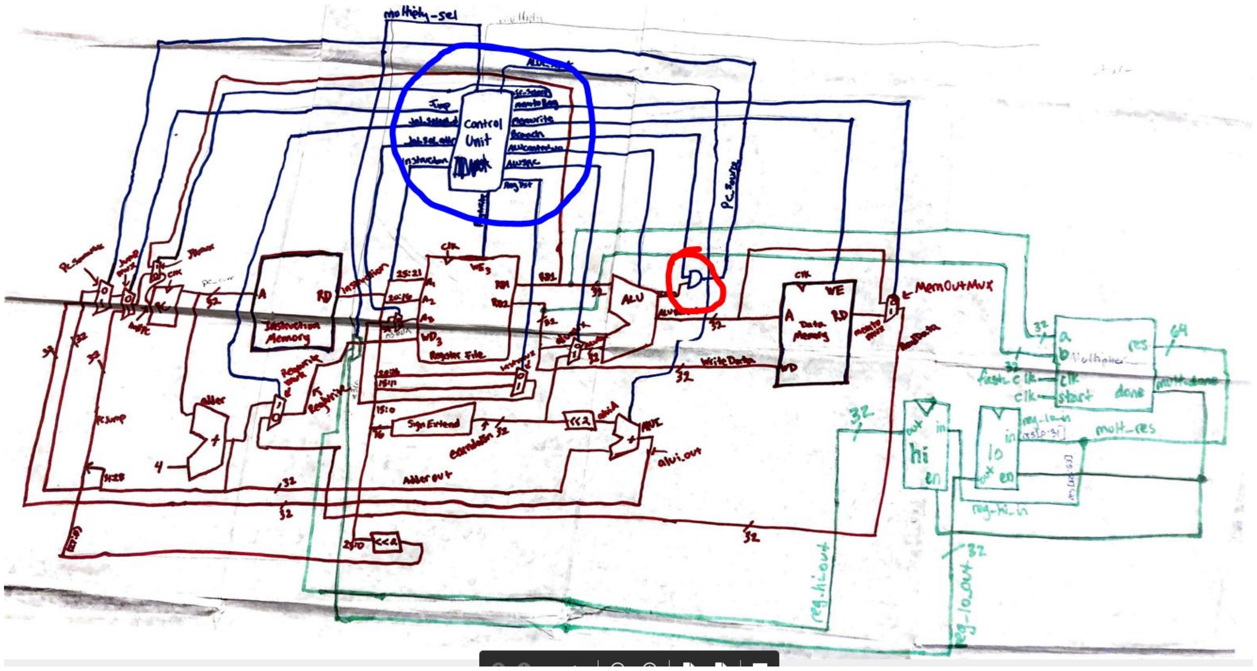
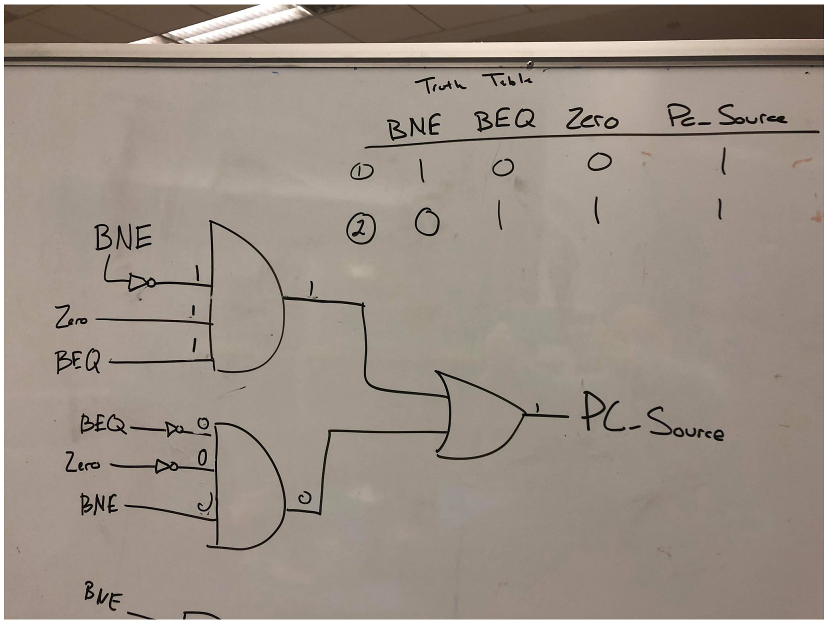


Figure 2: Schematic of our single cycle 32-bit CPU with the additional functionality of 32-bit multiplication. Modification to the red and blue regions occurred during our revision of our 32-bit single cycle CPU from lab 3.The revisions to the red region can be seen in Figure 3. The revision to the blue region can be seen in 4.


Truth Table

| | BNE | BEQ | Zero | PC_Source |
|---|---|---|---|---|
| ① | 1 | 0 | 0 | 1 |
| ② | 0 | 1 | 1 | 1 |

Figure 3: While revising our CPU desgin, we added new control signals, *bne* and *beq*, as well implemented new logic to determine the value of *PC_Source*. *PC_Source* is a address for one of the muxes used to select the next program counter value.

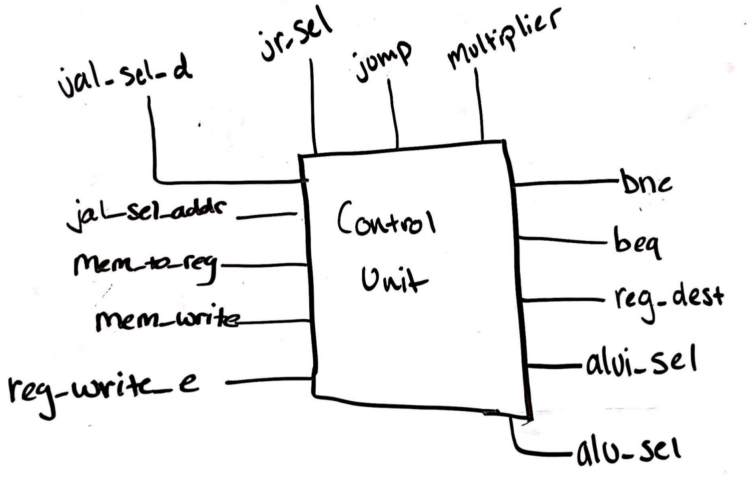
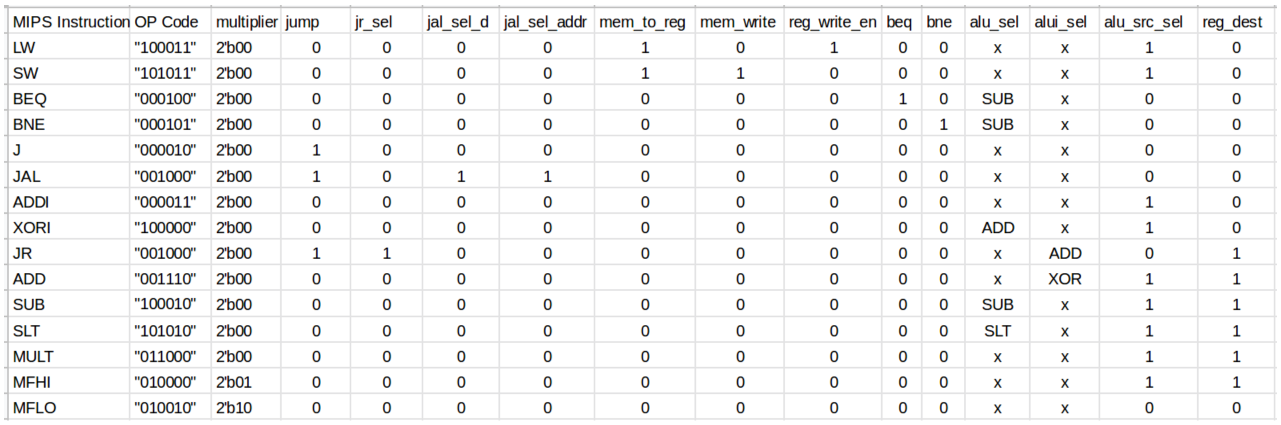


Figure 4: While revising our CPU design, we added new control signals to our control unit for *bne* and *beq*. We also added new control signals for our multiplier data path.



| MIPS Instruction | OP Code | multiplier | jump | jr_sel | jal_sel_d | jal_sel_addr | mem_to_reg | mem_write | reg_write_en | beq | bne | alu_sel | alui_sel | alu_src_sel | reg_dest |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LW | "100011" | 2'b00 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | x | x | 1 | 0 |
| SW | "101011" | 2'b00 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | x | x | 1 | 0 |
| BEQ | "000100" | 2'b00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | SUB | x | 0 | 0 |
| BNE | "000101" | 2'b00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | SUB | x | 0 | 0 |
| J | "000010" | 2'b00 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | 0 |
| JAL | "001000" | 2'b00 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | 0 |
| ADDI | "000011" | 2'b00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | 0 |
| XORI | "100000" | 2'b00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ADD | x | 1 | 0 |
| JR | "001000" | 2'b00 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | ADD | 0 | 1 |
| ADD | "001110" | 2'b00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | XOR | 1 | 1 |
| SUB | "100010" | 2'b00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | SUB | x | 1 | 1 |
| SLT | "101010" | 2'b00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | SLT | x | 1 | 1 |
| MULT | "011000" | 2'b00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | 1 |
| MFHI | "010000" | 2'b01 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | 1 |
| MFLO | "010010" | 2'b10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | 0 |

Figure 5: A table of the control signals implemented in our Control Unit look-up table.

Our current CPU status is that we can properly execute LW, SW, J, JR, JAL, BEQ, BNE, ADDI, and ADD, but are still working with XORI, SUB, and SLT. XORI works in 3/4 cases, but for some reason is executing addition when it is fed 1,1. SUB is never getting written to the register, which is confusing because it has the same datapath as add, which we feel we understand quite well from all the debugging! SLT should also be part of the ALU and under the same datapath as ADD, but somehow is producing very very large numbers. Overall, we might not have gotten a full CPU working, but we feel like we cleared up a lot of misconceptions throughout the debugging process, as well as just had that chance to compare verilog to schematics to correct mis-wirings and human errors. The process of discovering error by tracing datapths in the the diagram and in gtkwave output are skills we had the opportunity to practice many time during this lab and they seems like meaningful learnign outcomes.

Our status on the multiplier is that we have it all wired up in verilog, but did not get a chance to debug. We are fairly confident that our wiring is correct, or at least close to correct, but believe there might be an issue with our control logic. Regardless, adding the multiplier in was a good thought exercise in implementing something that required different clock timing/cycling, as well as a MIPS instruction that would require more 64 registers when our reg file only holds 32.

## Test Plan and Results

Our first order of business was to figure out where there were problems with our CPU and try to address those. Initially we were having a whole slew of problems, and it seemed too cyclic to narrow down a cause. Eventually, we checked mux outputs and found that many of our mux's were implemented opposite from how we drew them in our schematic. This was causing weird program counts, and thus weird instructions to be executed. After this we went through the schematic and look-up table mux by mux to make sure all of our addresses in verilog matched what we had been using in our schematic.

Another issue that we faced was every time we ran an assembly file with JAL, our CPU was completing the jump, but never doing the link. After a lot of gtkwave and comparison to our schematic, we learned that the expected control was not appearing in gtkwave. Further exploration of our LUT led us to discover that multiple instructions were being executed on because of a mismatch in our if statements. The problem was resolved by re-framing our if statements for R-type instructions and the default case.

Our next big hurdle was figuring out why our beq function was not working. Upon further investigation and some discussion with Ben, we learned that our design did not account for the beq instruction. Somehow we had subconsciously decided that since bne worked, beq was fine, and really skipped fixing the architecture to accommodate for it. We ended up adding a new control signal for BEQ in order to recognize the instruction in our CPU. Additionally, we added some combinational logic between the BEQ, BNE, and zero flag to determine what our PCsource address line should be feeding into our pcsource mux. We used a series of and and or gates to fulfill our desired truth table. You can see the truth table requirement and the logic used in

Figure 3 above.

To upgrade our CPU, we first modified our 4-bit multiplier to be able to do 32-bit multiplication. We verified this functionality with a test bench before initializing our multiplier in the CPU. The architecture of our multiplier includes the multiplier itself, which takes RD1 and RD2 from the register file as inputs and sends the 64-bit output data to two registers. Since the output is 64 bits, we save the highest 32 bits in one register and the lowest 32 bits in the second register. If the user just executes the multiplication instruction, nothing will be saved to a register in the register file. To save data, one must use the move from low (mflo) and move from high (mfhi) commands. This saves first the low 32 bits and then the high 32 bits into the register file for the user to obtain. the two registers feed into a mux with the third signal that is given to the register file when multiplication is not taking place. The control signal for this selects which signal to used based on the MIPS instruction given. We designed the multiplier in our circuit diagram and then implemented it in Verilog code. Unfortunately, we ran out of time to test the multiplier, but in this exercise of working through the multiplier integration data path and controls we gained a deeper understanding of the CPU.

## Reflection

Our CPU is quite deceiving, and despite it seeming like we were close to finishing it right when we started this lab, there ended up being so many layers that we could not finish. This was frustrating for us, and made it challenging to keep morale. It felt like a lot of let downs every time we thought we had fixed it, and then some new problem would be revealed. Despite this, the debugging gave us a lot of opportunities to reveal misconceptions and then correct them. Adding in the multiplier proved to be a good chance to think more flexibly and creatively about computer architecture because of the clock and register differences required for its implementation. Office hours have been a challenge because almost always only one of us (if any) is available, which put a lot of pressure on the person able to go to office hours. However, office hours were an essential part of this lab process, and where a lot of the clarity was brought on.