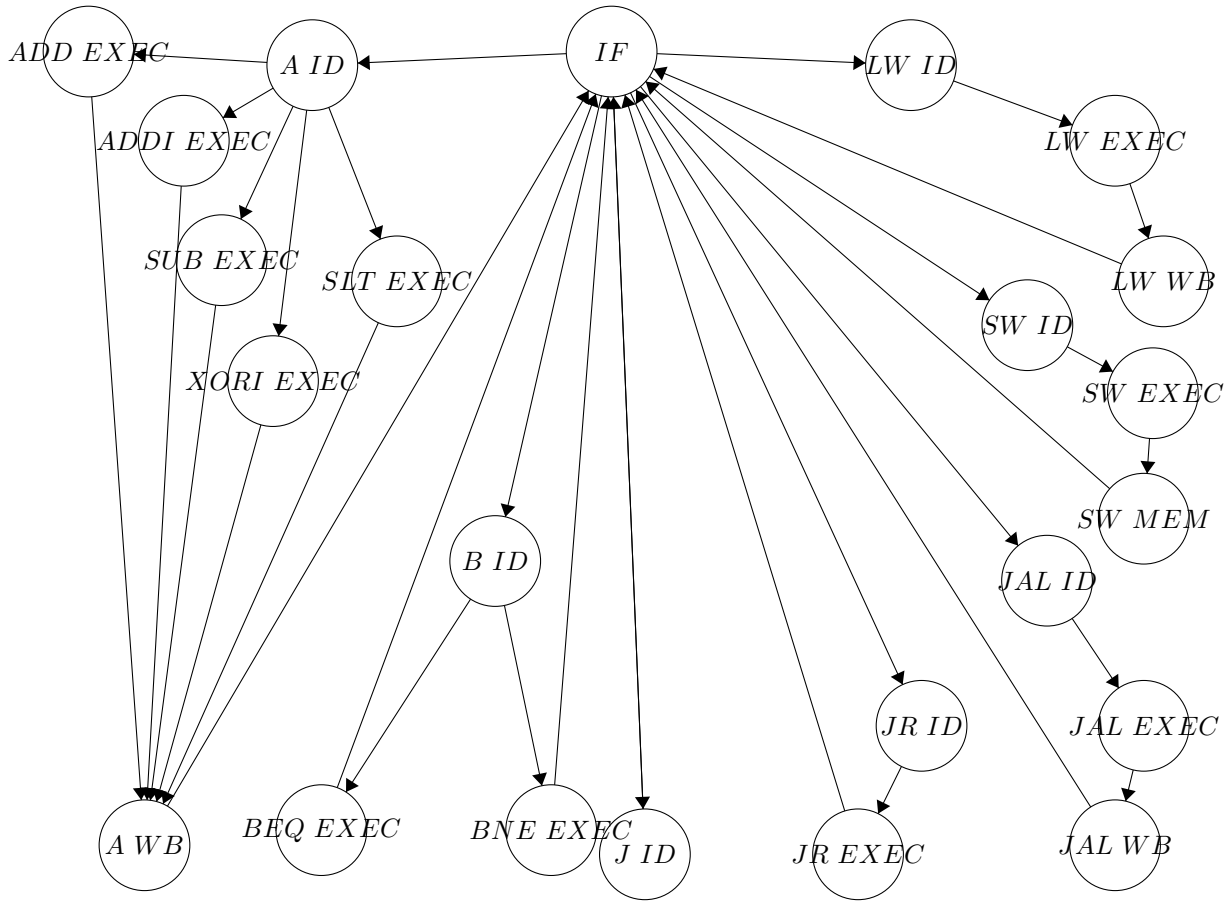
Lab 4: Multicycle CPU

Allison Busa
Jordan Crawford-O'Banner
Shashank Swaminathan

November 20, 2019

Verilog

We based this multicycle CPU based on our implementation of the single cycle CPU and the class notes on the multicycle CPU. On a higher, systematic level, this is how our CPU works:



In terms of what we actually implemented, a circuit diagram of our CPU can be found below.

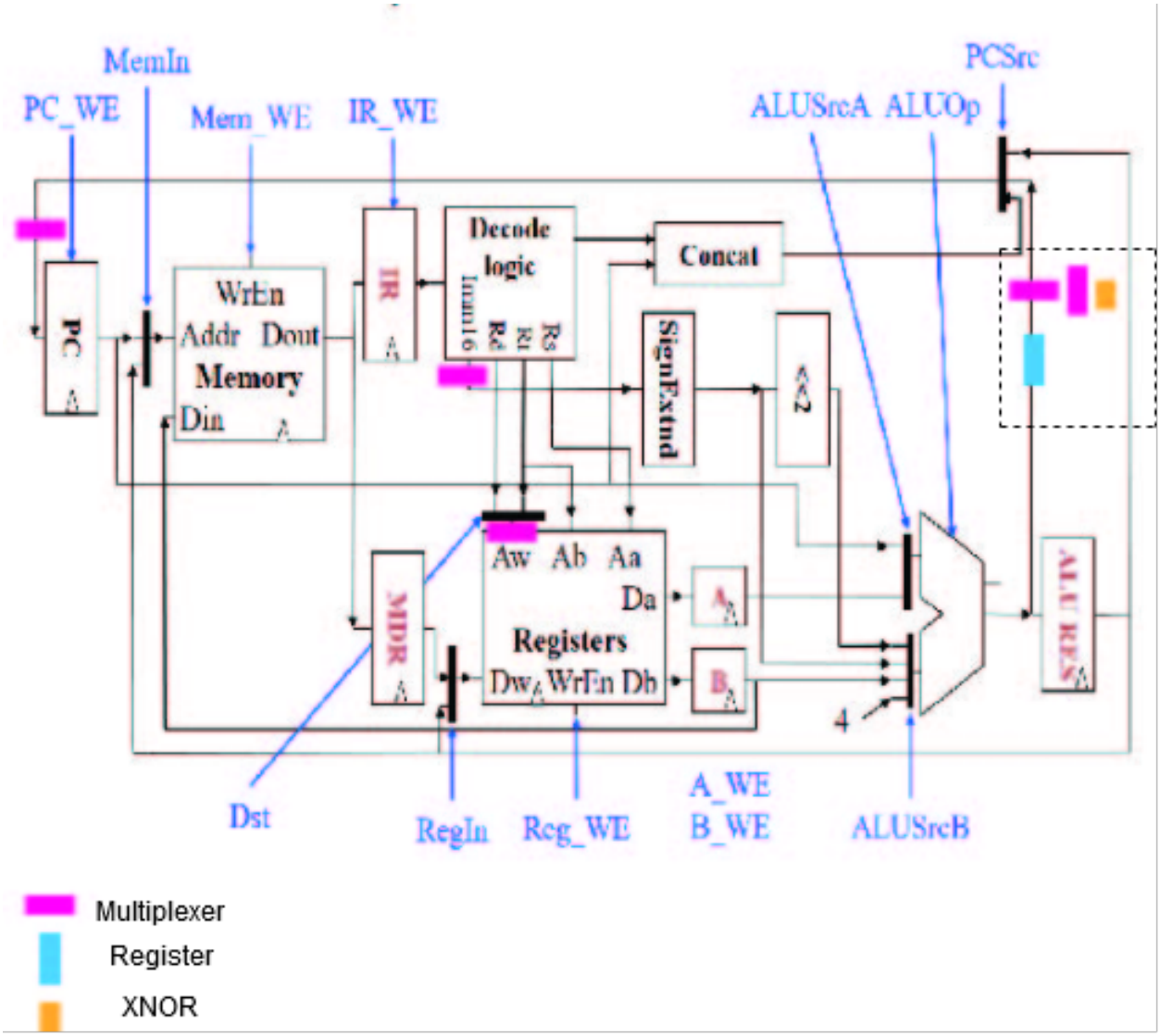


Figure 1: Circuit Diagram

We heavily based this circuit diagram on the CPU diagram provided in class. We've highlighted the new components we've added as rectangles. The pink rectangles represent multiplexers, the blue rectangle represents a register, and the orange square represents a XNOR gate. We will describe each multiplexer in greater detail below. Also, the section highlighted by the dashed line has a lot of components, so we will provide another diagram, describing this section in greater detail.

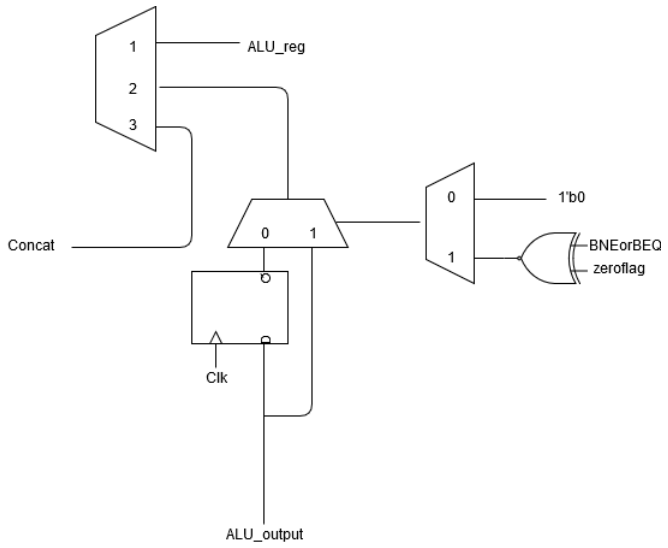


Figure 2: BNE/BEQ Added Section

The way that we progressed through this design, is that we listed out all of the operations we have to do, and we traced out how it would progress through the circuit. If we saw that current hardware wasn't capable of completing all the functions then we added more hardware which had the capacity to do what we needed. For example, for the **jal** command, we wanted to shift the write address between the output of the decode logic and 31 (ie: register 31). We also used this process to make our look-up table and our finite state machine. Each of the states was decided by the common functionality that is involved in each command. The common states between each command are Instruction Fetch, Instruction Decode, Execute, Memory and Write Back.

Here are a few of the changes we made to the original design to fit our purposes:

- Save Register: The save register saves the value of the address of the next line of the MIPS code. This register ensures that we always know where the next instruction will be while we are doing the current instruction.
- JAL Mux: The intention of this multiplexer is to make sure that the address is saved to the 31st register during the JAL instruction. Since the address is not given in the instruction, it must be directly fed to the regfile.
- Immediate Mux: During some instructions, we wished to add zero to the value coming into the ALU. We decided that adding a multiplexer after the immediate that came out of the instruction decoder would solve the solution best. The multiplexer takes in the immediate value from the instruction decoder and 0.
- Branch Items: Both of the branching instructions required extra hardware to make them functional. A mutliplexer and a XOR gate needed to be added to ensure that the correct operation was done and whether the two values given to the CPU are equal. A second multiplexer also needed to be added to make sure that
- Reset Mux: We also had a multiplexer that set the program counter to 0. This is necessary because the program counter does not initially start with a value and needs to be set with some intial value.

Each of the controls are based upon both the current instruction and the state of the system. The finite state machine has collapsed some of the states into one state based upon the fact that the controls for each of those states are the same, so they will have the desired result. For example, a lot of the arithmetic commands have the same controls for the Instruction Decode stage, so this has been simplified to the Arithmetic Instruction Decode state. Also all of the final states for each command will lead back to the Instruction Fetch stage, so that the system can receive the next instruction.

Testing

Testing Plan

We're using the same test benches as for our single cycle CPU. We're doing this, because both are CPUs and should both be able to do the same operations. We are also using the same tests, because our test benches for the single cycle CPU were exhaustive of all the operations that a CPU can do.

Errors Found During Testing

- Initializing registers.
  When we ran the tests for the first time, everything was all X's. We traced it through, and realized that the input to the PC was X. We realized that we needed to set the input to PC to zero, in order to start the system. At first, we tried adding a reset multiplexer, which was initially set to zero. This didn't work, because the IR write and Mem write were set to low, so the address wasn't getting through to the LUT (which sets the commands to the rest of the circuit). This error led us to our next error, which we describe below.

- Setting LUT

  For much of the time we worked on Lab4, we had an incorrect implementation for our finite state machine. Initially, we had the state of the system as an input to the MCPU, and simply had it constantly update through the look-up table. However, this proved infeasible as our state would never update correctly since it was never actually stored for more than a single clock cycle, In order to counteract this problem, Professor Hill suggested that we should add a register to our finite state machine in order to store the state until it should be updated again. This worked far better and proved to be an effective way to keep the state updating only when we finished executing a command, instead of updating every clock cycle.

- Saving PC + 4

  We had a lot of issues updating our program counter throughout our implementation of the multi cycle CPU. One of the biggest issue is that we could not consistently update our program counter to the next address in memory. Originally, we had assumed that the multiplexer would correctly send the updated value to the PC. However, we realized that there was an issue where we would update the program counter too many times, and the program counter would update the address too many times, which would make the program counter much higher. In order to ensure that the program counter would update to the correct move to the correct value we add a register to the system that would specifically hold the next memory address that the program would need to update to at the end of the command unless the program was intending to jump or branch.

- Improving BNE/ BEQ

  Originally, we had a very complicated implementation of BNE/BEQ that involved going through the execute stage multiple times. We believed that there was a better implementation that would increase the speed of our overall MCPU. This new implementation was an idea that we had while we were improving how PC+4 was being implemented. We realized that since there was now a register with enable that was holding PC+4, we didn't need the register that was just holding the PC+4 address in the BNE/BEQ instructions (which we couldn't use to fix PC+4). We figured that we could add a multiplexer to choose between PC+4 and PC+4+branchaddress, which would also be in the wire that's the output of the ALU. We assumed that there would be one control signal, that tells you if the instruction is BNE or BEQ, where BNEorBEQ =0 for BNE and 1 for BEQ. We also knew that there is a zero flag that is output from the ALU. This is what the multiplexer to choose between PC+4 and PC+4+branchaddress looks like :

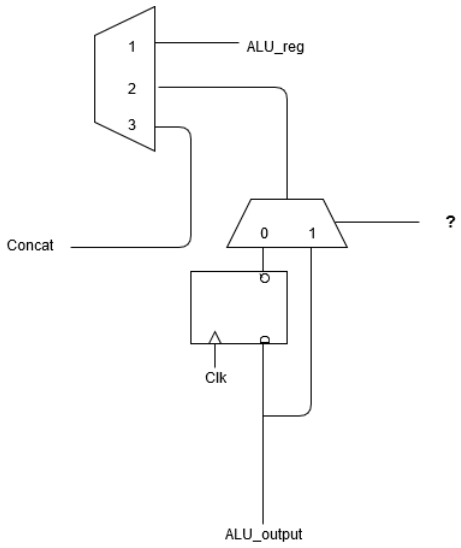


Figure 3: Initial Design of BNE/BEQ

So, in order to figure out how to create this functionality, we started with a logic table.

Desired Functionality	BNEorBEQ	ZeroFlag	Desired Output (In Words)	Desired Output (In Mux Inputs)
Check if BNE, but they're equal	0	1	PC + 4	0
Check if BNE, and they're not equal	0	0	PC + 4 + Branch Address	1
Check if BEQ, and they're equal	1	1	PC + 4 + Branch Address	1
Check if BEQ, but they're not equal	1	0	PC + 4	0

Figure 4: First Logic Table for BNE/BEQ

We figured that since the zero flag and the BNEorBEQ are the ones which are driving the behavior, we can make a smaller logic table, that describe how those two factors influence the output of the multiplexer drawn in the previous figure.

Zero Flag	BNEorBEQ	Chosen Mux Address
1	0	0
0	0	1
1	1	1
0	1	0

Figure 5: Second Logic Table for BNE/BEQ

According to this table, we decided that we needed to XNOR the zeroflag and BNEorBEQ. Furthermore, because of the other instructions, we need to choose between these input and just the normal output of the register. Therefore, we add another multiplexer that chooses between the XNOR output and the 0th output (the register output).

- Setting enable signal at the wrong time
  In our multicycle CPU configuration, we have a set of modules which have enable signals. For some of these modules, like the IR, A and B registers, we correctly set the the enable signal to high one clock cycle before the module was used. However, when we were testing and debugging errors in gtkwave, we saw that the PC register wasn't updating on the IF state, but the ID state. This happened because the modules probe the enable signal right before and during the positive edge. So, instead, we decided to set all the enable signals to high one state before the modules were used. For example, we set the PC Write Enable signal (`PC_WE`) to high in the terminal state of the last operation. By terminal state in the last operation, I mean, if for example, the last operation was ADD, then it's last state would be Write Back (WB). In this Write Back state, `PC_WE` = 1.

Performance

The whole point of doing a multicycle CPU is to optimize the speed of a CPU, so that it's faster than a single cycle CPU. We can compare the speeds of our single CPU and multicycle CPU, but, in order to do that, we have to calculate the speed of the multicycle CPU.

The multicycle CPU needs a clock length that is at least as long as its longest operation. In this case, the longest operation is a memory access, since it requires interfacing with the disk of the CPU. Since we know the access time required for such a command is so long, we can assume it to be something along the scale of 10000 time units. Hence, every clock cycle needs to be of length 10000 time units. The breakdown of cycles required by each command is shown in Table 1.

Instruction	IF	ID	EXEC	MEM	WB	Total Num. of Cycles
ADDI	✓	✓	✓		✓	4
ADD	✓	✓	✓		✓	4
SUB	✓	✓	✓		✓	4
SLT	✓	✓	✓		✓	4
XORI	✓	✓	✓		✓	4
J	✓	✓				2
JR	✓	✓	✓			3
JAL	✓	✓	✓			3
LW	✓	✓	✓		✓	4
SW	✓	✓	✓	✓		4
BEQ	✓	✓	✓			3
BNE	✓	✓	✓			3

Table 1: Number of cycles for each MIPS instruction in the multi-cycle CPU

In comparison, a single cycle CPU combines all these stages together. Hence, the clock cycle period needs to be long enough for the longest operation - which would be a `LW` command. The high-level steps in a single cycle CPU operation would be: Instruction fetch - from the memory unit; Instruction decode - via an LUT or something similar; Operation execution phase - through an ALU component; and Writing to memory.

The two memory operations would be 10000 time units long each. Based on the in-class slides, we make the assumption that the other operations will take about 20000 time units (making an approximation off of the length of operations displayed). Hence, the single cycle clock period is 40000 time units.

With the clock periods defined, we can now take a look at an example instruction set, and compute the overall time each CPU would take. For our instruction set, we will use an insertion sort algorithm (which is linked here). To find the total number of cycles, we manually went through the instruction set of the insertion sort test, followed the program execution, and counted up how many of each instruction would occur (counting jumps and all). The total number of cycles for the single cycle CPU is equal to the total number of instructions. For the multi cycle CPU, the total number of cycles is the sum of all the instruction counts, weighted by their respective cycle number (seen in Table 1). Finally, we multiply the number of

cycles calculated by each CPU's respective clock cycle periods. The results of these calculations are shown in Tables 2 and 3.

Instruction	Count	Num. Cycles
ADDI	46	184
ADD	41	164
SUB	5	20
SLT	20	80
XORI	1	4
J	26	52
JR	6	18
JAL	6	18
LW	50	200
SW	56	224
BEQ	37	111
BNE	15	45

Table 2: Count of how many times each instruction happens in the insertion sort algorithm.

CPU type	Number of Cycles	Total Time Taken
Multi-cycle CPU	1120	11200000 time units
Single-cycle CPU	309	12360000 time units

Table 3: Time taken by each CPU implementation for the insertion sort algorithm

This indicates to us that the multi-cycle CPU performs noticeably better than the single-cycle CPU, by 1160000 time units.

Conclusion

Unfortunately, our multi-cycle CPU did not work, despite the fact that we spent over 20 hours on this lab. We decided to take our Professor's advice and not spend more time than necessary on the assignment. Although we did not create a working multi-cycle CPU, we still learned a lot through this experience. For example, we made the most complicated LUT that we've ever done- and we learned about going back and debugging errors in it. We had to practice systematically looking for errors. We've spent more time than in other labs looking into the gtkwave and trying to understand what we should be seeing and what is actually happening, in order to pinpoint our problems. However, if we had the chance to approach this lab again, we know some of the things which we would change. One issue with our approach was that we would often get sidetracked by smaller errors when there was a larger error that needed to be solved. Often this was because we incorrectly believed that the smaller error caused the larger error. In the future, we will be sure to debug in a more structured manner than investigating the first error we see. By making sure to look at the core parts of our design first before moving on to the more peripheral aspects, we can accurately find the causes of errors and understand whether that error is caused by a more core part of the design. Another of the core issues with our approach with this project is that we didn't plan ahead of time how the lab would play out and how we should portion our time. At the start of the project, we decided to plan out our whole LUT, which we did in shorter sessions more often. Although the planning step is very important, we should have front loaded a lot of the planning work with longer sessions in order to ensure we had the proper structure in place to implement the MCPU. We should have done longer sessions, in order to waste less time remembering where we left off. Frontloading would have prevented us from spending a lot at the end of project debugging our implementation. We could have spent this time earlier in the project debugging our verilog code, and we would have had a better chance at being done earlier.