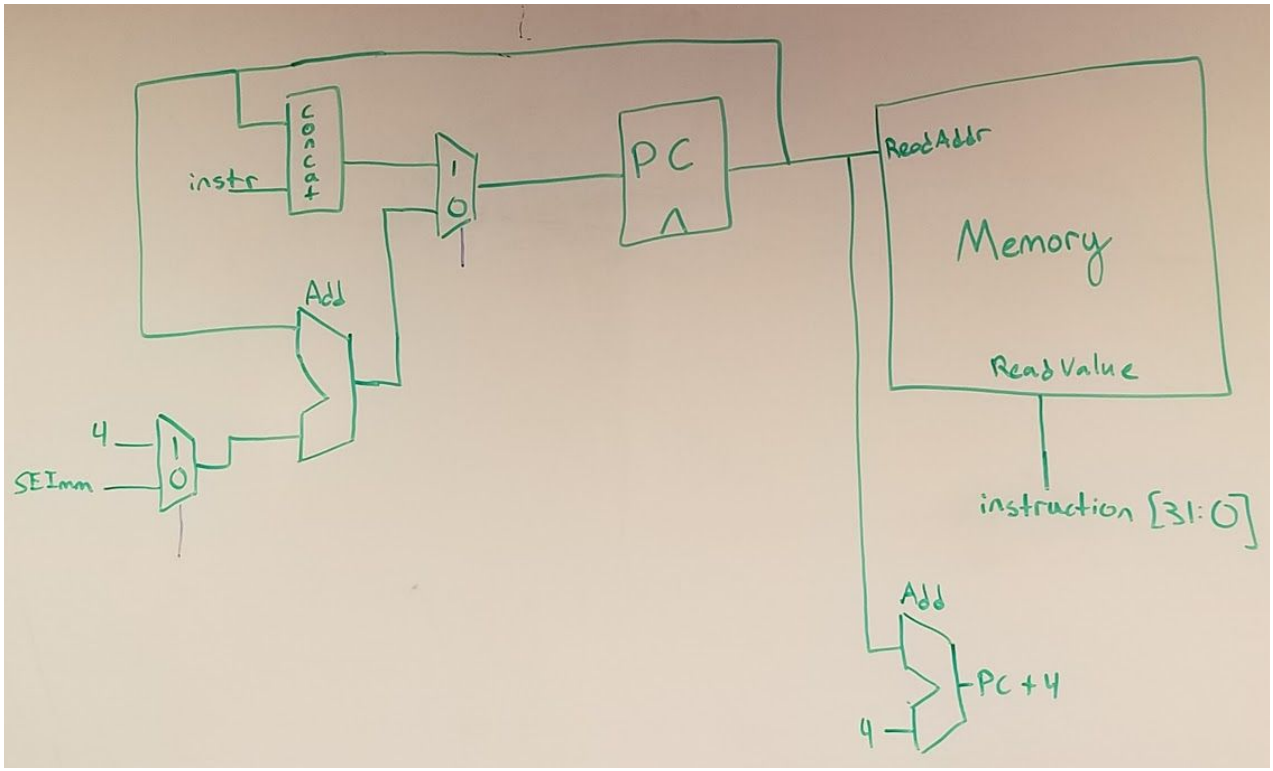
Lab 3 Report

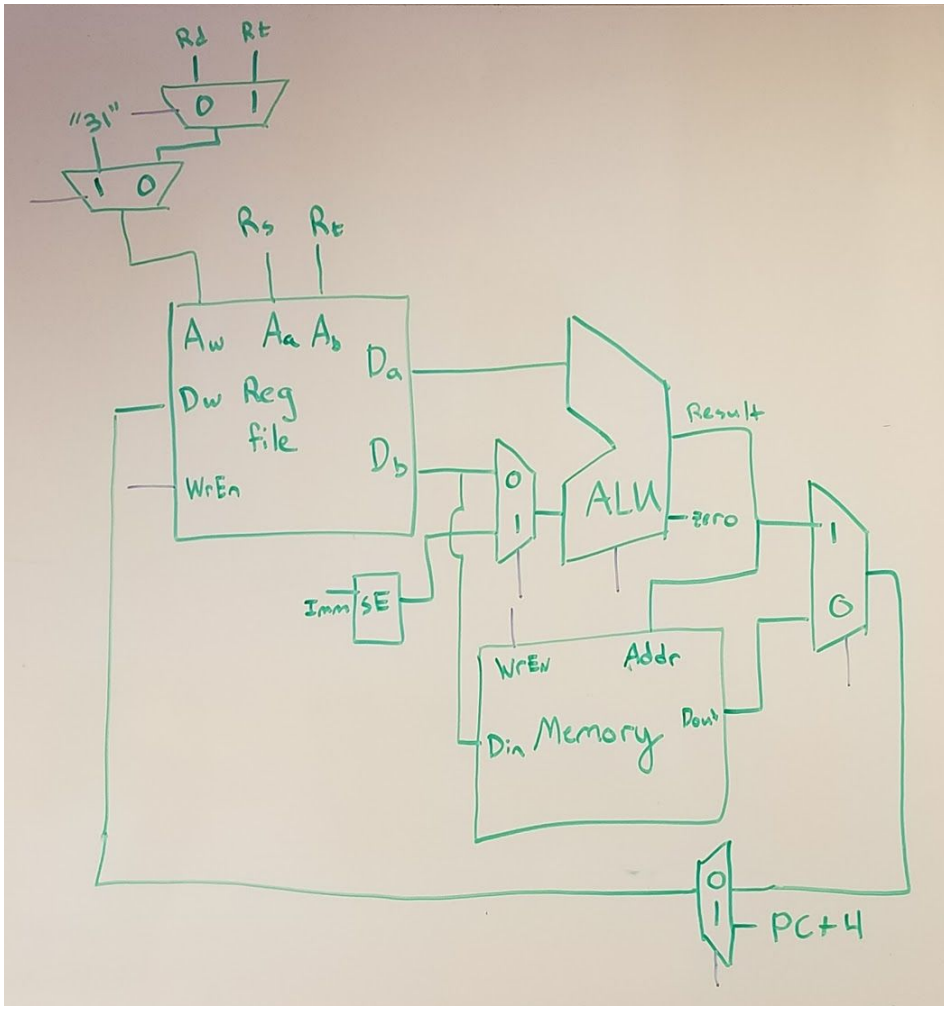
Coleman Ellis and Paige Pfenninger

Our block diagram has two components: the program counter logic, and the execution logic. Please note that although there is a memory module in each block diagram, it is in fact the same memory module (there is only one memory module in our CPU). The memory module has one read write port (for data reading and writing) and one read port (to read instructions).



At each positive clock edge, the PC register is updated with the next PC register value and the memory outputs the next instruction. The updated program counter is either 4 plus the current program counter (step), a sign extended immediate plus the current program counter (branch), or a concatenation of the current program counter and the instruction (jump). What happens to the updated program counter depends on what type of instruction comes out of the memory.

When the instruction is outputted, the rest of the circuit starts completing calculations based off of the instruction. The ALU has the ability to add, subtract, slt, and xor. The signals propagate around the rest of the circuit controlled by the instruction that comes out of the memory unit. The regfile and memory both continuously output data based on the address inputted, but will only write data on the positive edge of the clock.



Here is the logic for the execution portion of our diagram:

Instruction	Opcode	Function	Rd or Rt Mux	Is 31 Mux	ALU Operation	Db or SE Immediate	ALU Res or Dout	PC+4 Mux
LW	100011	-	0 (Rd)	0	ADD	0 (Db)	1 (Dout)	0
SW	101011	-	0	0	ADD	0	0 (ALU Res)	0
J	000010	-	0	0	Operand A	0	0	0
JR	000000	001000	1 (Rt)	0	Operand A	1 (SE Immediate)	0	0
JAL	000011	-	0	1	Operand A	0	0	1 (PC + 4)
BEQ	000100	-	0	0	SUB	1	0	0
BNE	000101	-	0	0	SUB	1	0	0
XORI	001110	-	0	0	XOR	0	0	0
ADDI	001000	-	0	0	ADD	0	0	0
ADD	000000	100000	1	0	ADD	1	0	0
SUB	000000	100010	1	0	SUB	1	0	0
SLT	000000	101010	1	0	SLT	1	0	0

We created this logic using various gates that took the function and opcode as inputs.

RTL of Commands:

1. LW: R[rt] = M[R[rs] + SE Immediate]
    a. The load word function inputs two registers (rs and rt) and a immediate. The ALU adds R[rs] and the sign extended immediate together and that sum becomes the address input to the memory block. The memory outputs the value at that address and then it gets written to the register file at rt.
2. SW: M[R[rs]+SignExtImm] = R[rt]
    a. A store word stores the value of R[rt] to the memory address of R[rs] plus a sign extended immediate. To complete this operation, write is enabled for the memory and the ALU is set to addition.
3. J: PC = JumpAddr
    a. Jump causes the PC address to be set to the jump address. The jump address is only 26 bits long, so it is concatenated with the most significant 6 bits of the current program counter.
4. JR: PC=R[rs]
    a. In this instruction, the program counter is just being set to whatever is stored in the register file at address rs.
5. JAL: R[31]=PC+4;PC=JumpAddr
    a. Jump and link performs two functions. First it saves the current program counter plus 4 to register 31 of the register file. It then changed the current program counter to the jump address.
6. BEQ: if(R[rs]==R[rt]) PC=PC+4+BranchAddr
    a. In our CPU, we used the subtract feature and the zero flag of the ALU to determine BEQ and BNE. If the zero flag is high, that means that the two inputs are equal, and so our CPU jumps.
7. BNE: if(R[rs]==R[rt]) PC=PC+4+BranchAddr
    a. For BNE, if the zero flag is low when R[rs] is subtracted from R[rt], then the two numbers are not equal so our CPU branches.
8. XORI: R[rt] = XOR(R[rs], SignExtImm)
    a. For an XORI operation, the ALU takes the XOR of R[rs] and the sign extended immediate and that value is then stored in rt
9. ADDI: R[rt] = R[rs] + SignExtImm
    a. For an ADDI operation, the ALU adds R[rs] and the sign extended immediate and that value is then stored in rt
10. ADD: R[rd] = R[rs] + R[rt]
    a. For an ADD operation, the ALU adds R[rs] and R[rt] and that value is then stored in rd
11. SUB: R[rd] = R[rs] - R[rt]
    a. For an SUB operation, the ALU takes the difference between R[rs] and R[rt] and that value is then stored in rd
12. SLT: R[rd] = (R[rs] < R[rt]) ? 1 : 0

13. For an SUB operation, the ALU calculates if R[rs] is less than R[rt] and that value is then stored in rd

Our Testing Strategy

We started off testing a simple division in a loop program that we wrote. This caught several errors. All of our errors were caused by control logic of the various muxes, the regfile, and the alu. That program tested everything except for LW, SW, and XORI. We then tested SW and LW and again found logic issues in our ALU. Afterwards we ran the fibonacci test that the ninjas created. We found more logic errors when running that test. After running all of those tests, we believe that our CPU fully works. We tested all of the functions of our CPU in the three tests, and our CPU behaved correctly for all of the tests.

The majority of the issues were caused by the fact that we wrote our first block diagram out on a whiteboard, and a lot of the muxes on the whiteboard had the inputs switched and some of the control signal notation was smudged which made it difficult to correctly copy the control signals into code. We also ran into some issues where our CPU was using immediates with branch instead of two register values which caused a lot of problems. If we were going to do this again, we would first create the block diagram on a whiteboard and then copy it onto a piece of paper where we would ensure that all of the control signals are legible.

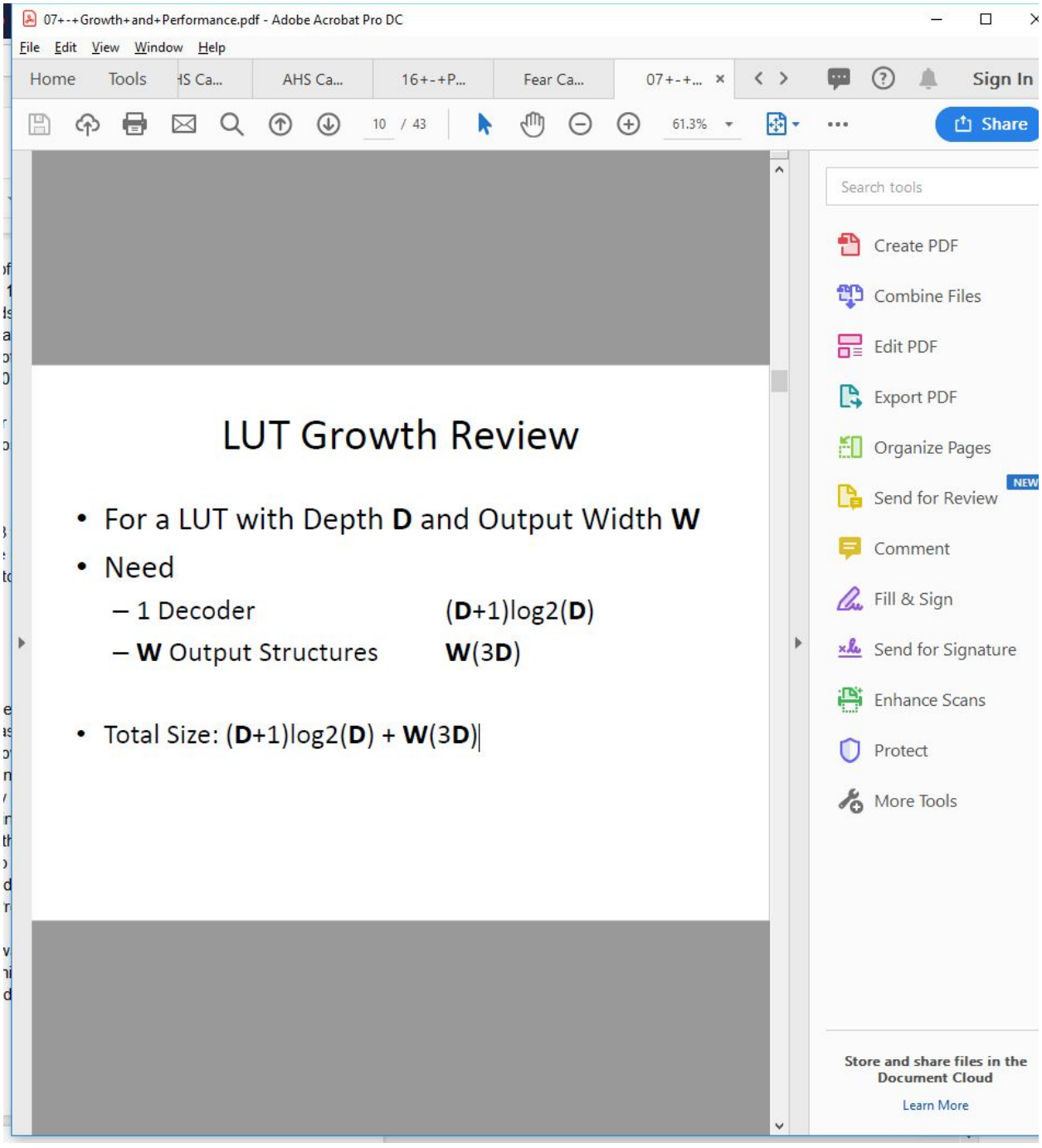
Area Analysis

1. ALU - 38874
    a. Each bit slice consists of one full bit adder (2 XOR, 2 AND, 1 OR - 15), an XOR gate (3), lookup table, and three AND gates (3 each) and an OR gate for calculating the output (3). The lookup table outputs 4 bits (width) and has a depth of 2^6, so the total cost of the lookup table is 1158. So each bitslice has a total area cost of 1188.
    b. The main part of the ALU has 32 bitslices (1188 each), a lookup table with a depth of 2^6 and two outputs (744), 35 2-input AND gates (3 each), a 2 input OR gate (3), a 3 input OR gate (4), and two inverters (1 each). Our total ALU has a cost of 38874.
2. RegFile - 28736
    a. The regfile has 32x32 bits of memory, which assuming that each bit of memory is made up of an SR flip-flop with a cost of 4. The regfile memory has a total cost of 4096.
    b. The regfile also has two muxes: one for reading and one for writing. Each of these muxes is a 32 bits (so it has 32 sub-muxes) with a 6 bit control signal. Each sub mux has 32 7-input AND gates (8 each), 1 32-input OR gate (33), and 32*6/2

inverters (1 each) for a total cost of 385. With 32 sub-muxes and two full muxes, our total mux cost is 32*2*385 = 24640.

3. Memory - 538318241
    a. Our memory 32 bits wide and 4096 memory address deep, which is 131,072 bits. Assuming that each bit is a set reset flip flop, our memory would need 131,072 SR flip flops. Assuming that each SR flip flop is made from two two-input NAND gates that each have a cost of 2, our memory storage has a cost of 4*131072 = 524288.
    b. Our memory also has three muxes (two read ports and one write port). We don't actually know how the memory storage is implemented, but the worst case scenario is probably with a MUX. Each mux has a 12 bit address and outputs 32 bits (so 32 sub-muxes). Each submux has 4096 13-input AND gates (14 each), 1 4096-input OR gates (4097 each), and 4096*12/2 = 24576 inverters (1 each). This gives us a total of 16806061 for each bit. With 32 bits, the total cost of the memory control logic is 537793953. This seems way too high, but we are unsure exactly how data memory is written to.
4. Muxes - 1710
    a. In our CPU we have 8 two-input muxes. 5 of these muxes are 32 bit muxes, 2 of them are 5 bit muxes, and the last one is a 1 bit mux. For these calculations, we are assuming that each bit of the mux needs its own separate logic. For a two input mux, one bit needs: 1 two-input OR gate (3), 1 inverter (1), and 2 two-input AND gates (3), which gives us a total cost of 10 per bit of each MUX. Therefore, the total cost of the MUXes in our CPU is 10*(5*32 + 2*5 + 1*1) = 1710.
5. PC Register - 416
    a. We are assuming that the program counter register is made up of 32 D-flip-flops. From the midterm, each D-flip-flop has a cost of 13, so our total PC register cost is 32*13 = 416.
6. Lookup tables - 584
    a. We have one additional lookup table to calculated RegWrite. This look up table has one output and a depth of 2^6, so its total cost is 584.
7. Extra Control Logic - 42
    a. To create the extra control logic, we have: 3 two-input AND gate (3 each), 1 six-input OR gate (7), 2 five-input AND gate (6 each), 9 inverters (1 each), 1 seven-input NOR gate (7). This gives us a total cost 42 for the extra control logic.

Adding all of the pieces together our final CPU has a cost of: 538388603. This number seems really high. The majority of it comes from our assumption that the memory outputs data using a giant MUX.



Work Plan Reflection

1. Collect and make nicer test files for existing modules (~3 hours) (10/26)
    a. This took a bit longer than expected and was about a day late. This took longer because our ALU needed more significant overhaul than we originally thought. We also had to write some additional components (like MUXes and an Adder) that we didn't include in the workplan. This probably ended up taking like 5 hours

2. Make control logic on paper/whiteboard (~2 hours in a meeting) (10/26)
    a. We scheduled the right amount of time for this. Making our CPU on a whiteboard was super helpful. We were actually able to keep it on the same whiteboard for the entire project which meant that we could constantly refer back to it.
3. Port from whiteboard to verilog (~3.5 - 8 hours) - Programming in verilog from whiteboard drawing (10/29)
    a. The porting from the whiteboard to verilog was actually done in a reasonable amount of time (about two hours). In our whiteboard diagram we specified down to the gate level with all of their inputs. The debugging took a lot longer, but almost all of the debugging took place when we were running the assembly tests. So this section took about 2.5 hours and the assembly section took a lot longer.
4. Write assembly tests (~1.5 hours) (10/29)
    a. It only took about 45 minutes in total to write both assembly tests. At this point we were behind due to illness.
5. Run assembly tests and debug (~0.5 hours or ~4 hours depending on how well we made the CPU the first time) (11/1)
    a. It probably took about 8 hours of us working together to debug our CPU. A lot of the issues we encountered were bad control logic (almost all of the issues were around control logic). They tended to either be caused by hard to read handwriting on our whiteboard CPU or an incorrect understanding of logic in the first place. This was completed many days late due to illness.
6. Write report (~3 hours) (11/2)
    a. This report took about three hours to write. Once we were far enough into debugging where having two people debug wasn't useful, one of us worked on the report during meetings (while helping with debugging) and the other worked mainly on just debugging. While the debugging took a substantial amount of time, it never felt like we were stuck-- we had one issue that we emailed Ben about, but figured it out immediately after we sent the email. Outside of that, we were constantly making debugging progress, regardless of whether it was slow.