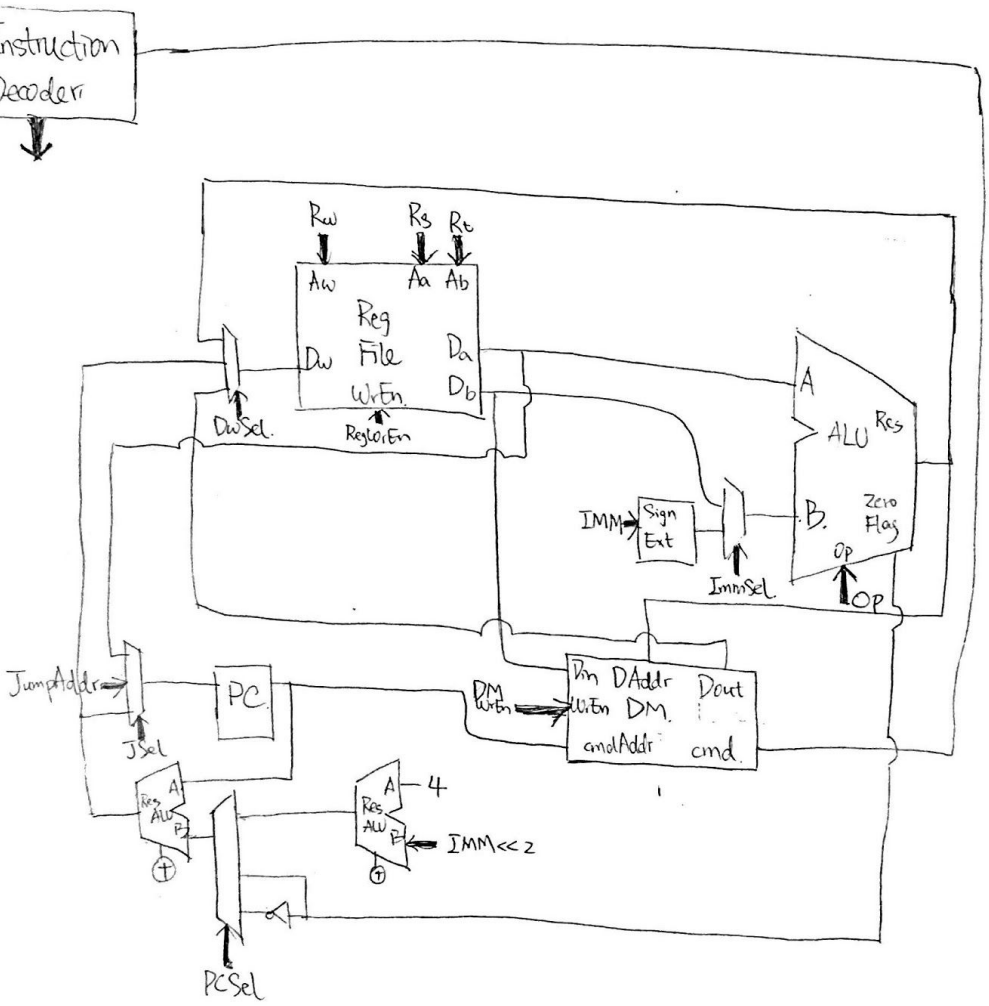
Comp Arch Lab3
Adam, Nick, Yichen
10/31/2018

If you just want to run our tests:

1. In terminal, navigate to the top level directory containing our project
2. Run: chmod +x full_test_runner.sh (if necessary)
3. Run: ./full_test_runner.sh

Section 1: Block diagram

The block diagram of our single-cycle CPU design is included below. All the control signals or data coming out of the **Instruction Decoder** are labeled with thick arrows on the diagram. Our data memory has two data output to make sure that the memory can access the data and the command in a single clock cycle.



Now we will use a few instructions to illustrate how our CPU works:

At the beginning of each clock cycle, data memory receives the command address from PC and outputs the command to the instruction decoder. The decoder then parses the command to get the Op code and create control signals accordingly. The control table of the instruction decoder is shown below:

Instruction	Opcode (hex)	Funct (hex)	ImmSel	ALUop	DWSel	JSel	PCSel	MemWrEn	RegWrEn
LW	23		1	ADD	2	2	0	0	1
SW	2B		1	ADD	0	2	0	1	0
J	2		0	N/A	0	1	0	0	0
JR	0	8	0	N/A	0	0	0	0	0
JAL	3		0	N/A	1	1	0	0	1
BEQ	4		0	SUB	0	2	1	0	0
BNE	5		0	SUB	0	2	2	0	0
XORI	e		1	XOR	0	2	0	0	1
ADDI	8		1	ADD	0	2	0	0	1
ADD	0	20	0	ADD	0	2	0	0	1
SUB	0	22	0	SUB	0	2	0	0	1
SLT	0	2a	0	SLT	0	2	0	0	1

LW: R[rt] = M[R[rs]+SignExtImm]
When the LW instruction is given, the ALU calculates the result from R[rs] and IMM and passes that to the data memory as the address. The data memory then fetches the data from that address and passes it to the register to be saved.

JAL: R[31]=PC+4;PC=JumpAddr
When the JAL instruction is given, R[31] receives the result PC+4 from an ALU, and PC gets updated by the Jump Address given by the instruction decoder.

BEQ: if(R[rs]==R[rt]) PC=PC+4+BranchAddr
When the BEQ instruction is given, the ALU subtracts the data from R[rs] and R[rt], and if the zero flag is high, The branch ALU will be activated and PC will be updated with PC+4+BranchAddr(IMM). Otherwise, PC will just update normally by PC+4.

ADD: R[rd] = R[rs] + R[rt]
When the ADD instruction is given, the ALU adds data from R[rs] and R[rt], and the result is passed to the register file and stored in R[rd].

Section 2: Test plan and results

Before running any of our advanced tests, we ran two batches of test scripts: the test scripts for each component within the CPU and the simple tests that can be seen under our /bleep_vim/simple folder. These tests checked the basic functionality of the cpu. Once these tests passed, we added some more advanced tests, and ended with the

following completed list of tests within our script to ensure that every CPU function is tested.

Unit structure tests:

- Alu.t.v
    - Tests all of the alu's different functions.
- Decoder.t.v
    - Ensures that the control signals are correct for each command.
- Mux.t.v
    - Double checks the different muxes' capabilities.
- Regfile.t.v
    - Ensures register stability and addressing.
- signExt.t.v
    - Makes sure that the sign extension is working.
- Cpu.t.v (used with below tests)
    - Runs actual programs, and ensures their outputs are correct.

Assembly tests run through cpu.t.v:

- Hanoi
    - Hanoi is our complicated test we created to submit for the class as a whole to use. It tests the number of movements required to complete the "towers of hanoi" recurrence relationship. It is used to test addi, jal, sw, add, lw, jr, bne, and j.
- Fibonacci
    - Fibonacci was created by the Ninjas as a test for us. Although it tests the same set as hanoi, we included it in case there was a harder combination that we missed that would cause problems with our cpu.
- Yeet
    - Yeet, created by team StoreMoney, was used to test beq.
- Test_1
    - Test_1 was created by team DazedandConfused; we used it to test sub.
- Test_3
    - Test_3 was created by team DazedandConfused; we used it to test xori
- (Simpletest) slt
    - We used our simple test slt to test slt because nobody else used it!

As we created the tests, we debugged the CPU and caught a variety of problems, including miswired components and incorrect logic (specifically, when calculating the PC). From this, we then expanded cpu.t.v to allow for multiple scripted tests, which are run by running ./scripted_tests.sh. This runs the 6 tests in the assembly tests category

above. We got all results as expected, which reconfirmed the fact that our CPU was correctly made.

In order to run the unit structure tests and assembly tests:

4. In terminal, navigate to the top level directory containing our project
5. Run: chmod +x full_test_runner.sh (if necessary)
6. Run: ./full_test_runner.sh

Section 3: Performance Analysis

We want to find the fastest clock speed at which this CPU could run. To find this, we will make the assumption that every logic gate takes 10 picoseconds per input, plus 10 if it does not invert.

Data memory: We assume our data memory is a separate SRAM chip, with a random-access time of 8ns.

ALU: The timing analysis for ALU is broken down for each individual operand, and the detailed gate level analysis was presented in the Lab1 write-up. Because our prior timing analysis is done in terms of gate units instead of seconds, we will assume a 10ps/gate unit conversion rate here. For the ALU operations that are used in this lab, ADD takes 10.72 ns, SUB takes 10.72ns, XOR takes 5.21ns and SLT takes 10.78ns.

Multiplexer (mux): Our muxes are written using indexing and ternaries, but structurally they are usually implemented using:

- One not gate per address wire (in parallel)
- One nand gate per input wire (in parallel)
- One n-input nand gate, where n is the number of inputs, for the output

Between these, the mux will take 30 + 10*n ps, where n is the number of inputs.

Register file: The register file is written behaviorally, but we can make some inferences about its structure. The chain of events for reading data looks roughly like:

- The read address changes.
- A 32-input mux selects a new input; according to above, this will take 30 + 320 = 350 ps.
- The previous step happens another 31 times, in parallel.
- The output address changes.

Because all of the muxes are in parallel, we can assume the register file will take about 350 ps to read. It will take a bit longer to write, but our slowest path does not include writing.

Fetch and decode: The flow of information starts at the rising edge of the clock with the PC updating to its new value (usually 4 more than last cycle). This takes 40 ps. Next, its output is sent to the data memory, to fetch the instruction; this takes 8ns. The instruction is then sent to the decoder. The delay associated with the decoder might depend on how the synthesizer arranges the gates (it's written in structural Verilog), but it should take no more that 90 ps. So, the fetch and decode phases together take about 8130 ps.

Slowest path: The outputs from the decoder go to a variety of places; at this point, we have to find which path will be the slowest to find the CPU's minimum clock cycle time. Among all the sub modules of the CPU, the ALU and Data Memory have the worst propagation delay times. And among all the CPU instructions, BEQ and BNE are the only two that will go through two ALUs sequentially, and BNE takes one extra NOT gate at the PCSel mux than BEQ. We therefore believe that BNE will have the worst case propagation delay in our CPU design.

The execution flow of BNE will be:
Fetch and Decode(8130 ps) ->
Register File (350 ps) ->
Result ALU (10720 ps) ->
NOT gate (10 ps) -> PCSel Multiplexer (60ps for 3 input mux) ->
PC ALU (10720 ps for ADD) ->
JSel Multiplexer (60 ps for 3 input mux)-> PC (40 ps).

The propagation delay of the critical path in our CPU is 30090 ps. This gives a maximum clock speed of about 33.2 MHz. At one instruction per clock, this is a decent speed for a microcontroller, but too slow for medium- or high-performance applications.

Section 4: Work plan reflection

In general, the time we spent on this lab is consistent with our estimation from the work plan. The creation of individual modules took about 3 hours, which is close to our estimation of 3 hours. The creation and integration of the entire CPU took up the most time, totalling about 8 hours, and we anticipated to spend 3-8 hours on this part.

The creation of assembly test took about an hour, longer than the estimated half an hour, because we ended up making modular tests for individual commands before we could run more complicated tests. The test benches took us about 5 hours to finish, which was a bit more than the estimated upper bound of 4 hours. This is because we

spent some time to figure out how to load memory files correctly in the CPU and to make individual test benches for each module. The scripting at the end took us an hour, because we spent some time to figure out how make files work.

Overall, our work plan reflected the actual number of hours that we spent on this lab, and we are becoming more experienced in estimating the difficulty of the lab.