Multi Cycle CPU

Yichen Jiang, March Saper, Nick Sherman

In this lab we constructed a 32-bit multi cycle CPU with functionalities LW, SW, J, JR, JAL, BEQ, BNE, XORI, ADDI, ADD, SUB, and SLT.

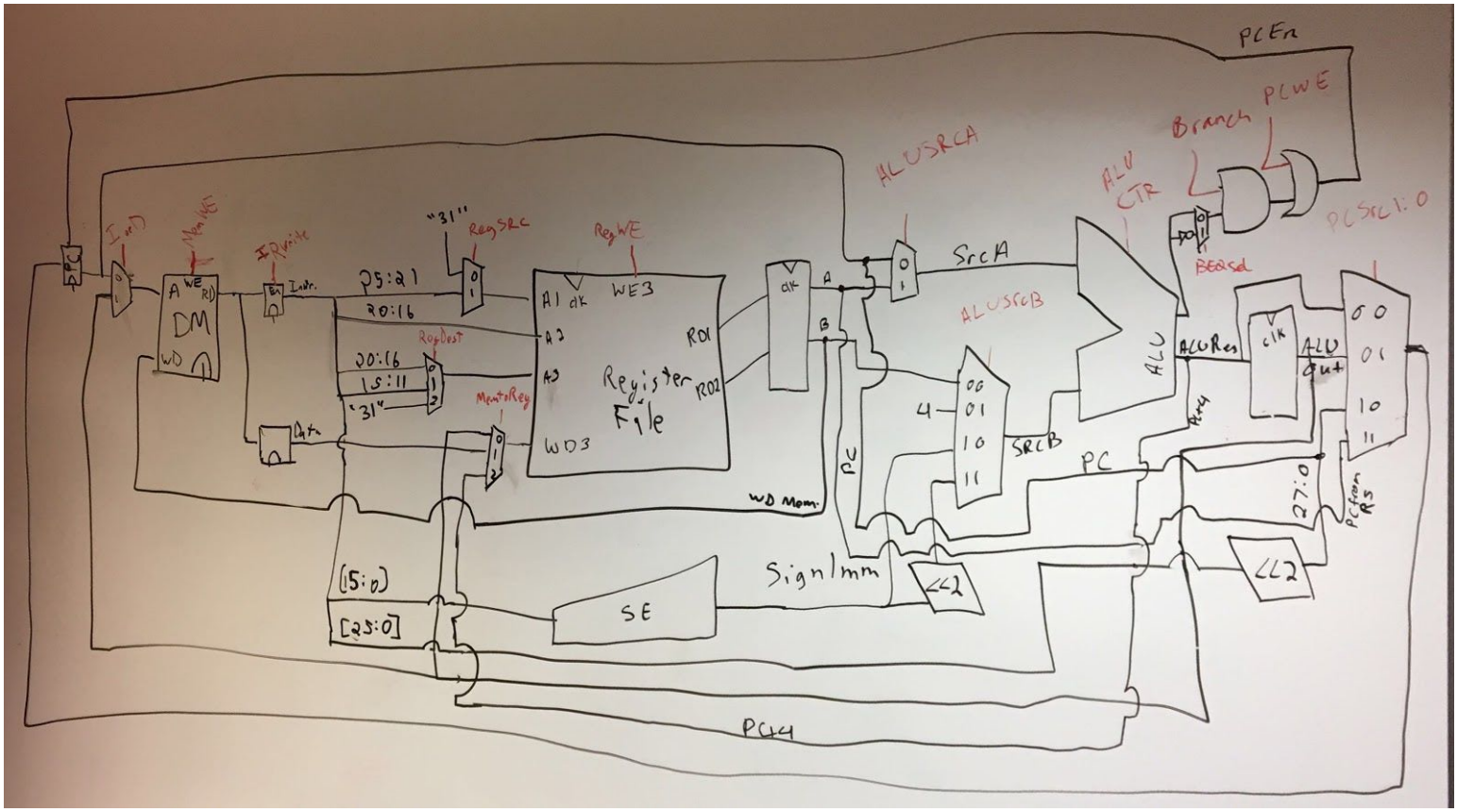
How to Run

In order to run our tests for the entire project, in terminal run ./full_test_runner.sh. This will create an instance of our cpu in verilog as well as instances of testable components, test them with the appropriate unit tests, then remove all created files. You can normally create our cpu through iverilog commands as well.

Section 1: CPU infrastructure

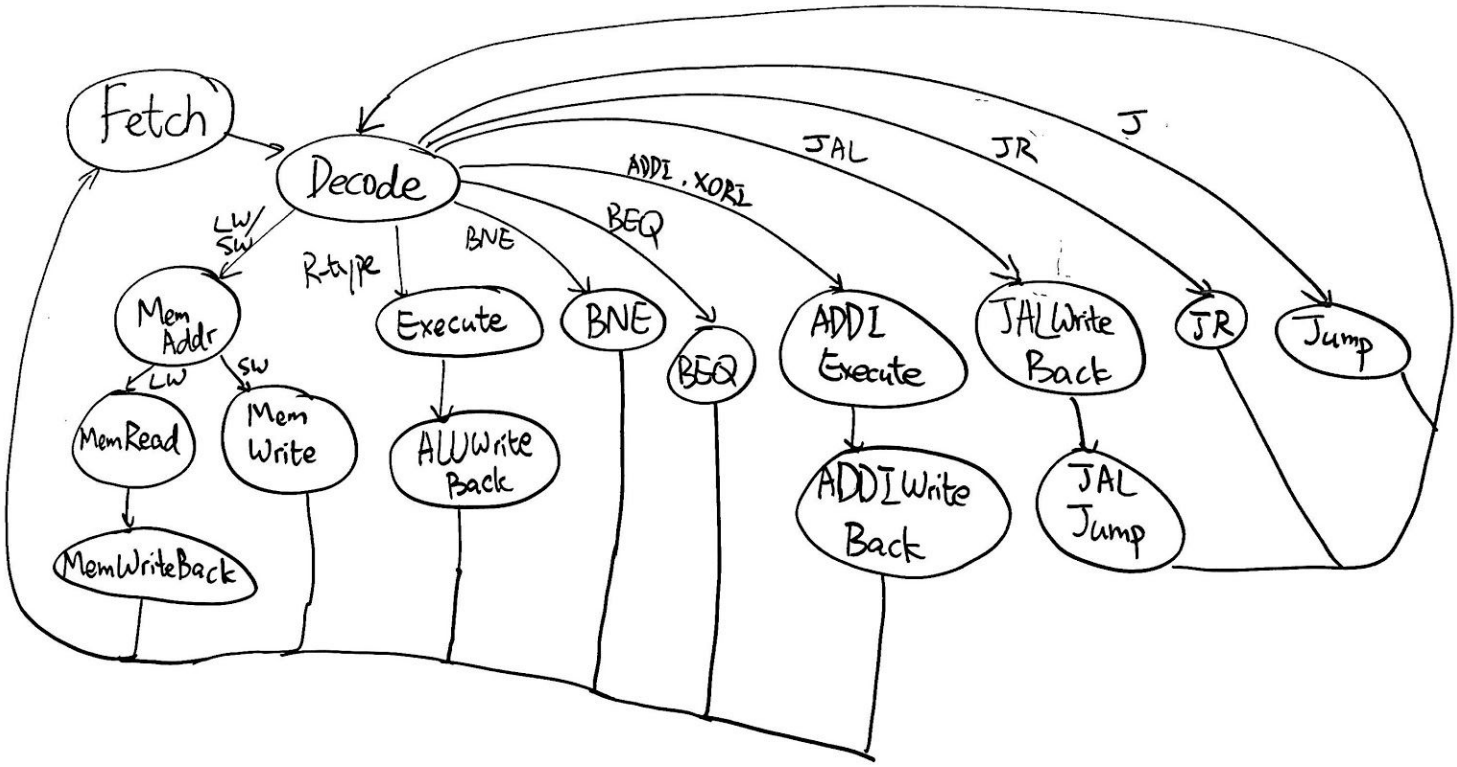
Block Diagram:

The block diagram of our multi cycle CPU is included below. All the control signal that are assigned by the finite state machine are labeled red in the diagram. We added multiple registers between components to separate each instruction into different phases and allow the ALU to be reused multiple times.



Finite State Machine:

The finite state machine that we used to transition between different states:



Control Table

The control table for our FSM:

State	PCWE	IorD	MemWE	IRWrite	RegDest	RegSrc	MemToReg	RegWE	ALUSrcA	ALUSrcB	ALUOP	PCSrc	Branch	BEQSel
Fetch	1	0	0			1			0	1	ADD	0		
Decode				1		1			0	3	ADD			
MemAddr						1			1	2	ADD			
MemRead		1				1								
MemWriteBack					0	1	1	1						
MemWrite		1	1			1								
Execute						1			1	0	R			
ALUWriteBack					1	1	0	1						
BEQ						1			1	0	SUB	1	1	0
BNE						1			1	0	SUB	1	1	1
ADDIExecute						1			1	2	ADD			
ADDIWriteBack					0	1	0	1						
Jump	1					1						2		
JR	1					0						3		
JALWriteBack					2	1	2	1						
JALJump	1					1						2		
Legend														
Enable														
Mux Select														
ALU Op														

Example Instruction Flow

LW: R[rt] = M[R[rs] + SignExtImm]
When the Load Word instruction is inputted, the instruction opcode is passed to the finite state machine which then controls the flags for the remainder of the operation (as for every instruction). The immediate and R[rs] register value are added together through the ALU before being passed to the data memory as an address. The data memory collects the relevant data and passes it to the register to be saved.

BEQ: if (R[rs] == R[rt]) PC = PC + 4 + BranchAddr
When the BEQ instruction is inputted, the ALU subtracts the values of R[rs] and R[rt]. The resulting Zero flag controls the program counter update. Because the program counter is incremented by 4 in the Fetch FSM state (prior to the ALU subtraction) the program counter is already correctly incremented for the case when the if statement is false. A series of AND and OR gates comparing the Zero flag from the ALU and the Branch and PCWE flags from the FSM determine the state of the PCen wire (which will update the program counter with PC = PC + 4 + BranchAddr from the ALU if appropriate).

Add: R[rd] = R[rs] + R[rt]
When the ADD instruction is inputted, the ALU adds the data values of R[rs] and R[rt]. The result of this addition is stored in R[rd].

Section 2: Test Plan & Results

As we were using the same structural pieces behind the CPU that Nick and Yichen's group used in Lab 3, the tests for the alu, muxes, regfile, and sign extender were all reused from the last lab. The assembly tests to test the overall structure (Hanoi, Fibonacci, Yeet, Test_1, Test_3, and slt) were the same set as last time. These can all be run by running ./full_test_runner.sh.

As we were debugging, we realized we needed to have some additional tests to more closely analyze the different cpu commands. As such, the series of tests run in cpucmds.sh tests branching, jumping (J, JR, and JAL), immediate operations (ADDI/XORI), and non-immediate operations. All expected results are contained in cpu.t.v. We also improved our test bench by enabling easy flagging of tests which return all "X"s from registers without values. Although these tests were incorporated into full_test_runner, these tests can be individually run by uncommenting the two lines. The new series of tests (with associated tested commands) are:

- Branch_test
    - Primarily tests: beq
    - Secondarily tests addi; j; add
- Jump_test
    - Primarily tests: jal; j; jr; xori
    - Secondarily tests: addi; add
- Simple_sl
    - Primarily tests: sw; lw
    - Secondarily tests: addi; j
- Simple_test
    - Primarily tests: addi
    - Secondarily tests: j
- Sw_test
    - Primarily tests: j; jal; lw; sw
    - Secondarily tests: addi
- Slt_test
    - Slt_test was actually the same as what we used last lab

- Primarily tests: slt
- Secondarily tests: addi; j

Section 3: Performance Analysis

For the performance analysis of the multi-cycle CPU, we are interested in finding the slowest phase in our design so that we can determine the fastest clock speed at which the CPU could run. The time analysis of each single component is taken from our Lab 3 work and is briefly shown below. We assume that our approximation will have a 10% error margin.

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. 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): The mux will take 30 + 10*n ps, where n is the number of inputs.
Register file: We assume that the register file will take about 350 ps to read.

The time analysis of the each state are discussed below.

Instruction Fetch: The data flow for this state is from the PC register to the ALU then got saved back to the PC. It takes about 40ps to read from PC, 10.72ns to do ADD in the ALU and 40 ps to save back to the PC. The total elapsed time is 10.8ns.

Instruction Decode: Instruction decode is split into two clock cycles. In the first clock cycle, the instruction is loaded from the data memory, which takes 8ns to complete. In the second clock cycle, the data is loaded from the register and the PC branch value is calculated for a possible Branch operation. Register read takes 350ps and ALU ADD operation takes 10.72ns, and the second clock cycle, in total, takes 11.07ns to complete.

Execution: The flow for execution revolves almost exclusively around the ALU. There is a 4-input multiplexer before being fed to it (which is 30 + 10*4 ps or 0.07ns), but the main execution delay is form the ALU. As the longest ALU operation is SLT (at 10.78ns). Put together, the longest amount of time that could be spent on execution is 10.78 + 0.07 or 10.85ns.

Memory Write: The memory write state is controlled by the output of the register file's second read port being sent to DM. The time to write to DM, assuming a random-access time of 8ns, is approximately 8 ns. Therefore, the entire time spent in memory write is 8 ns.

Register Write Back: The register write back state writes the calculated value back to the register file, and takes about 0.35 ns to complete.

Among all these states, the second clock cycle of the **Instruction Decode** state takes the longest to complete (11.07ns) and should determine our CPU clock speed. The runtime are generally well distributed across for all states except **Register Write Back**, which only takes 0.35ns to complete.

Section 4: Work Plan Reflection

For the first time, the work plan worked out fairly well. The team did a good job dividing work and front loading such that our CPU was completed and passing all necessary tests on Tuesday. We didn't waste that much time as a group and were able to work together efficiently, often pairing up to ensure that components were hooked up correctly. We also debugged as a group to help expedite how quickly bugs were caught and fixed. Overall, the time was broken down as:

- First attempt at making a good FSM/diagram: 1.5 hours
- Reading up more about multicycle CPUs to better inform ourselves of how to create the CPU: 1 hour
- Redoing the FSM/CPU infrastructure: 2 hours
- Implementing a first-pass of the CPU infrastructure: 2.5 hours
- Debugging the first-pass CPU infrastructure and updating it while updating tests: ~4.5 hours
- Writing up the report, neatening/commenting code: ~2 hours

We ended up getting lucky with our debugging thanks to good structuring techniques learned through earlier labs, and overall were able to complete the lab faster than expected.

Section 5: References

Harris, David Money Harris, Sarah L.. (2007). *Digital Design and Computer Architecture - 7.5 Pipelined Processor.* (pp. 381-400). Elsevier. Retrieved from https://app.knovel.com/hotlink/pdf/id:kt00951F84/digital-design-computer/pipelined-processor