CompArch Lab 3

Vivien Chen, Lauren Pudvan, Samantha Young

November 2, 2018

1 Running our CPU Verilog tests

Please see our HOW_TO_RUN.md in https://github.com/vivienyuwenchen/Lab3.

2 CPU

2.1 Schematic

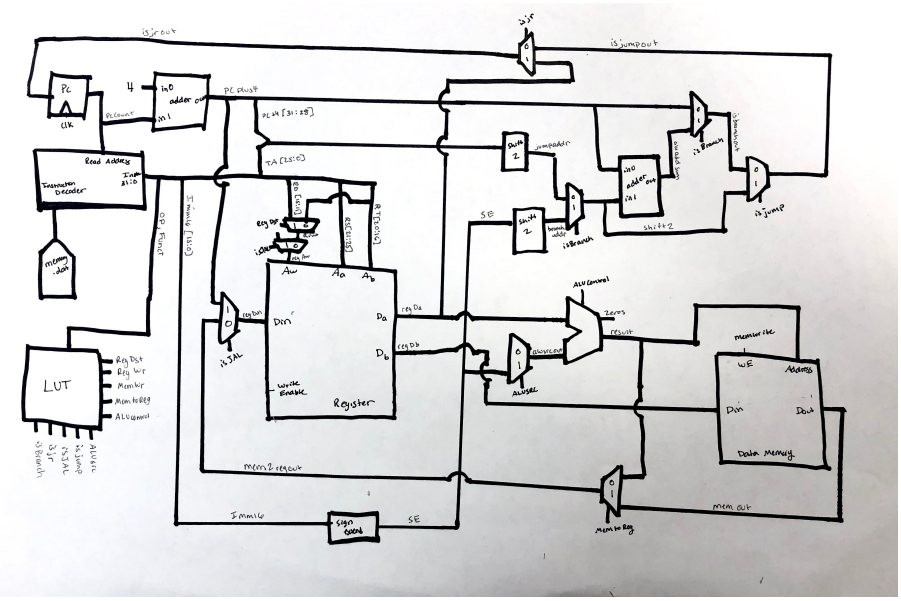

Figure 1: This is a diagram of our single cycle CPU Schematic. The only difference between this and our code is that the memory aspect of the instruction decoder is included in the data memory module. We chose to keep them separate in the drawing to make the schematic easier to understand.

2.2 Instruction LUT

	RegDst	RegWr	MemWr	MemToReg	ALUctrl	ALUsrc	IsJump	IsJAL	IsJR	IsBranch
LW	0	1	0	1	000	1	0	0	0	0
SW	0	0	1	0	000	1	0	0	0	0
J	0	0	0	0	000	0	1	0	0	0
JR	0	0	0	0	000	0	0	0	1	0
JAL	0	1	0	0	000	0	1	1	0	0
BEQ	0	0	0	0	001	0	0	0	0	*
BNE	0	0	0	0	001	0	0	0	0	*
XORI	0	1	0	0	010	1	0	0	0	0
ADDI	0	1	0	0	000	1	0	0	0	0
ADD	1	1	0	0	000	0	0	0	0	0
SUB	1	1	0	0	001	0	0	0	0	0
SLT	1	1	0	0	011	0	0	0	0	0

*If the zero flag is 1 and overflow is 0, IsBranch is set to 1 for BEQ and 0 for BNE; otherwise, it is 0 for BEQ and 1 for BNE.

2.3 Description

We created our CPU schematic incrementally. Focusing on how to implement R type instructions, then I type, then J type. We broke up our CPU design into a instruction decoding section, an instruction execution section, and a jumping/ branching section.

The schematic for the CPU was designed directly from the RTL as specified by MIPS. The CPU needs to complete the following operations, LW, SW, J, JR, JAL, BEQ, BNE, XORI, ADDI, ADD, SUB, SLT.

These operations are defined by the following RTL.

ADD : R[rd] = R[rs] + R[rt]
ADDI: R[rt] = R[rs] + SignExtImm
SUB : R[rd] = R[rs] - R[rt]
SLT : R[rd] = (R[rs] < R[rt]) ? 1 : 0
XORI: R[rt] = R[rs] XOR SignExtImm
BNE : if(R[rs]!=R[rt]); PC = PC + 4 + BranchAddr
BEQ : if(R[rs]==R[rt]); PC = PC + 4 + BranchAddr
J : PC = JumpAddr
JR : PC = R[rs]
JAL : R[31] = PC + 4; PC = JumpAddr
LW : R[rt] = M[R[rs] + SignExtImm]

SW : M[R[rs] + SignExtImm] = R[rt]

The CPU is controlled by a program counter that indexes the instruction memory. The program counter is 32 bits, however the memory is indexed by addresses of 12 bits. In our memory each index corresponds to 4 bytes of instruction therefor we must divide our counter by 4 in order to index through the memory properly. The instruction memory holds 32-bit instructions. The instruction decoder breaks apart the 32-bit instructions into into sections based upon the type of instruction. Our Look Up Table utilizes the opcode and funct code to set the control signals.

The ALU and Memory perform operations based on the instruction code. Then the ALU would perform any necessary operations to calculate the data in to the register or the address for the memory, depending on what the instruction specified. As specified by the above RTL many of the operations are reliant on adding data from two registers or adding a register and an immediate. Much of this adding is done with an ALU. Additionally the other operations like SUB, XORI and SLT utilize the other functionalities of the ALU. BEQ and BNE were computed by subtracting the two registers in question with the ALU and then inspecting the ZEROS flag and OVERFLOW Flag. If the zero flag was raised and overflow was not raised then the two registers were equal. If not then they were not equal.

Operations like SW and LW interact directly with both the ALU and the memory. First an address is computed by addition through the ALU. Then depending on whether it is LW or SW, data is either loaded or stored to that address.

We have a series of adders and muxs to handle the jumps and branches.

Figure 2 shows the path that an ADD instruction would follow using our CPU.

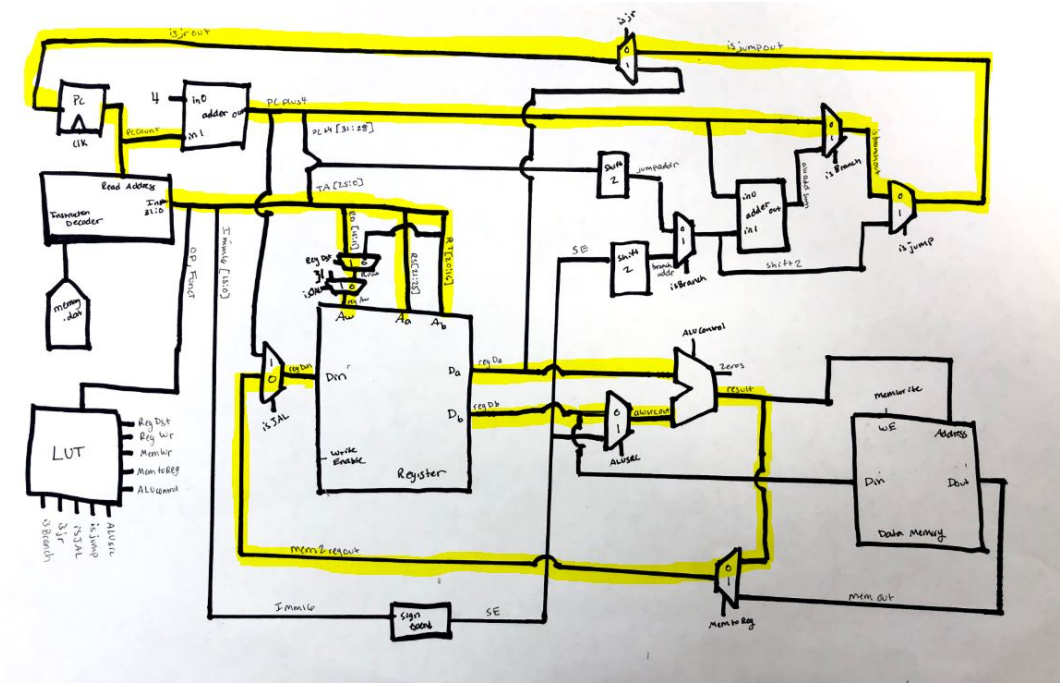

Figure 2: The highlighted path follows that of an ADD coded instruction

3 Testing Plans

Each module is used in our CPU includes a test bench to insure the quality of its functionality. Through these modular test benches we found several bugs in our seemingly working modules.

We pulled in our old test benches for the ALU and regfile. Although the ALU test bench passed all of its tests, we realized after we implemented the ALU into our CPU that the delays we modeled for Lab 1 messed up our CPU process and had to delete them accordingly.

The DFF and MUX2 were simple building blocks, so we only tested the function of enable and select, respectively, in determining their outputs.

The instruction decoder and look up table had similar functions. Our test benches made sure that the output register fields and control lines matched what we expected for each instruction or operation.

Originally, our memory was split into data memory and instruction memory. After combining them, we wrote a test bench that read in our addN.dat file and output the correct instruction given an instruction address, as well as write to a

data memory location given an address if write enable is true.

The testbenches we wrote for the full CPU focused on testing the functionalities of the 13 intended operations, LW, SW, J, JR, JAL, BEQ, BNE, XORI, ADDI, ADD, SUB, SLT. Each test bench focused on a different set of operations so that when run in full all benches would together cover all of the operations. These tests were written in assembly and loaded into the instruction memory in verilog. MARS was used to simulate the assembly and determine the expected outputs. These expected outputs were then compared against GTKWAVE waveforms for debugging purposes.

fibfunc.asm tested the fibonacci sequence. It was pulled from the NINJA tests on the class Github. It ensured the functions of ADD, ADDI, BNE, LW, SW, J, JAL, JR. The final answer can be found in $v0$ = 32'h3a.

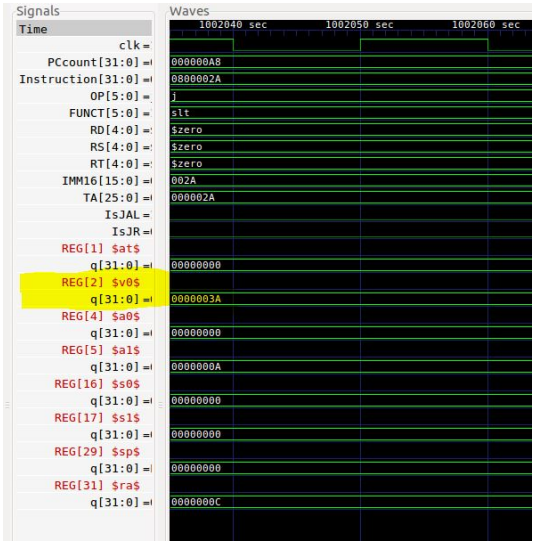

Figure 3: The waveform output for the fibonacci test bench

addNintegers.asm loops through the numbers 0-10 summing each together. This tests the BEQ, ADD, ADDI, J functionalities of the CPU. The final answer can be found in $t1$ = 32'h37

xor_sub_slt.asm is a basic test that computes XOR SUB SLT sequences. This tests the ADDI, XORI, SUB, SLT, BNE, J functionality. Outputs are found in $t0$ through $t6$ Where:

$t0$ = 32h'1

$t1$ = 32h'1

$t2$ = 32h'7

$t3$ = 32h'fffffffb
$t4$ = 32h'3
$t5$ = 32h'fffffff2
$t6$ = 32h'fffffff7

We heavily relied on GTKWAVE for debugging and understanding failed tests. We stepped through a code in MARS in conjunction with stepping through the waveform in order to understand what functions or operations were not working as expected. By seeing where our waveform was failing and working backwards we were able to find elusive bugs like unnecessary propagation delays in our ALU, improperly initialized counters, and wrong address calculations.

4 Area Analysis

In our design there used to be the main ALU and then 2 more ALUs to assist in adding for the jumps/branches and the PC counter. We decided to make these other two ALUs into adders. There was no need for a device that could do more than adding, like the ALU. This change reduced the size of those components by roughly 1/2. We made Our ALU in behavioral verilog, so this is our best guess of an ALU cost.

The ALU cost per bit:

Task	Cost per bit
Add/Sub/SLT	16
XOR	2
And	2
NAND	1
OR	2
NOR	1
TOTAL	24

The adder cost per bit:

Subcomponent	Cost per	Number Used	Total
2XOR	3	2	6
2AND	3	2	6
2OR	3	1	3
			15

5 Work Plan Reflection

This is a summary of our original work plan:

- Obtain good code for ALU, shift register, data memory, mux, other basic building blocks - 1 hour - DUE OCT 22nd
- Design on paper CPU, figure out modules required - 2 hours - DUE OCT 24th
- Write instruction modules and respective test benches in verilog - 5 hours total - DUE OCT 24th
- Master Testing Suite - 3 hours - DUE OCT 24th
- Runs all modules test bench - 1 hour
- Debugging - 2 hours

We started off staying on schedule. Every aspect of the Lab (except for debugging) took within half an our what what we expected it to take. We think these sections were really well scoped! We feel like we are getting a hold of the time it takes to make schematics and pull together modules. Hopefully, this will continue to get even more accurate in the future lab and project.

Debugging took us around 16 hours. This is an order of magnitude more than we expected. Overall we will make sure to give much more time assigned for debugging. It is difficult to predict the unknown. This is probably why the debugging was scoped so poorly. In the future we will scope out much more time for debugging. Which will be more important as the labs and project get more complex.