Lab 3: Single Cycle

Abby Fry, Grace Montagnino, Vienna Scheyer

November 5, 2019

Introduction

In this lab we developed a single-cycle 32-bit CPU to execute a subset of general MIPS instructions. The instructions we aimed to implement were: LW, SW, J, JR, JAL, BEQ, BNE, XORI, ADDI, ADD, SUB, SLT. As our CPU is single cycle, only one instruction can be executed on every clock cycle.

Architecture

Written description and block diagram of your processor architecture. Consider including selected RTL to capture how instructions are implemented.

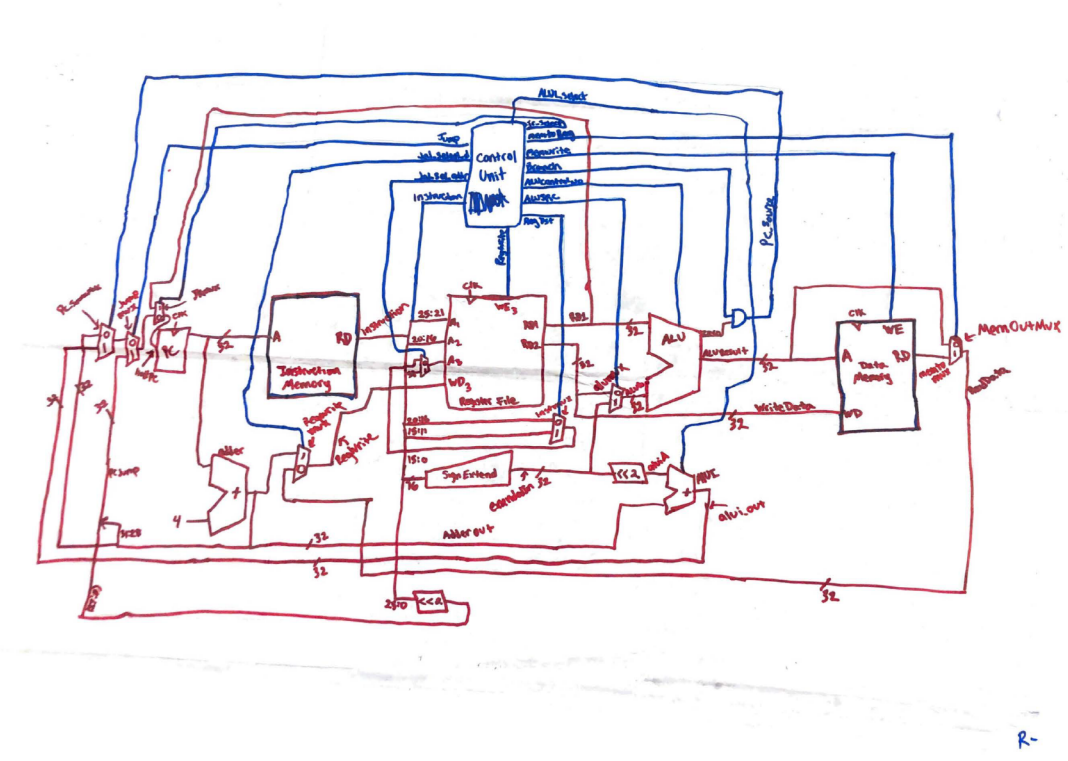

Figure 1: This is the final schematic and Control Unit structure that we have implemented.

We referenced the book linked on Canvas called Digital Design and Computer Architecture by Harris Harris for help getting started. We built off of this model and added more instructions that weren't in the

book.
Since this is the most complex circuit we have implemented yet, so we tried to be intentional about our approach to designing the circuit. To do this, we started with LW and SW, and then progressively added circuitry for adding and jumping operations. This served us well because firstly, it helped with our understanding, and secondly, it reduced the possibility for errors in our schematic drawing process.

Program Counter

The program counter gets reset to zero at the beginning of each program run, and then from there it has a few options for the next PC value. For instructions like ADD, ADDI, and XORI, the next PC value is $PC+4$ because when it finishes the current instruction it just goes on to read the next instruction in memory (each instruction is 4 bytes long). In hardware, this just looks like an ALU that is always wired to add 4 to the current program counter. For instructions like BNE, BEQ, J, the PC finishes the current instruction and then goes to the calculated address. For JR, the PC jumps to the address saved in the register $s. For JAL, the PC jumps to the calculated address and saves the current address in register #31 so that it can return to it's current address later (this is the 'link' part). To implement these options for the PC, we used muxes that take signals from the Control Unit to choose the next PC value.

Control Unit

We took all of our control signal and enable wires for our CPU and organized them look up table style in order to implement our logic at the desired times. You can see an excel version of this look up table below.

MIPS Instruction	OP Code	jump	jr_sel	jal_sel_d	jal_sel_addr	mem_to_reg	reg_write	mem_write	reg_write_en	branch	alui_sel	alu_sel	alu_src_sel	RegDst
LW	"100011"	0	x	0	0	x	0	1	1	x	x	x	x	rt
SW	"101011"	0	x	0	0	1	1	0	0	x	x	x	x	rs
BEQ	"000100"	0	0	0	0	x	0	0	1	1	x	SUB	1	x
BNE	"000101"	0	0	0	0	x	0	0	1	0	x	SUB	1	x
J	"000010"	1	x	0	0	x	0	0	0	x	x	x	x	x
JR	"001000"	1	1	0	0	x	0	0	1	x	x	x	x	x
JAL	"000011"	1	0	1	1	x	0	0		x	x	x	x	x
ADD	"100000"	0	x	0	0	x	0	0	1	x	x	ADD	1	rd
ADDI	"001000"	0	x	0	0	x	0	0	1	x	ADD	x	x	rt
XORI	"001110"	0	x	0	0	x	0	0	1	x	XOR	x	x	rt

Figure 2: This is the look up table logic that we have implemented.

The table goes through each MIPS instruction and denotes which control lines should be what in order to implement that instruction. To implement this in verilog we used case ideology to separate R type instruction from the rest. We had to do this because unlike other types of instructions R types really hold their directions in their FUNC block, not their OP block. In other words, we would look at all the OP codes given with the input instructions, and see whether the OP code was indicative of an R type, or of a specific instruction. In the case where it was R type, we then had to move to looking at the FUNC code to translate the instruction. There is a screenshot of the LUT logic below as implemented in verilog to give the visual for what we are describing above.

```
initial begin
  jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b0;branch=1'b0;alu_sel=1'b
end

always @(instruction)begin

    case (opcode)
        `LW:   begin jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b1;bran
        `SW:   begin jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b1;mem_write=1'b1;reg_write_en=1'b0;bran
        `BEQ:  begin jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b0;bran
        `BNE:  begin jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b0;bran
        `J:    begin jump=1'b1; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b0;bran
        `JAL:  begin jump=1'b1; jr_sel=1'b0;jal_sel_d=1'b1;jal_sel_addr=1'b1;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b1;bran
        `ADDI: begin jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b1;bran
        `XORI: begin jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b1;bran
        `R:begin
             if (funct==`JR) begin jump=1'b1; jr_sel=1'b1;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_wri
             else if (funct==`ADD) begin jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;r
             else
               jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b0;branch=1'
       default: begin jump=1'b0; jr_sel=1'b0;jal_sel_d=1'b0;jal_sel_addr=1'b0;mem_to_reg=1'b0;mem_write=1'b0;reg_write_en=1'b0;br
     endcase
   end
endmodule
```

Figure 3: This is our verilog LUT logic, yes it is cut off, but that is because you only need to see the logic part, not the setting of the control lines.

The final component to our control logic is the default state which we needed to initialize the control signals such that the components needing control lines would be able to work right off the bat.

Memory and Register File

The memory in our CPU is comprised of instruction memory and data memory. Data memory grows down while instruction memory grows up. The implementation used in our lab was provided in the starter code for lab 3. The register file is a type of memory that the CPU uses to store values that are calculated during the execute stage. Our register file is comprised of 32 individual 32-bit registers where the first register is hard coded to zero. Our register can read two registers and write to one register at a time. The register used in the implementation fo our CPU is from HW 4.

Instruction Walk-Through Examples

Because there are so many MIPS instructions we decided to just walk through a subset in this report. We will explain the data path for one instruction of each type.

R Type – ADD

The equation for the MIPS ADD function is R[rs]+R[rt]=R[rd]. In our datapath, this process starts by the instruction being read from memory as usual. Then the instructions are broken into rs and rt, both of which get fed into the register file. The register file outputs the values of those registers. In the execute stage, we add together rs and rt with an ALU and then send the results back to the register file to get written in at rd.

J Type – JUMP

The MIPS equation for JUMP is PC=JumpAddr. Once a J Instruction comes through memory we use muxes in the PC datapath to select the circuit that includes a bit shifter, instead of the path that includes the ALU. The bit shift by two multiplies the current address by 4, which leaves us with 28 bits, the remaining

four bits are the MSD of the previous PC. Unlike the R-type instructions, Jump does not need to write to memory to the register file because it is only updating the PC.

I Type – ADDI

The MIPS formula for ADDI is R[rt] = R[rs] + SignExtImm. Addi relies on the first register in the register file which is hard coded to zero and adds that zero to a constant value. The constant value in question is in bits 15:0 of the instruction, so we sign extend these 16 bits to be compatible with our 32-bit ALU. Before entering the ALU, the value is bit shifted by 2 to the left (i.e. multiplied by 4) to get to the new program counter value. The result is written to another register in the register file.

Test Plan and Results

To make this CPU, we made sure each element individually was fully functional before working on the integration and actual wiring of the CPU. After each element had been tested individually, we worked on wiring the whole CPU.

After we finished the wiring, we started testing the CPU with asm tests. When we first started running ASM tests, we were getting a lot of x's. Upon looking into gtk wave, we were able to see that our program counter was not properly incrementing. The ALU we were using to try and increment our PC by 4 appearing to be passing it's test bench but further investigation showed it wasn't actually adding properly. To resolve this and future problems, we used a different groups ALU. After resolving this, our adder was still not working. This led us to suspected it had something to do with our setting of the default case properly passing in default values to allow the CPU to run. Upon some review from Ben, we learned that we were instantiating the default wrong, and after some edits to the LUT were able to move beyond that.

Once our PC was working, we were able to look past this, and look at how our instructions were being processed. At first, it appeared that our CPU would only read in a max of three instructions. We later learned that this was because our JAL was not properly working, so the PC was being forced to jump to the end of the program which was in turn resulting in all x's. Upon suggestion, we tried running a simpler asm test, one that consisted of only add operations, which wouldn't require anything crazy of our PC or involving jumps. While this did allow us to read all of the instructions (not just the first three), when we went to look inside the register where the answer should have been saved, there was noting there. This leads us to believe that for some reason we are unable to write to the register we are think we are writing to? But at this point we are out of leads to chase, and would need outside help in order to get this resolved.

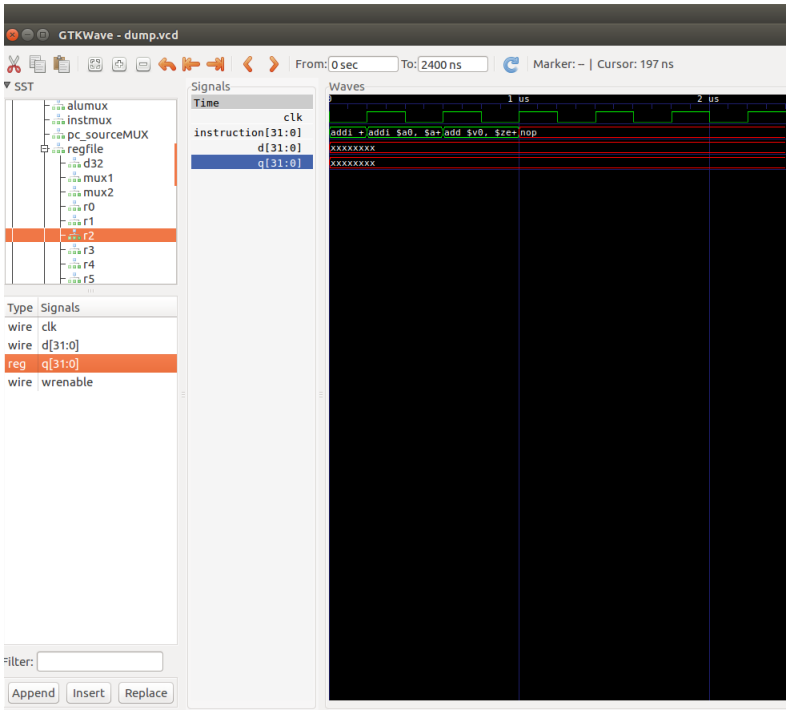

Figure 4: GTKWaveof our CPU instruction over time as well as the input and output of register 2 in the register file. Register 2 is where we expected to see a result.

Performance Analysis

Some performance/area analysis of your design. This can be for the full processor, or a case study of choices made designing a single unit. It can be based on calculation, simulation, Vivado synthesis results, or a mix of all three.

The performance of a single-cycle CPU is not optimized because it allows as much time as is needed to execute the most time-costly instruction for every type of instruction. In the future, we could improve the single-cycle CPU by making it multi-cycle and using pipelining architecture. This would be more efficient because it would have multiple phases running simultaneously.

Reflection

Our work plan for this lab was to work as often as possible, talk a lot with Ben, and try to be more organized in our approach. We broke the lab into a few sections/categories we expected to pass through. Below you'll find our time estimates for each section of the lab, along with more accurate estimations of time spent on each part. Because we were working on a shorter timeline, our work plan was pretty much to try and finish one of these categories each day. But because we often needed Ben or a NINJA's input this did not work out so cleanly...

Task	Expected (hrs)	Reality (hrs)
Generate block diagram and data path	1.5	4
Create individual components and test	2	3
Integrate Components	3	6
Write Test Bench for Integration	2	2
Test Module with class assembly tests	1	∞
Write Report and Document	1.5	2

Because we had to spend all of our time from the first week of this lab on finishing the last lab, we had to squeeze a lot into 9 days. Considering that we were working on a shorter timeline, we are fairly happy with where we got. We believe that on this trajectory, given two weeks, we would have successfully finished this lab. But in order to prevent us from continuing on this path of always being behind, we decided to stop and turn this lab in unfinished. We feel like we were very proactive during this lab, we talked with Ben pretty much daily, met everyday, and tried to do things in a logical and organized way. We feel like we have a decent handle on the material despite not finishing the lab. If we were to do it again, we probably would have not spent the time fixing the multiplier, and instead spent our time on the CPU. But at the time we thought that finishing the multiplier was significant, so we did that instead. Looking solely at the time spent on this lab, it was much less chaotic and murky than previous labs, so despite this resulting in an unfinished lab, we do feel like we improved and learned.