Comparch Lab 3: Central Processing Unit

Lisa Hachmann, Tom Heale, Anisha Nakagawa

November 2016

1 Introduction

For the Central Processing Unit (CPU) lab, we implemented a single cycle CPU in Verilog. The single cycle CPU had to be able to execute the following instructions: load word, store word, jump, jump to register, jump and link, branch if not equal, XOR (exclusive OR) with immediate, addition, subtraction, and set less than. These instructions are all found in the MIPS Instruction set, which we referenced. The limited instruction set has manipulations of I, J and R-type instructions, and were implemented exactly according to the MIPS specifications, see table 1. The main difference in our implementation is that our program counter increments through the instructions by the address of the instruction in memory, rather than by bytes (which is the MIPS standard). Because of this, we increment the program counter by 1 address every time, rather than by 4 bytes. We also scaled the jump and branch addresses accordingly in our implementation.

Table 1: MIPS Instructions Explanation

Instruction	MIPS Format	Behavior	Binary Encoding
LW	I-type	Reg[rt] = Mem[Reg[rs]+imm]	1000 11ss ssst tttt iiii iiii,iiii iiii
SW	I-type	Mem[Reg[rs]+imm] = Reg[rt]	1010 11ss ssst tttt iiii iiii,iiii iiii
J	J-type	PC=JumpAddr	0000 10ii iiii iiii iiii iiii,iiii iiii
JR	R-type	PC=Reg[rs]	0000 00ss sss0 0000 0000 0000 0000 1000
JAL	J-type	PC=JumpAddr Reg[31]=PC+8	0000 11ii iiii iiii iiii iiii,iiii iiii
BNE	I-type	if(Reg[rs]!=Reg[rt] PC=PC+4+BranchAddr	0001 01ss ssst tttt iiii iiii,iiii iiii
XORI	I-type	Reg[rt]=Reg[rs] XOR imm	0011 10ss ssst tttt iiii iiii,iiii iiii
ADD	R-type	Reg[rd]=Reg[rs]+Reg[rt]	0000 00ss ssst tttt dddd d000 0010 0000
SUB	R-type	Reg[rd]=Reg[rs]-Reg[rt]	0000 00ss ssst tttt dddd d000 0010 0010
SLT	R-type	Reg[rd]=(Reg[rs][rt])?1:0	0000 00ss ssst tttt dddd d000 0010 1010

2 CPU Design

The single cycle CPU can be described by the block diagram in figure 1. Each of the major modules are represented, such as program counter, memory, register file and ALU. Their control logic unit along with the rest of the modules each have much more detailed wiring, shown in the schematic. In figure 1, you can see that the program counter iterates through instructions unless instructed (by a branch or jump instruction) to move to another instruction. After the instruction is found and decoded, the rest of the modules either compute, store or load based on the rest of the instruction. Then, the next instruction is executed, similarly. Everything is done in one cycle, and no parallel computing is done.

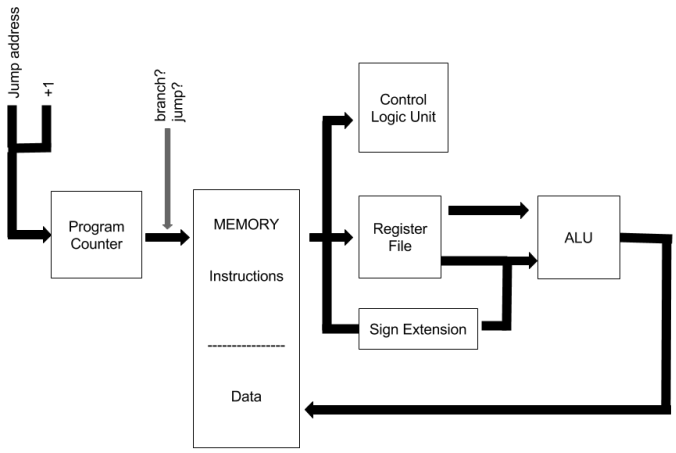

Figure 1: Block Diagram of the single cycle CPU

The full diagram with the wiring is shown in the schematic.

3 Schematics

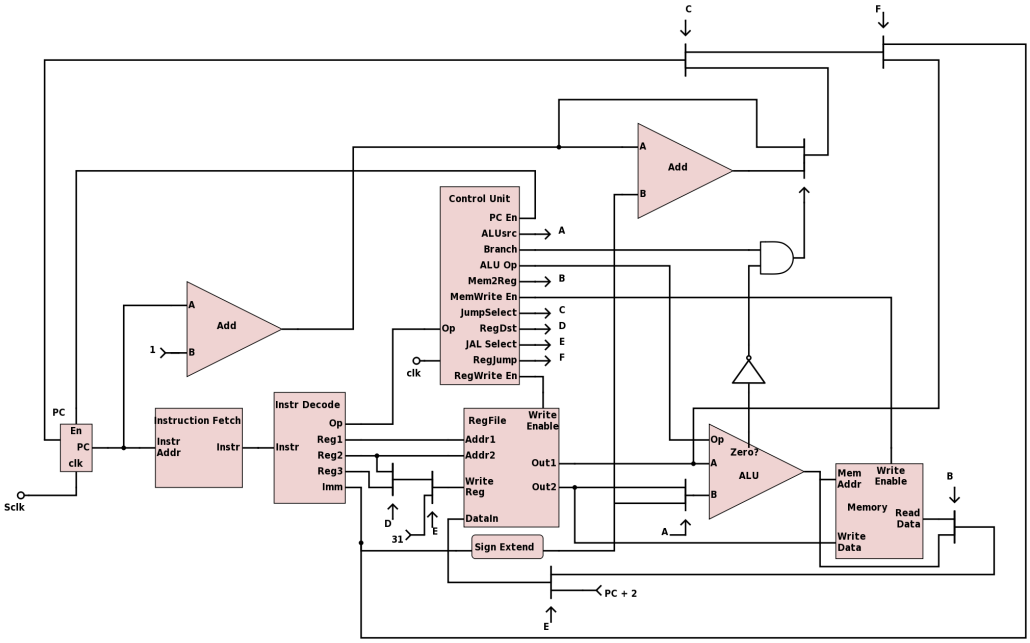

Figure 2: The full schematic for our single cycle CPU. It is modified from the standard block diagram.

Our full CPU design can be seen in 2. Our biggest changes from the standard block diagram we learned in class are for our jump procedures. We inserted muxes that can select between iterating our program counter by 1 and jumping to a new instruction from an immediate input, either from a branching or jumping instruction. This changed our controller to ensure we could select between options properly. There are also

new muxes that allow the jump and link instruction to perform correctly, like the two muxes that allow jump and link to perform properly and have the pc+2 instruction saved directly into register 31 (with 31 being implied from the instruction).

We don't have schematics for either of our memory units, Instruction Memory or Data Memory, because they were implemented in behavioral verilog and simulate memory using a text file.

Previously, in Homework 4 we made a register file, which we implemented here as part of our CPU. The schematic is shown in Figure 3 and was taken from the class GitHub repository.

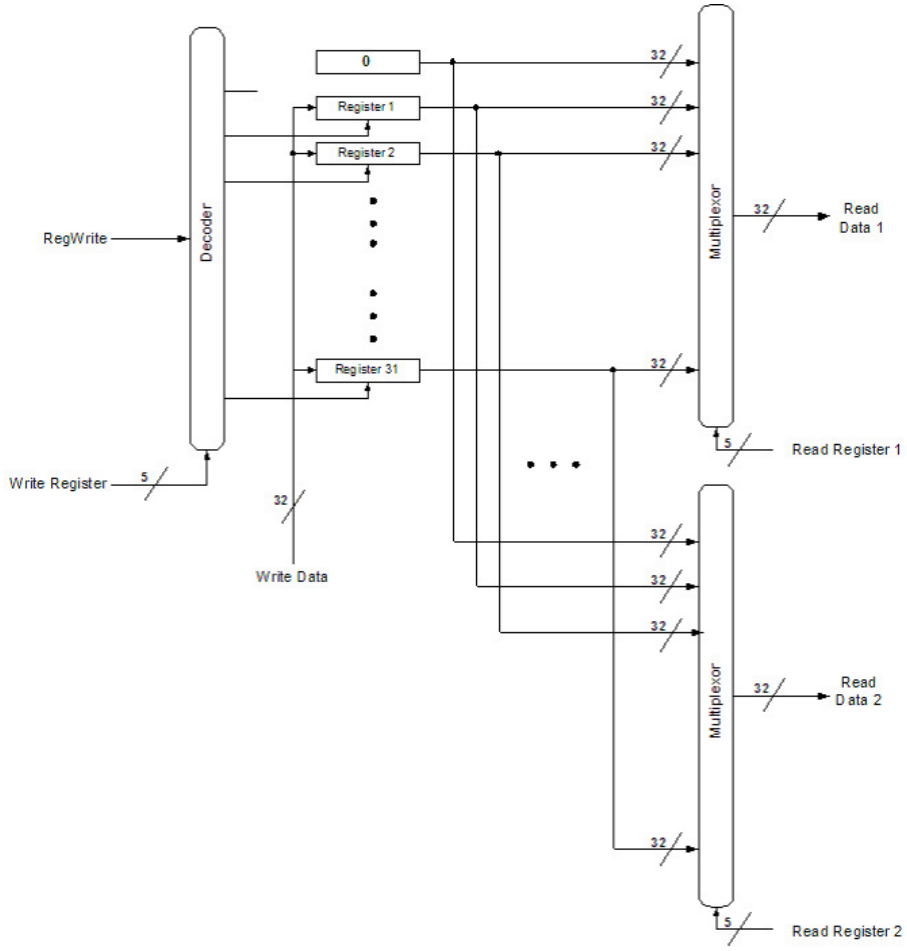

Figure 3: We reused the register file we made for Homework 4 and this is the schematic for that implementation

Another difference between our implementation and the version we learned in class is that we implemented a slow clock at 1/4 the speed of the system clock. We used this slower clock as input to the PC module, and it ensures we don't start a new instruction before we are able to branch/jump. Our slow clock module schematic is shown in Figure 4 on the next page

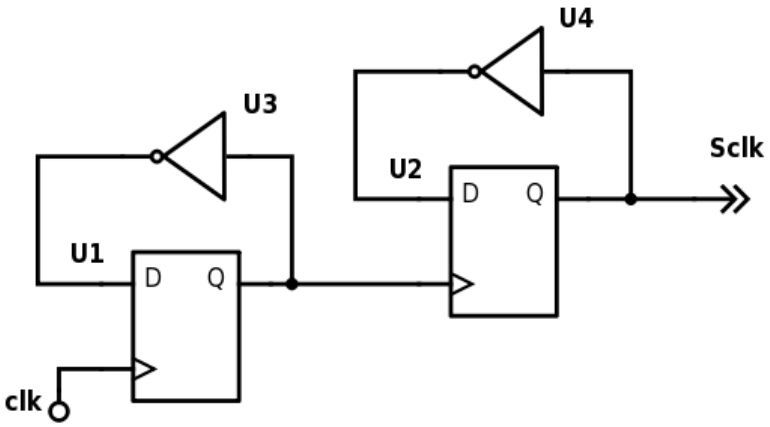

Figure 4: Schematic of our slow clock implementation that runs our PC block.

4 Test Plan

In general, we were able to test the components inside the CPU with individual test benches. For the modules that were previously implemented (ALU and register file), we used the modules and their test benches from the previous labs. Some of those test benches don't have the "DUT passed" structure, but we had previously verified that they behave as expected. For the modules made for this lab specifically, we made individual test benches that included having the "DUT passed" flag to show pass/fail. Every component within the CPU was tested with test benches and then the CPU was overall tested with the following assembly instructions and verified with the waveform. The waveform verification will be explained in section 4.2.

4.1 Assembly Instructions

To test our CPU, we wrote assembly code that implemented every instruction in our instruction set. The psudo-assembly code below walks through the test. To see the full assembly code in hex, see fullMem.dat, the first lines up to ffffffff. To see the full explanation of the binary used to create the instructions in hex, please see TestingInstructions.txt. We built the instructions using the binary format in table 1, then converted the binary to hex. We wrote the instructions directly into binary, because our addresses in memory work differently than the default addresses in MARS assembly.

In order to load numbers into our registers, we used XORi.

- xori register zero with the immediate 6, save into register 5
- xori register zero with the immediate 2, save into register 2
- xori register zero with the immediate 2, save into register 1

Now we tested branching with adding.

- add register 2 and register 1, save (and overwrite) register 2
- branch not equal: if register 5 (holding 6) is not equal to register 2 (holding 4), branch to instruction 3 (indexed from 0, adding 2 again). Once it is, continue with instructions.

Now, we'll test the rest of the computations: set less than and subtract

- sub register 1 from register 5 (should be 4), save into register 3.

- Stl (set less than): set register 4 equal to 1 if register1's contents is less than register 2's contents (it should be). If not, set it to 0.

Now, we'll test storing, loading and the various jump tests.

- sw, Store the contents of register 2 to memory at address 22.
- lw, load the contents of memory at address 22 to register 7.
- jump (the 9th instruction) says to jump to the 11th instruction.
- There is an xori instruction here, but is skipped because it is the 10th instruction.
- Jump and link, jump to the 13th instruction and save the address of the next-next (in 2) instruction to register 31.
- There is an xori instruction here, but it is skipped by the jump as it is instruction #12.
- xori register zero with the immediate 16, save into register 11.
- Stl (set less than), set register 6 equal to 1 if register 1's contents is less than register 5's contents. If not, set it to 0.
- We then translate this to binary using the corresponding R, J or I-type instruction and then translate it to hex, which is what our memory expects, and put it on the top of our memory array.
- ffffffff instruction tells the CPU to be "done" and stop operating.

This assembly code tests every implemented instruction and the results can be seen in the waveform below, Figure 5.

4.2 Waveform to Test Assembly Code

We analyzed a waveform of the CPU running the assembly code, which verified that the CPU behaved as expected. The waveform is included in Figure 5.

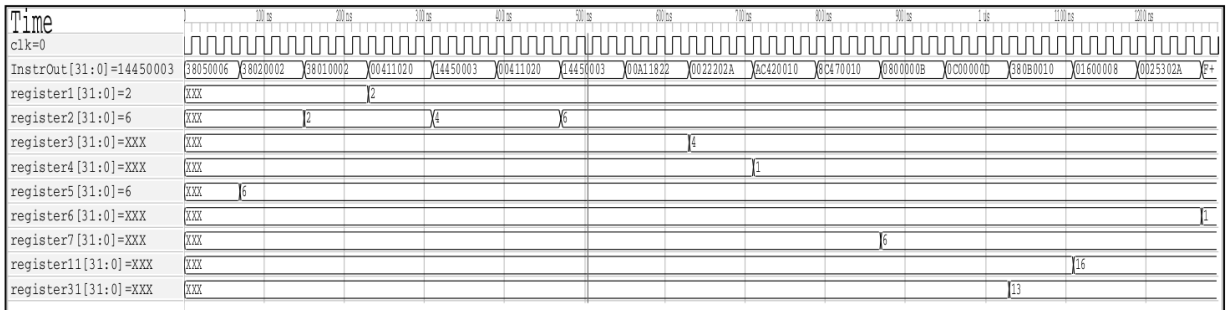

Figure 5: Waveform of CPU executing assembly test instructions.

We will explain the instructions in order, to show that they behave correctly. The hex values for the instructions are in the InstrOut signal. A full explanation of how the hex codes were generated is in the TestingIntructions.txt file. For each of these instructions, please see the waveform, which verifies that the signals behave as expected.

1. The first instruction of the program is the XORI instruction given by h'38050006. We expect this instruction to write the value 6 to register 5 (see above), and the waveform shows that register 5 correctly gets set to 6.
2. The next instruction in the waveform in h'38020002, which corresponds to the second XORI instruction. During this cycle, register 2 is set to the value 2, which is what we expect.

3. The third XORI instruction is given by h'38010002, and we observe that register 1 is correctly set to the value 2.

4. The next instruction is ADD, which is given by the h'0041102 instruction code. We add register 1 and 2 (which hold the values 2 and 2), and save the sum in register 2. In the waveform we can see that register 2 is set to 4 in this instruction, which is the correct sum of 2+2.

5. We then have a branch not equals instruction given by h'14450003. This checks whether register 2 and register 5 are equal. Since these registers hold the values of 4 and 6 respectively, they are not equal, so we branch to address 3, the add instruction.

6. Since we branched back to the add instruction, we see the instruction is h'0041102 again. We again add registers 1 and 2, which are 2 and 4 respectively. The sum is 6, which we store back into register 2. This is reflected in the waveform.

7. After add comes the BNE instruction again, given by h'14450003. This time, registers 2 and 5 both have the value 6. Since they are equal, we do not follow the branch address and instead continue on to the next instruction.

8. The next instruction after branch is h'00A11822, which is subtraction. We subtract register 1 from register 5, which is 6-2, and store the difference in register 3. We can confirm in the waveform that register 3 gets set to the value 4 in this step.

9. We then test SLT, with the instruction given by h'0022202A. Since register 1 is indeed less than register 2 (2 is less than 6), we set register 4 to 1. The waveform reflects what is expected.

10. Store word is the next instruction, which has the code h'AC420010. Here, we take the value from register 2 (which is 6), and store it at address 22 in memory. We confirmed this verifying that address 22 in the fullMem.dat file is set to 6. The fullMem.dat file acts as our memory.

11. We then use load word to read that word from memory, with the instruction h'8C470010. We read memory at address 22 (which holds the value 6, because we just wrote the that address in the previous step), and we set register 7 to that value. In the waveform, register 7 is correctly set to 6.

12. The next instruction is h'0800000A, which is a jump instruction. We want to jump to address 11. If we look at the memory file, we see that address 11 is h'0C00000D, so we expect this to be the next instruction.

13. The next instruction is in fact h'0C00000D, as expected, so we have verified that jump works as expected. This instruction controls jump and link. First of all, we want to set register 31 to the value of PC+8 according to the definition in MIPS. Because our implementation of program counter is set by address rather than bytes, we set register 31 to PC+2 instead (because 8 divided by 4 bytes is 2). Since the current PC address is 11, we want to set register 31 to the value 13, which is what we observe in the waveform. Secondly, we want to jump to address 13 in memory, which holds instruction h'380b0010.

14. In the waveform, the next instruction is h'380b0010 as expected, so we know that jump and link worked properly. This instruction uses XORI to set the value of register 11 to 16, and we can see in the waveform that register 11 is correctly set to 16.

15. The next instruction is h'01600008, which is a jump register instruction. This should jump to the address in register 11, which is 16. If we look at address 16 in memory, we see that we expect the next address to be h'0025302a. Since this is the next instruction in the waveform, we can verify that jump register works correctly.

16. The final instruction in the program is h'0025302a, which is a set less than instruction. Since register 1 is less than register 5 (2 is less than 6), we set register 6 to 1. This is all reflected in the waveform.

Since the instructions work as expected, as verified by the waveform, we have shown that our CPU is correctly able to compute these instructions. Therefore, we have a functioning CPU for all the instructions: LW, SW, J, JR, JAL, BNE, XORI, ADD, SUB, SLT.

5 Performance Analysis

5.1 Timing

The single cycle cpu has each instruction take 4 clock cycles time. This gives us a CPI (cycles per instruction) of 4. The four clock cycles are necessary because the ALU takes one whole clock cycle, and then the register file takes one more clock cycle after that to update the register output. However, we are leaving one clock cycle in between the two updates, so that there are no timing issues with the ALU and register file simultaneously updating. This brings us to 3 clock cycles. Since the PC updates with a slower clock that uses a multiple of 2 clock cycles, we round up to 4 clock cycles for each instruction. (The slower clock works by passing the system clock into a D-flip-flop with feedback, resulting in a signal that is high for two clock signals and low for two clock signals.)
Our CPU doesn't have the capacity, due to its design, to do any parallel processing. Because the CPU is written in behavioral verilog and not structural, the timing is not accurate to a real hardware implementation.

5.2 Cost

To calculate the cost of silicon area, we will assume that cost linearly scales with the number of gate inputs for basic inverting gates. Some of the modules were written in behavioral verilog instead of structural verilog, which makes it much more difficult to find the cost. The modules that were written in structural verilog are: Instruction Memory, Data Memory, Control Unit, Instruction Fetch Unit, and Sign Extend. In order to make cost approximations, we calculated the cost of these modules based only on the number of wires input to each module. For example, the Data Memory module takes two 32-bit inputs and one 1-bit input, for a cost of 2*32 + 1 = 65. We realize that this would underestimate the actual cost in terms of logic gates.

Table 2: Cost estimates for entire system based on costs of components

Subcomponent	Cost per	# Used	Total
ALU	3268	1	3268
Register File	1063	1	1063
Instruction Memory	32	1	32
Data Memory	32+32+1	1	65
32-bit mux	10*32=320	6	1920
4-bit mux	10*4=40	2	80
Control Unit	6+6=12	1	12
Program Counter (DFF+32-bit adder)	11+(25*32)=811	1	811
Instruction Fetch Unit	32	1	32
Sign Extend	16	1	16
32-bit add/subtract	25*32=800	4	3200
Slower clock (DFF)	11	1	11
Clock	2	1	2
			10512

Here we have a table to calculate the cost of the CPU, with the breakdown and calculations given table 2. Since the cost depends on the ALU and the Register file, we also have the calculations for those in tables 3 and 4 respectively.

The cost for each subcomponent of the ALU was calculated according to the exact gates used in the structural verilog module from Lab 1. Since all the gates are listed in the module, we have just included the final sum of the gate inputs in this table.

The cost estimates for the register file are based on the circuit diagram for a register file from homework 4. This implementation uses two 32-input MUXes, a 32 output decoder, and 32 registers.

Table 3: Cost estimates for ALU

Subcomponent	Cost per	# Used	Total
1-bit adder	25	32	800
1-bit subtract	30	32	960
NAND	2	32	64
NOR	2	32	64
AND	3	32	96
XOR (4 NAND gates)	4*2=8	32	256
OR	3	32	96
32-bit SLT (32-NOT + 32-bit ADD + OR)	32+(25*32)+3 = 835	1	835
Zero flag	(2+1)*31 +1=97	1	97
			3268

Table 4: Cost estimates for Register File

Subcomponent	Cost per	# Used	Total
32-input,32-bit mux (32*6-input AND + 32-input OR)	(32*(6+1))+(32+1)=257	2	514
register (treating this as the same cost as a DFF)	11	32	352
32-output decoder (32*5-input AND + 5 inverters)	(32*6)+5=197	1	197
			1063

6 Work plan reflection

Our work plan was made before we understood what implementing a CPU meant, and it shows. The verilog test benches should have been allocated for the modules like memory and alu instead of for each instruction. The work plan should have been to make all the modules, test all that we haven't made before, wire them up and then test the major CPU. To combat this, next time we should start making a plan of attack before we make a work plan.

Our original assembly test (factorial.asm), was written nicely and could have been a comprehensive test, but it included some instructions not able to be implemented on our CPU and was too complicated to use in debugging. Therefore, while we finished making the (factorial) assembly test by monday as planned, another, more simple, assembly had to be made for debugging. It was immediately translated to hex and is the first lines until 'ffffffff' in "fullMem.dat".

The report time estimate was mostly correct. Because we couldn't all work at the same time, it is hard to verify.

What we included again from last time was a couple hours set aside for General Confusion, which we agree still applies and is a realistic allocation of time. We think that General Confusion captures the time spent on unexpected problems, since we know we will encounter problems, but can't predict which parts of the project will be unexpectedly challenging at the beginning of the project.

Overall, we had hit some roadblocks to making our CPU that put us past the original deadline of Friday November 18. One of the team members was sick for a week with pneumonia, so we received an extension to be able to all work on it together. Also, in the last week, we scrapped the pipeline design we had and implemented a single cycle cpu in order to simplify our lives and not force our pipeline CPU to act as a single cycle CPU by inserting no-ops between instructions (to avoid hazards). Overall, this change was a good choice, as we were able to reuse most things, and had a much easier time wiring up the CPU and making the control logic unit. While the pipeline register was able to be compartmentalized for testing, the single cycle CPU was easy to "peek" into with the waveform analyzer, so we had to spend less time debugging than we would have with the pipeline CPU. However, the change did mean that we had to spend some extra time restarting instead of debugging the pipeline CPU, as our work plan expects. This means that the work plan is largely too low of an estimate, but it might have been accurate without these roadblocks.