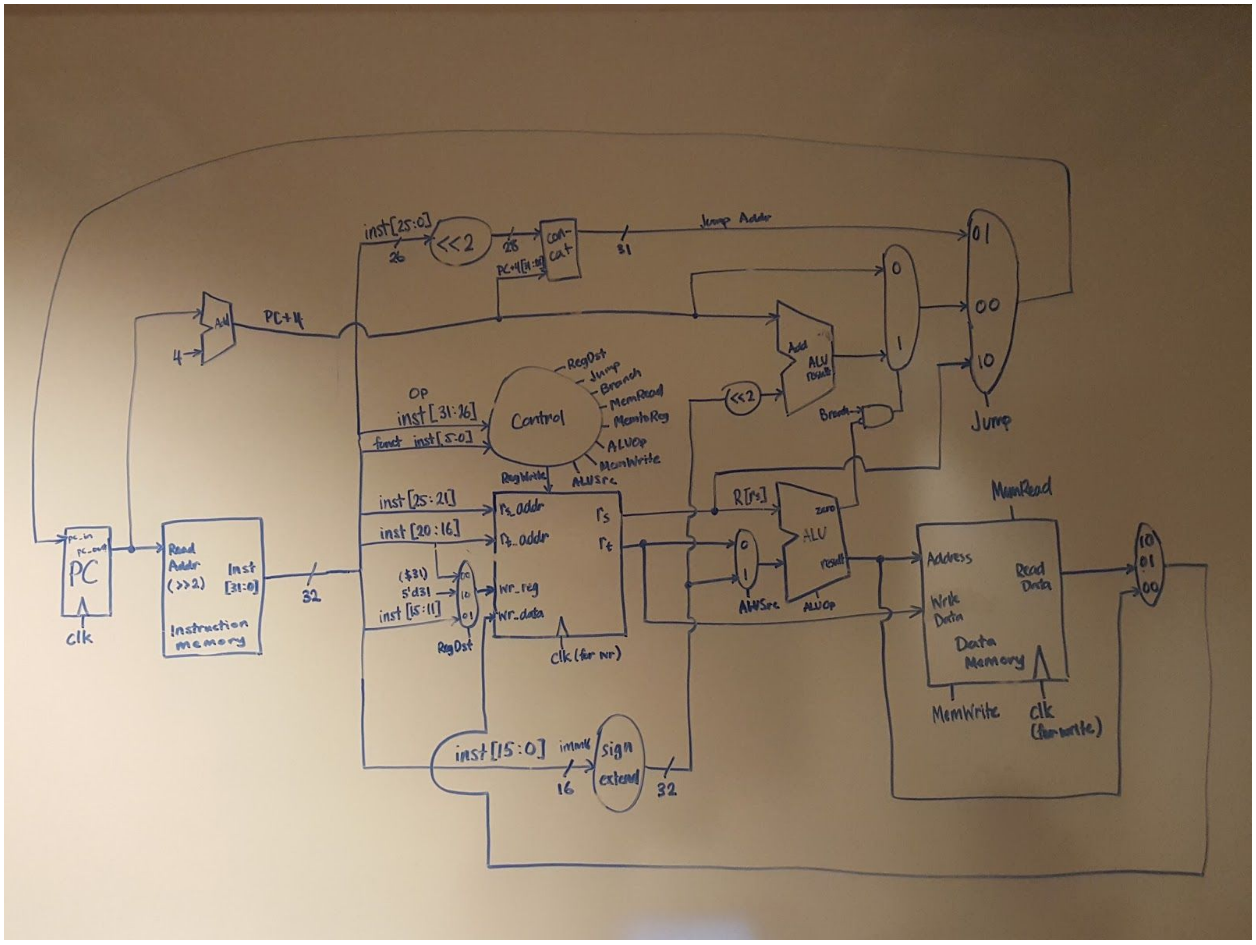
Computer Architecture
Lab 3: CPU

Anna Buchele, Jee Hyun Kim, Apurva Raman

Processor Architecture

CPU Block Diagram:



Above is the diagram for our design of a single-cycle CPU. It implements the following subset of the MIPS ISA:

LW, SW, J, JR, JAL, BNE, XORI, ADD, SUB, SLT

In each clock cycle, the PC is triggered on the positive edge of the clock and the corresponding address is accessed in the instruction memory.
The instruction at that address is split into parts, where from the op code and function, the control LUT determines the appropriate control signals.

Based on the instruction, the path through the CPU from this point on varies. Below is a description for the three types of instructions we implemented. The RTL we used was from the MIPS instruction reference; because we did not separate the instructions into stages as we would have had to for a multi-cycle or pipelined CPU, we did not break up the RTL into stages either.

R type (ADD SUB SLT JR):
Add, sub, and slt follow the same path through the CPU but have different control signals to the ALU.

ADD: R[rd] = R[rs] + R[rt]
SUB: R[rd] = R[rs] - R[rt]
SLT: R[rd] = (R[rs] < R[rt]) ? 1 : 0

The addresses for rs and rt are given by the instruction, and the values at those addresses are read from the register file continuously and passed to the ALU after rt is chosen by the mux with ALUSrc (which is 0 for these instructions) as the selector. Depending on the operation (ALUOp), the correct computation is performed by the ALU. The result is written back into R[rd] in the register file on the positive clock edge when its write enable is high.

JR: PC=R[rs]
The address for rs is given in the instruction, and the value is read from the register file and does not get written to the register file. (The ALU will produce some intermediate unused result.) Instead, the signal for the Jump mux will be 10 (instead of the 00 it was for the other R type instructions), which will write the value rs to the PC.

I type (XORI SW LW BNE):

The immediate is given in the instruction, and it is extended to 32 bits. The address for rs is also given in the instruction and the value is read from the register file. The ALUSrc flag should be 1.

XORI: R[rt] = R[rs] ^ SignExtImm(15:0)
The sign extended immediate is XORed with rs by the ALU. The result is then written back into R[rt] in the register file on the positive clock edge when its write enable is high.

LW: R[rt] = M[R[rs]+SignExtImm]
The sign extended immediate is added to rs by the ALU. The value is then read from data memory and written back into R[rt] in the register file on the positive edge of the clock when its write enable is high. Store word (SW: M[R[rs]+SignExtImm] = R[rt]) works similarly to LW, but reads from a register and writes to data memory.

BNE: if(R[rs]==R[rt])
PC=PC+4+BranchAddr
The addresses of rs and rt are given by the instruction, and they are subtracted by the ALU. The immediate is BranchAddr. If the zero flag is raised, it goes through an inverter and an AND gate with the Branch control signal, which becomes the selector for the Branch mux. The immediate gets left shifted and added to PC+4, which is the value chosen by the Branch mux. This value then goes back into PC.

J type (J JAL):

The PC jumps to the JumpAddr which is the concatenation of least significant 26 bits of instruction memory shifted left by 2 and most significant 4 bit of PC+4. The PC is updated to JumpAddr because the control code Jump has been set to 1.

J: PC=JumpAddr
(Goes through the basic J type path.)

JAL: R[31]=PC+4; PC=JumpAddr (We changed the MIPS instruction slightly to be PC+4 instead of PC+8.). PC+4 is written to R[31] in the register file. This allows the PC to resume the operation it was doing before the JAL once all the operations in the block has been completed.

Test Plan and Results

For testing, we tested individual modules and then tested our integrated cpu module with both a test bench and from the waveforms.

Data Memory test bench (datamem.t.v)
For testing the data memory, we have 3 test cases. The first test case checks the writing and reading function by writing the value 16 to register address 1 and reading value saved in address 1. The second test case is similar to the test case 1. It writes the value 99999999 to register address 2 and checks the address 2. The third test case checks if the value stored in register 1 is the same as the value set in the test case 1. All of the test cases passed.

Sign Extend test bench (signextend.t.v)
The sign extend was tested by giving in 4 different values and checking if the output matched what we expected to get. The 4 input values used for the test were 16'hffff (16'b1111…), 16'h7fff (16'b0111…), 16'h8000 (16'b1000…), 16'h4000 (16'b0100…) and all the test cases passed. These values were used to make sure the sign extend was extending correctly both negatively and positively, and that it was doing so correctly.

ALU test bench
The ALU was tested by running through each of the ALU operations using the corresponding control signals, and checking if the output was correct. The ALU test bench also tested the

control module in that the inputs were the “raw” instructions (both 6-bit control inputs) and the two values to be operated on. With this method, the control module had to give the correct signals and the ALU had to perform the correct operations in order to be correct.

Control test bench

For the control test bench, we checked all the control codes generated for all of the 10 different operations supported by our CPU. All the test cases passed.

Instruction Memory test bench

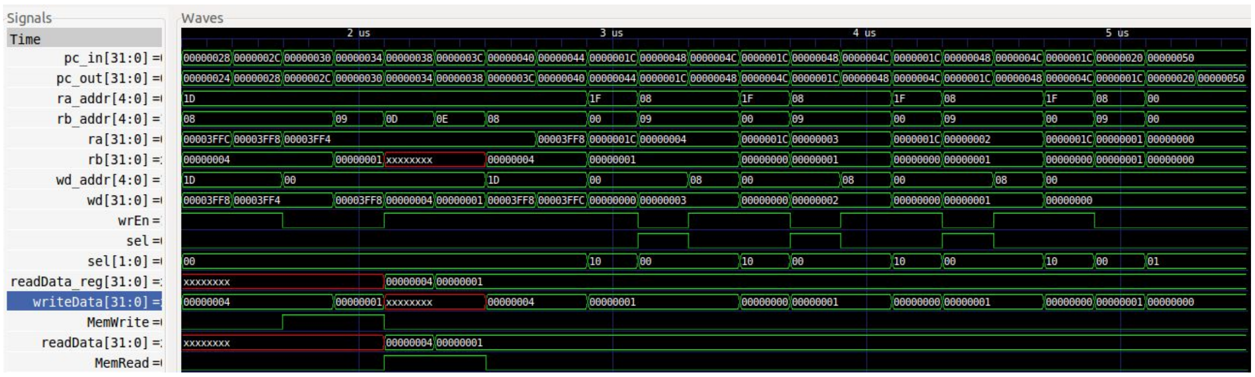
For the instruction memory, we checked that the first three instructions were being generated correctly. We assumed that if the instruction memory module was able to generate those three instructions correctly, that it would generate the others correctly as well.

CPU testing

For the full CPU testing, we tested all of the above individual modules, as well as the cpu itself. For the CPU test bench, for each instruction we selected one value which we deemed “important” for that instruction- that is, if the value is correct, the cpu can be assumed to have operated correctly. We tested this way because with 28 instructions in our cpu test, it seemed superfluous to test every output, especially since the outputs that are “important” depend on the command. The values we checked were the data memory read data, the memory write data, the PC addition ALU result, the read data from the registers, and the ALU result.

Waveform

We also tested the CPU using the hexadecimal data file generated from asm file. We first wrote a program in the asmtest folder to create a program which uses all the 10 supported MIPS instruction. Then, we exported as hexadecimal folder and ran the code with our CPU. We dumped all the variables and checked the variables in the gtkwave. We checked if the CPU was carrying out the instruction properly by evaluating the PC_out, register out, relevant control codes, rs, rt, rd and R[rs], R[rt], R[rd] of the register file, the input and output of data memory.



The figure above shows a snippet of the waveform generated. We checked all the relevant data and they were what we expected them to be.

Performance/Analysis of design

We analyzed our CPU design and approximated the area of the CPU using gate input equivalent approach.
Area total cost **1148480 GIE**

System Clock **2 GIE**
The system clock has an area of 2 GIE since it is composed of two inverters.

PC Module **416 GIE**
The PC module consist of 32 bit positive edge triggered d flip flops. 1 bit d flip flop is made up of 5 2NAND gates and 1 3NAND gates, which takes up 13 units of area.
The PC module has total area of 416.

Instruction Memory Module **3774 GIE**
The instruction memory array is sized at 32 by 21 for our implementation. The memory module will consist of one 32Decoder, one 32Mux and 672 tri state buffer. The 32Decoder will consist of 5 inverters and 32 5AND which cost 197. Assuming the 32Mux was built from 2Mux which has area cost of 7, the 32Mux will have area cost of 217 (from 31 (from 16+8+4+2+1) times 7). A tristate buffer is made up of a 2NAND, an inverter and a 2XOR with some transistors. Ignoring the transistors, we will assume that one tristate buffer has cost of 5 and that the tristate buffer in the register module cost 3360. Given the assumptions, the instruction memory will have total area cost of 3774.

Register Module **224249 GIE**
For our implementation, the register module has register array size of 32 by 1001, with 32032 tristate buffer, which cost 224224. There are also 2 32Mux for the R[rs] and R[rt] output which cost 434, and 3 32Decoder for reading the address for rs, rt and rd which cost 591. In total, the register module cost 224249. The cost can be greatly reduced by decreasing the array size.

Control Module **105 GIE**
The control module is a lookup table that stores 14 values. This means we need 15 (8+4+2+1) 2 input muxes, which has an area cost of 105 GIE.

Sign Extend Module **5 GIE**
The sign extender is wires along with one xor gate to determine whether the value of most significant bit of the immediate is 1 or 0. This has a GIE of 5.

Adders (x2) **480*2 GIE**

A one bit full adder has an area cost of 15, so chaining 32 adders together gives us a cost of 480.

ALU Module **988 GIE**
The ALU consists of a LUT (3 2-input MUX), 32 bit adder-subtractor (32 bit adder + 32 XOR), a zero checker (32 XOR gates), a mux, and a 32 bit XOR. This gives it a GIE cost of 988(21+ (480+160) + 160+ 7+ 160).

Data Memory Module **917925 GIE**
For our implementation, the data memory module has register array size of 32 by 4096 (16383-12288+1 from h3fff - h3000 + 1). It will have 131072 tristate buffer which cost 917504.
There is one 32Decoder for the address input, and one 32Mux for the ReadData output, which cost 414. It also has one positive edge triggered d flip flop for the memory write control which cost 7. In total, the data memory cost 917925.

Additional Modules in CPU
In the CPU, we also implemented three 3Mux and two 2Mux, which cost 14 and 7 respectively, costing 56GIE. The CPU has total area cost of 1148480 GIE. Of the total area cost, 1145088GIE, which accounts for 99% of the total area, comes from the tri state buffer used for memory. Reducing the memory size, by using only the memory we need, could greatly improve the area cost.

Performance:

Since we wrote the CPU in behavioral verilog, the timing for each element is dependant on how the synthesizer organizes the architecture. Because we chose the single-cycle CPU, each clock cycle must be at least as long as the time for the longest instruction to execute. This means that for many instructions, there is wasted time at the end of the clock cycle. Multi-cycle designs can make individual instructions faster, and pipelined CPUs can execute multiple different stages of instructions simultaneously, which make the amount of time to run programs on the CPU faster.

Work Plan Reflection

We have allocated a lot of time in debugging because we have encountered many problems while debugging. We took a little longer than our estimated time to write the modules and test benches. However, we did not spend too much time debugging to make the CPU work. We think it helped a lot to have scheduled meetings with ninjas and our instructor. The sessions allowed us to understand the CPU better and cut down the time spent on making the CPU work.