Lab 6

Pre-Knowledge

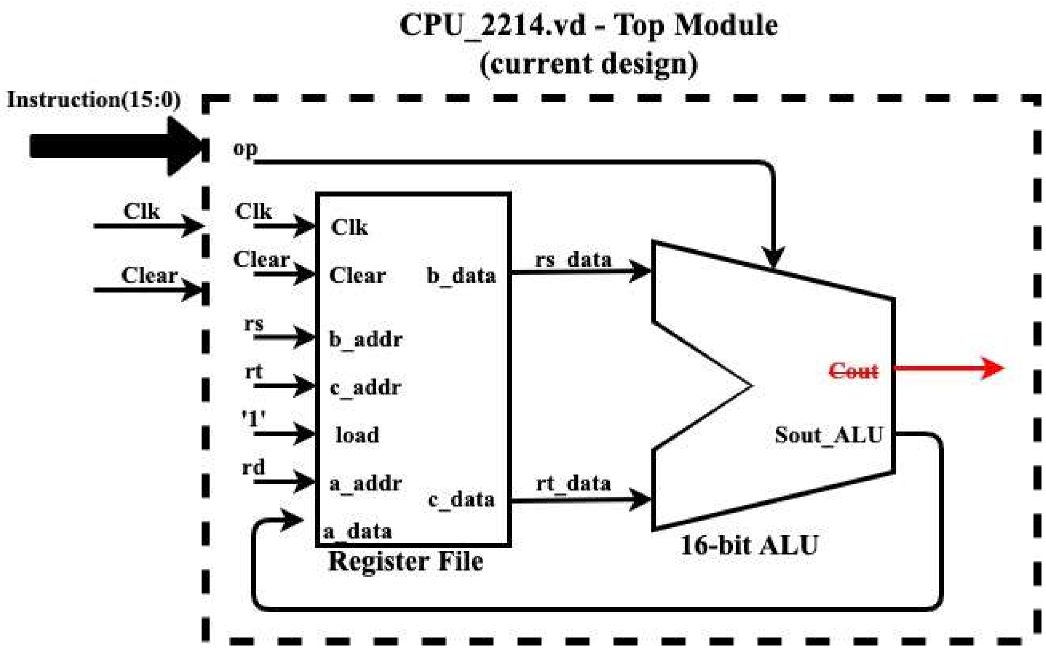
In order to complete this lab, you will need to understand the function of the Sign Extend block and the CPU control block, especially how it interfaces with each of the elements in the circuit.

Objective

In this lab the students will build off their existing Register File and ALU blocks and implement the Sign Extend and Control blocks. **<u>This lab will not include a testbench or require a lab report.</u>**

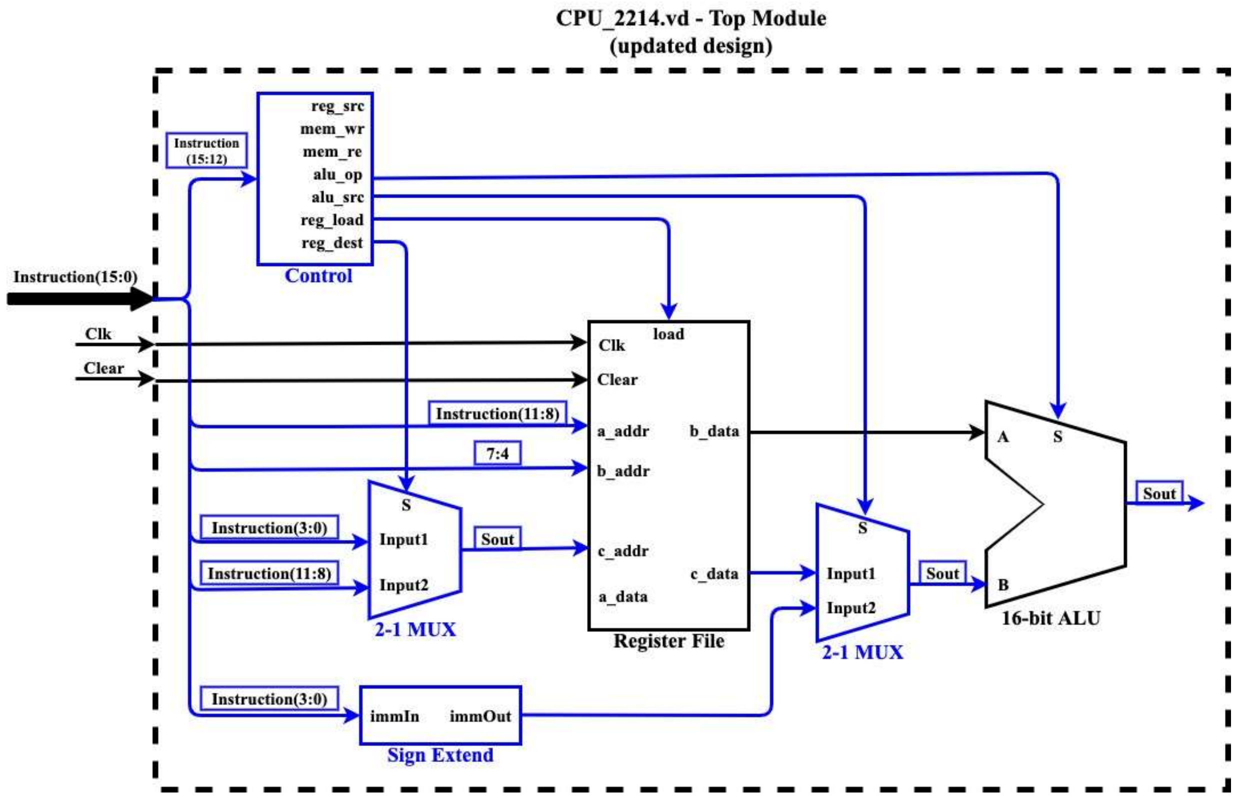
Overview

In the previous lab, we have assembled a CPU up to the following level of functionality below:



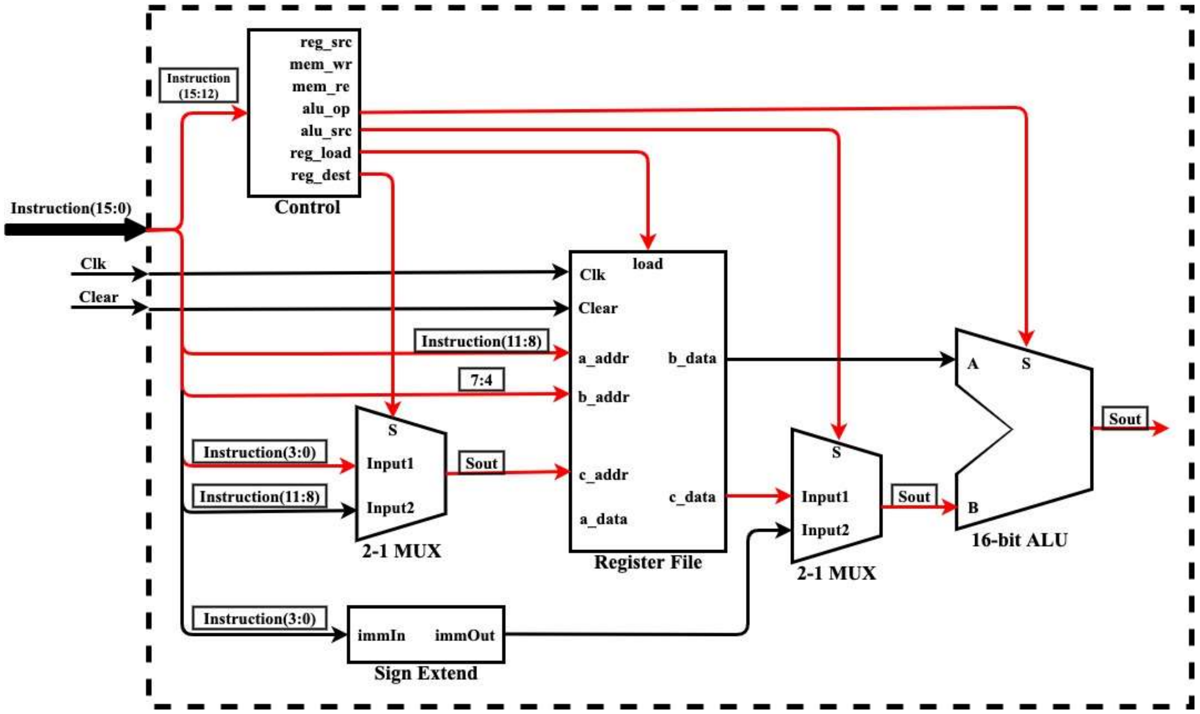
This design allows for a user to do simple additions, subtractions, AND's, and OR's. In this Lab we wish to add more functionality to the CPU by implementing the sign extend and control blocks. We will do this by adding a completed control block, a couple of multiplexers, and the sign extension component. The completed code for the following blocks will be provided you: ALU_16Bit.vhd, ALU.vhd, and_gate.vhd, full_adder.vhd, mux2_1.vhd, mux3_1.vhd, MUX21_4Bit.vhd, MUX21_16Bit.vhd, MUX31.vhd, or_gate.vhd, and Registers.vhd. Your job will to be to finish the partially completed code for the following files: Signextend.vhd, Control.vhd, and CPU_2214.vhd.

This lab will build up to lab 7, the final lab, which will implement the data memory block and the program counter. The design that will be created upon completion of this lab will have some driver signals that are not connected to anything because they require the data memory component which has yet to be implemented. The following modifications allow us to execute simple R-Type & I-Type instructions. A visual representation of what our new circuit will look like is shown below with the blue lines representing the new circuits:

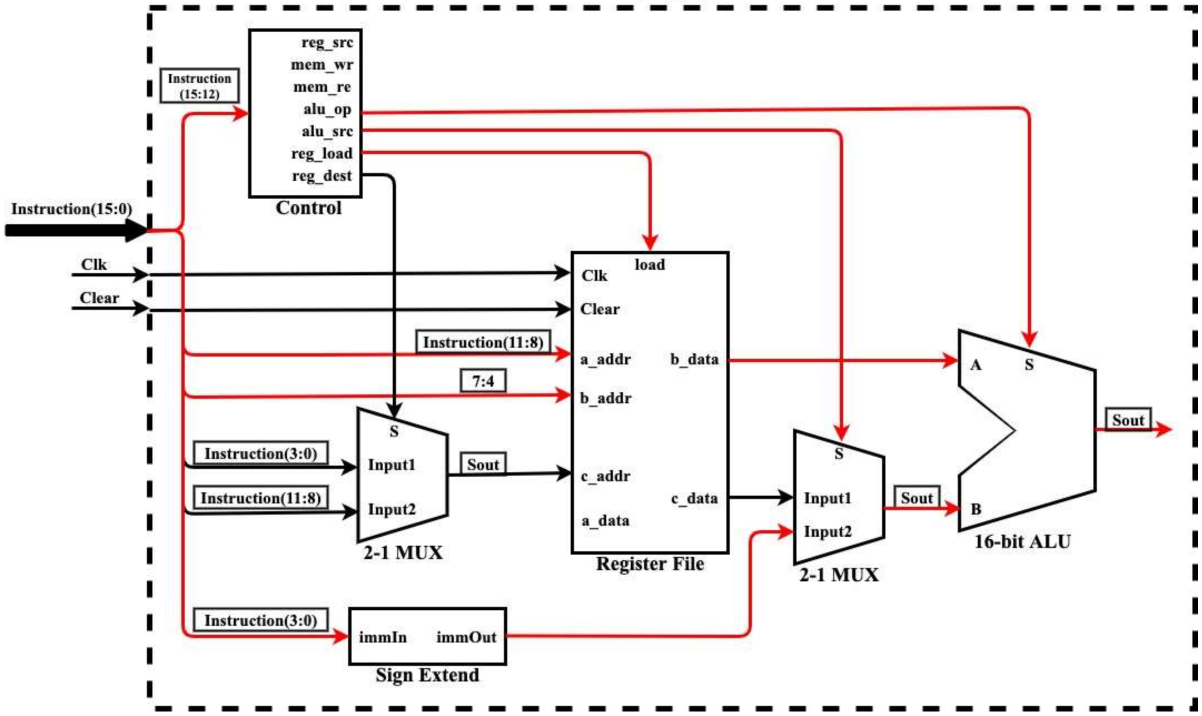


The next two images below show the execution path for R-Type instructions and I-Type instructions. If any of the figures are hard to read, they have been included in your downloaded files.

CPU_2214.vhd - Top Module (R-Type)



CPU_2214.vhd - Top Module (I-Type)



Lab Execution

Control Unit: The opcode value from the previous lab will be changed from 2-bits to 4-bits. **Please Note the opcode coming into "Control" (the entity), is 4 bits long, it decodes the 4-bit number and then sends a 2-bit number (*alu_op*) to the ALU ($2^2 = 4$ ALU operations□ ADD, SUB, AND, OR).**

As an example, if the opcode = 0100, the operation to execute is the immediate addition (ADDI) with the value of the second operand coming from the sign extend unit. Therefore, the signal *ctrl_alu_op* should be equal to "00" for addition in the ALU and *ctrl_alu_src* be set to **1** so that the mux will select the incoming signal from the sign extend unit instead of the register file.

The following table is provided to show each opcode and the corresponding instruction:

Opcode (Binary)	Operation
0000	ADD
0001	SUB
0010	AND
0011	OR
0100	ADDI
0101	SUBI
1000	LW
1100	SW
0111	SLT

You should design the remainder of the sign extend and control blocks. Implement the two MUX's that are shown in the figures above. Add to the CPU_2214.vhd file to implement the port maps of the two MUX's, the control block, and the sign extend block. The provided VHDL files are labeled with TODOs in the comments for you to follow. Use the signals that are already defined for you as assistance.

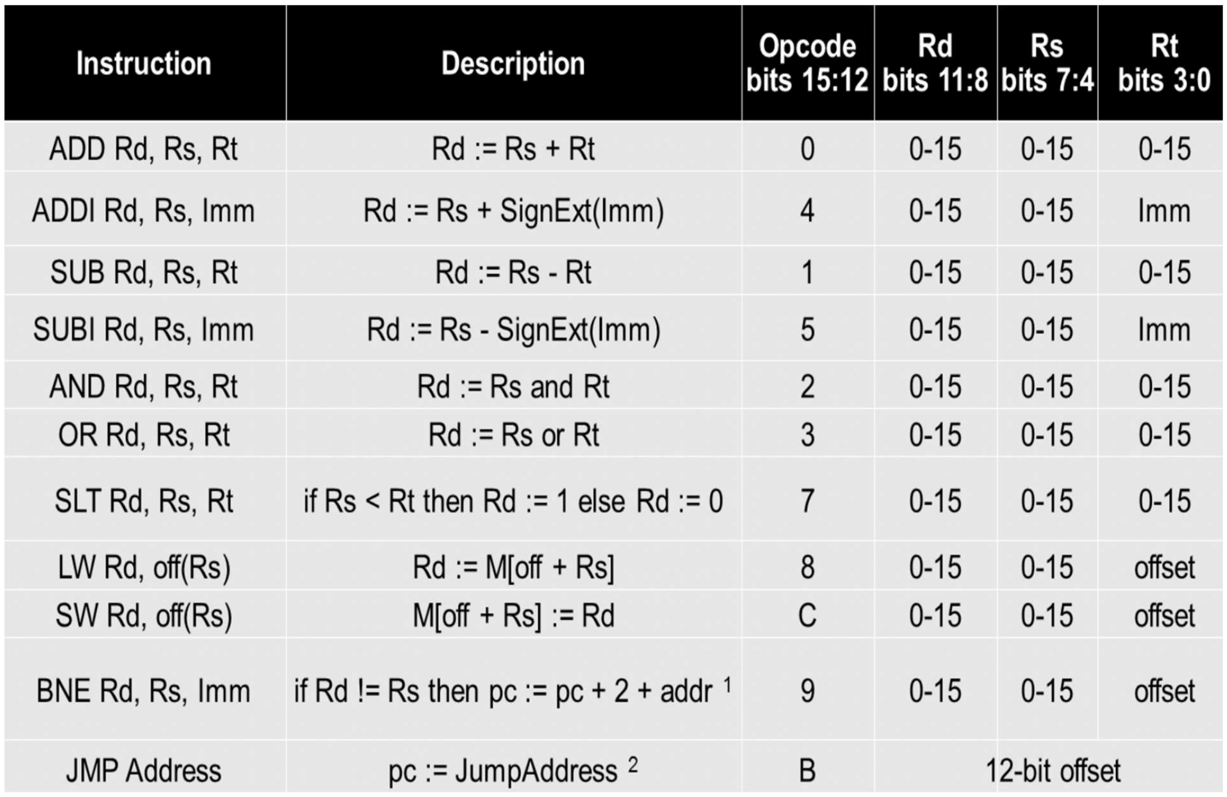
- reg_src - [currently unused] This control signal decides whether the value driving the a_data input is coming from Data Memory, an ALU operation, or the SLT operation.
- mem_re - [currently unused] This signal enables or disables memory reading.

- mem_wr - [currently unused] This signal enables or disables memory writing.
- alu_op - the 4-bit opcode that connects to the ALU to determine which instruction is selected
- alu_src - connects to the MUX to select which input will go into the B input of the ALU, either the sign-extend or the register file.
- reg_load - connects to the register file to initiate a load of the inputted data. This signal enables writing to the register.
- reg_dest - connects to the MUX to select which input address will go into the c_addr input of the register file. This signal decides which portion of the instruction is applied to c_addr.
    - It is set to 1 for the SW instruction
    - It is set to 0 for the R-type instructions and the SLT instruction
    - It is a don't care for the I-type instructions

Sign Extend: For our immediate instructions to work we need to design a sign extend module. This module will take a 4-bit number and extend the number to be 16 bits. If the 4-bit number ends in a 1 then 1's is extended (to keep the sign negative). If the 4-bit number ends in a 0, then 0's is extended (to keep the sign positive). An example of a sign extend is shown below.

Mux Units: this is simply adding two 2-to-1 multiplexers. This has been provided to you and there is no need to modify it. Be aware of the defined signal names and how to connect them to the ALU, register file, etc.

NOTE: The images do not show it, but for the time being the "Sout" signal from the ALU can be mapped directly to the "a_data" input on the register file.



Instruction	Description	Opcode bits 15:12	Rd bits 11:8	Rs bits 7:4	Rt bits 3:0
ADD Rd, Rs, Rt	Rd := Rs + Rt	0	0-15	0-15	0-15
ADDI Rd, Rs, Imm	Rd := Rs + SignExt(Imm)	4	0-15	0-15	Imm
SUB Rd, Rs, Rt	Rd := Rs - Rt	1	0-15	0-15	0-15
SUBI Rd, Rs, Imm	Rd := Rs - SignExt(Imm)	5	0-15	0-15	Imm
AND Rd, Rs, Rt	Rd := Rs and Rt	2	0-15	0-15	0-15
OR Rd, Rs, Rt	Rd := Rs or Rt	3	0-15	0-15	0-15
SLT Rd, Rs, Rt	if Rs < Rt then Rd := 1 else Rd := 0	7	0-15	0-15	0-15
LW Rd, off(Rs)	Rd := M[off + Rs]	8	0-15	0-15	offset
SW Rd, off(Rs)	M[off + Rs] := Rd	C	0-15	0-15	offset
BNE Rd, Rs, Imm	if Rd != Rs then pc := pc + 2 + addr [1]	9	0-15	0-15	offset
JMP Address	pc := JumpAddress [2]	B	12-bit offset		