Computer Organization Project: Write Up

Members: Abdul Razzak (razzaa2), Bjourn Etienne (etienb), Erwin Hitgano (hitgae), and Michael Newby (newbym4)

Get_Instructions

For Get_Instructions, we based our solution off of a function from Lab 6. We then extended that function to accommodate for all the other instructions used in this project. First, we have a helper function called set_register, this function takes in a register and gives the corresponding binary representation of it. We then move onto Get_Instructions. For this function we read in the instructions and parse each word. We parse the line into the instruction, op1, op2, and op3. We check the instruction (hardcoded) and go into the corresponding if branch. Then within the if branch we set the correct registers, opcode, and address and also the correct funct and shamt if necessary. Finally we store the binary representation of the whole instruction into our 32x32 instruction array, while also returning the number of instructions.

Instruction_Memory

For Instruction_Memory we use the 5 least significant bits of the Read Address as our inputs to a 5-32 decoder. The output of the decoder then tells us what index of our MEM_Instructions to pull from in order to set the instruction accordingly. We do this by using a 2 multiplexor using the output from the 5-32 decoder as the selector bit, the instruction array itself and the MEM_Instruction array as the inputs. This will either set instruction to whatever is in the correct index in MEM_Instructions, or not change instruction at all (effectively acting as an if statement).

Control

Input Or Output	Signal Name	R-format	lw	sw	beq	JAL	Jump	Addi	Jr
I	op5	0	1	1	0	0	0	0	0
I	op4	0	0	0	0	0	0	0	1
I	op3	0	0	1	0	0	0	1	0
I	op2	0	0	0	1	0	0	0	0
I	op1	0	1	1	0	1	1	0	0
I	op0	0	1	1	0	1	0	0	0
O	RegDst[1]	0	0	X	X	1	X	0	X
O	RegDst[0]	1	0	X	X	0	X	0	X
O	ALUSrc	0	1	1	0	X	X	1	X

O	MemtoReg	0	1	X	X	X	X	0	0
O	RegWrite	1	1	0	0	1	0	1	1
O	MemRead	0	1	0	0	0	0	0	0
O	MemWrite	0	0	1	0	0	0	0	X
O	Branch	0	0	0	1	X	X	0	X
O	ALUOp1	1	0	0	0	0	X	0	X
O	ALUOp0	0	0	0	1	0	X	0	X
O	Jump[1]	0	0	0	0	0	0	0	1
O	Jump[0]	0	0	0	0	1	1	0	0

For Control we just used a 64-mux with op code as selector bits to determine the settings of the control lines. Although the op code for all the R-format instructions is same, we used a different op code for jr instruction only because the control lines for jr are different than the other R-format instructions.

Read_Register

For Read_Register we use the 5 least significant bits of ReadRegister1 as our input to a 5-32 decoder. We then do the same process we did in instruction memory, using a 2 multiplexor to pull the correct instruction and set ReadData1 correctly. We repeated this process for ReadRegister2 and ReadData2.

Write_Register

For Write_Register we use the 5-bit address WriteRegister as our input for a 5-32 decoder, and the control bit RegWrite as our enabler for the decoder. The output of the decoder then tells us the correct index for MEM_Register. We use a double for loop and a 2 multiplexor, with the output from the decoder as the selector bit, and MEM_Register and Write_Data as our inputs. The multiplexor correctly decides when to write to MEM_Register (acting as an if statement).

ALU_Control

For ALU_control we calculated the SOP for each instruction, add, sub, or, and slt (we did not need to implement it for and, because in our UpdateState, we initialized the ALUControl bits to 0, 0, 0, 0 and then set them to the output of the ALUControl function (this is essentially having the default ALUControl bit as AND). Once the SOP was calculated for each bit, we set the corresponding bit to the correct or_gate. The 0th bit does not matter, as it is always zero for all the instructions we process.

ALU

For ALU we calculated all the results (add, sub, slt, and, or) and passed these results to an 8-mux with ALUOp as the selector bits to store the desired output in the Result variable. For the Zero flag we

created a 32 or_gate to or the 32 bits of the result from the ALU with zero. The result of the or is then sent to a not gate and then stored in the Zero flag.

Data_Memory

For Data_Memory, we first used a decoder 5-32 to calculate all 32 bits of the output. We then used a for loop to use a multiplexor2_32 to correctly write to MEM_data and read from MEM_data. For writing, the selector bit was each bit of the output from the decoder, and it determined whether to leave MEM_data the same, or write our data to it. For reading, it was the exact same, however the selector bit determined what the correct index of MEM_Data to read from.

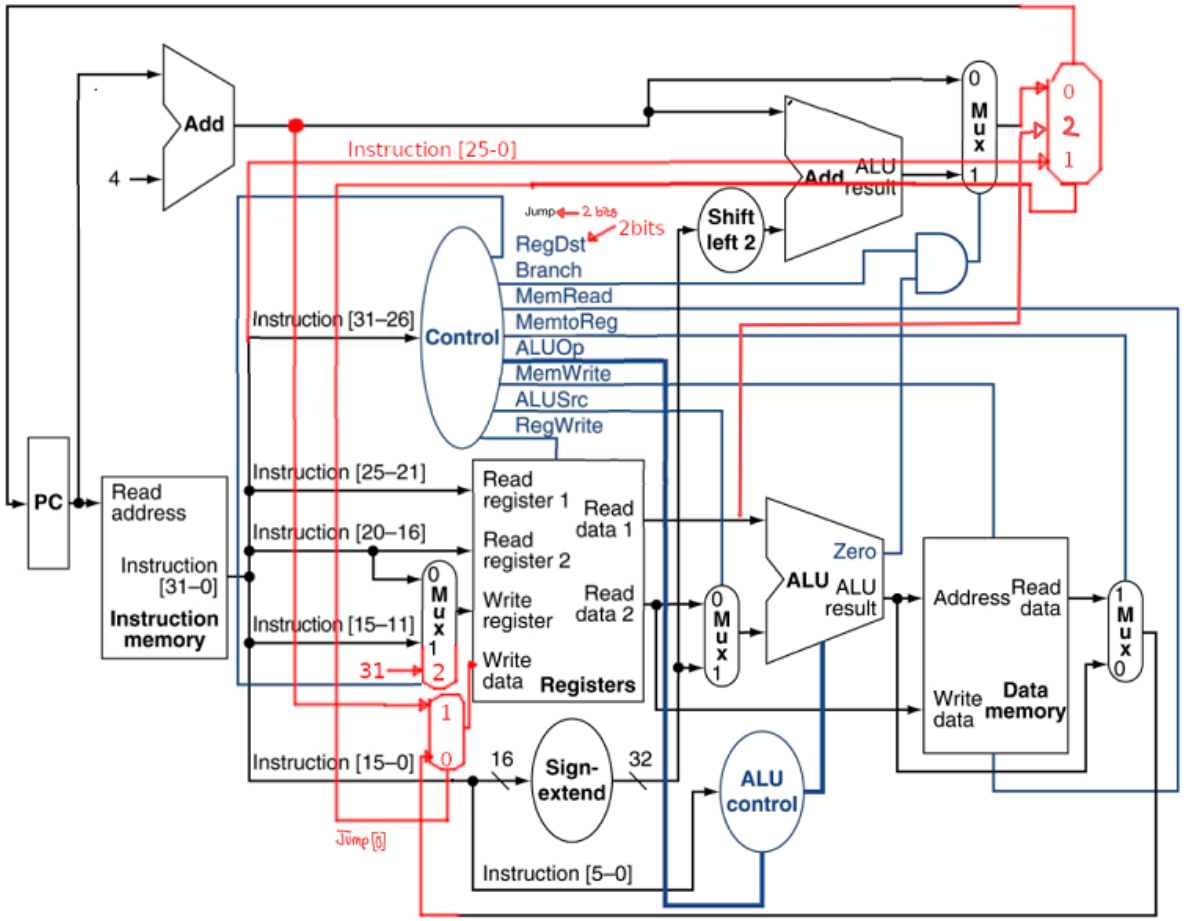
Extend_Sign16

For Extend_Sign we took the leading bit of the 16 bit binary, made that first bit the leading 16 bits in a new 32 bit binary variable then copied the entire 16 bit binary into the last 16 bits of the 32 bit binary variable.

UpdateState

For UpdateState we pulled up the image for the datapath and followed it, but also included jump paths for jr and jal instructions. Both Write_Dest, and Jump instruction are two bits. We first called control to set most of the registers to what they needed to be. Then we set the ReadReg1 and ReadReg2 bits equal to their perspective Instruction bits. We then called the read register, stored the outputs in new arrays, then set WriteRegister2 to the necessary instruction bits. We then extend instruction 25-0 and 15-0 to 32 bits then use the extended instructions as an input for a 2-mux. We then set funct to another portion of the instruction bits, then used the ALU to complete the operation indicated by ALU_Control. We take that result, then use DataMemory to write the data back to the registers. Finally, we increment by 1, and set the final result.

Updated DataPath to Support JAL and Jr



Contributions:

Abdul: Control, Write_Register, ALU_Control, ALU, Data_Memory, UpdateState,Write Up, Debugging

Bjourn: Get_Instructions, Instruction_Memory, Control, Read_Register, Write_Register, ALU_Control, ALU, Data_Memory,Write Up, Debugging

Erwin: Get_Instructions, Instruction_Memory, Control, Write_Register, ALU_Control, ALU, Data_Memory,Write Up, Debugging

Michael: Get_Instructions, Instruction_Memory, Read_Register, Write_Register, Extend_Sign16, UpdateState,Write Up, Debugging

Citations:

For ALUOps and funct fields and what the ALUControl output should be
https://cs.wellesley.edu/~cs240/f14/lectures/18-control.pdf

Lab 6 Solution

HW 5 Solution

HW 5 Solution from Abdul