Project Writeup

Functions

Instruction Parsing

void intconvert_to_binary(int a, BIT *A)

As the function name suggests this takes in an int a and a bit pointer A then it converts the int into binary. To do that it first checks if its negative in which then it will have 1 added to it then flipped to its positive counterpart. Afterwards it keeps dividing itself by two and adding one to the bit position if it has any remainders. If the original int is negative the bits are then inverted.

void operators(char *op, BIT *instruction)

This function just checks if the operator that is passed in is the same as specific instructions like lw or sw that is hardcoded. Then it modifies the instruction Bit array that is passed in. This accounts for the opcode and funct depending on the operator given.

void register_address(char *reg, BIT *address, int begin, BIT *instruction)

This function takes in the reg which is the register (t0 t1 etc) Then it compares with all the possible registers. It also takes in where the register should be placed in the final instructions BIT that is where the int begin is. The address is just a storage bit array which will store the binary version of the registers. After comparing and finding the binary equivalent of the register, it will then put that binary into the instructions bit array.

int get_instructions(BIT Instructions[][32])

This function uses all the previous functions by first figuring out the type of operation it is (I J and R type) through the operator given. For I types it just converts the registers into binary then puts them into the output array then it takes the integer at the last place and converts that into binary, for J types it does the same except it is 3 registers. For R types it just takes in the int that it contains. After that is done it puts the output into the instructions array and adds one to the instruction count.

ALU and associated functions and Memory

void ALU32(BIT* A, BIT* B, BIT Binvert, BIT CarryIn, BIT Op0, BIT Op1, BIT* Zero, BIT* Result, BIT* CarryOut)

The Zero bit is implemented to indicate whether the result of the operation is zero. The function initializes a temporary variable Z to FALSE, and then performs a bitwise OR operation between all the bits of the Result array except for the most significant bit. The resulting value is inverted using the not_gate function, and the final value of Z is stored in the output operand Zero.

void ALU(BIT *ALUControl, BIT *Input1, BIT *Input2, BIT* Zero, BIT *Result)

This function simply calls ALU32 setting CarryOut to FALSE.

void Instruction_Memory(BIT *ReadAddress, BIT *Instruction)

This function implements the instruction memory of the processor. It takes in a 32-bit instruction address "ReadAddress" and returns a 32-bit binary instruction "Instruction". The function uses a 32x32 array "MEM_Instruction" to store the instructions.

The function uses the “decoder5” function which sets the temporary array "I" element corresponding to the binary value of ReadAddress, which sets the select line for the multiplexor that will read the instruction from memory. The function then iterates over the 32 bits of the Instruction array. For each bit, it calls the function "multiplexor2_32" with the corresponding element from "I", "MEM_Instruction" and the current Instruction bit. This selects either the current Instruction bit or the corresponding bit from "MEM_Instruction" based on the value of the select line "I".

void Data_Memory(BIT MemWrite, BIT MemRead, BIT *Address, BIT *WriteData, BIT *ReadData)

This function implements the data memory of the processor. The function has many parameters such as “MemWrite” and “MemRead” flags that control memory access, a 32-bit “Address” for memory retrieval, the “WriteData” used to store information in memory, and a “ReadData” output that retrieves data from memory.

The function uses the “decoder5” function which sets the temporary array "I" element corresponding to the binary value of “Address”, which sets the select line for the multiplexor that will read or write the instruction from memory. The function then iterates over the 32 bits of the data array. For each bit, it calls the function "multiplexor2_32" twice to either overwrite or maintain the values of “ReadData” and “MEM_Data[i]”. For each bit, the function uses “multiplexor2_32” to select between the current memory content “MEM_Data[i]” and the “WriteData", based on the “MemWrite” control flag and the current address bit “I[i]”. If “MemWrite” is TRUE and “I[i]” is TRUE, the “WriteData” is selected to write into the memory. Otherwise, the current memory content “MEM_Data[i]” is selected. This same method is applied for reading the memory into “ReadData”.

void Write_Register(BIT RegWrite, BIT *WriteRegister, BIT *WriteData)

This function is also similar to those above. This function writes the data provided in the WriteData input into a register specified by the “WriteRegister” input, based on the “RegWrite” control bit. First, it decodes the WriteRegister input into a 32-bit binary value using the “decoder5” function. Then, uses “multiplexor2_32” to write the data into the specified register. However, the register value is only updated if the RegWrite control bit is TRUE.

void Read_Register(BIT *ReadRegister1, BIT *ReadRegister2, BIT *ReadData1, BIT *ReadData2)

This function is very similar to the function “Instruction_Memory”. It takes in two 32-bit instruction addresses "ReadRegister1" and “ReadRegister2” which returns two 32-bit binary instructions "ReadData1" and “ReadData2”. The function uses a 32x32 array "MEM_Register" for the registers.

The function uses the “decoder5” function which sets the temporary array "l" and “r” element corresponding to the binary value of "ReadRegister1" and “ReadRegister2”, which sets the select line for the multiplexor that will read the instruction from memory. The function then iterates over each bit, calling the function "multiplexor2_32" twice with the corresponding element from "l" or “r”, "MEM_Register" and the current register (which is different for each “l” or “r” register). This selects either the current register or the corresponding bit from "MEM_Register" based on the value of the select line "l" or “r” accordingly.

<u>Extend_Sign16(BIT *Input, BIT *Output)</u>

The Extend_Sign16 function takes a 16-bit Input and extends it to a 32-bit Output, sign-extending the 16th bit. It works by first copying the Input to Output, then setting up two temporary arrays of 32 bits each. One array is set to all TRUE values, and the other is set to the Input array. The function then uses a multiplexer to select between the two temporary arrays based on the value of the 16th bit of the Input. If the 16th bit is 0, the temporary array that contains the Input is selected, otherwise, the temporary array that contains all TRUE values is selected, effectively sign-extending the 16th bit to the remaining 17-32 bits of the Output. Finally, the function writes the resulting value to the Output array.

Control and ALU Control

<u>void Control(BIT *OpCode, BIT *funct,
BIT *RegDst, BIT *Jump, BIT * JAL, BIT* JR, BIT *Branch, BIT *MemRead,
BIT *MemToReg, BIT *ALUOp, BIT *MemWrite, BIT *ALUSrc, BIT *RegWrite)</u>

This function set the control bits depending on the six bits that came in the OpCode and the funct. The funct was added as an input to determine the added JR control bit. This function had a lot of code but it was relatively simple since it followed the table in the project pdf with Jump, JAL, and addi included. Each control bit was essentially determined based on which instructions’ OpCode set it to TRUE vs set it to FALSE. We or_gated the OpCodes that determined if the bit was TRUE, or FALSE in the case of RegWrite because we wanted to shorten the code.

void ALU_Control(BIT *ALUOp, BIT *funct, BIT *ALUControl)

The ALU control takes in the ALUOp determined by Control and the funct of the instruction. It determines the function of the ALU (add, sub, slt, etc.). We did what we did in Control and just followed the table given in the project since the jump instructions didn't rely on the ALU while addi followed the same format as lw/sw.

Update State

void AssignInstructions(BIT* instr, BIT* op, BIT* reg1, BIT* reg2, BIT* write, BIT* ad, BIT* funct, BIT* jumpad)

This was a helper function for updateState() in the Decode portion. It separated the instruction into its OpCode, registers, addresses, or constants.

void updateState()

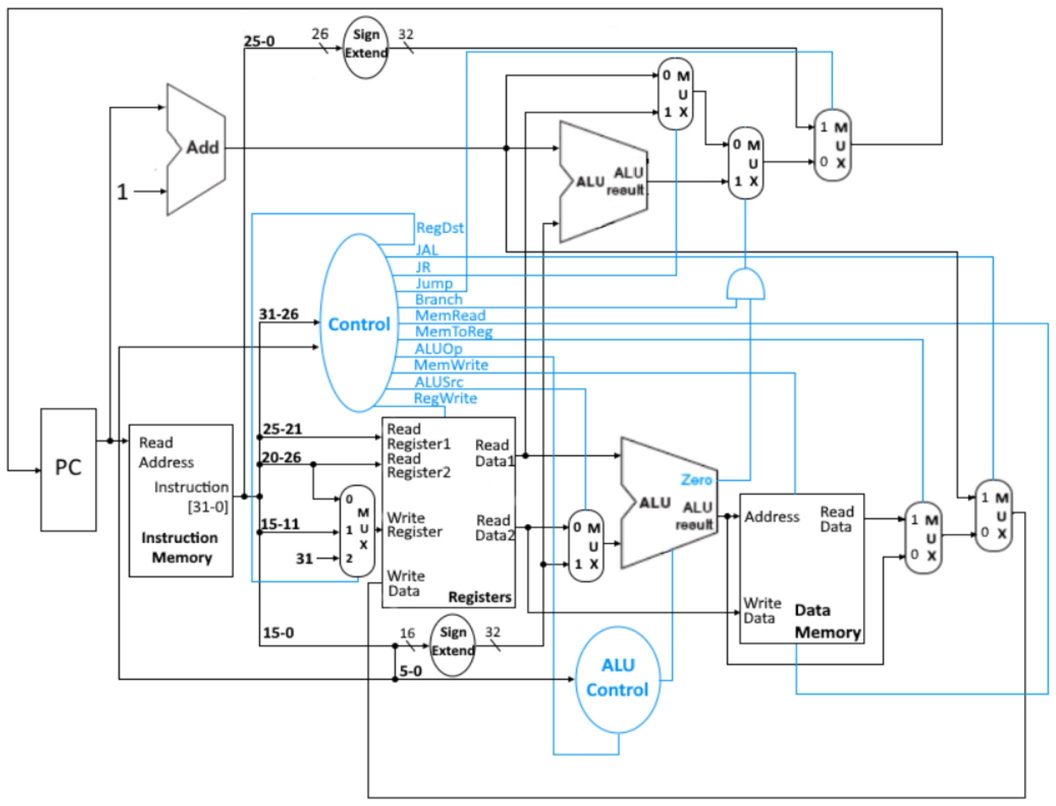
This function essentially follows the datapath given on the project instructions but also accounts for the modifications for addi, jal, and jr which will be discussed on the following page. In the Fetch stage, we call InstructionMemory to fetch the current instruction from memory. In the Decode stage, we split the instruction into its respective parts (opcode, registers, addresses, etc.), set the control bits, and read from register memory. In the Execute stage, we process any arithmetic depending on the instruction. In the Memory stage, we use the processed arithmetic to read data from memory or just store a value. In the Write Back stage, we determine if we want to write read data into the register files. Then we just update the PC, whose value depends on the type of instruction.

Modifications to Datapath for addi, jr, jal

The only modifications for addi were its Control bit settings. There was no need to add anything to the ALU_Control regarding addi because its an I-type instruction and its ALUOp was the same as lw and sw.

The purpose of jr is to jump to the read register so we created a control bit that was set true if the instruction was a jr. Since its opcode is the same as R-type instructions (000000), we had Control take in the Funct (Instruction[5-0]) to distinguish between it and other R-types. Since it jumps the PC to the read register, we had a mux that takes in JR as the selector, PC+1 as input 0, and the read register as input 1.

For jal, we needed to store the next PC instruction to the 31 register and also jump to the instruction specified. To do the storing, we used a 4-mux instead of a 2-mux to determine the write register. This also means we had to have a 2-bit RegDst to represent the 3 inputs—Read Reg2, Instruction[15-11], 31—which we determined in Control. We also needed another mux to determine the data to write back and takes in JAL as the selector, the next instruction (PC+1) as input1, and the original Write-back data as input0. Jumping to the instruction specified was done as a regular jump instruction.



<u>Contributions</u>

Charlie Chen - Implemented the parsing functions, helped with debugging and contributed to the writeup.

William Shin - implemented functions under section “ALU and associated functions and Memory”, helped with debugging, contributed to the writeup

Joseph Park - implemented Control and ALU_Control, contributed to writeup

Christopher Poon - implemented the UpdateState portion of the project, debugged other functions, modified the datapath, contributed to the writeup