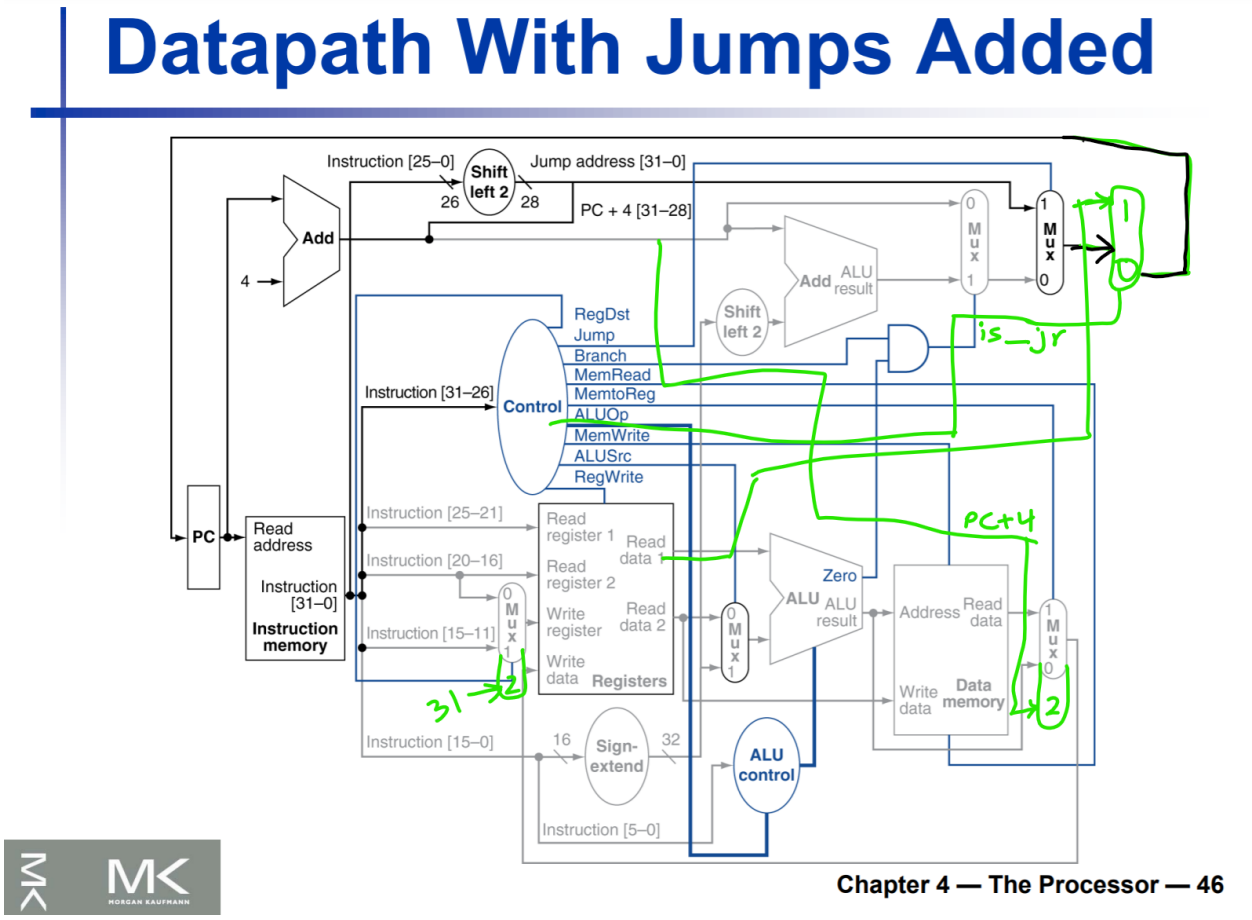
CSCI 2500 - Group Project
MIPS Processor in C

December 10, 2021

The Char Stars
Julian Matthews (matthj8) - matthj8@rpi.edu
Nadia Choudhury (choudn2) - choudn2@rpi.edu
Priya Goel (goelp) - goelp@rpi.edu
Cherry Bommu (bommuc) - bommuc@rpi.edu
Keegan Herron (herrok) - herrok@rpi.edu

Modified Datapath from Figure 4.24 for New Instructions



This project implements a full gate-level circuit representing the datapath for a reduced MIPS ISA. Our team expanded on the provided datapath by developing our own implementation for jump register and jump and link.

For jump register, a mux shown in the upper right of the datapath above was added, with a control line in updateState() deciding whether the read register should be the new address on is_jr == 1, or the normal PC on is_jr == 0;

For jump and link, two muxes were extended to support an extra input as shown. The control lines RegDst and MemToReg were likewise extended to two bits. The RegDst mux was updated to support passing in the 31st register, or the ra register, and the MemToReg mux was updated to support writing pc+4 (or +1 in the case of this project) to the write data register. Both of these muxes are set by control to 2 if the instruction is jal.

Design and Implementation Choices

UpdateState

UpdateState was implemented by using lots of temporary 32 bit variables. For each function that UpdateState calls, it pases in one of these temporary variables as the output parameter. Then, if this temporary variable needs to be used at a later point, it is passed into a 2-32 bit mux that decides if it or a different 32 bit variable continues through in the circuit.

Other than this, the implementation of UpdateState involves calling each of the other functions in the correct order, and passing in the correct control lines. As mentioned earlier, one control line was implemented in update state so that jump register would be possible. Because jr is an R instruction, an additional control line could not be put into the control function, as the control function does not have access to the funct bits. Therefore, this control line was created as a bit at the start of the UpdateState function, and affects a mux at the end of the function, where the PC is updated.

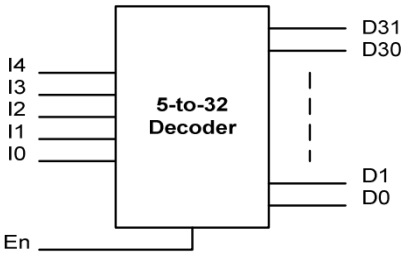
ExtendSign was simply implemented by copying the 16th bit to all the previous bits using a for loop.

Control

Control was implemented by first creating temporary bits for each possible instruction, such as is_lw and is_addi. For each temporary bit, the opcode was passed into an and gate to figure out which instruction was to be 1. For instance, for lw which has the opcode 110001, the code was BIT is_lw = and_gate6(OpCode[0], OpCode[1], not_gate(OpCode[2]), not_gate(OpCode[3]), not_gate(OpCode[4]), OpCode[5]);
These bits just simplify the coding, while still allowing the code to act as an actual circuit. Each of the control lines are dependent on which of these instruction bits are 1. For instance, jump is 1 if the instruction is jump or jump and link, and so the code for the jump control bit is *Jump = or_gate(is_j, is_jal).

Memory

This involved 3 separate 32 × 32-bit 2D arrays representing storage for the register file, instruction memory, and data memory. We addressed memory on the word boundary. Like mentioned in the instructions when we updated the PC we added +1 instead of the typical +4. Additionally, the jump and branch addresses did not require a shift left by 2. For instance, any jump value in an instruction was the actual address that it was jumped to. We utilized a 2-to-4 Decoder and a 3-to-8 Decoder to develop the 5-to-32 Decoder that was very useful in the circuit implementation. This 5-to-32 decoder was used to access and read/write a specific memory address that was passed into the function. The diagram below shows the 5-to-32 Decoder which was useful prior to implementation and function planning.



For each memory function we created SOP tables to determine the gates and required functionality.

SOP Table for Instruction Memory

0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	1	1	1
1	0	1	0

Instruction_Memory was implemented using a 5-to-32 Decoder with combinations of AND, OR, and NOT gates using the above SOP table to find the correct binary bit for each bit of the resulting binary instruction being built. For example, if a bit in MEM_Instruction is 0, the result from the decoder is 1, and the original bit in Instruction is 0, the new bit for Instruction becomes 0.

SOP Table for Read Register

0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1

0	0	1	0
0	1	1	1
1	1	1	1
1	0	1	0

Similar implementation to Instruction_Memory. Uses 2 5-to-32 Decoders for each register ReadData1 and ReadData2 while using AND, OR, and NOT gates to find the correct binary bit for each bit of the resulting binary sequence. The same gate logic was used in Instruction Memory, so the SOP table looks the same as well.

SOP Table for Write Register

This table was different as there was a modification when a column had a value of 1. This is due to the extra AND gate used with RegWrite.

0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1

In addition to using a 5-to-32 Decoder like the previous 2 functions, Write Register was implemented using a 2-input Multiplexor as well with the logic gates (using the SOP table above as a reference) for bits from RegWrite, MemRegister, WriteData, and the result of the decoder as input. The extra AND gate changes the logic in that when the third column is 1, the resulting output is the opposite of what it was previously (in the second highlighted group, the result is flipped when compared to the result in the first highlighted group). In short, when the third column is 1, the resulting binary bits are affected.

SOP Table for Data Memory

0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	1	1	1
1	0	1	0

0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1

Similar implementation to WriteRegister. The SOP table has the same changes from WriteRegister as well, as there was a modification when a column had a value of 1. Specifically, as stated in the previous function, when the third column is 1, the resulting binary bits are affected. This is due to the extra AND gate used with either MemRead or MemWrite (two implementations of the same logic for each variable, one for reading data and one for writing to memory). Therefore, a big difference between the functionality of this function and WriteRegister is that WriteRegister does not implement the SOP logic twice, while this one does. When writing to memory, we used a 2-input Multiplexor to pick an output bit from the gate logic and store it in MemData for later use.

ALU

The ALU was implemented by creating a driver function that calls the ALU32 from lab 6. Similar to how control figures out which instruction is being run from the opcode, the ALU first creates temporary bits to figure out which instruction is being run by the ALUControl bits. Then, setting Binvert, Op0, Op1 and CarryIn is straightforward. For instance, Binvert is 1 if the instruction is subtract or slt, so the code for Binvert is BIT Binvert = or_gate(is_sub, is_slt).

The only other needed addition to the ALU was the Zero bit. To implement this, the is_zero function was added, which is just an and gate to check that all the bits of the result are zero. If so, the Zero bit is set to 1.

ALU_Control

Similar to the implementation of Control, ALU_Control was also implemented by creating temporary bits for every possible instruction. For each of these temporary bits, the ALUOp for the instruction was passed into an AND gate to determine whether this instruction would be run. For all the R-type instructions, the funct is additionally passed into the AND gate.

These temporary bits are then used to determine the ALUControl. For each bit of the ALUControl, the temporary bits for all the instructions that would make the ALUControl bit a 1 are passed into an OR gate.

We implemented 2, 3, 4, 6, 8 input AND gates and 2, 3, 4, 5, 6 input OR gates.

Group Member Contributions

Julian Matthews

- GetInstructions
- UpdateState
- Control
- ALU
- Debugging
- Write Up

Nadia Choudhury

- InstructionMemory
- ALU Control
- DataMemory
- Write Up

Priya Goel

- InstructionMemory
- ReadRegister
- WriteRegister
- Write Up

Cherry Bommu

- DataMemory
- WriteRegister
- Write Up
- Code comments

Keegan Herron

- On Tuesday (12/7) he sent a message saying he was unable to contribute to the project in a meaningful way and would accept a 0 on the project.