CS 3410

Project 2 Design Document

Andres Antonsson & Jesse Potts

September 24, 2018

1. **Overview**

The intended purpose of this project was designing a 32-bit MIPS CPU which processes most MIPS code instructions in an efficient manner. To achieve this, the CPU uses a pipelined system with data forwarding to read and process instructions.

1. **Component Design Documentation**

**2.1 MIPS32**

A general diagram of the 32-bit MIPS CPU is shown below. It can process 23 of the main functions required for this CPU, as described below. The processor is pipelined into 5 different sections highlighted in different colors. In order they include: fetch, decode, execute, memory, and write back stages. Each stage is separated by registers that hold the previous stages information and pass it onto the next stage when the clock rises.



**Data Paths**

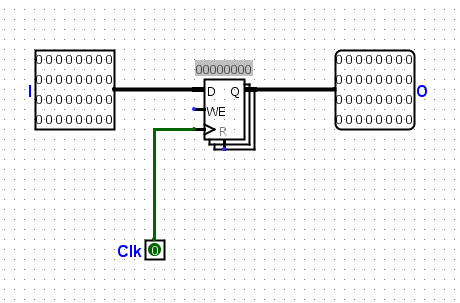
**Instruction Paths**

Instructions begin in the MIPS program ROM in the fetch stage before being passed into decode stage. Here, the instruction is decoded (explained later under data paths) but also passed into the control unit for the next stage, execute. Here, the instruction is used to decide what data to use through MUX controls including: IMM, LUI, Com, SU, ALU, and MOV. IMM decides weather to use the last 16 bits of the instruction as B for the ALU. LUI decides weather to use the ALU output or the LUI sub-circuit result if the instruction is a LUI. Com works the same way LUI does but selects for the compare sub-circuit result if the instruction is a SLT, SLTU, STLI, or SLTIU. SU gets sent to the compare sub-circuit to decide whether to use the signed or unsigned comparator. ALU decodes the instruction in order to send the right arithmetic instruction to the ALU. Finally, MOV decides whether to pass-over the ALU because the data should remained un-changed. After the execute stage, the data is passed into the memory stage. While the memory stage has nothing implemented for memory, the control unit is necessary for forwarding in the execute stage (which is discussed later). Finally, the writeback stage control unit decodes the instruction to find the destination of the data (just like the memory stage control unit) and signals if writing to the register file should be enabled.

**Data Paths**

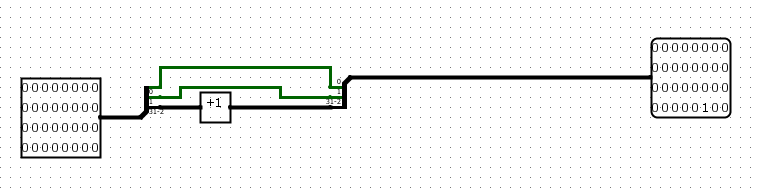
After reaching the decode stage, the instructions are decoded to find the two values used for A and B or rs and rt as well as the shift amount and the immediate value if necessary. The data is then passed into the execute stage where it is all processed. This is the stage that handles forwarding and every computation the instructions ask for. Forwarding works by comparing the register addresses required for the current instruction to the destination register addresses of the past two instructions since they haven’t been written to the register file yet. The destination addresses are fed to the forwarding sub-circuit by the control units of the memory and write back stages. This outputs control signals to MUX’s that control where the inputs for A and B for the ALU come from whether it is the register file, memory stage, or write back stage. Depending on the instruction, the data can be directed to the ALU, compare, LUI, and/or MOV circuits. While the data flows through all these circuits at all times, the outputs are selected for as discussed previously. For MOV however, the full instruction is always passed through it and is usually unchanged except for certain move instructions. If an instruction calls for MOVN or MOVZ, this circuit tests if the move should happen (based off of requirements given by the MIPS manual) and if the move should not happen, it changes the destination register to the source register so it does not move. Finally, the data passes through the memory stage and gets written to the register file in the write back stage.

**2.2 PC**



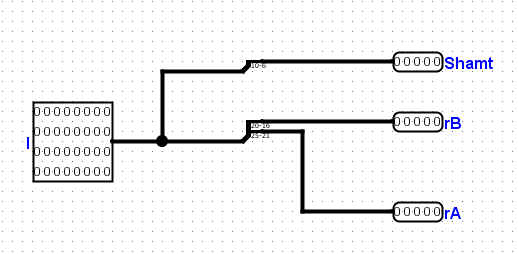
The PC is just a register that holds the current value of the instruction address which passes it into the incrementer which it also gets its input from.

* 1. **Incrementer**



The incrementer adds 4 to the current instruction address by adding one to the top 30 bits and concatenating the bottom two bits back onto it (which should generally always be 0’s anyway).

* 1. **Decode**

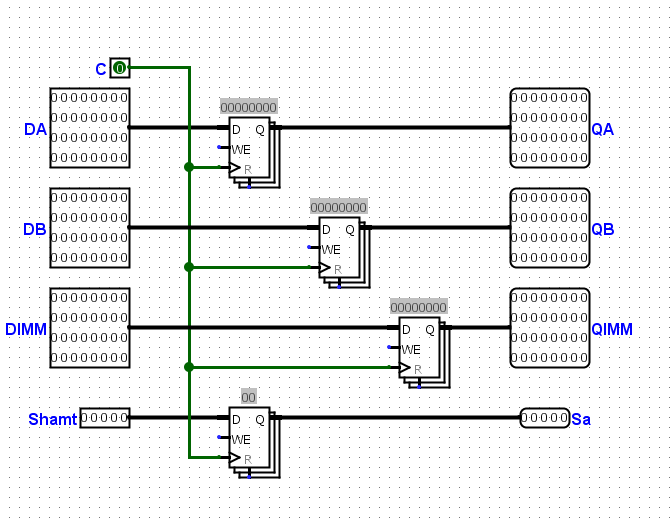


The decode sub-circuit takes in the instruction and outputs:

1. The shift amount just in case it is a shift instruction
2. The register address for rs which corresponds to A
3. The register address for rt which corresponds to B

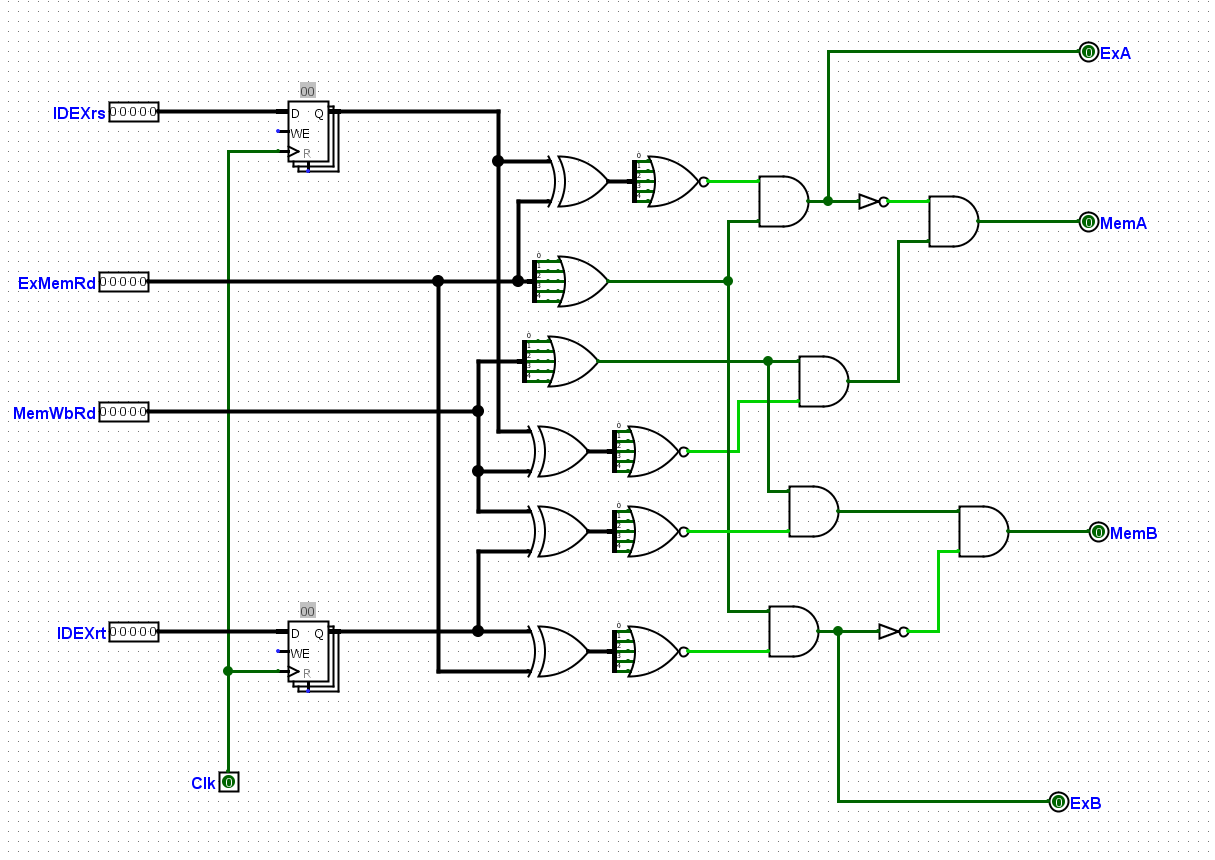
And outside of the decode circuit, the last 16 bits are passed into IDEX just incase it is an immediate instruction.

* 1. **IDEX**



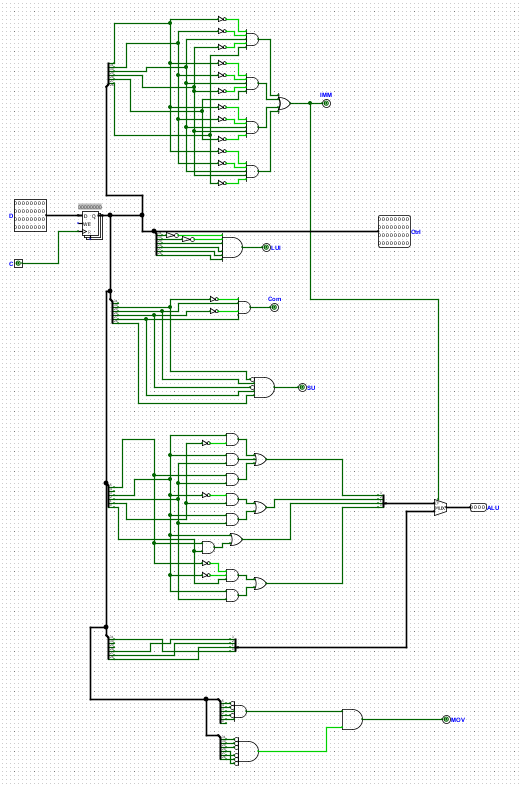
The IDEX sub-circuit just acts as a hub for all of the registers required to separate the decode and execution stages.

* 1. **Forward**



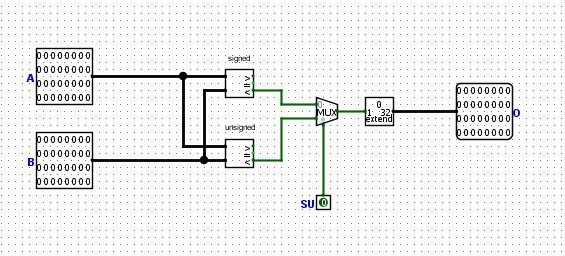
The forward sub-circuit contains logic that decides if the input for the execute stage needs to be sourced from the register file, memory stage, or write back stage. Using the logic from the project overview, this logic successfully controls four MUX’s in the execute stage connected to the sources previously mentioned.

* 1. **CtrlIDEX**



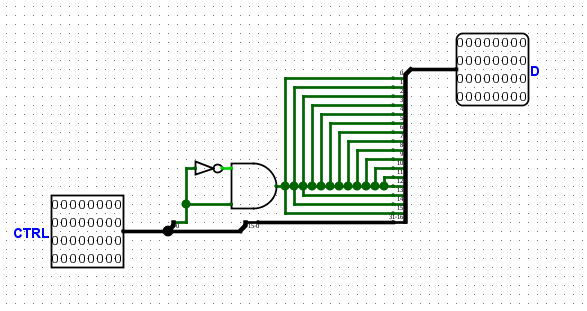
The ctrlIDEX takes in the full instruction from the decode stage and outputs control signals for most of the MUX’s in the execute stage and passes on the instruction to the MOV sub-circuit. Most of the selector signals function as previously described in ‘Data Paths.’ ALU however outputs the correct ALU code based on the last 6 bits of the instruction unless it is an immediate instruction in which case the immediate control signal switches the ALU opcode to the secondary logic beneath the R-type logic.

* 1. **Compare**



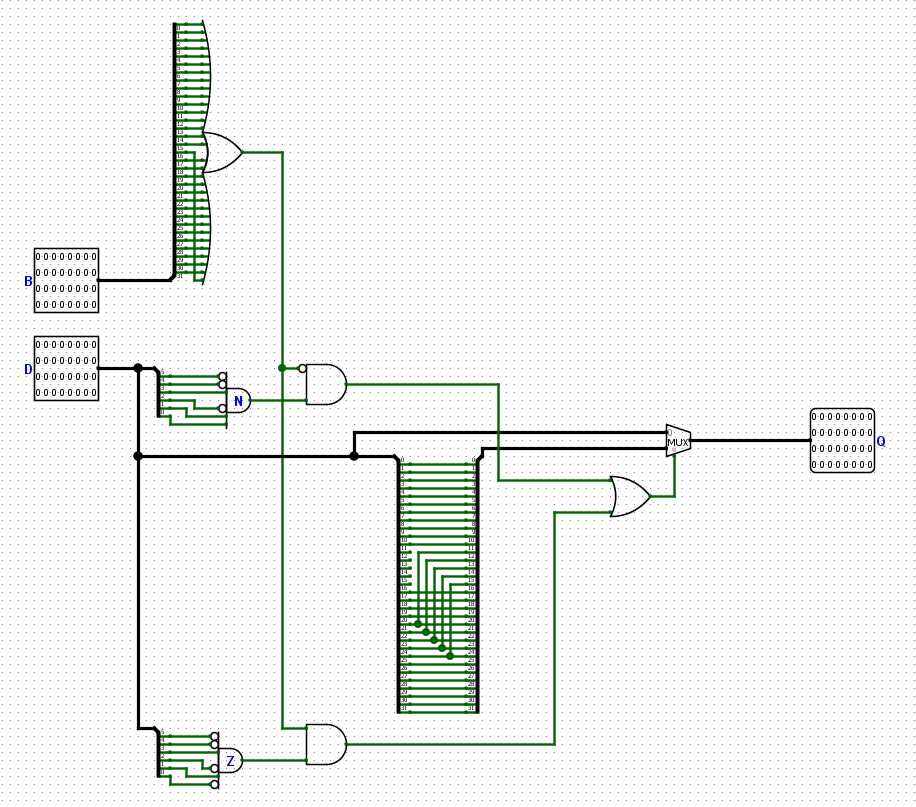
The compare sub-circuit takes in A and B as data and SU as a selector signal to choose between the signed and unsigned comparators. The result from the comparators are bit extended with zero’s to become 32-bit.

* 1. **LUI**



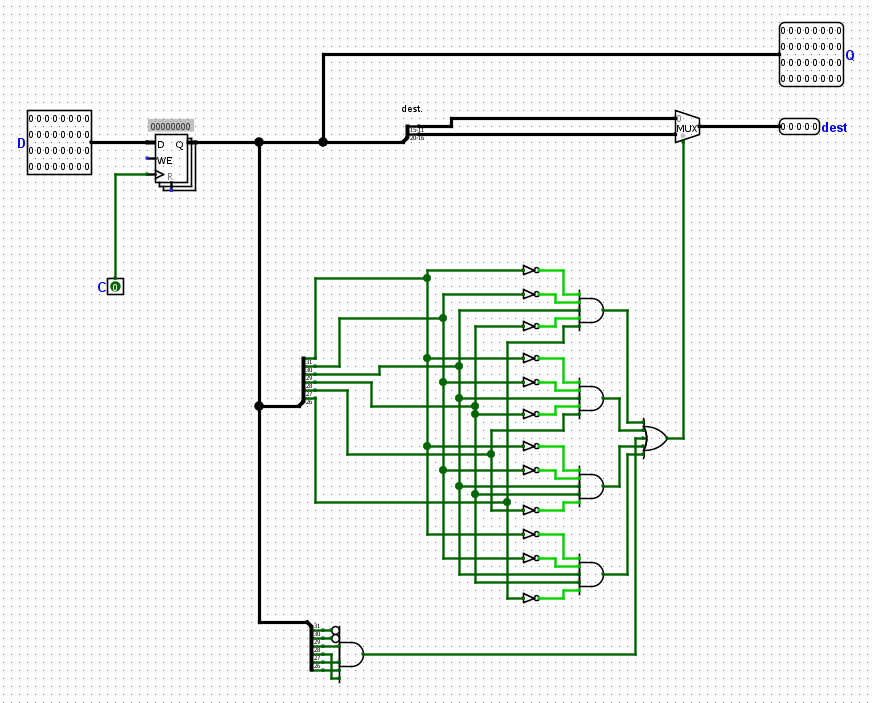
The LUI sub-circuit takes care of load upper immediate functions by copying the lower 16 bits to the top 16 bits of the output and replacing all 16 lower bits with zeros. It acquires the zero value by ANDing the least significant bit with it’s negative so that it can never output a one and passes that into the 16 least significant bits.

* 1. **MOV**



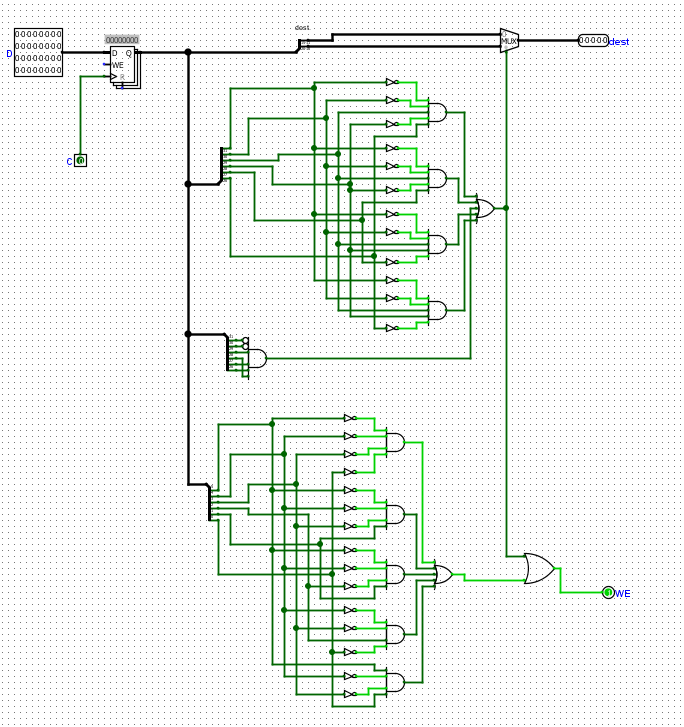
The MOV sub-circuit tests if the instruction is a MOVN or MOVZ type. It also tests if those moves should occur based on the value of B (whether it equals 0 or not). If it is a move instruction but it is not supposed to move based on the move rules, the destination register (rd) is changed to the source address (rs) using the splitters.

* 1. **CtrlEXMEM**



The CtrlEXMEM sub-circuit holds and passes the instruction onto the next stage but also passes the destination register address back to the forward sub-circuit using the logic shown. (Testing if it is an immediate or LUI instruction vs an R-type).

* 1. **CtrlMEMWB**



The CtrlMEMWB sub-circuit returns the correct destination register address to forwarding logic and register file in the same way CtrlExMem did. It also returns a control signal to the register file whether to enable write or not. It returns a one if the instruction is one that can be handled (as shown in table A on the overview).

1. **Design Justification**

One design choice we made was in the MOV sub-circuit and CtrlMEMWB circuit. Instead of just disabling write enabled (WE) in the CtrlMemWB, which in hindsight probably would have been easier, we changed the write location of the move instruction in the MOV circuit so that it would not be written to the original destination register if the move is not supposed to happen.