ECE 4435 Computer Architecture & Design

Lab 12 Final Project Report

Name: Hong Moon

Date: 4/29/2014

This is the final report for the ECE 4435 project for building a 5-stage general purpose pipelined CPU, given an Instruction Set Architecture (ISA) specification. The control signals to be decoded from the ROMs in the Decode Stage have been described in a separate Excel Spreadsheet. The CPU worked correctly, passing the test cases. The project has been checked off by the TA's.

1. Fetch Stage (lab 7):

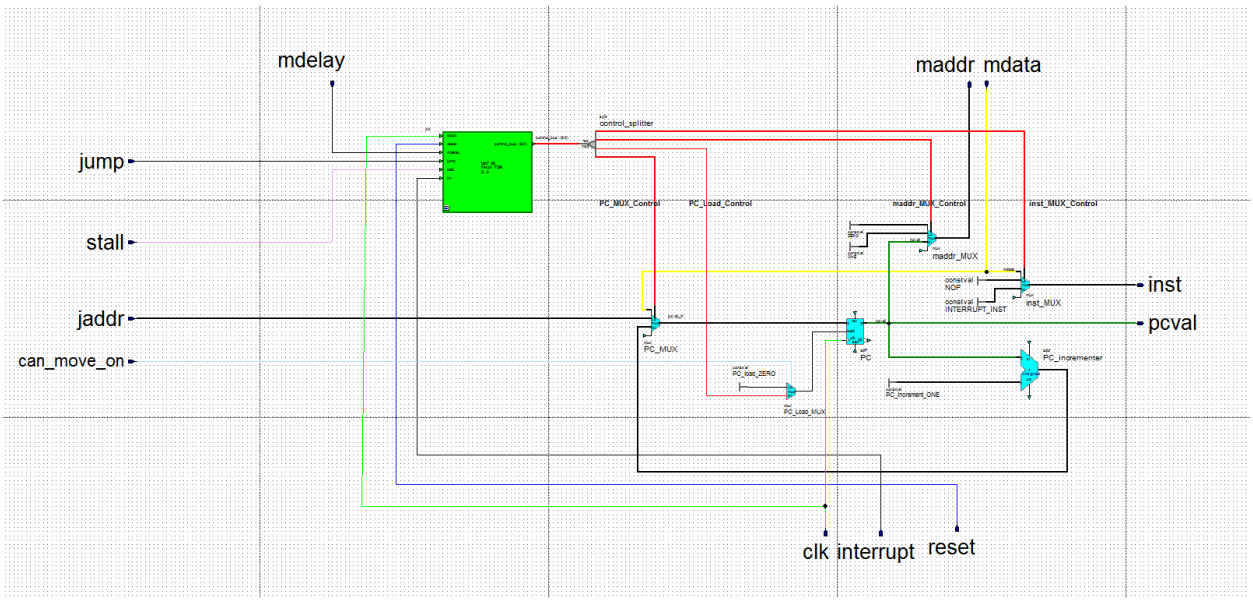

Figure 1. Fetch Stage Overview

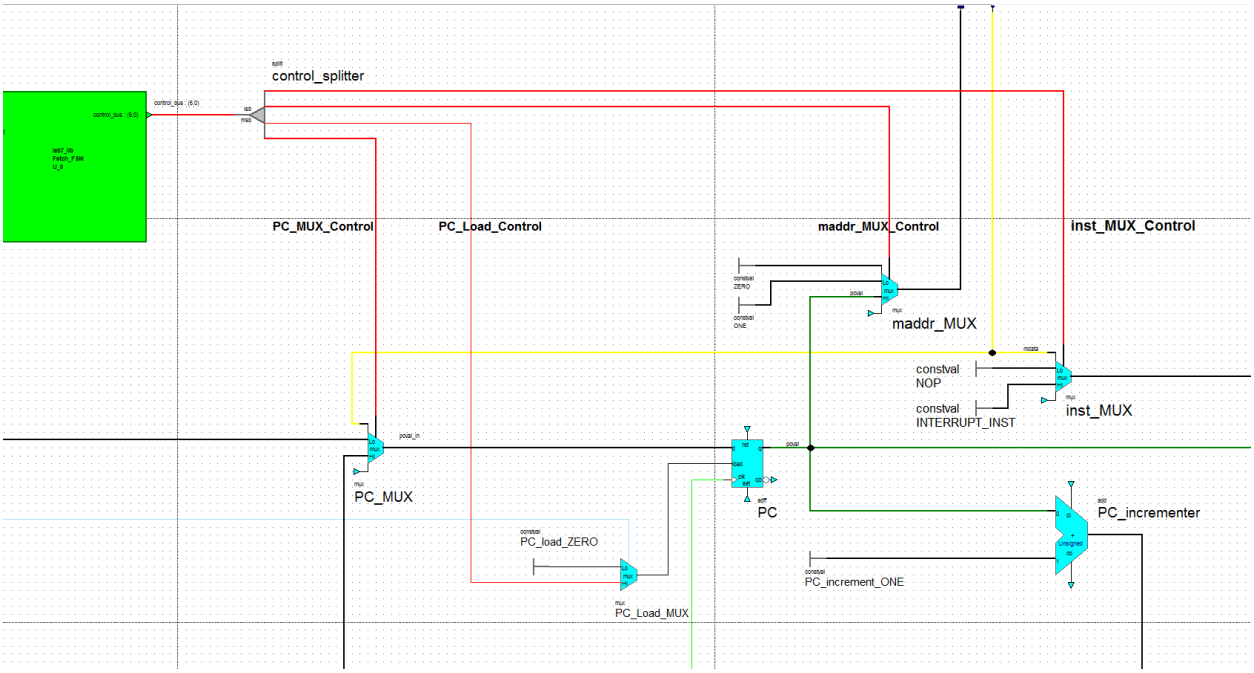

Figure 2. Fetch Stage Close-up View

The *Fetch Stage* fetches an instruction from the memory by sending the address of the instruction. The address is sent to the *Memory Arbiter* to mitigate the structural hazard. The *Fetch Stage* centers on a Finite State Machine (FSM), which outputs control signals for the multiplexers within the stage. The FSM has been coded in VHDL. The program counter is incremented by one if the next instruction needs to be fetched.

The FSM has three states: reset, interrupt, and the normal fetch. While in the reset state, the Fetch Stage fetches the address of the first instruction, which is at memory address 0, and sends a No op instruction. While in the interrupt stage, the *Fetch Stage* fetches the instruction at memory address 1 and sends a special instruction indicating an interrupt. While in the normal fetch state, the Fetch Stage fetches the next instruction and loads a new pc value to the program counter.

However, while still in the normal fetch state, there are some control inputs to the FSM that override the decisions that have been made. The jump input has the highest priority, and the stall and

memory delay signal have the next highest priority. If the jump signal has been asserted, the jump address is loaded to the program counter instead, and a No op instruction is sent. If a stall or memory delay has been asserted, the load to the program counter is disabled, and a No op instruction is sent.

To mitigate the data hazard, there is an input signal named "*can_move_on*", which indicates whether the instruction currently decoded in the Decode Stage can proceed without creating a data hazard. The details for this process will be explained in greater detail in the section for *Register Tracker*. If the instruction in the decode stage cannot move on to the Execute Stage, the program counter is disabled to stall the pipeline.

2. Decode Stage (lab 8)

The *Decode Stage* decodes the instruction, sends the addresses of the registers to be used, and produces a set of control bits from either the Decode ROM or the ALU ROM. If the instruction bit 15 to 13 is equal to "101", then the ALU ROM is used, for arithmetic and logical operations. Otherwise, the Decode ROM is used. To mitigate the data hazard, the load enable to the registers in the *Decode Stage* pipeline is enabled when there is no stall signal from the *Memory Stage* and the "*can_move_on*" signal is high. To mitigate the control hazard, the Control output should be turned into one equivalent to a No op when the stall or jump input signal is high.

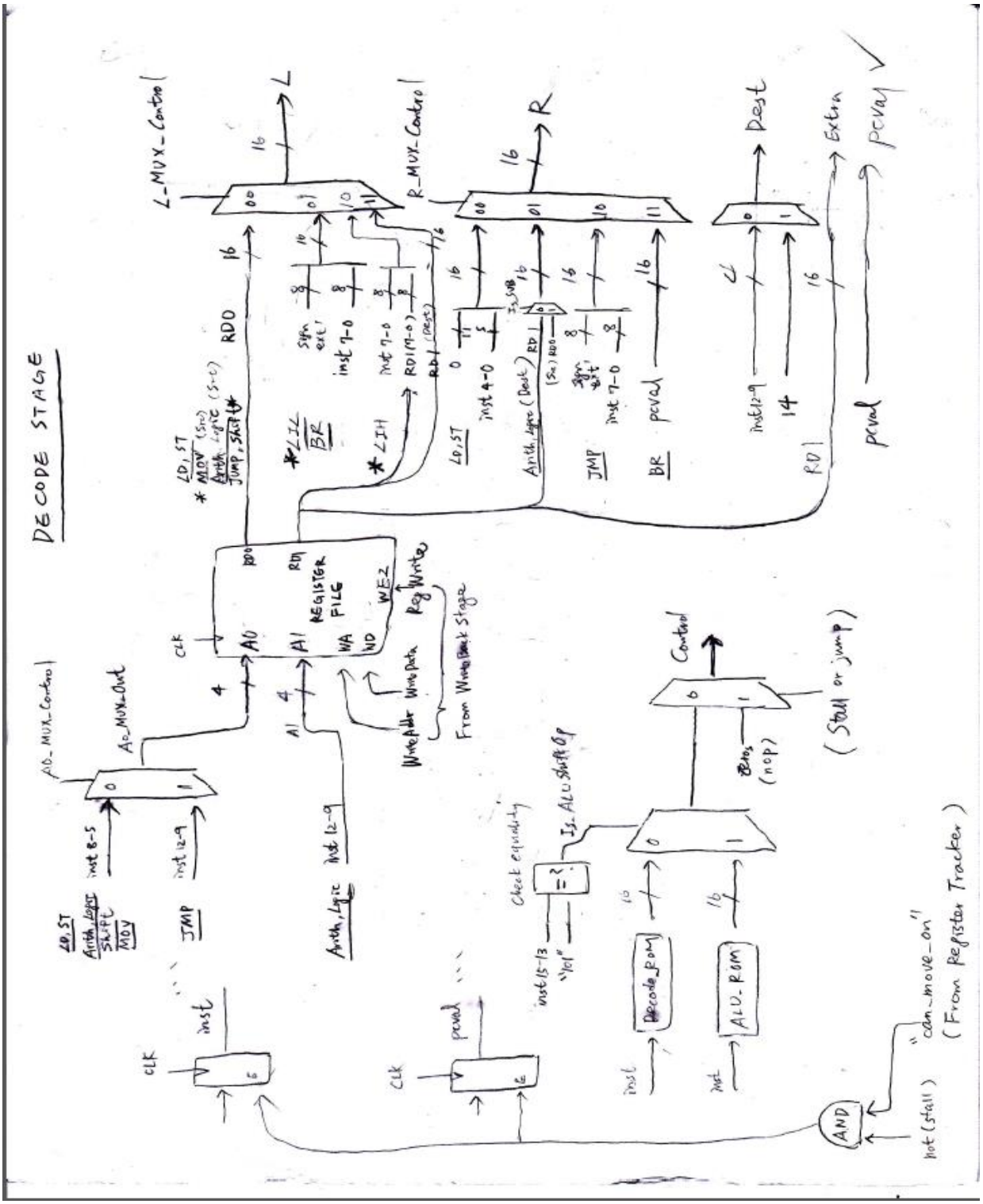

Figure 3. Decode Stage

3. Execute Stage (lab 9)

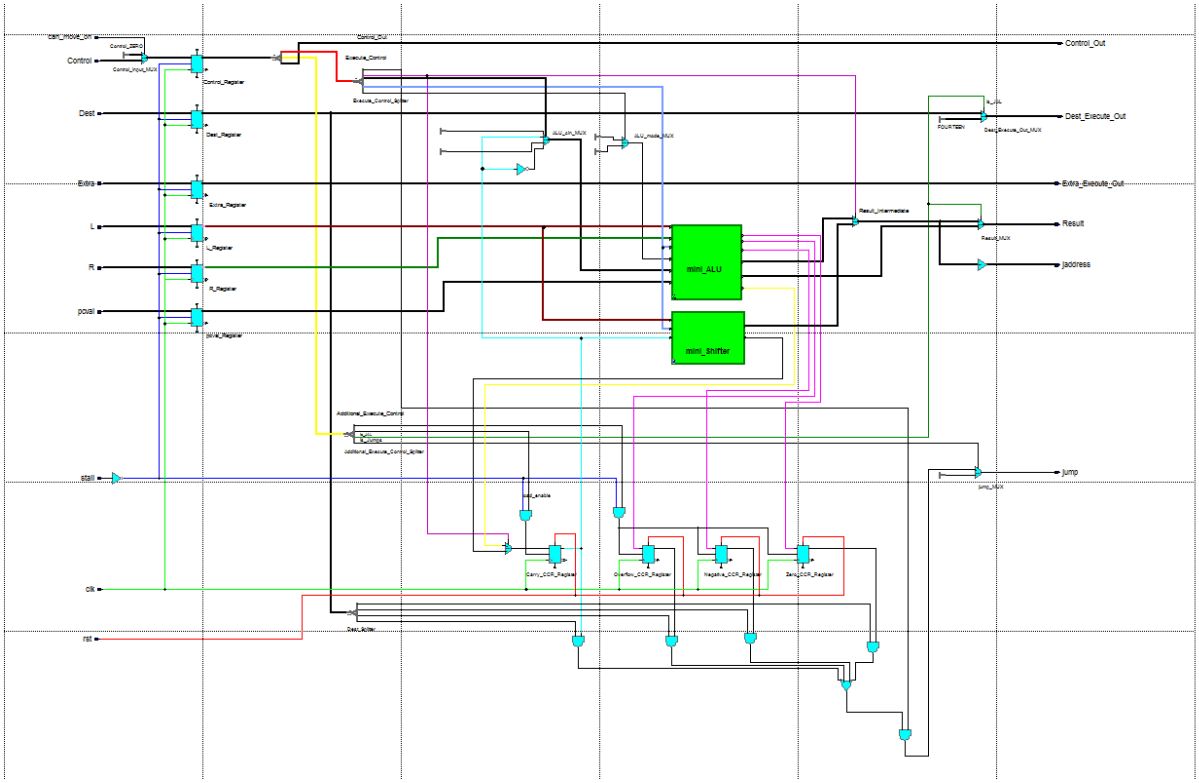

Figure 4. Execute Stage Overview

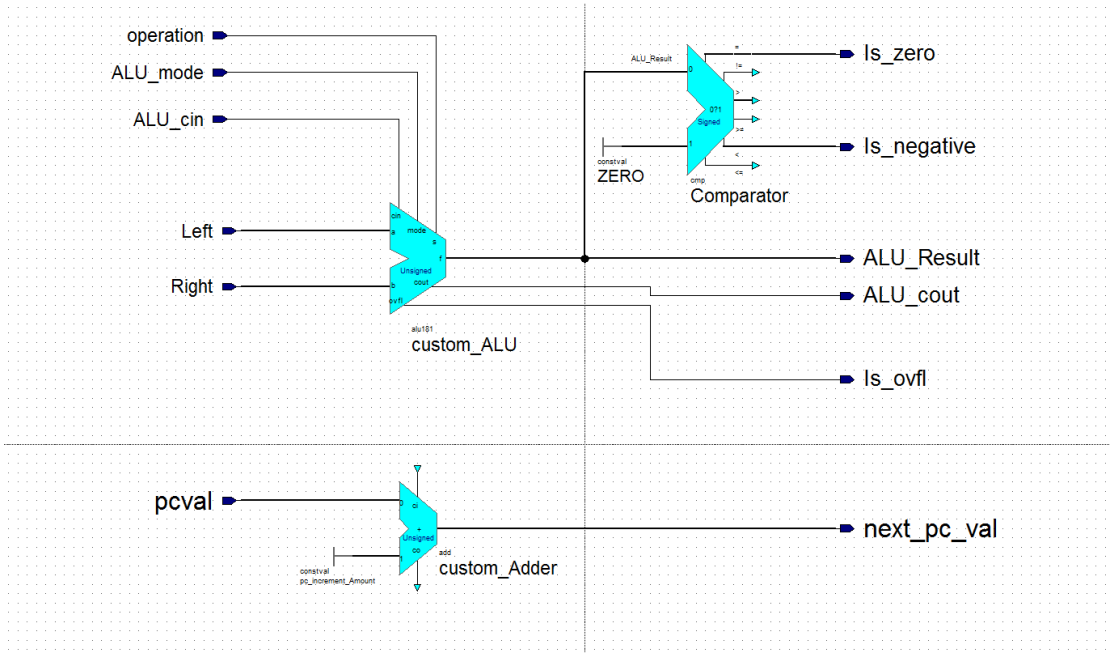

Figure 5. mini-ALU Unit inside the Execute Stage

"*Mini-ALU*" is a customized ALU block producing the ALU result as well as creating extra outputs: (1) whether the result is zero, (2) whether the result is negative (3) whether the result has generated a carry-out (4) whether the result has generated an overflow. Also, the adder incrementing the pc value calculates the next instruction following the JAL instruction. "Mini-Shifter" is a customized Shifter block produced by coding in pure VHDL.

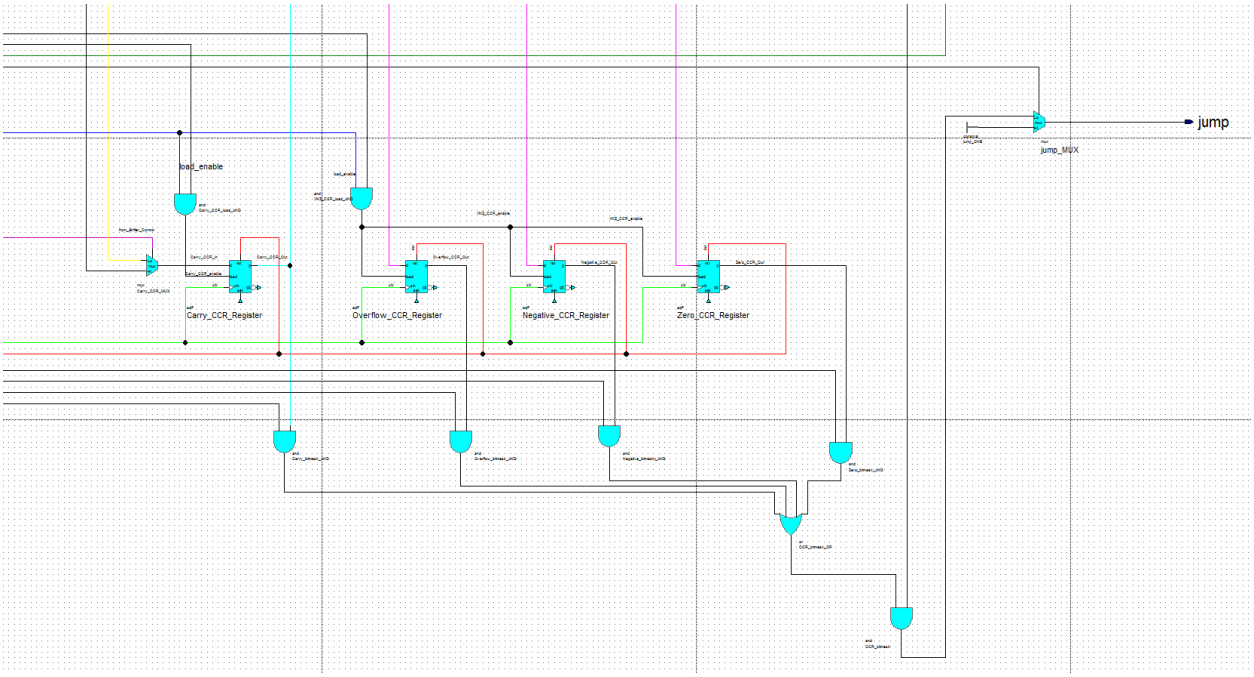

Figure 6. Execute Stage Jump Control Block Close-up View 1

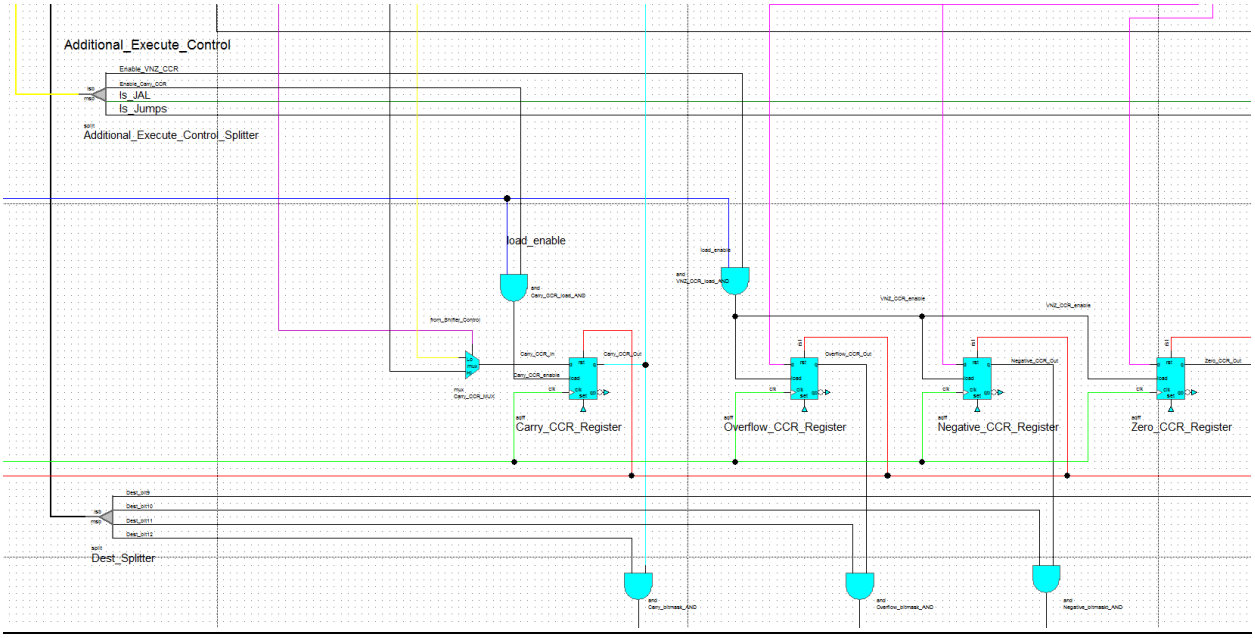

Figure 7. Execute Stage Jump Control Block Close-up View 2

There is also a block handling the jump and branch control. To handle the BC, BO, BN, BZ instructions, the bits stored in the Condition Code Registers (CCR) from the previous instruction are bit-masked with bits 12 to 9 (then AND'ed with a jump control bit in case the bit mask result is high in an instruction completely unrelated to branch) to decide whether to branch to a new instruction or not. In case of an unconditional jump (JMP, JAL, BR), there is a MUX that overrides the previous logic to set the jump output high. To mitigate the control hazard, the instructions in the *Fetch Stage* and *Decode Stage* should be turned into No op's when the jump signal is high.

4. Memory Stage (lab 10)

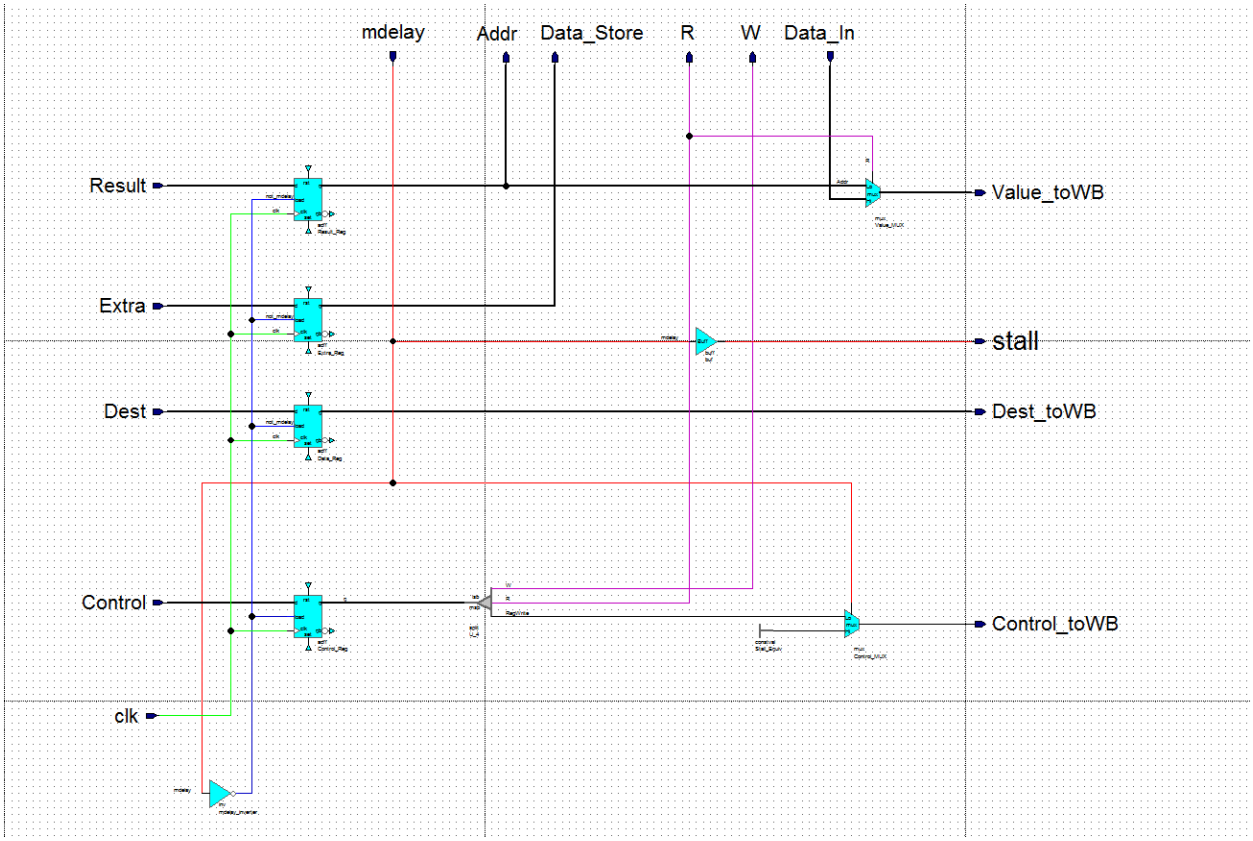

Figure 8. Memory Stage Overview

Memory Stage is very simple. If the R input has been asserted (in case of a LD instruction) the data from the memory is sent to the *Write-back Stage*. If not, the result value calculated from the *Execute Stage* is sent to the *Write-back Stage*. If the W input has been asserted (in case of a ST instruction) the data from *Extra* (which is the source register content obtained from the *Decode Stage*) is sent to the memory to be stored. If there is a memory delay, the pipeline is stalled by disabling the load enable to the registers, as well as the registers from the previous stages in the pipeline. However, although the initial design considered such memory delay, the provided memory was an SRAM with no delay so this case would not actually happen.

5. Write-Back Stage (lab 10)

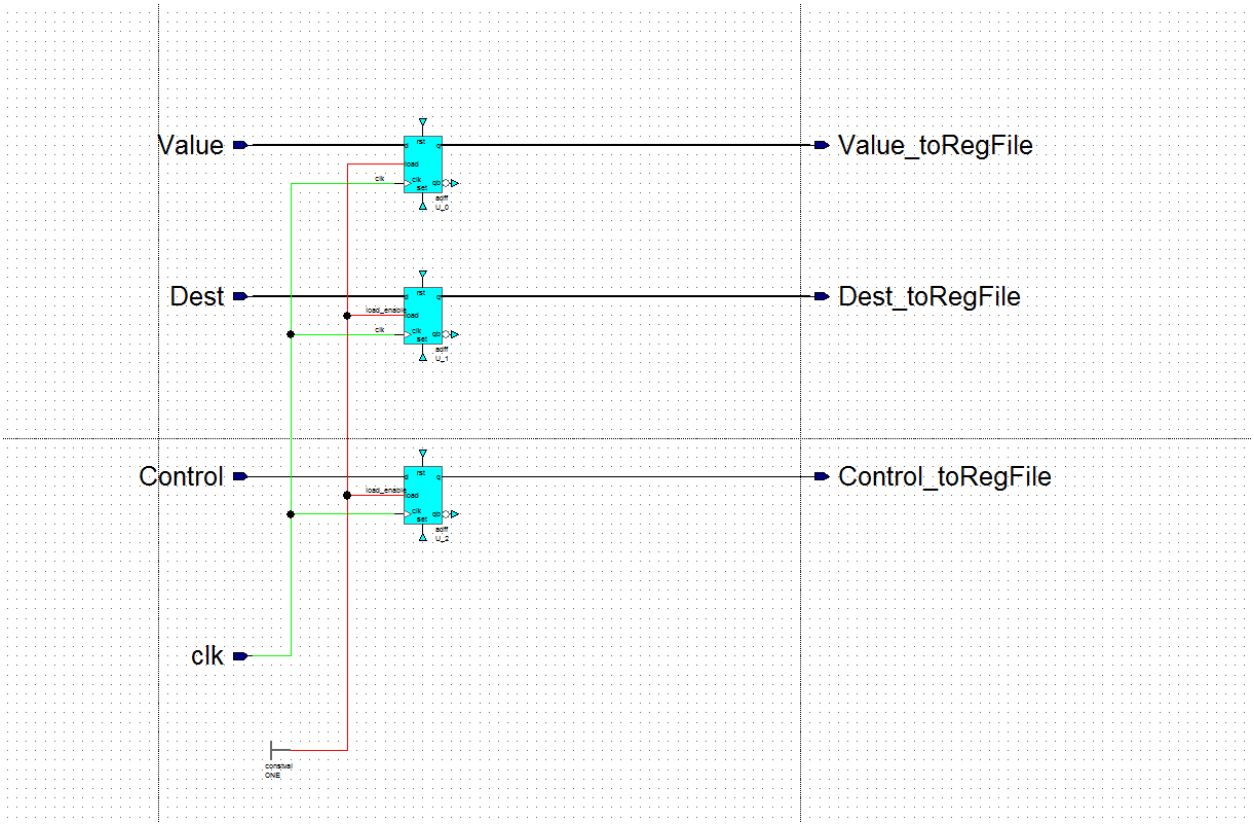

Figure 9. Write-Back Stage Overview

Write-back stage is very simple. The outputs are fed to the Register File and the Register Tracker. To the Register File, the *Write-back stage* just sends the value of computation to the destination register. The only 1-bit control is whether to enable writing to the destination register. To the *Register Tracker*, these outputs are used to decide whether to clear off the reservation status of the destination register.

6. Register File (lab 10)

Register File is simple and is the same as the one designed for one of the previous labs. There are 16 registers that have been modified to have one 16-bit write input but two read outputs. There are two Read Decoders, each receiving a 4-bit read address and producing a one-hot output, which will be fed to an array of registers to read values. Likewise, there is one Write Decoder that receives a 4-bit write address and produces a one-hot output, which will be fed to the array of registers to write a value.

7. Memory Arbiter (lab 11)

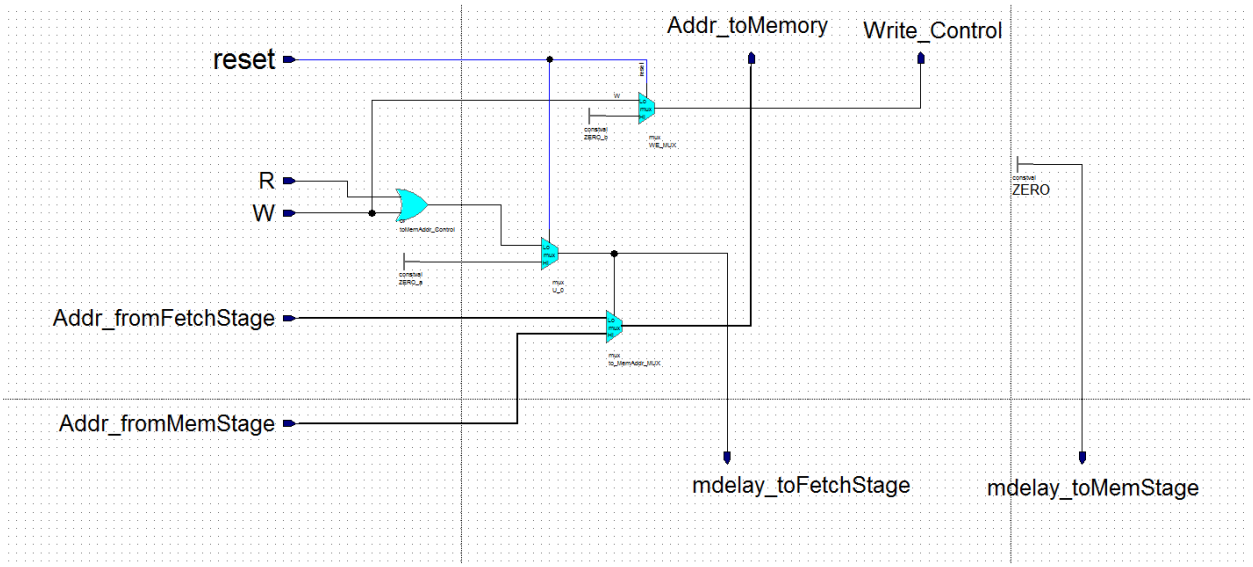

Figure 10. Memory Arbiter Overview

Memory Arbiter was introduced to mitigate the structural hazard. Since both *Fetch Stage* and *Memory Stage* need to use the same single memory at the same time, both are sending an address. Instead of sending the address directly to the memory, the arbiter chooses which address to send to the memory. If the R or W signal (Read or Write) has been asserted, choose the address from the *Memory Stage*. If neither of the R and W has been asserted, choose the address from the Fetch stage. Since the memory provided (lab 12) for the project is an SRAM with no delay output, to incorporate it into the existing design, the memory delay signal is connected to a constant zero. In case of a reset, choose the memory from the *Fetch Stage* to read the start instruction from the memory.

8. Register Tracker (lab 11)

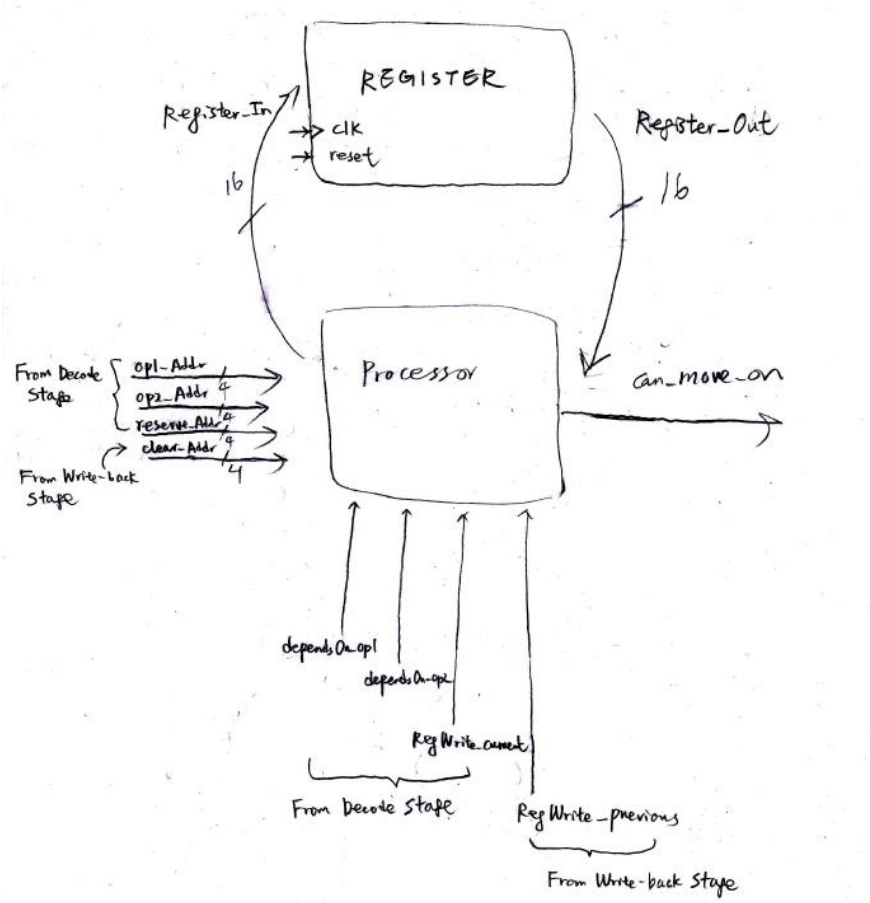

Figure 11. Register Tracker

Register Tracker was introduced to mitigate the data hazard. There are situations in which the operands from one instruction are dependent on the results of the previous instructions that have not yet been completed. This happens specifically when the destination register for the uncompleted instructions matches with the source registers from the current instruction.

Upon starting or resetting the machine, all bits to the Register Tracker are cleared off to 0. Then, for each instruction, set the bit in the *Register Tracker* corresponding to the destination register as “dirty” or reserved. If the instruction is completed, the register reservation status in the *Register Tracker* is cleared off. The *Register Tracker* consists of a 16-bit register and a combinational logic, named

"*Processor*," that does all the heavy lifting to produce an output of whether the instruction decoded in the *Decode Stage* can move onto the *Execute Stage* of the pipeline.

There are cases in which decoded instructions do not actually depend on the operand registers, or depend on only one of the registers. So there are inputs indicating whether the instruction is actually dependent on registers for operand 1 and 2. The *Processor* also creates a 16-bit output called *"Register_In",* indicating the updated register status, and feeds it into the register.

If the "*can_move_on*" is disabled, the control to the *Execute Stage* should be converted to one equivalent to a No op, and the load-enable to the pipeline registers to the *Fetch Stage* and the *Decode Stage* should be disabled.

9. Memory (lab 12)

The VHDL source for memory was provided by Professor Williams. Unlike the initial expectation, the implemented memory was an SRAM with no "wait" output because it would always respond within one clock cycle. The memory has a 16-bit address input, a write enable, and a 16-bit data input and an output.

10. Overall Architecture

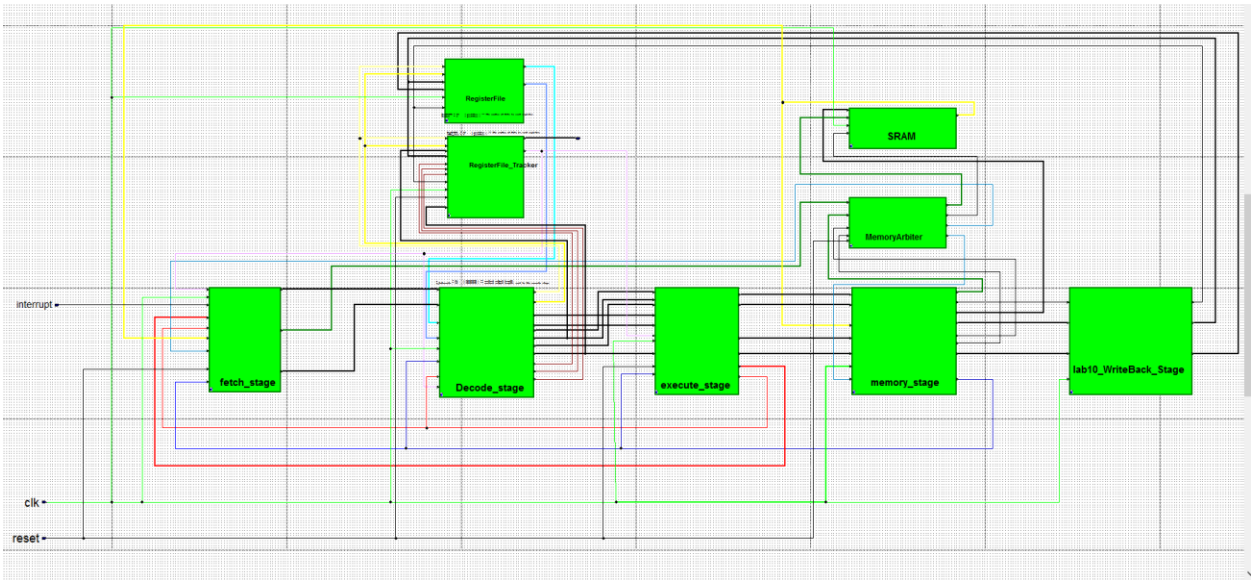

Figure 12. Overall Design

Here is the top level view of the overall design. The inputs to the system are clock, reset, and interrupt signals. The five stages (Fetch, Decode, Execute, Memory, Write-back) of the pipeline are connected with the Register File, Register Tracker, Memory Arbiter, and the SRAM memory.

Note

To run the program, open the “lab10_MergeAll” and lab10_Top_Level_lib. The following libraries should have been imported: (1) lab7_lib, (2) lab8_new_lib, (3) lab9_new_lib, (4) lab10_memory_stage, (5) lab10_WriteBack_Stage (6) lab10_RegisterFile_lib, (7) lab10_TopLevel_lib, (8) lab11_MemoryArbiter_lib, (9) lab11_RegisterTracker_lib, (10) lab12_Memory_lib. “lab10_MergeAll” sounds confusing, but that is when I started putting things together, and that is the top level directory, not the one starting with lab12.