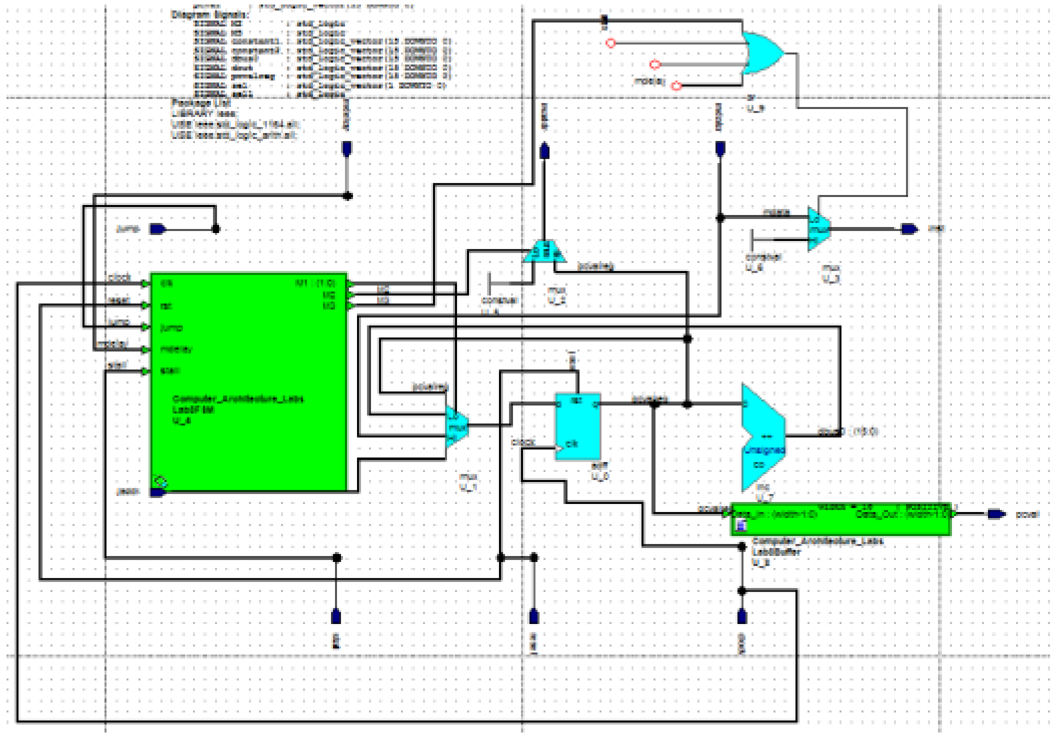
Computer Architecture
Final Lab Report

By Nader Maharmeh

To have a semester's worth of work be reliant on two lines of text is a special form of torture, but when those two fateful numbers appear after going cross-eyed from staring at green and red lines for days on end you realize the meaning of life was not in fact 42, but 0 and 16.

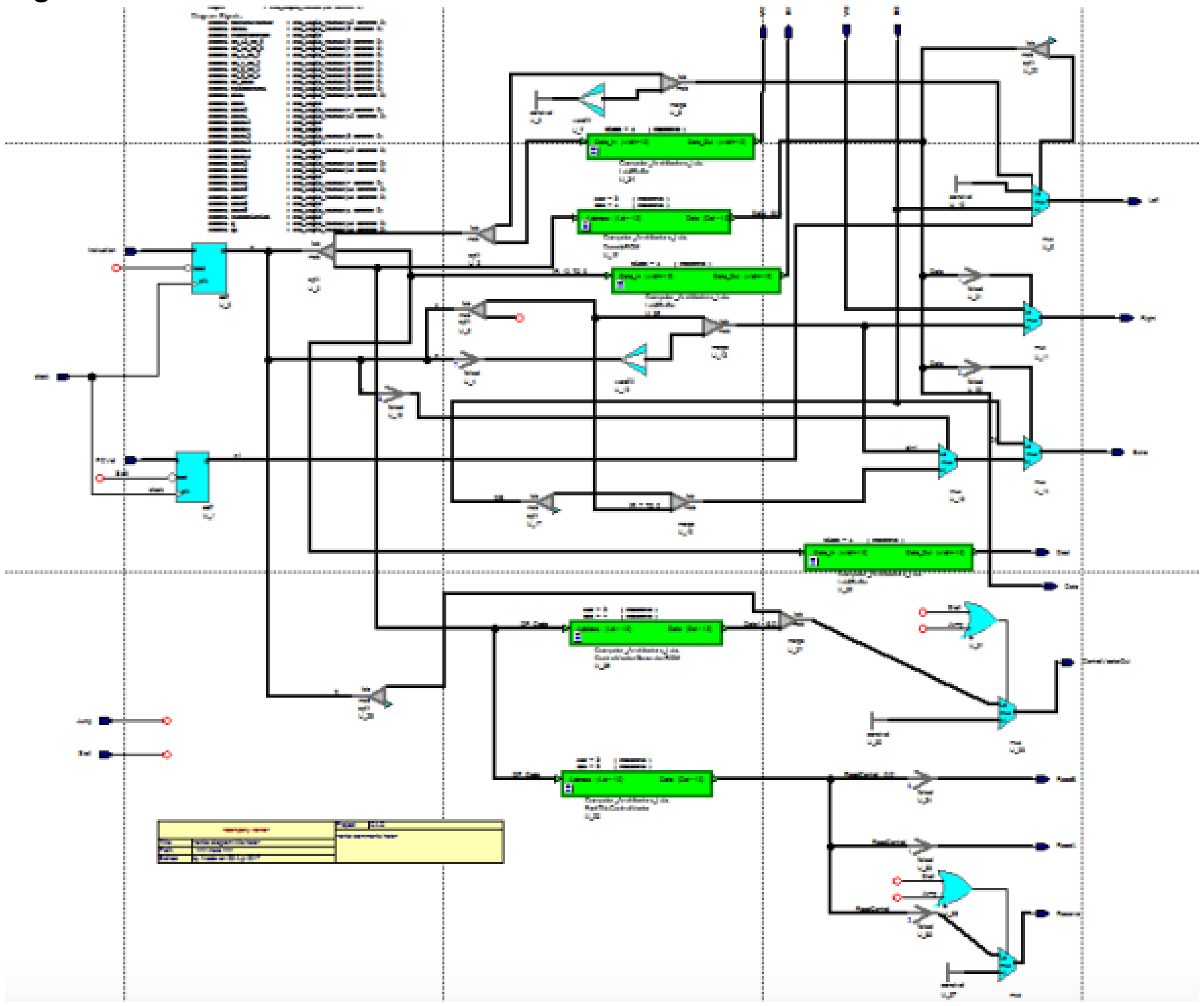
The processor consists of 5 main stages, plus a few peripheral components to account for hazards and to interface with memory. Below is a brief description of each stage and component.

Stage 1: Fetch



The fetch stage upon reset will fetch the address from memory by sending maddr 0 to memory, and it will receive back the address to where the program is contained. Then fetch will send that to memory and receive back the first instruction of the program. The instruction is then sent to Decode. In typical operation, fetch sequentially sends the next instruction and the PC is incremented. If mdelay, jump, or stall is received, a no-op is sent instead.

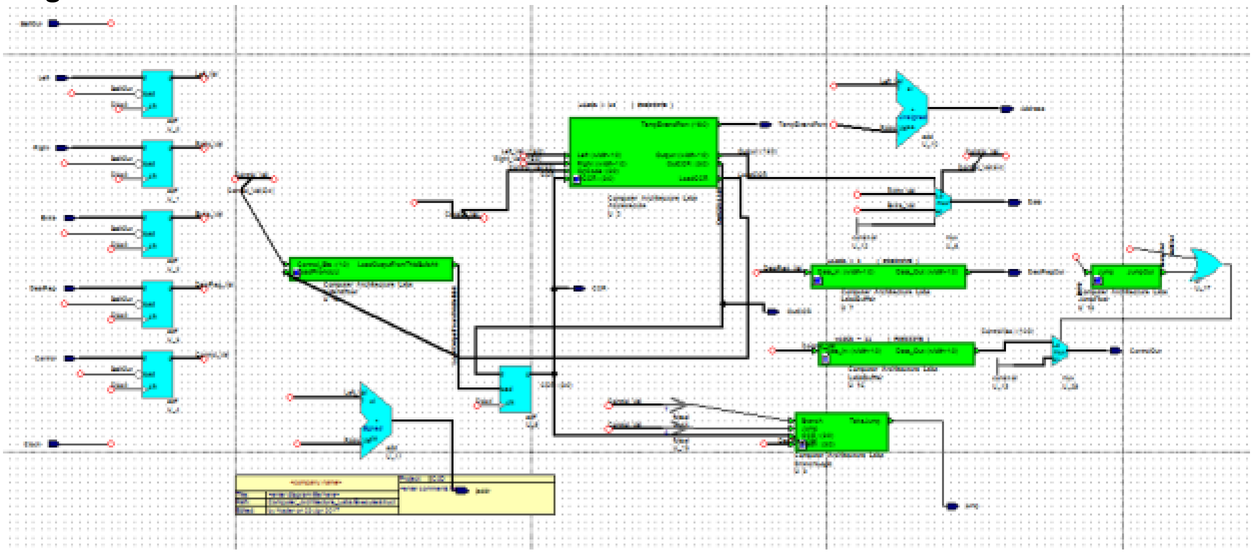
Stage 2: Decode



In short, the Decode stage takes in the instruction from the fetch stage and breaks it out to construct the control vector that is sent out to the other stages. This stage also involves getting and sending stuff from memory such as the address that want to be written to or read from. In terms of hazards, this stage also sends out the reserve bit command to mitigate for RAW hazards. Multiple Control Vectors were also made for different components such as inducing stalls in other stages and to distinguish as to when the ALU is required.

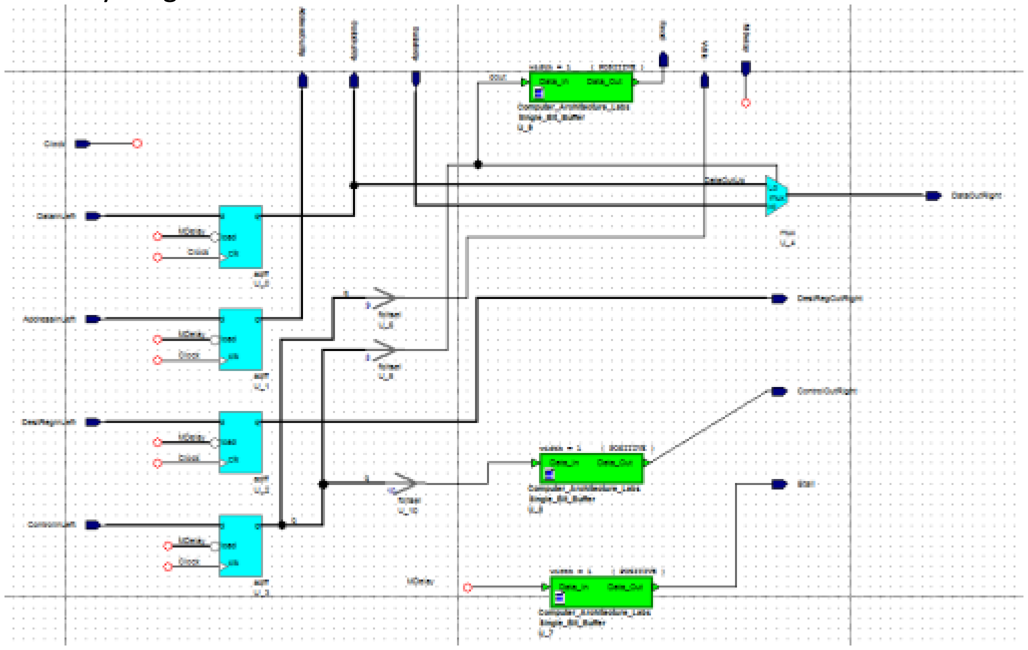
One grand error in the Decode stage that gave me a lot of trouble was not realizing that the No-Op instruction of all 0's I was sending was being piped into the ALU of the Execute stage (up next), which caused the first operation (ADD) to be triggered mistakenly. This was easily fixed by changing the control vector for No-Op to be completely unique and adding an if statement for that unique op-code within the ALU that did not load the CCR register.

Stage 3: Execute



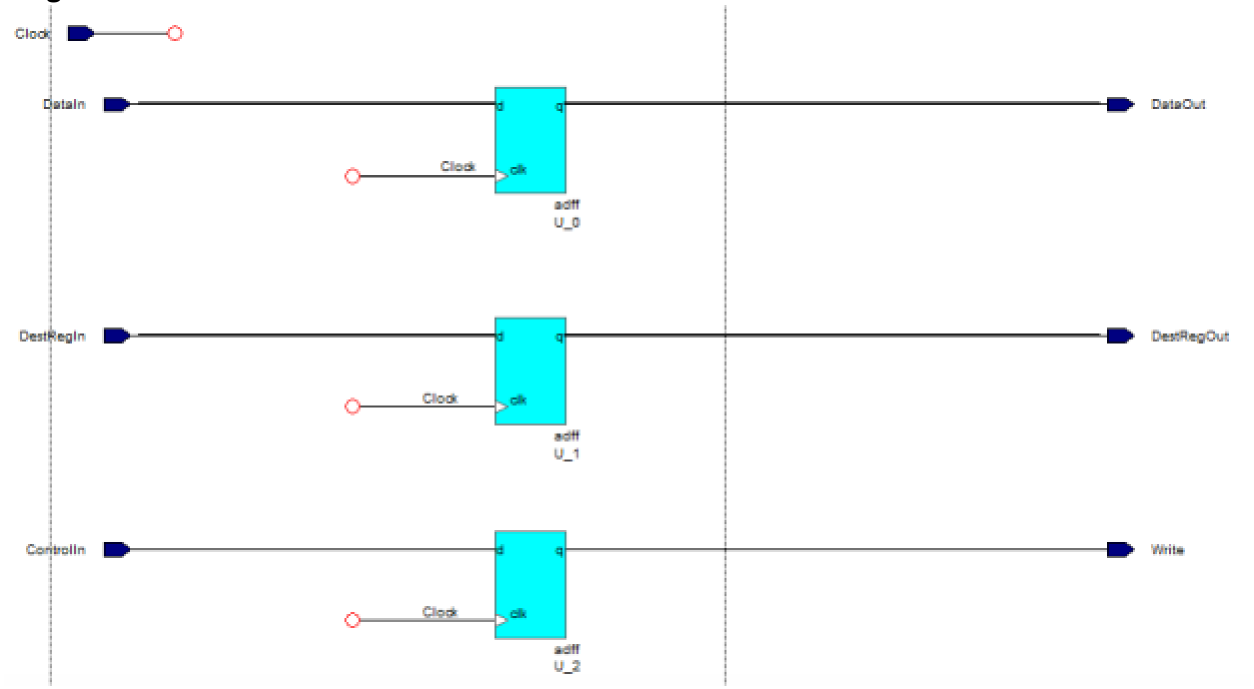
The Execute stage contains all the logic and computation of the processor. Here the ALU is in charge of performing the necessary computations to actually complete the tasks given by the instructions of the program. The Execute stage also maintain the condition code register or CCR, which is essentially a 4 bit vector consisting of the Carry, Overflow, Negative, and Zero indicators. This stage also implements the logic for branching and jumping, and output the appropriate bits and address depending on the control vector, CCR, and mask from the instruction (destination register).

Stage 4: Memory Stage Block



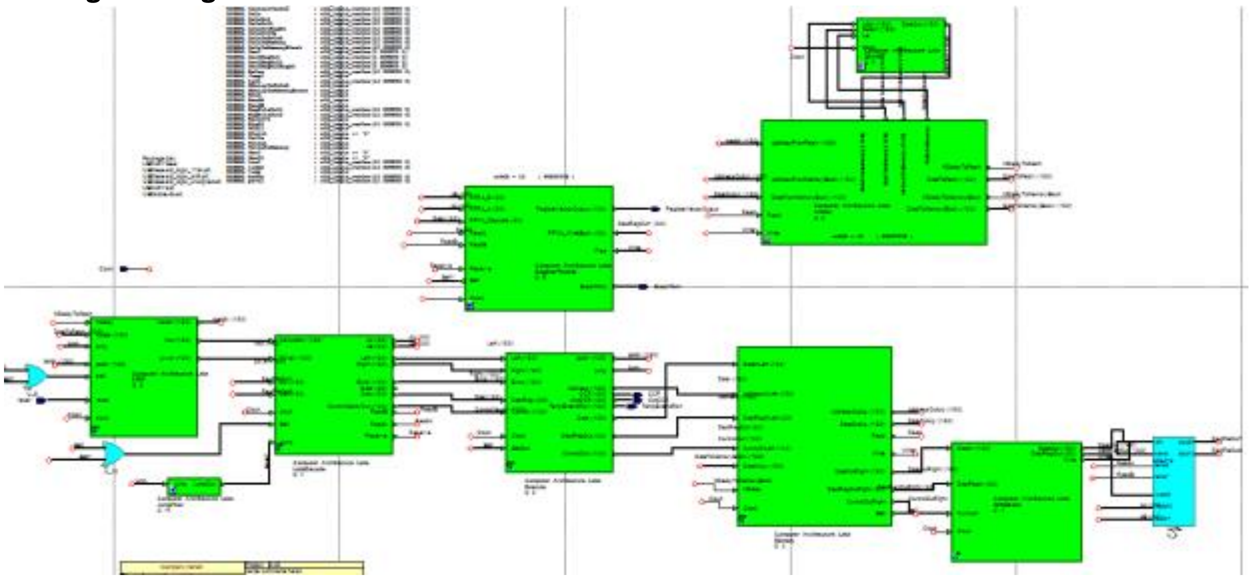
In this stage, almost everything is simply pipelined through. The Data coming out however must be chosen between the data coming from execute or from memory. If it is a ‘Load’ instruction then the Data from memory would be pipelined through, otherwise its always from execute.

Stage 5: Write Back



The write back stage simply pipelines the data, destination register, and the write control bit through to the register file.

Putting it all together:



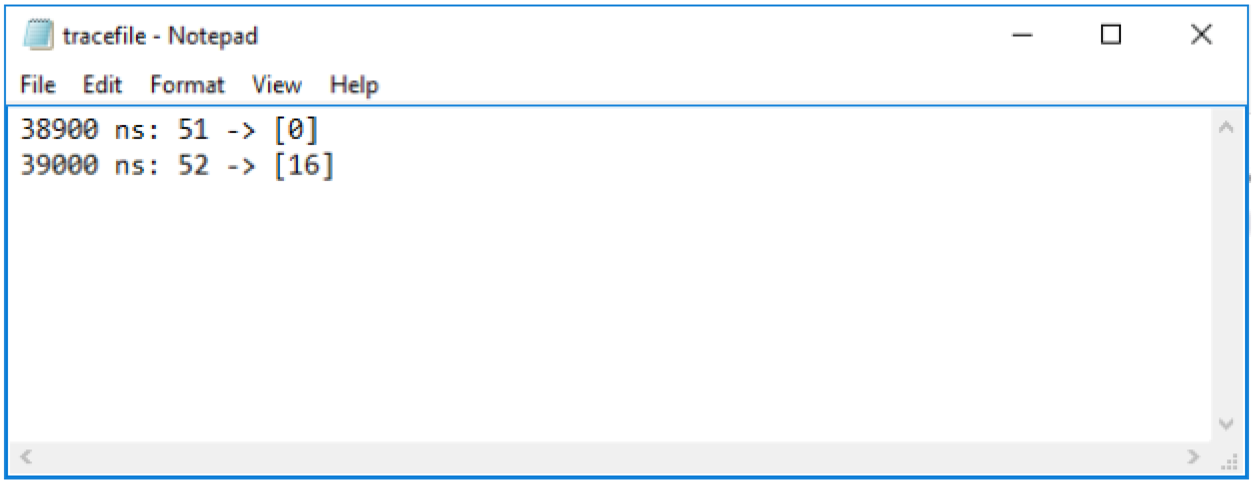
The five stages were then connected together, along with an Arbiter (big block on top), Register Tracker (to the left of the arbiter), and the SRAM (small block on top of the Arbiter).

The arbiter serves as a mediator between the Memory Stage, Fetch Stage, and SRAM memory. The arbiter receives either write, read, or no command. If the command is to Write, then the arbiter will send the address and data it receives from the Memory stage to the SRAM, as well as sending back the data it receives from the SRAM to the Memory Stage. Most importantly, the arbiter will send MDelay to Fetch and send the 'Write' bit to SRAM. When the arbiter receives the 'Read' command instead, it will do the same thing, sending Mdelay to Fetch, but send a 0 instead for the 'Write' bit to SRAM. If neither read or write commands are received then the normal operation of not delaying fetch occurs. This arbiter mitigates the structural hazards that arises when there is only a single memory for both instructions and data. This arbiter was implemented programmatically using VHDL code in a combined entity.

The Register tracker keeps account of what registers are currently being reserved and if a reserve command is received from the Decode stage. The most important thing about this component is that it is clocked, and some operations occur synchronously and others asynchronously. Sending a stall command out when a Read command is received and the register is currently reserved occurs instantly (asynchronously) while freeing or reserving a register when the Free or Reserve command is received occurs on the next clock edge (synchronously). This register tracker prevents the occurrence of Read-After-Write data hazards.

Control Hazards were taken into account when putting the five stages together by having stall outputs from the memory stage be input into the previous stages.

Results:
To run the program, the memory files from collab were put into the work folder for the VHDL project and the entire processor was run through modelsim. The clock was set and the reset was forced high then low again. After running through the entire program, the tracefile was checked for the correct values. The first couple of runs wrote too many things on the tracefile which required modifying the register tracker over the course of a few days to fix. The next set of run thrus resulted in the wrong values being written into memory. This, as mentioned above, was due to the control vector being sent for No-Op being wrong, and so the Decode Stage and the ALU within the Execute stage were modified to correct it. In the end, the program worked flawlessly. Below is a screen shot of the resulting trace file (which will also be attached along in the collab submission).



```
38900 ns: 51 -> [0]
39000 ns: 52 -> [16]
```