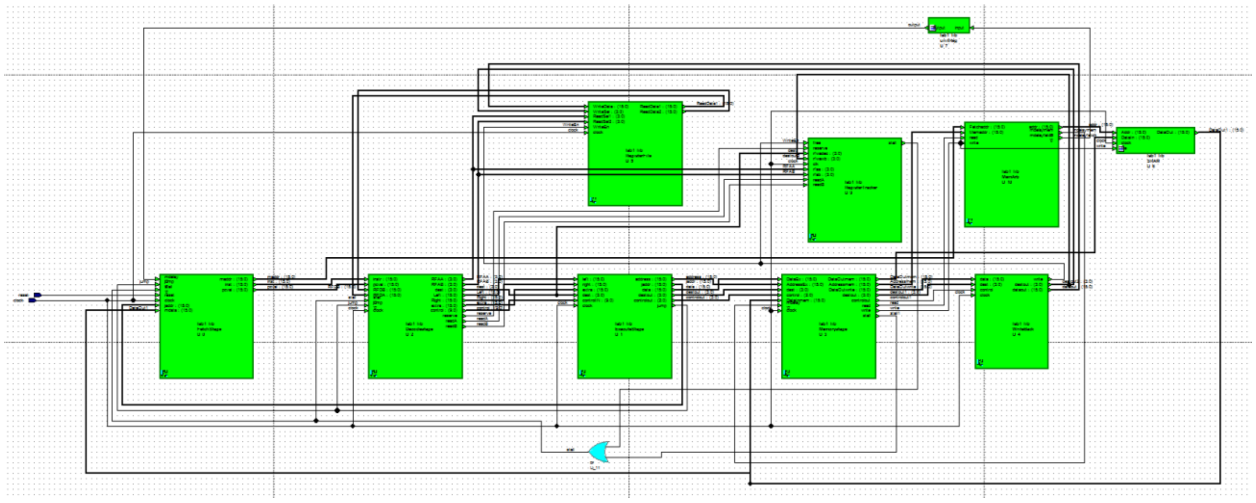
Phillip Seaton

Processor Report

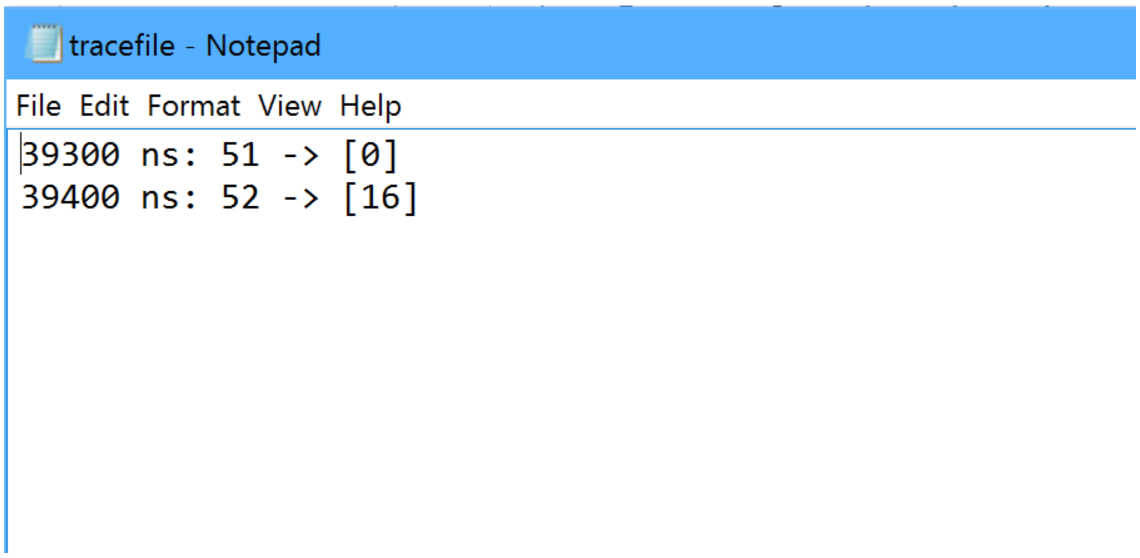
For the pipelined project micro architecture, I was able to complete the processor design and get the correct result in the trace file. I will first go through my design of the stages then problems I had implementing the processor. Most of my processor was made structurally with some small components made in code. Designing the fetch stage was based on getting the address from memory and figuring out what to do based on the different single bit input like stall and jump which generally sending a no op to the instruction. The fetch stage also dealt with incrementing the PC Value. The decode stage gets the operands ready for the execute stage. This could mean reading from the register file to get ready for an addition. The primary focus of the decode stage is to break apart an instruction from the fetch stage and prepare it for the execute stage. The instructions were split up by opcode, destination, source register or various other elements. The decode stage also determines the control vector for the execute stage.

The execute stage does the computations for the processor. This includes arithmetic, logic, and shifting of data. This also deals with the jump destination addresses and branch conditions. I made an ALU in the execute stage to do all the computations within the execute stage. This also set the control bits. The memory stage dealt with passing data to and from the memory to pull certain values from memory. The write back stage was primarily made to be able to write to the register file as the fifth stage of the processor. The next step in the processor was setting up for data hazards with the register file and memory. The only data hazards are read after write hazards. The memory arbiter was the easier of these two steps. If the memory stage was trying to read or write, then the fetch stage would get mdelay. The memory stage gets priority for accessing the memory. For the register tracker, the write back stage get priority. I made a single bit register that makes Q equal to one when the decode register knows the instruction needs to write several clock cycles later in the write back stage. Then the register is freed after it writes. Stall is asserted when the reserve is asserted and decode is also trying to read. This stall is sent to the fetch and decode stages. This system allows for proper handling of read after write hazards.

There were several problems that I came across when building the processor. The fetch stage needed reset to be in a state machine based on clock or it would change asynchronously. The decode stage needed a generate bits component whether it need to reserve, readA or readB for the register tracker. In the execute stage I had several problems when setting the conz bits that were resolved later in debugging of the final processor. The memory stage, write back, and memory arbiter had no problems. The register file proved to be quite a challenge. I tried it behaviorally and later found it was much easier to implement with structurally. When initially debugging I got infinite stall because the priority between free and reserve was flipped. Also some of the input single bit values needed to be initialized to zero on the fetch, decode and arbiter stages. My programs was not fetching instructions without these initializations. I found this project to be very useful in understanding the how a pipeline processor is designed and what problems typically arise when designing a processor.



Overall Processor



```
39300 ns: 51 -> [0]
39400 ns: 52 -> [16]
```

Tracefile