**EE480 Implementer’s Notes: Assignment 3**

**“Saghmo' loj rIntaH chenmoH!”**

Ethan Gibson

University of Kentucky  
Lexington, KY 40506

ethan.gibson@uky.edu

Coleman Platt

University of Kentucky  
Lexington, KY 40506

mcpl222@g.uky.edu

Madison Tolson

University of Kentucky  
Lexington, KY 40506

Madison.tolson@uky.edu

**ABSTRACT**

This paper presents the implementation details behind the pipelined version of the IoQ Don processor for assignment “Saghmo' loj rIntaH chenmoH!”. The previous project involved a multi-cycle implementation of the IoQ Don instruction set, but this project involved amending this into a faster, pipelined, single-cycle processor.

In addition to simply making a faster processor, this project introduced new problems that come alongside pipelining. One major issue addressed was how to handle various dependencies and anti-dependencies between instructions. Solutions considered involved forwarding and hardware interlocks, and ultimately forwarding was chosen. Additional considerations made for the pipelined edition included setting appropriate pipeline stages.

# IMPLEMENTATION

For this implementation, initial decisions included deciding on opcodes and previous implementations to base the pipelined processor off of. The opcodes decided were from the previous multi-cycle implementation of the processor and are defined at the top of the Verilog file.

Other considerations included which modules and stages to include. Separate ALU and processor modules were initially decided, and then later a separate decode module was added as well.

Different stages within the pipeline were defined using different “always” blocks, including stages such as fetch instruction, reading the instruction, sending it to the ALU, and writing the result to a register.

# 1.2 ALU

Since the ALU was now freed of its timing responsibilities (these now were exclusively determined within the processor module), the clock from the previous implementation was removed from the ALU module. This ALU module does, however, accept two inputs on which arithmetic operations are performed as well as a 4-bit wide OPcode input and an output data bus.

# 1.3 DECODE

The decode module operates by accepting the instruction as input and then having output wires for the opcodes, destination, source, and arguments. This module checks, via if statements, if the extended opcode fields are applicable and decodes the instructions for those requiring extended fields as applicable. Otherwise, the bits for the instructions are assigned to destination, source, and argument when it’s appropriate.

# 1.4 PROCESSOR

The processor underwent significant changes in order to accommodate a functioning pipeline. Many buffer registers were instantiated in order to support multiple instructions being processed and executed. This support involved created flag fields in order to support forwarding, which is acting as the selected defense against dependency issues (chosen over hardware interlock). This module also includes the aforementioned “always” blocks which act as the stages in the pipeline. In addition to those previously mentioned, there is also a “reset” included, a block to set flags for forwarding, and a block for getting the next value of the program counter.

# KNOWN DEFECTS

The processor compiles successfully. We were unsure of how to test it for single cycle operation, but the compiler did generate an output.

# REFERENCES

1. Dr. Hank Dietz. Home Page of Fall 2016 EE480. <http://aggregate.org/ee480/>
2. Shalom Greene