# Processor Defense: Technical Report

## Duke Community Standard

By submitting this LaTeX document, I affirm that

1. I understand that each `git` commit I create in the processor repository is a submission.
2. I affirm that each submission complies with the Duke Community Standard and the guidelines set forth for this assignment.
3. I further acknowledge that any content not included in this commit under the version control system cannot be considered as a part of my submission.
4. Finally, I understand that a submission is considered submitted when it has been received by the server.

# Introduction

# Subcomponents

## Regfile

The regfile was set to be negative-edge triggered to save one cycle for pipelining and data hazard protection. The regfile includes 32 32-bit registers.

## ALU

The ALU, consisted of a 32-bit CLA, 32-bit SLL, 32-bit SRA, 32-bit AND module, and a 32-bit OR module. Tristate buffers were used to output the correct data using the ALU opcode.

## Mult/Div

Unfortunately, the mult/div subcomponents were not fully implemented into the processor due to time constraints of the project itself and complexity with respect to the pipelining of the processor.

## Memory elements

D flip flops were used within registers. An SR latch was used mult/div to perform the respective operation based on the specified multiplier/divider control bits.

# 1 Full Processor

## Design implementation details of each stage (F, D, X, M, W)

The implementation of the pipelined processor was based off lecture slides.

The Fetch stage retrieves the current instruction and PC address bits from the PC latch and Instruction Memory module.

The Decode stage stores the current instruction and PC address bits from the F latch. The Decode stage also controls the outputs of the read ports of the regfile.

The Execute stage performs ALU operations and branches/jumps. The Execute stage receives current instruction and PC bits from the D latch. It also controls the input address to the PC.

The Memory stage stores and loads bits to/from the Data Memory module.

The Write stage controls the destination port of the regfile and the write enable.

## Handling hazards

The processor handles data hazards (read after writes) by using stalls and no ops. It also handles control hazards (branches/jumps) by flushing stages when a branch is taken.

## How fast the processor can be clocked

The processor can be clocked at 50 MHz.

## Bypassing logic

Due to time constraints and priority over correctness of computations, bypassing was not implemented. Bypassing was attempted near the end of the project timeline, but due to fear of producing incorrect data, bypassing was left out of the processor.

### Efficiency details

The processor takes priority of correctness over efficiency, so the processor safely stalls instructions with no ops in order to avoid data hazards. Since no bypassing was implemented the processor's efficiency is at least much better than a nonpipelined processor.

## 2 Instruction Implementation

Include details about how you implemented each instruction.
R instructions were implemented using the ALU and register read ports and performing operations within the ALU.

I instructions were more complex. $addi$, $bne$, $blt$ required modification of the ALU opcode since I instructions do not have an ALU opcode within the ISA. Because value comparisons are made within the ALU, multiplexers were used to set the ALU opcode.
JI instructions were computed in the execution stage, which controls the input to the PC latch. For $jal$, the ctrl$_{w}riteRegbitwassettor31duetoinstructionrequirements.$
JII instructions were implemented very similarly to JI instructions. The only difference was that $rd$ was the only register read from the regfile. The data from the register was multiplexed within the Execute stage in order to set the PC address equal to the data within this register, since Execute stage controls the input to the PC (i.e. it controls the current Instruction Memory address).

# 3 Process

#### Why did you choose your implementation?

This implementation was chosen due to simplicity of design and thoroughness of details given in lecture slides.

### How did you test your implementation?

A Python script was used to generate test benches and convert MIPS to machine code.

#### What errors exist in your processor?

Include details on what these issues are, why they occur, how you attempted to fix them, how you tested your processor to find these issues, and how these issues can be fixed.
The processor doesn't support mult/div due to time constraints and extreme clock cycle management. This error can be fixed given more time to implement the component.
Another error is detecting data hazards for register $rstatus$. Data hazards are checked for instructions $bex$ and $setx$ but not for others due to time constraints. However, data hazards are checked for register $r31$.

### Challenges - what were the main challenges you faced and how did you overcome them?

The main challenge faced was debugging the pipelined processor. Much time was devoted to building python script that could create testbenches by converting MIPS to machine code.

### Main learning points - what did you learn by making this processor?

I learned about the several possible data hazards that can occur in a pipelined processor. I also learned briefly why negative edge triggering a regfile can help save one cycle per instruction and for data hazards (inserting no ops).

## Conclusion