Cool Processor

Team Violet

Ian Barthel, Jordan Ansari, Hunter Masur, Logan Bryant

Table of Contents

[Introduction 3](#_Toc72697695)

[Overview 3](#_Toc72697696)

[Instruction Set 3](#_Toc72697697)

[Implementation 4](#_Toc72697698)

[Final Datapath 5](#_Toc72697699)

[Final State Diagram 5](#_Toc72697700)

[Testing Methodology 5](#_Toc72697701)

[Unique Features 6](#_Toc72697702)

[Extra Features 6](#_Toc72697703)

[Conclusion 6](#_Toc72697704)

# Introduction

The Cool Processor is an accumulator, but not just any accumulator. Our processor has three accumulators, which are able to be switched between with the use of a simple command. The ability to have multiple accumulators allows for code to be written that bypasses the need to frequently store or load from the memory file. This allows our processor to run at much quicker speeds compared to normal accumulators. As will be discussed in further detail later, we also have many more unique ideas linked to our processor. These include different length opcodes, two separate register files, and thw multiple accumulators.

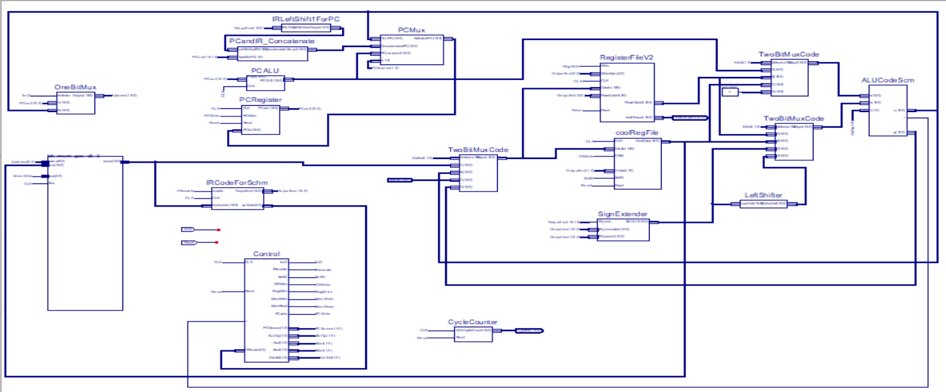
# Overview

## Instruction Set

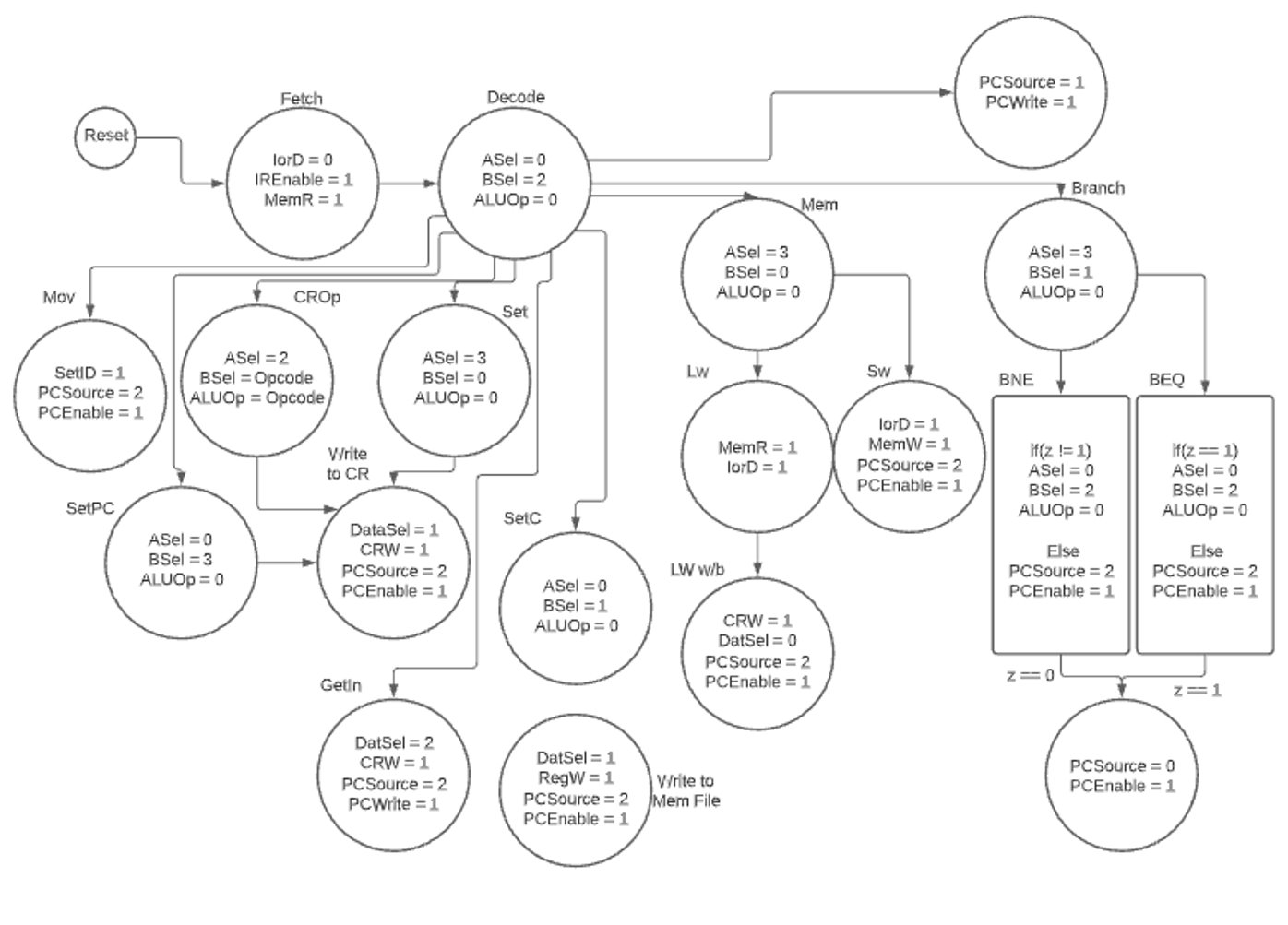
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instruction name | Syntax | Behavior | Semantics | Addressing Type | OpCode |
| Mov | Mov $cr\_ | $cr = $cr(n) | Switch current accumulator register to $\_ | Psuedo Direct | 00000 |
| Set | Set $\_ | $cr = $\_ | Set the value of the current accumulator register to $\_ | Direct | 00001 |
| Setc | Setc $\_ | $\_ = $cr | Set the value of $\_ to the current accumulator’s value | Direct | 00010 |
| Sw | Sw $\_ | Mem[$\_] = $cr | Store $cr at address $\_ | Offset | 00011 |
| Lw | Lw $\_ | $cr = Mem[$\_] | Load the value from $\_ into $cr | Offset | 00100 |
| SetPC | SetPC n | $cr = PC + 2 + n | Store $PC+2 + n shifted left by 1 (usually will be 0) | Direct | 00101 |
| GetIn | GetIn $\_ | $\_ = input | Load the outside input into $\_ (selected register in register file) | Direct | 00110 |
| J | J LABEL | $pc = JumpAddr | Jump to the label in the instructions | Psuedo Direct | 100 |
| Beq | Beq LABEL | if($cr == 0) then $pc = pc + 2 + BranchAddr | Jump to the label in instructions if $cr is equal to n | PC relative | 110 |
| Bne | Bne LABEL | if($cr != 0) then $pc = pc + 2 + BranchAddr | Jump to the label in instructions if $cr is not equal to n | PC relative | 111 |
| Sub (R) | Sub $\_ | $cr = $cr – $\_ | Subtract register data from $cr | Direct | 01001 |
| Add(R) | Add $\_ | $cr = $cr + $\_ | Add register to $cr | Direct | 01010 |
| Addi (I) | Addi n | $cr = $cr + immediate | Add the immediate n to $cr | Immediate | 01011 |
| Slt | Slt n | Slt: $cr > $\_ ? $cr = 1 if true, = 0 false | Set $cr to 0 if $cr is less than n, or 1 if it is greater than n | Direct | 01110 |
| Sll (R) | Sll n | Sll: $cr = $cr shifted “left” by immediate (negative values shift right) | Shift the bits of $cr to the left by n | Immediate | 01111 |

# Implementation

## Final Datapath



## Final State Diagram



## Testing Methodology

We implemented bottom-up testing to build up the data path.

Phase 1: We tested each individual component by writing loops that would test every combination of signals and comparing their outputs to the expected values. These loops were designed to test arbitrary inputs as well as edge cases. Some tests do not pass despite working correctly for the sake of simplistic testing, but they are verified by the tester to have the correct outputs. For example, testing changing each register to a value can be done easily in a loop, but the 0 register will fail because it cannot be changed. This behavior is intentional for edge cases as they can be manually verified by looking at the inputs and outputs without having to write code specifically for those cases.

Phase 2: We combined the components in to 3 separate subcomponents. The first was the PC with its multiplexer, concatenator, shifter etc. This was tested by inputting the starting PC and the intended signals and trying each of the functions, ensure that they all worked individually and that they could be used consecutively over multiple cycles. The second was the registers (both cool and regular) with their multiplexers feeding in to the ALU and the ALU itself. This was tested by placing values in the registers and trying different combinations of registers and constants with different operations in the ALU and comparing them to the expected output. The third was the IR and the control to ensure that the control would set the proper signals when given an op code.

Phase 3: We combined the subcomponents to have the data path complete except for access to memory. We manually input instructions and used the simulation model in Xilinx to verify that the correct values were being stored. We tested one of each instruction individually and tested them all consecutively to confirm that they worked correctly.

Phase 4: We added the memory unit which completed the data path and tested running instructions from memory and verified that they correctly by looking at the data in the simulation model in Xilinx.

Phase 5: We put Euclid’s algorithm in memory and ran it from start to finish with different values.

# Unique Features

Our processor has three big unique features; two register files, three accumulators, and different sized opcodes.

### Two Register Files

The idea behind the two register files is that it allows for the accumulator and a different register to be put into the ALU at the same time, which allows for easier calculations.

### Three Accumulators

Using three accumulators allows for fewer memory accesses and results in a much faster runtime.

### Different Sized Opcodes

We have different sized opcodes depending on if it is an R-type or a jump instruction. An R-type has an opcode length of 5 bits, while a J-Type only has 3 bits. A J-Type always starts with 1, while an R-Type always starts with 0. This was done so that J-Types have extra room in the instruction to make larger jumps, which is extremely useful for programmers wanting to make a very large assembly code set.

# Extra Features

We added an assembler in Java that takes a .txt file with assembly code and generates a .txt file with the equivalent machine code. We slightly changed our implementation from when we made the assembler so beq and bne instructions need to be done manually but the rest of the instructions work.

Ex input:

RELPRIME: mov $cr1

set $0

addi 2

setc $v2

setc $a2

Ex output:

0000000000000000

0000100000000000

0101100000000010

0001000000001001

0001000000000111

# Conclusion

For the most part, our team designed and assembled the processor without a hitch. One big issue we had was the assembler and the assembly code itself. We quickly made these when we initially started the project and failed to update it as time went on and we changed how instructions worked. Other than that, we did not have much of an issue. Our instruction set is wide enough that it can be used to do a very large set of algorithms and instructions, along with the instructions themselves being very straightforward and easy to understand.

# Test Results

The following data is for running the RelPrime method for a value of 5040:

* No. of Bytes required: 106
* No. of instructions: 122,484
* No. of cycles: 469,460
* Average CPI: 3.83
* Cycle Time: 11.871ns = 84.24 MHz
* Execution Time = 5.573 ms

Table

Description automatically generated

# Appendix