Ouckah Processing Unit (OPU)

Aidan Ouckama

December 9th, 2023

Abstract

This document describes the capabilities of the `Ouckah Processing Unit` or the `OPU`. Within this instruction manual, you will learn about assembling instructions, architechure, and implementation for our astounding `Ouckah Processing Unit`. Have fun!

Contents

1 Introduction

1.1 Author, Building Process and Resolution

During the creation of this project – the author **Aidan Ouckama** – went through many *sleepless* nights and headaches, however, a lot was also learned during the process.

Author: The **contents** within this manual was written with the blood, sweat and tears of the author.

When reading through this manual, please take great care in the effort put in by our great author.

Building Process: The **process** of building this CPU – or OPU – was a great challenge. However, going through the motions of making an ALU, Register File, and even an entire assembler was a great experience at the end of the day.

Within the layout of the CPU are circuit components, such as the ALU, Register File, Control Unit, and Data Memory Unit. Along with all of this is a custom-made assembler which converts `ouck` code into image files for the OPU to read from.

Resolution: Overall, this project was a great learning experience and I'm very happy with the knowledge I now know in regards to CPU and computer architecture.

I loved showing off the project as I was building it to friends and family, even if they didn't completely understand what was going on (me neither), and I will definitely be tweaking little things, maybe even try a full on 32-bit OPU.

2 Layout & Design

2.1 Components

The Ouckah Processing Unit is composed of a multitude of different components, including the **Register File**, **ALU**, and **Control Unit**. In this section, the different components and their uses will be discussed, along with the architecture of each component. You may see how each component is layed out in `Figure 1`.

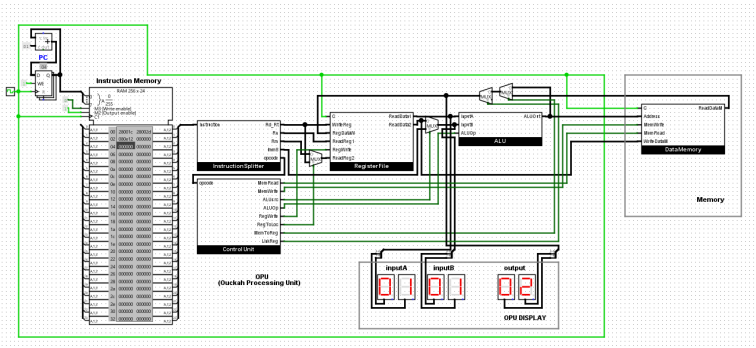

Figure 1: An image of the OPU architecture

2.2 Register File

The `register file` is a component within the OPU that houses the registers which store data for quick reading and writing. In the OPU, there are 3 registers, three of which are read and write, and one register, the **OZR** register, which is read-only.

Usage : The registers within the register file are used to quickly store and load data. It is housed near the `ALU` for fast access, in comparison to retrieving data from memory.

Registers : The registers within the register file are named **O0**, **O1**, **O2**, and **O3** or **OZR**. As stated before, registers O0-O2 are read and write accessible while O3 or OZR is read-only, and permanently stores the value 0.

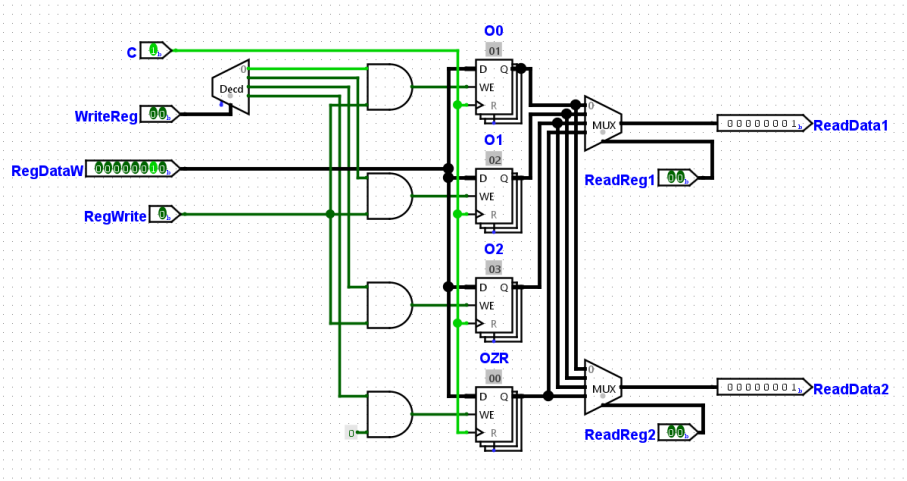

Figure 2: An image of the register file architecture

→ Section 3

Section 3 describes how to utilize the registers in `ouckahASM`.

2.3 ALU

The `ALU` or the *Arithmetic Logic Unit* does the computations for instructions, including arithmetic operations for instructions like `ADD` and `SUB`, and calculating offsets for instructions like `LDR` and `STR`.

Usage : The ALU is used to do computational services for the OPU. It is located near the `register file` for quick access to the data that the operations are being executed on.

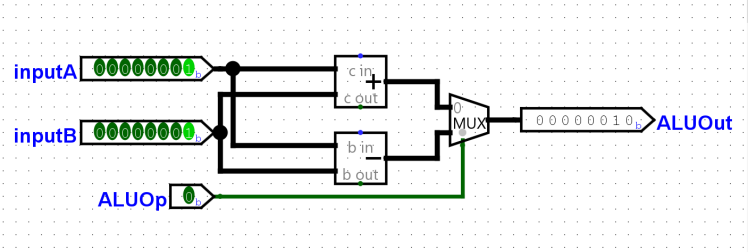

Figure 3: An image of the ALU architecture

Operations : The operations that ALU can execute are addition and subtraction. These two basic operations can do a multitude of things within the OPU, including basic math and offset calculations.

→ Section 3 Section 3 how to utilize the ALU in `ouckahASM`

2.4 Control Unit

The `control unit` is a component within the OPU that translates the opcode [1] into a multitude of outputs utilized by the OPU architecture.

Usage : The control unit is used to translate the opcode of an instruction into outputs that the OPU can use. The outputs translated by the control unit include: `MemRead`, `MemWrite`, `ALUsrc`, `ALUOp`, `RegWrite`, `RegToLoc`, `MemToReg`, and `LinkReg`. The uses of these outputs are listed below.

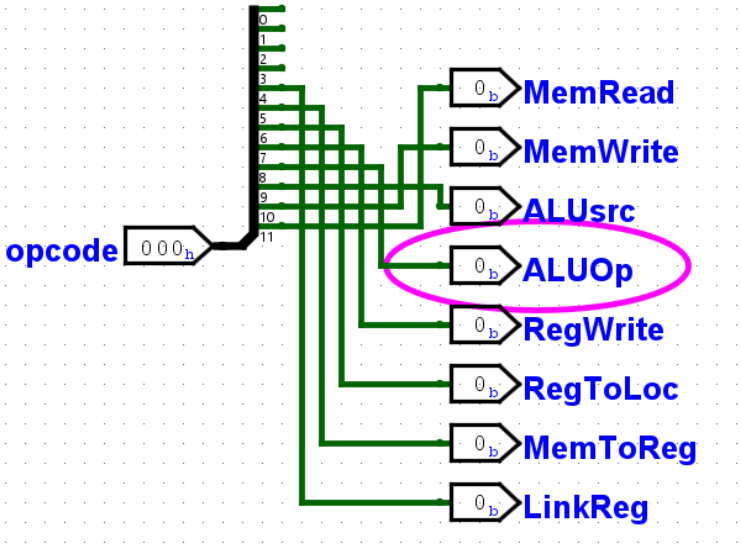

Figure 4: An image of the Control Unit architecture

Outputs : The ouputs translated by the control unit have the following usages:

- **MemRead:** the instruction reads data from memory.
- **MemWrite:** the instruction writes data to memory.
- **ALUsrc:** the instruction uses an immmediate 8-bit value instead of a register.
- **ALUOp:** the opcode used to determine what arithmetic operation is executed. Addition and subtraction are represented by 0 and 1 respectively.
- **RegWrite:** the instruction writes data to a register.
- **RegToLoc:** the instruction uses another register for calculations instead of the destination register.
- **MemToReg:** the instruction sends data to the register file to write from memory.

[1] A binary encoding of an instruction which tells the OPU what the instruction does, e.g. whether or not the instruction utilizes the `ALU`.

- **LinkReg:** the instruction writes data to the program counter.

3 Assembling Instructions

3.1 Using the Assembler

ouckahASM: With the OPU architecture, the instructions **MUST** be in a `.ouck` file in order for the assembler to read it. This verifies that the code within the file is `ouckahASM`, or ouckah assembly.

The language is very similar to ARM, with minor keyword changes. ouckahASM takes the following instructions:

- **RIZZ Rd, Rn, Rm:** rizzes the values stored in Rn and Rm together and stores the sum in Rd.
- **RIZZ Rd, Rn, imm8:** rizzes the value stored in Rn and an inputted immediate 8-bit value together and stores the sum in Rd.
- **FTAX Rd, Rn, Rm:** fanum taxes the value of Rm from Rn and stores the difference in Rd.
- **FTAX Rd, Rn, imm8:** fanum taxes the value of and immediate 8-bit value from Rn and stores the difference in Rd.
- **GYAT Rd, [Rn, Rm]** gyets the value at address Rn with offset Rm from memory and stores it in Rd.
- **GYAT Rd, [Rn, imm8]:** gyets the value at address Rn with offset of an immediate 8-bit value from memory and stores it in Rd.
- **YEET Rd, [Rn, Rm]:** yeets the value stored in Rd and stores it in memory at the address Rn with an offset Rm.
- **YEET Rd, [Rn, imm8]:** yeets the value stored in Rd and stores it in memory at the address Rn with an offset of an immediate 8-bit value.
- **MOVE Rd, Rm:** stores the value from Rm into Rd.
- **MOVE Rd, imm8:** stores an immediate 8-bit value into Rd.
- **NOPE** no operation

The logical design makes it easier for the author to write consistent code.

→ Section 3.4 Section 3.4 examples of `ouckahASM`

3.2 Image Files

Once you have created your code you must **assemble** it into image files. To do this do the following in your terminal:

```
python assembler.py <filename>
```

After the code has been assembled, two `image files` should be produced, a **text** image file and a **data** image file.

- The **text** image file stores the hexadecimal representation of the instructions within your ouckahASM code.
- The **data** image file stores the hexadecimal representaion of the data declared in the `.data` section of your ouckahASM code.

To use these image files, they must be stored in their respective RAM, text going into the instruction RAM and data going into the data RAM. View the figures below to see how this is done.

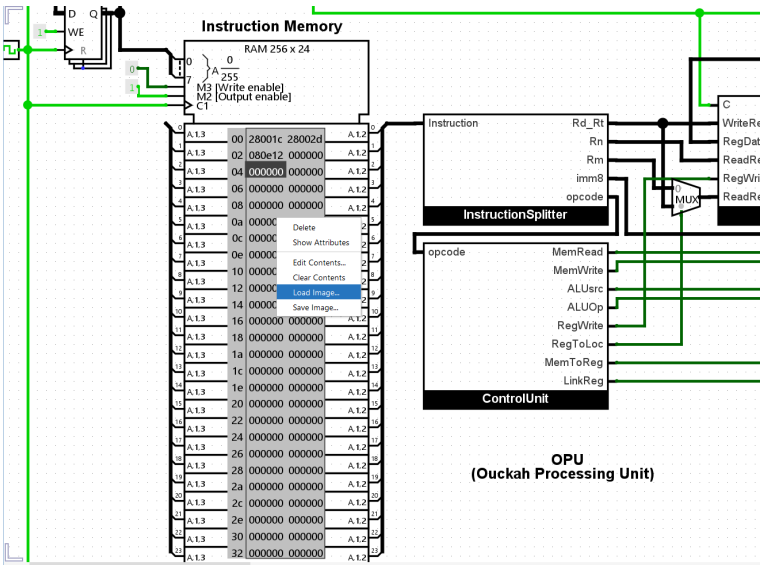

Figure 5: Loading an image into the instruction RAM

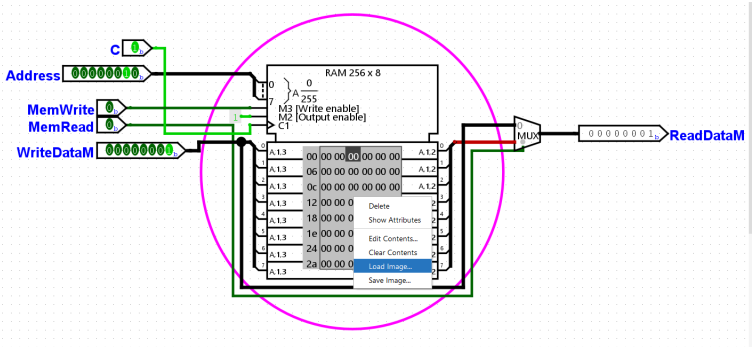

Figure 6: Loading an image into the data RAM

3.3 Instructions & Decoding

The OPU handles instructions in 24-bits, in the following order:

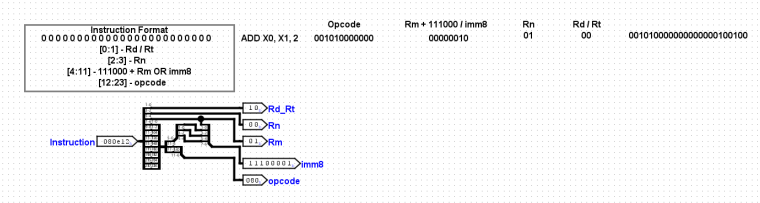

Figure 7: The distribution of bits of each instruction

The different bits and their uses are described in section 2.4.

3.4 Examples & Use Cases

Here are some examples of ouckahASM to help you while you make your own programs!

```
source.ouck
1   .data
2       a: 1
3       b: 2
4   .text
5       MOVE 00, a
6       MOVE 01, b
7       RIZZ 02, 00, 01
8
```

Figure 8: Adding two numbers

source.ouck

```
1   .data
2       a: 4
3       b: 3
4   .text
5       MOVE O0, a
6       MOVE O1, b
7       FTAX O2, O0, O1
8
```

Figure 9: Subtracting two numbers

source.ouck

```
1   .data
2       a: 4
3   .text
4       MOVE O0, a
5       MOVE O1, 0
6       GYAT O0, [O1, 0]
7       YEET O2, [O1, 0]
8
```

Figure 10: Loading and storing