

Todd Harlow | ECE 586 | June 5, 2016

A Simple Pipelined Neuromorphic Processor

Final Project Report

Table of Contents

[Introduction 1](#_Toc452847765)

[Motivation 1](#_Toc452847766)

[Problem Description 1](#_Toc452847767)

[Constraints Given 1](#_Toc452847768)

[Solution Description 1](#_Toc452847769)

[Overview 1](#_Toc452847770)

[Architecture 1](#_Toc452847771)

[Registers 1](#_Toc452847772)

[Data Type 2](#_Toc452847773)

[Addressing Modes 2](#_Toc452847774)

[Instruction Formats 2](#_Toc452847775)

[Operations 3](#_Toc452847776)

[Pipeline 4](#_Toc452847777)

[Diagram 4](#_Toc452847778)

[Stages 4](#_Toc452847779)

[Hazards 5](#_Toc452847780)

[Design Justifications 6](#_Toc452847781)

[Instructions 6](#_Toc452847782)

[Stage Count 6](#_Toc452847783)

[Hazard Avoidance 6](#_Toc452847784)

[Performance Report 6](#_Toc452847785)

[Summary 6](#_Toc452847786)

[Strengths 6](#_Toc452847787)

[Weaknesses 6](#_Toc452847788)

[Graduate Assignment 6](#_Toc452847789)

# Introduction

## Motivation

The purpose of this project was to gain a deeper understanding of pipelined microarchitectures.

## Problem Description

The processor is to

### Constraints Given

* All memory accesses take 1 cycle to complete
* The processor must be pipelined
* No superscalar or multicore solutions
* Throughput is the ultimate design goal

# Solution Description

## Overview

The processor designed to meet the challenges of the project is a 4-stage, RISC type processor.

## Architecture

### Registers

All registers are 32-bit.

Instruction Registers

1. Program Counter Register (PC) – Contains the memory address of the current instruction
2. Instruction Register (IR) – Contains the current instruction

Data Registers

1. Input Pointer Register (rIP) – Contains the memory address of the current input
2. Weight Pointer Register (rWP) – Contains the memory address of the current weight
3. Output Pointer Register (rOP) – Contains the memory address of the current output
4. Input Register (rI) – Contains the current input word
5. Weight Register (rW) – Contains the current weight word
6. Output Register (rO) – Contains the final result word of input/weight calculations
7. Accumulator Register (rA)– Contains the current sum of the input/weight calculations

### Data Type

All data is stored as bytes in 32-bit words, so each word contains either four neuron states or four weights. The bytes are stored in a big-endian manner. This is the only data type used by the processor.

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 0 | Byte 1 | Byte 2 | Byte 3 |
| State/Weight | State/Weight | State/Weight | State/Weight |

### Addressing Modes

Two addressing modes are employed by the processor:

1. Immediate – This is used by the pointer load operations to place memory addresses directly into the pointer registers.
2. Autoincrement Register Indirect – All other load and store operations make use of the pointer registers and increment the respective pointer by four after loading or storing a word.

### Instruction Formats

There are three types of instructions for the processor:

1. Pointer Load (P-type)

|  |  |
| --- | --- |
| 4 | 28 |
| Opcode | Immediate |

This type of instruction is used for the pointer load operations. The immediate value is a memory address that will be loaded into a pointer register determined from the opcode.

1. Memory (M-type)

|  |  |
| --- | --- |
| 4 | 28 |
| Opcode | Unused |

This type of instruction is used for loading and storing between registers and memory. It is different from the P-type instruction because it does not take an immediate operand as its memory address. Instead, both the register and memory location required for the operation are implicit for each M-type instruction.

1. Regular (R-type)

|  |  |
| --- | --- |
| 4 | 28 |
| Opcode | Unused |

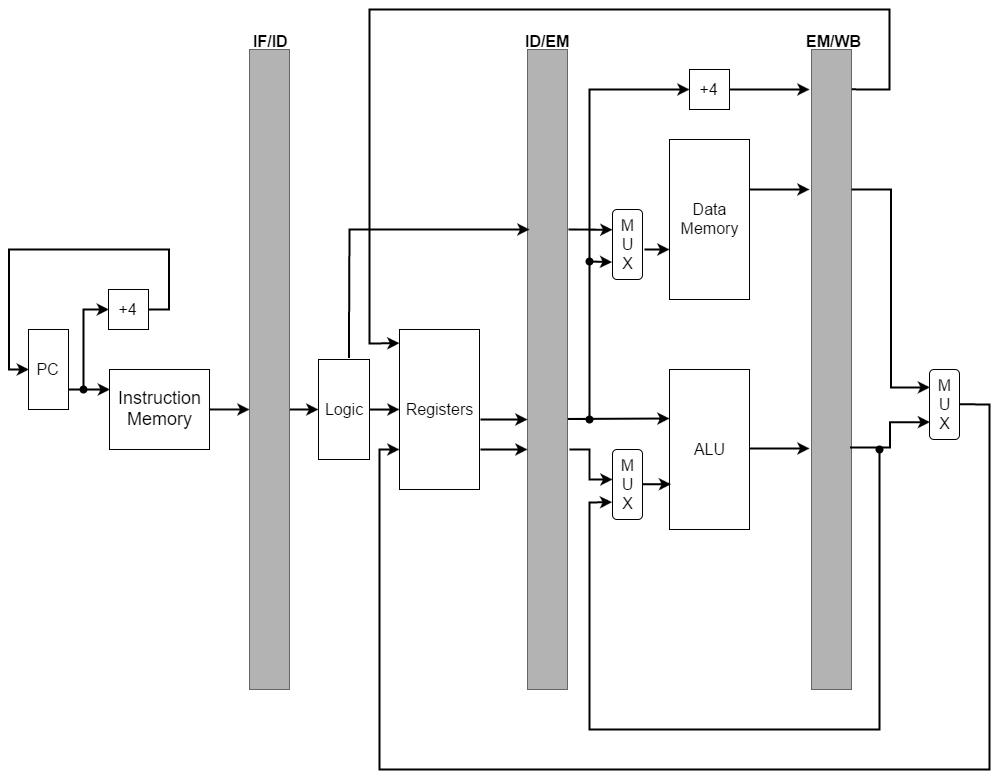
All of the other instructions for the processor are regular type and also only use the opcode portion of the instruction. Register usage is implicit just like M-type instructions.

### Operations

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Mnemonic | Opcode | Description |
| No Operation | nop | 0000 | Does nothing |
| Load Input Pointer | ldip | 0001 | Loads immediate value into rIP |
| Load Weight Pointer | ldwp | 0010 | Loads immediate value into rWP |
| Load Output Pointer | ldop | 0011 | Loads immediate value into rOP |
| Load Input | ldi | 1000 | Loads word at memory location pointed to by rIP into rI and increments rIP by 4 |
| Load Weight | ldw | 1001 | Loads word at memory location pointed to by rWP into rW and increments rWP by 4 |
| Store Output | sto | 1010 | Stores word in rO to memory location pointed to by rOP and increment rOP by 4 |
| Multiply-Sum-Add | msa | 1011 | Byte-wise multiples contents of rI and rW, sums results, places sum into rA |
| Threshold 0 | t0 | 1100 | Checks rA. If less than zero, writes a zero to Byte 0 of rO. Otherwise writes a one. |
| Threshold 1 | t1 | 1101 | Checks rA. If less than zero, writes a zero to Byte 1 of rO. Otherwise writes a one. |
| Threshold 2 | t2 | 1110 | Checks rA. If less than zero, writes a zero to Byte 2 of rO. Otherwise writes a one. |
| Threshold 3 | t3 | 1111 | Checks rA. If less than zero, writes a zero to Byte 3 of rO. Otherwise writes a one. |

## Pipeline

### Diagram



### Stages

This processor makes use of a 4-stage pipeline.

1. Instruction Fetch (IF)

Loads instruction at memory location pointed to by PC into IR

Increments PC by 4

1. Instruction Decode (ID)

*P-type instructions*

Extracts immediate value to be used for memory access

*M-type instructions*

Retrieves pointer from register to be used for memory access

*R-type*

Retrieves required values from registers into places them into ALU inputs

1. Execute / Memory (EM)

*P-type instructions*

Loads word from memory

*M-type instructions*

Loads/stores word to/from memory

*R-type instructions*

Executes ALU operation

Forwards result back to ALU input (to be used by operations needing forwarded result)

1. Write Back (WB)

*P-type instructions*

Writes word from memory into destination register

*M-type instructions*

Writes word from memory into destination register (if load)

*R-type instructions*

Writes results from ALU operation into destination register

### Hazards

*Data Hazards*

There are two kinds of data hazards that occur in this design:

1. *ALU operations immediately after a memory load.* These occur when a multiply-sum-add ALU operation immediately follows a memory load operation so the value needed from memory will not be ready in time. Because forwarding cannot be used in this circumstance (the value just isn’t there yet), a “no operation” is inserted between any M-type load operation and a R-type instruction.
2. *ALU operations require result from previous ALU operation.* These occur when a threshold operation immediately follows a multiply-sum-add operation. The threshold operation needs to read the accumulator register which the multiply-sum-add operation writes to. This is avoided, however, by forwarding the ALU result back to its own input. Because threshold operations will always follow multiply-sum-add operations in this design, logic in the ID stage will use this forwarded value when decoding a threshold operation.

*Structural Hazards*

There are no structural hazards to contend with because of the theoretical nature of this project. All circuitry is assumed to be sufficiently fast to complete its necessary functions within a clock cycle in order to keep the cycles required for each stage equal to one.

*Control Hazards*

There are no control hazards to contend with because there are no branches or jumps in this architecture.

## Design Justifications

### Instructions

### Stage Count

### Hazard Avoidance

# Performance Report

# Summary

## Strengths

## Weaknesses

# Graduate Assignment