A Simple Pipelined Neuromorphic Processor

FINAL PROJECT REPORT

Todd Harlow | ECE 586 | June 5, 2016

Table of Contents

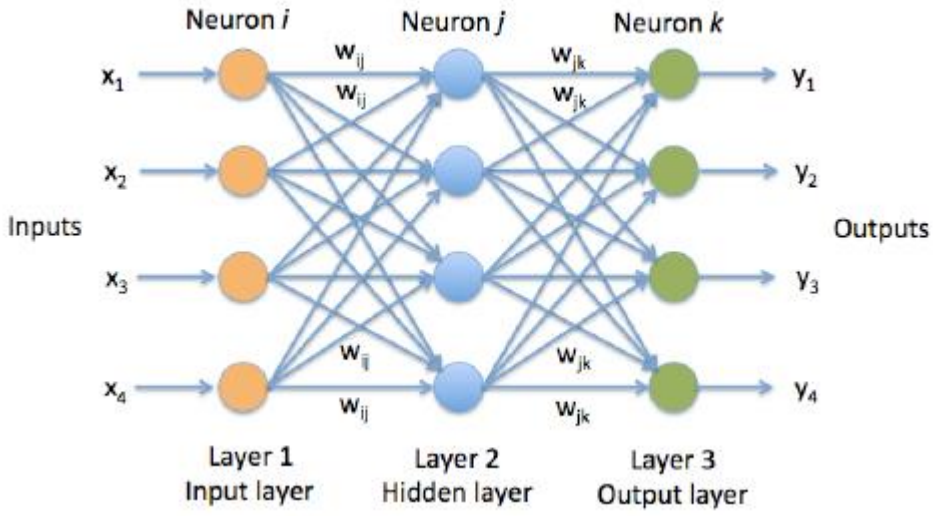
Introduction

MOTIVATION

The purpose of this project was to gain a deeper understanding of pipelined microarchitectures as well as learn more about neuromorphic architectures. The project required designing a processor from the ground up to meet the specific requirements. A high-level language (Python) was used to simulate the design.

PROBLEM DESCRIPTION

The processor needs to be specialized for processing a simple feed-forward neural network. The example given in the project document contains 4 input neurons, 4 hidden neurons, and 4 output neurons. All input neurons are connected to all hidden neurons and all hidden to output. The following diagram illustrates this network:



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 I designed to meet the challenges of the project is a 4-stage processor. Because of the extreme specificity of this project, many features found in a typical general purpose computer architecture were not needed. Instead, this design favors simplicity and focuses on the few instructions required to compute the problem neural network. There are load and store memory instructions, special ALU instructions for calculating the nodes of the network, and a "no-op" instruction which is used to avoid data hazards. The majority of the instructions do not take operands and instead operate on implicit registers. There is no branching or jumping to complicate the pipeline.

ARCHITECTURE

Registers

All registers are 32-bit.

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
   NOTE: rA is a special register that can only be accessed by the ALU

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. Auto-increment 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.

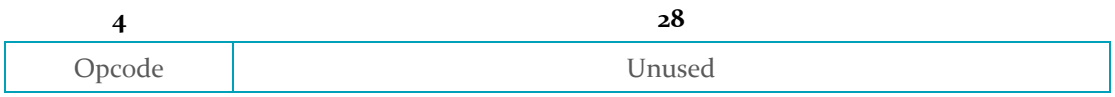
Instruction Formats

There are three types of instructions for the processor:

1. Pointer Load (P-type)

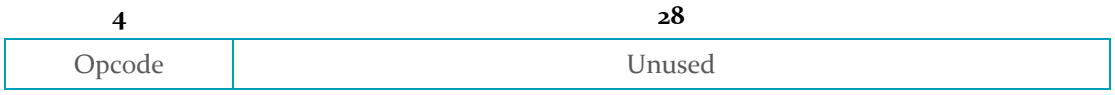
This type of instruction is used for the pointer load operations. The immediate value is a memory address that will be loaded into a pointer register determined from the opcode.

2. Memory (M-type)



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.

3. Regular (R-type)



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.

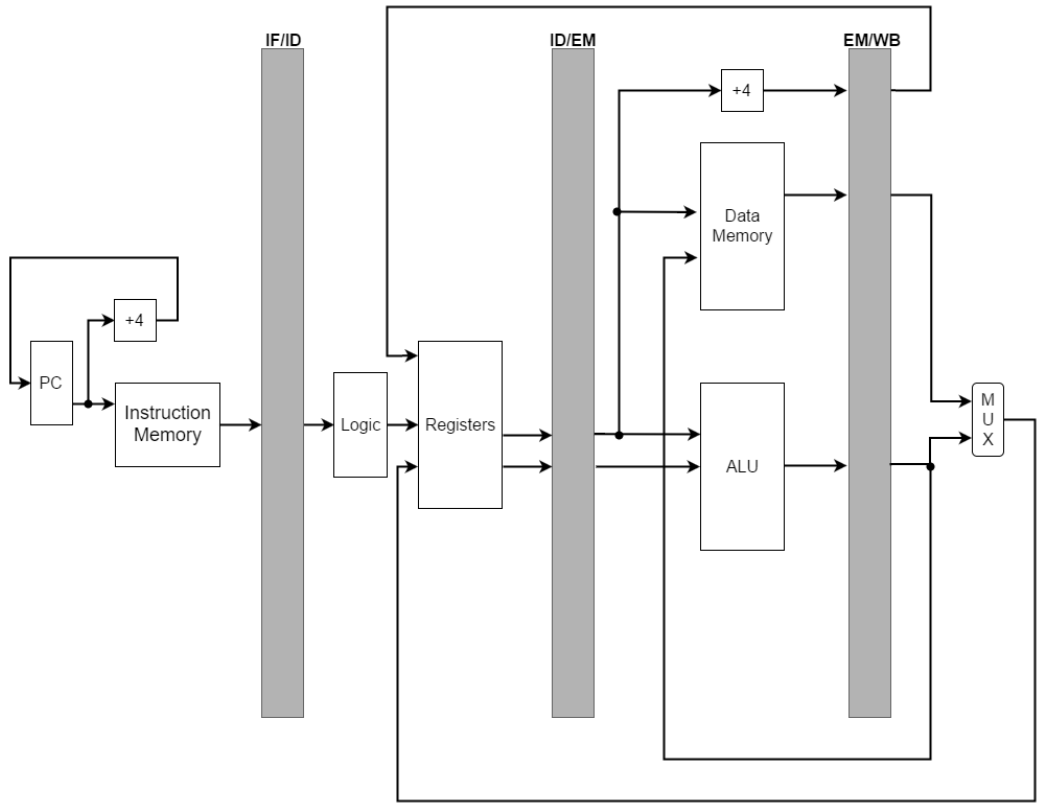
Operations

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

2. Instruction Decode (ID)
    P-type instructions
        Stores immediate value directly into register
    M-type instructions
        Retrieves pointer from register to be used for memory access
    R-type instructions
        Retrieves required values from registers and places them into ALU inputs

3. Execute / Memory (EM)
    P-type instructions
        Does nothing
    M-type instructions
        Loads/stores word to/from memory
    R-type instructions
        Executes ALU operation
        Stores result in special accumulator register rA (if multiply-sum-add)
4. Write Back (WB)
    P-type instructions
        Does nothing
    M-type instructions
        Writes word from memory into destination register (if load)
    R-type instructions
        Forwards ALU result to EM stage (if threshold)

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. *Memory store operation immediately after a threshold operation.* These occur when the memory store operation follows the fourth threshold operation because then the output is ready to be written. The solution to this hazard is forwarding the ALU output directly to the memory unit to be written. This is a simple, always-on forwarding that is possible because the only memory write operation for this processor uses this forwarded value as its implicit operand.

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

ALU

A key feature for this design is the special behavior of the ALU. First, it has a special accumulator register which is assumed to be very close to the ALU. Second, the ALU operates byte-wise on 32-bit word operands. Because this is assumed to be a highly customized processor, I think it is reasonable to assume such an ALU for my design. The main drawback for such a multi-operation ALU is the expected increase in propagation delay should it be implemented with real world circuitry. However, the multiply-sum-add operation is the only ALU operation that could foreseeable have this problem and this operation exclusively writes to the special accumulator register which is assumed to have fast access for the ALU. I felt that this fact justified using the ALU in my design.

Instructions

The majority of the instructions for this design use implicit operands. My main motivation behind this decision was the very specific nature of the processor. It is only meant to implement an algorithm to calculate the type of neural network described for this project. Because both the layout of data memory and the order of operations are known and expected (completely unlike a general purpose computer), I was able to eliminate most operands all together. The only operands used are the immediate values in pointer load instructions (P-type). Once these pointer values are in their respective registers the processor runs until the end of instruction memory. All memory accesses use these pointers and auto-increment them after use.

The major downside to this approach is wasted space. I only use 4 bits for my opcodes which leaves 28 unused bits for nearly all the instructions in memory. This means about 76% of instruction memory is wasted. I felt comfortable sacrificing some space in the name of simplicity, especially with the small 4-4-4 neural network. I would revisit this decision if the minimum size of the network were to increase.

Stage Count

At the beginning of the project, I tried to implement the processor with a MIPS-style 5-stage pipeline. However, after deciding on my algorithm and writing the assembly it became obvious that speedup gained by an extra stage would only be wasted in my design. This is because ideally my algorithm goes to memory every third instruction. With the 5-stage processor this meant inserting two no-ops after every load to prevent data hazards. That meant my speedy 3 instruction calculations would take 5 instructions, or 66% longer.

I then swung in the opposite direction and tried a 3-stage design. While this worked out well in my pipeline diagrams, further thinking revealed it to be an impossible design, even with the magic 1-cycle memory we were given to work with. I tried to cram memory operations and all other ALU operations into the execute stage. If we were to assume that the memory accesses alone take one cycle, then the execute stage would certainly take longer than one cycle to complete. This

condition would defeat the purpose of pipelining because the theoretical clock on the processor would have to be slowed to accommodate for the overweight execute stage.

So I settled on a 4-stage design. I still have to use no-ops after memory loads, but only one. I combined the execute and memory stages of the MIPS design because memory addresses are loaded directly from registers to the memory unit, unlike MIPS were they are calculated using the ALU. Additionally

The only data hazard that arises from this configuration is on a memory store operation. This is avoided by forwarding the ALU output back to memory unit input.

Simulation

The simulator created to test the processor was written in Python. There is a generator script and a simulator script. The generator prompts the user for the number of input, hidden, and output neurons as well as the number of simulations to run. Then all the necessary files are created in sub-directories. The simulator script can then be run and automatically processes the newly created files. The individual simulation results are recorded to new files and the total simulation performance is printed to the console upon completion. All memory files are UTF-8 text files with 32-bit hexadecimal strings on each line. All plain text files are UTF-8 as well.

INPUTS

- *Instruction Memory File*

  Contains the machine code instructions to be loaded and run by the processor simulator.

- *Assembly File*

  Contains the contents of the instruction memory file but in readable form.

- *Data Memory File*

  Contains the data of the neural network in the following order: Input-to-Hidden Weights, Hidden-to-Output Weights, Input States, Hidden States, and finally, Output States.

- *Calculated Output File*

  As the memory files are created, the generator also calculates the expected output of the network and writes that to this file.

OUTPUTS

- *Processor Trace File*

  Contains a trace of the processor after running a simulation

- *Simulated Output File*

Contains the simulated output of the network. This is compared to the calculated output to check for correctness.

RESULTS

Total Simulations	Total Cycles	Total Instructions	CPI
1000	43000	39000	1.10

Because this is a simulation with no real-world variability, the results are the same no matter the number of simulations. I cannot say how performance would differ for calculating one of the other network as I did not have time to run them.

Project Summary

STRENGTHS

Simplicity is this design's greatest feature. The algorithm implemented to process the network is simple and is automatic in nature. There is no branching to contend with so there is minimal hazard control. Additionally, the ALU is capable of processing 4 input/weight combinations at a time which means faster computation compared to any design operating on only 1 combination at a time. Because the ultimate goal of this project was throughput, I feel that this design is quite successful.

WEAKNESSES

This design does waste a large percentage of instruction memory. This is not only because of the unused bits in the instructions, but also the no-branching style means much of the instruction code is redundant. This was done to keep the project simple, but I there is much room for improvement in this regard. I think the processor as it currently exists is a great starting point and clever usage of branching could make this design even faster.