Viplava

(A neuromorphic processor)

A project report submitted in partial fulfillment of the requirement for ECE 586 computer architecture course

Submitted by

DHEERAJ CHAND V (947507669) VENKATA SAIMOHAN KAMMILA (983819264)

(TEAM: PATH FINDERS)

Objective: Our objective is to design, simulate and benchmark a pipelined neuromorphic processor for a fixed feed-forward threshold neural network. The network we implemented has 4 input neurons, and 4 output neurons. We are not implementing any learning mechanism in this network so, there will be no feedback paths. It can be organized into layers that are only connected by "forward connections". We need to optimize the entire system for maximum throughput.

Motivation:

ONFUCIUS once said, "I hear and I forget, I see and I remember, I do and I understand"**[1]**, to understand the complex functionality of brain, we need to build one. Human brain is an extremely capable part in this world. On an average, it consumes just 20W and processes 2.2 Billion megaflops per second **[2]**. "Biology really does a lot with a little"**[1]**. Even if we have fastest and advanced computers now a days, their capability really doesn't even match to a monkey's brain in recognizing the patterns and processing the visual information **[1]**. Our brains can think and respond appropriately to a given context even if it is the first ever to face that kind of situation. Brain is able to do all these miracles by enabling huge parallel processing of data. To achieve the efficiency and power of such a device, we need to build one. Are we just looking for power? No! It is estimated that Worldwide some 180 million people are blind or visually disabled—the equivalent of two-thirds of the entire U.S. population **[3]**.Across all age groups, in the United States, approximately **1,000,000** people over 5 years of age are "functionally deaf"**[4]**. Even If we have artificial ear implants, they are nowhere near the functioning of original ear. The inner ear uses just 14 microwatts and could run for 15 years**[5]**. All these organs communicate with the brain for processing the sensory information. So, If we can mimic the functionality of brain, then it not only results in the innovation of fast and efficient computers, it can also answer many unsolved mysteries related to our sensory organs leading to highly sophisticated body implants.

Neuromorphic architectures:

As Moore's law is officially dead**[6]**, the search for the unconventional computing methods is at its pinnacle now. Neuromorphic computing is a variant of non-Von Neumann architecture which essentially involves mimicking the brain's functionality. Our brain has billions of neurons that communicate with each other. Main phenomenon that sets or our brain apart from computers is "Cognition". These neurons have three main parts, they are axon, cell body and dendrites. Neurons will be connected to other neurons through dendrites. But this specific connection is called 'synapse'. Scientists have found that, as we start practicing any new process, these synaptic connections are increasing their width this process is called learning. To simulate neurons in digital systems, we call this synaptic width as "weight". If we have feedback happening in the network we can change the weight of a net or a connection, which is almost equivalent to the changing of synaptic widths inside the brain.

Project Overview:

As mentioned earlier, we need to simulate a neural network which is 4 inputs wide. The network has 4 input neurons, 4 hidden neurons, and four output neurons. It is depicted in the below figure. Every neuron in the current layer is connected to all the other 4 neurons in the next layer. This network is divided into 3 layers which are input, hidden and output layers respectively. Every connection to other neuron has some associated weight denoted by 'w' with a subscript denoting the source and target neurons respectively.

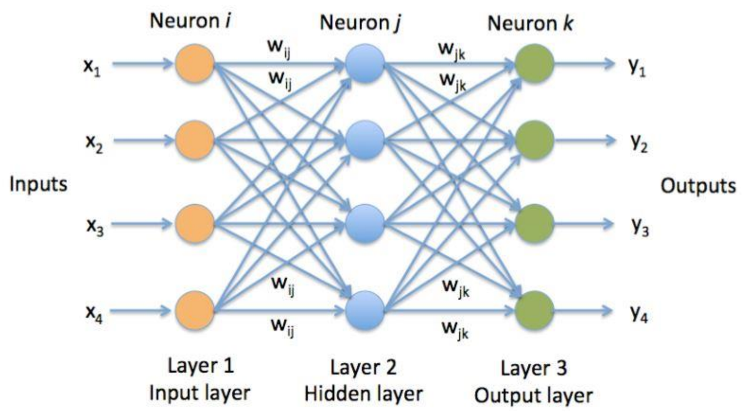


Figure (1)

So, in the first layer, neurons will take the input, denoted as X, and as every neuron is connected to all neurons of hidden layer, the input data will be transferred to them. But here every connection has a weight associated with it just like synapse width. Input value will be multiplied with the weight and it will be acquired by the hidden neuron. Hidden neuron will add all these weight-input products, and produces an output which acts as an input for the output layer. The mathematical representation for clear understanding is provided below

$$Z = \sum_{i=1}^{4} x_i \, w_{ij} \qquad \text{------------- EQ (1)}$$

$$Y = f(x) = \begin{cases} 1 \; if \;\; Z \geq 0 \\ 0 \; if \; Z < 0 \end{cases} \qquad \text{--------------EQ (2)}$$

Equation (1) denotes the operation performed at every neuron i.e. summing up the input-weight products. Equation (2) can be termed as threshold function which determines the output **Y** of every neuron.

Instruction set architecture:

As a computer architect, whenever we hear the name “processor”, first thing that comes to our minds is “Instruction set architecture”. Project specification gave us the freedom to select from the different types of instruction set architectures, so that we can optimize the design for the throughput. There are mainly four types of instruction set architectures. Their advantages and limitations are stated in the below table.

ISA	Advantages	Disadvantages
Stack	No need to have explicit operands	Random access is not possible
Register-memory	Data can be accessed without load for the first instruction. Instruction format is easy to encode	Clocks per instruction may vary depending upon the operand position
Register-register	Simple code generation, instructions take similar number of clocks to execute	More instructions tends to larger programs
Accumulator	Only one explicit operand, so shorter instructions	Only one register target for storing, leading to high memory traffic

Table (1)[7]

Upon careful examination, we decided to adopt **Load-store architecture**. Because, if we use stack architecture, we can’t access the data randomly. If we use register memory or memory-memory architectures, it leads to the structural hazards, because every instruction wants to access memory. If we use accumulator architecture, it results in heavy memory traffic which will dominate and becomes the critical path in the pipeline. **So, if we use load-store (register-register) architecture, we can have instructions accessing the memory in the memory stage, and instructions in the execution stage, they both can happen independently as the operands are already fetched. This really simplifies the pipeline operations so, we decided to use Load-store architecture.**

Micro-architecture:

Micro architecture is a way to implement the instruction set architecture on hardware. Project specification clearly stated that we need to implement a **pipelined architecture**, we decided to do so. But we were given freedom to pick as many stages as we want to optimize it for the throughput. A pipelined processor needs to perform few tasks to execute an instruction. These are listed below.

Name of the Stage	**function of the stage**
Fetch stage	Instructions are grabbed from the memory
Decode stage	During this stage, instructions are analyzed to figure out operations to be performed
Execute stage	Instructions are executed and results are computed
Memory stage	Results will be written to the memory if necessary
Write back stage	Target result will be updated with the result produced at execution unit

Table (2) [8]

As we are implementing pipelined architecture, **we decided to make these five phases, as five equally balanced stages in the pipeline.** We decided not to increase the number of stages because, fetch, decode, memory and write back stages are simple and can't be split again. The only hope is to divide the execution stage into multiple sub stages to increase the throughput. But after looking at the assembly code, we realized that the code is having simple arithmetic operations which doesn't qualify for a multi-staged execution unit. So, we decided to stick on to the simple 5-stage pipelined architecture. Increasing the number of stages in a pipeline yields a direct increase in the final throughput of a processor[8]. As we decided the keep the 5–stage pipeline architecture, we began exploring the various options by which we can increase the throughput. A detailed microarchitecture of our 5-stage pipelined processor is depicted below.

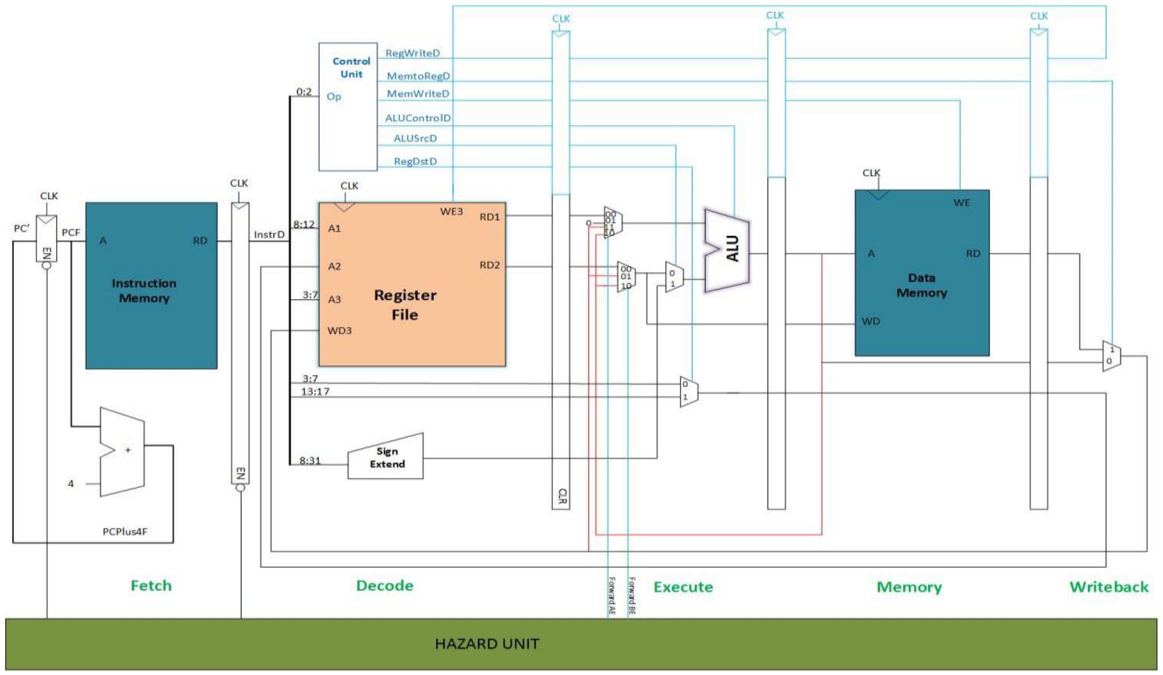


Figure (1) [8]

Load path:

First, the instruction is fetched from the instruction memory during fetch state, then it is given to the decode state during the next clock cycle. As we used fixed instruction length encoding, it is very easy to decode the instruction in the decode unit. From the instruction, opcode is given to the control unit which enables the select lines of all multiplexers. Address will be specified as an immediate value in the load instruction. Now, it is given to the ALU, through a sign extender which makes the immediate value as 32 bits. Now, as we don't support the indexed addressing mode, '0' will be given to the ALU as another operand, ALU sums them up and produces the same immediate value as output to the memory. Now data is fetched from the given address and is transferred to the write back state. Meanwhile, PC value is incremented to fetch the next instruction.

Data path:

If the NMUL or NADD instruction is given, then the control unit selects two registers from the register bank. Our custom made 32- bit ALU performs the required operation, and produces output and then it traverses through the memory stage, finally written back to the register bank during wright back stage

High level implementation:

These 5 pipeline stages were simulated using a high level language. We used Python to simulate these stages. As there is no notion of 'clock' in high level languages, we maintained a global clock variable, and it increments by 1 whenever a new instruction is fetched. To simulate concurrency in the high level language, we have used temporary registers between successive stages for transferring data

Architecture summary:

Processor type	5-stage pipelined processor
Data bus	32-bit wide
Address bus	32-bit wide
Total number of registers used	32
Length of registers	32-bit length registers
Addressing mode used	Immediate addressing mode
ISA	Load-store (register-register)

Table (3)

Hazards:

We can achieve better throughput using pipelined architecture, but it comes at a cost. As we are working on multiple instructions in the pipe stages, we need to face few problems known as hazards. Those hazards and solutions to mitigate them are listed below.

Data hazards: There are two data hazards in the implemented code. We are using "**Data forwarding**" technique to eliminate this hazard. (Forwarding paths are shown in red color). A dedicated Hazard unit will take care of these data hazards.

Hazards unit working:

The implemented hazard unit basically tries to detect RAW hazards. We will have two forwarding paths, one from memory stage and another from write back stage to the execute stage. If it detects hazards then hazard unit enables the appropriate forward path to eliminate the hazard.

Structural hazards: We are using dedicated ALU for incrementing PC value to avoid ALU hazards, we have used separate instruction and data memories to avoid hazards related to memory accessing.

Control hazards: Because of the custom made ALU and instruction set combination, we have no 'JUMPS' in the implemented code. So there will be no control hazards.

Once as Alan Kay said, **"people who are serious about software, should make their own hardware**."[10]To achieve high throughput, we need to obtain the output layer result with as few instructions as possible. We took this issue seriously so, we decided to make our own custom- hardware to get this job done efficiently.

Minimizing the memory accesses required:

As memory accesses directly effects the throughput, we first decided to minimize it to the possible extent. From the specification, we can know that memory image is 32-bits wide and is byte addressable. We decided to use **aligned memory** to reduce the number of memory accesses required to fetch operands [9]. Product specification also says that, inputs can be 0's or 1's and the weights can be '-1' '0' or '0'. So, in order to distinguish '-1' from '1' and '0' we just need to bits. We decided to encode every value with just two bits .It works as follows.

01-- Represents '1'	11-- Represents '-1'	00 -- Represents '0'

Here the MSB bit represents the sign of the value. And the '10' combination is unused. We can conveniently represent all the input and weight combinations with this encoding. For example, an input combination 1,0,0,1 can be represented as "01000001". These 8bits are sufficient to encode the four inputs, and to preserve the symmetry during a memory access, these 8 bits are replicated 4 times as 32-bits.

Example:

Input value: 1 0 0 1 **32-Bit representation:** "01000001010000010100000101000001"

Encoded value: "01000001" **Hex Value in memory image:** "41414141"

We know that there are four input neurons so, there will be 16 connections to the 4 neurons in the next layer. Each connection will have its own weight, leading to 16 weights between the input layer and hidden layer, 16 weights between hidden and output layer making a count of total 32 weights. As we need just two bits for representing a value, we can store the '16' weights values using 32-bits.So, **If we use 32-bit registers, Just one memory access** is sufficient to grab all the weights between the input and hidden layers. So, using this scheme we can get the input values in one memory access, 32 weight values in another 2 accesses. So, **with just 3 memory accesses we can obtain all the information present in the given network.**

Minimizing the Logical operations:

After reducing the memory accesses to the lowest possible extent, we decided to compute the result with least possible instructions. We wanted to keep the computation simple yet powerful. So, we first started to explore all the possible outputs, for the given inputs and tried to establish a logical relation between inputs and outputs. After many failed attempts, we were able to produce a logical relation between inputs and outputs. If we closely observe the network functionality, it is clear that we need to do the following three operations to produce the output.

1. Multiply the input and weights
2. Add the intermediate products produced at each node
3. Compare if the sum is greater than Zero or not.

We can use the conventional ALU for doing multiplication, addition and comparison. Here we just need to perform simple arithmetic operations to produce the output. For this particular job, we don't need a massive ALU. So, we decided to build an ALU to reduce the number of instructions required to produce the output. Its implementation and working is as described below.

'NMUL' unit: This unit acts as a replacement for the multiplication unit in an ALU. This is a 32-bit unit that takes the given input values, weights and performs a simplified multiplication process to produce the multiplication result. The beauty of this unit is, it requires just 64-AND gates to get the multiplication done. More interestingly, multiplication of 4 inputs with 16 weights can be done in just one clock cycle.

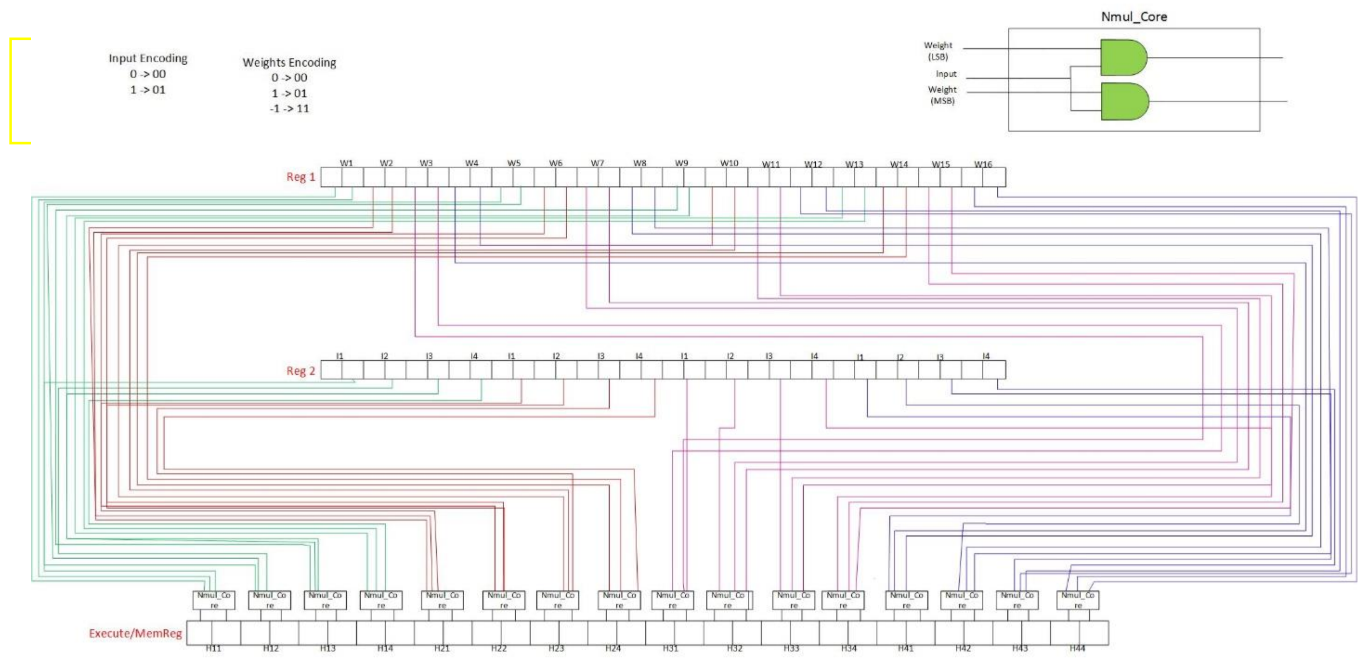

Input Encoding
0 -> 00
1 -> 01
Weights Encoding
0 -> 00
1 -> 01
-1 -> 11
Nmul_Core
Weight (LSB)
Input
Weight (MSB)
Reg 1
W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 W11 W12 W13 W14 W15 W16
Reg 2
I1 I2 I3 I4 I1 I2 I3 I4 I1 I2 I3 I4 I1 I2 I3 I4
Nmul_Core
Execute/MemReg
H11 H12 H13 H14 H21 H22 H23 H24 H31 H32 H33 H34 H41 H42 H43 H44

Fig (2): Multiplication unit

Instruction format:

0 – 2	3 – 7	8 – 12	13 – 17	18 – 31
OPCODE	Dest_reg	Source_reg_1	Source_reg_2	Unused Bits

Example:

NMUL R3,R1,R2

Here, NMUL is opcode. R3, R1, R2 are destination, source_1, and source_2 registers respectively. By using this NMUL instruction, we can perform the multiplication of 4 inputs and 16 weights with a single instruction. Let us suppose, we need to multiply '1' (input) with '-1' (weight). From our encoding format, it is clear that '1' can be represented as '01' and '-1' can be represented as '11'. Here, from the specification it is clear that the input can be 0 or 1. So, it can be encoded as 00 or 01. From this it is clear that we can identify the value of input just by looking at the LSB bit. So, now we are going to apply this LSB bit and the weight value to a module called as Nmul_core which is shown in the figure. It is a combination of two AND gates. Here the input LSB bit is given to the 'input' terminal of Nmul_core and the two bits of the weight are given to their respective pins as specified in the figure. As the input value is '1', the weight value will be reflected at the output of the Nmul_core module. If the input is zero, then irrespective of the weight value, AND gate produces "00" at the output of the module. With this simple two AND gate setup, we can perform the multiplication. We replicated this logic 16 times so that we can perform 16 Multiplications with just one instruction. Figure (2) shows its implementation at hardware level. We can observe there are three 32-bit registers, and 16 Nmul_core modules. Once we stored the required input and weight values in the registers, Nmul_core performs the operation on the given data and produces 32-bit result.

'NADD' unit: After performing multiplication, we will get four input-weight products at each hidden neuron. Hidden neuron will sum up these four products and compares it whether it is greater than '0' or not. We are proposing a new architecture for this operation. With this architecture, we can complete the addition and comparison of all hidden neurons in just one cycle. Instruction format and hardware implementation is given below.

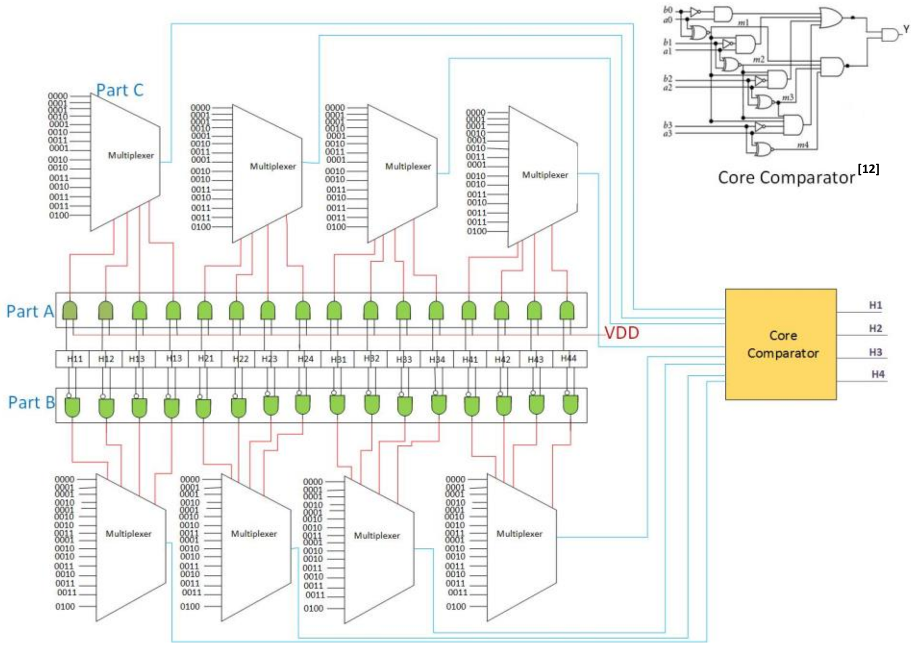


Figure (3) : NADD unit

Instruction format:

0 – 2	3 – 7	8 – 12	13 – 31
OPCODE	Dest_reg	Source_reg	Unused Bits

Example: NADD R3,R4

Here NADD is opcode, and R3, R4 are output and input registers respectively. R4 register will be having 16 input-weight products which in total makes 32bits. Now we are going to perform addition and comparison with just a single instruction using the customized architecture shown in the figure (3).

A close observation of the threshold condition reveals that we actually need to know whether the added sum is less than '0' or not. For this, we necessarily don't need to do the exact addition. We can just compare the number of '-1' s , '1' s in the given data. If the '-1's are higher in number, result will be negative. If the number of '1' is greater than '-1' then the result will be positive. If they both are equal in count, then the sum will be zero. Depending upon this sum value, threshold function outputs zero or one.

This entire operation can be achieved by the hardware shown in the figure (3). It works as follows

Part A:

It is a module containing 16 AND gates. Each hidden neuron requires 4 gates so, in total 16 gates are used. It is used to detect number of "-1" (i.e."11" combinations) in the obtained multiplication result.

Part B:

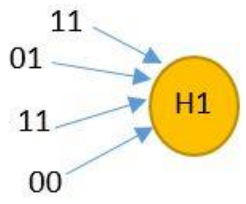
It contains 16 "not A AND B" gates. Each hidden neuron requires 4 gates so, in total 16 gates are used. This setup is used to detect number of '1's (i.e. "01" combinations) in the obtained multiplication result.

Part C:

This part is a 4-bit 16:1 multiplexer. It acts as a look up table for comparing the output of part A and part B. The result from the part A and part B is fed to these multiplexers to convert it to the binary numbers.

Working:

Let us suppose, we have 4 input-weight products coming to the ‘H1’ as shown in the figure(4).According to our encoding format, ‘11’ represents ‘-1’, ‘01’ represents ‘1’. So, we have -1,1,-1,0 at the hidden neuron. Its representation at the bit level is ‘11011100’.



Figure(4)

So, this pattern will be given to both part A and part B. ‘part A’ produces ‘1010’. This Means, we have two ‘-1’ s in the combination. As we are giving it to multiplexer select lines, It selects ‘0010’ from the given combination. The part B produces ‘0100’ which selects ‘0001’ at the multiplexer output. Now these two outputs will be given to a 4-bit comparator (core comparator) and as the number of ‘-1’s is larger than ‘1’s, it produces ‘0’ at the output. Which is exactly the result of the threshold function. Similarly, result will be computed for all the remaining neurons. From this it is clear that, we can add, compare these input-weight products in just one clock cycle.

Memory type instructions:

Load:

0 2	3 7	8 31
OPCODE 3’b001	Dest_Reg	Immediate Value

This instruction is used to fetch data from memory. This instruction uses immediate addressing mode. Dest_reg field is used to specify the destination register among 32 available registers.

Ex: LOAD R2, 00000008

Store:

0 2	3 7	8 31
OPCODE 3’b010	Source_Reg	Immediate Value

This instruction is used to store data to memory. This instruction uses immediate addressing mode. Source_reg field is used to specify the source register among 32 available registers.

Ex: STORE R1,00000004

Halt:

0 2	3 31
OPCODE 3’b111	Unused bits

It is used to stop the execution of the program irrespective of the context.

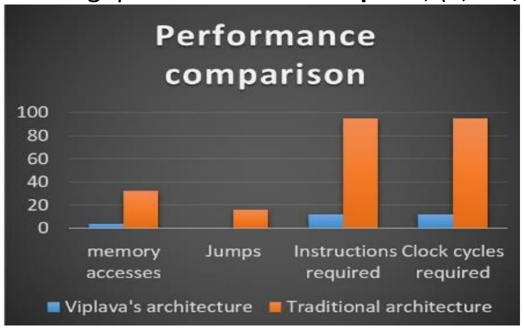
Ex: HALT

Performance Analysis:

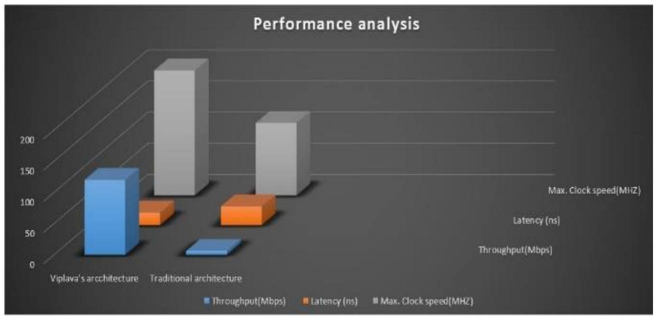
By making the hardware fully application specific, we were able to produce exceptionally good performance. By incorporating the efficient encoding scheme, we were able to load 4 inputs, 32 weights with just **‘3’** memory accesses. By designing fully customized multiplication unit, we were able to multiply 16 weights with 4 inputs with just a **single NMUL instruction**. Refined NADD unit helped us to achieve **addition and comparison with just a single instruction**. Because of these massive improvements in the architecture, we were finally able to achieve one full network iteration with just **‘13’** instructions including store. For simulating 1000 random input vectors we just need **8005** cycles. Below table clearly illustrates the performance gain achieved by **‘viplava’** over traditional architecture.

- CPI achieved= Number of instructions with NOPs/ Number of useful instructions =8005/8001 = 1.0004
- Number of instructions for single iteration (for one input vector) including NOPS = 13
- Latency for single instruction= 5* 4ns= 200ns
- Clock frequency = (1/4ns)= 200MHZ
- Number of instructions=8005
- Code size= 8005*32bits= 32,020 bytes
- Execution time = No.of instructions* CPI*clock period = 8005*1*4ns=**32 µS**

Throughput= 4000 bits in **32 µS** so, (4,000/32µS) = **125Mbps**



Graph (1): For single Iteration



Graph(2)

Performance Analysis	
Traditional architecture*	**Viplava’s architecture**
Memory accesses required:**32,000**	Memory access required:**4,000**
Code length : **50,000+ lines**(for 1000 iterations)	Code length: **8005 lines** (for 1000 iterations)
JUMP required: **YES**	JUMP required: **NO**
Registers required: **64+**	Registers required: **less than 32**
Achievable clock frequency: **119 MHz**	Achievable clock frequency: **200 MHz**
Probability of stalls : **Moderate**	Probability of stalls: **NIL**
Number of transistors[13] : **1640** (For Execution unit)	Number of transistors : **900** (For Execution unit)
Pipe line flush: **May be**	Pipeline flush: **Not required**
Clock cycles required: **70,000+** (for 1000 input vectors)	Clock cycles required: **8,005** (for 1000 input vectors)
Achievable Throughput = **6.172Mbps**	Achievable Throughput= **125Mbps**

*These are approximate values (for 1000 input vectors), assumptions are stated below

Assumptions:

We are assuming 45nm transistor technology (5.1 ps gate delay)[11] for building both conventional and Viplava’s architecture. 4ns critical path delay was obtained by multiplying the number of transistors in critical path (750 approx.) and the time taken by all those transistors. For the conventional processor [13], all those 1640 transistors are effecting the critical path yielding a path delay of 8.5ns and frequency of 119MHz, Conventional architecture is assumed to have general purpose ALU, without special encoding schemes. As there will be JUMPS without any speculation units, we are assuming flushing will happen in conventional architecture, resulting in higher CPI. Both architectures uses 32-bit instructions.

Strengths:

- Custom built hardware and instructions resulted in **20X** improvement in throughput, **10X** improvement in code size.
- Aligned memory accesses minimizes the number of memory accesses.
- Uses less number of transistors than the conventional ALUs, So low cost and lower current leakages.
- Faster clock rates than conventional architecture

Weaknesses:

- No support for indexed addressing mode.
- This architecture currently supports only for 4 Input neurons

Future work:

- We want to scale this architecture to support 1000 Input neurons.
- Adding support for different addressing modes

Conclusion:

In this work we have designed a neuromorphic processor for simulating 4-input feed forward network. Even if it is a 5-stage pipeline similar to the MIPS architecture, we struggled hard and customized the entire architecture from the core. It resulted in significant improvement in the throughput. This architecture is optimized for throughput and can only work with 4 input feed forward neuron networks. In future, we want to scale this architecture to 1000 input neuron network.

References:

[1] Mohammed Riyaz Ahmed; B. K. Sujatha, A review on methods, issues and challenges in neuromorphic engineering Communications and Signal Processing (ICCSP), 2015 International Conference on 10.1109/ICCSP.2015.7322626
[2] http://www.scientificamerican.com/article/computers-vs-brains/
[3] www.who.int/mediacentre/factsheets/fs282/en/
[4] https://research.gallaudet.edu/Demographics/deaf-US.php
[5] R. Sarpeshkar, Brain power - borrowing from biology makes for low power computing [bionic ear] ,IEEE Spectrum Year: 2006, Volume: 43, Issue: 5 DOI: 10.1109/MSPEC.2006.1628504
[6] https://www.technologyreview.com/s/601102/intel-puts-the-brakes-on-moores-law/
[7] https://classes.soe.ucsc.edu/cmpe110/Spring11/lectures/04_MIPS_ISA%20.pdf
[8] ECE 586 L9 Week5.pdf
[9]ECE 586 L7 Week4.pdf
[10] http://www.relatably.com/q/img/alan-kay-quotes/alan-kays-quotes-2.jpg
[11] Mistry, K. ; Allen, C. ; Auth, C. ; Beattie, B. ; Bergstrom, D. ; Bost, M. ; Brazier, M. ; Buehler, M. ; Cappellani, A. ; Chau, R. ; Choi, C.-H. ; Ding, G. ; Fischer, K. ; Ghani, T. ; Grover, R. ; Han, W. ; Hanken, D. ; Hattendorf, M. ; He, J. ; Hicks, J. ; Huessner, R. ; Ingerly, D. ; Jain, P. ; James, R. ; Jong, L. ; Joshi, S. ; Kenyon, C. ; Kuhn, K. ; Lee, K. ; Liu, H. ; Maiz, J. ; McIntyre, B. ; Moon, P. ; Neirynck, J. ; Pae, S. ; Parker, C. ; Parsons, D. ; Prasad, C. ; Pipes, L. ; Prince, M. ; Ranade, P. ; Reynolds, T. ; Sandford, J. ; Shifren, L. ; Sebastian, J. ; Seiple, J. ; Simon, D. ; Sivakumar, S. ; Smith, P. ; Thomas, C. ; Troeger, T. ; Vandervoorn, P. ; Williams, S. ; Zawadzki, K,
“A 45nm Logic Technology with High-k+Metal Gate Transistors, Strained Silicon, 9 Cu Interconnect Layers, 193nm Dry Patterning, and 100% Pb-free Packaging”, 2007 IEEE International Electron Devices Meeting, Dec. 2007, pp.247-250 DOI: 10.1109/IEDM.2007.4418914
[12] http://www.hindawi.com/journals/ase/2010/323429.fig.002.jpg
[13] http://www.asee.org/documents/zones/zone1/2014/Student/PDFs/232.pdf